Andreas Rüping

WHERE CODE AND CONTENT MEET

Design Patterns for Web Content Management and Delivery, Personalisation and User Participation

WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

Where Code and Content Meet

Where Code and Content Meet

Design Patterns for Web Content Management and Delivery, Personalisation and User Participation

Andreas Rüping

WILEY

A catalogue record for this book is available from the British Library.

ISBN 9780470748459

Set in 10/12 point Sabon by WordMongers Ltd, Treen, Penzance, Cornwall
Printed in Great Britain by TJ International Ltd, Padstow, Cornwall

Contents

Foreword

About 60 years ago the American scientist Vannevar Bush envisioned a new way of accessing and exchanging knowledge. In a paper that was originally drafted before World War II, he outlined a machine he called MEMEX that would extend human memory in a way that had never been seen before. Equipped with a microfilm reader, the knowledge worker of the future would be able to access all the content that had ever been written and create links between it. These ideas later influenced key computer scientists such as Ted Nelson and Douglas Engelbart, and eventually led to the web as we know it today, which builds on this vision – although it is still not able to reach the level of interaction that was foreseen by Bush. Imagine a world in which we were able to create reading trails between any items of knowledge.

When I think back to my first encounters with the hypertext systems that were developed in research contexts in the late 1980s, I can still remember special-purpose hypermedia readers providing things like assistance for museum visitors. It was really exciting to see how soon after that the World Wide Web became reality. I'm not sure what I would have said if someone had told me that I would find web pages for almost everything only a few years later. I still remember how funny it was to find the first web page for cat food, or towels. I thought that this was really useless – however, even the web pages for cat food were exciting, somehow: they offered an engaging user experience. Today, Google lists more than 75 million pages that mention cat food – more than one for every UK citizen. Such pages do not exist in isolation: they are connected to other pages to form a huge collection of websites maintained by people like you and me.

With the advent of Web 2.0 we see an increasing number of interactive sites where visitors become the authors of the content and link the content together. Wikipedia is one of the largest examples, where a community of authors creates and manages a huge collection of content. As you can imagine, you will find a page explaining cat food on Wikipedia as well, at some considerable length. Links on such pages help you to research any subject of your choice: an initial page can become an entry point for exploring biology, chemistry or any other subject. I confess that I did not imagine this effect in the early 1990s. Maybe you were more visionary in those days?

This book is not about cat food or towel sites, although you will find valuable hints for building a special-interest site if that is what you need. Nor is it about museums and the history of technology, although it illustrates web content management by designing a website for a museum. Instead, the book will teach you how to plan, build and maintain new or existing websites by providing essential guidelines for their architecture and design without delving into the details of their implementation. It will help you to decide which content management system to use, and give you the background knowledge required to really understand the difficulties of content management, explaining good practices that help you to avoid such difficulties in your project. If you have some experience of content management already you might consider some of the advice obvious at first glance, but even in these cases I became more and more excited when following the explanation of the details.

Each area of advice is presented as a design pattern, a format that is easy to memorise and understand. One of the benefits of this style is that you can browse quickly through the whole book by looking at the text set in bold font. I suggest that you try this first, or simply look at the pattern thumbnails section at the end of the book. Just as web pages are connected in the web to form a website, the patterns in this book are connected to form a pattern language. Just as the web as a whole becomes more powerful because of the connections, the pattern language will leverage your understanding of web content management to give a holistic perspective – the pattern language gives you the vocabulary to talk about websites. Try it out and investigate the presence of the patterns in one of your preferred websites. I suspect that you will find at least some, and that you will be able to suggest improvements by looking at those patterns that weren't used for the site.

This is where we come back to the original vision of Vannevar Bush. Using the right intellectual tools, you will be empowered to analyse, manage, critique and structure the knowledge that you want to present on the web. You will be able to improve the reading paths and you will become aware of means for personalisation and participation for your readers. All this will help you to create websites that are easier to maintain. Finally, visitors to the site will thank you for content that is up-to-date, easy to find and – above all – exciting.

Till Schümmer

Cooperative Systems Group
FernUniversität in Hagen

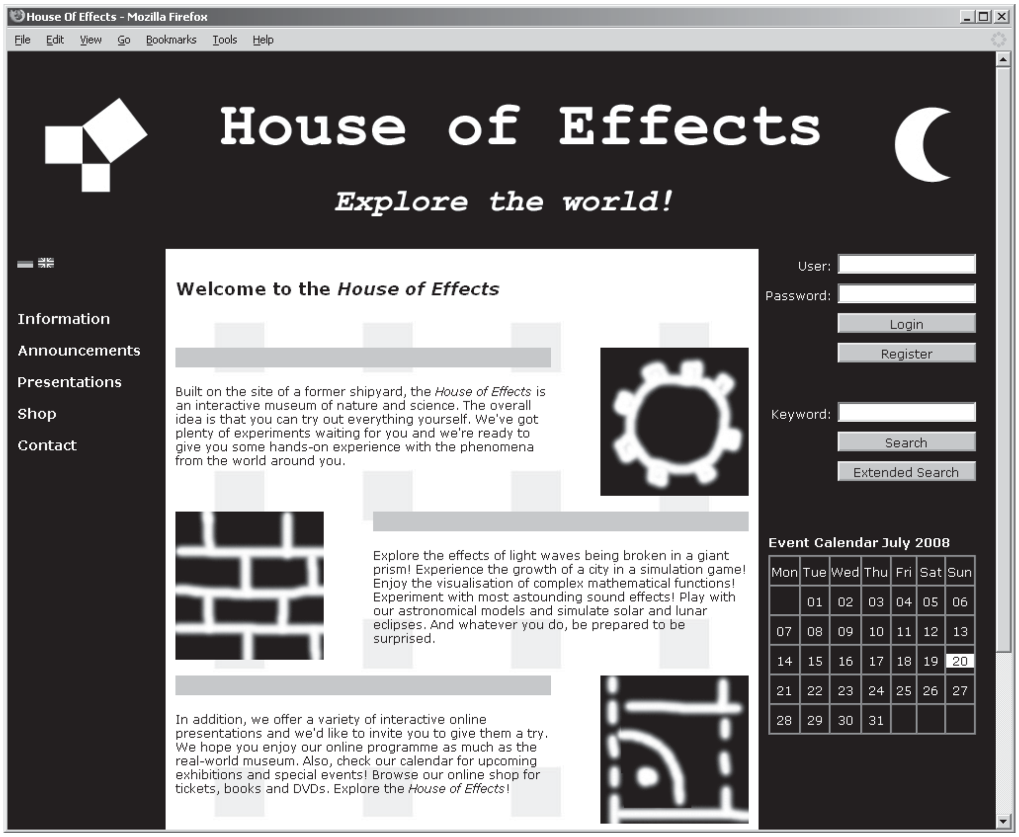
Preface

Imagine that you are involved in a project that plans to develop a website for the House of Effects – a museum of nature and science dedicated to explaining natural phenomena to an interested audience. The website should inform visitors about what the museum has in store, offer interactive online presentations and announce forthcoming special events. In addition, there is going to be an online shop, where registered users will be able to buy tickets as well as books, DVDs and the like. Finally, the site is supposed to express a community feel. It will allow registered users to comment on and rate online materials and shop items. Users will also be able to receive personalised recommendations that match their interest profiles.

The screenshot in Figure 1 shows the start page for the House of Effects. It gives you a rough idea of how this website will look.

However, there are more requirements. Content editors must be able to add presentations and announcements smoothly, so straightforward workflow processes are required. Some online presentations might invite user-generated content, so the website must also be able to receive submissions from users. The site must be reliable, as it will be the main channel by which the House of Effects will be advertised to the general public. Response times should be reasonably short. The site has to scale to a potentially large number of users. Last but not least, the website's owners want to retain the option to change or extend the site later, so the software behind it should be easily maintainable.

All this is a fairly typical scenario: today pretty much every organisation has a website. Company profile, service portfolio and contact information are among the things that all organisations need to make available via the Internet. But many sites consist of much more than this. Interaction and services for the users are common enough. In addition, some sites include an online shop where digital products are sold over the Internet. Personalised sites tailor their content to specific users or user groups. With the growing popularity of Web 2.0, some sites have turned into collaborative platforms where users can actively participate in a community.


House Of Effects - Mozilla Firefox
File Edit View Go Bookmarks Tools Help

House of Effects

Explore the world!

Information
Announcements
Presentations
Shop
Contact

Welcome to the *House of Effects*

Built on the site of a former shipyard, the *House of Effects* is an interactive museum of nature and science. The overall idea is that you can try out everything yourself. We've got plenty of experiments waiting for you and we're ready to give you some hands-on experience with the phenomena from the world around you.

Explore the effects of light waves being broken in a giant prism! Experience the growth of a city in a simulation game! Enjoy the visualisation of complex mathematical functions! Experiment with most astounding sound effects! Play with our astronomical models and simulate solar and lunar eclipses. And whatever you do, be prepared to be surprised.

In addition, we offer a variety of interactive online presentations and we'd like to invite you to give them a try. We hope you enjoy our online programme as much as the real-world museum. Also, check our calendar for upcoming exhibitions and special events! Browse our online shop for tickets, books and DVDs. Explore the *House of Effects*!

User:
Password:
Login
Register

Keyword:
Search
Extended Search

Event Calendar July 2008

Mon	Tue	Wed	Thu	Fri	Sat	Sun
	01	02	03	04	05	06
07	08	09	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Figure 1: Home page for the House of Effects

Almost all advanced websites or web platforms require a good deal of custom software. Of course, you can (and should) use tools such as a content management system, a web server, web application frameworks, tag libraries, a search engine and perhaps others, and these tools provide much of the necessary standard functionality. But standard functionality isn't all you need – it is likely that you will have to meet more specific requirements. Possible examples include a specifically designed content model, personalisation strategies, individual page layouts and support for different output media. To meet such requirements, custom software development becomes necessary.

An essential part of that custom software needs to handle the web content required by a site. Web content can cover all sorts of digital assets, including text, pictures, videos or other multimedia objects. To this end, there are two worlds that you need to balance. One is the world of content, with its underlying models, classifications and lifecycles. The other is the world of code, where the term 'code' is meant to summarise those software techniques and mechanisms necessary to represent content on the web and to implement user interaction.

If you are interested in either of these two worlds individually, plentiful information is available. There are several books on web content management and enterprise content management, usually offering an information management perspective. Similarly, there are many good books on software development in general and on web applications specifically.

This book sets out to do something different. Starting at the point where code and content meet, it aims to explain how these two worlds go together, offering an approach that aligns the requirements from either perspective. Code and content are viewed as interlocking jigsaw pieces, as Figure 2 shows. Neither content nor code are going to be seen in isolation.

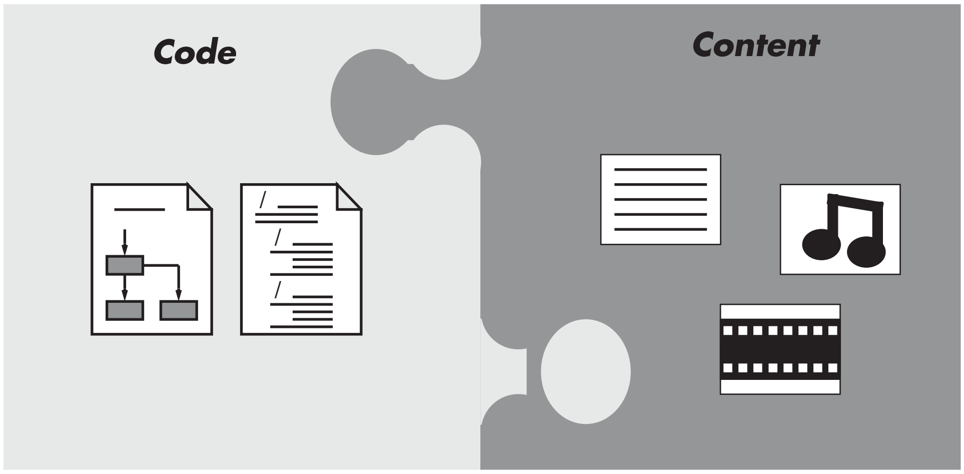


Figure 2: Content and code complementing each other

What this book is about

This book assumes a software designer's perspective and presents a collection of patterns that address the content-related aspects of custom software development for advanced websites or web platforms.

So what are patterns? Patterns represent good practice and suggest solutions that have worked well in many practical cases. Mined from a series of successful web projects, the patterns in this book represent the collected expertise of designers from several software development teams.

Key topics in the book include content modelling and content organisation, navigation, findability, personalisation and user participation. The patterns are independent of specific tools and technologies. They focus on non-functional requirements, with the overall goal of defining a sustainable software architecture. In addition, checklists for managing a web project and for the selection of a web content management system give practical and straightforward advice.

I am not going to present one specific architecture, though. The reason for this is simple enough: websites are much too heterogeneous to be designed in a uniform way. Requirements differ a lot, as do the underlying technologies. There is no point in trying to define a single all-purpose architecture. Instead, the patterns in this book should serve as a practical guide to designing your own content-related custom components for your web project. Since the patterns are independent of any specific tool or programming language, you can use them to define an architecture that matches your specific needs.

This isn't to say that you have to develop all software components yourself. Actually, you should not do so. First, you'll probably be using a content management system. There are many systems available on the market, so it is unlikely you will have to develop your own. Second, you will probably be using frameworks for web development, such as Struts, Java Server Faces, Ruby on Rails or something else. These tools can save you a lot of work, but you will have to integrate them, and there is no question that a non-trivial site with individual requirements will also require a good deal of custom software. This is where this book can help you.

The book is also meant to help you to define a *sustainable* architecture – one that has a lasting positive effect on your website's system characteristics. The patterns not only seek to align code and content, but also focus on non-functional requirements. For example, the site should be fast, despite the fact that good response times can be hard to achieve, especially in the presence of personalisation. Next, maintainability is essential. Only a solid architecture can ensure that later changes to your site won't require a large and unjustified effort. There are also other important non-functional requirements, including security, reliability and scalability. Since the patterns in this book keep an eye on these things, they can help you to design a website with an underlying high-quality software architecture.

What this book is not about

There are of course things that I cannot deal with. To further clarify what the book is about, let me contrast it against related topics that are outside its scope:

- The book is not about web development in general. Its focus is on the interplay of code and content, which is more specific, although there may be overlaps. If you are interested in web development in general, I would refer you to the literature that is available, including books by Martin Fowler on *Patterns of Enterprise Application Architecture* (Fowler 2003) and Paul Dyson and Andy Longshaw on *Architecting Enterprise Solutions* (Dyson Longshaw 2004).
- I am not going to talk about specific programming languages or tools. Java, PHP, Ruby, JavaScript, XML, HTML and CSS are just a few examples of languages that

play a role in web development, but this book isn't an introduction into any of them, nor is it an introduction to any specific content management system. Instead, it describes techniques and strategies that work well with different languages and tools.

- The book is not about developing a content management system. The book tells you how to define an architecture that *involves* a content management system, but how to *develop* such a system is outside its scope. However, when you select a content management system for your project, you can use this book to evaluate different systems and see how well they support the architectural principles for advanced websites and web platforms.
- Groupware applications, even if they are web-based, are outside the scope of this book. Forums, chatrooms and the like are becoming more and more popular, but although they often appear next to web content, content management and content delivery on one hand and groupware on the other are different things. True enough, there is a connection if a website welcomes user participation and invites user-generated content. I will deal with user participation in this book, but the focus will be on the content aspects, not on the interaction models for user collaboration. If you are interested in collaboration techniques and strategies, refer to the book by Till Schümmer and Stephan Lukosch on *Patterns for Computer-Mediated Interaction* (Schümmer Lukosch 2007).
- This book is not about web design and web usability. I am not going to tell you what your web pages should look like, and not going to talk about what kinds of interaction are good and what are not. If you are interested in these topics, refer to books on web page ergonomics (for example Hackos Redish 1998, Krug 2006) and on interface design (for example Tidwell 2005, Scott Neil 2009).
- A current trend in many companies, *enterprise content management* refers to strategies used to evaluate, maintain and access content in an organisation (Rockley 2002). It is related to information architecture, which can be defined as 'the combination of organization, labelling and navigation schemes within an information system' (Rosenfeld Morville 2006). These topics may enter into what I discuss in this book, but they are not in its focus.

Why this book matters

You should now have a pretty good idea of what the book is about. As a next step I'd like to tell you why this book matters – why it contains material that is important and why it can be crucial to your project's success. Here are some reasons why you should benefit from reading this book:

- Many existing websites started several years ago. If you look behind the scenes you will notice that quite often the architecture hasn't evolved to the degree to which the functionality has grown. Over the years rich interactions, backend integration or personalisation may have been added, but often in an ad-hoc way. Maintainability and scalability problems are the logical consequence. By addressing these issues, the patterns in this book can improve your site's longevity.

- The last few years have seen several new trends and technologies emerge. Web 2.0 has brought us personalisation and user involvement. Ajax (Asynchronous JavaScript and XML) as an underlying technology has made browser-based rich-client techniques popular, as well as asynchronous communication between browser and server. It's safe to assume that these techniques are here to stay. But these more advanced techniques have also lead to increased complexity. The patterns in this book tell you how to use these techniques with a reasonable degree of caution, so that unnecessary complexity can be avoided and maintainability improved.
- These days websites are more crucial to an organisation's business model than they were a few years ago. The information and services offered on a website are often part of an organisation's fundamental business processes. Think of the plethora of corporate websites, e-government sites or online shops. As Chris Anderson has outlined in his famous article *The Long Tail* (Anderson 2006), niche markets have flourished a lot over the last couple of years as a consequence of using the Internet as a distribution channel. Of course, non-niche markets can benefit from Internet-based distribution too. A reliable site based on a solid architecture can therefore represent solid business value.
- Different groups of people are involved in the development and operation of a website. There are users, content editors, software developers, operators and sometimes others. These different groups take different perspectives and have different priorities. This book acknowledges this fact and thus helps you to reconcile the requirements brought in by the different stakeholders in a web project.

Who should read this book

This book is aimed at the IT people involved in a web project. If you're involved as an architect, a software designer, a developer or an IT manager, then this book should be useful for you. You should be able to benefit from it in different ways:

- You can use the book if the plan is to develop a new website or web platform and you want to ensure a solid architecture as a starting point. The patterns here will help you to create a 'big picture' for your architecture and will also guide you through more detailed design decisions.
- The book should be equally useful if you are working towards the relaunch of an existing site. Adding lots of new features can be a challenging task. The patterns in the book can help you to evolve the architecture of your site and to integrate new functionality smoothly.

- It is sometimes a good idea to improve the internal structure of a software system without changing its external behaviour. This process is known as *refactoring* (Fowler 1999) and is a good strategy if software longevity is a goal. If you plan to refactor your website, this book can help you to head in the right direction and take the right steps.
- Finally, you can use this book when you have to select a tool or a technology. For example, you might have to decide between the Java world, the PHP world, the world of Ruby on Rails or others. Similarly you may have to decide on a specific content management system. There are plenty of systems available, and picking the right one can be difficult. Which technology and which tools best fit your platform depends on how well they can be used to implement your architecture. You can use the patterns in this book to evaluate the options you have.

Whatever your interest in web projects, and whatever role you may take in such a project, I wish you a pleasant journey through the patterns presented here.

Acknowledgements

Writing a book always involves more people than just the author. I'd like to thank everyone who has helped me with this book project over the last couple of years.

Project thanks

First I'd like to thank those people with whom I worked on the projects where I mined the patterns. The customers for the projects should remain anonymous, but if you recognise any of the projects and you were involved, I'd like to thank you for good ideas, insightful discussions and fruitful collaboration.

Over the years I've presented many of the ideas expressed in this book at various conferences. I've received lots of encouragement and valuable feedback from many people:

- I presented some early ideas in a workshop on web content management that I ran at the OT 2004 conference in Cambridge. Thanks to everybody who participated.
- I presented sections of the book at various pattern conferences – EuroPLoP 2005, 2006 and 2007, and VikingPLoP 2006. Prior to these conferences I received much helpful feedback from the people who acted as shepherds for the conference papers: Uwe Zdun, Michael Weiss and Jorge Luis Ortega Arjona, whom I'd like to thank for encouragement, helpful comments and thought-provoking questions. At the conferences the papers were taken to writers' workshops, where they received in-depth reviews. Thanks to all workshop participants for many helpful comments.
- Bird-of-a-feather sessions at EuroPLoP 2005 and 2006 spawned more interesting ideas on the areas addressed here. Thanks go out to Joe Bergin and Michael Weiss, Allan Kelly, Rui Lopes and Patrick Morisson.
- A focus group at EuroPLoP 2007 shed much light on the pros and cons of Ajax technology. I'd like to thank all participants for their valuable contributions.

I've also received a lot of help and support during the publication process, and I'm equally grateful for that:

- First of all, thanks go out to Birgit Gruber of John Wiley and Sons for her work as editor. She offered lots of enthusiasm and encouragement following our first meeting at EuroPLoP 2008 and provided a lot of help in making this book come to life.
- Thanks also to Rosie Kemp and Sally Tickner, then of John Wiley and Sons, for their support during the early stages of this effort.
- I'd also like to thank Chris Webb, Colleen Goldring, Ellie Scott, Claire Spinks and Louise Breinholt of John Wiley and Sons for their support during the production process.
- Steve Rickaby of WordMongers was the copy-editor for this book and, as with *Agile Documentation* six years ago, it's again been a smooth ride through the copy-editing process. Thanks for taking care of layout and language, and for ultimately turning the manuscript into a book.
- Several people provided feedback on the book proposal and made helpful suggestions for improvement. Thanks are due to Peter Sommerlad, Stephan Lukosch and Till Schümmer.
- Special thanks go to Till Schümmer who, in addition to his earlier feedback, provided an in-depth review of the full manuscript. He offered many comments at an amazing level of detail and helped me fine-tune the manuscript, making it clearer, more complete and more comprehensible. His support had a lasting and very positive effect on the book.

Publishers and IT folks, it's been a pleasure to work with you. Thanks a lot!

Family thanks

It's not just publishers and IT folks whom I'd like to thank. Family support has always been important, whatever form it may have taken. Wholehearted thanks go out to all of you: Hiltrud, Jutta, Sven-Folker, Magnus, Nils Johann and Mareike.

It was my father who first got me interested in computing. This book is dedicated to his memory.

Introduction

The main contribution of this book is a collection of patterns. Patterns describe what has worked well in many practical cases. They are meant to capture mature knowledge, not to express novel ideas. This is why patterns aren't invented, but instead are observed from practical experience.

Project background

The patterns in this book have emerged from years of practice. The following table gives an overview of the projects from which I have mined them.

INDUSTRY	PROJECT DESCRIPTION
Insurance	An insurance portal. Employees from different branches of an associated bank can use this portal for selling insurance products to their customers. The portal offers detailed product information and is an entry point for ordering transactions.
Government	The website for a German federal body. The content includes political brochures, press releases, member information and announcements of public events.
Government	The Digital Town Hall. An architectural framework for a municipal service portal, the Digital Town Hall offers a wide range of communal information and public administration services.
Government	A portal for e-government knowledge exchange. Targeted at various e-government initiatives all over the country, this portal allows registered users and workgroups to plan joint efforts and to exchange documents. The portal also offers a community platform for discussions about e-government.
Retail	A trade portal for electronic parts. This highly personalised site sells electronic parts to industry customers and offers user-specific navigation, a product catalogue and a newsletter. The personalisation is based on user-specific interest profiles.

INDUSTRY	PROJECT DESCRIPTION
Retail	An online shop with a personalised bonus system. Users can buy items from any of the associated retail companies and receive bonus points for their purchases. They can redeem bonuses at the shop. A recommendation engine suggests products that match the user's profile.
Entertainment	An online shop for selling digital assets, especially music files. Much in the vein of Web 2.0, the site includes an integrated community platform. Apart from buying music, users can rate it, offer comments, suggest items to other users and publish their favourite playlists.
Retail	A catalogue for household devices sold by a retail company. The catalogue includes product descriptions, images and technical data. It is maintained by content editors and used for marketing purposes.

These projects were carried out for different customers and applied a wide range of technologies. I was involved in different roles, including architect, designer, developer and reviewer, so I had the chance to look at what worked well and what did not from very different angles.

Although these projects were quite different in some respects, including size and technology, I found they had quite a few things in common. I repeatedly ran into similar situations, facing similar questions or design problems. All projects required a good deal of custom software development, despite the content management systems and other tools that were of course used. All projects had to align information architecture (the content perspective) with software development (the code perspective). All projects were faced with similar non-functional requirements, most importantly efficiency and maintainability.

This is what motivated me to collect the lessons learned from these projects. I've always liked the idea of using patterns for capturing knowledge, so I soon decided to shape these best practises into pattern form. And this is where you have now arrived – at a collection of patterns for architecting advanced websites.

Organisation of the book

Before starting with the actual patterns, let me briefly explain how the book is organised:

- Chapter 1 starts with the big picture. It introduces the major components that are usually part of the software architecture behind a website. On one hand, this includes standard components like a content management system, a web server, an application server and a search engine. On the other hand, it includes the custom components that you must develop yourself. The patterns in this chapter explain how these components interact, what architectural options exist and what major design decisions you have to make.
- Chapter 2 discusses the organisation and management of content. This begins with various aspects of content modelling and goes on to include things like navigation,

classification, findability and validation. The content model that is developed in this chapter is rather fundamental. It forms the technical basis for how content is maintained by content editors, for how content is delivered to the web and for how content is perceived by the users.

- Chapter 3 deals with content delivery. This includes content retrieval from a repository, its rendering for presentation on the web, possible user interaction and the inclusion of client-side functionality. It addresses the design decisions that you have to make when you develop your server-side custom components. Because non-functional requirements such as maintainability, efficiency and scalability play an important role here, the focus of this chapter is the conceptual separation of structured content from its layout and presentation.
- Chapter 4 specifically introduces personalisation and user involvement. Personalisation refers to the idea of tailoring the content of your site to specific users or user groups. User involvement is more than this. Made popular by Web 2.0, user involvement refers to things like user-generated content, folksonomies and content rating. Implementing such features requires a few additions to the content model, as well as additions to the web delivery functionality, which is what this chapter is about.
- Chapter 5 presents patterns on infrastructure and deployment. These patterns address the environments and processes for website development, testing and operation on one hand and the environments and processes for content creation and maintenance on the other.

Figure 3 gives an overview of the patterns in the book. Arbitrary relationships between patterns are represented by lines. The patterns of each chapter form a cluster, as they address related topics. There are also relationships between patterns from different chapters, as you can see from the grey lines that are drawn across clusters. Specifically, the patterns in Chapter 1 define an overall architecture which gradually unfolds as we introduce more patterns in the chapters that follow. At the beginning of each chapter I'll zoom into this big picture and present a diagram that gives an overview of the patterns of that chapter, including an explanation of how they are related.

Following the main chapters, I'll conclude with two brief checklists that may be helpful for you from a project management point of view. The first checklist identifies the typical tasks and work packages for a web project. The second summarises important criteria for the evaluation and the selection of a content management system.

The preface introduced the House of Effects – the museum of nature and science that is in need of a new website with information, online presentations, event announcements and an online shop. Throughout this book I'm going to use the House of Effects as a running example. Although the House of Effects is a fictitious example, it is in fact a blend of requirements and concepts from the real-world projects that I've mentioned earlier. The House of Effects is actually quite realistic. You'll learn more about it as we go.

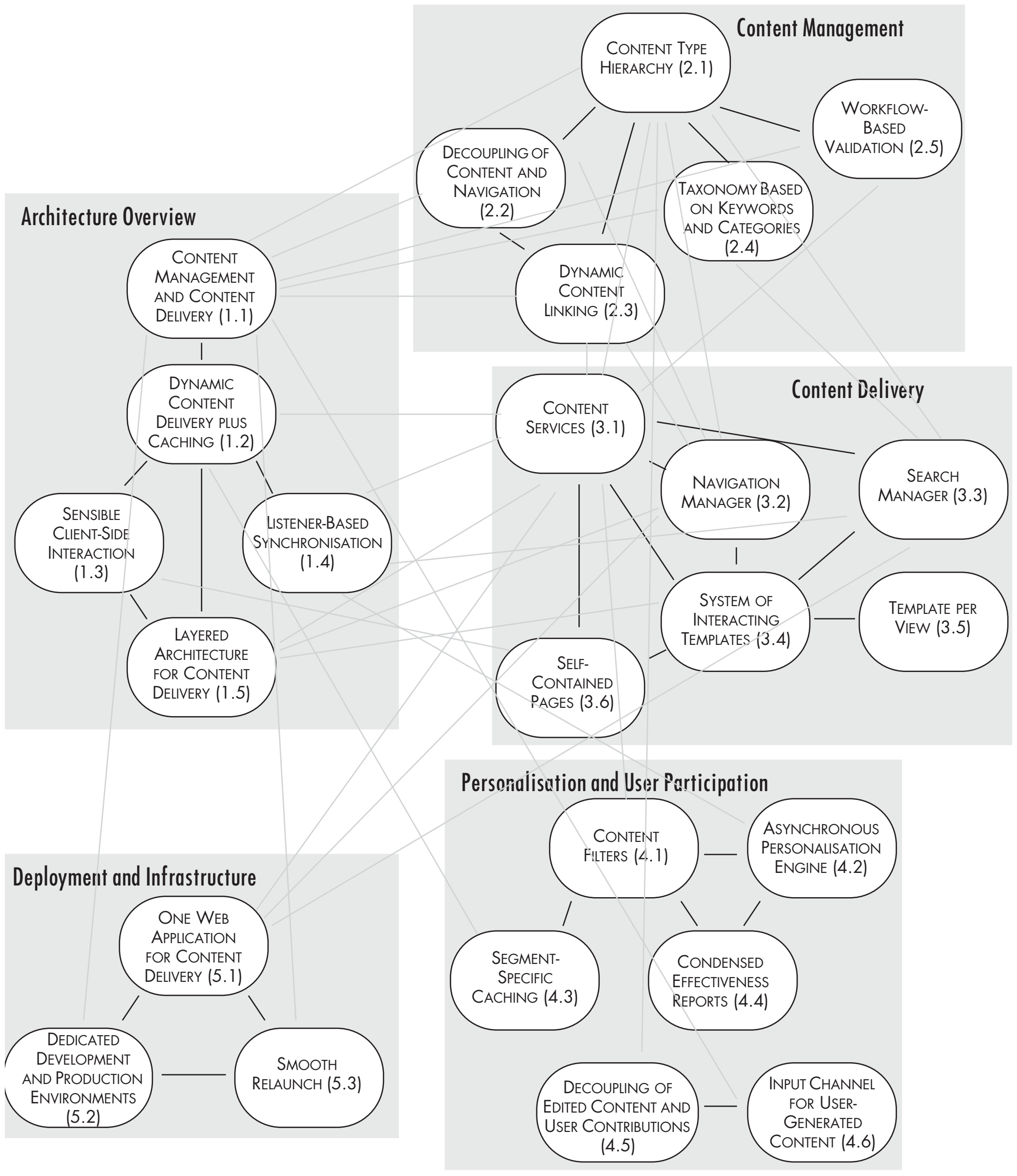

Content Management
Content Type Hierarchy (2.1)
Workflow-Based Validation (2.5)
Decoupling of Content and Navigation (2.2)
Taxonomy Based on Keywords and Categories (2.4)
Dynamic Content Linking (2.3)
Architecture Overview
Content Management and Content Delivery (1.1)
Dynamic Content Delivery plus Caching (1.2)
Sensible Client-Side Interaction (1.3)
Listener-Based Synchronisation (1.4)
Layered Architecture for Content Delivery (1.5)
Content Delivery
Content Services (3.1)
Navigation Manager (3.2)
Search Manager (3.3)
System of Interacting Templates (3.4)
Template per View (3.5)
Self-Contained Pages (3.6)
Personalisation and User Participation
Content Filters (4.1)
Asynchronous Personalisation Engine (4.2)
Segment-Specific Caching (4.3)
Condensed Effectiveness Reports (4.4)
Decoupling of Edited Content and User Contributions (4.5)
Input Channel for User-Generated Content (4.6)
Deployment and Infrastructure
One Web Application for Content Delivery (5.1)
Dedicated Development and Production Environments (5.2)
Smooth Relaunch (5.3)

Figure 3: Overview of the patterns

CHAPTER

1 Architecture Overview

If you refer to the preface you will see that there are many things on our agenda for this book – we are going to address a wide range of topics. We'll look at how web content is managed, how interactive platforms are built and how personalisation can be brought in. We'll study modelling and design as well as infrastructure and deployment.

But before we go into the details, let's study the big picture first. I'd like to develop the patterns by piecemeal growth, beginning with the more fundamental principles and then letting the big picture unfold to reveal more detailed aspects. This first chapter should therefore get us started with an introduction into the overall architecture of a website or web platform.

On the next few pages, I'd like to address the following basic questions:

- What are the main components involved in a website architecture? What are their responsibilities and how do they relate?
- What are the dynamics underlying a website architecture? What do the processes for content management and content delivery look like?
- Which of the components are typically standard and which are typically custom components? How can you bring in an individual design?

- What non-functional requirements are essential? What challenges are caused by possibly conflicting requirements?

The big picture that I'll give is independent of any technologies or any tools (as is the whole book). It also abstracts over hardware equipment, load-balancing and the like. It presents the logical model for an advanced website.
Of course a website is not that different from any other Internet-based system, or from distributed systems in general. It is no surprise that we can adopt many of the solutions that people have successfully applied in a broader context. Patterns from software architecture in general (Buschmann Meunier Rohnert Sommerlad Stal 1996) are a valuable source of information. Many of the patterns in Martin Fowler's book on *Enterprise Application Architecture* (Fowler 2003) apply. The same is true for Paul Dyson and Andy Longshaw's patterns on *Architecting Enterprise Solutions* (Dyson Longshaw 2004).

Our context is more specific, though. Our emphasis is on the amalgam of content and code, and therefore we are specifically interested in patterns that address the synthesis of content management and web application development. The overview diagram in Figure 4 gives you an initial impression of what this chapter has in store for you.

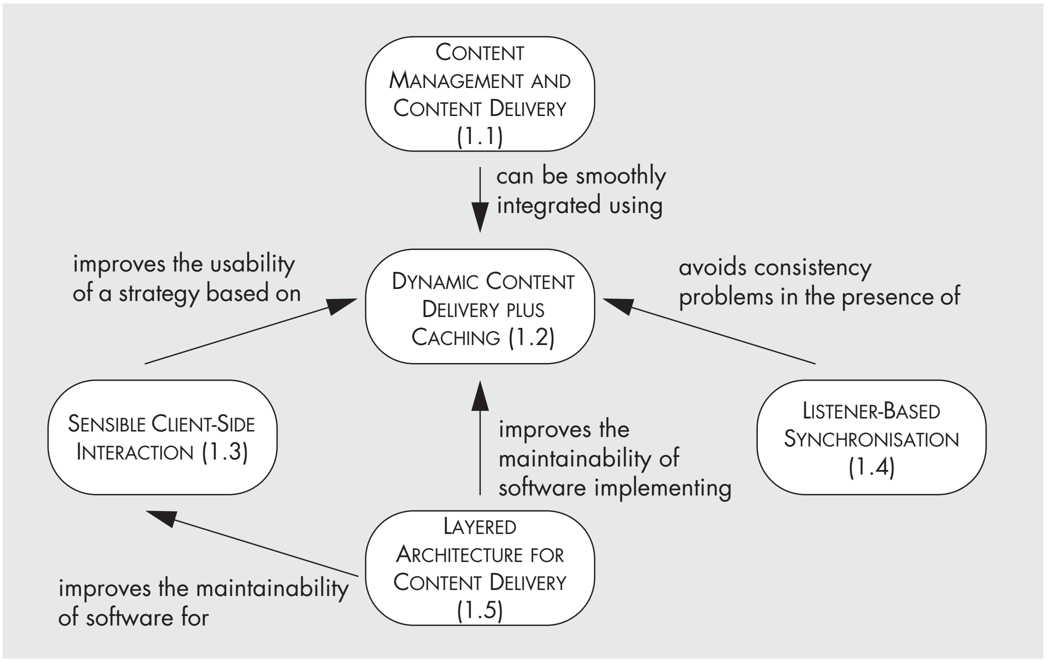

CONTENT MANAGEMENT AND CONTENT DELIVERY (1.1)
can be smoothly integrated using
DYNAMIC CONTENT DELIVERY PLUS CACHING (1.2)
improves the usability of a strategy based on
avoids consistency problems in the presence of
SENSIBLE CLIENT-SIDE INTERACTION (1.3)
improves the maintainability of software implementing
LISTENER-BASED SYNCHRONISATION (1.4)
LAYERED ARCHITECTURE FOR CONTENT DELIVERY (1.5)
improves the maintainability of software for

Figure 4: Road map to the patterns for the architecture overview

By the end of the chapter we should have achieved a common understanding of some fundamental architectural principles and of the underlying terminology. In a field as diverse as Internet-based software systems, this common understanding will be crucial for the follow-up chapters that will look at more specialised aspects.

1.1 Content Management and Content Delivery

Context

You plan to build a website using today's technology. The idea is to use a content management system that allows a team of editors to create, update and maintain the content that should be made available to the site's visitors. Visitors should be able to navigate the site, search for content and interact with the system.

Problem

How can you accommodate both the users' and the content editors' needs?

Example

The House of Effects is a museum of nature and science. The plan is to launch a new website for this museum, featuring online presentations, announcements, an event calendar and an online shop. Figure 5 shows how a typical page is going to look – in this case an exhibition announcement.

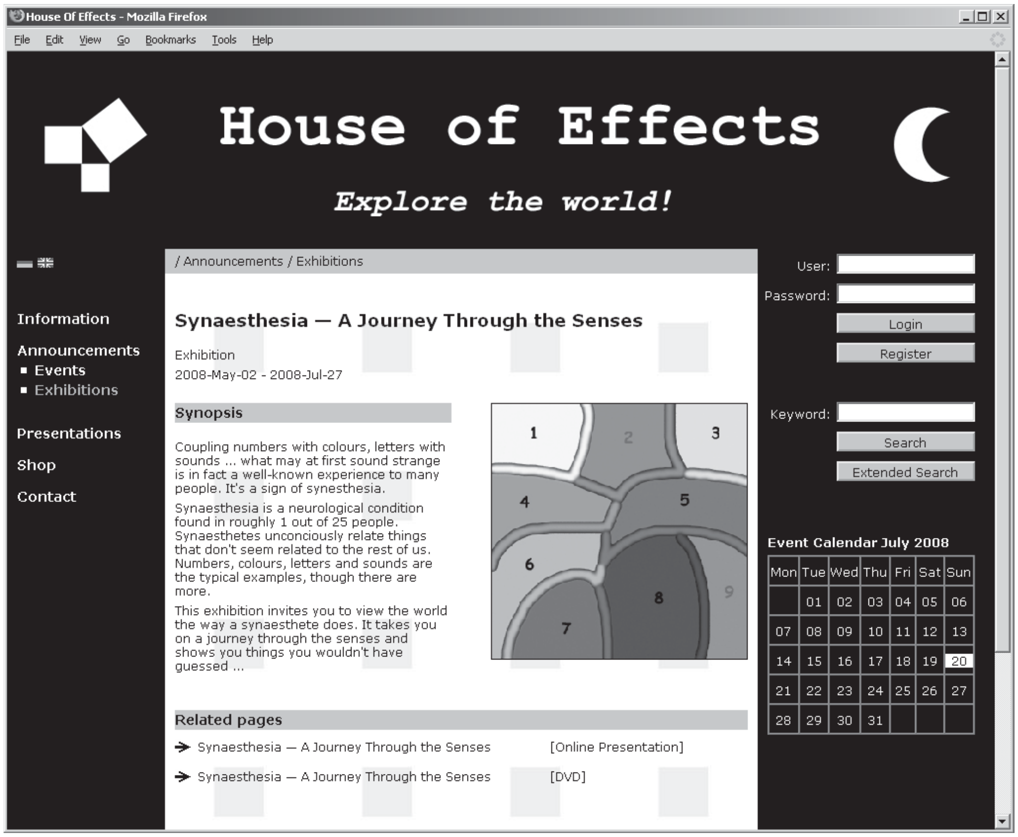
There will be a group of content editors who will be in charge of putting the necessary information together. They will maintain presentations, announcements, calendars, product information and so on. It will be their job to make sure that content is accurate, that proper proofreading takes place and that content is updated at reasonable intervals.

Users will be able to view the content once it is delivered to the web. Visiting the site with their web browsers, users will be able to view online presentations, read announcements or make purchases from the online shop. Navigation mechanisms and search functions will help them find their way through the pages. To some degree, users will also be able to make contributions to the site, for example by sharing comments or by rating a shop item.

Forces

There are two distinct groups of people who take distinct perspectives on a website or web platform: users and content editors.

Users primarily think of content as the information that is presented to them on a set of web pages. They think of text, pictures and multimedia objects. They're aware of navigation mechanisms and search functions that are available to them. They may see blogs and newsletters to which they can subscribe. In the days of Web 2.0 they may be able to



/ Announcements / Exhibitions

Synaesthesia — A Journey Through the Senses

Exhibition
2008-May-02 - 2008-Jul-27

Synopsis

Coupling numbers with colours, letters with sounds ... what may at first sound strange is in fact a well-known experience to many people. It's a sign of synesthesia.

Synaesthesia is a neurological condition found in roughly 1 out of 25 people. Synaesthetes unconciously relate things that don't seem related to the rest of us. Numbers, colours, letters and sounds are the typical examples, though there are more.

This exhibition invites you to view the world the way a synaesthete does. It takes you on a journey through the senses and shows you things you wouldn't have guessed ...

Related pages

➔ Synaesthesia — A Journey Through the Senses [Online Presentation]

➔ Synaesthesia — A Journey Through the Senses [DVD]

Event Calendar July 2008

Mon	Tue	Wed	Thu	Fri	Sat	Sun
	01	02	03	04	05	06
07	08	09	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Figure 5: Exhibition announcement for the House of Effects

contribute user-generated content. We're all familiar with this perspective – it's the perspective we all take when we visit a site.

Content editors, however, see things differently: they look behind the scenes. They are concerned with the information model (or *content model*) that underlies a website or web platform.

In her book on *Content Management for Dynamic Web Delivery*, JoAnn Hackos explains that an information model is 'an organizational framework that you use to categorize your information resources' (Hackos 2002). Louis Rosenfeld and Peter Morville, in their book on *Information Architecture*, use the term *content model* and point out that a content model consists of 'chunks, relationships and metadata' (Rosenfeld Morville 2006). These explanation summarise the content editors' view quite accurately.

Actually, content editors are concerned with content artefacts that can represent all kinds of digital information. In addition, they maintain relationships between these artefacts, provide metadata and establish classification schemes. Relationships between content artefacts are relevant to how web pages are composed from these artefacts, and may result in hyperlinks. Classification schemes and metadata are, among other things, important for ensuring a content artefact's findability on the web.

Moreover, content editors have to be aware of content life cycles and workflow processes. Content is created and updated, edited and published, until eventually it expires and is removed. Teamwork among content editors can, for example, result in the application of a four-eye principle prior to publication.

It's clear from this discussion that there is more to a website than meets the user's eye. Users and content editors may look at the same thing, but their perspectives are very different.

Solution

Provide software for two distinct purposes. On one hand, you need content management software that supports the content editors in their job. On the other, you need content delivery software that makes content available to the web and controls possible user interaction. You won't have to develop the complete software yourself – a content management system typically provides some of the necessary functionality – but you must expect to develop a certain amount of custom software.

No content management system can foresee the specific requirements for your site. The more non-standard functionality you want, and the more you wish to integrate your site with backend systems, the more you'll have to expect to develop custom software that goes beyond merely customising the components that your content management system may provide.

The overall architecture for a website or web platform is illustrated in Figure 6. Let's now dig a little deeper and explore the two clouds in this diagram. We'll start with the cloud on the right-hand side – the software for content management:[1]

- Content management software has to provide functionality for creating and maintaining content artefacts, based on an underlying information model. This includes not only the definition of the actual content elements, but also the assignment of metadata and the linking of content elements.
- Because content editors need a feel for how their content will ultimately look, most content management software comes with a preview function that gives editors an impression of the resulting web pages.

[1] There is no unambiguous definition of the term 'content management', neither in the literature nor in practical web application development. Sometimes content management is supposed to include content delivery, sometimes it's not. Throughout this book I'll assume a narrower but more precise meaning: *content management includes the techniques and processes necessary for the creation and maintenance of content.* Content delivery is outside this definition of content management.

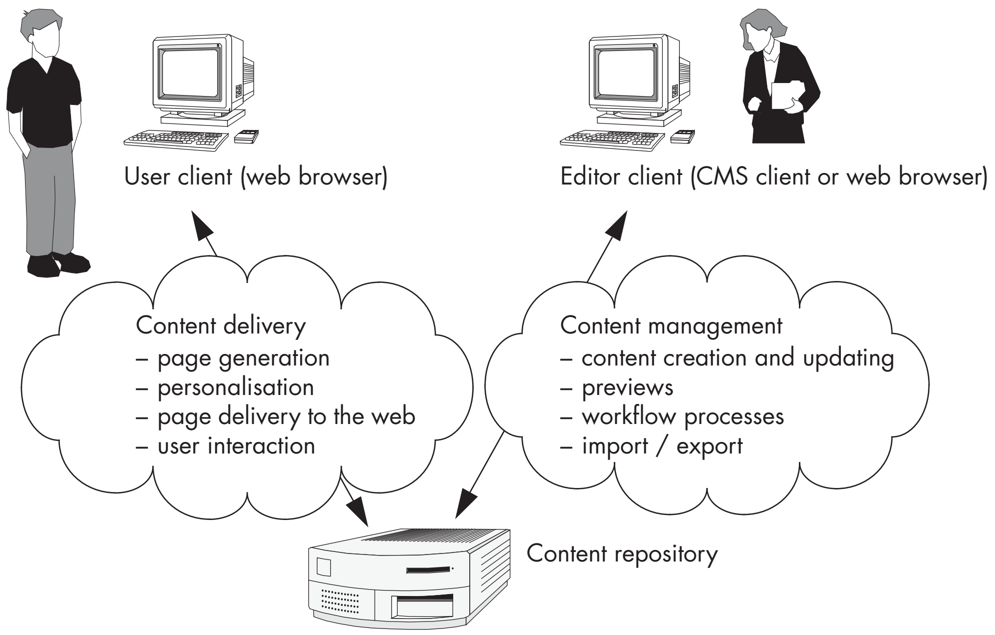


Figure 6: Content management and content delivery

- Content management software usually has to support workflow processes for content editors. This includes access permissions as well as rules that specify and control the collaboration between content editors.
- Content management software usually has to include import and export functions. Typically relying on an XML-based format, such functions make it possible to exchange content with other data sources.

Most content management systems offer tools that provide this functionality. There is usually an editor client, which could be a stand-alone application or one that's integrated into the web browser. Import and export functions might be integrated into the editor, although normally they come as a separate application.

Experience shows that on the content management side customisation is often sufficient to meet site-specific requirements. Developing your own tools, such as your editor client, should be the exception rather than the rule.

Things look different on the content delivery side. The software here is usually faced with more complex, site-specific requirements, so more custom software is necessary. Here are the main aspects of what content delivery software has to do:

- Before web pages can be delivered to the web they have to be generated from content artefacts stored in the content repository. This includes the choice of a layout as well as the creation of hyperlinks and interaction elements. For the time being, let's not make any assumptions about how or when this is going to happen. Suffice it say that web pages have to be generated somehow.

- Page generation gets more complex for a personalised site, as pages need to be tailored to specific users or user groups. Typically this means that, prior to page generation, content elements have to be selected that match a user's profile, so that the resulting pages include precisely the information intended for that user.
- After a web page has been generated, it is ready to be delivered to the web. Software is therefore necessary to react to requests received from a user's browser. This is primarily the job of a web server, but it may also involve a search engine or other backend components, depending on the domain logic that must be processed to fulfil the request. Custom software is usually necessary to implement the domain logic, which typically requires some kind of application server.
- Collaborative sites allow users to contribute content themselves. Strictly speaking the upload of user-generated content isn't an aspect of content delivery – the direction is actually the other way round, from the user to the site. The upload of user-generated content, however, is of course handled within the same request response scheme that is otherwise applied to content delivery. The difference is that a user request might result in write access to the content repository, so additional software is necessary to examine and process the user-generated content that is submitted.

In a rather simplistic scenario, a web server and a few off-the-shelf components from your content management system are all you need. In such cases you can typically specify the layout for your site by providing page templates in some scripting language, or by configuring the existing ones.

However, a larger and more complex site usually requires a good deal of custom software, especially for the implementation of domain logic and for embedding the site into a larger application landscape. Many content management systems react to this requirement by providing a framework that offers more hooks for you to customise the site. This can cause the software for your site to become a mix of prefabricated and custom components. In such cases it is wise to choose a content management system with a relatively 'open' architecture – one that is reasonably flexible and makes a smooth integration of your own components possible.

Figure 6 summarises the overall architecture, but intentionally leaves the details open. It emphasises the distinct chunks of software for content management and content delivery, but doesn't try to show what they look like in detail: what is covered by the clouds in Figure 6 can actually be implemented in many different ways.

Example resolved

The website for the House of Effects isn't just a collection of web pages. We are going to integrate a search engine, a personalisation engine and an online shop, so an out-of-the-box solution won't do. As far as content delivery is concerned, we'll have to develop custom software for implementing the domain logic and for gluing the pieces together.

However, we will be happy to use off-the-shelf components for the technical infrastructure, so we won't have to worry about XML processing, HTML generation, HTTP requests and the like. Ideally these things will be covered by our content management system, so we're looking for a system that allows us to use several prefabricated components and to bring in our own components at the same time.

For content management the plan is to use the editor client that the content management system provides. Ideally, this will be a web-based client that doesn't require a roll-out to all workplaces. It's clear that a configuration with regard to the underlying content model and workflow specification will be required, but no software development for the client should be necessary.

Benefits

+ The solution supports what is often considered to be the most fundamental principle of content management – the separation of content and layout. Content, when it's created and maintained, is completely decoupled from any layout aspects, which are only added later as part of the content delivery process. Content maintenance becomes straightforward, as it focuses on content and content alone.
+ Because content and layout are clearly decoupled, it is easy to create different layouts for the same content element. In other words, you can create different sites based on the same content, like an intranet and an extranet. Similarly, you can to support different output channels, such as browsers and mobile devices.
+ The solution emphasises the importance of the software on the content management side. A well-designed editor client makes life easier for content editors: well-chosen workflow processes help them work efficiently and ultimately contribute to higher content quality.
+ The solution also emphasises the flexibility that's necessary on the content delivery side. As you are able to integrate your own custom components, you can ensure that the site implements the domain logic you want it to implement.

Liabilities

– An information or content model is required as the basis for all content management and content delivery software. You have to define a model that reflects the domain-driven requirements on your site. The definition of a CONTENT TYPE HIERARCHY (2.1) is a good starting point.
– The solution emphasises the fact that web pages have to be generated from content artefacts, but it intentionally doesn't reveal when and how this should take place. While several strategies are possible, DYNAMIC CONTENT DELIVERY PLUS CACHING (1.2) is usually best.

– You'll have to develop custom software for the server side. When you do this, several more fine-grained components will turn out to be useful, including CONTENT SERVICES (3.1), a NAVIGATION MANAGER (3.2), a SEARCH MANAGER (3.3) and a SYSTEM OF INTERACTING TEMPLATES (3.4).
– You will have to choose an appropriate content management system. Most available systems support the two distinct perspectives described in this pattern, but when it comes to details content management systems differ widely. The ability to define a domain-driven content model in a straightforward way, an easy-to-use editor client and a sufficiently 'open' architecture are among the key requirements you should place on a tool. We will come across more criteria in some of the follow-up patterns. A checklist at the end of the book will help you to select a tool for your specific purposes.

1.2 Dynamic Content Delivery plus Caching

Context

You're in the process of defining the architecture for a website or a web platform. The big picture includes software for CONTENT MANAGEMENT AND CONTENT DELIVERY (1.1), so now a more refined architecture for content delivery and user interaction is required. Personalisation and user involvement may be on your agenda, too, and the architecture has to acknowledge that.

Problem

How can you ensure that the site is always up-to-date and reflects the latest changes made by the content editors? How can you lay the foundation for interaction and personalisation?

Example

Like other museums, the House of Effects relies on its website as a primary publicity channel. The owners therefore have an obvious interest in presenting accurate and up-to-date content. New content should be made available to visitors as soon as it's available. In addition, the site is supposed to present personalised content to registered visitors, so our software will have to take users' profiles into account when delivering any web pages.

Moreover, the site is going to be highly interactive. For example, there are going to be interactive online presentations to attract visitors. Next, registered users should be able to leave comments, buy books or DVDs from the online shop and rate shop items, which of course imposes more requirements on the content delivery software. The software architecture therefore has to ensure that these requirements can be met in an effective and efficient way.

Forces

Before pages can actually be delivered to the web they have to be assembled from content elements. This process, known as *page generation* or *page rendering*, consists of several smaller tasks: obtaining the required content artefacts from the repository, applying domain logic, adding links to other pages, perhaps adding personalised information, generating the actual HTML page, adding stylesheets.

Different strategies exist for the point at which page generation should take place. *Static delivery* assumes that pages are generated off line after publication and are stored in the repository in HTML format. *Dynamic delivery* assumes that pages are rendered on request – that is, when the server receives an HTTP request from a browser. *Hybrid delivery* is an intermediate strategy that combines static and dynamic delivery. These strategies have different pros and cons.

Static delivery is extremely fast, but dynamic delivery is much more flexible. First, dynamic delivery allows changes to the content to be reflected on the web pages immediately on publication, while static delivery might yield pages that are slightly outdated. Second, user interaction and personalisation usually require dynamic delivery, since the pages that are delivered depend on user input and the identity of the current user.

These days, most sites prefer dynamic delivery for its greater flexibility. Dynamic delivery is supported by almost all content management systems available on the market – which, however, leaves us with the performance issue to be resolved.

Solution

Combine dynamic content delivery with powerful caching strategies. Choose a content management system that generates web pages on request and offers caching mechanisms sufficient to meet your performance requirements.

Assuming dynamic content delivery, we can now draw a more concrete picture of what happens when pages are requested from a browser and are delivered to the web. It's clear that a web server is necessary to react to browser requests, and usually an application server is necessary as well to host the components that implement the domain logic. Figure 7 shows an overview of an architecture that implements this concept. This is a refinement of the cloud representing content delivery that appears on the left-hand side in Figure 6.

The following list provides an overview of the possible steps that constitute the page delivery process:

- The web server accepts a page request as well as possible user input from a browser. The request is mapped onto a template or other component that should be invoked to generate the response. This is essentially a lookup functionality that is usually provided by the content management system.
- The component that is invoked obtains the necessary content elements from the repository. Applying the underlying domain logic, it processes the content

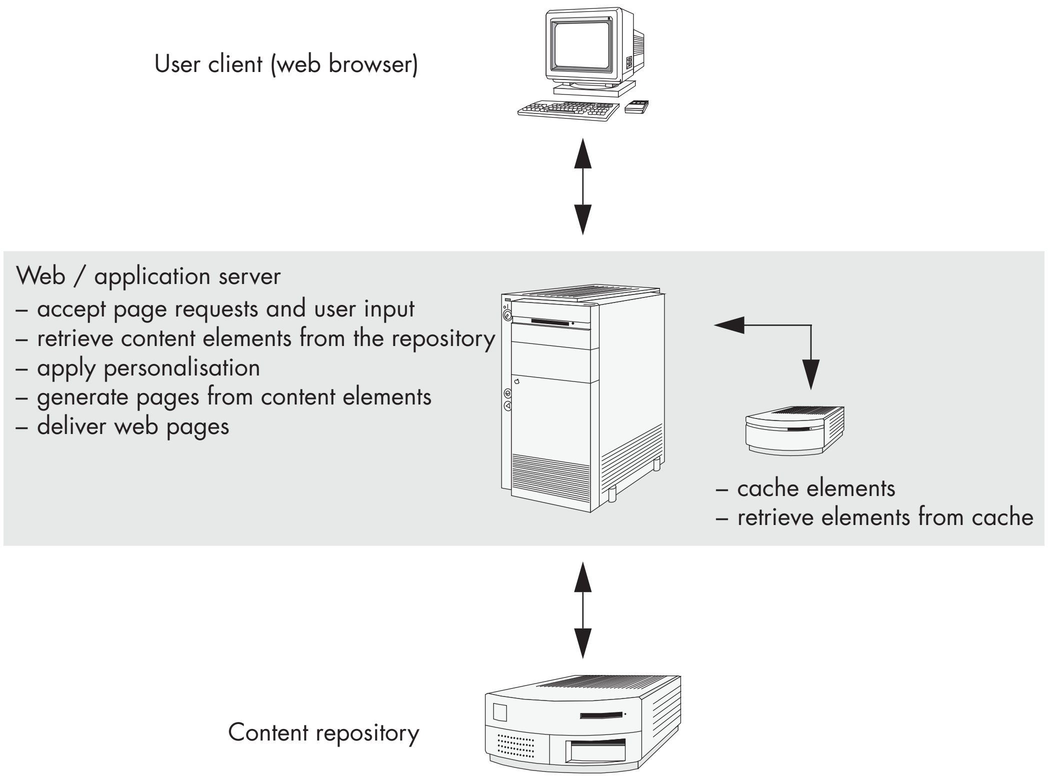

User client (web browser)
Web / application server
– accept page requests and user input
– retrieve content elements from the repository
– apply personalisation
– generate pages from content elements
– deliver web pages
– cache elements
– retrieve elements from cache
Content repository

Figure 7: Dynamic content delivery and caching

elements, performs link management and, if necessary, calls other backend components to collect all the information that should go into the web page.

- If required, personalisation is applied. What content elements are included and how they are processed may depend on the current user. Details vary, as personalisation can take on different forms.
- Finally, HTML has to be generated for the web page, which includes the assembly of page fragments representing the individual content elements and the addition of CSS styles. This is usually done by templates that implement the desired page layout.
- Once this is completed, the page can be sent to the browser.

Caching can take place throughout these steps. The general idea behind caching is always the same – frequently used objects are stored somewhere that offers fast access, as

Paul Dyson and Andy Longshaw explain in their book on *Architecting Enterprise Solutions* (Dyson Longshaw 2004).

However, different caching strategies are possible and are applied at different levels of granularity.

- An initial option is to keep content artefacts from the repository cached within the application server. Because content artefacts are typically requested over and over again, this strategy clearly reduces the number of repository calls, which are usually calls to a remote machine.
- A further option is to cache objects that are composed from content elements from the repository. These elements, as well as the way they are composed, represent the domain-driven content model.
- This strategy not only reduces the number of repository calls, but also reuses the application of domain logic.
- Finally, caching can be applied to HTML fragments or complete web pages. This allows HTML to be reused across user requests, which not only reduces the effort for the application of domain logic, but also reuses the application of templates or other components in charge of HTML generation.

All caching strategies require that cached objects be invalidated when their original source changes. Whenever a content element is updated in the repository, any cached object that relies on the element becomes invalid, meaning that it has to be regenerated if requested.

The more complex and dynamic is a cached object, the smaller the probability that it can be successfully reused before it undergoes invalidation. For example, personalised elements can easily become too numerous to be cached, as they will typically differ between one user and another. Similarly, page elements that depend on user input aren't easy targets for caching strategies. In general, content artefacts from the repository are more generic and can therefore be reused more easily than HTML fragments. It is therefore important to be careful when choosing the level of granularity at which caching should be applied.

Caching can be difficult to implement, and the need for content invalidation isn't going to make things any easier. Fortunately many content management systems come with their own – often quite powerful – caching mechanisms. Different systems favour different strategies, and may even combine several strategies to achieve significant performance improvements. Ideally you can rely on the capabilities of your content management system and won't have to implement any caching yourself.

Example resolved

The House of Effects website requires dynamic content delivery. First, this ensures that the site reflects changes made by the content editors immediately. Second and more importantly, dynamically generated pages are essential in the presence of interaction and personalisation. Dynamic delivery is a precondition for ensuring that, for example, pages can be tailored to the current user, or that comments left by users become visible immediately.

Obviously we have to expect specific page elements to be requested many times and by many different users, so we let the content management system apply caching whenever possible. For the time being, let's assume that our content management system offers advanced caching mechanisms and is able to combine caching strategies at different levels of granularity, so as to effectively improve performance even in the presence of interaction and personalisation. Fortunately for us, there is little we need to do at this point.

Benefits

+ Because all web pages are generated on request, they are up-to-date when they are delivered to the user. Changes made by content editors are reflected immediately on publication.
+ Dynamic delivery sets the stage for creating a website rich with user interaction. Because pages are generated dynamically, their contents can depend on user input. This gives you the chance to integrate interactive forms, display results from search engines and incorporate backend systems to build web applications.
+ Dynamic delivery also makes a personalised site possible, in which content is tailored specifically to the current user. Since pages are generated on request, the content chosen for inclusion on a web page can depend on who is currently logged in.
+ Caching speeds up the site, as it can significantly reduce the volume of remote calls and database access. Caching is particularly useful for large objects such as pictures or multimedia artefacts.

Liabilities

– Since dynamic page delivery and caching strategies both involve a series of non-trivial tasks, maintainability and scalability problems can occur. A LAYERED ARCHITECTURE FOR CONTENT DELIVERY (1.5) supports the separation of concerns and so offers a good solution.

– There is a limit to the usefulness of caching, especially for personalised and heavily interactive sites. Personalisation and caching are natural enemies, as are interaction and caching. As we have mentioned before, one way to alleviate the problem is to apply caching to relatively generic elements such as artefacts from the repository. However, there are other techniques that tackle this problem. At the HTML level, a well-tailored SYSTEM OF INTERACTING TEMPLATES (3.4) can increase the effectiveness of caching. If personalisation is applied to user segments rather than to individual users, SEGMENT-SPECIFIC CACHING (4.3) can improve efficiency.

- Dynamic page delivery requires content stored in the repository to be well-formed and consistent, or problems might occur during page generation. WORKFLOW-BASED VALIDATION (2.5) can help you ensure reasonable content quality.
- Cache invalidation requires that the cache be informed of all significant changes made to content artefacts in the repository. A LISTENER-BASED SYNCHRONISATION (1.4) between the repository and the application server can provide the necessary information.
- Caching isn't the only way to make a site faster. Moving functionality from the server to the browser is an option, especially for a heavily interactive site. This is the idea behind SENSIBLE CLIENT-SIDE INTERACTION (1.3).

1.3 Sensible Client-Side Interaction

Context

You are in the process of defining the architecture for a website or a web platform. The overall architecture embraces components for CONTENT MANAGEMENT AND CONTENT DELIVERY (1.1), and applies DYNAMIC CONTENT DELIVERY PLUS CACHING (1.2) to ensure the presentation of up-to-date content and meet the demands of user interaction and personalisation.

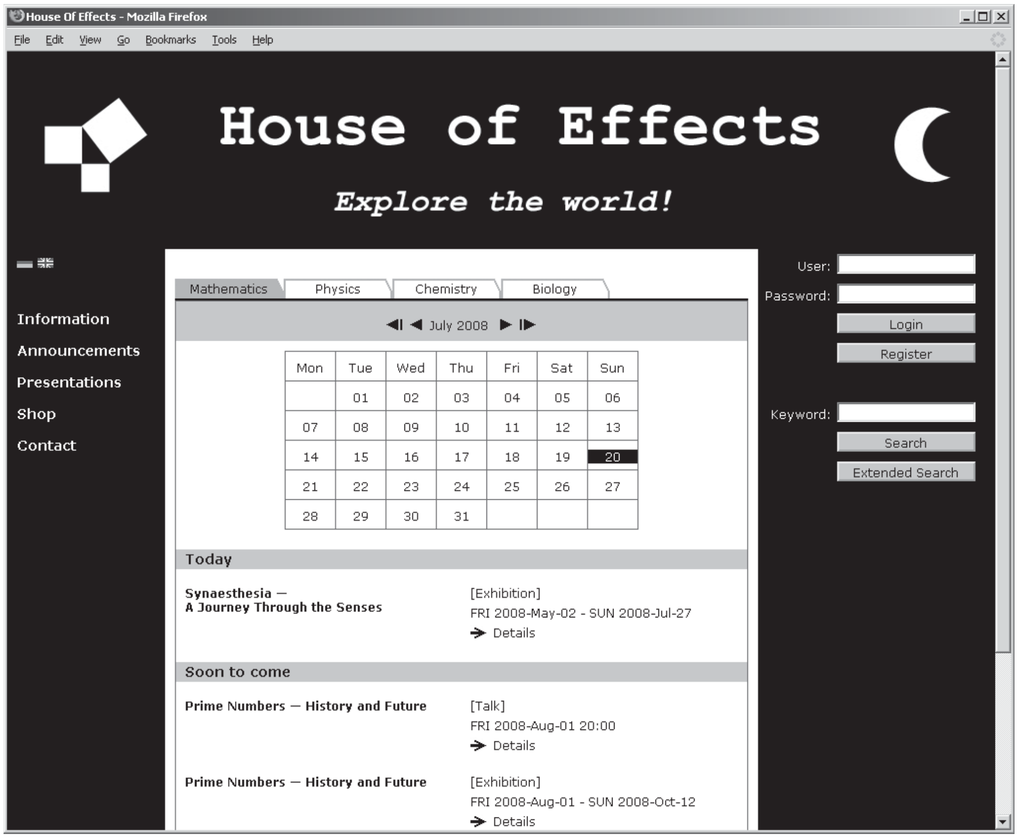
Problem

How can you ensure that your site features the desired degree of interaction and user participation while maintaining reasonable system performance?

Example

The website for the House of Effects is going to offer various kinds of interaction. Users will be able to navigate the site, submit search requests and filter and sort the search results. There will be interactive online presentations. Users will be able to submit comments, buy items from the online shop and rate shop items. Essentially there are two different ways in which the necessary user interaction can be implemented: on the server or on the client.

To go into more detail, let's look at the event calendar shown in Figure 8. Users can select a tab ('Mathematics', 'Physics', 'Chemistry' or 'Biology') and so apply a filter to the list of events shown below. Implementing these tabs can be done in different ways, either by using standard hyperlinks and traditional server communication or by Ajax-based event handling that involves only the client. Which is preferable?



July 2008

Mon	Tue	Wed	Thu	Fri	Sat	Sun
	01	02	03	04	05	06
07	08	09	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Today

Synaesthesia —
A Journey Through the Senses
[Exhibition]
FRI 2008-May-02 - SUN 2008-Jul-27
➔ Details

Soon to come

Prime Numbers — History and Future
[Talk]
FRI 2008-Aug-01 20:00
➔ Details

Prime Numbers — History and Future
[Exhibition]
FRI 2008-Aug-01 - SUN 2008-Oct-12
➔ Details

Figure 8: Event calendar for the House of Effects

Forces

With the advent of Web 2.0 a high degree of interaction has become increasingly common among websites world-wide. On the technical level, Ajax (Asynchronous JavaScript and XML) is the key concept behind Web 2.0 (Garrett 2005). Ajax makes it possible to react to user events directly in the browser without having to direct any HTTP requests to the server, provided that the necessary event-handling mechanisms have been deployed to the browser in the first place. Although it's not a precondition for the 'Collaborative Web', Ajax can facilitate the implementation of user participation and collaboration.

There is no doubt that this type of client-side interaction has some powerful advantages. First, the browser can sometimes process input submitted by a user without having to load a new page, which reduces network traffic. Second, asynchronous loading is possible, which means that large objects such as multimedia artefacts can sometimes be loaded in the background while a page is already displayed. The combination of both techniques makes a degree of interaction possible that is unknown from traditional websites.

Traditional websites aren't necessarily a thing of the past, but it's clear that Web 2.0 techniques play a more important role than they did a few years ago. Because Ajax-based sites can be both more interactive and faster, Ajax technology has become an integral part of today's advanced websites.

But there are drawbacks to Ajax technology. First, extensive use of Ajax can blur the concept of a web page. If navigation can be handled by the browser, content that used to be distributed over several web pages may end up on what is technically no more than a single page. As a consequence, bookmarkability suffers – only a page can be bookmarked, but not the pieces of information loaded into it by a client-side JavaScript mechanism. Similarly, search engines have a hard time referring to information contained within an interactive page, as they can only return links to full pages and not to any content fragments that are made available by client-side functionality.

A second important disadvantage lies in the fact that code written in scripting languages such as JavaScript is notoriously difficult to understand, test and maintain. In defence, there are Ajax libraries on the market that alleviate this problem, and to some extent it is possible to produce well-structured JavaScript code. But larger applications remain tricky when written in JavaScript.

Yet another disadvantage lies in the browser dependencies that are inevitable once Ajax is introduced. Users must have JavaScript switched on if they want to use an Ajax-based site, and not everybody has. Some users rely on a speech or Braille output device for which JavaScript isn't available, which raises an accessibility issue. Even if you can assume that all users have JavaScript turned on, exactly what they see in their browser still depends on the browser they're using. A lack of standardisation causes different browsers to interpret JavaScript slightly differently, at least for the time being.

You can avoid all these disadvantages if you restrict your site to server-side interaction. But this would slow down response time and, as a consequence, would make highly interactive platforms virtually impossible.

Solution

Use Ajax-based client-side interaction, but use it with care. Retain the concept of a web page and apply server-side event handling for all navigation purposes, but also apply event handling inside the browser to adjust the way in which information elements are presented within a web page. Combine this with asynchronous server calls if the browser has to load data from the server.

The idea is not to set up a single page and let Ajax-based techniques load whatever information is requested. Such an approach, referred to as *Ajax deluxe* in Michael Mahemoff's book on *Ajax Design Patterns*, can be the right choice for web applications that should ‘feel similar to a desktop in that the browser is driving the interaction’ (Mahemoff 2006).

Things look different, though, for a web platform that is supposed to combine information with a certain amount of user interaction. It makes perfect sense to have several web pages and so to distribute information over some kind of navigational space. The trick is to combine traditional server calls for travelling from page to page with Ajax-based event handling for the presentation of information in the browser. Michael Mahemoff calls this strategy *Ajax lite*. It is a well-balanced approach in which Ajax mechanisms are carefully used in those places where they can do good.

Exactly what interaction should happen on the client (the browser) and what should take place on the server? Although to some degree a decision will be a matter of personal taste, it is possible to give some concrete advice.

Michael Mahemoff describes two fundamental Ajax patterns for display manipulation, ‘display morphing’ and ‘page rearrangement’ (Mahemoff 2006). Both have in common that they alter the view of what is presented on a page through relatively simple manipulations of the domain object model (DOM). Interactions that result in this kind of display manipulation are best handled inside the browser with Ajax-based techniques. The following list presents a few typical examples:

- Tabs and scrolling, or similar GUI techniques for making information visible on the screen. Well-known from desktop applications, these techniques often make sense for websites too. Ajax allows you to use these techniques on a web page without making any server calls.
- Filtering and sorting lists of items. Lists of items are common enough, and users are often given the choice of how such a list should be presented. With Ajax-based techniques you can allow users to change the sort order or apply a filter without having to make a server call.
- Interactive forms. Choosing values from selection boxes and the like can easily be dealt with on the client side. It is also common for forms to spawn additional fields depending on the input already made by the user. While this is generally impossible with static HTML, Ajax allows you to implement dynamic forms in a straightforward way.
- Asynchronous loading of large objects, such as videos or other multimedia objects. Loading such an object is typically invoked by the page that contains it, directly after the page itself has been loaded. The page is made available to the user while some of its contents are still loading in the background, for example by using some kind of streaming mechanism.
- Use of multimedia objects, once they have been loaded. For example, events for starting or stopping a video should be handled directly in the browser, with no server-side event handling at all.
- Content updates. Certain content elements, such as news items, booking information and so on can change frequently. You can apply Ajax-based

asynchronous loading, which is usually triggered by the client that requests up-to-date content from the server at regular intervals.

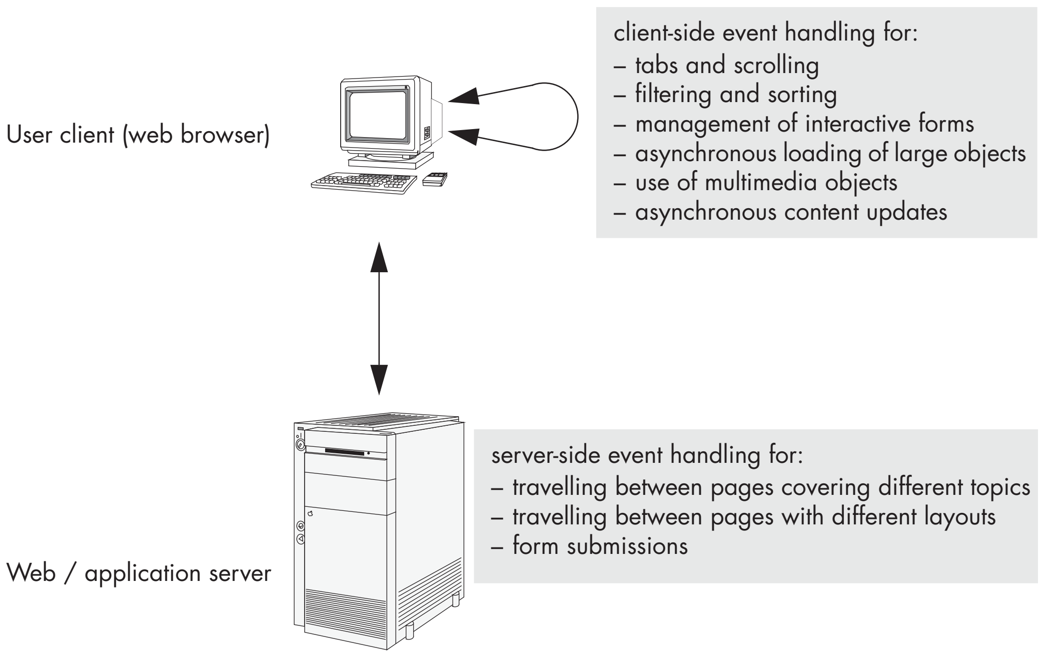

User client (web browser)
client-side event handling for:
– tabs and scrolling
– filtering and sorting
– management of interactive forms
– asynchronous loading of large objects
– use of multimedia objects
– asynchronous content updates
Web / application server
server-side event handling for:
– travelling between pages covering different topics
– travelling between pages with different layouts
– form submissions

Figure 9: Event handling for user interaction

On the other hand, what kinds of user interaction are better handled on the server side? The following list gives some typical examples:

- Pages addressing different topics. Imagine a logical, domain-driven site map, in which different topics are represented by different pages. What appears as a distinct page in this logical model should be technically implemented as a distinct web page as well. This allows travelling from page to page to become a matter of server-side event handling. The concept of web pages is retained.
- Pages with different layouts. If two pages have different layouts, then it's probably a good idea to keep them as separate pages and not map them onto one. The different layouts suggest that they present different kinds of information to visitors.
- Form submissions. While filling in an interactive form can usually be handled on the client side alone, the submission of the form marks the end of a use case and typically triggers a backend transaction. In most cases this justifies a new page (invoked by a server-side event) so that the user is informed of the transaction being completed.

Neither of the two lists is necessarily complete, but they should still give you a good impression of the two types of interaction and how to tell them apart. Figure 9 gives a brief summary.

Finally, there are two things you should keep in mind when implementing this pattern. First, make sure you use Ajax libraries whenever possible. Several good libraries are available these days, some of which have been published by open source projects (www.icefaces.org, labs.jboss.org/jbossrichfaces). Using such libraries helps you to reduce the amount of client-side functionality that you have to develop yourself.

Second, you need to be concerned with accessibility issues, especially if you develop a public administration site or any other site that has to comply with accessibility standards. If you can't be sure that the output devices you have to support are capable of Ajax-based mechanisms, you'll have to supply a version of your site that is completely independent of any browser functionality and relies on server-side event handling alone. In such a case, the server has to check for the availability of an Ajax-capable browser when receiving a page request, and deliver the correct version accordingly.

Example resolved

We are going to use Ajax techniques to add rich interaction to the House of Effects site. For example, the event calendar from Figure 8 is going to be implemented using Ajax. The page will contain a complete list of events, but its actual view will be adjusted whenever the user selects a tab without any server-side event handling becoming necessary. We will also be using Ajax for the interactive online presentations that we plan to implement.

On the other hand, the House of Effects site is going to use traditional server-side event handling for all navigation purposes. All navigation elements and other references to related pages will be implemented through HTTP requests, as will the submission of a search term, booking requests and purchase orders. The idea of a website as a navigational space will, after all, remain intact.

What about accessibility? We don't have to meet any special requirements, but what if we did? We could offer a completely Ajax-free version if we tested, at the start of each session, whether the user had JavaScript switched off, and delivered traditional HTML in this case. This would represent extra effort, though, and as it's not required we have no plans to implement this option.

Benefits

+ Client-side interaction gives your site a higher degree of interaction. You can embed interactive mechanisms simply that would not be possible if every user input resulted in a new page request. Starting or stopping a video embedded in a page is only one example – the interaction needed for user participation is another. You can turn your site into an interactive platform and improve its usability.

- + Client-side interaction doesn't depend on network resources and is much faster than a series of server requests. In addition, asynchronous server communication allows you to load large objects in the background. Client-side interaction has a positive impact on your site's performance.
- + The moderate use of client-side interaction retains the concept of a web page. Bookmarkability and searchability therefore aren't impaired, as they would be if you used Ajax extensively.
- + The moderate use of client-side interaction also means that less JavaScript code becomes necessary, as opposed to a heavily Ajax-based site. As you only adjust the view of page elements but don't use Ajax to change an entire page's content, client-side interaction will not result in any fundamental changes to the domain object model (DOM) behind a web page. To implement the necessary SELF-CONTAINED PAGES (3.6), a small amount of standard JavaScript code will do, which you should probably be able to find in typical JavaScript libraries. Because you don't have to develop extensive JavaScript functionality yourself, comprehensibility, maintainability and scalability are clearly improved.

Liabilities

- – Client-side interaction, even if applied in a disciplined way, introduces browser dependencies. Either you accept the fact that different browsers might present your site slightly differently, or you have to develop functionality targeted specifically at different browser types. This, of course, represents an additional effort for software development.
- – If accessibility is an issue, you may even be forced to provide a non-Ajax version of your site. If, for example, you're required to support speech or Braille output devices, current technology demands that you make a version of your site available that is independent of client-side interaction altogether. This has an influence on the entire architecture for your website, and you must expect significant additional effort for its development.
- – Testing a platform that uses both client-side and server-side interaction is more difficult than testing a site that relies on server-side interaction alone. This is partially due to the inherent complexity of a more sophisticated architecture, and partially due to the lack of support for client-side interaction by today's development environments. The latter may change – it is likely that development tools will soon become available that support Ajax better than most do today. The increased complexity will still take its toll on the development effort, however.
- – Security requirements demand that users must not be able to tamper with critical data. Some data shouldn't even be visible to users. It's a wise strategy to assign functionality to the client only if this functionality doesn't have to process any data that users shouldn't be able to modify, let alone data that users shouldn't be able to see.

1.4 Listener-Based Synchronisation

Context

You're in the process of defining the architecture for a website or a web platform. The overall architecture consists of software for CONTENT MANAGEMENT AND CONTENT DELIVERY (1.1). The actual content is stored in a repository where it is maintained by content editors following specific workflows. DYNAMIC CONTENT DELIVERY PLUS CACHING (1.2) is applied to ensure the presentation of up-to-date content and to meet the demands of user interaction and personalisation. Additional components such as a search engine or a personalisation engine might also be part of the overall architecture.

Problem

How can you avoid inconsistencies between content in the repository and content stored by other components?

Example

The content management system's repository is, of course, the primary place where content for the House of Effects site will be stored. This is where editors will create and maintain content according to workflow processes.

However, it will be necessary to store content in other places as well. An initial example is the content management system's cache, which keeps copies of elements that are frequently requested. A second example is the search engine. It may not store complete content elements, but it will maintain links to pages that are generated from content elements, along with specific metadata that's necessary for processing a search request. A further example is the personalisation engine. Regardless of whether this engine is part of the content management system or a stand-alone application, it must know about the content elements that are subject to personalisation. A final example is the fact that the software required to support our online shop will have to keep lists of shop items as well as pricing information, which may overlap with the information stored in the content repository.

It's clear that in all these cases inconsistencies have to be avoided.

Forces

Although there is no question that the content repository is the primary source for content elements throughout the system, some components may have to store their own copies of content elements. There are different reasons for this.

The first and most important is performance. Most notably, a cache stores objects redundantly so that they can be retrieved quickly, and so to some extent avoids the normally costly access to the content repository. However there is a price to pay if you want to reduce remote calls and database access. Whether the cache is part of your content management system or part of the custom software, whether the cache stores objects from the domain model or HTML fragments, in either case cached elements must be invalidated when their source in the repository undergoes a change.

A second reason is the use of a third-party component that requires its own repository. Examples include an external search engine, an external personalisation engine, online shop software or a billing component. It is highly likely that there are overlaps with the content repository, so replicating the necessary content elements is the straightforward solution. But then again, you introduce redundant data, so if you want to avoid things such as invalid search results, inaccurately personalised pages or invalid transactions, consistency has to be ensured.

In fact, ensuring consistency has to be done in a way that's quick and robust. Interested components must learn of changes in the content repository immediately. Whatever notification mechanism you use, it must be able to deal with any of the components involved being down.

Solution

Establish repository listeners – asynchronous processes that react to specific workflow events and notify interested components of relevant changes made to content artefacts in the repository.

Good content management systems offer a listener interface or a similar mechanism that you can use to react to specific events in the content management workflow. Typical events include the creation, change, publication or deletion of a content element. On such an event, a listener can be invoked and will then execute a call-back method. You can implement repository listeners that notify other components of all relevant events.

In principle, you need a repository listener for each component that has to be informed of content changes. Typically though, you won't have to implement all listeners yourself:

- If your content management system uses a built-in cache (which it probably does), it will also have a built-in listener that invokes the necessary content invalidation mechanisms for that cache.
- If your content management system uses a built-in search engine, it will also have a built-in listener that notifies the search engine of any events that make it necessary to rebuild the index.
- Similarly, any other redundant data storage that is internal to your content management system should come with its own repository listener.

Since repository listeners react to workflow events, they usually run on the content management server – the machine that hosts the content editor workflows. The content management server is a core component of any content management system, therefore little custom software should be necessary here. Nonetheless, it is here where you have to register your custom repository listeners in order to add them to the built-in ones. Figure 10 gives an overview, representing notification by dotted lines.[2]

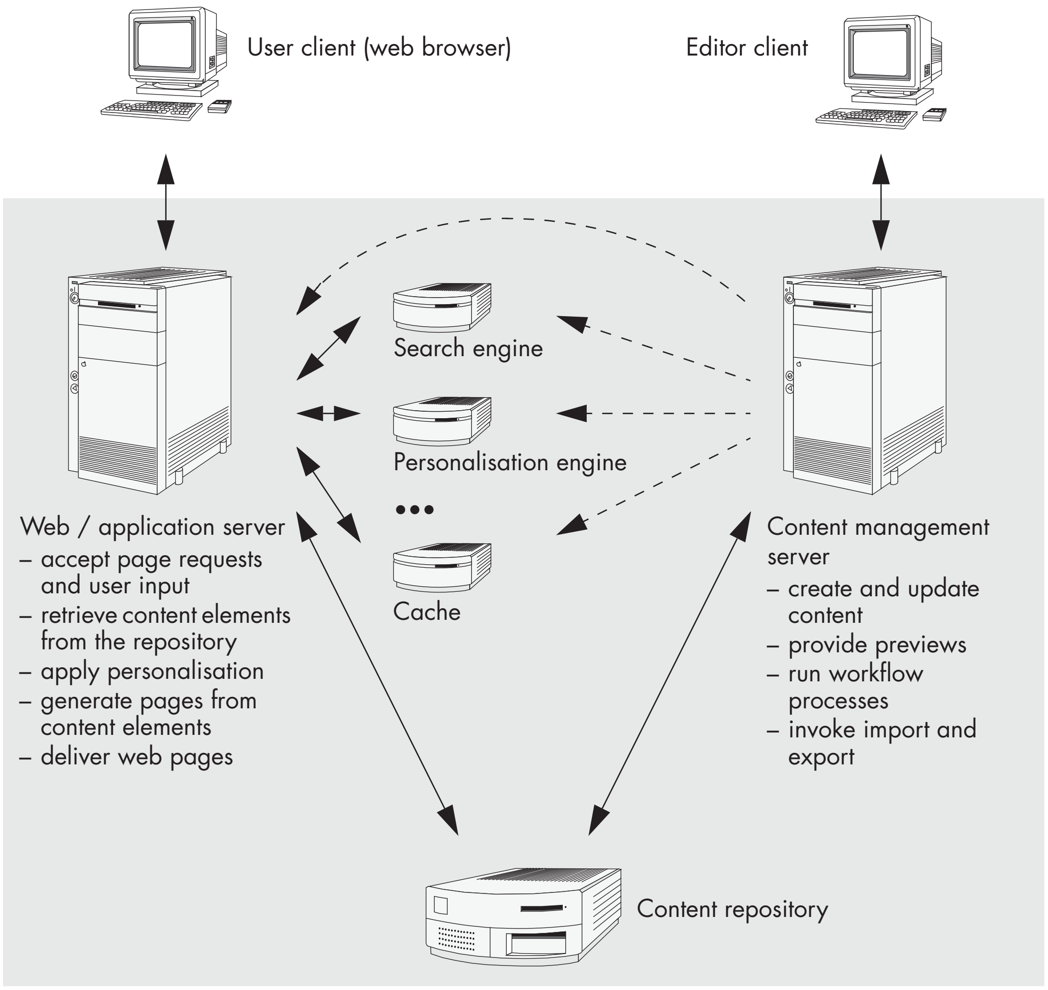

User client (web browser)
Editor client
Search engine
Personalisation engine
•••
Cache
Web / application server
– accept page requests and user input
– retrieve content elements from the repository
– apply personalisation
– generate pages from content elements
– deliver web pages
Content management server
– create and update content
– provide previews
– run workflow processes
– invoke import and export
Content repository

Figure 10: Repository listeners

This architecture is an implementation of the *Publisher-Subscriber* pattern (Buschmann Meunier Rohnert Sommerlad Stal 1996), which is a large-scale variant of the *Observer* pattern (Gamma Helm Johnson Vlissides 1995). The sole source of all content

[2] Bear in mind that Figure 10 shows a logical architecture. Concrete installations can deviate from this. For example, the search engine and personalisation engine could either be stand-alone components or be hosted by the application server. The cache typically resides in the application server and is visualised here only to underline its importance. The content management server and the content repository may be hosted by different machines or by the same machine.

artefacts, the content repository acts as the publisher, while the components that require notification take on the role of subscriber.

To work reliably, all repository listeners must be able to cope with the content repository, the content server or any other component being down. When you implement a repository listener, be sure to apply buffering logic at both ends:

- Let a listener look for past events during its start-up – events that occurred while the listener was down.
- Let a listener write all notifications into a queue from which a notification is only removed once the subscriber has successfully processed it.

This ensures that neither repository events nor notifications can get lost, turning the listeners into fail-safe synchronisation mechanisms between the different components of your architecture.

Example resolved

Let's make the (realistic) assumption that our content management system has a built-in mechanism for cache invalidation. As we don't implement any caching ourselves, no custom listener is necessary here.

However, there are three listeners that we will provide. First, we have to implement a repository listener that reacts to changes in the published content and feeds the search engine with the necessary indexing information. Second, we have to implement a listener that notifies our personalisation engine of any relevant changes in the repository, such as updates to user segments. Third, we have to implement a listener that reacts to changes made to item descriptions and informs the online shop system.

To ensure robustness, our repository listeners will implement a buffering logic. First, each listener uses a persistent time stamp to document its last activity, and will at start-up ask the content management server for all events after that time. Second, each listener stores the notifications it generates in a queue, from which they are only removed after they have been received and acknowledged by the target component.

Benefits

+ One of the most prominent examples of listener-based synchronisation is cache invalidation. This pattern therefore facilitates the implementation of caching strategies (either as part of a content management system or as a custom component) and so contributes to a website's efficiency.

+ Listener-based synchronisation makes it possible to keep content consistent across several components. It is therefore the precondition for successful and robust integration of different software modules. Listener-based synchronisation allows you to pursue a best-of-breed strategy when it comes to choosing tools – a content management system, a search engine, a personalisation engine, shop software and so on.

Liabilities

– Content consistency relies on the fact that all repository listeners work reliably. If a listener is down, its subscribers are no longer informed of relevant workflow events. To avoid inconsistencies (which, for a while, might even go unnoticed by content editors and users alike) you can establish watchdog processes to make sure that repository listeners are restarted automatically.
– The solution assumes that your content management system provides a listener interface that you can implement. You should make the possibility of implementing and registering repository listeners an evaluation criterion when choosing a content management system.

1.5 Layered Architecture for Content Delivery

Context

You plan to develop a website or web platform. You have set up the overall software architecture, whose most important constituents are the software packages for CONTENT MANAGEMENT AND CONTENT DELIVERY (1.1). Along the way, you have applied DYNAMIC CONTENT DELIVERY PLUS CACHING (1.2), SENSIBLE CLIENT-SIDE INTERACTION (1.3) and LISTENER-BASED SYNCHRONISATION (1.4) to refine the architecture, which allows you to meet important functional and non-functional requirements.

Perhaps a few – but typically not many – custom components will become necessary on the content management side. Your content management system should provide most of the required functionality – a small amount of customisation is usually all you need. However, the content delivery side often requires a considerable quantity of custom components, as it is here where most of the domain logic has to be implemented.

Problem

How can you prevent the server-side custom software for content delivery from becoming difficult or impossible to maintain? How can you avoid a server-side architecture that doesn't scale properly?

Example

The website for the House of Effects doesn't require much custom software for content management. We certainly have to configure the content management server to match the underlying content model, we have to specify workflow processes, and we have to implement a few repository listeners. However, this isn't exactly what you would call extensive custom software development.

We need a good deal of custom software development on the content delivery side, though, as many web platforms do. Among other things, we have to define our own domain logic, design templates that match our layout requirements, and implement some personalisation functionality. We have to integrate third-party components such as a search engine and an online shop. All in all, we had better not underestimate the amount of custom software.

Of course, the site owners are interested in keeping the website maintainable, despite the undeniable complexity. Future changes must be possible with reasonable effort. The owners are also interested in keeping it scalable. It must be possible to adapt the architecture should the amount of content or the number of users increase.

Forces

Dynamic delivery often involves a large number of different components. Server-side components have to retrieve content elements from the repository, apply domain logic, maintain a session state, apply personalisation, apply templates to generate HTML and apply caching. If there is going to be client-side interaction, then server-side components must provide the JavaScript functionality that is to be executed in the browser. Finally, third-party products may have to be integrated. Examples include a search engine, a personalisation engine or shop software. A content management system usually covers some of this functionality, but typically a considerable amount of custom software remains.

As you implement much of the necessary functionality yourself, you have to be concerned with important non-functional requirements such maintainability, extensibility and scalability. The more you make use of a content management system's open architecture – the possibility of integrating custom components smoothly – the more you're responsible for the architecture that evolves.

Moreover, you will typically come across different technologies and different programming languages. Usually there is some scripting code (JSP or the like) for HTML generation, programming language code (especially for the domain logic) and JavaScript (for the functions that will be executed directly in the browser). This adds to the architecture's complexity.

However, unmanaged complexity makes software difficult to understand, maintain and extend. Yet experience shows that in existing websites and web platforms the server-side software is often a mess, especially if server pages are used extensively. In his book on *Enterprise Application Architecture*, Martin Fowler notes: 'When domain logic starts turning up on server pages it becomes far too difficult to structure it well and far to easy to duplicate it across different server pages. All in all, the worst code I've seen in the last few years has been server page code.' (Fowler 2003).

Solution

Define a server-side architecture that consists of three distinct layers. The bottom layer encapsulates all access to the content repository. The middle layer provides the domain logic. The top layer contains the templates that are used for page generation.

The *Layers* pattern, a long-valued architectural principle, achieves a separation of concerns through vertical decomposition. The introduction of layers allows you to decompose an application into groups of subtasks at different levels of abstraction (Buschmann Meunier Rohnert Sommerlad Stal 1996).

The architecture sketched in Figure 11 is the result of applying the *Layers* pattern to content delivery software. The server-side architecture consists of three layers, much in agreement with the *three principle layers* that Martin Fowler recommends for web applications in general (Fowler 2003). Similar ideas are expressed in Michael Weiss's *Patterns for Web Applications*, especially a strict separation of content and presentation and the use of services to provide an application with the content it requires (Weiss 2006).

Let's go through these layers from bottom to top:

- The repository layer encapsulates all access to the content repository. Every content management system provides an interface for accessing content in the repository, and the modules behind this interface could very well constitute the repository layer. However, you may choose to develop some custom software that wraps this interface and provides, for example, syntactical validation or simple formatting routines. This allows the repository layer to make 'polished up' content elements available to the logical layer.
- The logical layer hosts the domain logic. This is where domain objects are composed from content elements, which may involve link management, session handling and personalisation. The logical layer is usually connected to external components such as a search engine, a personalisation engine, or arbitrary backend systems. The logical layer makes domain objects available in two different ways. First, it makes them available to the template layer. Second, it makes them available through a web service interface that client-side Ajax modules can use for server communication. While a content management system may provide a framework for integrating all this functionality, you must expect a good deal of custom software to be necessary to implement the domain logic.
- The template layer is where HTML generation takes place. This is the only place where server pages seem appropriate, though alternative techniques (such as servlets) could also be used. Relying on domain objects provided by the logical layer, templates generate web pages and include style sheets and possible client-side functionality that embody the page layout.

Caching can, in principle, take place in all layers. Depending on the layer, different kinds of objects can be subject to caching, ranging from content artefacts on the repository layer, through domain objects on the logical layer, to HTML fragments on the template layer. Which of these options becomes effective depends, of course, on your content

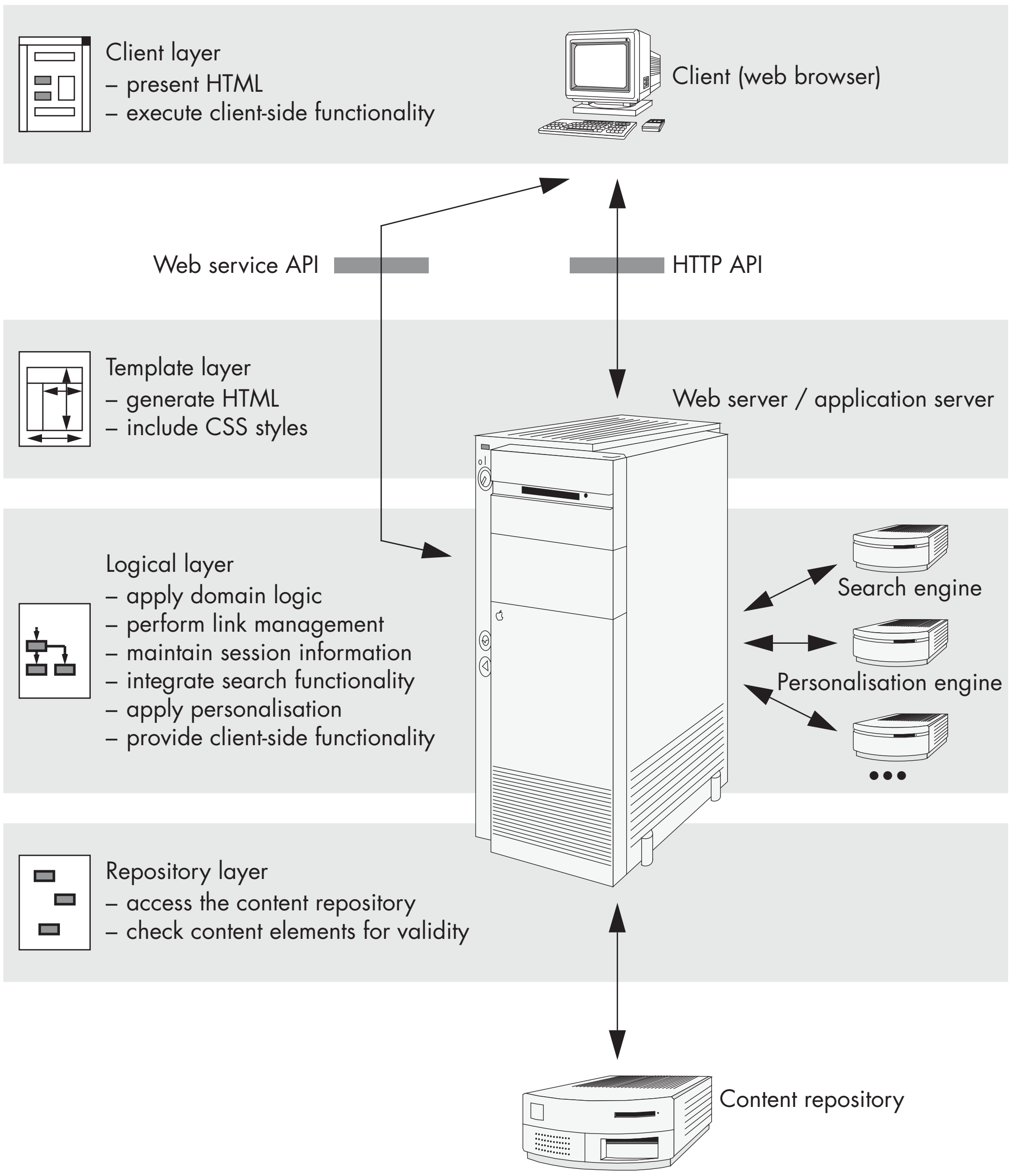

Client layer
– present HTML
– execute client-side functionality
Client (web browser)
Web service API
HTTP API
Template layer
– generate HTML
– include CSS styles
Web server / application server
Logical layer
– apply domain logic
– perform link management
– maintain session information
– integrate search functionality
– apply personalisation
– provide client-side functionality
Search engine
Personalisation engine
Repository layer
– access the content repository
– check content elements for validity
Content repository

Figure 11: Layered architecture for dynamic content delivery

management system or on your own caching strategies, should you decide to implement any yourself.

Figure 11 gives a rather general picture of a layered architecture for content delivery. The details depend on your content management system, the interfaces it offers and the underlying technology. Details vary with regard to the programming language (which may or may not be Java), available frameworks (such as Struts or Spring), backend integration and caching strategies.

Whatever content management system you use and whatever architectural consequences this has, you should aim for a layered architecture that implements a separation of concerns at different levels of abstraction.

Example resolved

We choose a Java-based content management system that allows us to integrate custom components into the content delivery process. Since maintainability is a critical issue for the House of Effects site, we make sure that the components for content delivery will be organised in a layered architecture.

The repository layer is going to be quite simple. It deals with the various kinds of content artefacts that are stored in the repository – multimedia objects such as online presentations, announcements, shop item descriptions and so on.

The domain objects on the logical layer are more complex than this. Things like presentations or announcements are meaningful in the domain, but there are also domain objects that relate or aggregate several content elements. Examples includes lists of events for the event calendar, complete with references to individual events, or comprehensive online presentations including detailed background information and user comments. Personalisation is applied too, so the way in which domain objects are composed may depend on the current user's profile. As we have opted for a Java-based architecture, Java beans are the natural choice for implementing these objects.

The template layer relies on JSP technology, but also uses tag libraries to reduce the amount of server page code. There are tags for smooth integration of domain objects into the final web page, so the actual server page code only defines the page structure, includes the CSS sheets that specify the page layout and provides the JavaScript functions that the browser needs for client-side interaction.

Benefits

+ A layered architecture avoids monolithic blocks and so decreases the coupling between system components. The vertical decomposition – the clear separation of domain logic and presentation – leads to reduced dependencies between components of all kinds, which improves comprehensibility and maintainability.

+ Different layers can be implemented using different technologies and different programming languages. This, too, contributes to improved comprehensibility and maintainability. In particular, the use of server pages is confined to the template layer, which avoids the feared 'spaghetti code' scenario of extensive domain logic implemented in a scripting language.
+ The software from different layers can be deployed onto different physical machines. Scalability can therefore be improved, as you can effectively address performance requirements by selecting appropriate hardware for each layer specifically.
+ There are several places where caching can be applied. Different caching strategies can be combined to achieve significant performance improvements. For example, you can cache content on the repository level if it is subject to personalisation, and use the template level to cache HTML fragments that are unaffected by personalisation. This allows you to maximise the efficiency benefits that caching brings.
+ The domain logic implemented on the logical layer can be used in two distinct ways: by the templates that generate HTML and by client-side functions for browser-server communication. The introduction of a well-defined logical layer therefore avoids redundant domain logic code to a large extent.

Liabilities

– There are few drawbacks associated with the definition of a layered architecture. The most critical issue is that the solution requires the content management system to support the vertical decomposition of custom components, and not every system on the market does. In fact, if your content management system is inflexible, implementing a layered architecture may turn out to be difficult. In such cases the ultimate recommendation is to consider using a different content management system. Better yet, make sure initially that you select a system that gives you the necessary freedom to organise your own server-side components.

– Since the introduction of layers is a high-level architectural pattern, it cannot extend so far as to facilitate good designs for the individual layers. There is no doubt that on a more fine-grained level there is still work to be done. Once you have implemented this pattern, you can start designing the individual layers and think about the introduction of CONTENT SERVICES (3.1), a NAVIGATION MANAGER (3.2), a SEARCH MANAGER (3.3) and a SYSTEM OF INTERACTING TEMPLATES (3.4).

CHAPTER

2

Content Management

In the previous chapter we gained an overview by looking at the big picture of the software architecture behind a website. We identified software for content management and software for content delivery as the prime constituents, and we analysed some of their characteristics and variations. It's now time to move to a more detailed level. We'll begin with patterns for content management in this chapter, while content delivery is on our agenda for Chapter 3.

Much of the content management functionality your site requires is likely to be provided by your content management system. Whatever system you choose, the chances are that it will provide you with a content editing tool and a workflow engine, so you will probably not have to develop much custom software for content management. What you *have* to do, however, is come up with an appropriate content model for your site and make the necessary configurations to your system.

This is where the patterns in this chapter start. They address the following questions:

- How can you model the content artefacts for your site? How are artefacts composed from smaller elements? How are content artefacts related?
- How can you integrate content into the navigation hierarchy for your site? How do you specify what content should appear on which page?
- What about findability? What mechanisms can content editors use to increase the likelihood that users find the content they're looking for?

You can see from these introductory questions that much (though not all) of this chapter is about modelling. But although the focus is on modelling, the following patterns still take an IT perspective. They are targeted at software designers and developers involved in a web project and they address modelling and design problems that, in the context of such a project, are common enough.

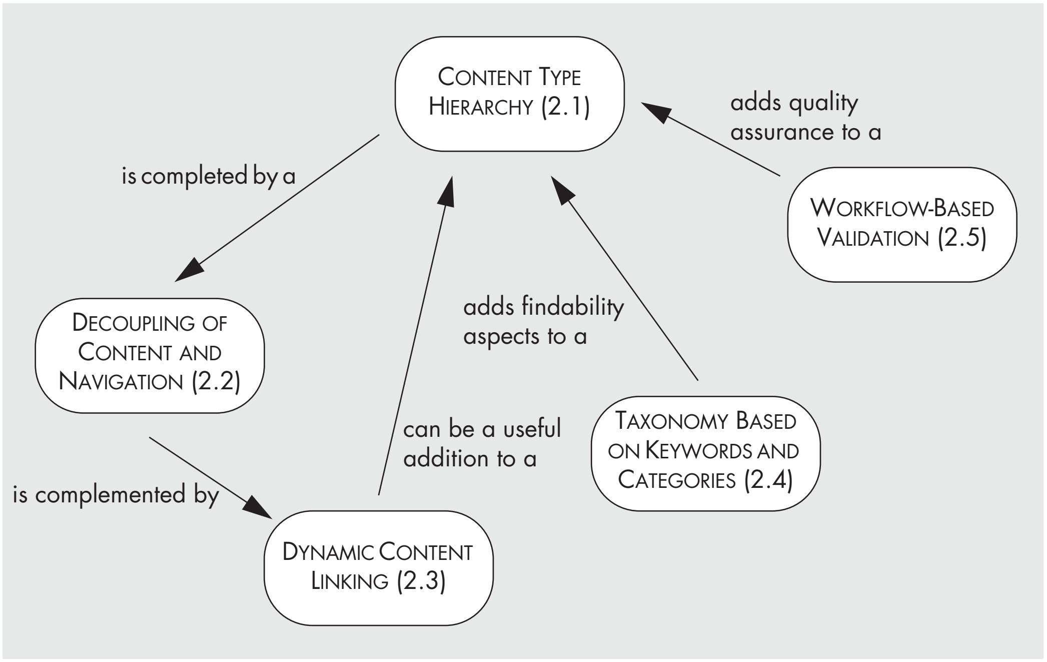

CONTENT TYPE HIERARCHY (2.1)
adds quality assurance to a
is completed by a
WORKFLOW-BASED VALIDATION (2.5)
adds findability aspects to a
DECOUPLING OF CONTENT AND NAVIGATION (2.2)
can be a useful addition to a
TAXONOMY BASED ON KEYWORDS AND CATEGORIES (2.4)
is complemented by
DYNAMIC CONTENT LINKING (2.3)

Figure 12: Road map to the patterns for content management

However, we won't be looking at the business side of content modelling. I won't be talking about how to align content models with business strategies – questions like these are outside the scope of this book. For further information on that topic, refer to books such as *Information Architecture* by Louis Rosenfeld and Peter Morville (Rosenfeld Morville 2006) and *Content Management for Dynamic Web Delivery* by JoAnn Hackos (Hackos 2002).

Again, I'd like to take an approach of piecemeal growth, so the patterns begin with basic modelling techniques and conclude with more specific aspects of findability and content validation. Figure 12 presents a road map.

2.1 Content Type Hierarchy

Context

You plan to develop a website that will present various kinds of content artefacts to its visitors. Content can take many forms, ranging from text to graphics, sound, video and other multimedia objects.

It's now time to specify exactly what content the site is going to use. You have to come up with a content model that specifies what content elements will be stored in the central repository. The content model will form a conceptual basis for all your CONTENT MANAGEMENT AND CONTENT DELIVERY (1.1) software.

Problem

How can you ensure that content editors can maintain artefacts that are meaningful to them in a way that is straightforward and avoids redundant information?

Example

The website for the House of Effects uses different page types. Apart from simple articles (such as those on the welcome page), there will be event announcements, an event calendar, online presentations and shop item descriptions. Many of these will feature pictures or videos.

However, there's not going to be a 1:1 relationship between content and page types. For example, some information found in individual event announcements will also appear in the event calendar, and pictures and videos might be used several times across the site. Nevertheless, we want to avoid a scenario in which content has to be maintained redundantly.

Forces

Your content model must meet several sometimes conflicting requirements.

First, the content model must be meaningful to content editors, as it forms the basis for their editorial work. The content artefacts they will create and maintain will be categorised according to the model that you set up, so your model has an immediate consequence on their job. You must expect content editors to 'think in terms of their domain', whether that is company information, items for an online shop, e-learning materials, e-government or something else. The content model must reflect their understanding of the domain.

Second, the content model has an effect on how content is presented. Content artefacts of the same type are likely to share the same layout. Whether or not two content artefacts should be of the same type depends on how these artefacts should be presented on the web.

Third, the content model must be convincing in terms of data modelling. One of the fundamental ideas of content management is to compose web pages from smaller elements. Your content model has to describe the different types of content elements at various levels of granularity, and it must introduce relationships between content elements that are sufficiently expressive to make page composition possible. The content model should also avoid redundancy, as any good data model should, to avoid inconsistencies and unnecessary maintenance effort.

Solution

Introduce a content model in which content types represent domain-motivated artefacts consisting of basic building blocks such as texts, pictures, multimedia objects and links. Apply object-oriented modelling techniques to express abstraction and association relationships between content types.

Content editors will create, maintain and publish instances of the content types that you define. Web pages will be composed from one or several such elements, applying the same layout to content elements of the same type. It's therefore a good rule of thumb to introduce separate content types for elements that are going to receive clearly different layouts or require different attributes, and to introduce just one type for content elements that will be handled in much the same way.

What attributes does a specific content type require? Each content element needs to be assigned those informational bits and pieces that are necessary to generate its possible views. Attributes typically fall into the following categories:

- Attributes for the content element's constituents, such as formatted and unformatted text, pictures, hyperlinks, multimedia objects, binary large objects (blobs) etc.
- Attributes for keywords or tags to be provided by content editors for categorisation purposes.
- References or lists of references to other artefacts within the content repository.
- Attributes that relate a content element to a possible context. This includes parameters like a locale or a distribution channel (such as an intranet or extranet) that specify the context in which a content element is valid.

- Attributes that specify a layout variation, in case you wish to allow content editors to choose from several variations. If so, keep in mind that simple variations can be fine (like allowing content editors to choose a 1-column or a 2-column layout), but that layout details (such as font sizes, type faces etc.) belong to the template definitions – to ensure a consistent layout across your site, among other things.
- Meta attributes such as the author's name and the publication date. Most content management systems define these meta attributes automatically and assign appropriate values whenever a content element is created, updated or published.

The references between content elements deserve special attention. Your goal should be to use these associations to 'normalise' the content model – much in the sense of normalisation in database design. It's a good strategy to extract attributes that would otherwise be shared by different content types into types of their own and to apply association to express the relationship. In addition, you can introduce abstraction when several content types have attributes in common, meaning that the content model evolves into a content type hierarchy.

Although mainly a conceptual thing, the content model is the basis for almost all aspects of a website architecture. Most content management systems require the configuration of a content model – this is how the content management system gets to 'know' the content types and their attributes. Different systems provide different configuration mechanisms: XML files and database tables are among the most commonly used techniques. Regardless of the system you choose, while the system may provide you with some off-the-shelf content management functionality, you still have to provide the content type model.

Example resolved

The first step towards an effective content model is to identify the necessary content types and their major attributes. We perform a thorough requirements analysis, which involves discussions with domain experts, especially the site owners and the content editors.

This analysis leads to the following list of domain-motivated content types for the House of Effects website:[3]

CONTENT TYPE	DESCRIPTION
article	An article in general, featuring a title, a subtitle, a main text, possibly pictures and videos, as well as a list of related articles. To be used for example for the portal's home page.

[3] In a real-world project there would be more content types and the content types listed here would require more attributes. If you look hard enough, you can probably spot several things that are a little too simple to work in practical cases. But let's not introduce too much detail into our example model, for simplicity's sake.

CONTENT TYPE	DESCRIPTION
event announcement	A special kind of article, announcing a singular event like a talk or a film, featuring an event type, a date, an entrance fee and possibly a list of related items in the online shop.
exhibition announcement	A special kind of article, announcing an exhibition, featuring a start and end date, an entrance fee, and possibly a list of related items in the online shop.
presentation	An online presentation, featuring (in addition to the standard article attributes) a list of animations, as well as additional details.
book description	A special kind of article describing a book that's sold in the online shop; featuring an author, a publisher, a release date, a price, as well as additional details.
DVD description	A special kind of article describing a DVD that's sold in the online shop, featuring a duration, a publisher, a release date, a price, as well as additional details.
details	Text sketching details or background information regarding an online presentation or a shop item.

All content for our site falls into one of these categories. The next step is now to establish a content model, applying data modelling techniques such as normalisation to get rid of possible data redundancy and applying abstraction to model inheritance-like relationships. Figure 13 presents the UML sketch of the resulting content type hierarchy.

A few important aspects should be pointed out. First, we introduce separate content types for event announcements and exhibition announcements, although both types feature similar attributes. This is a trade-off: we could force all announcements into just one type, but we opt for separate types because event announcements and exhibition announcements will later receive quite different layouts. The same is true for the different kind of shop item descriptions (books and DVDs).

Second, we extract pictures, videos and animations into content types of their own and use association to express the relationship. Not only do these objects get some attributes of their own, but the advantages of normalisation materialise as well: pictures, videos and animations are maintained only once and can still be shared across different articles.

Third, moderate use of abstraction has lead to a few inheritance relationships, introducing two abstract types for announcements and item descriptions in general. This allows us to express things like 'all kinds of announcements' without having to list all announcement types explicitly.

Fourth, because all articles can be assigned a list of references to related articles, it is possible for content editors to express arbitrary links between the main content elements. Useful examples include the link from an online presentation to the announcement of a special event, or the link from an exhibition announcement to a DVD in the online shop.

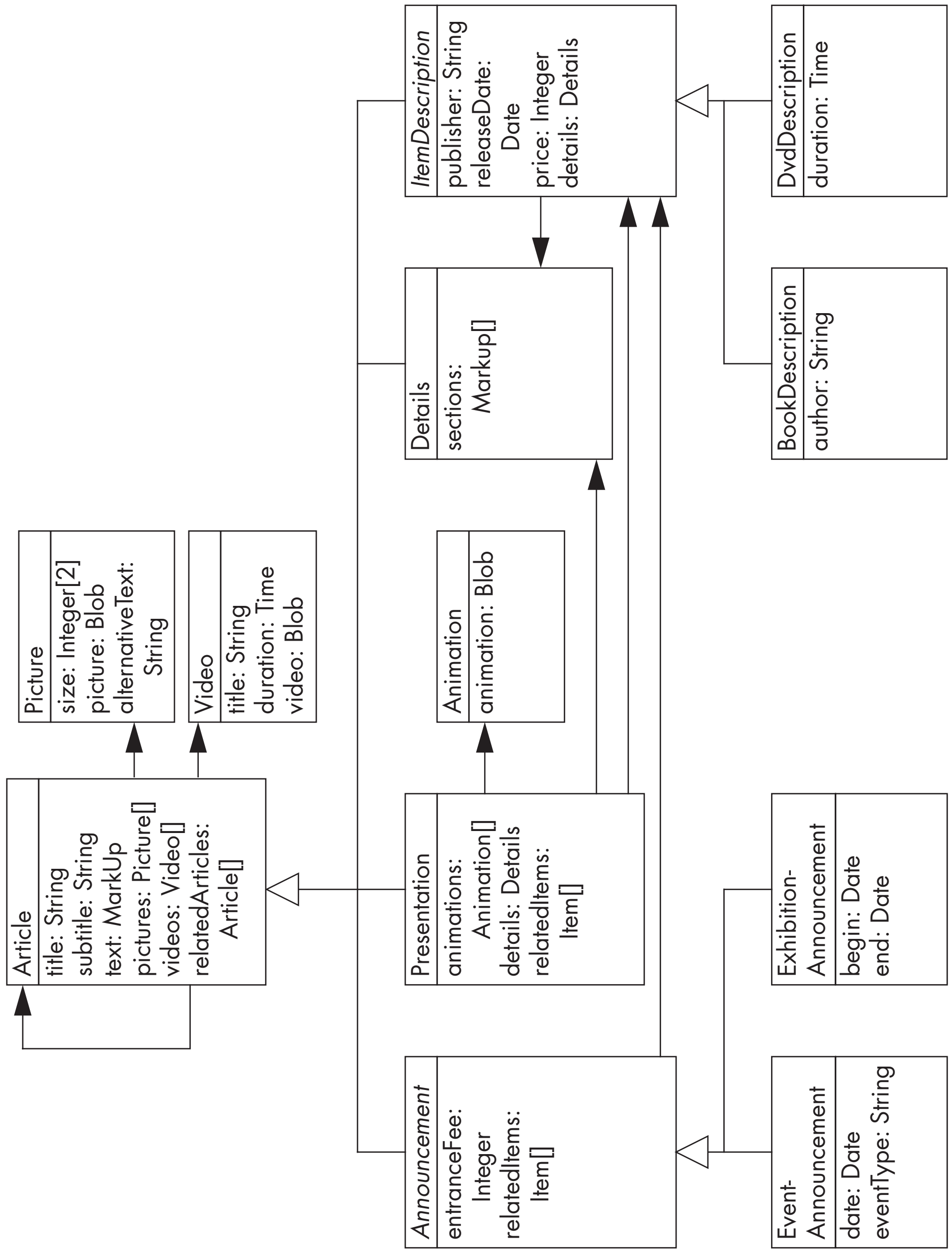


Figure 13: The content type hierarchy for the House of Effects

We configure our content management system so that it 'becomes aware' of our content type hierarchy, the individual content types and their attributes. Let's assume that meta attributes such as the content editor's name and the publication date will be generated automatically, which is why we haven't included them in our UML sketch.

We have now established the basis for content editors to create content elements.

Benefits

+ Because the content model is motivated by domain analysis, it is meaningful to the content editors. The content model will underlie their content maintenance workflow processes and they will be comfortable using it.

+ Because the content model is normalised in the sense of data modelling, you avoid redundant content to a large extent. This makes life easier for the content editors, as they don't have to maintain the same content in different places. Also, inconsistencies are avoided.

+ The 'normalised' content model also saves space in the repository – that is, on the database on which your content management system is based. This is a real benefit when it comes to larger objects such as videos or other multimedia artefacts that might appear in several places on your site.

+ If you introduce abstraction relationships carefully, the content model becomes a well-designed hierarchy. This makes potential classification schemes more powerful. For example, if you choose to implement a search function that is able to return only content of a certain type, the availability of super-types (abstraction of several content types) will make this feature more effective.

+ Although the content model abstracts over concrete layouts, it provides the basis for the consistent use of different layouts across your site. Once you define a SYSTEM OF INTERACTING TEMPLATES (3.4) you can choose to implement a TEMPLATE PER VIEW (3.5) for each content type.

Liabilities

– Experience shows that the number of content types tends to become rather large. The more the domain analysts think about it, the more content types they seem to identify. However, if there are too many, the model becomes complex and incomprehensible. It's a good strategy to bring domain experts and software designers together to develop a well-balanced model – one in which separate content types are introduced only when clearly motivated by distinct attributes and distinct layouts.

– The content model gives you a tentative set of content type definitions, but it's important to understand that more content types may have to be added for technical reasons. The DECOUPLING OF CONTENT AND NAVIGATION (2.2) and the concept of DYNAMIC CONTENT LINKING (2.3) will both require additional content types.

– Not all content management systems support the concept of abstract content types and abstraction (or inheritance) relationships between content types. If your content management system doesn't, you will have to use a 'flat' model instead in which attributes aren't inherited, but are defined separately wherever they are required. Nevertheless, a type hierarchy is still useful on the conceptual level.
– Finally, the content model may have to change as your site evolves. As a consequence, content migration may become necessary if you plan to relaunch your site. Be sure to apply appropriate migration techniques such as staging to achieve a SMOOTH RELAUNCH (5.3).

2.2 Decoupling of Content and Navigation

Context

You have established a CONTENT TYPE HIERARCHY (2.1) that embodies the content model for your website. The hierarchy describes what content types exist and relates them to each other. The next step is to define a navigation hierarchy. You have to provide a means for the content editors to specify how content should be distributed over web pages and how users should be able to travel from one page to another.

Problem

How can you ensure that content editors can organise both the content and the navigation hierarchy in a straightforward and flexible way? How can you support different content hierarchies for different sites, like an intranet and an extranet, or for different countries or distribution channels?

Example

Like many other sites, the website for the House of Effects will be organised mainly in a tree-like navigation hierarchy. As you can see from the various screenshots (Figures 5, 8, 18, etc.), first-level navigation nodes include *Information, Announcements, Presentations* and *Shop*. There will be subnodes whose identity will be established by the editors.

However, things are slightly more complex than this. The website for the House of Effects is supposed to support different languages. For the time being, English and German are sufficient, but a French version is likely to be added in the near future. Flag icons indicate that users can change from one language to the other. Although they cover separate sets of content elements, the navigation hierarchies for the different languages are identical as far as their structure is concerned.

Forces

A website doesn't just make content available, but presents content to visitors as a navigable space. This navigable space allows visitors to travel from one web page to another. It can take many different forms, but is usually organised as a structure that more or less resembles a tree. Everyone's familiar with this – tree-like navigation hierarchies are found all over the web.

The consequence is of course that in addition to maintaining the actual content, content editors must define a navigation hierarchy that fits the site – or more than one hierarchy, as we will see. The specification of navigation hierarchies has to meet some typical requirements.

First, content editors must be able to specify several navigation hierarchies with identical structures. Examples include navigation hierarchies for different languages or navigation hierarchies for different output channels (such as standard web browsers and mobile devices). Although such hierarchies refer to separate sets of content elements, they are structurally identical, or at least parts of them are. Content editors should be able to reuse hierarchies they have already specified, but shouldn't have to maintain identical structures redundantly.

Second, content editors must be able to specify overlapping hierarchies. For example, they may have to specify navigation hierarchies for an intranet and an extranet that overlap where they cover the same content, but differ where content should only appear in one of them.[4] The resulting requirement is that it must be possible to use content elements in more than one hierarchy.

Third, maintenance of the navigation hierarchy or hierarchies should be simple and straightforward. Rearranging a hierarchy shouldn't be too much work for the content editors and, most importantly, shouldn't require the republication of any content elements.

These requirements influence the usefulness of different approaches you can take to represent a navigation hierarchy.

An initial approach is to use content element attributes to express the navigation hierarchy. Attributes could specify the web pages on which a content element should appear. There are important drawbacks to this approach though. Not only does this technique prevent content editors from reusing structurally similar hierarchies, but worse – rearranging a content hierarchy would require that content elements be changed (with regard to their position within the hierarchy) and republished, which would be extremely awkward.

A second approach is to use the organisational structures that the content management system offers. Most such systems allow content editors to organise content in folders and subfolders, and these structures can be employed to represent a navigation hierarchy. This approach avoids the republication of content when a navigation hierarchy changes.

[4] An intranet and an extranet may technically qualify as different sites, as they will probably be reached via different URLs. The argument holds good though, as long as they share the same content base and we can regard them, from a logical point of view, as one site.

However, it would be difficult to represent several navigation hierarchies, and what's more – you would introduce dependencies on your content management system and its organisational principles. The approach is too inflexible to be generally useful.

Solution

Decouple the navigation hierarchy from the content. Introduce dedicated navigation nodes that span the navigation hierarchy. Allow the navigation nodes to be attributed with configuration information if the navigation hierarchy needs to vary across different contexts.

The navigation nodes represent the individual pages of your website. Identified by a unique URL, every node references those content elements that should appear on a page. In many cases this will be just one content element – the one that provides the main content for the page and that itself may reference other content elements. However, pages can represent more than one content element, in which case the navigation node has to maintain several content references.

To be able to specify a content hierarchy, you have to extend the tentative CONTENT TYPE HIERARCHY (2.1) developed so far. The idea is to introduce a content type for navigation nodes – a pseudo type in fact, since navigation nodes represent pages, not content elements. It takes the following attributes:

- Attributes describing the navigation node itself, such as its name (the name that should be used when displaying navigation elements).
- References to subnodes – a node's children in the hierarchy. Often these references span a tree, although this isn't always the case. If it is, you can include an additional attribute for the reference to the unique parent node.
- References to one or several content elements assigned to the navigation node – the actual content that should appear on the page.

If you have to support several navigation hierarchies, you can extend this model in a straightforward way. A navigation node will still be identified by a unique URL, but what content it holds will depend on what is best described as a *request context*. Technically, the navigation node's attributes become mappings that map the request context onto concrete values.

The definition of a request context depends on the flexibility that your site requires. Typical constituents include the following:

- The current language and country. This allows you to specify language-dependant navigation hierarchies.
- The current distribution channel, which for example allows you to specify separate hierarchies for an intranet and an extranet, or for different output devices.

A request context can be determined in different ways. Typical examples include the evaluation of the URL parameters, the HTTP request parameters or information stored in the user's session. Based on parameters such as these, you can set up a highly configurable navigation hierarchy.

Example resolved

The UML diagram in Figure 14 shows the additions we have to make to the content model for the House of Effects website. We introduce *NavigationNode* as a new type and establish the necessary associations to *Article* – the content type for articles in general that we have introduced in the UML diagram back in Figure 13.

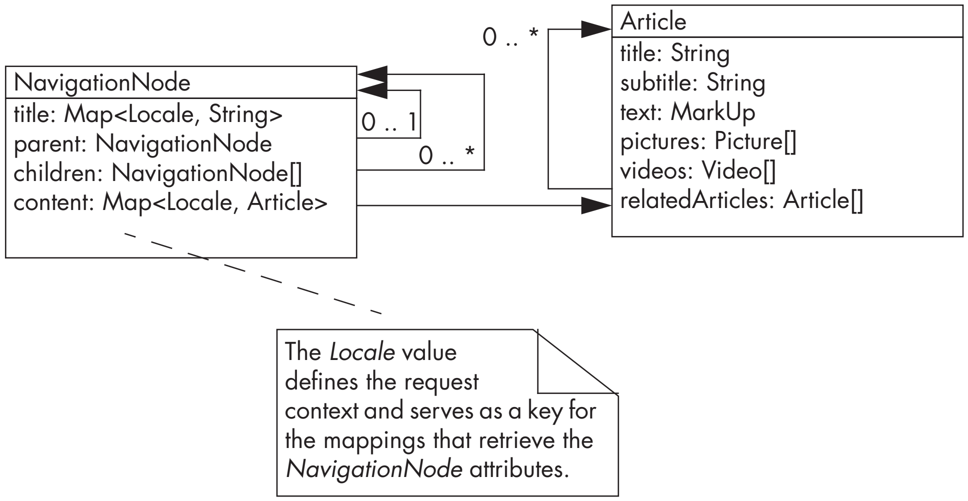

NavigationNode
title: Map<Locale, String>
parent: NavigationNode
children: NavigationNode[]
content: Map<Locale, Article>
0 .. 1
0 .. *
0 .. *
Article
title: String
subtitle: String
text: MarkUp
pictures: Picture[]
videos: Video[]
relatedArticles: Article[]
The Locale value defines the request context and serves as a key for the mappings that retrieve the NavigationNode attributes.

Figure 14: Configurable navigation nodes for the House of Effects

In our example, the request context only consists of the current locale. We'll add the locale to all our URLs as a URL parameter. This allows us to establish two separate navigation hierarchies for an English and a German version of our site. This is all we need to do, at least for the time being, as there are no plans for other variations of the navigation hierarchy (such as intranet versus extranet).

The attributes of *NavigationNode* can therefore depend on the current locale, although they don't have to. First, there is the navigation node's title. It does depend on the locale, as there are both English and German versions. Second, there are references to the navigation node's parent and children. These references are independent of the locale, spanning a navigation hierarchy that is used consistently for English and German.

Finally, there is a reference to the actual content – the article that should appear on the page which the navigation node represents. This attribute, too, depends on the locale, as different content will be used for different languages. Once the locale is known, the correct article for the page can be obtained, as well as pictures, videos etc. that are associated with that article.

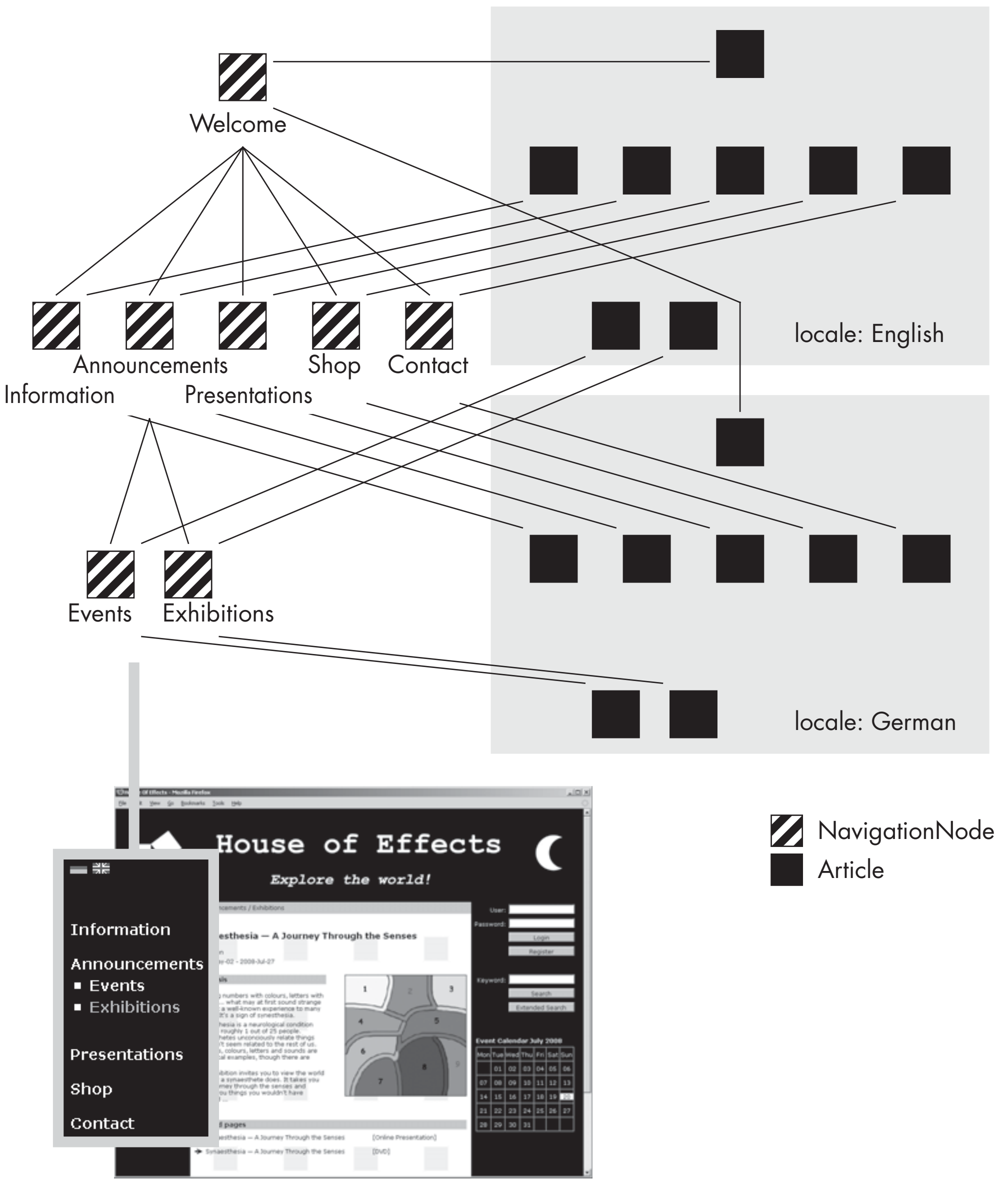


Figure 15: Navigation nodes and content elements for a specific locale

Figure 15 illustrates how two distinct sets of articles are referenced from one navigation hierarchy.

Benefits

+ Changes to the navigation hierarchy don't result in any updates to, and republication of, the actual content. The real content elements (all except the pseudo elements that represent navigation nodes) are completely unaffected by any changes to the navigation hierarchy. Whether you rearrange the navigation hierarchy or introduce additional variations of it, only navigation nodes have to be updated and published, which keeps the editorial workflow convenient and straightforward.

+ The mapping from navigation nodes onto content elements is extremely flexible. It gives you the freedom to support navigation hierarchies for different languages, different audiences (intranet versus extranet, for example), different distribution channels and the like. Navigation nodes can depend on the request context, so content editors are free to configure navigation hierarchies as they see fit, reusing those parts that several hierarchies may have in common. The editorial workflow is improved, as the definition of variable navigation hierarchies becomes significantly easier.

+ The navigation hierarchy is completely independent of how content is organised in the repository. It's probably a good idea to establish repository structures that largely mirror the navigation hierarchy, but ultimately content editors are free to organise the content as they like.

+ The solution also paves the way for a personalised site. Once you implement CONTENT FILTERS (4.1) to tailor the site to individual users, you can use the navigation nodes to configure which parts of the navigation hierarchy should be visible to which user groups.

Liabilities

– Whenever a page element requires any navigation-related information, the hierarchy spanned by the navigation nodes has to be evaluated. This isn't a serious problem, but it may require substantial computation: the request context has to be determined and references to subnodes have to be followed. To avoid scattering navigation-related code all over your custom components, you should implement a NAVIGATION MANAGER (3.2) that encapsulates all navigation-related computation.

– When a new page is added to the site, content editors have to create two elements in the repository, one for the navigation node and one for the actual content. This is the price you have to pay for the flexibility you get from decoupling content and navigation. However, there may be cases when you don't really need this flexibility.

In these cases you can apply DYNAMIC CONTENT LINKING (2.3) to make content maintenance easier.

– The solution slightly depends on your content management system's ability to support non-trivial content models. Your content model will include mappings, at least for some of the attributes for navigation nodes. Check your content management system to make sure there is a convenient way to implement this pattern.

2.3 Dynamic Content Linking

Context

You have established a CONTENT TYPE HIERARCHY (2.1) and you've made the necessary extensions to express a navigation hierarchy as well, implementing the DECOUPLING OF CONTENT AND NAVIGATION (2.2) principle. Content editors can now create content artefacts of different types and refer to these artefacts from potentially configurable navigation hierarchies.

Problem

How can frequently changing content be maintained without burdening the content editors with the tedious job of manually linking the new content into the navigation hierarchy?

Example

Some of the content for the House of Effects website is stable, some changes frequently. For example, presentations are relatively stable. New presentations are added occasionally, but existing ones are kept for long periods. When a new presentation is added, content editors decide their location within the subhierarchy below *Presentations,* perhaps adding a subtree to locate it.

On the other hand, announcements change frequently. A new announcement is added to one of the two lists that appear under the menu entries *Events* and *Exhibitions*. Similarly, new shop item descriptions are added on a regular basis. They always appear under the *Shop* menu. None of these cases require elaborate additions to the navigation hierarchy.

Forces

A principle liability caused by DECOUPLING OF CONTENT AND NAVIGATION (2.2) is that the addition of a new page requires both a new navigation node and one or more new content elements. The advantage of this approach is the flexibility that content editors gain for the definition of the navigation hierarchy. But what if this flexibility isn't needed?

Think of material that is best organised as a list of items. Regardless of what such a list looks like in detail, regardless of whether or not it should be sorted, what all lists have in common is a linear structure. If all you need is a list of content elements, the flexibility that you gain from a highly configurable navigation hierarchy doesn't add any benefit.

Worse yet, it's often the content that takes the form of list items that undergoes frequent changes. This makes the situation doubly unsatisfactory. First, the effort required to create the content and to link it into dedicated navigation nodes doesn't seem justified, as there's no advantage to a highly flexible navigation hierarchy where this hierarchy is actually flat. Second, this scenario could occur often, causing the content editors' job to become unnecessarily tedious.

Solution

Establish dynamic lists that, instead of maintaining links to any content elements, specify criteria for potential list items, meaning that when a page featuring a dynamic list is generated, the repository is searched for content elements that match these criteria.

Dynamic lists are best implemented with another pseudo content type that you add to the CONTENT TYPE HIERARCHY (2.1) developed so far. This pseudo content type requires a number of attributes that the content editors must supply to specify what items the list should include. These attributes fall into the following categories:

- First, potential list items have to be identified. You can specify a query, which will typically cover a repository path, a name pattern and perhaps a content type. Content elements found in the specified repository folder qualify as list items if their names and types match the specified criteria.
- Next, the list itself can have properties. A typical example is the default order that should be applied to the list items.

There are two ways in which dynamic lists can be used. A dynamic list can represent the main content of a page, in which case it is directly referenced by a navigation node. Alternatively, a dynamic list could be embedded into manually edited content, meaning that it would be referenced by real content elements.

Example resolved

There are two places in the content model for the House of Effects website where dynamic lists make sense. One is the lists of announcements (for special events and exhibitions), the other is the list of shop items offered by the online shop.

We introduce a new subtype of *Article* to hold these kinds of lists. Named *Overview*, this special article is different from a standard article only in that it maintains an association to a dynamic list whose list items are, for example, announcements for upcoming events. Since these *Announcement* elements won't be referenced from any dedicated *NavigationNode* element, no menu items will be created for them. Nonetheless, all announcements will be referenced from the overview page.

Figure 16 shows the UML sketch for the necessary additions to the CONTENT TYPE HIERARCHY (2.1).

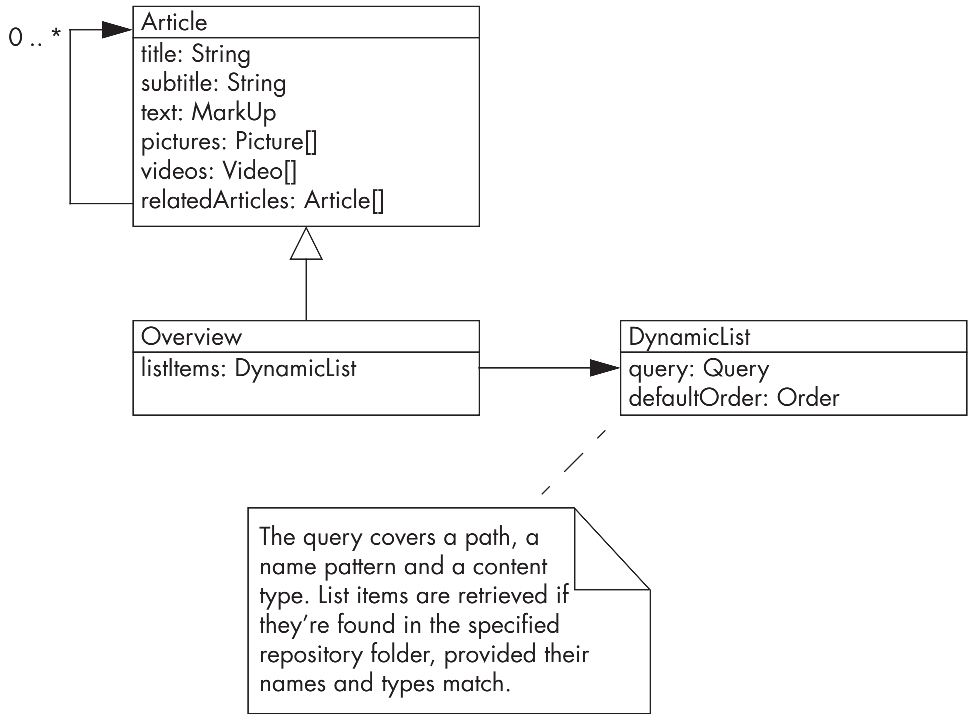

0 .. *
Article
title: String
subtitle: String
text: MarkUp
pictures: Picture[]
videos: Video[]
relatedArticles: Article[]
Overview
listItems: DynamicList
DynamicList
query: Query
defaultOrder: Order
The query covers a path, a name pattern and a content type. List items are retrieved if they're found in the specified repository folder, provided their names and types match.

Figure 16: Specification of a dynamic list

Benefits

+ Dynamic lists represent a quick and easy way to add content to a website, provided that it's sufficient to present the content elements in the form of a list and that no real navigation hierarchy is required. Creating content elements and storing them in the specified repository folder is all the content editors have to do when they wish to add content – no navigation nodes have to be defined. Removing content elements after they have expired is equally simple, so content maintenance becomes more straightforward.
+ Dynamic lists also represent a hook for personalisation. You can apply CONTENT FILTERS (4.1) to the lookup functions that search for list items and so tailor dynamic lists to specific users or user groups.

Liabilities

– Dynamic lists can become quite long when more and more content elements are placed in the repository folder from which list items are taken. However, if there is no limit to the number of list items, performance problems can be a consequence, both when content is retrieved from the repository and when it is delivered to the web. It is the responsibility of CONTENT SERVICES (3.1) to return content portions of reasonable size and to apply pagination to dynamic lists if necessary.

– List items aren't referenced from any navigation node and so aren't embedded into the navigation hierarchy. In a way they are 'orphaned' content elements. It is the responsibility of a NAVIGATION MANAGER (3.2) to provide a default navigation context for them.

– The implementation of dynamic lists depends to some extent on your content management system. The solution assumes that it's possible to use a query to identify content elements in the repository. The query will consist of things like a name pattern and a path, but may not include the unique id that a content management system typically assigns. Most content management systems provide such a lookup mechanism, but if your system doesn't you may have to develop it yourself.

– Dynamic lists are affected by reorganisation of content within the repository. Content editors must be aware that moving list items from one folder to another can have unwanted effects on the content that is delivered to the web.

2.4 Taxonomy Based on Keywords and Categories

Context

You have designed a CONTENT TYPE HIERARCHY (2.1) for a website that you plan to set up. In addition to the navigation mechanisms, you plan to offer a search function that will help users find the information they are looking for.

Problem

How can you lay the foundations for an effective and powerful search function?

Example

It quickly becomes clear that a mere full-text search won't be sufficient for the House of Effects website. The reason is quite simple – words people are likely to choose as search terms don't always appear verbatim within a content element's textual attributes, even if that content element is indeed a good search result. For example, visitors might search for online presentations on astronomy using the word 'astronomy' as a search term. However, a mere full-text search wouldn't return an online presentation on solar eclipses unless that presentation featured the word 'astronomy' in its title or its description. Full-text search alone isn't powerful enough.

Sometimes, however, full-text search isn't restrictive enough. What if a visitor is looking for online presentations, but not for shop items? How could this be expressed? If all the site offered was a full-text search, the necessary distinction could not be made.

Forces

With more and more content populating websites world-wide, findability has become increasingly important. Peter Morville, in the subtitle to *Ambient Findability*, even states that 'what we find changes what we become' (Morville 2005). If a website wants to reach its target audience, it had better ensure that potential visitors can successfully and effectively find the content they're looking for.

Of course, findability across the web is not an easy matter. Achieving ambient findability is one of the goals behind the 'Semantic Web' (Berners-Lee Hendler Lassila 2001) and is the subject of current research. The idea is to develop models and ontologies that bring structure to the meaningful content of web pages, so that information is related in an effective way. However, some are sceptical over the chances for success, as the Semantic Web relies heavily on metadata, and metadata is intrinsically difficult to organise in an 'open space' such as the web (Morville 2005).

Achieving findability is less problematic in the confined space of a single website. A single website has – or at least should have – a consistent content model. Based on this content model, you can implement a search function designed specifically to meet the findability requirements for the site.

It is likely that in many cases a full-text search alone won't be sufficient – the limits to its usefulness are all too evident. Full-text search suffers from a trade-off between *recall* and *precision*. Recall describes the percentage of relevant information that a search function is able to find, while precision describes the percentage of relevant information among all search results. Several studies have been performed that investigate how precision and recall relate to each other. The consensus is that you can only improve one at the expense of the other: if you make a full-text search less restrictive, it will return more relevant results, but the number of insignificant results increases as well, and vice versa (Blair Maron 1985). Values for recall and precision that add up to 100% are considered to be fairly typical.

As Peter Morville describes in *Ambient Findability*, the introduction of meta information can significantly improve the situation (Morville 2005). A first option is to let content editors assign descriptive meta attributes to content elements. Well-chosen keywords can improve both recall and precision, especially when thesauri and fuzzy search algorithms are brought in to handle spelling variations and synonyms.

However, to increase precision significantly, you have to find a way to discard content in which users are *not* interested. Assigning descriptive meta attributes alone may not do the trick. Instead, it may be helpful to allow users to select specific content categories and deselect others when they submit a search request.

Solution

Build a taxonomy that combines arbitrary keywords with well-defined content categories. Make sure that individual content elements can be attributed with lists of keywords. In addition, establish a set of possibly overlapping content categories to which individual content elements can be assigned.

Keywords and categories represent different concepts:

- Keywords serve to *describe* content elements. They are essentially tags that are assigned to content elements. It's fine to allow arbitrary terms that describe a content artefact well – there is no limit to the set of keywords that can be used.
- Content categories serve to *classify* content elements. The set of categories is always limited – each content element will have to belong to at least one. There are different ways in which you can identify useful categories. You can specify a list of categories motivated by the application domain, you can use the list of content types, or you could use a combination of both, making the categorisation even more powerful. Although it may be unusual, it is generally acceptable for categories to overlap, meaning that content elements may belong to more than just one category.

Keywords and categories complement each other: it is the combination of both that allows you to define an effective taxonomy.

The introduction of keywords and content categories makes a few extensions to the CONTENT TYPE HIERARCHY (2.1) necessary. In most cases you will need a few additional attributes that allow you to store keywords and categories along with the content elements. The one exception is the content type – content elements are typed anyway, so using the content type as a category doesn't require an addition to the model.

Keywords and categories will be assigned to content elements when these content elements are created or updated, so you are now ready to establish a mechanism that notifies the search engine of these keywords and categories. Search engines essentially consist of two components: the indexer and the query engine. Right now we're only concerned with the indexer, as this is the component that needs to be notified of any changes to the content. You can apply LISTENER-BASED SYNCHRONISATION (1.4) to guarantee that the indexer is informed of content artefacts, including their keywords and categories.

If you plan to use your content management system's search engine, configuring the built-in listener is probably all you need to do. If you plan to use a stand-alone search engine, you may have to develop your own custom listener. In either case, ensure that the indexer is notified of the keywords and categories that you have added to your model.

Example resolved

The first addition we make to the content model for the House of Effects website is to assign a list of keywords to all articles. These keywords will be provided by the content editors when they create an article. They can be changed in the course of content maintenance.

The second addition lies in the introduction of domain-motivated content categories. We use the four categories *Mathematics*, *Physics*, *Chemistry* and *Biology*, and allow these categories to overlap. Content editors can assign not just one category, but a list of categories to all articles.

These additions are reflected by additional attributes for the content type *Article*. Figure 17 shows the relevant excerpt from the resulting UML sketch of our CONTENT TYPE HIERARCHY (2.1).

Moreover, we want to extend the categorisation, and also use the content type (such as *Presentation*, *Announcement* or *ItemDescription*) as a classification scheme. This gives us two orthogonal sets of content categories – *Mathematics*, *Physics*, *Chemistry* and *Biology* on one hand, and the different content types on the other. Since the content type is available anyway, no further additions to our CONTENT TYPE HIERARCHY (2.1) become necessary.

Figure 18 gives an example of the kind of search request that we can now plan to make possible.

Users will be able to enter a search term, but unlike a mere full-text search, an article will be found not only if the search term occurs in its text, but also if the search term

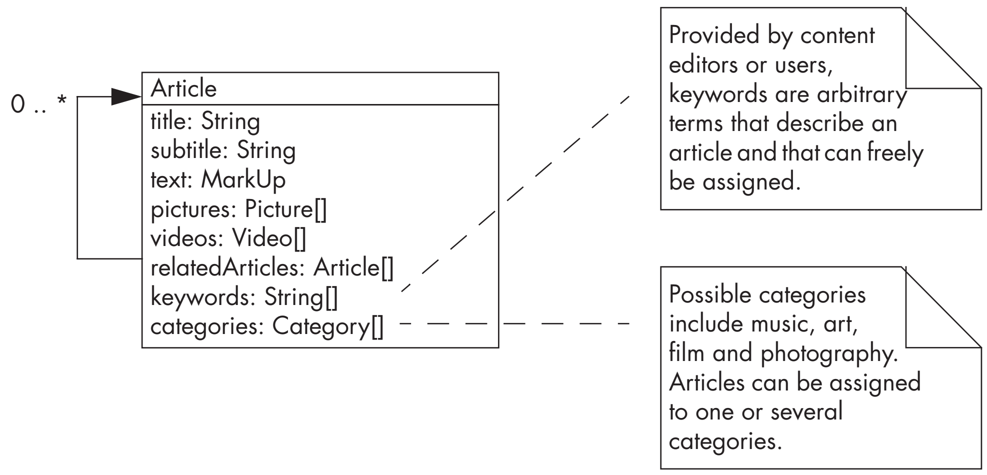

0 .. *
Article
title: String
subtitle: String
text: MarkUp
pictures: Picture[]
videos: Video[]
relatedArticles: Article[]
keywords: String[]
categories: Category[]
Provided by content editors or users, keywords are arbitrary terms that describe an article and that can freely be assigned.
Possible categories include music, art, film and photography. Articles can be assigned to one or several categories.

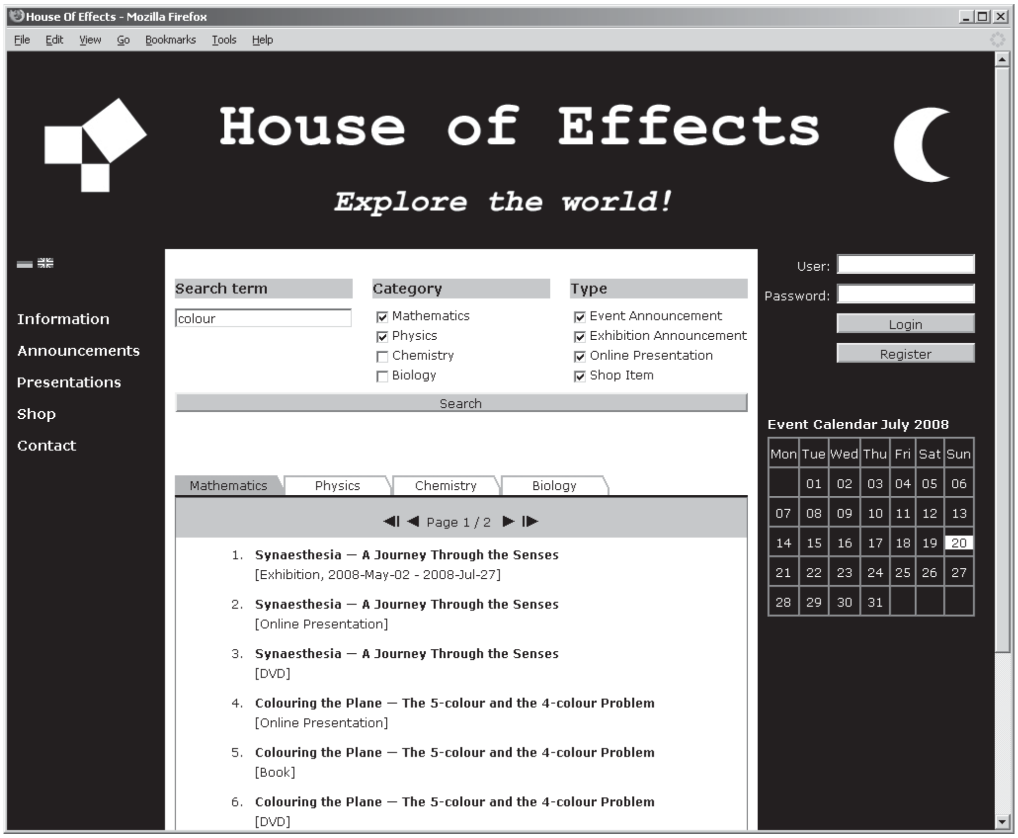
Figure 17: Addition of keywords and categories as metadata

matches the (hopefully well-chosen) keywords. As you can see in Figure 18, a search request for ‘colour’ should find the announcement of a special exhibition on synaesthesia – a neurological condition that causes people to associate different sensory modalities, for example numbers and colours – even if the word ‘colour’ doesn’t appear in the actual announcement text. The precondition for this is of course that the content editors have chosen to use ‘colour’ as a keyword.

Moreover, users will be able to select or deselect content categories and content types. This will give them the freedom to express more complex search requests, such as the search for biology presentations or the search for shop items categorised under either physics or chemistry.

With the additions to the content model we’ve just made, we have laid the foundations for a powerful content taxonomy, but obviously there a few more things we have to do. We plan to use an external search engine, so we have to configure it to implement the kind of taxonomy-based search function that we wish to make available. At least we’ll have to configure the search engine to make it aware of the keywords and categories that its indexing algorithms should take into account in addition to the full-text attributes.

We must also implement a mechanism that actually provides the search engine with all significant information. We will develop a dedicated repository listener that will notify the search engine of any relevant changes to content elements or their metadata, which of course includes the keywords and categories. This way, LISTENER-BASED SYNCHRONISATION (1.4) will ensure that our search engine is able to implement a search function based on the taxonomy we have defined.



Search term: colour

Category: [✓ Mathematics] [✓ Physics] Chemistry Biology

Type: [✓ Event Announcement] [✓ Exhibition Announcement] [✓ Online Presentation] [✓ Shop Item]

Mathematics | Physics | Chemistry | Biology

Page 1 / 2

1. **Synaesthesia — A Journey Through the Senses**
   [Exhibition, 2008-May-02 - 2008-Jul-27]
2. **Synaesthesia — A Journey Through the Senses**
   [Online Presentation]
3. **Synaesthesia — A Journey Through the Senses**
   [DVD]
4. **Colouring the Plane — The 5-colour and the 4-colour Problem**
   [Online Presentation]
5. **Colouring the Plane — The 5-colour and the 4-colour Problem**
   [Book]
6. **Colouring the Plane — The 5-colour and the 4-colour Problem**
   [DVD]

Event Calendar July 2008

Mon	Tue	Wed	Thu	Fri	Sat	Sun
	01	02	03	04	05	06
07	08	09	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Figure 18: The House of Effects's search page using categories and document types

Benefits

+ You have laid the foundation for a search function with improved *recall*. Recall is improved mainly through the definition of keywords. The search function can return a content element whenever a keyword matches the search term, so it is no longer restricted to the unsatisfactory results of a full-text search.
+ You have also laid the foundation for a search function with improved *precision*. Since categories are well-defined and their number is limited, you can implement a search function in which users can actively deselect those categories that are irrelevant to them. The results that are returned therefore tend to be more significant.

+ When you generate web pages from content elements, you can use the keywords and categories to add meta attributes to the HTML you create. Search engines on the web may evaluate these meta attributes. Findability on the web, though not the primary goal of this pattern, may be improved as well.

Liabilities

– Because arbitrary keywords are possible, undesired mismatches between search terms and keywords can occur. You may consider implementing fuzzy search techniques to tackle this problem on different levels. On a syntactical level, applying the Levenshtein distance to the matching algorithm can handle spelling variations and the like (Gusfield 1997). On a semantic level, a thesaurus allows you to identify matches in the presence of synonyms.

– The reorganisation of categories can be expensive. It is likely that the custom code for your search function will rely on existing categories one way or another (for example to generate a search page, as in Figure 18). Of course it would be unwise to hard-code any categories, but adding or removing categories may still result in code or configuration changes.

– Storing keywords and categories together with the content elements is one thing, while using this information for a powerful search function is quite another. What you have done so far is to build a useful taxonomy and notify the search engine of it. Querying the search engine and handling its results is a task that still needs to be done. You can implement a SEARCH MANAGER (3.3) – a component that encapsulates this functionality.

2.5 Workflow-Based Validation

Context

You have completed the content model for your site. Your CONTENT TYPE HIERARCHY (2.1) specifies the various content types, their relations and their attributes, including those attributes that are required to set up a TAXONOMY BASED ON KEYWORDS AND CATEGORIES (2.4). The content model may include a pseudo type for navigation nodes (caused by the DECOUPLING OF CONTENT AND NAVIGATION (2.2)) and a pseudo type for dynamic lists (motivated by the need for DYNAMIC CONTENT LINKING (2.3)).

Problem

How can you avoid content elements with illegal or inconsistent attribute values?

Example

The CONTENT TYPE HIERARCHY (2.1) for the House of Effects website (as in Figures 13, 14, 16 and 17) specifies content types, their attributes and their relationships. However, so far we haven't specified any constraints.

Such constraints of course exist. Some attributes are mandatory and must not be empty, markup has to be well-formed, date values have to be valid, pictures and videos have to use legal formats, references to other content elements must not be dangling and the navigation hierarchy must not contain circular links. It is the content editors' job to be aware of these constraints and to ensure that no constraints are violated, but we would still appreciate it if our system could offer support.

Forces

Content can meet all kinds of constraints. Constraints can apply to elementary content elements or to larger artefacts. Constraints can address single content elements or relate different ones. Constraints ensure things like plausibility, validity, completeness and consistency.

If constraints are violated, the appearance of a website suffers. Incomplete web pages, unusable links, worthless multimedia objects or flawed layout can be the consequence. It's only natural that most sites are faced with the requirement that important constraints should be tested automatically during the workflow process.

Most content management systems offer support, at least to some extent. Checking for mandatory attributes and for valid links is something that many systems are able to do – all that's required is the necessary configuration. Other constraints require custom software: if you want to ensure specific domain-driven constraints for your content, you will probably have to develop the routines that check these constraints.

However, there can be too much of a good thing. If you are too restrictive about certain constraints, workflow processes may suffer. For example, content editors should be free to handle work in progress in which specific plausibility requirements may not be met: it makes no sense to enforce all constraints at all times. Once content is being published and delivered to the web, things look different. At this point existing constraints can no longer be ignored and need to be enforced.

Solution

Apply validators that check content for plausibility. Integrate these validators into the content editors' workflow in two ways: validators that reject illegal attribute values should be applied during the editing process, while validators that check for completeness and consistency should only be applied on publication.

In principle all content types can be subject to validation, including the pseudo types for navigation nodes and dynamic lists. The first thing you have to do is to decide on the necessary validators for each content type and when those validators should be applied.

The following list gives a few typical examples of validators that check for legal attribute values and should therefore be applied during the editing process:

- Check that markup is well-formed and valid.
- Check that date values are correct and plausible.
- Check that e-mail addresses and URLs are well-formed.
- Check that blob objects use a legal file formats and have an acceptable file size.
- Check that referenced content elements are of the correct type.

Checks for completeness and consistency should be deferred until publication. Here are a few examples of typical validators from this category:

- Check that mandatory attributes aren't empty.
- Check that no URL or path expression specifies a dangling link.
- Check that no circular links occur in the navigation hierarchy.

A very sophisticated workflow process might require more than just two classes of validators (for example to differentiate between validation on editing, reviewing and publication). In most cases, though, the two classes of validators described above are adequate. This typical model is described in the UML state diagram given in Figure 19.

Once you have decided what validators are necessary, you have to implement those that your content management system doesn't supply. The more common validators, such as checks for mandatory elements, or simple formatting checks, are sometimes readily available. Also, almost all content management systems check that no content element is deleted while it is still referenced from other content elements, an idea similar to the *referential integrity* concept from database design.

However, non-standard validators usually require some custom software. Most content management systems provide an interface for implementing and registering custom validators. You can use this interface to integrate the validators you need.

Example resolved

Let's assume that our content management system offers a few basic validation mechanisms, including checks for mandatory attributes and checks for correctly typed

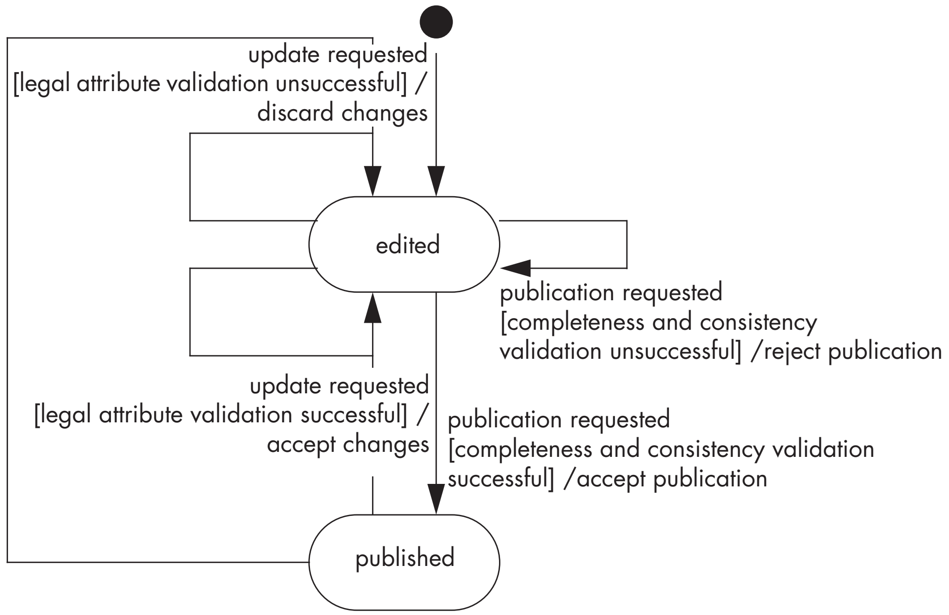

update requested
[legal attribute validation unsuccessful] /
discard changes
edited
publication requested
[completeness and consistency
validation unsuccessful] /reject publication
update requested
[legal attribute validation successful] /
accept changes
publication requested
[completeness and consistency validation
successful] /accept publication
published

Figure 19: State diagram for the content editors' workflow

references between content elements. Let's also assume that our content management system offers an interface for registering arbitrary custom validators.

We can now set up the list of validators we need, and split up the overall list into two lists, for validation during content editing and validation on publication.[5]

Let's look at the validation during the content editing process first. We'll use the built-in type-checking validator which, for example, checks that the attribute of *Presentation* that is supposed to point to the presentation's details indeed holds a reference to a *Details* object, but not to any other. We'll also use the following custom validators:

CONTENT TYPE	ATTRIBUTE	VALIDATION
Article	text	well-formed markup
EventAnnouncement	date	valid date
ExhibitionAnnouncement	begin	valid date
ExhibitionAnnouncement	end	valid date
ItemDescription	releaseDate	valid date

[5] We only list validators for the most important constraints. You could of course think of more validators, but as before, let's keep the example simple.

CONTENT TYPE	ATTRIBUTE	VALIDATION
DvdDescription	duration	valid time value
Details	sections	well-formed markup
Picture	picture	legal picture format, file size must not exceed a specified maximum
Video	video	legal video format, file size must not exceed a specified maximum
Animation	animation	legal Flash file, file size must not exceed a specified maximum
DynamicList	defaultOrder	legal order specification

Next, validation on publication includes the built-in checks that no mandatory attributes are left empty, as well as the following custom validators for consistency and completeness:

CONTENT TYPE	ATTRIBUTE	VALIDATION
ExhibitionAnnouncement	begin, end	*begin* earlier than *end*
NavigationNode	children	must not cause circular links
DynamicList	query	valid query

Most of the custom validators are quite simple and can be implemented easily. The most challenging implementation is the one that checks the navigation hierarchy for unwanted circular links. Registering our custom validators with the content management system, we can ensure that our content will meet the domain-driven constraints that are placed on it.

Benefits

+ Invalid, inconsistent and incomplete content is rejected during the editing process. Content editors benefit from the feedback the system gives them. As the underlying content is accurate, complete and consistent, your website gains authority and reputation.
+ Content editors retain the freedom to work on the content in any way that they see fit. Since completeness and consistency aren't checked until publication, content editors can store their work in progress in the repository without being bothered by plausibility checks that are too restrictive.

Liabilities

- The solution depends on your content management system. To make workflow-based validation possible, the system must be flexible enough to integrate validators for arbitrary content types at different points in the workflow process.
- Validation is useful, but it doesn't make robust content delivery software unnecessary. When you design the custom software for content delivery for your site, you must ensure that it can deal with content that hasn't been properly validated. First, this software could be used in the context of a preview function. A preview function has to work reasonably well with content that hasn't been validated. Second, if for some odd reason non-validated content makes it to the 'published' state unintentionally, your site should still be able to work. Robustness is essential – a missing validation can never be the excuse for severe errors in the content delivery process.

CHAPTER

3

Content Delivery

While the previous chapter was concerned with how content is managed and organised, this chapter is devoted to how it is brought to the web. We'll look at patterns for the server-side architecture for content delivery. We'll study what components are typically included in such an architecture and what the underlying principles are.

In many respects software for content delivery is similar to any other web application, so a variety of patterns apply that are targeted at web applications in general. The context of this book is more specific though, as we're not looking at web applications in general, but focusing on the kind of web applications that are necessary to make content available on the Internet. This chapter therefore deals with issues such as content retrieval, navigation and search functionality.

Before we start with the actual patterns, I'd like to revisit a fundamental principle of application development, as it's going to set the stage for the patterns to come. The *Model-View-Controller* pattern, well-known from the literature, divides an interactive application into three constituents. The model represents the data, the view implements possible visualisations and the controller handles user requests and interaction. The pattern recommends separating all model aspects from all view aspects and from all controller aspects. Due to a reduced coupling, there will be fewer dependencies and the system's

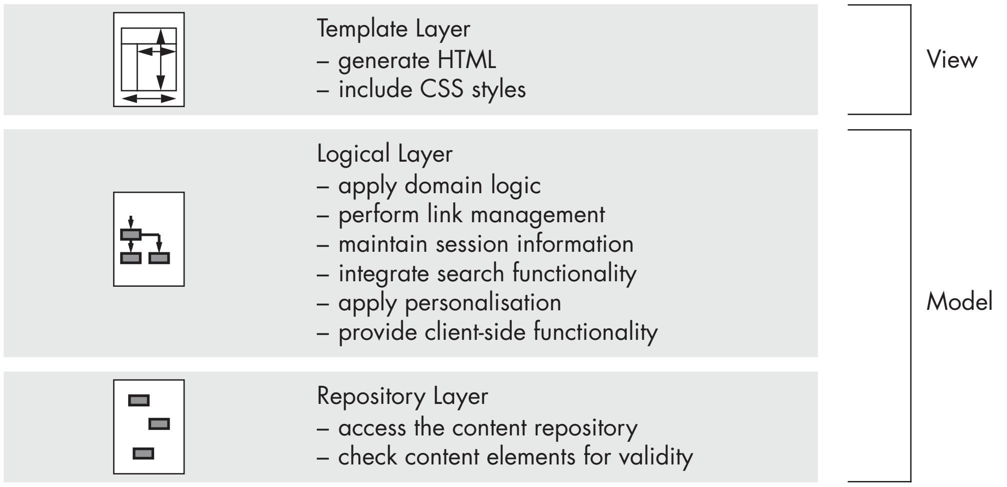

Template Layer
– generate HTML
– include CSS styles
Logical Layer
– apply domain logic
– perform link management
– maintain session information
– integrate search functionality
– apply personalisation
– provide client-side functionality
Repository Layer
– access the content repository
– check content elements for validity
View
Model

Figure 20: Layered architecture for dynamic content delivery

comprehensibility and maintainability are improved (Buschmann Meunier Rohnert Sommerlad Stal 1996, Fowler 2003).

As Martin Fowler points out in *Patterns of Enterprise Application Architecture*, the *Model-View-Controller* pattern is particularly important for web applications (Fowler 2003). In fact it implements two distinct principles. The first is the separation of the model and the view, which is regarded as 'one of the most fundamental heuristics of good software design'. The second is the separation of the view and the controller, which serves to make the control flow through a website independent of its presentation. Examples can be found in web frameworks such as Struts (struts.apache.org), Spring (www.springframework.org) or Ruby on Rails (www.rails.org).

We have already come across the separation of model and view in this book. You can sense this separation in the three layers that constitute the LAYERED ARCHITECTURE FOR CONTENT DELIVERY (1.5). While the logical layer and the repository layer implement the model independently of all presentation aspects, it is the template layer that implements the view. Figure 20 summarises what we already know from the first chapter.

It's now time for us to zoom into this picture and study things in a little more detail. Applying the *Model-View-Controller* pattern to content delivery, we come to the following sequence of actions that is performed after a request has been received from the browser:

1. The request is handled by a controller that analyses the request and invokes all necessary actions. Many architectures define what is often referred to as a *front controller* (Fowler 2003) – a singleton object that acts as a central dispatcher for all kinds of requests, whether they're caused by following an HTTP link or by submitting an HTML form.

2. The controller calls components in the logical layer that are in charge of providing the necessary content. These components usually rely on repository layer functionality that retrieves content artefacts from persistent storage. The components on the logical layer add domain logic: they perform link management, make search results available and apply personalisation. They collaborate with external components and depend on the user's HTTP session to maintain data across HTTP requests.
3. The controller also invokes the templates that generate output for the web. Templates rely on the content that is provided by the logical layer, and may include Ajax functionality in the HTML they generate. Their output is sent to the browser as an HTTP response.

This, however, is just the server-side request handling. If you decide to include SENSIBLE CLIENT-SIDE INTERACTION (1.3), you will also require client-side request handling – request handling that happens directly in the browser – which means that you'll have to provide specific Ajax components.

Fortunately, some of the overall functionality will probably be provided by the tools and frameworks that are available to you:

- First of all, your content management system should provide large parts of the server-side controller logic, including a lookup mechanism that finds the right template to invoke after receiving a page request, and so on. Details depend on the actual system.
- Most content management systems provide a framework for content retrieval from the repository, usually including one or more caching mechanisms. This saves you having to implement the technical infrastructure yourself, although of course you still have to fill in the domain logic.
- Almost all content management systems provide a mechanism for template construction. Either there is a set of templates for you to customise or, in a more flexible setting, there is a well-defined way to integrate your own templates. Again, details depend on the content management system.
- You can use an Ajax library or Ajax framework that gives you the client-side components you need. Examples include ICEfaces (www.icefaces.org) and RichFaces (labs.jboss.org/jbossrichfaces). You can embed Ajax elements from such a framework into the pages you deliver to the web.
- Finally, you may want to use a search engine for indexing your web pages and for retrieving search results efficiently. Various search engines are available, including Lucene, a well-known open-source tool (lucene.apache.org).

This is where the patterns of this chapter come in. While the tools and frameworks save you a lot of work, implementing your domain logic is of course outside their scope. Typical questions that remain include the following:

- How can you compose meaningful page content from the individual content elements that you receive from the repository?

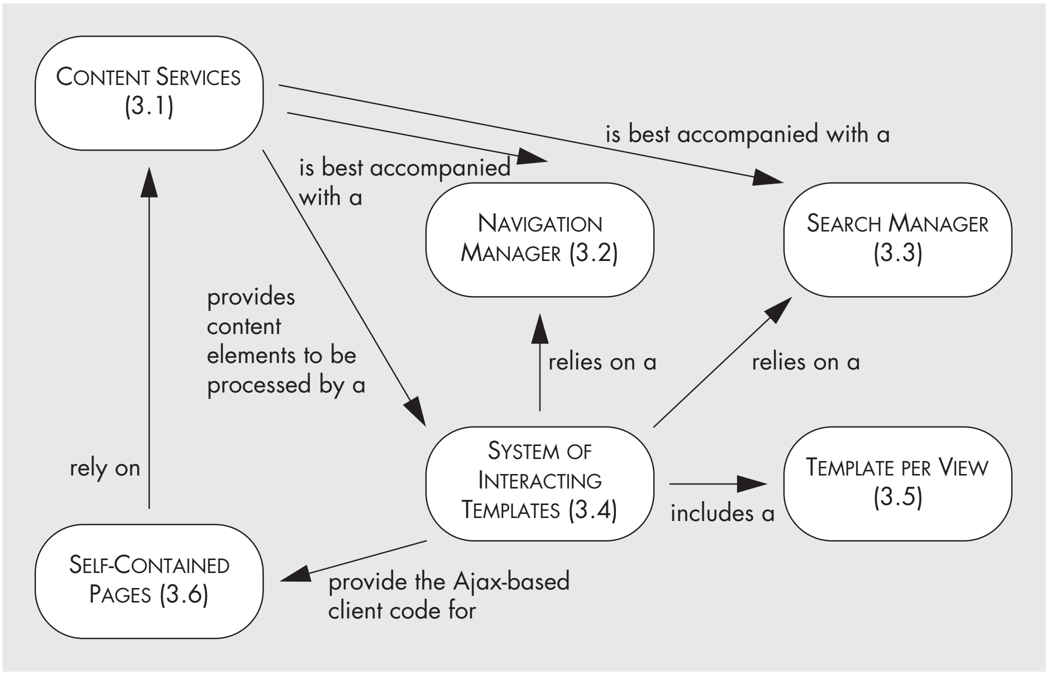

CONTENT SERVICES (3.1)
is best accompanied with a
is best accompanied with a
NAVIGATION MANAGER (3.2)
SEARCH MANAGER (3.3)
provides content elements to be processed by a
relies on a
relies on a
rely on
SYSTEM OF INTERACTING TEMPLATES (3.4)
includes a
TEMPLATE PER VIEW (3.5)
SELF-CONTAINED PAGES (3.6)
provide the Ajax-based client code for

Figure 21: Road map to the patterns on content delivery and interaction

- How can you integrate navigation elements and search functionality smoothly?
- How can you add personalisation in an efficient way?
- How can you organise templates and reuse layouts?
- How can you deliver client-side functionality to the browser?

These are the questions I'd like to address in this chapter, with the exception of personalisation, which we'll study specifically in Chapter 4. Figure 21 presents a road map to the patterns in this chapter.

3.1 Content Services

Context

You have kicked off your web project by defining a CONTENT TYPE HIERARCHY (2.1), naming the various content types along with their attributes, as well as possible association and abstraction relationships. You have also implemented WORKFLOW-BASED VALIDATION (2.5) to ensure that complete and consistent content artefacts are stored in the repository.

You are now ready to design the server-side components that will process content for its delivery to the web. The plan is to begin with a standard model-view-controller architecture and to add custom components that implement the necessary domain logic.

Problem

How can you avoid domain logic being scattered all over your server-side components?

Example

The content model for the House of Effects is organised around a few central content types. Articles, announcements, presentations and shop item descriptions are the most notable artefacts stored in the content repository. Pictures and videos are stored as separate content elements, although of course they are referenced from the objects that use them.

Typically, a web page will require more than just one content artefact. You can see that from the example screenshots in Figure 22 (in fact there will be more). All pages use different layouts, but what they have in common is that they are all composed from different content elements.

The first page shows an article with its associated pictures, three in this case. The second page contains an announcement with one picture. The third page uses a list of announcements to generate thumbnails for the event calendar. It doesn't use all the attributes of an announcement though – a few key attributes are sufficient to create the list entries. The fourth page relies on a list of articles, including announcements, presentations and item descriptions. Again, the list entries only require a subset of the available attributes.

It will be our job to provide software for assembling the necessary content for the different page types. Of course we don't want this functionality to be scattered all over our server-side components. In particular, we don't want these model aspects to enter into the templates or any other view layer components.

Forces

There is no 1:1 correspondence between the content elements stored in the repository and the content that is used to populate web pages. Content in the repository is organised according to the demands of data modelling, as expressed in the content model. When content is assembled for page generation, however, the modelling aspect takes a back seat to the concrete demands of the individual web pages. Web pages usually require what is best described as 'content aggregates', such as assemblies of related content elements or lists of content elements.

The server-side model components have to bridge this gap. Content artefacts from the repository have to be mapped onto the content aggregates the web pages require.

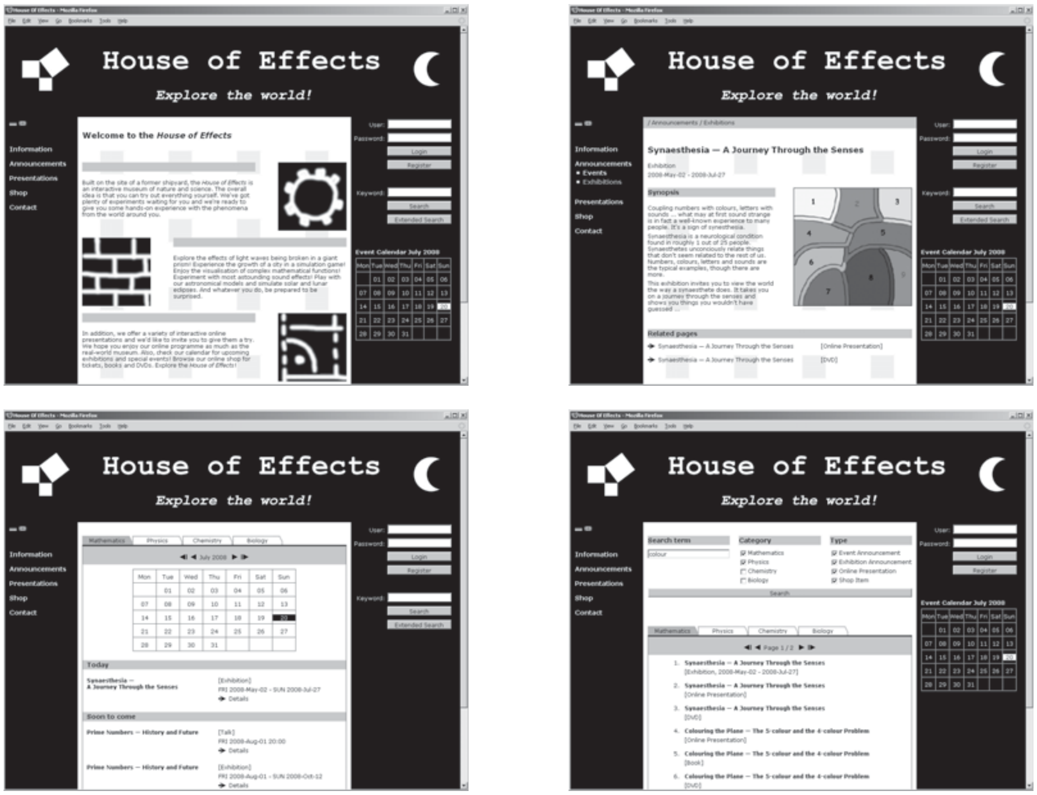


Figure 22: Different page types for the House of Effects site

However, the model components that implement such a mapping have to fulfil three important requirements.

First, all necessary domain logic must have been applied before content aggregates are made available to any view components. View components must be able to generate HTML fragments without applying any further domain logic, or the separation between model and view would be violated – with dire consequences for maintainability, as Martin Fowler points out (Fowler 2003).

Second, view components must be able to use those content aggregates in a highly efficient way. Efficient processing must be possible even in the presence of comprehensive content structures, such as long lists of content artefacts. It's the responsibility of the model components to provide the view components with content aggregates that are adequately structured.

Third, reliability is a requirement. Usually WORKFLOW-BASED VALIDATION (2.5) takes place before content elements are used for page generation. However, there are scenarios in which no validation has been performed, such as a preview function. But even in cases like these, the view components must be able to process the content aggregates they receive properly, so the domain logic has to ensure specific consistency requirements.

Meeting these requirements adds to the complexity of the domain logic, but there is no alternative. Burdening the view components with model aspects would lead to much complex scripting code scattered over server pages – a scenario that is notorious for its disastrous software structures and, consequently, for its extremely poor maintainability.

Solution

Implement services that provide the content aggregates that the various pages of your site may require. This begins with single content elements from the repository, but also includes lists or compositions of content elements. If necessary, content services must apply personalisation. Whatever the content services make available, it must be ready to be processed by the presentation logic.

To provide ready-to-use content aggregates, a content service has to rely on functionality that extends over the logical layer and the repository layer within the LAYERED ARCHITECTURE FOR CONTENT DELIVERY (1.5), as Figure 23 explains.

Here is the functionality that you'll have to include when you implement a content service:

- Content retrieval is necessary to obtain the required content elements from the repository. It's only natural that content retrieval relies heavily on the interface the content management system offers for repository access.
- Validation has to be applied to the content elements retrieved from the repository, even if WORKFLOW-BASED VALIDATION (2.5) has been applied previously. You can plan to reuse validation functionality that you have already implemented.
- Because repository access is usually a remote operation, many content management systems provide caching mechanisms to speed up content retrieval. In many cases this will be adequate, but if these mechanisms turn out to be insufficient you can opt for a custom-built cache that stores content elements locally.
- It sometimes makes sense to 'polish up' content elements from the repository before passing them to the logical layer. Examples include resizing images, breaking down text into paragraphs, integration of images into a text flow, formatting of date values and so on. You can choose to define 'wrapper' objects around content elements and so prepare the elements for use by view components.
- The logical layer's most prominent task is to receive objects from the repository layer and assemble them into the content aggregates the various page types require. Content aggregates can take many different forms. This begins with single, and

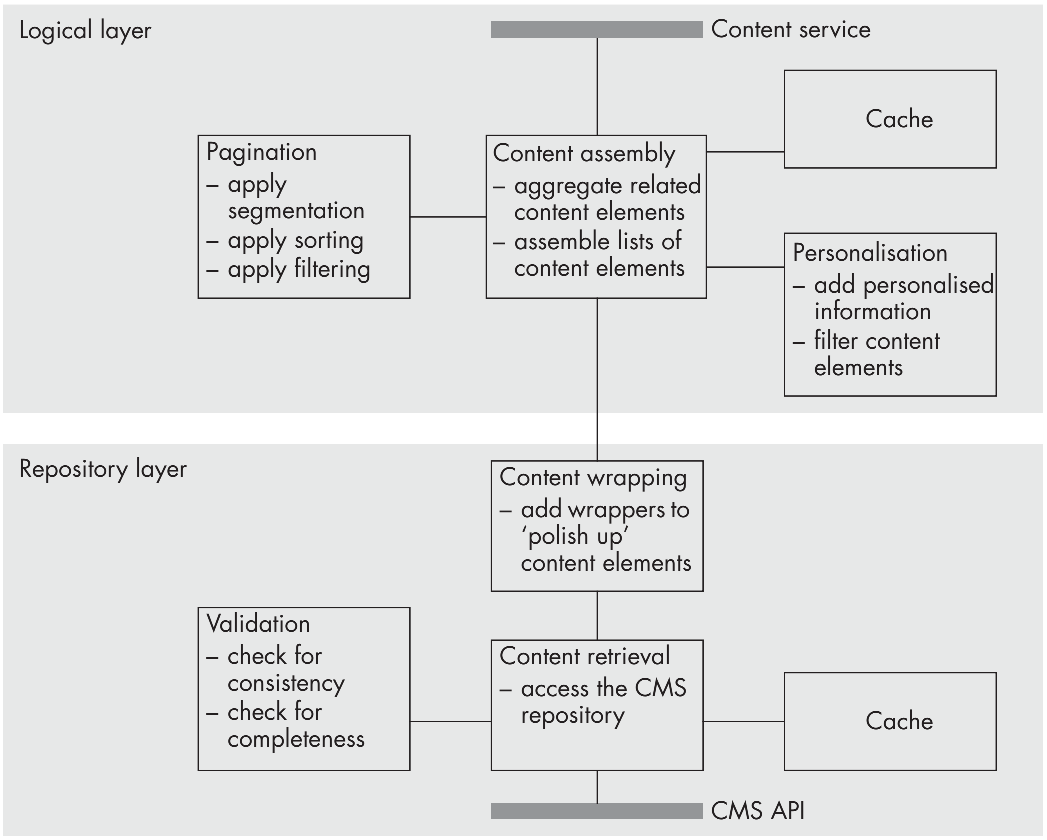

Logical layer
Content service
Cache
Pagination – apply segmentation – apply sorting – apply filtering
Content assembly – aggregate related content elements – assemble lists of content elements
Personalisation – add personalised information – filter content elements
Repository layer
Content wrapping – add wrappers to 'polish up' content elements
Validation – check for consistency – check for completeness
Content retrieval – access the CMS repository
Cache
CMS API

Figure 23: Model components involved in a content service

possibly wrapped, content elements, but also includes compositions of associated content elements or lists of content elements.

- In the case of a personalised site, personalisation strategies must be applied before content is made available to any view component. You may choose to implement your own personalisation component, or you can delegate personalisation requests to an external personalisation engine.
- Long lists of content elements might require pagination. Obviously, view components might be concerned with pagination to some extent, but content services should be able to split lists of content elements into segments anyway, if only to ensure that no overly long lists of content elements are sent to the browser.

In addition, to make things easier for the view components, content services that return lists of content elements should be able to apply sorting and filtering.

- Finally, a content service may introduce a cache on the logical layer. Unlike the repository layer cache, this cache provides fast access not only to basic content elements, but also to content aggregates that might have been subject to personalisation and pagination.

A content service that implements this functionality is a powerful tool. Integrating all necessary model aspects, it manages to provide precisely those content aggregates the view components require.

Content services can be used by server-side and client-side view components alike:

- First, templates from the server-side template layer can call content services from the underlying logical layer. Technically, there are various ways in which templates can do this, ranging from direct service invocation to the use of tag libraries. Whatever method is used, when a template generates HTML fragments it can directly use the content aggregates it receives from a content service.
- Second, client-side Ajax components can rely on content services. The idea behind SENSIBLE CLIENT-SIDE INTERACTION (1.3) is to handle specific user requests directly within the browser. To handle such requests, Ajax components may choose to use a content service as a web service.

Content services embody much of the domain logic for your site. Interpreting associations between content elements from the repository, they compose content aggregates in a way that is meaningful in terms of the application domain.

It's no surprise that implementing content services is a non-trivial task. How much of a challenge it is depends on the complexity of your CONTENT TYPE HIERARCHY (2.1) and on how heterogeneous the different page types are.

Example resolved

Let's analyse what content services make sense for the website for the House of Effects. First we need to identify the page types that we have to support. We've already seen a few in Figure 22. Others include online presentations and shop item descriptions.

Taking this information as a basis, we can identify the following content services:

CONTENT SERVICE	CONTENT AGGREGATE	PAGINATION
ArticleService	A single standard article, including all referenced pictures and videos	–
AnnouncementService	A single announcement, including all referenced pictures and videos	–

CONTENT SERVICE	CONTENT AGGREGATE	PAGINATION
AnnouncementListService	A list of announcements over a specific time period	Filtering by category (mathematics, physics, chemistry, biology), pagination, sorting
PresentationService	A single presentation, including all referenced details, pictures, videos and animations	–
ItemService	A single shop item description, including all referenced details, pictures and videos	–
ItemListService	A list of shop items	Filtering by type (books, DVDs), pagination, sorting
ArticleListService	A list of search results	Filtering by category (mathematics, physics, chemistry, biology) and type (presentation, event announcement, exhibition announcement, shop item), pagination, sorting

Each content service is responsible for making a specific type of content aggregate available. Some services return sets of associated content elements, while others return lists of content elements. The services that fall into the latter category also offer pagination, including filtering and sorting. These services are required for those content types that appear in dynamic lists.

How can our content services be implemented? Let's study two example services in a little more detail.[6]

Let's look at the *AnnouncementService* first. Any component that uses this service has to provide a request context that includes the *id* of the announcement in question and the current locale. We know from the CONTENT TYPE HIERARCHY (2.1) presented in Figure 13 that an *Announcement* object can be associated with several *Picture* and *Video* objects, so this gives us the content aggregate we're looking for. The necessary content elements are retrieved from the repository and, after some validation, are transformed into 'wrapped' objects that, for convenience, offer some additional functionality. The wrapper around an *Announcement* object offers functionality to identify those places within the announcement's markup where pictures can be anchored, while the wrapper around a *Picture* object offers resizing functionality. The *AnnouncementService* can now return the requested content elements in a way that makes it easy to generate HTML fragments for an announcement page.

The *EventListService* works similarly. As it returns a list of content elements, the request context is different, though, and includes parameters necessary for pagination, fil-

[6] For the moment we'll be ignoring the influence that personalisation has on our content services. The services will have to integrate personalisation, but we'll be dealing with this aspect in Chapter 4.

tering and sorting. The service interprets these parameters to retrieve the accurate set of *EventAnnouncement* elements from the repository.

Benefits

+ Content services may cover a wide range of functionality and add a lot to the clarity and straightforwardness of your software architecture. Content services free view components from model aspects, so that once you implement a SYSTEM OF INTERACTING TEMPLATES (3.4) less scripting code will be necessary. Content services also strengthen the separation of concerns within the logical layer, so model components become more focused (as shown in Figure 23) and redundant functionality is avoided to a large degree. Overall maintainability benefits.
+ Since less scripting code is necessary, testability is also greatly improved. Your domain logic resides completely in the logical and repository layers, where a programming language is used, as opposed to any scripting mechanisms, so that writing unit tests becomes possible.
+ Server-side efficiency benefits from the option of introducing local caches (Dyson Longshaw 2004) at different levels of object granularity. You can cache content elements received from the repository, you can cache the 'wrapped' objects or you can cache the (possibly personalised) content aggregates. It is likely that you won't need all these different caches, and you'll probably prefer to focus on those caching mechanisms that are readily available from your content management system. In any event, the architecture allows you to tune the system performance by introducing caching at different stages during the content delivery process.
+ Client-side efficiency benefits from the integration of pagination into content services. Browsers will only receive reasonably sized lists of content elements. Client-side Ajax components may still require pagination functionality for presentation purposes, but they won't be sent lists that are too voluminous for them to handle.
+ Robustness is improved as a consequence of validation. Even if for some reason no WORKFLOW-BASED VALIDATION (2.5) could be performed during content management, some basic validation will still take place during content delivery, before content is made available to any view components. This avoids corrupted web pages as a consequence of invalid or inconsistent content – a scenario that would damage the reputation of your site.

Liabilities

– The solution sketched above mentions personalisation, yet it doesn't offer any concrete advice about how personalisation strategies can be integrated. In fact there are other patterns that complement this one as far as personalisation is concerned. Most importantly, you can integrate CONTENT FILTERS (4.1) into your content services if you wish to tailor your site to specific user preferences. Also, caching is sometimes difficult in the presence of personalisation, and if that turns out to be the case for your site, SEGMENT-SPECIFIC CACHING (4.3) can represent a viable solution.

– While caching can clearly speed things up, its implementation isn't always simple. There isn't much you have to worry about as long as you rely on off-the-shelf components alone, such as the caching mechanisms your content management system may provide. But if you decide to establish your own caching strategies, you have to be concerned about implementation details and the necessary effort, as Paul Dyson and Andy Longshaw point out (Dyson Longshaw 2004). The trickiest part is usually the implementation of an invalidation mechanism that updates a cached object whenever the original object has changed. In most cases, the method of choice is a LISTENER-BASED SYNCHRONISATION (1.4) between the content repository and the local cache.

– Content services make *content* available, as the name suggests. However, they intentionally ignore the navigation hierarchy – any navigation-related information is outside their scope. On the other hand, view components do require navigation-related information in addition to the mere content. A NAVIGATION MANAGER (3.2) can help you make navigation-related information available.

– Content services are fine for looking up content if the requested content element is known by its identity. This is the case when you look up content for a specific page, or when you follow references from one content element to another. Finding content elements that meet arbitrary search criteria is quite a different matter. A SEARCH MANAGER (3.3) can help you to integrate a search engine that offers sophisticated search strategies.

– The implementation of your content services depends on the CONTENT TYPE HIERARCHY (2.1) you have designed for your site. Obviously there is little you can do about this – an implementation will always depend on its underlying model. Ideally your content model should be fairly stable, but if changes to the model become inevitable, you'll have to adapt your content services accordingly.

– Robustness requires effective content validation during content delivery in addition to possible WORKFLOW-BASED VALIDATION (2.5). You must ensure that you reuse all validation code if you want to avoid redundant functionality. This may come down to some cross-application reuse, as the workflows belong to the content management application, while the content services are part of your content delivery components.

3.2 Navigation Manager

Context

You have implemented CONTENT SERVICES (3.1) that make content of various types and in various combinations available. As a consequence of the DECOUPLING OF CONTENT AND NAVIGATION (2.2), however, all navigation structures are outside the scope of these services.

It is clear that view components need to be aware of navigation structures. You have laid the basis for specifying these navigation structures by adding a pseudo content type for navigation nodes to your CONTENT TYPE HIERARCHY (2.1). You may also have introduced DYNAMIC CONTENT LINKING (2.3). Navigation information is readily available and it's now time to use this information properly.

Problem

How can you prevent templates and other view components from being burdened with the calculation of navigation-related information?

Example

Like most other websites, the website for the House of Effects features a set of standard navigation mechanisms. As you can see from the various screenshots in Figure 22, a tree-like navigation bar on the left can be found on every page. It represents the navigation hierarchy specified by the content editors, and it is context-sensitive in the sense that the branch of the tree that opens depends on the current page. Most pages (announcements, presentations and shop item descriptions) also feature 'breadcrumbs' near the top of the page – essentially the page's path within the navigation hierarchy.

Of course there are more links that serve navigation purposes. Links from an article to related articles, links from the event calendar to individual announcements, links from a list of search results to the full pages: these links, too, have to be provided somehow.

Forces

Linking content is what makes the World Wide Web a web: hyperlinks are what distinguishes a website from a mere information catalogue.

The internal links of a website essentially fall into two categories.[7] First, there are links that represent the navigation hierarchy that is maintained by the content editors. The

[7] External links – those that refer to other websites – are outside the scope of this pattern, as they cannot be generated and therefore aren't subject to server-side computation. External links are essentially just attributes that are maintained along with other content, though some sites choose to employ tools that test links for correctness and sort out dangling links.

typical navigation bar and breadcrumb navigation fall into this category. Calculating links for such navigation elements requires the traversal of the tree-like structure spanned by the individual navigation nodes. As a consequence of the DECOUPLING OF CONTENT AND NAVIGATION (2.2), these navigation nodes are specified separately from the actual content.

Second, there are links that represent associations between arbitrary content elements, regardless of their position in the navigation hierarchy. However, associations hold between content elements, whereas links connect pages. Calculating links from associations can be non-trivial. For example, if one content element keeps a reference to another and you have to generate a page for the first, then obviously that page should include a link that points to the second one. But what if that content element appears on more than one page? To which page should the link point?

Whatever strategy you implement to solve these problems, it will involve a certain amount of computation. Moreover, in the context of DYNAMIC CONTENT LINKING (2.3), you'll have to take dynamic navigation structures into account as well.

Things are even more complex if the navigation hierarchy depends on an overall request context. For example, the navigation hierarchy can vary between different languages, or it could be different for the Internet and for an intranet. Personalisation can lead to different navigation hierarchies too – different users may get to see different parts of the hierarchy. Cases like these require that the current request context be evaluated while the links for a page are being generated.

However, there are downsides to the computation of links. First, because complexity is involved, it is crucial that link management doesn't spread all over your server-side – let alone client-side – components. Maintainability demands that the non-trivial evaluation of navigation structures and the computation of links should be concentrated in one place.

Second, link computation relies heavily on the completeness and consistency of the underlying navigation hierarchy. Generated links will always represent what the content editors have specified. But even if the content editors have failed to come up with a well-formed navigation hierarchy, dangling links should never be the consequence, and links to empty pages shouldn't exist either. If you want an appealing website, you have to find a way to prevent inaccurate links from being generated.

Solution

Establish a navigation manager that provides the various kinds of navigation-related information that your site requires. Evaluating relationships between content elements and mapping navigation nodes onto URLs, the navigation manager encapsulates the link management for your site.

An integral component of the server-side logical layer, the navigation manager can be called by model and view components alike. It makes navigation-related information available through an interface, which typically includes the following:

- A function that returns the URL for a given navigation node. This function needs to be aware of possible prefixes it has to add to the navigation node's path to construct the URL.
- A function that yields the path to a given page. The path is an ordered list of navigation nodes, beginning with the node that represents the start page and ending with the node that represents the given page itself. The path can be obtained efficiently by following the parent references that you introduced into navigation nodes when implementing DECOUPLING OF CONTENT AND NAVIGATION (2.2).
- A function that yields the navigation tree for a given page. The navigation tree represents the complete navigation hierarchy, although usually it opens only along the branch that contains the given page, and is otherwise closed. You may also choose to ignore empty nodes – that is, with no children or content element assigned to them. More variations are possible, similarly to the different ways in which folder hierarchies are presented in user interfaces. The navigation tree can be obtained by traversing the tree-like structure spanned by the entirety of navigation nodes. The result can be returned as a tree object or as a list, whatever is easier to process. Be careful to avoid endless loops caused by circular links when you implement the necessary tree traversal.
- A function that yields all navigation nodes that reference a given content element. The result set may include more than one navigation node if a content element appears on several pages.
- A function that yields the default navigation node for a given content element. If the content element is referenced from only one node, then the decision is clear. If it appears on several pages, then this function defines which of these nodes best represents the content element. It's usually the node that features a full view of the content element in question and not just, say, a teaser view.

Functions like these usually take the request context as an input parameter. The request context typically consists of things like the current language, the current domain or (in the case of personalisation) the user's identity. The request context is a prerequisite for calculating navigation paths, navigation trees and so on.

To implement these functions, the navigation manager has to evaluate the different kinds of relationships that hold between navigation nodes. This includes the static parent-child relationships between navigation nodes, as well as the dynamic relationships that are expressed through DYNAMIC CONTENT LINKING (2.3). A navigation manager designed in this way provides you with all the information you need for generating the links for your site, whether they appear in navigation elements or in the actual content of any pages.

Of course, you may choose to design a navigation manager that offers different or additional functionality, depending on your navigation requirements. However, this pattern illustrates what a navigation manager is supposed to do and how it can implement the desired functionality.

Navigation-related information is required throughout a website. As a consequence, a navigation manager has to work efficiently. When implementing the navigation manager, you can opt to cache those navigation trees and subtrees that are requested frequently. Also, whatever information the navigation manager makes available, its results must be organised in data structures that are straightforward to process by other components, especially view component such as templates.

Example resolved

We decide to develop a dedicated navigation manager component that will handle all navigation-related requests. It has to offer the functionality that's necessary to generate the navigation elements our site requires, most notably the tree and breadcrumb navigation. The navigation manager will also cover the mapping of content elements onto navigation nodes, so that it can provide links between arbitrary pages.

The following table summarises the most important functions the navigation manager makes available:

FUNCTION	RESULT
getUrl (NavigationNode node, Locale locale, User user)	Url The full URL for the given navigation node.
getPath (NavigationNode node, Locale locale, User user)	List <NavigationNode> The full path from the start page to the current page, represented by the given navigation node.
getTree (NavigationNode node, Locale locale, User user)	Tree <NavigationNode> The navigation tree, unfolded along the path to the current page, including the current page's siblings and children.
getAllReferrers (Article article, Locale locale, User user)	List <NavigationNode> An unsorted list of all navigation nodes that contain references to the given article.
getDefaultNode (Article article, Locale locale, User user)	NavigationNode The default navigation node for the given article. If there is more than one node that refers to a certain article, the 'most prominent' one is chosen.

These functions allow us to generate all navigation elements as well as other internal links within our web pages:

- To generate the tree navigation, we first call *getTree* to obtain the current navigation tree – the visible part of the overall hierarchy. Next, we call *getUrl* for all tree elements to obtain the URLs that we need to include in the HTML fragment.
- To generate breadcrumb navigation we do essentially the same, but calling *getPath* instead of *getTree*.
- To generate a link that represents an association between arbitrary content elements we first call *getDefaultNode* for the target. To receive the URL we need, we then invoke *getUrl* on the navigation node that is returned.

The functions our navigation manager offers will be used mainly by templates that rely heavily on navigation-related information. They can embed the results from these functions in the HTML fragments they generate without having to do much computation themselves.

Benefits

+ Maintainability of the logical layer is improved. As a consequence of the separation of concerns all link management and all navigation-related functionality is encapsulated by one dedicated component, largely avoiding complex or redundant code. It will be relatively easy to make changes to the navigation logic should the need arise.

+ Maintainability of the view components is equally improved. View components are freed from navigation-related computation completely. Template code especially is much simpler, as all navigation information is readily available and no tedious computations have to be performed. As a consequence, it will be easier to design a SYSTEM OF INTERACTING TEMPLATES (3.4).

+ It becomes easier to test navigation-related functionality. The algorithms around the computation of navigation information are all implemented by the navigation manager, so applying unit tests is straightforward. This would be difficult or impossible if the functionality offered by the navigation manager was scattered across several components, let alone view components.

+ Efficiency is improved, because the navigation manager can implement fast algorithms for tree traversal that efficiently calculate lists of navigation nodes. In addition you can choose to enhance the navigation manager with a cache to further improve efficiency.

+ Reliability is increased, as the navigation manager applies the necessary consistency checks on navigation elements. Illogical links, dangling links or links that point to empty pages can all be avoided by simple routines that the navigation manager can perform.

Liabilities

– Obviously the navigation manager can only use the information it obtains from the navigation structure specified by the content editors. While it can detect inconsistencies and dangling links, it cannot always *repair* inaccurate or inconsistent navigation structures: content editors are still responsible for providing navigation structures that are accurate and meaningful.
– If you choose to implement a local cache for the navigation manager, you must allow for the extra effort necessary to do so. In particular, you will have to develop cache invalidation mechanisms that update the cached objects whenever any navigation structures change.

3.3 Search Manager

Context

You have implemented CONTENT SERVICES (3.1) that make various types of content aggregates available. Looking up content artefacts based on their identity, these services work well to retrieve content required for specific web pages, making use of association relationships between content elements whenever necessary.

This, however, isn't enough. You also need to be able to look up content elements that match specific search criteria without knowing their identity – or at least this is the case if you plan to offer a search function, as is likely.

You have already laid the foundation for implementing the required functionality by building a TAXONOMY BASED ON KEYWORDS AND CATEGORIES (2.4) and by integrating the necessary attributes into your content model. You have also established a mechanism for LISTENER-BASED SYNCHRONISATION (1.4) between your content repository and your search engine, so that the search engine is notified of all relevant content changes.

Problem

How can you implement a powerful and user-friendly search function and reduce the number of expensive search queries?

Example

Users of the House of Effects website should be able to do more than browse the site via the existing navigation schemes. They should be able actively to search for presentations, announcements or shop items.

We'd like to offer a powerful search function. Specifically, we'd like to integrate the categories we introduced into our content model earlier: mathematics, physics, chemistry and biology on one hand, and presentations, announcements and shop items on the other. Users should be able to use these categories as criteria for searching, sorting and filtering.

We'll also have to bear in mind that the volume of content that a search engine returns can become large. The search function should be fast, and it shouldn't return too many results, at least no more than the client can handle, no matter how unspecific the original search query may have been.

Forces

A search engine is a powerful tool. Assuming that you have provided the search engine's indexer with the necessary information, you can now take advantage of this by integrating the search engine into your server-side architecture and making its results available to your content delivery components.

To do this, some component will have to translate search queries entered by users into search engine calls. However, a large variety of search queries are possible, and different searches can be combined in many different ways. In his pattern language on *User Interfaces for Search Queries*, Tim Wellhausen gives several examples, including searches specific to a given context, searches that refine previous search results and searches that users can save to persistent storage (Wellhausen 2005). If you choose to implement such features, your search function will be more convenient, but translating search queries into search engine calls becomes a non-trivial business that, among other things, may involve storing objects in the HTTP session.

Adding to the complexity is the fact that the presentation of search results is a good candidate for SENSIBLE CLIENT-SIDE INTERACTION (1.3). Sorting and filtering search results is often done by some client-side Ajax component, which invites a discussion of how search functionality should be distributed over client and server. It is possible for both client and server to be involved.

Whatever design you choose, you have to ensure robustness and efficiency. The most unfortunate case is probably that of a user triggering a search query that would in principle yield an extremely large number of results. Even in a case such as this, the search function must respond in reasonable time. In addition, any view component in charge of displaying the results must be able to handle the result set without difficulty.

Solution

Implement a search manager that receives and handles all search requests throughout a user's session. The search manager contacts the search engine and makes the search results available to the components that will process them. For convenience, the search manager may store previous queries and results, and it can check queries for plausibility before forwarding them to the search engine.

Processing search results can involve the server and the client. Known from traditional websites, a search query can result in a standard server request in which server-side templates generate a page in response that displays the results. On the other hand, it's fairly common these days to introduce an Ajax component for submitting search queries, as well as for sorting and filtering the results. This means that search results may or may not have to be processed by client-side software.

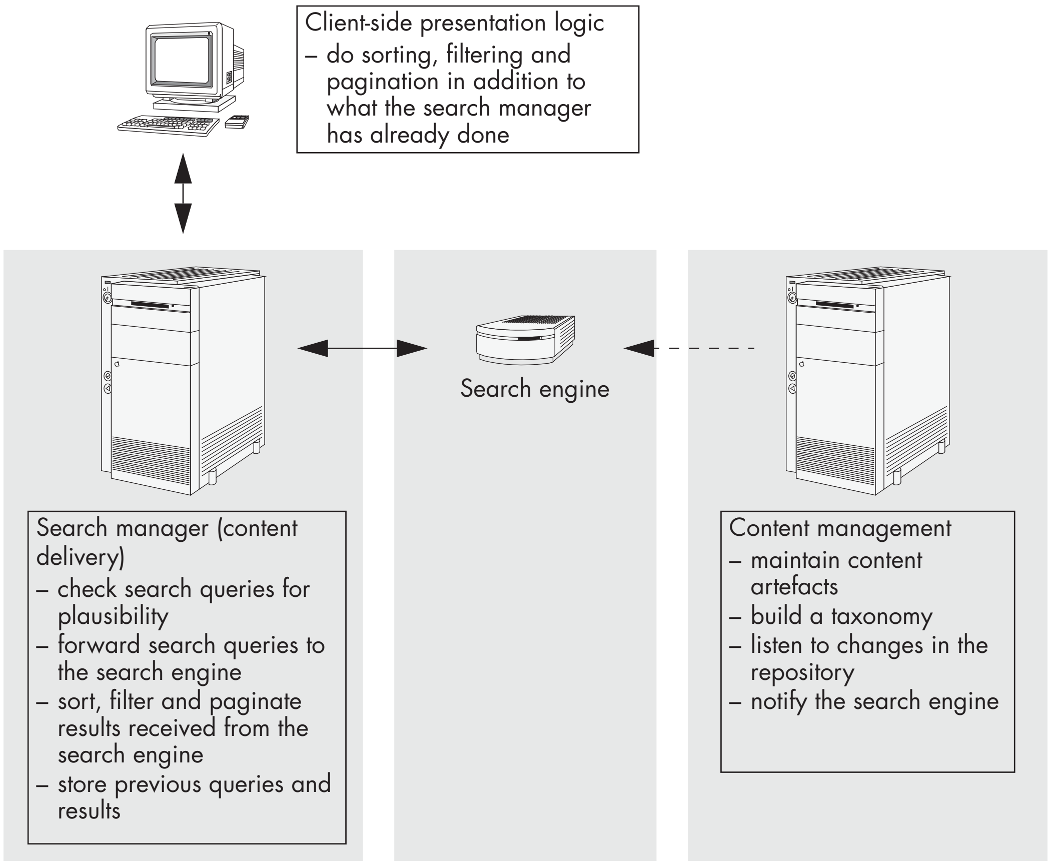

Client-side presentation logic
– do sorting, filtering and pagination in addition to what the search manager has already done
Search engine
Search manager (content delivery)
– check search queries for plausibility
– forward search queries to the search engine
– sort, filter and paginate results received from the search engine
– store previous queries and results
Content management
– maintain content artefacts
– build a taxonomy
– listen to changes in the repository
– notify the search engine

Figure 24: Search manager and search engine

As a logical consequence, the search manager has to be designed in such a way that it can be invoked from server-side or from client-side components, the latter via a web service interface that allows Ajax components to receive data from the server.

In either case it is the search manager's job to encapsulate the entire search functionality by accepting search queries and making sets of search results available, performing the following tasks:

- Checking search queries for plausibility. In particular, the search manager has to implement functionality that protects the system against queries that would return an excess number of results, for example by ignoring search terms that are too short and therefore not sufficiently specific.

- Analysing a query entered by a user and translating it into a search engine call. What a search engine call looks like depends on what search engine you're using. In any case, the search manager needs to pass both the search query and possible keywords and categories to the search engine, so that the search engine can scan its database efficiently for entries that match the request.
- Applying filters, sorting and pagination to the results received from the search engine. This makes sense even if the primary responsibility for the presentation of search results lies with some client-side Ajax component. The search manager can still apply a default order, a default filter and so on before an initial set of results is sent to the client.
- Storing previous queries and previous results. It may make sense for the search manager to store previous queries and previous results in the user's session. In this case the most recent query or queries, as well as their search results, will remain available should the user navigate to other pages and later revisit the search page.
- Storing search queries. The search manager may also store a user's search queries across sessions, or provide the basis for sharing search queries between users.

Figure 24 summarises the search manager functionality in its overall context. The search engine is fed the necessary information by the content management server and is later prompted for results by the search manager. The search manager makes the results available to the client, which may choose to offer additional mechanisms for presenting the results.

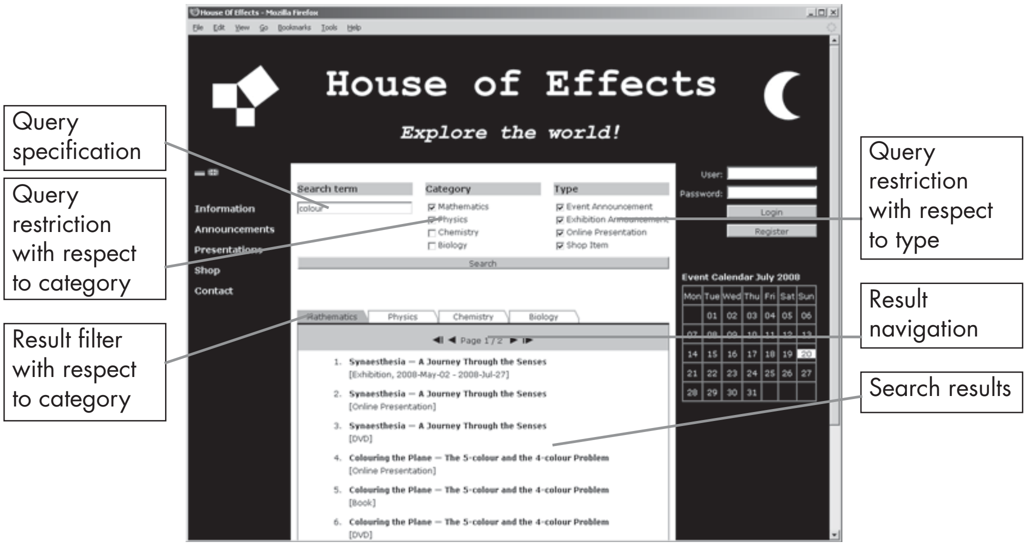
Example resolved

Figure 25 shows the search page of the House of Effects website. Its central area is divided into two parts: the search query at the top and the search results at the bottom. This is convenient, as it makes it easy for users to refine the query they have entered previously.

Presentation of the search results is completely implemented with Ajax components and so is handled solely by the client. This applies to the filtering of results (the tabs representing the different categories) and to the navigation through the results (the 'forwards' and 'backwards' buttons).

Making an actual search query, however, has to involve the server, but it does not mean that the search page is reloaded completely. If the user presses the 'Search' button, an Ajax component forwards the query to the server-side search manager via the XMLHttp interface, receives the search results und displays them.

Things look different if a user submits a search query through the small search box on the right-hand margin that appears on every page except the search page itself (see Figure 22). The small search box only serves as an input form: the results will be presented on the main search page, as in Figure 25. As displaying the search results involves loading another page, the search manager is contacted via a standard HTTP request.



1. Synaesthesia — A Journey Through the Senses
   [Exhibition, 2008-May-02 - 2008-Jul-27]
2. Synaesthesia — A Journey Through the Senses
   [Online Presentation]
3. Synaesthesia — A Journey Through the Senses
   [DVD]
4. Colouring the Plane — The 5-colour and the 4-colour Problem
   [Online Presentation]
5. Colouring the Plane — The 5-colour and the 4-colour Problem
   [Book]
6. Colouring the Plane — The 5-colour and the 4-colour Problem
   [DVD]

Figure 25: Client-side and server-side search functionality

In either case the search manager receives the search query, including the categories and types a user may have selected. It calls the search engine and makes the results available in a default order. The search manager ignores search terms that are too short to be significant, and limits the number of search results to a configurable maximum to avoid clogging the client.

On top of this, the search manager retains the user's latest search query and its results in the user's session, so that it can make them available once the user re-enters the search page without having to make another search engine call.

Benefits

+ Maintainability is clearly improved, since most search-related functionality is encapsulated in a single component. It's true that the client may require specific Ajax components for the *presentation* of search results, but the model aspects of the search function are encapsulated in one server-side component. This reduces the effort should changes to the search logic become necessary.

+ The introduction of a search manager also gives you some independence from the search engine you're using. Should you ever decide to replace the search engine, this will only affect the search manager, not any other components.

+ Robustness and efficiency are both improved, as the search manager is able to reject queries that are too unspecific to be useful. The search manager protects the client from having to deal with lists of search results that would slow it down unacceptably or would be too large for it to handle.
+ Usability can be improved if you choose to implement convenience functions that rely on the search manager's ability to maintain a list of previous search results. For example, the modification of a previous query is a feature that's often desired, and the search manager can help you to implement it.

Liabilities

– If you choose to let Ajax components display, sort, filter and paginate search results on the client, you clearly make a move to a rich client architecture. There is value to Sensible Client-Side Interaction (1.3), but you have to be aware that you may have to implement some functionality twice. Sorting, filtering and pagination of search results are likely to be required on the server side too (within the search manager), so you have to be careful not to introduce any inconsistencies.

– Obviously you need a reasonably powerful search engine to implement the search function. Things to ensure include the search engine's ability to support the categorisation you wish to implement. Specifically, your search engine will have to support the use of special keywords, and it has to be able to search different kinds of content artefacts (HTML, PDF and so on).

– As with any other server component, you must be careful not to design a session state that's too large. Because the server has to provide sessions for all users simultaneously, overly large session objects use a lot of memory on the server side, which can result in performance penalties (Broemmer 2003). It is acceptable to store previous search results in the HTTP session, but you should define a limit to the amount of data that is stored.

3.4 System of Interacting Templates

Context

Based on your underlying Content Type Hierarchy (2.1) you have completed the custom software that constitutes the server side's repository layer and logical layer. In particular, you have implemented Content Services (3.1), a Navigation Manager (3.2) and a Search Manager (3.3).

In addition, someone, probably a web designer, has provided the layouts and style sheets for the different page types of your site, so you know what kind of pages your view components will have to generate. You're now ready to tackle the template layer within your server-side architecture.[8]

Problem

How can you avoid, to a large extent, redundant template code and inefficient page generation?

Example

Looking back at Figure 22, you can see that although the website for the House Of Effects features quite a few different page types, these page types still have many things in common. Figure 26 shows a 'wireframe' that demonstrates how a typical page is composed from smaller elements.

This of course means that there is potential for reuse. For example, the small event calendar on the right-hand margin shows up on almost all pages, and obviously we would like to reuse the template code that generates it. The same is true for other page elements.

We'd also like our site to be efficient, which suggests caching HTML elements. Although we're not focusing on personalisation at present, we need to bear in mind that personalisation has an effect on the appearance of specific page elements. As a consequence, there are a few page elements that we don't plan to cache.

First, the login area will look different depending on whether the current user is logged in or not. Caching HTML fragments is difficult if the cached elements depend on state information like this, so we will not cache the login area.

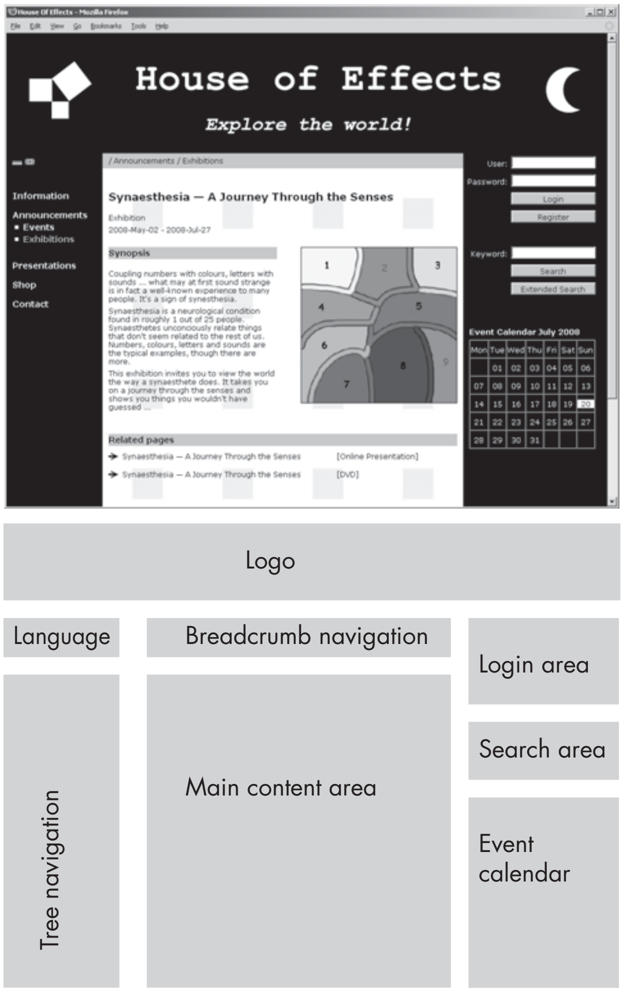
Second, the tree navigation might not be the same for different users – after all, registered users will be given access to additional online presentations, and will therefore get to see an extended menu. Since the tree navigation will depend on the current user, we won't apply caching here either.

Forces

Templates serve as blueprints for web pages. A template in this sense is essentially a piece of code that combines static HTML with HTML that represents content elements. This enables templates to specify a layout for web pages and the content elements they present. Similar pages share a template, while structurally different pages require separate templates. Where exactly the line should be drawn between 'similar' and 'structurally different' depends on the underlying technology, but the principle is the same whether you use JSP, XSLT, PHP, Ruby or something else.

However, even different page types are likely to have some page elements in common. If you provided a template for each page type, the code for the commonly used elements would have to exist multiple times. It's an old rule that redundant code is almost always undesirable, and template code is no exception.

[8] This should not suggest that implementing the logical layer and implementing the template layer have to be done in a strict order. You can of course choose to do things simultaneously, provided that the interface between logical layer and template layer has been defined.



/ Announcements / Exhibitions

Synaesthesia — A Journey Through the Senses

Exhibition
2008-May-02 - 2008-Jul-27

Synopsis

Coupling numbers with colours, letters with sounds ... what may at first sound strange is in fact a well-known experience to many people. It's a sign of synesthesia.

Synaesthesia is a neurological condition found in roughly 1 out of 25 people. Synaesthetes unconciously relate things that don't seem related to the rest of us. Numbers, colours, letters and sounds are the typical examples, though there are more.

This exhibition invites you to view the world the way a synaesthete does. It takes you on a journey through the senses and shows you things you wouldn't have guessed ...

Related pages

Synaesthesia — A Journey Through the Senses [Online Presentation]
Synaesthesia — A Journey Through the Senses [DVD]

Event Calendar July 2008

Mon	Tue	Wed	Thu	Fri	Sat	Sun
	01	02	03	04	05	06
07	08	09	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Figure 26: A wireframe for a typical page

To this end, it's helpful to remember that one of the central ideas behind web content management is to compose pages from smaller elements. As Oliver Vogel and Uwe Zdun explain in their pattern language on *Content Conversion and Generation on the Web*, templates combine content elements and basic HTML structures in a highly dynamic way to compose a multitude of HTML fragments and complete HTML (Vogel Zdun 2006). You can benefit from this flexibility if you include commonly used page elements across different templates.

You also have to take a further issue into account. If you assume DYNAMIC CONTENT DELIVERY PLUS CACHING (1.2) as a fundamental principle of your software architecture, you apply one or more caching strategies to ensure fast page delivery. The caching of HTML fragments is one possible strategy, and one that has an effect on template design. The reason is simple: caching is virtually impossible for HTML fragments that undergo personalisation or depend on user interaction, as they might look different each time a user visits the page. You have to design your templates with care if you don't wish to undermine the success of your caching strategies.

Solution

Define a system of interacting templates, from templates for the full page down to templates for individual page elements. Reuse templates for smaller page elements wherever possible. Make sure you extract any state-specific or personalised page elements into templates of their own, as this increases the potential for caching HTML fragments.

The template call hierarchy should mirror the structure of your web pages. The full page templates include templates for individual page elements, which may include templates for still smaller elements, and so on. You need a full page template for each page type you're going to support.

What the different templates do depends on their position within the call hierarchy:

- Page templates don't generate much markup themselves. They do generate the outermost page structure, things like the HTML header and body as well as references to style sheets, but otherwise only invoke other templates. Page templates are aware of the layout of the page type they represent, and so know which inner templates to invoke.
- Templates for content elements generate HTML fragments that represent individual content artefacts. Different templates represent different layouts. They all rely on the available CONTENT SERVICES (3.1) to generate a specific view of a content artefact, and may in addition use the NAVIGATION MANAGER (3.2) if they need to include links to related pages.
- Navigation templates generate the different kinds of navigation elements, such as tree or breadcrumb navigation. They obviously rely on information they obtain from the NAVIGATION MANAGER (3.2).
- Templates for search functions generate the various HTML fragments that are necessary for displaying search forms and search results. Search templates rely on the different functions a SEARCH MANAGER (3.3) offers.
- A few simple templates generate static content such as logos or footers.

In addition to the HTML fragments they generate, some of these templates may have to insert client code into their output. If you choose to apply SENSIBLE CLIENT-SIDE INTERACTION (1.3), then it is the templates' job to include the necessary client code in the pages they generate.

When tailoring the templates, you need to keep efficient caching in mind. It's often difficult to cache state-specific or personalised HTML fragments, so ensure you separate page elements that are state-specific or personalised from those that are not, and introduce separate templates for each.

Building a system of interacting templates is possible regardless of the technology you use. JSP, XSLT and PHP all provide some sort of inclusion mechanism and so allow you to implement a call hierarchy for your templates.

Example resolved

Figure 27 describes the call hierarchy for those templates that generate the announcement page from Figure 26. It begins with the template for the full page and goes down to those templates that render individual page elements.[9] Those templates whose HTML output we plan to cache are shown shaded.

This design maximises the potential for reuse among templates. Obviously there are going to be more page templates, for the start page, the search page, the event calendar page, the presentation pages, the item description pages and so on. But while these page types require page templates of their own, they can reuse many of the templates for individual page elements.

Our design also allows for a maximum of caching. Obviously the full page (generated by the *NavigationNode.renderAnnouncementPage* template) cannot be cached, as it contains state-dependant (login area) and personalised (tree navigation) elements. Except for these, however, all other page fragments are suitable for caching, including all larger multimedia objects.

Benefits

+ The overall amount of scripting code is reduced, as templates for page elements are reused wherever they're needed. Given that scripting code is often difficult to understand, this is a clear improvement for maintainability.
+ Since you have a system of templates that invoke each other, the individual templates will be relatively simple and easy to understand. Again, maintainability is improved.
+ Carefully tailored templates allow you to maximise caching. You can now apply caching to HTML fragments on the template layer of your server-side architecture (see Figure 20). Performance is improved, as you can apply caching to large objects, especially images, videos and other multimedia assets.

[9] We understand templates as methods of a specific content type, hence the notation that is reminiscent of object-oriented languages. A template that renders a page element operates on the underlying content element (or a navigation node in case of a full page) and has access to all its attributes.

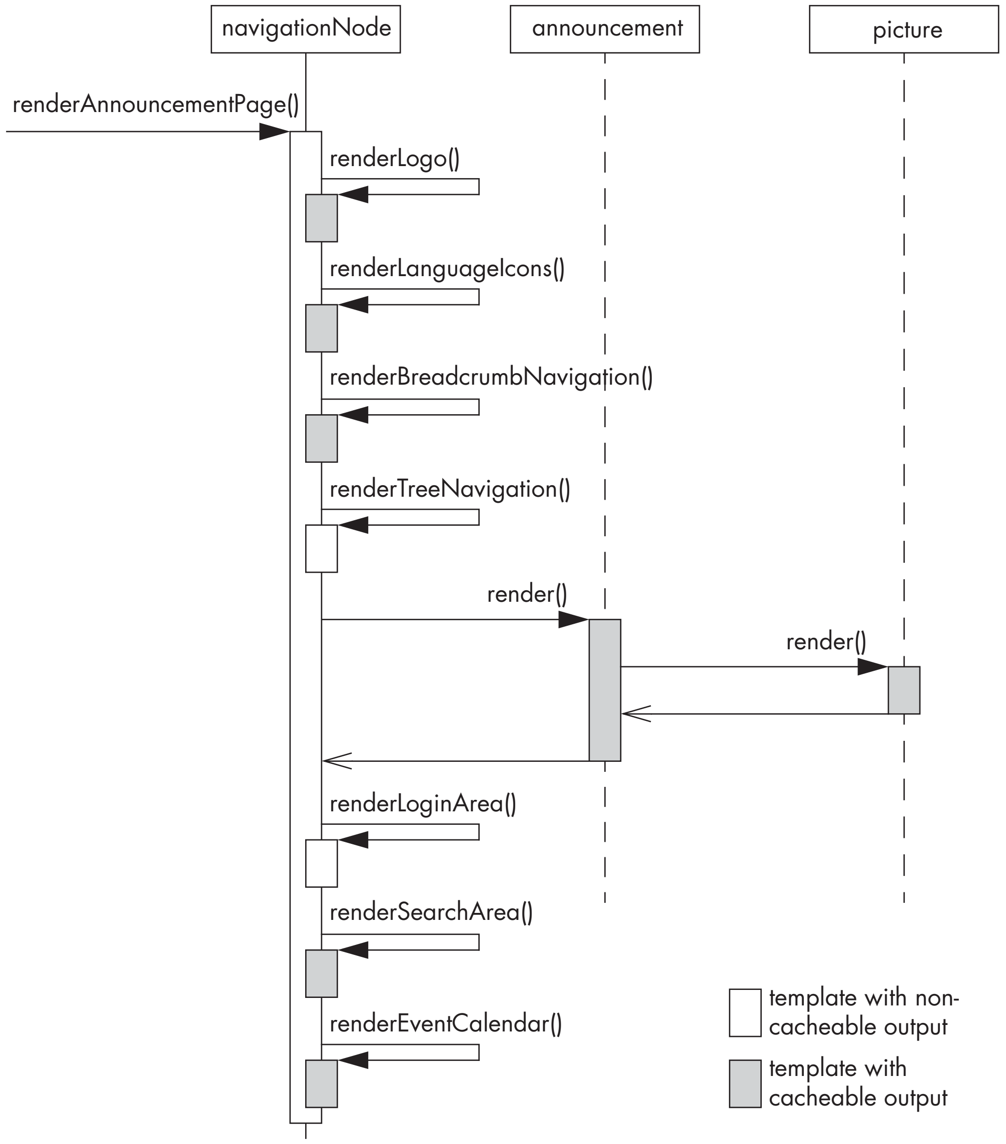


Figure 27: Call hierarchy of the template for an announcement page

Liabilities

- If you choose to use caching on the template layer, you will probably be using a mechanism provided by your content management system rather than one you implement yourself. The content management system needs to be informed about which templates' output should be cached and which templates' output shouldn't. This usually requires little more than the relevant configuration, but nevertheless it is a liability that should be mentioned.
- Splitting template code over several smaller templates is a good idea, but if you take it too far you can end up with a relatively large number of templates. This isn't necessarily problematic – just be careful not to introduce too many templates unnecessarily.
- The solution suggests that different templates may be necessary for one content type. However, it doesn't say exactly what templates are going to be necessary. Below we will see that there should be one TEMPLATE PER VIEW (3.5), where 'view' refers to the possible representations of a content element on a web page.

3.5 Template per View

Context

You are in the process of implementing a SYSTEM OF INTERACTING TEMPLATES (3.4) that generates markup for the full page and for individual page elements.

Problem

How can you support the different views that content elements may assume on different pages, for different variations of your site or for different output channels?

Example

Different views of the same content element occur frequently on the House of Effects website.

Announcements are one example. First, there is the full view of an announcement, complete with pictures and videos, if any, which is the view that occurs on the announcement pages. Announcements make appearances on other page types as well, however. The search page uses a 'teaser' view of an announcement – a view that only contains title, subtitle and date values and no pictures or videos. The same is true for the event calendar. It also uses a teaser view, although a different one (see Figure 22). Similarly, different views exist for presentations, shop items and articles in general.

In addition, we won't be using the same page layout for screen and print. There's no point in including interactive page elements in a printed page, so the plan is to provide print versions for the various page types to supplement the online versions.

Forces

It is common for content elements to require different views: there are plenty of examples. In addition to the standard view that displays a content element in its entirety, lists (such as search results) often require a teaser view – a small representation of each content element covering only its most important aspects, accompanied usually by a link to the full view. Accessibility issues sometimes demand that a text-only version of a page be available, which of course makes text-only versions of the individual page elements necessary.

Supporting different output media may also lead to a variety of views. A standard screen, the screen of a hand-held computer, a printed page and a Braille output device for visually impaired people all have different requirements for page geometry, and may support user interaction to different degrees.

This invites the question of how different views for a content element can be implemented. One option is to make the individual templates powerful enough to generate different views of the same content element. However, this strategy quickly adds to the complexity of template code. If one template has to generate different kinds of markup, depending on what view it is supposed to generate, it will quickly become complex as a consequence of alternatives and case statements.

The clear downside here is that complexity is the enemy of maintainability, especially in the context of the scripting languages that are typically used for templates. Scripting code that differentiates between several modes, introduces layout alternatives or features large case statements is notoriously difficult to understand and maintain. This isn't what you want.

Solution

Define a template for each distinct view a content type has to support. Typical examples include a full view, a teaser view, a text-only view or views for specific output channels such as mobile devices. This way all individual templates can be quite simple, responsible only for assembling the markup necessary for one specific view of a content element.

Exactly what templates you will need depends on the specific requirements for your website or web platform, but it is possible to give some concrete advice. Let's begin with page templates. Page templates have access to the attributes of the navigation node that represents the current page. Page templates generate the outermost page structures and 'know' which inner templates for individual content elements they have to invoke. Each page type requires at least one template. However, there may be more than one template per page type, for the following reasons:

- If your site supports different media, for example standard browsers and hand-held devices, you will have to handle different page geometries. It makes sense to introduce separate page templates that each use a specific CSS sheet.
- If accessibility is an issue you may have to provide page templates for text-only representations. Such templates often also use their own CSS sheets and invoke special text-only templates for the content elements from which the page is composed.

In many cases page templates send just one view of a page to the browser. However, page templates may choose to send alternative views and let the browser decide which it should display. This can involve some Ajax component that is able to switch between views, or it can be done by built-in browser functionality. Many browsers can distinguish between 'screen' and 'print' provided the HTML is attributed accordingly, while some also support 'hand-held', 'aural', 'Braille' and others.

Navigation templates also operate on navigation nodes and their attributes. Representing different views of the current navigation node, the following navigation templates make sense:

- A template for tree navigation, representing the current navigation node in the context of the overall navigation hierarchy. Parts of the hierarchy may be hidden, and only the branch that includes the current node may be shown.
- A template for breadcrumb navigation, representing the current navigation node along with its path.
- A template for the site map, representing the overall navigation hierarchy, and usually highlighting the current navigation node.

Navigation templates rely heavily on the NAVIGATION MANAGER (3.2).

Unlike page and navigation templates, templates for content elements operate on the attributes of genuine content artefacts. Typically, each content type requires the following templates:

- Templates for the full view of a content element. There is a long list of different full views that may become necessary: views for special output media, text-only views and so on. Sometimes separate templates for different layouts can make sense, such as a 1-column versus a 2-column layout.
- Templates for teaser views such as those required for list items, as in search results. You may have to provide a combined teaser text-only view in addition if this is what accessibility demands. Media-specific teaser views are less common but can also become necessary.

Templates for content elements rely heavily on CONTENT SERVICES (3.1).

Finally, there are going to be static templates – templates that don't depend on the current page and its contents:

- Templates for the search box or other interactive elements that have to be embedded into every page.

- Templates for logos, provided that the logos are truly static and their use doesn't depend on the current page.

This may add up to quite a few templates to represent various views of different content types. However, the advantage is that the individual templates will be straightforward.

Only let templates generate markup that describes the logical structure of a page or page element, and leave the definition of page geometry, fonts, font sizes, colours and the like to CSS style sheets. Page templates can include the necessary style sheets, as they are 'aware' of how the full page is supposed to look.

Finally, if you plan to enhance your web pages with SENSIBLE CLIENT-SIDE INTERACTION (1.3), you need to deliver the necessary client code to the browser. More specifically, the templates for individual content elements will have to include the necessary JavaScript code in the markup they generate.

Example resolved

Almost all content types for the House of Effects website require more than one view. Figure 28 shows a UML sketch that summarises the necessary templates for each content type. Templates are represented as methods of a content type.

The pseudo content type *NavigationNode* holds the full page templates and the navigation templates. Because the overall page layout depends on whether the page is a plain article, announcement, presentation or item description, different page templates become necessary. The navigation templates are fairly standard: we need a template for the tree navigation and one for the breadcrumb. We also assign the static templates for rendering the logo and the language icons to *NavigationNode*.[10]

The templates for content elements are assigned to individual content types. The main content types each require a template for the full view and one for the teaser view that's used for search results. In addition, the two announcement types require an additional teaser view for displaying calendar events.

What about a printed version? This is easy, as we let the page templates include CSS sheets for both screen and print and let the browser decide which should be used. Printed pages can hide unwanted interactive elements, or otherwise make changes to page geometry, without additional templates becoming necessary.

What about a text-only version? We currently have no such requirement, but a text-only version would be straightforward to add. Another set of templates would become necessary, but could be integrated smoothly into our overall SYSTEM OF INTERACTING TEMPLATES (3.4).

[10] As static templates aren't associated with any content type, we've chosen this solution for reasons of simplicity, although static templates don't rely on any *NavigationNode* attributes.

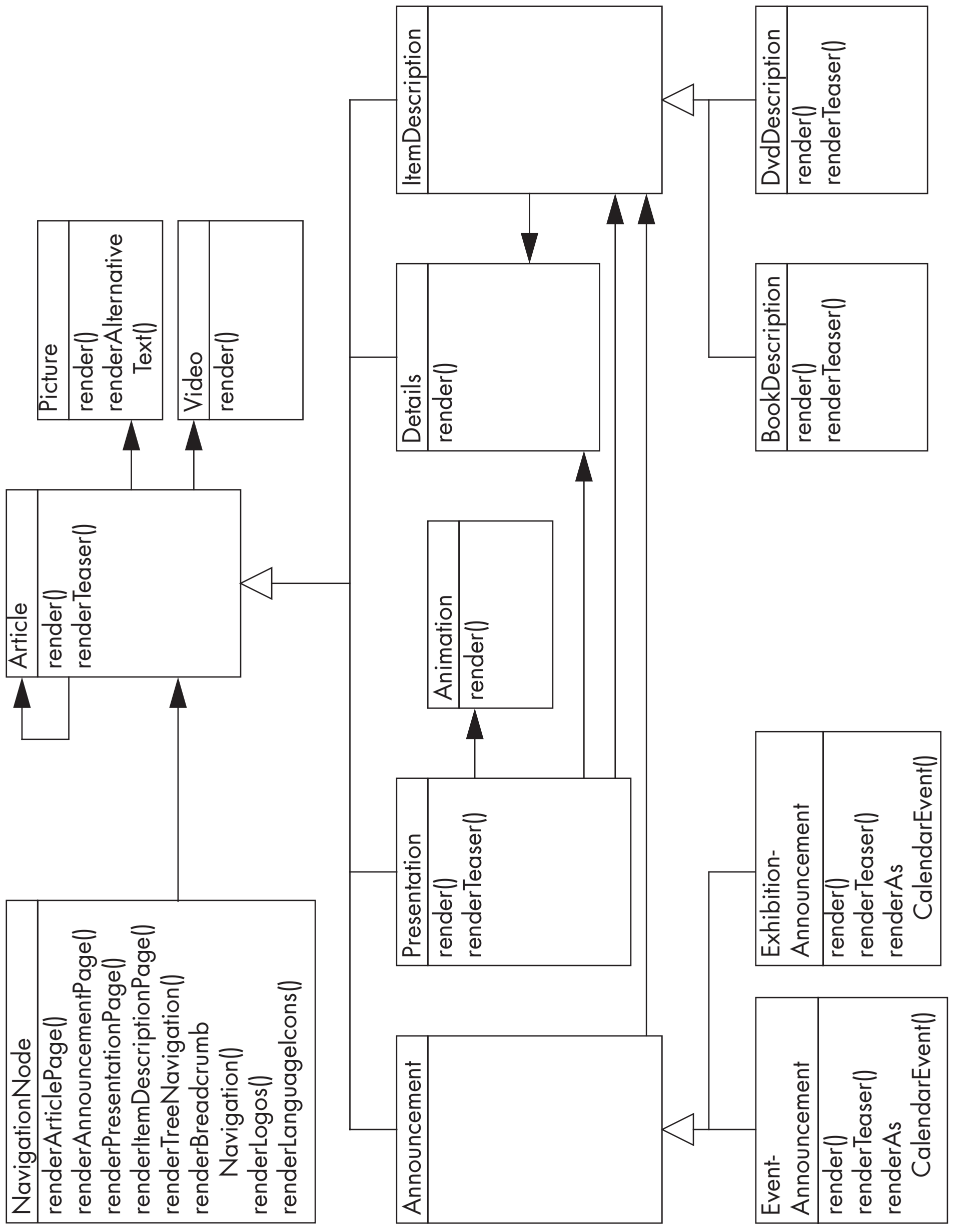


Figure 28: Content types and their templates

Benefits

+ Maintainability is greatly improved, as each template is relatively simple. There is no need for templates to embody layout variations, let alone implement any domain logic. If you've ever seen templates that lack this simplicity, you can gauge the benefit.
+ It's easy to provide the entire site with a consistent layout for fonts, font sizes, colour schemes, page geometry and so on. As layout definitions are concentrated in the styles sheets that the page templates use, small layout modifications can be implemented with little effort.
+ Usability and accessibility are improved, as adding views tailored towards specific user groups is straightforward. You can support different media (such as screen and print) and different output channels (such as hand-held computers and speech or Braille output devices).

Liabilities

– Because you introduce a template for each view, the overall number of templates increases. This isn't a serious problem, and clearly preferable to a small number of complex templates, but the need to organise the templates nevertheless arises.
– The solution introduces a certain degree of browser dependency. Some browsers are better than others at handling media assignments in CSS style sheets. While most browsers can handle 'screen' and 'print', 'hand-held', 'aural' and 'Braille' aren't always supported.
– The solution mentions the inclusion of client code (JavaScript) in the markup that the templates generate. It is straightforward to use components from Ajax libraries here. However, templates must use Ajax components sensibly to generate what is supposed to become SELF-CONTAINED PAGES (3.6) – pages that are sufficiently interactive to be useful.

3.6 Self-Contained Pages

Context

You are in the process of implementing a SYSTEM OF INTERACTING TEMPLATES (3.4). More specifically, you develop a TEMPLATE PER VIEW (3.5) for each content type from your content model.

As your site will feature SENSIBLE CLIENT-SIDE INTERACTION (1.3), a certain amount of client code becomes necessary. Your templates will have to include this client code in the output they generate. Client code can then be sent to the browser as part of the web pages that are delivered.

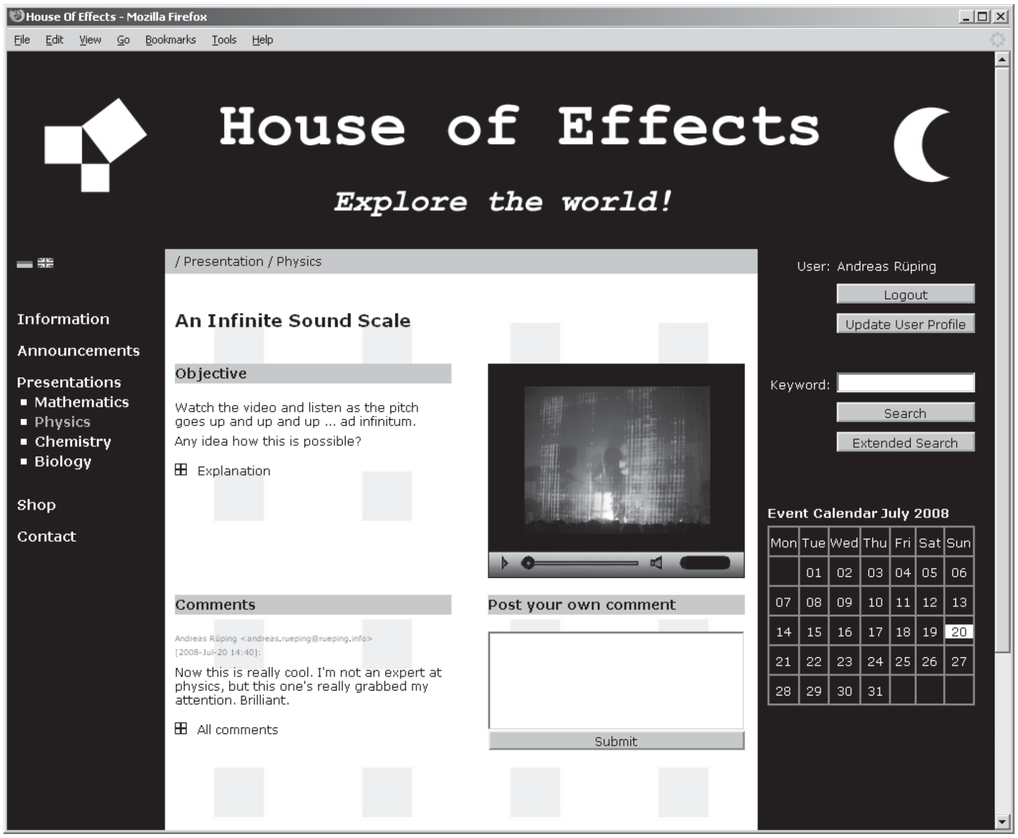
Problem

How much client-side interaction is sensible in terms of usability? How can you deliver the required client-side components to the browser?

Example

The website for the House of Effects is very interactive, so we will be using Ajax occasionally. We've already decided to use Ajax for the filtering and sorting that we'd like to apply to search results and our event calendar.

There will be more examples of Ajax throughout our website. Figure 29 shows an online presentation that includes a video that users should be able to start, stop and restart. The plan is to use a component from an Ajax library for this video object and so get the client code for the video for nothing.



An Infinite Sound Scale

Objective

Watch the video and listen as the pitch goes up and up and up ... ad infinitum.

Any idea how this is possible?

Comments

Andreas Rüping <andreas.rueping@rueping.info>
[2008-Jul-20 14:40]:

Now this is really cool. I'm not an expert at physics, but this one's really grabbed my attention. Brilliant.

Post your own comment

Event Calendar July 2008

Mon	Tue	Wed	Thu	Fri	Sat	Sun
	01	02	03	04	05	06
07	08	09	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Figure 29: Online presentation with video object

The question remains of whether we should use Ajax for other page elements as well, such as background information and lists of comments. Should we?

Forces

With the growing popularity of Ajax technology, more and more web pages feature multimedia objects such as videos that require a certain amount of interaction. Integrating interactive elements is relatively easy, since many prefabricated components are offered by various Ajax libraries. Several Ajax libraries are available commercially, some are open source (www.icefaces.org, labs.jboss.org/jbossrichfaces).

Interactive or multimedia objects introduce an additional aspect into page ergonomics that is relatively unknown from traditional web pages: time. The following example can explain this phenomenon. Imagine a web page that features a video along with its necessary controls. To enjoy the video properly, the average user wants to stay on the page for some time. Simultaneously, the user might want to study information related to the video that the website may offer. However, if looking up this related information leads the user to a different page via HTTP links, the video is interrupted. This effect is clearly disturbing if the user wants to continue watching the video.

You have to be aware of this effect when you design the interaction model for a web page, especially when you have to decide between an HTTP link that points to another page and an Ajax-based event that is handled without leaving the page. Moreover, such a decision refers to a mere link, but to make it properly, you have to take into account what other elements exist on that page and what their requirements are with regard to page visit duration.

Solution

Ensure that you create self-contained pages – pages that are sufficiently rich with content that visitors don't have to leave a page to look up closely related information. This is especially important for pages that feature multimedia objects that require a specific time to be viewed. Use Ajax components to implement the necessary interaction on these pages and integrate the client code into your templates.

Let's look at this in a little more detail. To come up with self-contained pages, you need to take the following steps:

- The first thing to do is to analyse each page type and work out what the average page visit duration is – the typical time an average user will spend on that page to appreciate it fully. If a page includes an animation, a sound file, a video or any other multimedia object with a specific duration, that object probably determines the average visit duration.
- The average visit duration tells you how long users will probably stay on a page. The next step is to determine what related content users might want to see during

that time without leaving the page. This may be hard to predict, but at least you can make an educated guess.

- Design the full page so that the content you have identified is embedded into the page such that it can be accessed with client-side event handling alone. Plan to optimise the presentation, for example through hide and show mechanisms.

The design of self-contained pages lies on the boundary between web page design and template programming. Obviously it forms part of the tasks that are usually performed by a web designer, rather than a software developer or software designer, but it clearly also affects the client-side functionality with its underlying interaction models. It is therefore something you have to do in the overall context of the design of a SYSTEM OF INTERACTING TEMPLATES (3.4).

The implementation of the necessary client-side functionality is straightforward if you use an Ajax library or framework that allows you to integrate prefabricated interactive components into the pages and page elements your custom templates create.

Example resolved

The video on the web page from Figure 29 requires a few minutes of visitors' attention to be useful. Explaining a specific sound effect, this presentation wouldn't make much sense if visitors needed to move on to other pages while the video was still showing. To allow users to watch the video uninterrupted, we plan to embed all related information directly on the page, especially detailed explanation that visitors might want to read while the video is playing.

Putting all related information directly on the page would be excessive – the page would suffer from information overflow. We therefore choose to use interactive boxes that can show or hide specific text: the text blocks *Explanations* and *All Comments* in this case. The event handling that opens and closes these boxes is implemented using Ajax, so no server communication is necessary. Users can keep watching the video while reading the explanation details or browsing through the list of comments that other users have left.

We use an Ajax framework that provides us with the necessary interactive elements, such as expandible text blocks with show/hide buttons, along with their underlying functionality and event handling. All that remains for us to do is to integrate these Ajax components into the full view template for presentations (*Presentation.render* from Figure 28).

Benefits

+ The usability of your site is increased, as requirements for page visit duration are taken into account. With self-contained pages, users stand a much better chance of enjoying multimedia elements properly, as there will be no need to follow links to other pages to look up related information. Even if no multimedia objects are involved, however, usability is still improved, as related information is readily available without the need to travel between pages.
+ When you integrate Ajax components into your custom templates, you reuse client-side functionality throughout your site. This has two advantages. First, this approach benefits from reduced effort when compared to developing all client-side components in JavaScript yourself. Second, the use of a standard library facilitates a consistent layout throughout your site.

Liabilities

– If you use an Ajax library or framework, you have to live with the constraints it may impose on your own design. Details vary depending on what framework you use, but in any case you must expect to expend some effort on integrating the framework components with your own templates.

– The solution invites a tendency towards an increased use of Ajax. There are probably various sorts of information that you can think of as being in some way related to the main topic of an individual page, so embedding this information into the page using Ajax-based event handling might seem like a good idea. This is fine as long as what you implement is still SENSIBLE CLIENT-SIDE INTERACTION (1.3), but you can have too much of a good thing. As there are still drawbacks to Ajax, such as security issues and browser dependencies, you should ensure that you do not introduce too many Ajax components.

CHAPTER

4

Personalisation and User Participation

Until now this book has dealt with the management and the delivery of web content in general. It's now time to add more spice to the content we're dealing with. When you look at websites around the world, you will notice that there's an increasing demand to present personalised content – content that is tailored specifically to a user's or a user group's interests and preferences.

But it is not just personalisation that is becoming more and more important. In the Collaborative Web, many web platforms these days make user participation an integral part of their domain model. Wikipedia is perhaps the most prominent example (www.wikipedia.org); Amazon, with ratings and reviews provided by customers, is another (www.amazon.com). Many platforms allow users to contribute their own content and share it with others. Users can communicate with each other and rate content that others have contributed. They can tag content, allowing tag clouds or other folksonomies to evolve. Everybody can get involved, and the roles of users and editors seem to fuse.

In some respects, personalised and user-generated content isn't that different from traditional web content. The patterns from the previous chapters still apply, even in the

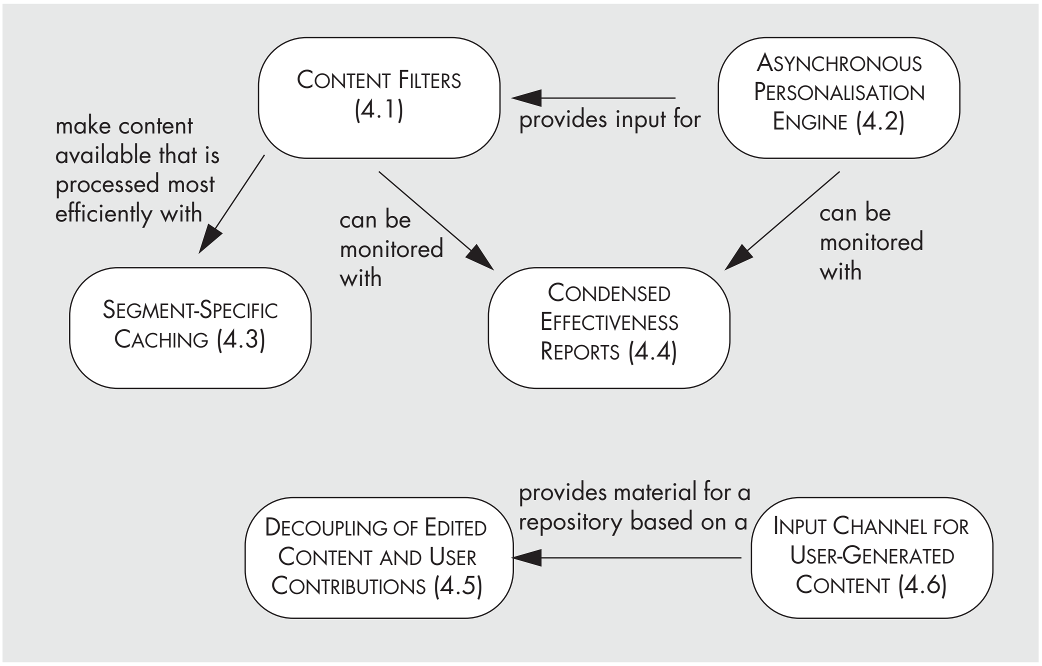

Content Filters (4.1)
Asynchronous Personalisation Engine (4.2)
provides input for
make content available that is processed most efficiently with
can be monitored with
can be monitored with
Segment-Specific Caching (4.3)
Condensed Effectiveness Reports (4.4)
Decoupling of Edited Content and User Contributions (4.5)
provides material for a repository based on a
Input Channel for User-Generated Content (4.6)

Figure 30: Road map to the patterns

presence of personalisation and user participation. But there are additional requirements and challenges that I'm going to address in this chapter:

- How can you implement personalisation strategies?
- Personalisation is known to be the enemy of performance. How can you handle this challenge and make sure that your content delivery software still runs efficiently?
- How can you monitor the effects of your personalisation strategies?
- How can you deal with user-generated content? How should it be related to edited content?

I'd also like to mention what's not in the focus of this chapter. First, this chapter doesn't contain any patterns of groupware applications such as forums or chatrooms. It's not about user collaboration processes either. The focus of this book is on web content, and therefore this chapter will cover aspects of user-contributed content. However, the models, techniques, processes and strategies that underlie groupware applications are outside its scope. Till Schümmer and Stephan Lukosch cover that area extensively in their *Patterns on Computer-Mediated Interaction* (Schümmer Lukosch 2007).

Second, this chapter does not focus on system security. Security is certainly an issue when you develop software architectures for the web, as you must protect data from being compromised or misused. This is even more true if you're dealing with highly sensitive data in the context of personalisation or user participation. While the patterns in this chapter cover security aspects occasionally, security is not their main topic. You can find a detailed discussion of security on the web in *Security Patterns* by Markus Schumacher et al. (Schumacher Fernandez-Buglioni Hybertson Buschmann Sommerlad 2006).

Figure 30 presents the road map to the patterns in this chapter. We begin with several patterns that deal with personalisation and conclude with different aspects of user participation.

4.1 Content Filters

Context

You plan to add personalised content to your website and tailor its appearance to the preferences of your visitors. Content editors should be able to define and implement personalisation strategies.

Problem

How can you let content editors define personalisation strategies?

Example

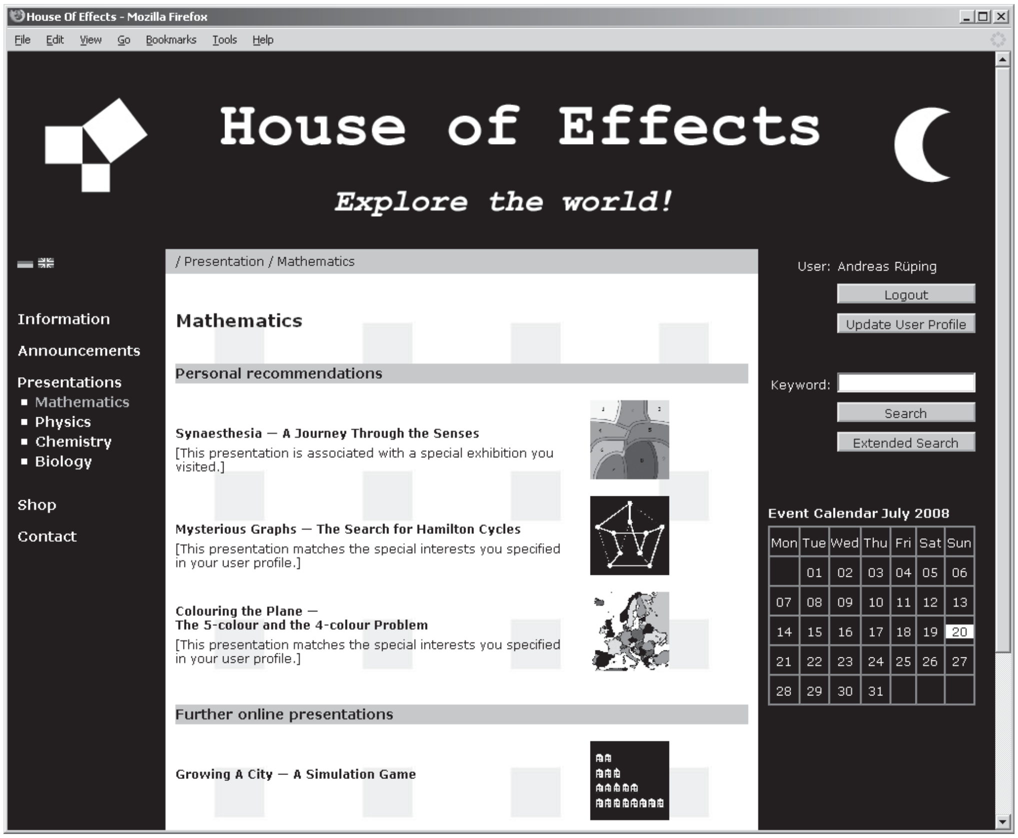
The website for the House of Effects allows users to register. Registered users can buy items from the online shop, can make ticket reservations for special events, and in some cases are given access to special online presentations. There are also going to be premium users – users who are frequent and regular visitors of the real-world museum. As a 'thank you' such users are granted access to a few online simulation games that are not otherwise publicly available.

The idea is to exploit our knowledge of registered users to personalise the website. We'd like to direct them to those exhibitions, events, presentations and shop items that we expect them to find particularly interesting.

The place to do this is will be the start pages of the different sections of our navigation hierarchy, such as the start page for all announcements, or the start page for the online shop. Figure 31 shows the start page for online presentations on mathematics, which begins with a list of presentations tailored specifically to the current user. The page should of course look different for another user, with different interests and a different profile.

Forces

People differ: different users are interested in different kinds of content. Many of today's websites therefore try to personalise the content they present and tailor it to individual users (Rosenfeld Morville 2006).



/ Presentation / Mathematics

User: Andreas Rüping

Mathematics

Personal recommendations

Synaesthesia — A Journey Through the Senses
[This presentation is associated with a special exhibition you visited.]

Mysterious Graphs — The Search for Hamilton Cycles
[This presentation matches the special interests you specified in your user profile.]

Colouring the Plane — The 5-colour and the 4-colour Problem
[This presentation matches the special interests you specified in your user profile.]

Further online presentations

Growing A City — A Simulation Game

Event Calendar July 2008

Mon	Tue	Wed	Thu	Fri	Sat	Sun
	01	02	03	04	05	06
07	08	09	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Figure 31: Personal recommendations

Personalisation can take different forms. The most common way to personalise a site is to make specific content available to some users and hide it from others. Various criteria enable you to decide who may see what, but usually user preferences or attributes from user profiles are evaluated and matched against specific content attributes.

Another form of personalisation doesn't make content elements visible or invisible, but affects the way in which content is placed on a page. List items or teasers that a user is expected to find interesting are placed more prominently, typically at the top of a page.

Ultimately it is the content editors who must decide what content is going to be personalised, and how. Content editors must define personalisation strategies that can be applied to the pages of your site. They also must be able to define these personalisation strategies in a way that is comfortable for them, preferably with a tool with which they are familiar. As content editors have to work with the content management system anyway, it should be possible for them to define personalisation strategies within this system.

Solution

Implement filters for content selection. Integrate these filters into the domain logic for your site: implement services that apply these filters to deliver content only if it matches an individual user's profile. Allow content editors to express personalisation strategies by configuring how the content filters work in detail.

Content filters check content elements against user profiles and allow content elements to pass only if specific criteria are met. Using content filters, your domain logic can express personalisation strategies such as the following: a content element should be included only if the user belongs to a specific customer segment, or a content element should be included only if the user's business value exceeds a specific threshold. Customer segments, threshold values and so on must be defined by the content editors, who can fine-tune personalisation strategies in this way.

From a technical point of view, content filters operate on lists of content elements and take two kinds of parameters. First, there are arbitrary characteristics of the user's profile.[11] Possible examples include:

- The user's membership of a specific customer segment.
- Tracked behaviour, for example with regard to navigation.
- The user's purchase record.
- Attributes from an online profile the user may maintain.

Second, each content filter requires a set of configuration parameters that have to be provided by the content editors. Examples include but are not limited to the following:

- Specification of customer segments.
- Threshold values.
- Maximum number of results.
- Default content (in case no matching content elements are found).

In addition to the actual filtering logic, content filters sometimes apply a sorting order on the content elements they pass. Usually filters return matches in descending order of relevance, running from the closest matches down to the weakest. As an alternative, filters can return content elements along with a value that measures their relevance so that other components can bring the results into the desired order.

Once you have implemented the content filters you need, you can integrate them into the CONTENT SERVICES (3.1) that run on your content delivery server, as Figure 32 shows. The integration of filters is common for services that return lists of content elements. These lists will look different from one user to another. You can also integrate filters into services that return individual content elements, but this makes sense only if specific

[11] The term 'user profile' refers to all user-related data that can be used for personalisation purposes. This includes both data users may have provided themselves (e.g. through an online profile) and data that can be deduced from user behaviour.

content elements have to be blocked from some users (in which case the service wouldn't return the content element in question, but no element at all).

Similarly, you should include your content filters in the services offered by the NAVIGATION MANAGER (3.2) and the SEARCH MANAGER (3.3), to ensure that unwanted elements are removed from navigation structures and search results for a specific user.

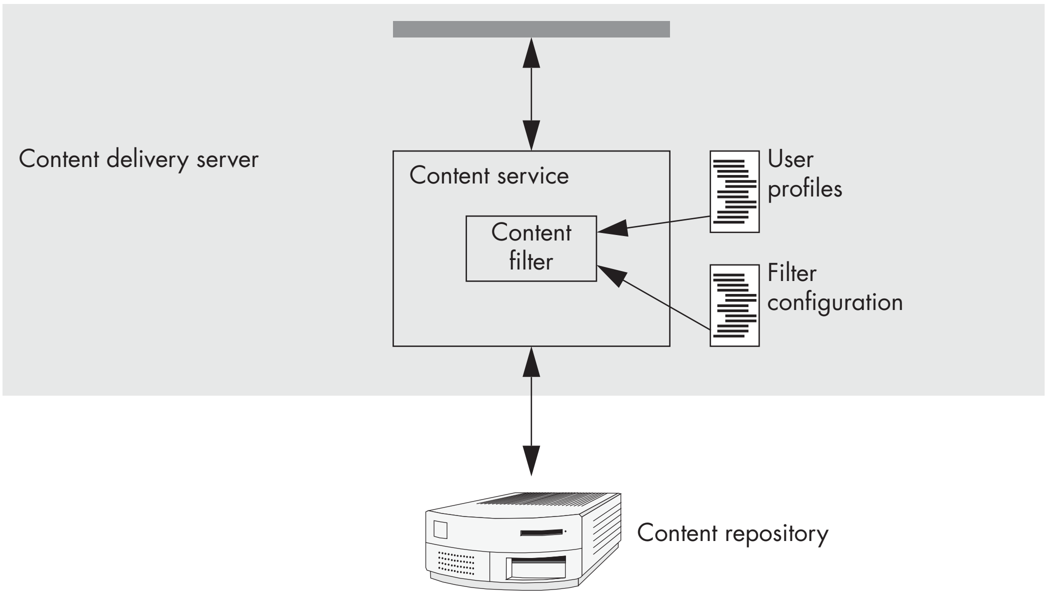


Figure 32: Integration of a content filter into a content service

The components involved in a personalised content filter interact as follows:

- The user profile is loaded into the HTTP session on login.
- The server receives an HTTP request and calls a content service.
- The content service receives candidate content elements from the repository and applies one or more filters. Details depend on the filter configuration.
- The content service makes the personalised results available to the presentation layer.
- The presentation layer uses the results and makes exactly those content elements available that the user should see. Content elements that are likely to meet the user's interest will be placed prominently to attract the user's attention.

The processes for services offered by the NAVIGATION MANAGER (3.2) and the SEARCH MANAGER (3.3) are similar.

Finally let's again take the content editors' perspective. Content editors are free to fine-tune content filters by changing the filter configuration. This allows them to influence the content individual users get to see. The question that remains is how content editors can actually provide a configuration. If you use a dedicated personalisation engine (either stand-alone or as part of your content management system) than it will probably offer a built-in mechanism for filter configuration.

However, if you implement personalisation through custom components alone, you need to come up with your own solution. A straightforward mechanism is to introduce special configuration objects into your content model as pseudo types. This enables content editors to use the content management application to specify the necessary filter configurations.

Example resolved

We implement an initial filter for online presentations. This filter lets online presentations pass if the current user is allowed to see them. The decision is based on the segment to which the user belongs: anonymous users, registered users or premium users, where content editors define the criteria for premium users. The filter must be able to operate on both *Presentation* objects (the content elements representing online presentations) and links to them. We integrate this filter into all CONTENT SERVICES (3.1) that may return a single presentation (*ArticleService*, *PresentationService*), into the service that the SEARCH MANAGER (3.3) relies on (*ArticleListService*) and into the functions offered by the NAVIGATION MANAGER (3.2) (*getPath*, *getTree*, *getAllReferrers*, *getDefaultNode*). If a user doesn't belong to a segment that can view a specific presentation, no trace of it will be visible.

Two other filters are required for the type of personalisation shown in Figure 31. The first takes a list of articles and returns those that match a topic the current user has rated as sufficiently interesting in a self-maintained online profile. Content editors can define what 'sufficiently' means by specifying a threshold value. The second filter also operates on lists of articles, but returns those that are related to an event at the House of Effects for which the user has bought tickets in the online shop. Content editors can specify the time span that has to be considered, as well as the maximum number of results for either filter. Figure 31 shows how both filters can be combined to generate a highly personalised page.

The following table summarises the content filters we need.

FILTER TYPE	SIGNIFICANT USER PROFILE ATTRIBUTES	FUNCTIONALITY	CONFIGURATION
SegmentBased PresentationFilter	segment assignment	Includes an online presentation (or a link to one) only if the user belongs to a segment that is allowed to view the presentation.	Segment specification
PersonalInterest Filter	user preferences	Includes an article on a specific topic only if the interest the user has expressed in this topic exceeds a threshold value.	Interest rating threshold Maximum number of results
EventAttendance Filter	event ticket sales	Includes an article only if it is directly associated with an event for which the user has bought tickets in the online shop during the past couple of months.	Exact time span Maximum number of results

Since we plan to implement personalisation with our own custom components, we have to offer a mechanism that allows content editors to configure and fine-tune the different filters. We add three types to our CONTENT TYPE HIERARCHY (2.1): *SegmentBasedPresentationFilter*, *PersonalInterestFilter* and *EventAttendanceFilter*. We then create a singleton instance for each of these types, which gives us three special content elements that serve as configuration objects within the content repository. The attributes of these configuration objects store the data necessary for filter configuration. This enables content editors to use the standard content management client to maintain the different configurations for each type of filter. Content filters access the configuration objects and operate accordingly.

Benefits

+ Content filters allow you to create a website with content tailored to individual users, taking their interests and preferences into account. If personalisation is applied sensibly, users will appreciate content that is particularly useful to them. Your site can also gain attention and attract more visitors as a consequence of personalisation.
+ Because content editors can fine-tune the content filters, they are to free to implement any personalisation strategies they want. Obviously there are limits to what a specific filter can do, but if its configuration is well-chosen a filter can easily be flexible enough to be adapted to changing business goals.

+ Content editors can specify personalisation strategies using tools they are familiar with anyway. Either your content management system or a specific personalisation engine offer the necessary mechanisms, or the content editors can use the standard content management client to make filter configurations with the pseudo content objects that you have added.
+ Since you integrate your filters into your CONTENT SERVICES (3.1), no aspect of personalisation shows up in the presentation layer, which contributes to the separation of concerns. In particular, your keep your templates simple and straightforward, as they won't be affected by personalisation at all.

Liabilities

– The solution tacitly assumes that user profiles are readily available. This is fair enough as long as mere attributes from a user-maintained online profile are concerned. However, content filters often rely on user data that isn't so obvious. What navigation or purchasing behaviour does a user exhibit? To which customer segment should a user belong? Evaluations like these are known as *implicit personalisation* and can involve algorithms that are too complex to be performed efficiently online. An ASYNCHRONOUS PERSONALISATION ENGINE (4.2) can help.

– Content services that integrate personalisation cannot apply caching, as their results depend on the current user. Templates that rely on personalised services cannot apply caching to the HTML output they generate either: it's a well-known fact that severe performance problems can result. The problem is not so severe for page elements that vary only across larger user groups rather than appearing differently for every individual user. You can apply SEGMENT-SPECIFIC CACHING (4.3) in this case.

– Complexity increases as you introduce personalisation. If you develop custom components you must ensure that you do not introduce too much complexity, as complex filters can be difficult to understand. If you use a third-party product for a personalisation engine, you must ensure that it integrates smoothly into your overall architecture.

4.2 Asynchronous Personalisation Engine

Context

You plan to apply CONTENT FILTERS (4.1) or other forms of personalisation that make content elements available if they match the current user's profile. What this means in detail depends on the personalisation strategies you wish to implement: different kinds of criteria are possible.

Problem

How can you implement personalisation strategies in an efficient way?

Example

Implementing personalisation strategies efficiently can be difficult, and the website for the House of Effects is no exception. To apply CONTENT FILTERS (4.1) properly, we have to know what event tickets a user has bought from the online shop during the last few months, so we need to access the database that stores all shop transactions. We also have to know whether or not a registered user is a premium user, which involves an evaluation of the user's long-term history of purchases from the online shop, the user's attendance of special events in the past and possible donations the user may have made.

Such factors make it necessary to retrieve data from different sources and to apply non-trivial algorithms. However, the website must remain responsive despite the personalisation requirements.

Forces

Personalisation always relies on user profiles – data that users provide themselves or data that is deduced from available sources. Known as *implicit personalisation*, the latter case comprises various and quite diverse scenarios. For example, a user's customer status can be calculated on the basis of previous purchasing behaviour. Recommendations can be based on the user's navigation habits, provided they have been monitored and recorded. Recommendations can also be calculated based on the interests of customers with similar profiles.

All these scenarios have in common the fact that they rely on information from different data sources. In other words, to come up with a sophisticated concept for implicit personalisation you often have to access various data sources, which means making calls to remote systems.

Moreover, strategies for implicit personalisation can be complex in their underlying algorithms. For example, imagine an algorithm that makes an educated guess about the content in which a user might be interested. It is quite clear that such an algorithm is far from trivial and in many cases will require a rule-based system for its implementation.

The consequence is that in many cases personalisation requires substantial computation time. This, however, is directly opposed to the performance requirements that most websites face, personalised or not. No degree of personalisation will serve as an excuse for a website being slow: if a site is too slow to be useful, users will simply ignore it.

Solution

Identify the complex algorithms necessary for assigning content to individual users or user groups. Take these algorithms off line to a dedicated personalisation engine that runs asynchronously and independently from the application server that handles HTTP requests.

Running asynchronously, these algorithms can update user profiles at regular intervals. Implementing advanced personalisation strategies, the algorithms can use data from different sources, such as the following:

- The user's navigation patterns.
- The user's shopping record and purchasing behaviour.
- Demographic data: the user's age, sex, profession, place of residence etc.
- User-maintained personal preferences.

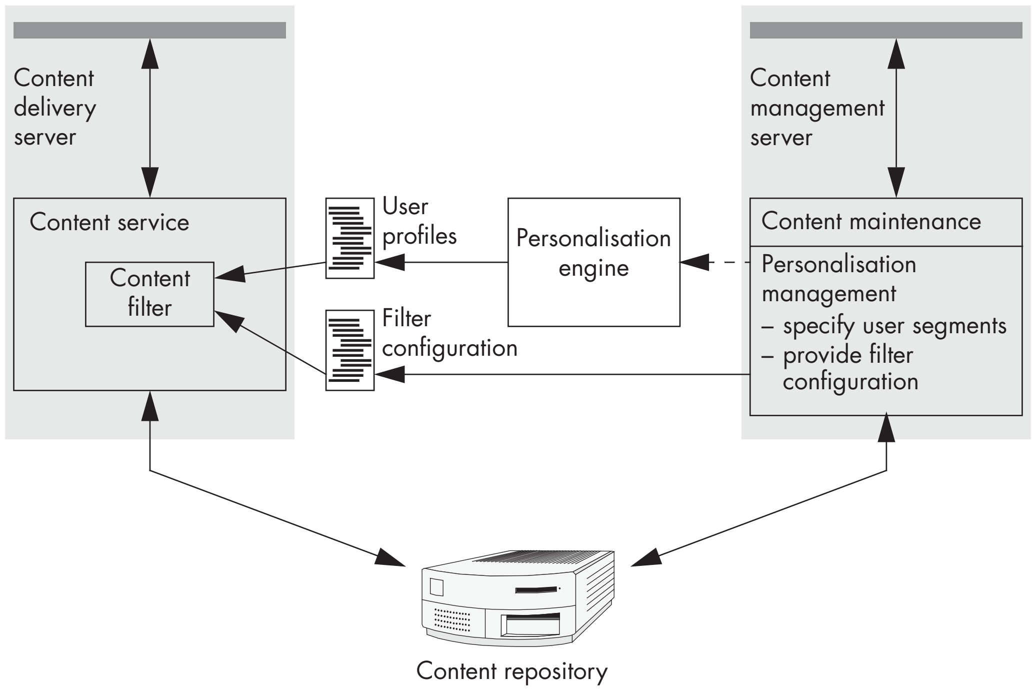


Figure 33: Personalised content services configured by an asynchronous personalisation engine

The entirety of personalisation algorithms hosted by the personalisation engine generate the sort of user profile data that the CONTENT FILTERS (4.1) require as input parameters. The concept originally sketched in Figure 32 evolves into the architecture shown in Figure 33.

If your content management system includes a built-in personalisation engine, or if you use a stand-alone tool, that engine will probably allow you to execute complex personalisation algorithms asynchronously. You also need to register the algorithms you have implemented and make the necessary configurations.

However, if you develop your own personalisation components, you must set up your personalisation engine as a distinct application running as a separate process, independent of the content management and content delivery applications. If content editors have to fine-tune the personalisation engine (which they probably will), you can introduce pseudo content types for configuration purposes (similar to those for the configuration of CONTENT FILTERS (4.1)) and let content editors maintain the configuration objects via the content management application. You can apply LISTENER-BASED SYNCHRONISATION (1.4) to notify the personalisation engine of any changes to the configuration objects in the content repository.

Example resolved

Personalisation for the House of Effects website requires a few algorithms that we do not want to perform online. The first is the collection of content elements that are related to events the user has attended in the past – content elements that should now be presented to the user as a personal recommendation. The second is the assignment of users to user segments, especially the assignment of premium user status.

Although these algorithms aren't extremely complex, they need to retrieve data from different sources, including the shop database. Executing these algorithms involves remote calls and the evaluation of different data sources, which might take time. For reasons of efficiency, we don't directly integrate them into the CONTENT FILTERS (4.1) that are applied when a content service is called, but take them off line to what is to become our custom personalisation engine – an asynchronous process running nightly, updating user profiles, especially for status and content recommendations. Nightly updates are fine, as there are no requirements to update user recommendations or premium content visibility immediately after the user has made a purchase or has qualified for a specific status: a delay of a few hours is perfectly acceptable.

Our personalisation algorithms rely on a few configuration parameters, such as the definition of user segments. These are stored in the content repository as special configuration objects and are maintained by the content editors. A notification mechanism informs our personalisation engine of any relevant changes.

Benefits

+ System performance is improved as complex algorithms are taken off line. CONTENT FILTERS (4.1) work much more efficiently if time-consuming computations are performed asynchronously.

+ Because the personalisation engine is a separate and asynchronous process, personalisation algorithms become possible that otherwise couldn't even be considered. Your personalisation strategies can become more advanced and more powerful, taking more and more different parameters into account. Advanced implicit personalisation becomes possible.

Liabilities

– Since the personalisation engine runs asynchronously, user profile data isn't calculated at the same moment as the user requests a web page. The consequence is that changes to the user's profile aren't immediately reflected by the page that is delivered. For example, after a user has qualified for specific content, there may be a short delay until that content becomes visible. Similarly, delays are possible when a user is no longer supposed to see specific content. If this unacceptable you may have to implement an invalidation mechanism.

– An asynchronous personalisation engine allows you to implement fairly advanced personalisation strategies, but there's a limit to what personalisation can do. As Louis Rosenfeld and Peter Morville point out in *Information Architecture for the World Wide Web*, 'in many cases, it's really hard to guess what people will want to do or learn or buy tomorrow' (Rosenfeld Morville 2006). Don't get carried away with the vast range of options that a powerful personalisation engine may give you. In most cases it's wise to stick to just a few personalisation mechanisms that you can expect to work well.

– Implicit personalisation can be powerful, but it can quickly go too far. Ensure that you don't compromise your users' privacy. First, there is a legal aspect to this: there are limits to what user data you are allowed to collect. Second, there's a cultural aspect to privacy: most users won't take kindly to a website that monitors everything they do. Be sure to apply personalisation in a way your users can appreciate. One possible strategy is to inform the users explicitly what data is collected and how that data will be used.

4.3 Segment-Specific Caching

Context

You have introduced CONTENT FILTERS (4.1) that allow you to generate pages tailored specifically to the individual visitors of your site.

As an initial step towards preventing performance problems, you have introduced an ASYNCHRONOUS PERSONALISATION ENGINE (4.2) that performs the complex algorithms involved in advanced personalisation strategies. This might not solve all performance problems, though.

Problem

How can you avoid efficiency problems with content delivery in the presence of personalisation?

Example

The introduction of personalisation can easily impair system performance. This is as true for the House of Effects website as it is for virtually any other. Despite our efforts to take complex algorithms off line to our ASYNCHRONOUS PERSONALISATION ENGINE (4.2), we still have to be careful not to introduce performance problems.

The reason for this is that several of our pages and page fragments cannot be cached. This applies to start pages for exhibitions, events, presentations and shop items, which will look different depending on the recommendations for the current user. It also applies to navigation elements and some articles, which may or may not include links to specific presentations, depending on whether the current user is anonymous, registered (and logged in) or premium.

Switching off caching for all these elements would be unfortunate, though, as it could slow down the site considerably.

Forces

Personalisation is sometimes said to be the natural enemy of system performance. This is due not so much to the complex algorithms that may be involved: you can use an ASYNCHRONOUS PERSONALISATION ENGINE (4.2) to solve that problem. However, personalisation makes caching largely impossible. If a content service returns different content elements for every user, its results cannot be cached. If the appearance of a page element depends on the current user, that page element cannot be cached either. In other words, caching is disabled at different levels of granularity – both on the logical and the template layer of our LAYERED ARCHITECTURE FOR CONTENT DELIVERY (1.5).

Still, caching is a paramount factor for a website's performance (assuming pages are generated dynamically). The difference in response time between a page that is cached and one that isn't can be significant, which is why so many content management systems offer sophisticated caching strategies. Unfortunately, personalisation threatens to render these caching strategies useless.

The degree of personalisation varies greatly. One content element might be specific for individual users, while others might be specific only for specific user segments: groups of users whose profiles have things in common. In the latter case you can benefit from the fact that only limited variations of a page or page element exist provided that the number of user segments is not large.

Solution

Identify content elements that are specific to user segments rather than individual users. If the number of segments is not large, implement segment-specific content services and segment-specific templates, and apply caching to the resulting personalised content elements and HTML fragments.

Make a distinction between personalised elements that are specific to individual users and personalised elements that are specific to user segments.

As far as the segment-specific elements are concerned, you can do either of the following, depending on whether you wish to apply caching on the logical layer or the template layer of your server-side architecture:

- Split up all personalised CONTENT SERVICES (3.1) into several segment-specific services. As neither of the segment-specific services takes the current user as a parameter, you can apply caching to the results either of them returns.
- Introduce several segment-specific templates for one personalised page element. What you then get is a TEMPLATE PER VIEW (3.5), taking the specifics of the presentation for each user segment into account. As neither of the segment-specific templates depends on the current user, their output can safely be cached. You can configure the templates accordingly.

As you can combine caching at different levels of abstraction, you may even choose to implement both techniques. All that then remains to do is to implement a small piece of dispatching logic that selects the appropriate content service or template based on the identity of the current user.

Example resolved

We cannot do much for the start pages for exhibitions, events, presentations and shop items. These pages will be different for every user, so there is no way to apply caching. This is not too severe a problem, however, as the actual recommendations are calculated asynchronously and putting the pages together won't take that long.

However, things look different as far as our navigation elements are concerned. These elements don't depend on the individual user, but on the user segment alone. There are no more than three variations: one for anonymous users, one for registered users and one for premium users. We therefore choose to replace the one template for tree navigation we have by three segment-specific templates, as the UML sketch in Figure 34 shows.

Our full pages still cannot be cached as a consequence of personalisation, but the tree navigation elements can. This is fortunate, as the mechanism for calculating tree navigation with its underlying tree traversal algorithm is relatively expensive in terms of execution time. It is much more efficient to calculate three different variations of the tree navigation and store them in the cache than to execute the tree traversal each time a page is requested.

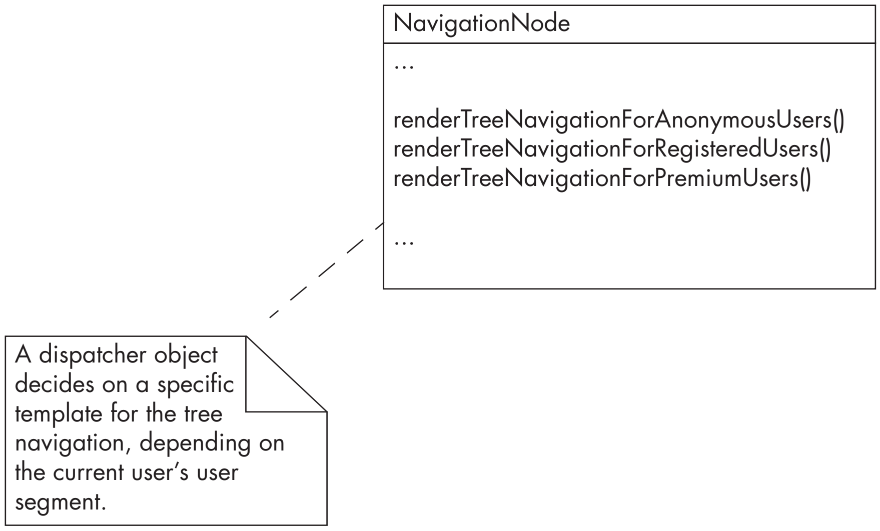

NavigationNode
…
renderTreeNavigationForAnonymousUsers()
renderTreeNavigationForRegisteredUsers()
renderTreeNavigationForPremiumUsers()
…
A dispatcher object decides on a specific template for the tree navigation, depending on the current user's user segment.

Figure 34: Three different templates for tree navigation as seen by three different user segments

Alternatively, we could have decided to keep one template for the tree navigation, but to use three segment-specific services instead. More specifically, we would have to replace the *getTree* function offered by our NAVIGATION MANAGER (3.2) by three functions.

Benefits

+ Segment-specific personalisation is quite common, and sometimes the bulk of the personalised elements rely on the current user's segment rather than the user's identity. You can expect an improvement in system performance that can, at least in some cases, be quite significant.
+ The solution is flexible enough to be applied to caching at different levels of granularity. You can apply it to content or navigation services on the logical layer of your architecture, or to templates on the template layer.

Liabilities

– The number of services and templates increases, which may be accompanied by a slightly increased complexity of the overall system. Actually the solution represents a trade-off: the more services and templates you are willing to accept, the more segment-specific caching you can apply. Clearly there is a limit to the number of different segments you can handle if you want a design that is still clear and maintainable.

– Because you cache objects for every user segment, your servers use more memory than they otherwise would. Specifically, if you introduce a significant number of user segments, you must evaluate the consequences this has on the memory requirements caused by your caching strategies.

4.4 Condensed Effectiveness Reports

Context

Your website offers personalised content to users. Content editors can configure and fine-tune the personalisation strategies that are in effect.

Problem

How can you provide content editors with effective feedback on the success of the personalisation strategies they have implemented?

Example

Content editors of the House of Effects website can use different mechanisms to fine-tune recommendations to users on the start pages for exhibitions, events, presentations and shop items.

For example, they can define the degree of interest a user has to express in a specific topic to be considered. They can define a time span for past events that are analysed when making recommendations. In addition, they are free to specify how the recommendations are sorted and how many will be shown.

The rationale is of course to attract users to presentations, events and shop items that they may find interesting, obviously with the idea of future bookings or sales in mind. The content editors will be interested in finding out how well their strategies work.

Forces

You don't introduce personalisation for personalisation's sake. Personalisation represents a service to the user, offering a content selection based on an educated guess of what the user might find particularly interesting. Obviously personalisation is in many cases introduced for commercial motives as well. Site owners wish to attract users to commercial offers with the hopes of increased revenue.

Given such a scenario, it is clear that content editors want to check whether their personalisation efforts are successful in terms of visitor attraction. This invites the question of how visitor attraction can be measured. Measurements are needed that allows content editors to draw conclusions about the fine-tuning of their personalisation strategies.

An initial idea is to measure how often personalised content is delivered and viewed. However, this approach has shortcomings. First, the fact that a page with personalised content is being viewed says nothing about how well it is received by the user. Second, personalisation is often expressed through lists of teasers with links to full articles, but you can't draw any conclusions about a single teaser from the fact that such a teaser list is requested.

It is more promising to analyse how well personalised links are received, especially those that underlie special teaser elements. Evaluating the success of personalised teasers can give content editors feedback about the effectiveness of their personalisation strategies. However, such an analysis must be presented in a way that is easy to understand: a large volume of statistical data might cause more confusion than insight.

Solution

Measure the effectiveness of personalisation strategies by comparing the *click ratio* of personalised teasers with that of non-personalised ones. The click ratio relates the number of times a teaser's underlying link was followed to the number of times the teaser was displayed. Use a reporting tool to apply metrics and condense the results.

The click ratio appears to be the most meaningful measurement, as it relates the attractiveness of one teaser to the attractiveness of others and so allows you to measure the relative popularity of personalised teasers.

To calculate the click ratio for a specific teaser, you need to take the following steps:

- Count the number of times a teaser is delivered (as part of a page). Every time a personalised teaser is counted, also log the reason why the teaser was chosen, such as what CONTENT FILTERS (4.1) were applied, the current user's user segment, etc.
- Count the number of times the underlying link is followed. To make this possible, you can add a parameter to the link that allows you to trace the request back to that link (and the teaser) when the landing page is generated.

Be careful that your monitoring results aren't obfuscated by your caching strategies. What matters is the number of times a teaser is delivered, not the number of times it is generated. Similarly, you're interested in the number of times a personalised link is followed, not the number of times an element on the landing page is generated. Integrate your monitoring techniques into the mechanisms of HTTP request evaluation and page delivery, not into any page generation mechanisms that might be omitted if caching takes place.

The best way to handle the collected data is to use a tool that can condense the results and present visualisations and meaningful statistics about the effectiveness of your personalisation strategies. Some statistics can easily be established, including the following:

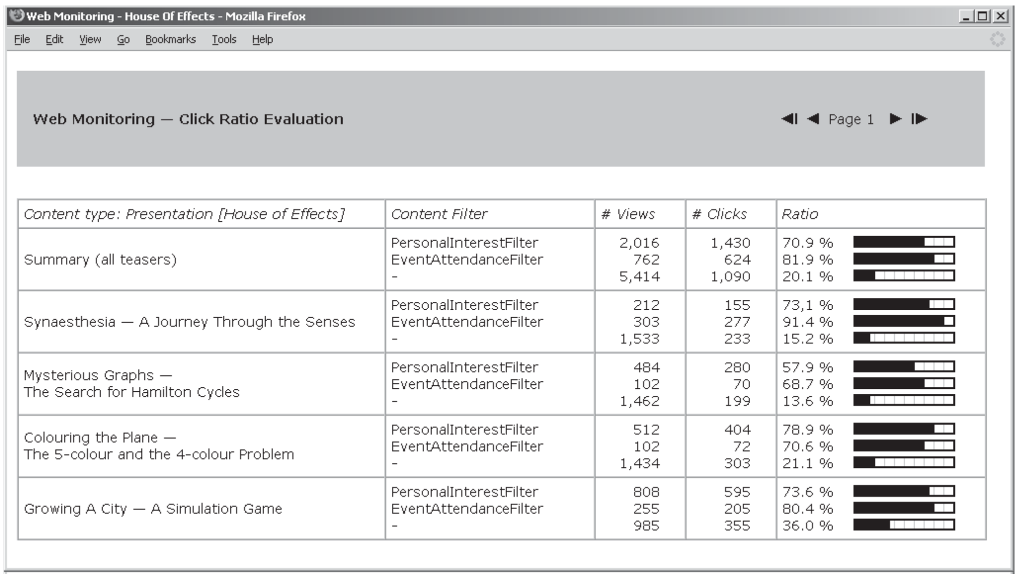
- How often a teaser was delivered.
- Why a teaser was delivered (which personalisation strategy was responsible for its inclusion).
- How successful a teaser was in terms of click ratio.

In many cases a relatively simple tool will be fine. Different tools are available: many of them are web-based tools that allow you to view the results through a web browser. It shouldn't be difficult to find one that suits your needs.

Example resolved

We're interested in finding out about the effectiveness of the two different kinds of filters we apply: the filter based on personal interests specified by the users themselves, and the filter based on events users have attended in the past. We count the number of times that teasers are delivered and break down the data into the different filters that were applied. We also count the number of times a teaser was selected by a user and its underlying link was followed.

A simple tool is sufficient for our purposes. Figure 35 shows the results for *Presentation* teasers. Although this web-based form of personalisation monitoring is simple, it still allows us to draw a few key conclusions about the effectiveness of our personalisation strategies.


Web Monitoring - House Of Effects - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Page 1

Web Monitoring — Click Ratio Evaluation

Content type: Presentation [House of Effects]	Content Filter	# Views	# Clicks	Ratio
Summary (all teasers)	PersonalInterestFilter EventAttendanceFilter -	2,016 762 5,414	1,430 624 1,090	70.9 % 81.9 % 20.1 %
Synaesthesia — A Journey Through the Senses	PersonalInterestFilter EventAttendanceFilter -	212 303 1,533	155 277 233	73,1 % 91.4 % 15.2 %
Mysterious Graphs — The Search for Hamilton Cycles	PersonalInterestFilter EventAttendanceFilter -	484 102 1,462	280 70 199	57.9 % 68.7 % 13.6 %
Colouring the Plane — The 5-colour and the 4-colour Problem	PersonalInterestFilter EventAttendanceFilter -	512 102 1,434	404 72 303	78.9 % 70.6 % 21.1 %
Growing A City — A Simulation Game	PersonalInterestFilter EventAttendanceFilter -	808 255 985	595 205 355	73.6 % 80.4 % 36.0 %

Figure 35: Effectiveness reports for teasers of online presentations

Obviously the *PersonalInterestFilter* is the more important one, as it returns more results than the *EventAttendanceFilter*, at least in most cases. Nonetheless, both filters are effective, as their click ratio (on average for all teasers) is relatively high and clearly better than the click ratio for non-personalised teasers shown in third row for each presentation.

Benefits

+ Content editors are given a tool that allows them to check the usefulness of their personalisation strategies. Because monitoring data is presented in the condensed form of effectiveness reports, content editors quickly get a feel for how well their personalisation strategies work.
+ The monitoring data that is collected and evaluated is accurate and meaningful. As you count teasers that are not just viewed but are actually followed, you receive more valuable and meaningful results.

Liabilities

– You need a reporting tool for visualising the results. This may represent extra cost, and certainly means that you'll have to evaluate possible candidates to find the right tool. Also, an extra tool adds to the overall complexity of your software architecture, if only slightly.

– If a detailed analysis is required, the monitoring data stored in the database can accumulate to significant proportions. Make sure you do not collect more data than you can properly evaluate.

– Personalisation monitoring can slow down the system. You can choose to delegate the actual monitoring (including database access) to an asynchronous process if the extra execution time would otherwise impair system performance.

– Although the collected data is valuable, there is still room for interpretation. For example, effectiveness reports tell you how often a page was viewed, but not for how long. They tell you how often a personalised link was used, but not how much it would have been used had it been a standard, non-personalised link. In other words: content editors should use the feedback they get on their personalisation strategies, but they should use it with care.

4.5 Decoupling of Edited Content and User Contributions

Context

You plan to let users contribute to your website actively and turn it into a collaborative Web 2.0 platform.

Specifically, you are going to invite user-generated content – that is, any kind of content authored and provided by the website users. However, users can submit other forms of contributions as well, including content ratings, tags, etc.

Problem

How can you invite user-generated contributions without disturbing workflow processes for edited content?

Example

Figure 36 shows a page from the House of Effects that invites user contributions. It's a presentation that involves an online game. Users must try to solve a mathematical puzzle as quickly as possible: they are asked to colour the European map, initially with no more than five colours, then with no more than four. The highest scores of the day are stored. Users can also leave comments that may be valuable for other users. Other pages will also include user ratings, for example for shop items.

Comments, ratings and high scores obviously have to be stored somewhere from which they can be retrieved during page generation. It makes sense to think of comments as user-generated content, so it's straightforward to introduce content types for specific user contributions and store these contributions in the content repository.

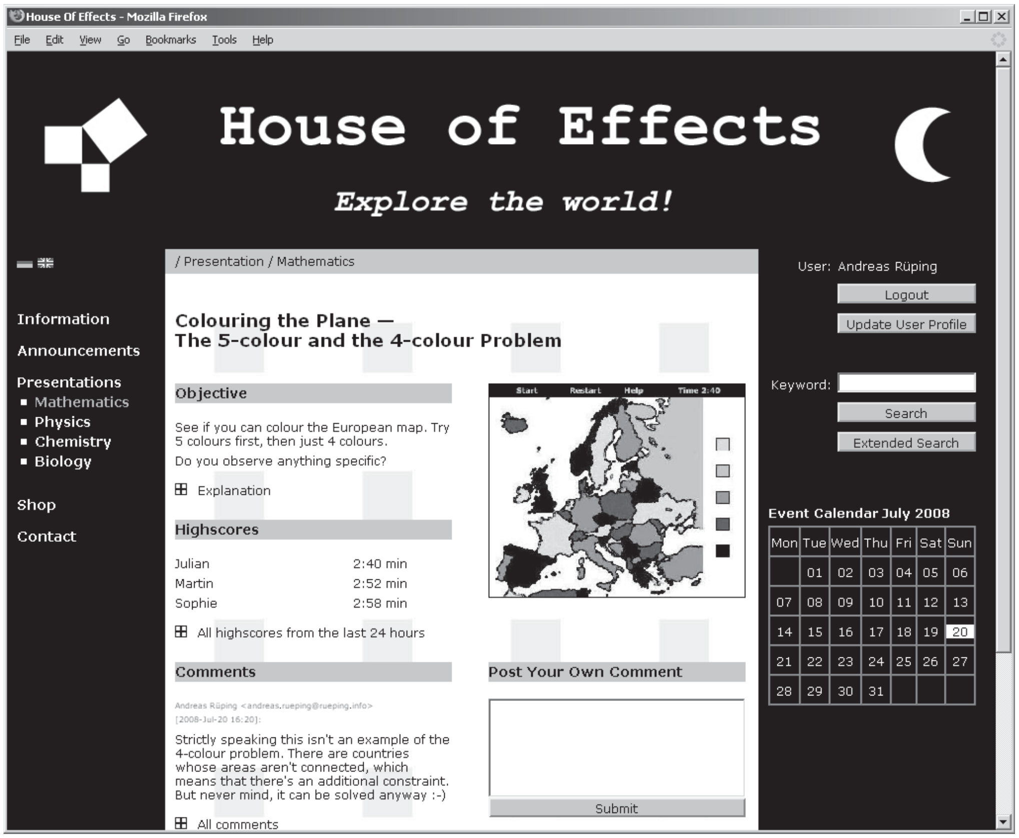
We have to bear in mind though that the workflow we have defined for edited content doesn't apply to user contributions: users will be able to make their submissions without any formal publication process.

Forces

In the age of Web 2.0 user participation has become more and more important. As Clay Shirky points out in *Here Comes Everybody*, 'collaborative production… is considerably harder than simple sharing, but the results can be more profound' (Shirky 2008). Wikis especially enjoy increasing popularity, as they allow users to contribute and collaborate in a very straightforward way (Leuf Cunningham 2001, Mader 2008). Groupware applications allow users to share content within a community (Schümmer Lukosch 2007).

User participation can take different forms. There is user-generated content: content artefacts authored and submitted by the website users. There are folksonomies: taxonomies provided by the website audience, often in the form of tags, as well as user comments and ratings. The conclusion to be drawn is that there is no longer a clear distinction between users and editors.

However, the workflow processes for edited content and user contributions are very different. Edited content undergoes a publication process and usually goes live only after it has been officially approved. User contributions may or may not go through some kind of moderation process, but even if they need editorial approval before going live, their publication process is different and usually less formal.



/ Presentation / Mathematics

Colouring the Plane —
The 5-colour and the 4-colour Problem

Objective

See if you can colour the European map. Try 5 colours first, then just 4 colours.
Do you observe anything specific?

⊞ Explanation

Highscores

Julian	2:40 min
Martin	2:52 min
Sophie	2:58 min

⊞ All highscores from the last 24 hours

Comments

Andreas Rüping <andreas.rueping@rueping.info>
[2008-Jul-20 16:20]:

Strictly speaking this isn't an example of the 4-colour problem. There are countries whose areas aren't connected, which means that there's an additional constraint. But never mind, it can be solved anyway :-)

⊞ All comments

Post Your Own Comment

Event Calendar July 2008

Mon	Tue	Wed	Thu	Fri	Sat	Sun
	01	02	03	04	05	06
07	08	09	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Figure 36: Edited content versus user submissions

From a technical viewpoint, elements of edited content in the content repository have to undergo a state change and need to be updated and re-published whenever any of their attributes change. This is true for all types of attributes defined in the CONTENT TYPE HIERARCHY (2.1), including associations to other content elements. In other words, if a content element keeps a list of references to other content elements and you add a reference, you need to re-publish the element with the additional reference for the change to become effective.

It's obvious that there will be relationships between edited content and user contributions, as user contributions, such as comments and folksonomies, often refer to specific content elements. The workflow for edited content must nevertheless remain undisturbed: no *edited* content must have to re-published as a consequence of any kind of user submission.

Solution

In your overall content model, decouple edited content from user contributions, whether they are user-generated content, tags, ratings or anything else. Avoid references from edited content to user contributions, and maintain the necessary relationships between edited content and user contributions as attributes of user contributions alone.

To integrate user contributions into your content model, you have to take the following steps:

- Add content types to the CONTENT TYPE HIERARCHY (2.1) to accommodate the various kinds of user-generated content. This may include additional content types for articles, plain text, tags, ratings, pictures, multimedia objects, etc. – whatever objects you expect the site's users to submit.
- If you prefer not to store specific user contributions in the content repository, you can choose to store them in any other data structure that appears appropriate, for example database tables.
- Establish unidirectional references that point from user contributions to the elements of edited content to which they refer. When a page with user contributions has to be generated, CONTENT SERVICES (3.1) can look up the various contributions related to a specific content element. The implementation of such a lookup logic becomes part of your site's domain logic.

As a consequence, no attribute of any edited content element will hold a reference to any user-generated content element. If a user contribution is added, no attribute of an edited content element will change.

Example resolved

User comments essentially consist of markup for formatted text, along with the user's name and e-mail address, so we introduce a new content type *Comment* into our CONTENT TYPE HIERARCHY (2.1). As Figure 37 shows, this new content type is not referenced from any existing content type.

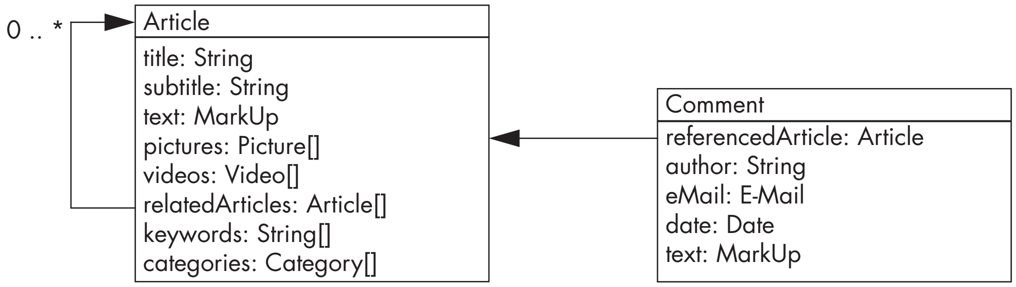


Figure 37: Content model including user-generated content

We will store other objects that result from user participation, such as high scores from online games and ratings of shop items. These objects are too simple to justify content types of their own, so we'll simply store them in plain database tables.

Benefits

+ The solution allows you to add user-generated content and folksonomies to your content model. It gives you a prime mechanism for meeting the requirements of Web 2.0.
+ The content editor workflow remains intact. The workflow for individual content elements, including processes of editing, updating and publishing, is completely independent of any user submissions and the way in which they are processed. No re-publication of edited content is necessary after user contributions have been added that refer to that content.
+ While the solution endorses a logical separation of edited content and user contributions, you can easily implement a physical separation as well if you store all user contributions in a separate database, should this be necessary for security reasons.

Liabilities

– To find the user contributions that refer to an element of edited content, you have to implement a lookup function that follows the unidirectional references backwards. Most content management systems offer a function that lists all referrers of a given content element. If your content management systems doesn't do this, you will have to maintain such a mapping yourself.

– This solution may have a slight impact on system performance. As there are no references from edited content to user-generated content, the lookup function has to be invoked every time such a connection has to be made. However, the additional level of indirection shouldn't prove too costly.

4.6 Input Channel for User-Generated Content

Context

You have made those additions to your content model that are necessary to accommodate user-generated content. You have established DECOUPLING OF EDITED CONTENT AND USER CONTRIBUTIONS (4.5). The workflow processes for edited content are independent of the artefacts, texts, tags or ratings that users may provide.

Problem

How can you ensure that user-generated content fits well into your content base?

Example

The website for the House of Effects accepts different types of user contributions. Most notably there are comments that users can write and attach to an article, a presentation or a shop item description. However, users can also rate shop articles, and the average rating is displayed on that item's page. Users can also participate in online presentations, and some of these presentations store the users' results, as in the example from Figure 36.

Forces

Unlike edited content, user-generated content usually doesn't follow a workflow process based on editing and publication. Users just submit or upload whatever they want to contribute using the mechanisms a site offers them.

There are two problems associated with this. First, if you accept all material that users may submit, you will soon run into problems: the chances are that users will submit material that cannot be properly processed. Typical examples include ill-formatted markup that fails to be parsed, pictures that are too large to be displayed correctly and multimedia objects that are too large to be used on a web page. This doesn't mean that your users intend to submit unusable material, it just means that they often aren't aware of the technical requirements and require a little guidance.

Second, there may be some users who aren't well-meaning. The inclusion of user-generated content into a website quickly invites malware such as viruses. Security becomes a major issue once user-generated content enters the scene – it's safe to assume that most sites that accept user-generated content will eventually be subject to a malware attack.

Ill-formatted, unwanted or inappropriate content can damage your site. The effect could be severe, beginning with awkward page layouts as a consequence of ill-formatted markup and ending with a fully-fledged virus infection. It's clear that this unacceptable.

Solution

Accept user-generated content only if it is submitted in a format that you endorse. Appropriate formats typically include the markup used by wikis, as well as standard formats for pictures and videos. If necessary, apply automatic transformations to adapt user-generated content to meet specific requirements.

In his article *Building Secure Web Applications*, George V. Neville-Neil emphasises that 'most sites now allow some form of user-generated content, and the smart ones do this by having a very restrictive whitelist of allowable things the user can upload' (Neville-Neil 2007).

Combining such a whitelist with a set of transformation processes, you can establish an input channel that handles user submissions in a way that is both effective and reasonably secure:

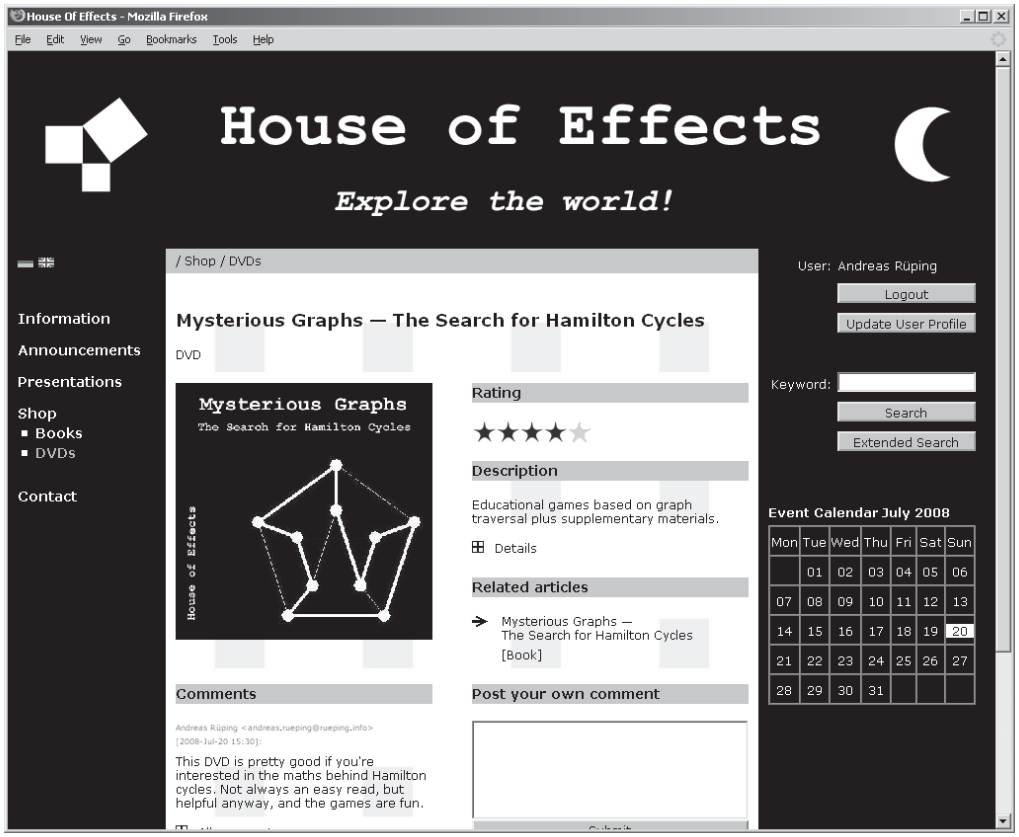
- To begin with, users can submit formatted text that may include some kind of markup. Tags for formatting purposes (such as headings, boldface or italics) are generally fine, while fully-fledged HTML and, especially, script tags are not (Neville-Neil 2007). A straightforward idea is to allow the kind of markup that wikis use. Also known as 'Wiki creole', this markup is powerful enough to express the necessary formatting while excluding tags that go beyond mere text processing. To ensure that users' submissions conform to this standard, you can implement a mechanism that parses the markup and automatically removes all unwanted tags.
- As far as images are concerned, accept only image types that you are able to process. Automatically resize all images to the size that you need immediately after they've been uploaded. This technique makes it easier for you to use user-submitted pictures on your web pages, while at the same time enabling the destruction of potential viruses, such as a virus in the header of a JPEG file (Neville-Neil 2007).
- If you allow the upload of sound files, videos or other multimedia objects, limit their size to acceptable proportions, if only to avoid unpleasant response times for the web pages where these elements will eventually show up. Check all incoming objects and reject them if necessary.
- Finally, some sites may allow users to submit instances of 'official' content types: objects that represent content types the content editors use as well. Although these objects are still user-generated content, you may want to apply the validators from WORKFLOW-BASED VALIDATION (2.5), such as those that check for valid date values, valid e-mail addresses and so on.

The overall principle should be clear. It's fine to accept a wide range of user-generated content, but it is important that you retain the right to discard content elements that don't fit your site or that fail to meet certain requirements, security or otherwise.

Example resolved

The House of Effects website is quite restrictive about user-generated content. For the time being users cannot submit Flash files, videos or other multimedia objects. These are not necessary, as the content editors are in charge of providing articles and presentations that represent the museum. On the other hand, the site does welcome feedback from visitors. Users are free to leave comments and to rate shop items. Figure 38 shows a sample page.

To allow users to leave comments, we'll use a simple markup that includes tags for formatting purposes. It's a markup that some users have probably already used with wikis and with which they should be relatively comfortable. A simple filter will remove any unknown tags or script code from the user-submitted markup.



Mysterious Graphs — The Search for Hamilton Cycles

DVD

Rating

★★★★☆

Description

Educational games based on graph traversal plus supplementary materials.

Related articles

Mysterious Graphs — The Search for Hamilton Cycles [Book]

Comments

Andreas Rüping <andreas.rueping@rueping.info>
[2008-Jul-20 15:30]:

This DVD is pretty good if you're interested in the maths behind Hamilton cycles. Not always an easy read, but helpful anyway, and the games are fun.

Event Calendar July 2008

Mon	Tue	Wed	Thu	Fri	Sat	Sun
	01	02	03	04	05	06
07	08	09	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Figure 38: Ratings and user comments as user-generated content

User ratings are made possible by a simple Ajax component that allows users to choose from one to five stars. A mouse-click is all it takes a user to offer a rating: no other user interaction is required, so only ratings between one and five are possible.

Benefits

+ A well-defined input channel provides users with some guidance regarding the submission of user-generated content. Usability is improved, as users will find it easier to add texts, tags, ratings, pictures, multimedia objects and whatever else your site may invite.

+ It is easier to incorporate user-generated content into your web pages if that content meets specific requirements, especially with regard to types and formats. There's a clear benefit that comes from using text that's properly formatted, pictures that are correctly sized, multimedia objects that aren't too large and so on.
+ Security is improved, as you avoid at least some malware. You can protect your site from script tags, weird markup fragments, malicious images and other malware elements by using a well-defined and reasonably secure input channel for user-generated content.

Liabilities

– This solution doesn't guarantee complete security. You will have to apply a set of general security patterns to protect your site completely against attackers. Important security patterns include a *Demilitarized Zone*, a *Protection Reverse Proxy* and an *Integration Reverse Proxy* (Schumacher Fernandez-Buglioni Hybertson Buschmann Sommerlad 2006).

– The solution doesn't guarantee content quality either. Even if user-generated content meets all the requirements for format and processing, it can still be valueless, embarrassing or even illegal. There is an ongoing discussion about the extent to which a web platform can be held responsible for the contributions its users make. Regardless of any legal aspects, most web platforms have an interest in shutting out inappropriate content, although what 'inappropriate' means depends of course on the individual website.

  There is no technical way to do this, but there are several site management strategies that can help you achieve your goal. First, you can accept user contributions only from a limited number of known and identified users, which quickly reduces the amount of nonsense contributions. Second, you can establish a moderation process in which a moderator approves user-contributed material before it is accepted for publication. This strategy is described in the *Quality Inspection* pattern (Schümmer Lukosch 2007). There is a downside to moderation, though, as it removes the spontaneity from user contributions, and might invite accusations of censorship on the part of the site owners. Third, you can accept all contributions, but let users rate the contributions for quality. The *Letter Of Recommendation* pattern takes this approach (Schümmer Lukosch 2007). Whatever strategy you apply to ensure the quality of user-generated content for your site, aim to achieve a balance of openness and security.

CHAPTER

5 Deployment and Infrastructure

In the preceding chapters we analysed how to structure, validate, organise and deliver web content. We looked at how domain logic can be applied, how search functionality can be integrated and how personalisation strategies can be added. We studied how s can be applied to create the actual web pages. We looked at quite a few software components, custom and otherwise, and at how they collaborate.

It's now time to examine how a website can go live. You might want to launch a new site or to relaunch an existing one, but in any case it's important to establish a set of reliable deployment processes and mechanisms. This is what this chapter is about.

This chapter is also about the infrastructure that's necessary to do so. It's about environments for development, testing and operation, and about how these environments have to be equipped to serve their purposes well.

Much of what can be said about deployment and infrastructure isn't specific to content management and content delivery, of course, but applies equally to web applications in general. I'm not going to talk much about web applications though, but rather keep the focus on content-related aspects of deployment and infrastructure. For a broader discussion of these topics, I'd like to refer you to the relevant literature, especially the patterns that Paul Dyson and Andy Longshaw describe in *Architecting Enterprise Solutions*

(Dyson Longshaw 2004). Putting a strong focus on the non-functional requirements of web-based systems, these patterns discuss techniques for making web applications fast, reliable, secure and maintainable. In addition, *Security Patterns* by Markus Schumacher et al. (Schumacher Fernandez-Buglioni Hybertson Buschmann Sommerlad 2006) plays an important role with regard to deployment and infrastructure for web-based systems.

Nonetheless, there are two patterns from the existing literature that I'd like to revisit, however briefly, before we move to the content-specific aspects of deployment and infrastructure. These two patterns are quite fundamental and form the basis for the rest of the chapter.

- The first pattern addresses the way in which web servers and application servers are normally distributed across different zones for security reasons. The idea is to introduce a *Demilitarized Zone* between the 'width' of the Internet on one hand and the secure inside world that hosts servers and databases on the other. The web servers reside in the *Demilitarized Zone*, where they are given limited access permissions. 'Ideally these web servers are not responsible for any business functionality', while 'business functionality is delegated to application servers that can be shielded from the outside world' (Dyson Longshaw 2004).
- The second pattern introduces *Load-Balanced Elements*, for reasons of scalability, performance and reliability. The rationale is to 'use multiple elements of similar capability and balance the load continuously across them to achieve the required throughput and response' (Dyson Longshaw 2004). As multiple web servers and multiple application servers operate in parallel, response times are improved and the risk of services not being available is reduced.

The application of both patterns results in the architecture sketched in Figure 39. It's a fairly typical architecture, found in many of today's web systems, with variations of course in the number of servers, the way in which these servers communicate, the firewall technology and other details.

This is the overall architecture we assume for the rest of this chapter, and it provides a good starting point from which to ask content-related questions about deployment and infrastructure:

- How exactly can you deploy software to the application servers – the software that is necessary for CONTENT MANAGEMENT AND CONTENT DELIVERY (1.1)?
- What are the consequences if the underlying content model changes, rather than only the software? How can you safely evolve your content model over time?
- How can you avoid downtime for the site during software deployment: periods during which the site isn't available?
- How can you reduce downtime for content management: periods during which the site is live but when content editors cannot create new, or maintain existing, content?

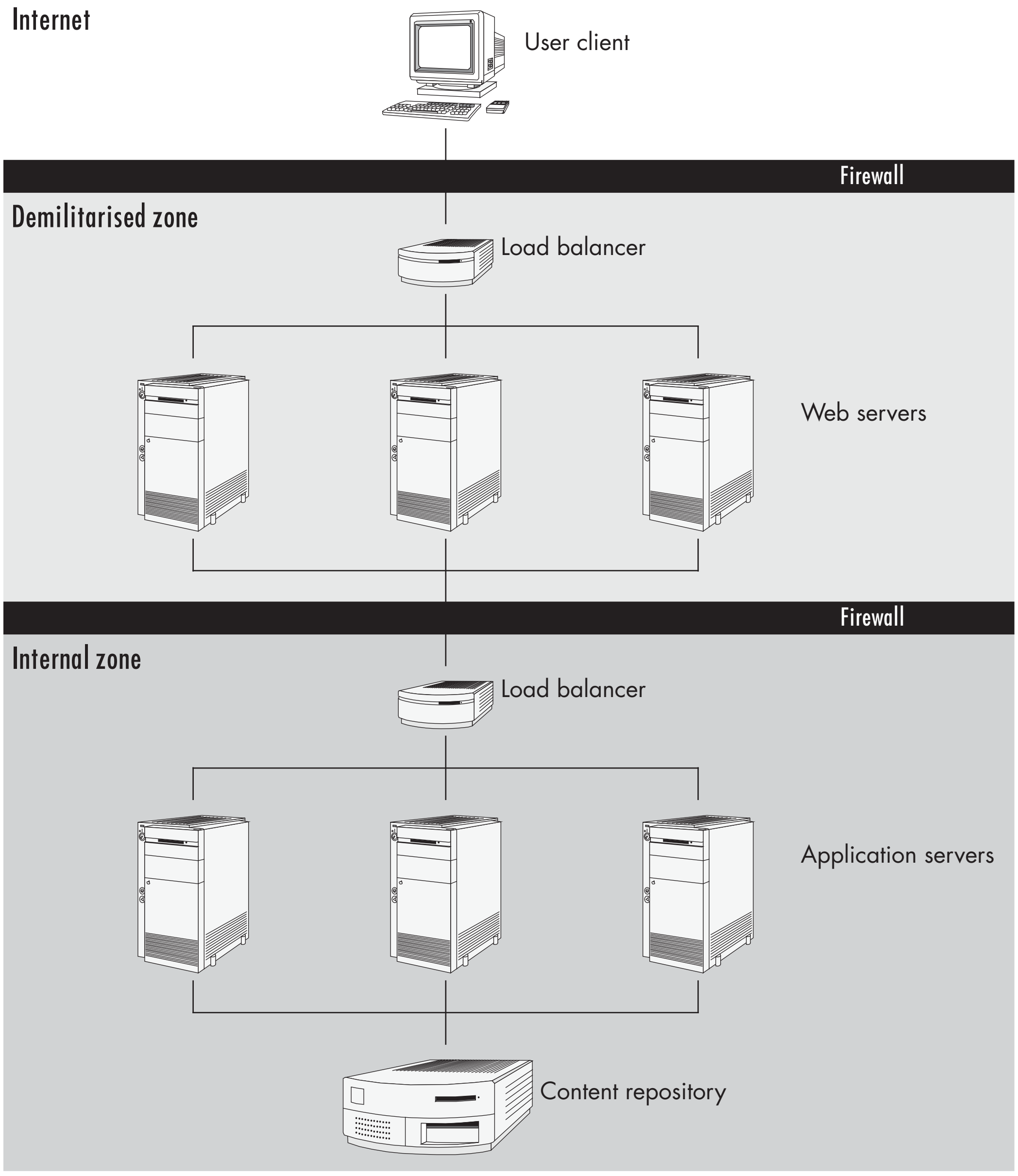


Figure 39: Load-balanced web and application servers distributed over internal environment and DMZ

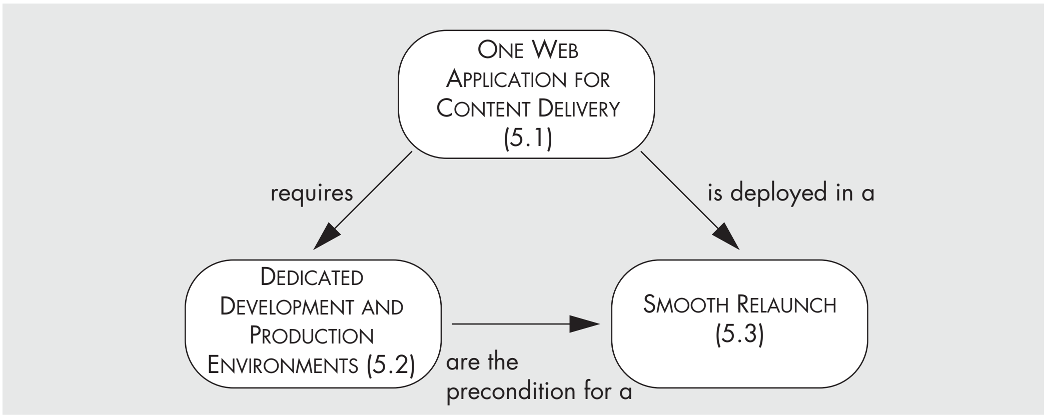

One Web Application for Content Delivery (5.1)
requires
is deployed in a
Dedicated Development and Production Environments (5.2)
Smooth Relaunch (5.3)
are the precondition for a

Figure 40: Road map to the patterns

The three pattern in this chapter answer these questions. They give both technical and organisational advice, taking the architecture described in Figure 39 as a basis. Figure 40 presents a road map to what the chapter has in store.

5.1 One Web Application for Content Delivery

Context

Software for a website consists of two main building blocks for CONTENT MANAGEMENT AND CONTENT DELIVERY (1.1). The content management components usually contain little custom software. The main task here typically lies in the correct configuration of the content management application, especially the specification of the underlying content model. You may have to register listeners for LISTENER-BASED SYNCHRONISATION (1.4), provide WORKFLOW-BASED VALIDATION (2.5) mechanisms or customise the editor client, but that should be all.

On the other hand, the content delivery software implements much of the domain logic for your site, so you have to expect a lot of custom software. Necessary components include CONTENT SERVICES (3.1), CONTENT FILTERS (4.1), a NAVIGATION MANAGER (3.2), a SEARCH MANAGER (3.3) and a SYSTEM OF INTERACTING TEMPLATES (3.4). To launch the site, you have to deploy all these components to the servers of the live system.

Problem

How can you establish a straightforward deployment process for your content delivery software?

Example

Content delivery software for the House of Effects website includes Java classes, JSPs and Ajax components that are all embedded in an overall framework provided by our content management system. To launch the site or to relaunch it implies deployment of all these components into the production environment.

How can this be done? What is the most straightforward way to do it? What kind of packaging is recommended when deploying content delivery components? These are questions we need to address before our software can go live.

Forces

Content delivery software consists typically of a variety of components, including classes, server pages, client code and configuration data. Depending on the underlying technology, you may have the option of defining one or more web applications. By definition, a single web application consists of resources that together provide a specific service to users. Different web applications are relatively independent of each other, while the resources within a single web application collaborate closely.[12]

Because different web applications are largely independent of each other, they can be deployed separately. This has the advantage that each software deployment is less complex and therefore less error-prone. It is considered good practice to avoid 'big-bang' style deployments, instead taking independent software packages live in sequence.

However, separate web applications have their drawbacks. Collaboration between them is possible but limited. First, it's possible for a component to call a component from another web application, but it's is more complicated and usually less efficient than a call made within the same web application. Second, application servers maintain different sessions for different web applications – exchanging session information across web applications is possible, but awkward.

These disadvantages are serious, since the different components for content delivery rely on each other rather heavily. Close collaboration in terms of service invocation is required, and the CONTENT TYPE HIERARCHY (2.1) represents a common basis for domain logic components, personalisation components, templates for HTML generation and Ajax-based client-server communication alike.

[12] The concept of web applications is common in the Java world – it's supported by virtually every application server on the market. Other technologies such as PHP don't yet offer support for separate web applications, although some frameworks are heading in this direction.

Moreover, the various components for content delivery usually share the same security configurations. Different kinds of security configurations are possible, specifying for example legal access paths, legal URL patterns and legal URL parameters for user requests. These configurations apply to content delivery as a whole, so it's hard to split them across separate web applications.

Solution

Define a single web application for all components directly involved in content delivery. This includes the custom components that you have developed for content retrieval, domain logic, personalisation and HTML generation, as well as the necessary configuration data. Define separate web applications only for loosely coupled tools, such as a stand-alone search engine or a personalisation engine connected through asynchronous communication.

A general principle from component-based software development is to 'provide an assembly package that contains all the component packages that make up an application', where a component package refers to the entirety of interfaces, implementations, annotations and configuration files required by a specific component (Völter Schmid Wolff 2002). More specifically, Paul Dyson and Andy Longshaw recommend that you 'adopt an application server architecture' and 'group all core system functionality in a single application' (Dyson Longshaw 2004).

The core components for content delivery of course include all the major custom components you have developed. The following components in particular will become part of a central web application for content delivery:

- CONTENT SERVICES (3.1), including their interfaces and implementations.
- Navigation Manager (3.2).
- Search Manager (3.3).
- CONTENT FILTERS (4.1) for providing personalised web content.
- Components implementing an INPUT CHANNEL FOR USER-GENERATED CONTENT (4.6).
- Components providing the Ajax code required for SELF-CONTAINED PAGES (3.6).
- A SYSTEM OF INTERACTING TEMPLATES (3.4) consisting of templates for the different layouts of various content types.

In addition, the web application should include all the necessary configuration data, ranging from access controls to deployment descriptors. Typical examples are:

- The specification of access permissions that express authentication and authorisation requirements, unless these requirements are implemented in server-side components such as CONTENT FILTERS (4.1).[13]

- The specification of legal URL patterns, for example to prevent the direct invocation of server pages or other scripting code 'from the outside', especially the invocation of templates reserved for logged-in users or special user segments.[14]
- The specification of legal URL parameters, for example to disallow parameters like '*mode=edit*' for a Wiki-style web page, in case the user isn't logged in or doesn't have the necessary permissions for write access.

These components and configuration files constitute a single web application, so they will be deployed together.

On the other hand, separate web applications are justified for tools that are sufficiently independent from the actual content delivery. Typical examples include an asynchronous search engine, an ASYNCHRONOUS PERSONALISATION ENGINE (4.2), shop software or a payment system. In many cases stand-alone third-party products will be used, but even if you choose to develop any of these systems yourself, they should become separate web applications with their own, independent deployment processes.

Example resolved

We introduce a single central web application for the delivery of House of Effects web content. Since our overall architecture is J2EE, this web application consists mostly of Java classes and JSPs and will be deployed as a '.war' file.

This web application embodies most of the custom components we have developed, but also includes a few components provided by our content management system. The following table gives an overview.

COMPONENT TYPE	COMPONENTS
CMS components	Components provided by the content management system implementing a framework for the integration of custom components, for example: ■ A dispatcher for the invocation of page template JSPs. ■ Repository layer functionality for database access. ■ Standard caching mechanisms.

[13] Most web servers allow you to express access constraints for specific web pages. This is a straightforward solution if access permissions for your site are simple. However, more advanced access permissions probably require dedicated server-side components to implement the desired behaviour.

[14] Most J2EE-based web and application servers allow you to define invocation constraints for JSPs and other components, usually through the specification of URL patterns. This enables you to ensure that specific templates cannot be activated directly through an HTTP request and must be invoked by an internal component.

COMPONENT TYPE	COMPONENTS
Custom Java classes	Various classes, mostly implementing the domain logic for the House of Effects website: ■ Content services. ■ The navigation manager. ■ The search manager. ■ Content filters and other classes implementing personalisation strategies. ■ Additional caching functionality.
JSPs	JSP code, representing layouts and client-side functionality: ■ Templates for all kinds of content elements, including articles, announcements, online presentations and shop item descriptions, as well as navigation nodes (page templates). ■ CSS definitions. ■ Ajax code to be included into JSPs, representing client-side functionality.
Configuration data	A web server configuration for URL patterns: ■ Legal:*.html. ■ Illegal:*.jsp. HTTP requests for JSPs therefore aren't routed to the application server, so users are prevented from bypassing the authorisation mechanisms implemented in our content filters and other personalisation components. In particular, users cannot call any JSPs that generate navigation elements for special user segments (*NavigationNode.renderTreeNavigationForRegisteredUser*, *NavigationNode.renderTreeNavigationForPremiumUser*). No further configuration data for access control is required, as all access permissions are handled by our server-side personalisation components.

There are of course components that are only loosely related to the content delivery processes and that therefore don't have to be included in the central web application.

The search engine and the shop software are in fact third-party products that come as separate stand-alone web applications. Our personalisation engine is not a third-party product, as we developed it ourselves, but it constitutes a separate web application anyway, since it's connected to the actual content delivery processes through asynchronous communication only. These stand-alone web applications can be deployed separately and independently of each other and of our content delivery web application, unless their interfaces and the specifics of their collaboration change.

Benefits

+ You create a central and self-contained web application that contains all components directly involved in content delivery. As you remove potential barriers, collaboration between related components becomes more straightforward. Comprehensibility, maintainability, even efficiency are improved as cross-application communication is reduced to a minimum.
+ You retain the option of independent deployment processes for any third-party products you use. Whether it's a search engine, a personalisation engine or shop software, any stand-alone application will also have its stand-alone deployment process. Software updates will be more straightforward, as loosely coupled components reside in separate web applications.
+ The single web application for content delivery provides a well-defined place for storing the configuration data that is necessary for your site to operate smoothly and securely. Web server and application server configurations are stored in a single central place, so it's easier to make sure they are accurate and complete. As a consequence it becomes easier for you to prevent users from bypassing authentication mechanisms and access control.

Liabilities

– You cannot establish separate deployment processes for specific components involved in your content delivery software. For example, you can't deploy CONTENT SERVICES (3.1), the NAVIGATION MANAGER (3.2) or any CONTENT FILTERS (4.1) separately.

– Your web application will require more configuration data than just a set of URL patterns for access control and the like. For example, load balancers will require additional configuration data, as will other tools you may use. Also, bear in mind that configurations differ from one environment to another. Development environment and production environment, internal zone and demilitarised zone all require specific configurations.

5.2 Dedicated Development and Production Environments

Context

Your website undergoes occasional changes, not just to the content, but also to the software that implements the domain logic. Whenever updates to the domain logic must go live, the site needs to be relaunched, and ONE WEB APPLICATION FOR CONTENT DELIVERY (5.1) must be deployed to the live system.

Problem

How can you evolve your website while ensuring its undisturbed operation?

Example

No website is ever complete, and it's safe to assume that the site for the House of Effects will not be an exception. Although we're quite happy with the site we have developed so far, there are ideas for future extensions.

Most notably, the House of Effects plans to open a section on computing. Once this section opens, it should of course be reflected by the museum's website. The idea is to allow users to contribute special online presentations that demonstrate example programs at work. This is likely to require additional content types.

These ideas are only plans for the future, and no concrete software development has to be done, but of course we want to make the preparations that are necessary to enable future additions to the site.

Forces

All software systems evolve. This is true especially for software systems in the quickly-changing world of the Internet. There is a virtually endless list of reasons why changes and additions to a website might become necessary. The information model may change, a new 'look and feel' might be required, demands for additional functionality might arise, an increased number of users might cause new performance requirements. The extent of these changes varies greatly, ranging from small updates to a complete relaunch.

An important aspect of this is the effect that changes to the domain logic may have on the content model. Although you can try to keep this effect to a minimum, modifications to the CONTENT TYPE HIERARCHY (2.1) might become necessary. These modifications could be simple additions (either of new content types or of attributes to existing content types), but might amount to a fully-fledged reorganisation of the content model.

However, reorganising the content model usually isn't something you can do on the spot. The content model, and more specifically the CONTENT TYPE HIERARCHY (2.1), is essential for content management and content delivery processes alike. These processes must continue undisturbed while new software is under development and being tested. This is quite clear for the content delivery processes, as the site has to stay on line all the time, but it is equally true for the content management processes – content creation, maintenance and publishing must continue as well.

Solution

Establish separate environments for software development and for the live system. Make sure that these environments can work on different content models if essential changes to the content model have to be expected. This may require the introduction of logical partitions for your content repository, or even separate installations of your content management system.

It is clear that separate environments are necessary for software development and testing (development environment) and for the live system (production environment).[15] The development and the production environments should be largely identical, except perhaps for hardware equipment and security configurations. For example, the live system is likely to have more load-balanced web and application servers than the development system and, unlike the development system, typically has its web servers placed in a demilitarised zone (Dyson Longshaw 2004). Distinct development and production environments make an undisturbed operation possible while new software is under development.

An important issue, however, is raised by a possible evolution of the content model:

- If no changes or mere additions to any content types have to be expected, the development and the production environment can use the same content model. In fact, they can use the same content repository. You can make backward-compatible changes to the CONTENT TYPE HIERARCHY (2.1) on the fly: they will be in effect for both environments immediately.
- Things look different if the CONTENT TYPE HIERARCHY (2.1) undergoes a reorganisation that isn't backward-compatible. This is the case if, for instance, content types or attributes are deleted or associations between content types are modified. Changes like these must only be made to the content model in the development environment and need to be tested there, leaving the production environment and its content model completely unchanged. This requires at least separate 'logical' repositories with distinct content models for either environment. If your content management system doesn't support logical repositories, separate physical repository installations become necessary.

The first case is of course easier to handle, but the second is more flexible and gives you more freedom to evolve your content model. Which case applies depends on the requirements you wish to implement.

Figure 41 shows an example infrastructure with dedicated development and production environments. The development environment on the left-hand side largely mirrors the production environment on the right save for different hardware equipment. Firewalls

[15] You can also introduce dedicated environments for development and testing, and in fact many projects choose to do this. The advantage is that integration testing is more effective in a dedicated testing environment in which no development takes place. However, this discussion is in no way specific to the handling of web content, and therefore beyond the scope of this pattern.

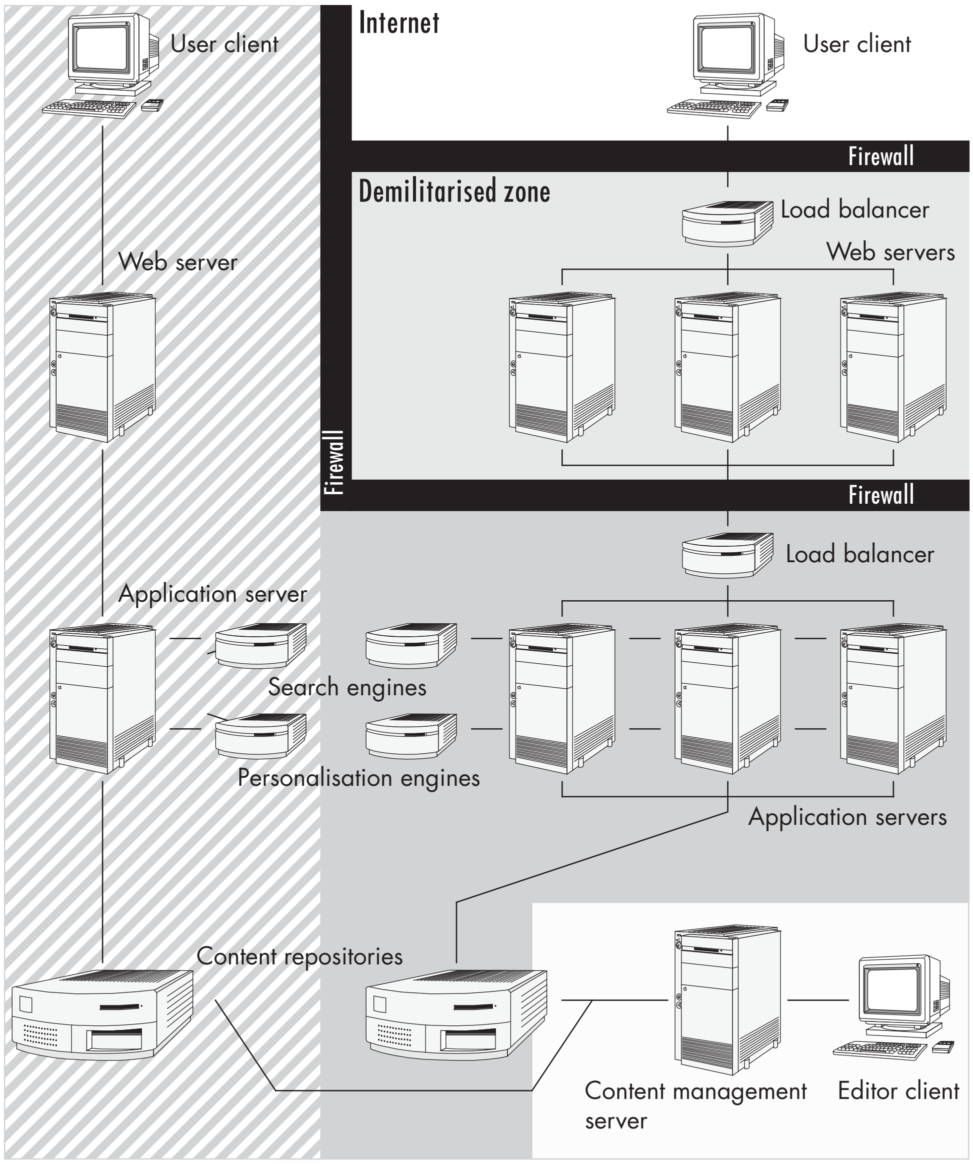


Figure 41: Dedicated development and production environments

separate the Internet from the demilitarised zone and the demilitarised zone from the internal environments. All development and testing activities take place in the development environment. Since both environments have their own content repository, an evolution of the content model is possible, as it would only affect the repository in the development environment. If your content management system supports repository partitions with distinct content models, the two repositories in the diagram can be implemented as logical repositories (partitions) residing within the same physical repository.

The infrastructure in Figure 41 also features separate installations of stand-alone tools such as search engine and personalisation engine. Separate installations give you the best possible freedom for tests in the development environment. If you plan to take new versions of any stand-alone tool live, you need separate installations for these, as only this approach will allow you to perform the necessary tests. However, if you're sure that a stand-alone tool will remain untouched, it is acceptable to have just one installation and share it across the development and the production environment.

Example resolved

Although currently there are no concrete plans for a relaunch that would require a reorganisation of our CONTENT TYPE HIERARCHY (2.1), we make sure that our development environment and our production environment can, in principle, operate with different content models.

Fortunately, our content management system allows us to split the content repository into separate logical partitions that reside in separate database schemas and are therefore free to use separate content models. This ensures that we are well prepared for the future. Should computing be introduced as an additional topic one day and should, as a consequence, new content types become necessary, we can make these changes in the development environment while the existing site can operate undisturbed.

Benefits

+ Website operation and software development are largely decoupled. No software development activities have an effect on the site that is currently live until the moment the new software goes into production. Most notably, arbitrary changes can be made to the content model without an effect on the live site.
+ Content maintenance and software development are largely decoupled. Content editors can create and update live content completely independently of any software development activities.
+ The dedicated development environment contributes much to the site's testability. The environment can serve as a platform for unit tests, system tests, performance tests and more, resulting in a more accurate and more reliable site.

Liabilities

- Maintaining separate development and production environments represents additional cost of both software and hardware, especially if separate installations of the content repository or other tools become necessary. Details depend on the licence models of the tools you use. The additional cost, however, is a price you have to pay to stay manoeuvrable in the presence of change.
- Although the development environment may simulate the production environment to some extent, these environments will probably not be exactly the same. Firewall installations and load-balancing configurations could differ, at least in their details, which puts a small question mark against the reliability of the tests you perform in the development environment. As important as these tests are, it's still wise to check the live site for possible deviations from the development environment.
- If a relaunch requires a reorganisation of the CONTENT TYPE HIERARCHY (2.1), the content models in the development and the production environment are different and have to be merged before the relaunch can take place. In other words, some content migration will become necessary immediately before the new site – including the new content model – can go live. While this migration effort is underway, no content maintenance will be possible: some period of content freeze is impossible to avoid. You need to plan for a SMOOTH RELAUNCH (5.3) to tackle this problem both at a technical and an organisational level.

5.3 Smooth Relaunch

Context

You are in the process of further evolving your site, making changes and additions to your custom software, and perhaps making changes to the CONTENT TYPE HIERARCHY (2.1) as well.

You have established DEDICATED DEVELOPMENT AND PRODUCTION ENVIRONMENTS (5.2) to decouple software development and content maintenance. Now you're planning to take the new software live.

Problem

How can you avoid conflicts when relaunching your site?

Example

Despite the separate environments we established for software development and the operation of the House of Effects website, we're aware that when we take software updates live, or even relaunch our site, conflicts with content maintenance can occur. This is to be expected, especially if significant changes are made to the content model.

However, we want to keep the impact on the content editors' workflow to a minimum, and we certainly don't want to take the site off line.

Forces

Once the development of new software has been completed and the software has been successfully tested, you need to deploy it in the production environment. Most importantly, this includes the ONE WEB APPLICATION FOR CONTENT DELIVERY (5.1) that you have established and which needs to be taken live to relaunch the site.

Even if your DEDICATED DEVELOPMENT AND PRODUCTION ENVIRONMENTS (5.2) are largely identical, they are still separate environments and they will probably differ in hardware and network configurations. Software that has been tested in the development environment is likely to work well in the production environment, but this cannot be guaranteed. It is therefore wise to perform a final test after the software has been deployed but before it's taken live.

Moreover, if the software you plan to release requires a new version of your CONTENT TYPE HIERARCHY (2.1), there are a few other things that you need to take into account. In such a case, content migration becomes necessary – existing content has to be transformed from the old model to the new one. Despite the possible need to migrate existing content, you have to avoid downtime during which the site has to be taken off line. For many sites, downtime is a clear 'no-go' due to its severe consequences on business and reputation.

It's also desirable to avoid content freezes as much as possible – periods during which content cannot be created or maintained. Such periods aren't popular with content editors, although short time spans are usually acceptable.

In fact a content freeze is hard to avoid in the case of content migration, which may take some time. Typical time spans range from a few hours to a weekend, depending on the content volume. To avoid conflicts, no editorial changes to the content should be made during that time. You should nevertheless try to minimise periods of content freeze.

Finally, the fact that different groups of people are involved, and that these groups have slightly conflicting requirements, adds to the difficulty of a site relaunch. On one hand there are software people who have an interest in proper development and testing, while on the other there are the site owners and content editors who want to reduce downtime and content freezes.

Solution

Apply staging to ensure that you can deploy a new software version to the production environment while the old version is still up and running. If necessary, migrate the live content in the staging environment. Launch the new version by a simple change of configuration, so that an equally simple change allows you to revert to the old version should anything fail.

A part of the production environment, the staging area has the overall purpose of making convenient changes between software versions possible. In *Architecting Enterprise Solutions*, Paul Dyson and Andy Longshaw make the following recommendation: 'Introduce a staging environment into the production servers. Implement a mechanism that allows you to swap the staging environment with the production environment, effectively swapping in the new version of the system.' (Dyson Longshaw 2004)

Although not primarily a test environment, the staging area can also be used for final tests. As the staging area is a part of the production environment, tests performed there are particularly meaningful.

The precise role of the staging area in content migration depends on the extent of changes that have been made to the content model:

- If the content model remains unchanged, or if only backward-compatible changes have been made, the live system (the 'old' site) and the system in the staging area (the 'new' site) share the same CONTENT TYPE HIERARCHY (2.1). No content migration is therefore necessary.
- If non-backward-compatible changes have been made to the CONTENT TYPE HIERARCHY (2.1), then different versions of the content model are used in the live repository (the 'old' content model) and in the development repository (the 'new' content model). In this case, you can use the staging area to perform the necessary content migration: export the entire content from the live repository, transform it into the schemas of the new CONTENT TYPE HIERARCHY (2.1), and populate the repository in the staging area by importing the results of the transformation process. The staging area repository is either a separate installation or a 'logical' repository, provided that your physical content repository allows for several logical partitions with distinct content models. A content freeze is in effect while the content migration takes place, so no content maintenance will be possible during that time.

In either case, the process of taking the new version of your site online consists of no more than a configuration change after the final tests in the staging area have been approved and any necessary content migration has been completed.[16] Moreover, you have the option of reverting to the old version in the case of an emergency. Should the new

[16] The configuration that is changed is usually the load-balancer configuration, as it specifies which of the servers (virtual hosts) in the production environment are used for the live system.

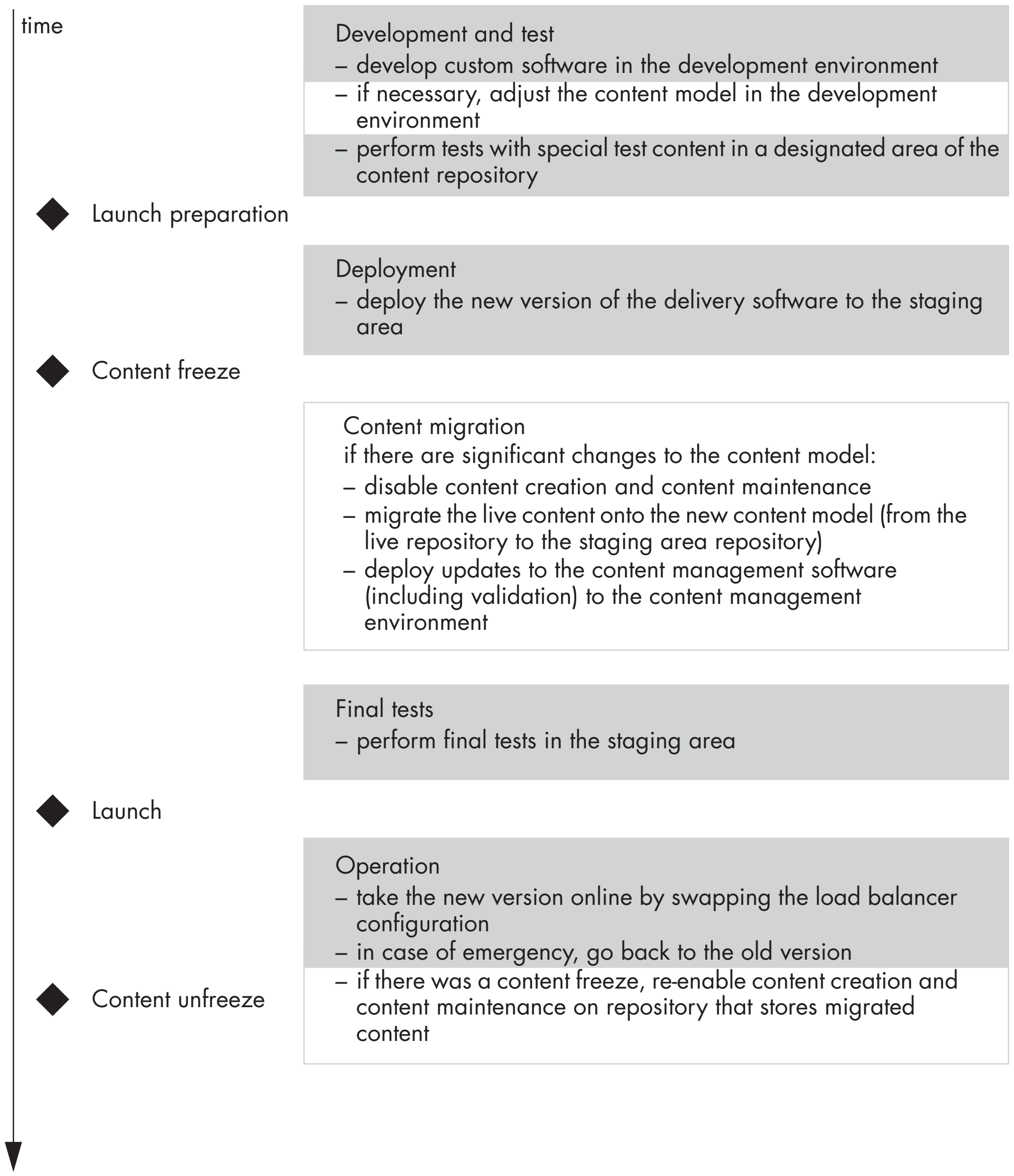

time
Development and test
– develop custom software in the development environment
– if necessary, adjust the content model in the development environment
– perform tests with special test content in a designated area of the content repository
Launch preparation
Deployment
– deploy the new version of the delivery software to the staging area
Content freeze
Content migration
if there are significant changes to the content model:
– disable content creation and content maintenance
– migrate the live content onto the new content model (from the live repository to the staging area repository)
– deploy updates to the content management software (including validation) to the content management environment
Final tests
– perform final tests in the staging area
Launch
Operation
– take the new version online by swapping the load balancer configuration
– in case of emergency, go back to the old version
– if there was a content freeze, re-enable content creation and content maintenance on repository that stores migrated content
Content unfreeze

Figure 42: Relaunch process using a staging environment

version of your site turn out not to work properly, you can undo the configuration change and swap the old version back in.

Ideally you always keep two versions of your site in the production environment – the one that is currently online and the previous one, until preparations for the next relaunch begin and servers that still host the previous version become the staging area for the upcoming release.

Figure 42 summarises the overall relaunch process with its major milestones. The steps shown shaded are those that are always necessary, while those shown unshaded are only needed if a significant change to the content model has been made.

Finally, there is also an organisational aspect to a site relaunch. Since a relaunch involves various groups of people – site owners, content editors, software developers and testers above all – collaboration is essential. It is generally good practice to appoint a site manager who is equally aware both of the software processes and the demands on content maintenance. The site manager must keep in touch with all stakeholders and ultimately has to coordinate all activities concerned with the site's evolution. The site manager has to negotiate deployment schedules and identify content freeze periods. A good deal of coordination by the site manager is the precondition for a smooth ride through the software release cycles and for an undisturbed content editors' workflow.

Example resolved

We decide to equip the production environment for the House of Effects website with four application servers. Two servers will host the live application and share the load between them, while the remaining two servers will form the staging area. Our load-balancer 'knows' which servers are currently online, and a simple configuration change is all it takes to let the servers change roles and swap in the new version.

This means that the previous version of our site will be available until the servers that host it become the staging area for the next release – the place to which the next version will be deployed, where a possible content migration is performed and where final tests will be made before taking that version online.

Benefits

+ Since you can simply swap in a new version, including a new content model if necessary, downtime for your site can be avoided altogether. Availability is improved, as you never have to take the site off line, which clearly is a great plus.
+ Periods of content freeze are reduced to a minimum. Even if the content model undergoes significant changes or a complete reorganisation, a content freeze is necessary only while the content migration takes place. The content editors' workflow isn't disturbed more than absolutely necessary.

+ Testability is improved, because new software can be tested after it has been deployed to the production environment and after a possible content migration has taken place. What you test is exactly the same version that will be swapped in and go live after you make the necessary configuration change.
+ As you retain the option of reverting to the old version in the event of a failed relaunch, the site's reliability is increased. This fall-back strategy is available to you even if that version uses an older content model. Of course this fall-back strategy isn't what you want, and of course you'll perform the necessary tests to avoid such a scenario, but if things *do* go wrong, going back to the old version is still better than being unable to do anything.
+ As the site manager coordinates all software deployments, relaunches, content freezes and so on, collaboration between software developers and content editors is improved. There won't be any unexpected periods during which no content maintenance is possible, let alone any unexpected downtime. Content freezes and other issues that require coordination can be planned ahead comfortably.

Liabilities

– The introduction of a staging area requires you to equip your production environment with additional servers. If you want to be perfectly flexible and keep the two most recent versions available at all times, you need twice the number of servers as you would for the live system alone. There is not much organisational overhead, as all servers are likely to be controlled by one load-balancer, but of course the additional servers represent additional cost for operating the site.

– The solution allows you to perform content migration without taking the site off line, yet the price you pay for this is to have two 'logical' repositories in the production environment, at least if you expect significant changes to the content model. If your content management system is able to partition the content base and use different content models simultaneously, there is not much of a problem. If it isn't, you'll need two separate repository installations to store content based on different content models. In the latter case, you'll probably face extra costs, with the details depending on your content management system's licence model.

– Although you keep the content freeze to a minimum, it still has to be handled with care. A complete content migration can take a few hours or even a couple of days, depending on how large your content base is, and during that time no content maintenance is possible. It's wise to choose a time when no urgent content updates are expected.

– The role of a site manager admittedly represents an additional organisation effort, and ultimately increased cost. Someone has to be paid to do the job, after all, but it's a worthwhile cost, as the cost of unexpected downtime can easily exceed the cost of proper site management.

Planning a Project

In the previous chapters we studied a collection of design patterns for custom software development of an advanced website. Now I'd like to change the perspective and look at web development from a project management point of view.

This chapter describes the tasks and activities that must appear on a typical web project's agenda, where 'web project' refers to the kind of project that seeks to establish an advanced website or web platform, and the necessary custom software development in particular. I won't be too specific, as details vary from project to project. Web projects can take on quite different forms and it is pointless to come up with a prefabricated project plan and assume that every project could use it. It's the old rule again: one size does not fit all.

Nonetheless, I'd like to present, in the form of a checklist, those topics that have to be addressed in a typical web project. The list should give you a fairly realistic impression of what a project plan might look like. Feel free to tailor this list to the needs of your specific project and make additions and modifications as you see fit.

Figure 43 gives an overview, while the subsections that follow list the tasks for the individual work packages. For convenience, I'll add references (following an arrow symbol) to relevant patterns from this book, and to other books in a few cases.

This chapter does not a assume a specific software development method, though to some degree I'll favour an agile approach. Agile methods are characterised by iterative processes that adapt well to changing requirements and by the close collaboration of everybody involved in a project. The principles of agile development are available on the web (www.agilemanifesto.org).

An agile approach seems well-suited for a web project, as it favours quick release cycles, early customer feedback and increased manoeuvrability – all things that are of paramount importance for projects that aim at a medium that is evolving as quickly as the web.

Various agile methods have been described in the literature (Beck 2004, Schwaber Beedle 2008, Cockburn 2002). This is not the place to discuss the pros and cons of any of them, although I'd recommend that you make yourself familiar with one of these more straightforward and more light-hearted approaches to software development before you start your web project.

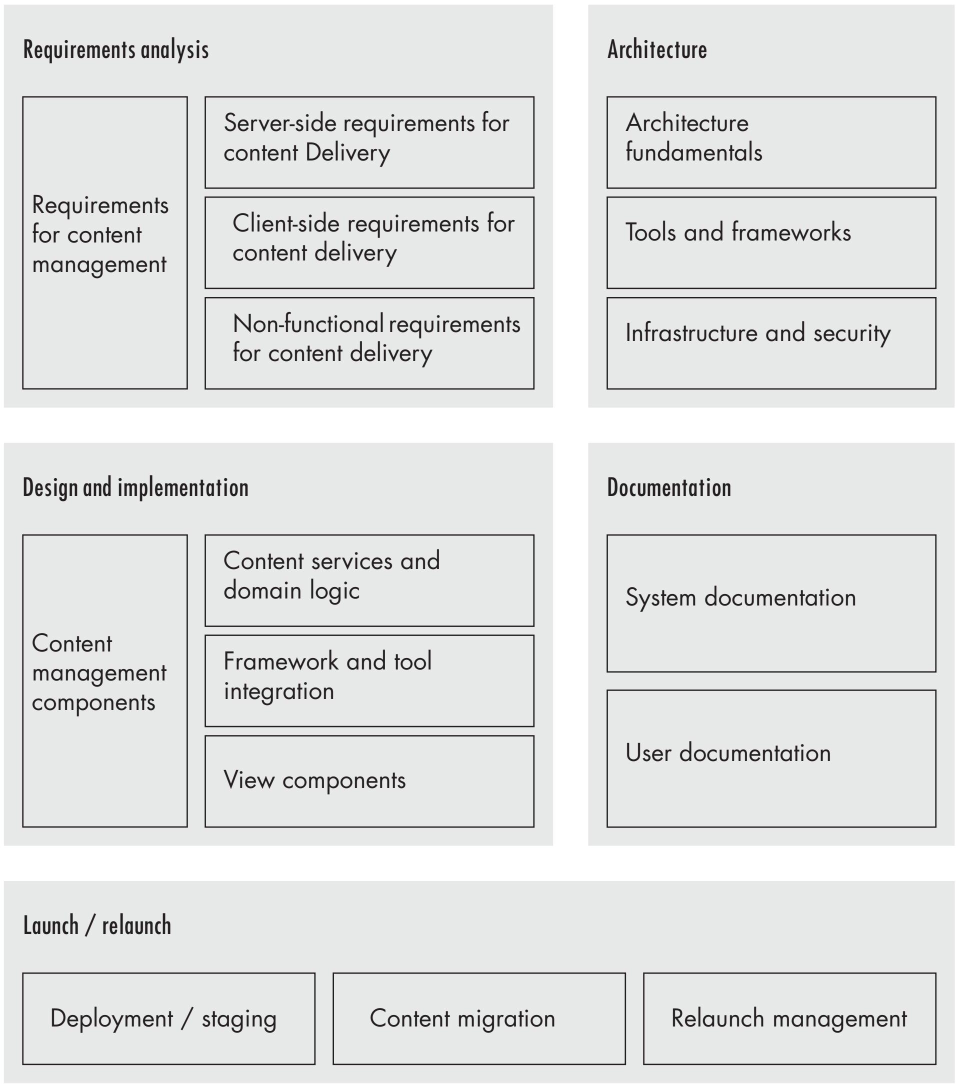

Requirements analysis
Requirements for content management
Server-side requirements for content Delivery
Client-side requirements for content delivery
Non-functional requirements for content delivery
Architecture
Architecture fundamentals
Tools and frameworks
Infrastructure and security
Design and implementation
Content management components
Content services and domain logic
Framework and tool integration
View components
Documentation
System documentation
User documentation
Launch / relaunch
Deployment / staging
Content migration
Relaunch management

Figure 43: Work packages of a web project

Requirements Analysis

Agile methods suggest that requirements analysis is not performed ad infinitum, but instead that you analyse the functional and non-functional requirements only to an extent that allows you to get properly started with the next steps, which include design and implementation.

Much in this vein, the work packages that follow aren't necessarily meant to produce overly detailed specifications. They simply summarise those things that are worth considering in the early stages of a web project. Discuss important requirements with your customer and show them your models and sample web pages. Use the feedback you receive to refine and improve your requirements analysis as you go. And as soon as it feels safe to do so, move on to design and implementation.

Server-Side Requirements for Content Delivery

Content model

- Analyse the content types your site requires. Focus on domain-driven content types, not on technical ones.
- Specify what kind of content users should be able to submit. Define special content types for user-generated content.
- Establish what relationships hold between content elements. Express this through associations between content types.
- Add attributes for classification purposes (tags, categories) to all content types. Make sure you meet the requirements that the intended search functionality has on meta information.
- Specify how users will be able to get involved in content classification (tags, folksonomies, ratings).

→ CONTENT TYPE HIERARCHY (2.1)

→ TAXONOMY BASED ON KEYWORDS AND CATEGORIES (2.4)

→ DECOUPLING OF EDITED CONTENT AND USER CONTRIBUTIONS (4.5)

Navigation and search

- Set up an initial navigation hierarchy. Content editors will later be able to adapt the navigation hierarchy as they see fit, but an initial hierarchy is necessary to get you started.
- Think about possible examples of dynamic content linking.
- Specify the search functionality for your site (full-text search, keyword search, category-based search, fuzzy search).

→ DECOUPLING OF CONTENT AND NAVIGATION (2.2)

→ DYNAMIC CONTENT LINKING (2.3)

→ TAXONOMY BASED ON KEYWORDS AND CATEGORIES (2.4)

Personalisation

- Define what user segments will exist, and the extent to which content editors will have to define criteria for grouping users into segments. User segments form the targets for the personalisation strategies that apply.
- Define the personalisation strategies for your site.
- Define possible algorithms that underlie these personalisation strategies. Examples include recommendation strategies and algorithms for assigning users to user segments.
- Decide what monitoring is required for the personalisation efforts.

→ CONTENT FILTERS (4.1)

Additional functionality

- Check what additional functionality your site requires (community support such as forums or chatrooms, shop software, etc.).
- Analyse the requirements on operations (reports, statistics, etc.).

→ (Schümmer Lukosch 2007)

Client-Side Requirements for Content Delivery

Page design

- Define the necessary page types. A page type for each domain-motivated content type is the norm, plus page types for search forms and the like.
- Design wire frames for the all page types, outlining how the pages are in principle composed from smaller page elements.
- Engage a web designer to provide sample pages (preferably in XHTML with the necessary CSS sheets) for the different page types. Several sample pages are necessary if different views are required (for example for different output channels, such as standard browsers and mobile devices). Make sure these sample pages meet the layout requirements for your site.

→ (Scott Neil 2009)

Client-side functionality

- Decide on realistic assumptions about browser technology, taking the target users' expected software infrastructure into account.
- Specify the client-side functionality (later to be implemented with Ajax technology) for all page types.

→ Sensible Client-Side Interaction (1.3)

→ Self-Contained Pages (3.6)

Non-Functional Requirements for Content Delivery

Performance requirements

- Analyse the performance requirements that hold for your site, especially the average and the maximum response time (assuming an average and a maximum number of users).

Availability requirements

- Analyse the required average availability.
- Analyse what downtimes and what content freeze intervals are acceptable during relaunch periods.

Scalability requirements

- Analyse the expected number of concurrent users at present.
- Analyse the expected number of concurrent users in the future.

Maintenance requirements

- Get an idea of features you may have to add to the site in the long run. More generally, get an idea of how the site might evolve in the future. This will give you a feel for the importance of maintainability.

Security requirements

- Analyse security requirements for authentication, privacy, etc. based on the site's criticality.

→ (Schumacher Fernandez-Buglioni Hybertson Buschmann Sommerlad 2006)

Requirements for Content Management

Workflow

- Analyse what workflows are desired for the content editing process (four-eye workflow, etc.).
- Analyse the extent to which content validation has to be integrated into these workflows.

→ WORKFLOW-BASED VALIDATION (2.5)

User Participation

- Analyse how user-generated content should be managed, and what approval processes should be in effect.

→ INPUT CHANNEL FOR USER-GENERATED CONTENT (4.6)

Architecture

Before you can start with the implementation, you need to make a few fundamental decisions about software architecture. Moreover, you have to select the tools and frameworks that you wish to use. In addition there are dependencies, as the architecture you define has an impact on the tools you choose and, vice versa, the tools that are available to you may impose constraints on the architecture that you can define.

The following work packages cover the essential aspects of software architecture for a web project.

Architecture Fundamentals

Technology

- Decide on an overall technology. A common option is a Java-based platform (including Java, JSP, XML, etc.). Alternative options include platforms such as PHP or Ruby on Rails. Which of these options should be given preference depends on the demands of your specific project, on what technology is readily available and on the available development skills. In general, Java is considered fine for expressing domain logic, so sites rich with domain logic may favour a Java-based approach, while on the other hand PHP or Ruby may appear more lightweight and may offer quicker release cycles. In principle, the patterns in this book can be applied regardless of the technology you choose, although many patterns suggest, for maintainability's sake, the reduction of scripting code to a minimum.

Overall architecture

- Define an overall architecture, consisting of several components (content management system, search engine, personalisation engine etc.). Define how these components interact (synchronous versus asynchronous communication).
- Develop a layered server-side architecture for dynamic content delivery, motivated by a strict separation of concerns.
- Organise the server-side functionality as a set of services, in the sense of a service-oriented architecture.
- Decide on an overall caching strategy. In particular, make a decision about the caching of content objects versus the caching of HTML fragments (or both), taking the possible effects of personalisation into account.
- Decide on the extent to which you wish to apply Ajax to support client-side event handling.

→ CONTENT MANAGEMENT AND CONTENT DELIVERY (1.1)

→ DYNAMIC CONTENT DELIVERY PLUS CACHING (1.2)

→ SENSIBLE CLIENT-SIDE INTERACTION (1.3)

→ LISTENER-BASED SYNCHRONISATION (1.4)

→ LAYERED ARCHITECTURE FOR CONTENT DELIVERY (1.5)

→ CONTENT SERVICES (3.1)

→ (Buschmann Meunier Rohnert Sommerlad Stal 1996)

→ (Fowler 2003)

→ (Dyson Longshaw 2004)

Tools and Frameworks

CMS selection

- Select a content management system that fits your choice of technology.

→ See the checklist in the next part of the book, *Choosing a Content Management System*.

Framework integration

- Plan the use of web development frameworks. Such frameworks usually support a layered server-side architecture. Frameworks are available for different technologies (Struts, Spring, Java Server Faces in the Java world, Zend in the PHP world, Ruby on Rails for Ruby).

→ LAYERED ARCHITECTURE FOR CONTENT DELIVERY (1.5)

Tool integration

- Plan the use of other third-party tools (search engine, personalisation engine, shop software, community software such as a chatroom or a forum). Make sure you select tools that fit your technology of choice and that integrate smoothly with existing tools and frameworks, most notably your content management system. Make sure you take licensing questions (for example commercial versus open source) into account when you make your decisions.

→ LISTENER-BASED SYNCHRONISATION (1.4)

Infrastructure and Security

Infrastructure

- Set up the technical infrastructure for your site, especially the hardware equipment (web servers, application servers, load balancers, etc.). Make sure you meet performance, availability and scalability requirements.

→ (Dyson Longshaw 2004)

Security

- Set up the security mechanisms for your site, for example a demilitarised zone, firewalls, authentication and authorisation mechanisms.

→ (Schumacher Fernandez-Buglioni Hybertson Buschmann Sommerlad 2006)

Design and Implementation

Once you have a reasonably good idea of the requirements for the site and the overall software architecture has been defined, you can move on to design and implementation.

In an agile project, design and implementation aren't performed in strict sequential order, but instead follow an iterative approach. The following work packages describe the areas that need to be covered by design and implementation, preferably in the course of several iterations, each of which will provide you with valuable feedback for the next.

An iterative approach also makes refactoring easier. Refactoring contributes to software maintainability and paves the way towards sustainable software architectures (Fowler 1999). The patters in this book can help you with design and refactoring activities alike.

Content Services and Domain Logic

Content hierarchy

- Extend the content model you have obtained in the analysis stage with 'technical' content types that aren't motivated by the application domain, but represent concepts such as navigation or configuration.
- Relate content types through association and abstraction.

→ Content Type Hierarchy (2.1)

→ Decoupling of Content and Navigation (2.2)

→ Dynamic Content Linking (2.3)

→ Taxonomy Based on Keywords and Categories (2.4)

Services

- Define and implement a set of services that represent the domain logic for your site. Services will cover content, search functionality, navigation and perhaps more.
- If necessary, integrate caching mechanisms into these services where appropriate. This depends on the caching strategies that your overall architecture requires and the extent to which these strategies are made available by your content management system.

→ Content Services (3.1)

→ Navigation Manager (3.2)

→ Search Manager (3.3)

Personalisation

- Define content filters that implement personalisation strategies by assigning content to specific user segments.
- If you use a dedicated personalisation engine, integrate it into your server-side architecture. Consider applying asynchronous communication.
- Implement mechanism for monitoring personalisation effectiveness.

→ CONTENT FILTERS (4.1)

→ ASYNCHRONOUS PERSONALISATION ENGINE (4.2)

→ SEGMENT-SPECIFIC CACHING (4.3)

→ CONDENSED EFFECTIVENESS REPORTS (4.4)

User participation

- Implement mechanisms for accepting and storing user-generated content.
- Add services for handling user-generated content to your service-oriented architecture.

→ DECOUPLING OF EDITED CONTENT AND USER CONTRIBUTIONS (4.5)

→ INPUT CHANNEL FOR USER-GENERATED CONTENT (4.6)

Frameworks and Tool Integration

Frameworks and tool integration

- Provide integration mechanisms for all external tools (search engine, personalisation engine, shop software, community software, etc.). This typically includes specific adapter components as well as communication components such as listeners.

→ LISTENER-BASED SYNCHRONISATION (1.4)

→ SEARCH MANAGER (3.3)

→ ASYNCHRONOUS PERSONALISATION ENGINE (4.2)

View Components

Template hierarchy

- Design a hierarchy of templates for page generation.
- Implement the necessary templates so that they generate the XHTML that matches exactly the pages and page fragments the web designer has provided.

→ SYSTEM OF INTERACTING TEMPLATES (3.4)

→ TEMPLATE PER VIEW (3.5)

→ SELF-CONTAINED PAGES (3.6)

Client-side functionality

- Implement the client-side functionality with Ajax.

→ SENSIBLE CLIENT-SIDE INTERACTION (1.3)

→ SELF-CONTAINED PAGES (3.6)

→ (Mahemoff 2006)

Content Management Components

Configuration

- Create the necessary configurations that inform the content management system of the content model (usually an XML file).
- Create more configurations if necessary. Typical examples include the customisation of the editor client, workflow definitions etc.

Workflow and event handling

- Implement the necessary validation mechanisms and integrate them into the workflow processes.
- Implement the custom repository listeners. Examples include listeners that notify the search engine or the personalisation engine of content changes.

→ LISTENER-BASED SYNCHRONISATION (1.4)

→ WORKFLOW-BASED VALIDATION (2.5)

→ ASYNCHRONOUS PERSONALISATION ENGINE (4.2)

Documentation

In agile development the idea is to cut down documentation to what is really needed. Most websites, even truly advanced web platforms, will hardly require extensive documentation – producing loads of documents will get you nowhere fast.

However, some documentation *is* necessary. The most important goal here is to capture knowledge that will be useful in later project stages, or once the project has been completed and the team has dissolved. Important targets of documentation include the fundamental models, the big picture of the overall architecture, and the rationale behind the design decisions that were made (Rüping 2003).

The following section describes what's worth documenting in a typical web project. You can mine the necessary information from team discussions or from special project retrospectives (Kerth 2001).

System Documentation

Requirements specification

- Provide a requirement specification. In most cases all that's required is a brief document that summarises the results from the requirements analysis phase and adds the sample web pages provided by the web designer.
- Document the relevant non-functional requirements.

Architecture sketch

- Provide an architecture sketch that documents the important decisions regarding the overall software architecture, including the integration of frameworks and third-party tools.

Design documentation

- Document the content model, including the attributes and meta attributes for every content type.
- Document the relevant design decisions. Web project teams are often relatively small. In an attempt to avoid frequent and costly documentation updates, it can be a good idea to document the relevant design decisions only at the end of a release cycle (but not in advance).

User Documentation

Content management manual

- Provide a brief manual for content editors in addition to the documentation of your content management system. The focus of the manual should be on concrete usage guidelines for your specific site.
- Offer interactive workshops to demonstrate use of the editor client.

Operations manual

- Provide a brief operations manual. Typical contents include server configurations, tool configurations, resource requirements etc.

Launch / Relaunch

A web project needs to be concerned with launching or relaunching the site. This covers aspects of deployment, testing, content migration and management. The following work packages summarise these issues.

Deployment / Staging

Installation

- Install all tools (content management system, database, search engine, personalisation engine, shop software, community software, etc.) and frameworks in all environments (development environment, staging environment, live environment).
- Perform all necessary configurations.

Deployment

- Establish the necessary software configuration techniques (version control, build mechanisms, etc.).
- Devise mechanisms for deploying content delivery software to the application servers of the staging environment or the live environment.
- Devise mechanisms for deploying configuration files to the staging environment or the live environment.
- Devise mechanisms for deploying workflow configurations and the like to all content editor installations.

→ ONE WEB APPLICATION FOR CONTENT DELIVERY (5.1)

→ DEDICATED DEVELOPMENT AND PRODUCTION ENVIRONMENTS (5.2)

→ SMOOTH RELAUNCH (5.3)

Testing

- Establish integration tests and performance tests in the staging environment.

Content Migration

Migration

- Analyse the need for possible content migration as a consequence of the evolution of the content model.
- Develop tools for content migration if necessary.

→ DEDICATED DEVELOPMENT AND PRODUCTION ENVIRONMENTS (5.2)

→ SMOOTH RELAUNCH (5.3)

Relaunch Management

Processes

- Plan deadlines for software development, unit tests, staging, integration tests, go-live.
- Plan periods of content freeze if necessary (in the case of content migration).
- Coordinate all efforts between site owners, content editors and software developers.

→ SMOOTH RELAUNCH (5.3)

Choosing a Content Management System

The patterns in this book are independent of specific technologies or specific tools. In particular, they don't assume any specific content management system. But at some point you may have to make the decision for a specific tool, either when you set up the software components for a new site or when you consider replacing the content management system used for an existing site.

Content management systems vary greatly. There are plenty of systems on the market, and they differ with regard to scope, usability, technology, licence model and more. Some support a clean and sustainable software architecture better than others. You need to find a content management system that suits your needs.

The following checklist presents a series of questions that can help you evaluate a content management system. This doesn't mean that there's any one single correct answer to any of these questions. There isn't – different kinds of solutions are possible and requirements differ from one project to the next. Still, evaluating a tool against this checklist should give you a feel for how well the tool matches your requirements.

The checklist addresses a wide range of topics. Figure 44 gives a short overview. Throughout the checklist references to individual patterns appear whenever the application of a pattern is directly affected by a feature that a content management system may or may not have. In these cases, it may be worth analysing the extent to which the tool in question will allow you to implement the solution the pattern suggests. In this way the checklist should help you select a tool that offers the best possible support for setting up a software architecture that is well designed, sustainable and technically sound.

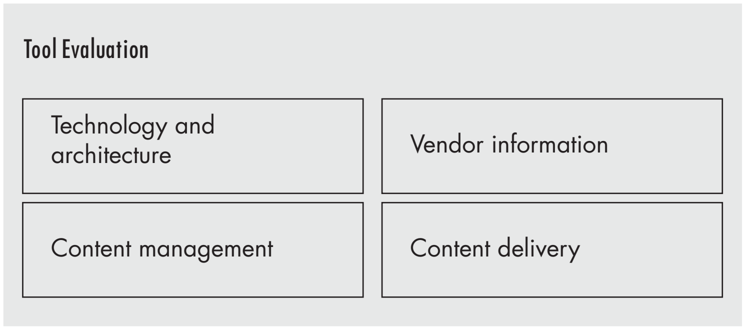


Figure 44: Areas of tool evaluation

Technology and Architecture

Technology

- What is the basic underlying technology (Java, PHP, Ruby, others) and how does it fit into the technology prevailing in your organisation?
- What formats are used for storing content (XML, proprietary formats)?

Architecture

- How is the separation of content and layout implemented?
- How is dynamic page generation and dynamic delivery supported?
- Does the system offer any caching mechanisms? If so, which?
- Is there an API that offers access to the content repository?
- How can custom software be integrated?
- Is it possible to register custom repository listeners that react to workflow events?

→ CONTENT MANAGEMENT AND CONTENT DELIVERY (1.1)

→ DYNAMIC CONTENT DELIVERY PLUS CACHING (1.2)

→ LISTENER-BASED SYNCHRONISATION (1.4)

→ LAYERED ARCHITECTURE FOR CONTENT DELIVERY (1.5)

System components

- What are the system's major components?
- Does the system include its own search engine? If not, is it possible to integrate a search engine into the content delivery components?
- Does the system include its own personalisation engine? If not, is it possible to integrate a personalisation engine into the content delivery components?
- Can shop software be integrated? How?
- Can community software be integrated? How?
- If the system is used alongside other web applications, is it possible to integrate a single sign-on component?

Non-functional properties

- What performance can the system guarantee? Are any measurements available?
- Does the system scale to a larger number of users? Can an installation be distributed over several machines? How exactly?
- What security mechanisms does the system include?

System requirements

- What operating systems can be used?
- What database system can be used?
- What web server and application server can be used?
- What are the hardware requirements?

Vendor Information

Licence model

- What kind of licences are available (depending on the number of users, depending on the number of processors, etc.)?
- Is the system an open source tool?
- Is it possible to have separate installations of the system in the development, the staging and the live environments? How many licences would that require?

Support and references

- What kind of support does the vendor offer?
- Are any reference websites available that are implemented using the system in question?

Content Management

Content modelling

- How can content models be specified? How powerful can these models be? What kinds of relationships can be expressed between content types? Is there a meta model for content modelling?
- How are content models made known to the system?
- What meta attributes are available? Which of them are managed automatically (author, creation date, publication date, etc.)?
- What support for content classification exists (tags, categories)?
- What content types are possible? Can multimedia objects or other blobs be handled?
- Is it possible to let one content type inherit from another? Is it possible to define 'abstract' content types?

→ Content Type Hierarchy (2.1)

→ Taxonomy Based on Keywords and Categories (2.4)

Link management

- Is there internal link management? Are content elements prevented from deletion if they are referenced from other content elements? Is the system able to avoid dangling (internal) links?
- Can links to external sources be checked?
- Is there a method of finding all referrers to a specific content element (that is, all content elements that hold a reference to it)? If so, is this method available to the custom software components?

→ Content Type Hierarchy (2.1)

→ Decoupling of Content and Navigation (2.2)

→ Dynamic Content Linking (2.3)

→ Decoupling of Edited Content and User Contributions (4.5)

Content evolution

- Is it possible to apply different content models to separate content partitions at the same time (as can be necessary if the content model evolves)?
- How else is the evolution of content models supported?

→ Smooth Relaunch (5.3)

Content import and export

- How can content be imported?
- How can content be exported?

Editor client

- What kind of editor client is provided? Is there a client application? If so, what is the roll-out process like? Is there a web client?
- Is it possible to customise the editor client (or clients)? How? To what extent?
- Does the editor client provide any built-in content validation mechanisms? Which?
- Does the editor client provide a spelling checker? In what languages?
- Does the editor client meet requirements for ergonomics?

→ WORKFLOW-BASED VALIDATION (2.5)

Workflow

- Is it possible to specify custom workflows (a four-eye workflow as well as others)?
- How can permissions be specified for content editors?
- Can custom repository listeners be connected to arbitrary workflow events such as creation and publication?
- What other kinds of notification are supported?
- Does the editor workflow include any built-in content validation mechanisms?
- Is it possible to integrate custom validation mechanisms?

→ LISTENER-BASED SYNCHRONISATION (1.4)

→ WORKFLOW-BASED VALIDATION (2.5)

Content Delivery

Controller logic

- Does the system offer support for content delivery, for example controller logic that maps incoming page requests onto template invocations?
- Alternatively, is it possible to use web frameworks such as Spring, Struts (in the Java world) or Ruby on Rails (for Ruby) to provide the necessary controller logic?
- If the system offers some controller logic, how can this logic be integrated into an application server or a servlet engine?

→ CONTENT SERVICES (3.1)

→ NAVIGATION MANAGER (3.2)

Domain logic

- Is there an easy and straightforward way to integrate custom components that implement the domain logic?
- Does the system support a service-oriented architecture?
- How can custom components use the API for accessing the content repository?
- Does the system provide any lookup mechanisms for content elements (that is, mechanisms that retrieve content elements not known by their internal id, based on their attributes or their path in the repository)? How fast are these lookup mechanisms?

→ CONTENT SERVICES (3.1)

→ NAVIGATION MANAGER (3.2)

Page generation and page delivery

- Does the system offer any mechanisms for generating web pages from content elements, such as a system of templates?
- Are users free to define their own templates? How?
- Is it possible to set up a hierarchy of templates that call each other?
- Does the system support multiple sites (for example intranet and Internet), multiple output channels, multiple output formats, multiple languages? How?

→ SYSTEM OF INTERACTING TEMPLATES (3.4)

→ TEMPLATE PER VIEW (3.5)

Search engine

If the system includes a search engine:

- What kind of content elements can be indexed (text, XML, PDF, blobs)?
- Will only meta attributes be indexed, or is a full-text search possible, for example for text and PDF objects?
- What meta attributes are covered by a search (author, publication date, tags, categories)?
- How fast is the search engine?

If the system does not include a search engine:

- What search engines are recommended? Are there any that are known to integrate well with the system?

→ TAXONOMY BASED ON KEYWORDS AND CATEGORIES (2.4)

→ SEARCH MANAGER (3.3)

Personalisation engine

If the system includes a personalisation engine:

- How can content be assigned to individual users? Does the personalisation engine implement concepts such as user roles or user segments?
- Does the personalisation engine operate in synchronous or asynchronous mode?
- How does the system support caching in the presence of personalisation? What system performance can be expected?
- Does the system offer any support for monitoring the delivery of personalised content?

If the system does not include a personalisation engine:

- Is it possible to integrate a third-party personalisation engine?
- If so, how exactly? How will the persistent storage of that personalisation engine be synchronised with the content repository?

→ ASYNCHRONOUS PERSONALISATION ENGINE (4.2)

→ SEGMENT-SPECIFIC CACHING (4.3)

→ CONDENSED EFFECTIVENESS REPORTS (4.4)

Final Remarks

A decade ago web projects were a challenge, not least because the technology was new and little experience was available. Although today the technology it still evolving, the web isn't new any more: by now there is loads of content on the web, there are web applications all over the place, and there are many sites for web-based user collaboration.

Given the large number of web projects that have been carried out over the last couple of years, it is no surprise that a few good practices have emerged. This doesn't mean that a web project cannot still be a challenge, but at least there is some experience to draw on.

The patterns in this book present this kind of experience. The whole idea of patterns is to describe what has worked well in the past, and here we are with a collection of twenty-five patterns that have been mined from a series of successful web projects.

The patterns address content modelling and management, content delivery and presentation, personalisation and user involvement, deployment and infrastructure, and so cover a large spectrum of topics that are relevant to a web project that keeps a focus on content. In addition, the two checklists should serve as a practical guide for planning a web project and for an evaluation of commercially available content management systems.

Obviously web technology is still moving fast, and future solutions will look different from those today. The focus of this book is on design principles, however, not on specific tools or technologies. The patterns represent solid experience, have long-term relevance, and should put you in a position to design your own solutions. I hope that the patterns prove to be helpful for your projects and that you will enjoy using them.

Pattern Thumbnails

For convenience, the following pages contain thumbnails of the individual patterns in the book. You can use these thumbnails to get an overview of the pattern collection as a whole, or to gain a first impression of individual patterns that you find interesting.

Architecture Overview

Content Management and Content Delivery (1.1)

How can you accommodate both the users' and the content editors' needs?

Provide software for two distinct purposes. On one hand, you need content management software that supports the content editors in their job. On the other, you need content delivery software that makes content available to the web and controls possible user interaction. You won't have to develop the complete software yourself – a content management system typically provides some of the necessary functionality – but you must expect to develop a certain amount of custom software.

Dynamic Content Delivery plus Caching (1.2)

How can you ensure that the site is always up-to-date and reflects the latest changes made by the content editors? How can you lay the foundation for interaction and personalisation?

Combine dynamic content delivery with powerful caching strategies. Choose a content management system that generates web pages on request and offers caching mechanisms sufficient to meet your performance requirements.

Sensible Client-Side Interaction (1.3)

How can you ensure that your site features the desired degree of interaction and user participation while maintaining reasonable system performance?

Use Ajax-based client-side interaction, but use it with care. Retain the concept of a web page and apply server-side event handling for all navigation purposes, but also apply event handling inside the browser to adjust the way in which information elements are presented within a web page. Combine this with asynchronous server calls if the browser has to load data from the server.

Listener-Based Synchronisation (1.4)

How can you avoid inconsistencies between content in the repository and content stored by other components?

Establish repository listeners – asynchronous processes that react to specific workflow events and notify interested components of relevant changes made to content artefacts in the repository.

Layered Architecture for Content Delivery (1.5)

How can you prevent the server-side custom software for content delivery from becoming difficult or impossible to maintain? How can you avoid a server-side architecture that doesn't scale properly?

Define a server-side architecture that consists of three distinct layers. The bottom layer encapsulates all access to the content repository. The middle layer provides the domain logic. The top layer contains the templates that are used for page generation.

Content Management

Content Type Hierarchy (2.1)

How can you ensure that content editors can maintain artefacts that are meaningful to them in a way that is straightforward and avoids redundant information?

Introduce a content model in which content types represent domain-motivated artefacts consisting of basic building blocks such as texts, pictures, multimedia objects and links. Apply object-oriented modelling techniques to express abstraction and association relationships between content types.

Decoupling of Content and Navigation (2.2)

How can you ensure that content editors can organise both the content and the navigation hierarchy in a straightforward and flexible way? How can you support different content hierarchies for different sites, like an intranet and an extranet, or for different countries or distribution channels?

Decouple the navigation hierarchy from the content. Introduce dedicated navigation nodes that span the navigation hierarchy. Allow the navigation nodes to be attributed with configuration information if the navigation hierarchy needs to vary across different contexts.

Dynamic Content Linking (2.3)

How can frequently changing content be maintained without burdening the content editors with the tedious job of manually linking the new content into the navigation hierarchy?

Establish dynamic lists that, instead of maintaining links to any content elements, specify criteria for potential list items, meaning that when a page featuring a dynamic list is generated, the repository is searched for content elements that match these criteria.

Taxonomy Based on Keywords and Categories (2.4)

How can you lay the foundations for an effective and powerful search function?

Build a taxonomy that combines arbitrary keywords with well-defined content categories. Make sure that individual content elements can be attributed with lists of keywords. In addition, establish a set of possibly overlapping content categories to which individual content elements can be assigned.

Workflow-Based Validation (2.5)

How can you avoid content elements with illegal or inconsistent attribute values?

Apply validators that check content for plausibility. Integrate these validators into the content editors' workflow in two ways: validators that reject illegal attribute values should be applied during the editing process, while validators that check for completeness and consistency should only be applied on publication.

Content Delivery

Content Services (3.1)

How can you avoid domain logic being scattered all over your server-side components?

Implement services that provide the content aggregates that the various pages of your site may require. This begins with single content elements from the repository, but also includes lists or compositions of content elements. If necessary, content services must apply personalisation. Whatever the content services make available, it must be ready to be processed by the presentation logic.

Navigation Manager (3.2)

How can you prevent templates and other view components from being burdened with the calculation of navigation-related information?

Establish a navigation manager that provides the various kinds of navigation-related information that your site requires. Evaluating relationships between content elements and mapping navigation nodes onto URLs, the navigation manager encapsulates the link management for your site.

Search Manager (3.3)

How can you implement a powerful and user-friendly search function and reduce the number of expensive search queries?

Implement a search manager that receives and handles all search requests throughout a user's session. The search manager contacts the search engine and makes the search results available to the components that will process them. For convenience, the search manager may store previous queries and results, and it can check queries for plausibility before forwarding them to the search engine.

System of Interacting Templates (3.4)

How can you avoid, to a large extent, redundant template code and inefficient page generation?

Define a system of interacting templates, from templates for the full page down to templates for individual page elements. Reuse templates for smaller page elements wherever possible. Make sure you extract any state-specific or personalised page elements into templates of their own, as this increases the potential for caching HTML fragments.

Template per View (3.5)

How can you support the different views that content elements may assume on different pages, for different variations of your site or for different output channels?

Define a template for each distinct view a content type has to support. Typical examples include a full view, a teaser view, a text-only view or views for specific output channels such as mobile devices. This way all individual templates can be quite simple, responsible only for assembling the markup necessary for one specific view of a content element.

Self-Contained Pages (3.6)

How much client-side interaction is sensible in terms of usability? How can you deliver the required client-side components to the browser?

Ensure that you create self-contained pages – pages that are sufficiently rich with content that visitors don't have to leave a page to look up closely related information. This is especially important for pages that feature multimedia objects that require a specific time to be viewed. Use Ajax components to implement the necessary interaction on these pages and integrate the client code into your templates.

Personalisation and User Participation

Content Filters (4.1)

How can you let content editors define personalisation strategies?

Implement filters for content selection. Integrate these filters into the domain logic for your site: implement services that apply these filters to deliver content only if it matches an individual user's profile. Allow content editors to express personalisation strategies by configuring how the content filters work in detail.

Asynchronous Personalisation Engine (4.2)

How can you implement personalisation strategies in an efficient way?

Identify the complex algorithms necessary for assigning content to individual users or user groups. Take these algorithms off line to a dedicated personalisation engine that runs asynchronously and independently from the application server that handles HTTP requests.

Segment-Specific Caching (4.3)

How can you avoid efficiency problems with content delivery in the presence of personalisation?

Identify content elements that are specific to user segments rather than individual users. If the number of segments is not large, implement segment-specific content services and segment-specific templates, and apply caching to the resulting personalised content elements and HTML fragments.

Condensed Effectiveness Reports (4.4)

How can you provide content editors with effective feedback on the success of the personalisation strategies they have implemented?

Measure the effectiveness of personalisation strategies by comparing the click ratio of personalised teasers with that of non-personalised ones. The click ratio relates the number of times a teaser's underlying link was followed to the number of times the teaser was displayed. Use a reporting tool to apply metrics and condense the results.

Decoupling of Edited Content and User Contributions (4.5)

How can you invite user-generated contributions without disturbing workflow processes for edited content?

In your overall content model, decouple edited content from user contributions, whether they are user-generated content, tags, ratings or anything else. Avoid references from edited content to user contributions, and maintain the necessary relationships between edited content and user contributions as attributes of user contributions alone.

Input Channel for User-Generated Content (4.6)

How can you ensure that user-generated content fits well into your content base?

Accept user-generated content only if it is submitted in a format that you endorse. Appropriate formats typically include the markup used by wikis, as well as standard formats for pictures and videos. If necessary, apply automatic transformations to adapt user-generated content to meet specific requirements.

Deployment and Infrastructure

One Web Application for Content Delivery (5.1)

How can you establish a straightforward deployment process for your content delivery software?

Define a single web application for all components directly involved in content delivery. This includes the custom components that you have developed for content retrieval, domain logic, personalisation and HTML generation, as well as the necessary configuration data. Define separate web applications only for loosely coupled tools, such as a standalone search engine or a personalisation engine connected through asynchronous communication.

Dedicated Development and Production Environments (5.2)

How can you evolve your website while ensuring its undisturbed operation?

Establish separate environments for software development and for the live system. Make sure that these environments can work on different content models if essential changes to the content model have to be expected. This may require the introduction of logical partitions for your content repository, or even separate installations of your content management system.

Smooth Relaunch (5.3)

How can you avoid conflicts when relaunching your site?

Apply staging to ensure that you can deploy a new software version to the production environment while the old version is still up and running. If necessary, migrate the live content in the staging environment. Launch the new version by a simple change of configuration, so that an equally simple change allows you to revert to the old version should anything fail.

Glossary

This glossary explains the relevant terminology used in the context of content management and content delivery. For some terms the meaning given here is specific to the context of this book, and therefore more specific than it would be in a broader context.

Ajax	Asynchronous *JavaScript* and *XML*. A collection of techniques that allow the browser to handle user input efficiently. Ajax leads to faster and richer user interaction, but is sometimes criticised for conceptual flaws and security drawbacks.
Caching	Storage optimisation techniques that provide fast access to frequently used objects (for example *content elements* or *web page* fragments).
Click ratio	The number of times a link underlying a page element is followed relative to the number of times that page element is displayed, usually expressed as a percentage.
Community software	Software that facilitates collaboration among communities with shared interests. Examples include forums and chatrooms.
Content	The sum of digital artefacts that are valuable to an organisation, including text, pictures, multimedia objects, as well as relations and dependencies between these objects. Referred to as *web content* if the intention is to make these artefacts available on the web.

Content delivery	Techniques and processes for distributing and delivering *content* to its target audience. The target medium is often, in the case of *web content*, the web.
Content editor	A person who creates, maintains and publishes *content* for an organisation, often with the purpose of making the *content* available on the organisation's *website* or *web platform*. The content editor's tasks usually follow a specific *workflow*.
Content element	A single *content* artefact, like a text, a picture, a video or a multimedia object. Content elements are usually organised by their *content type* and contain both attributes for the actual *content* and attributes for *meta information*.
Content management	1. Techniques and processes centred around the creation, maintenance, publication and delivery of *content*. 2. Techniques and processes centred around the creation, maintenance and publication of *content*, as opposed to techniques and processes summarised by *content delivery*. (The meaning assumed throughout this book.)
Content management system (CMS)	A software system that supports aspects of *content management* and *content delivery*. A content management system consists at least of a *content repository* and an *editor client* that supports the editor's *workflow*. In many cases, a content management system will also offer mechanisms or frameworks for content delivery, or additional tools such as a *search engine* or a *personalisation engine*.
Content model	A model that explains what *content types* exist and how they are related. It is driven by the requirements and the characteristics of the application domain.
Content migration	Processes that support the adjustment of existing *content* to the evolution of the underlying *content model*.
Content repository	A persistent storage for *content elements*. A content repository is typically hosted by the database that underlies the *content management system*.

Content type	Description of the content elements of one kind. A content type specifies a set of attributes (for storing the actual content) and a set of *meta attributes* (for workflow-related information).
CSS	Cascading style sheets, a widespread technique for assigning layout to *HTML pages*.
Demilitarised zone (DMZ)	A security mechanism that locates a web server in a network zone with limited access permissions and protects application servers and databases by shielding them from the outside world.
Deployment	Techniques and processes to take code live or, in the case of a *website*, to *launch* or *relaunch* the site.
Dynamic content	In the context of *web content*, *content* that underlies editing processes and *workflows*. *Web pages* are usually created from elements of dynamic content by an automatic *page generation* process.
Dynamic delivery	The delivery of *dynamic content* or, more often, *web pages* generated from *dynamic content*. Dynamic delivery usually involves *page generation* processes that use *content elements* as their source and apply domain logic to generate the results. Dynamic delivery is much better at presenting accurate and up-to-date *content* than static delivery. Any performance problems that result are often alleviated by *caching* techniques.
Editor	→ *content editor*
Editor client	A tool that allows *content editors* to create, maintain and publish *content*. An editor client implements specific editorial *workflows*. It can be stand-alone or browser-based – in the latter case it is often referred to as a *web client*.
Explicit personalisation	*Personalisation* strategies based on personal data or personal preferences actively provided by *users*.

Folksonomy	A blend of the words 'folklore' and 'taxonomy', a folksonomy offers a classification scheme for *content elements* based on criteria provided by the site's *users*, especially by *tags* that are attributed to *content elements* for description purposes.
HTML	Hypertext Markup Language, a markup language for the specification of *web pages*.
HTTP	Hypertext Transfer Protocol, the standard protocol for client-server communication on the web.
Implicit personalisation	Personalisation strategies based on observed user behaviour, such as *users*' *navigation*, interaction or purchasing habits. Usually more powerful than *explicit personalisation*, but sometimes criticised for a lack of respect for users' privacy.
Indexer	That part of a *search engine* that is in charge of receiving and storing information about individual *content* elements, for example content attributes, *tags*, *keywords* or *meta information*.
Java	A programming language common in web development. Used mostly for implementing domain logic.
JavaScript	A scripting language used to implement client-side user interaction directly within the browser. Part of the *Ajax* technology, JavaScript code is embedded into *web pages* and executed on loading a page.
JSP	Java Server Pages. A *scripting* language used for *web page generation* in a *Java*-based architecture.
Keyword	A term that is passed to a search function as a parameter. The search function returns objects that match the keyword, either because the keyword appears in a text object (full-text search) or because it matches specific *meta attributes*, such as *tags* that are assigned to the resulting object.
Launch	The process of taking a *website* or *web platform* live.

Meta attributes	Workflow-related information describing individual *content elements*, such as author, creation date and publication date. Typically assigned automatically by the *content management system*.
Meta information	→ *meta attributes*
Navigation	The set of *HTTP* links that connect the individual pages of a *website*. Almost all *websites* feature a tree-like navigation hierarchy with the start *page* at the root and pages with more specific *content* in the branches of the tree. The navigation hierarchy allows *users* to travel through a *website* in a structured way.
Output channel	Target media for *content delivery.* Typical examples include the web, an intranet or mobile devices.
Page	→ *web page*
Page element	An individual part of a *web page*, such as the main *content* area, a *navigation* bar, a logo etc.
Page generation	A set of processes that generate *web pages* from *content elements*, often applying domain logic along the way.
Personalisation	Strategies that aim to tailor *content* specifically to individual *users* or *user* groups. This begins with *content* that matches preferences specified by *users* themselves and ends with strategies that aim to select *content* based on observed behaviour of *users*' *navigation*, interaction or purchasing habits.
Personalisation engine	A software module designated specifically to implementing *personalisation* strategies, for example *content recommendations*.
PHP	A *scripting* language used mainly for *web page generation*.
Query engine	The part of a *search engine* that returns those *content elements* that match the search criteria specified in a search query.
Rating	An evaluation scheme usually applied to *content elements*, especially for assets that are sold in an online shop. Ratings are typically provided by users rather than by content editors.

Recommendation	A special *personalisation* strategy that seeks to make a *user* aware of *content* that is expected to be particularly relevant or interesting to the user.
Recommendation engine	A software module designated specifically to implementing *recommendation* strategies.
Relaunch	The process of taking a *website* or *web platform* live after a major revision or redesign.
Rendering	→ *page generation*
Repository	→ *content repository*
Search engine	A software module that supports the search for specific content elements. A search engine consists of two major components. The *indexer* receives information about individual *content elements*, for example attributes, *tags*, *keywords* or *meta information*. The *query engine* returns those *content elements* that match the specified search criteria.
Scripting	ASCII-based non-compiled programming instructions. Typical examples of scripting code include *JSP* code and *PHP* code.
Staging	A concept for testing a website under simulated live conditions before it actually goes live. After the site has been developed it is deployed to a staging area that resembles the live environment, where quality control takes place.
Static content	In the context of *web content*, *content* that is stored as *web pages* and or fragments of *web pages*. Once made available, static content rarely changes and no automated *page generation* process is implemented for its delivery to the web.
Tag	A term that is attributed to a *content element* for classification purposes. Assigned either by *content editors* or *users*, tags are part of a content element's *meta information* and can be used to make search functions effective.

Template	1. A blueprint for *web pages* of the same layout, usually provided a *web designer* in the form of *XHTML* documents and *CSS* files. 2. Code that generates *web pages* or page fragments from *content elements*. A template defines the layout for the web pages it creates and so becomes part of the *page generation* process for *dynamic delivery*. (The meaning assumed throughout this book.)
Template engine	A software module that executes *template* code, such as a servlet engine for *JSP*s, a *PHP* interpreter or an XSLT processor.
User	A person who uses a *website* or *web platform*. This covers people who merely consume information, but also includes people who submit *user-generated content* to a site that invites *user participation*.
User-generated content	*Content* provided by users, as opposed to *content* provided by *content editors*.
User interaction	The interaction between the *user* and the *website*. The necessary communication between browser and server can be implemented either with traditional HTTP or with Ajax.
User participation	The sum of processes that actively involve users in a *web platform* with the purpose of inviting user contributions or allowing user collaboration. User participation can take on different forms, including the submission of *tags*, *folksonomies*, *ratings* or *user-generated content*.
Web application	A software application made available on the web. Although a web application may feature some *web content*, its focus is on providing functionality and interaction, as opposed to a *website* or a *web platform* where the focus is on making *content* available. There is only a thin line between the two, as advanced *websites* or *web platforms* may involve *user interaction* and *user participation* and do require a certain amount of functionality.
Web client	A browser-based *editor client*.

Web content	*Content* specifically target at the web. In the case of dynamic web content, the web is the major *output channel* for the *content delivery* process.
Web design	The sum of tasks around the layout of *web pages*. Web design may include aspects of usability engineering and ergonomics, but has nothing to do with *content management*.
Web designer	A person in charge of *web design*, as opposed to a content editor who is in charge of content, its structure and its relations, but not its layout.
Web page	An HTML page that a user receives as a response to an HTTP request.
Web platform	A *website* with a special emphasis on *user participation*.
Website	A set of related *web pages*, usually connected by links and a common navigation structure, as well as the mechanisms for user interaction and *user participation* that the web pages may use.
Workflow	The entirety of tasks to be performed by *content editors*, organised in a way that reflects the importance of the individual tasks and the dependencies between them.
XHTML	Extensible Hypertext Markup Language. A more rigid version of *HTML* that conforms to the structure of *XML*. Generally preferred over *HTML* for reasons of increased reliability and browser independence.
XML	Extensible Markup Language. A meta language for the construction of custom markup languages whose principle purpose is sharing structured information in an easily-parsable form.
XSLT	XSL Transformation. A language for the transformation of *XML* documents. Part of the Extensible Stylesheet Language (XSL) family.

References

www.agilemanifesto.org

The Agile Manifesto. http://www.agilemanifesto.org.

www.amazon.com

Amazon. http://www.amazon.com.

Anderson 2006

Chris Anderson. *The Long Tail – Why the Future of Business is Selling Less of More*. Hyperion, 2006

Beck 2004

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2004.

Berners-Lee Hendler Lassila 2001

Tim Berners-Lee, James Hendler, Ora Lassila. The Semantic Web, in *Scientific American* (available at http://www.scientificamerican.com). May 2001.

Blair Maron 1985

David C. Blair, M. E. Maron. An Evaluation of Effectiveness for a Full-Text Document-Retrieval System, in *Communications of the ACM*, Vol. 28, No. 3. ACM, March 1985.

Broemmer 2003

Darren Broemmer. *J2EE Best Practices – Java Design Patterns, Automation, and Performance*. John Wiley and Sons, 2003.

Buschmann Meunier Rohnert Sommerlad Stal 1996

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture, Vol. 1 – A System of Patterns*. John Wiley and Sons, 1996.

Cockburn 2002

Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2002.

Dyson Longshaw 2004

Paul Dyson, Andy Longshaw. *Architecting Enterprise Solutions – Patterns for High-Capability Internet-based Systems*. John Wiley and Sons, 2004.

Fowler 1999

Martin Fowler. *Refactoring – Improving The Design Of Existing Code*. Addison-Wesley, 1999.

Fowler 2003

Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

Gamma Helm Johnson Vlissides 1995

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Garrett 2005

Jesse James Garret. Ajax: *A New Approach to Web Applications*. http://www.adaptivepath.com/publications/essays/archives/000385.php, February 2005.

Gusfield 1997

Dan Gusfield. *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology.* Cambridge University Press, 1997.

Hackos Redish 1998

JoAnn Hackos, Janice Redish. *User and Task Analysis for Interface Design*. John Wiley and Sons, 1998.

Hackos 2002

JoAnn Hackos. *Content Management for Dynamic Web Delivery.* John Wiley and Sons, 2002.

www.icefaces.org

ICEfaces. http://www.icefaces.org.

Kerth 2001

Norman Kerth. *Project Retrospectives – A Handbook for Team Reviews*. Dorset House, 2001.

Krug 2006

Steve Krug. *Don't Make Me Think – A Common-Sense Approach to Web Usability.* New Riders Press, 2006.

Leuf Cunningham 2001

Bo Leuf, Ward Cunningham. *The Wiki Way – Quick Collaboration on the Web.* Addison-Wesley, 2001.

lucene.apache.org

The Apache Lucene Project. http://lucene.apache.org.

Mader 2008

Stewart Mader. *Wikipatterns*. John Wiley and Sons, 2008.

Mahemoff 2006

Michael Mahemoff. *Ajax Design Patterns*. O'Reilly, 2006.

Morville 2005

Peter Morville. *Ambient Findability – What We Find Changes Who We Become*. O'Reilly, 2005.

Neville-Neil 2007

George V. Neville-Neil. Building Secure Web Applications, in *Queue – Architecting Tomorrow's Computing*, Vol. 5, No. 5. ACM, July / August 2007.

labs.jboss.org/jbossrichfaces

RichFaces http://labs.jboss.org/jbossrichfaces/.

Rockley 2002

Ann Rockley. *Managing Enterprise Content – A Unified Content Strategy*. New Riders Press, 2002.

Rosenfeld Morville 2006

Louis Rosenfeld, Peter Morville. *Information Architecture for the World Wide Web – Designing Large-Scale Web Sites*. O'Reilly, 2006.

www.rails.org

Ruby on Rails. http://www.rails.org.

Rüping 2003

Andreas Rüping. *Agile Documentation – A Pattern Guide to Producing Lightweight Documents for Software Projects*. John Wiley and Sons, 2003.

Schumacher Fernandez-Buglioni Hybertson Buschmann Sommerlad 2006

Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, Peter Sommerlad. *Security Patterns – Integrating Security and Systems Engineering*. John Wiley and Sons, 2006.

Schümmer Lukosch 2007

Till Schümmer, Stephan Lukosch. *Patterns for Computer-Mediated Interaction*. John Wiley and Sons, 2007.

Schwaber Beedle 2008

Ken Schwaber, Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2008.

Scott Neil 2009

Bill Scott, Theresa Neil. *Designing Web Interfaces – Principles and Patterns for Rich Interactions*. O'Reilly, 2009.

Shirky 2008

Clay Shirky. *Here Comes Everybody – The Power of Organizing Without Organizations*. Penguin, 2008.

www.springframework.org

The Spring Application Framework. http://www.springframework.org.

struts.apache.org

The Apache Struts Project. http://struts.apache.org.

Tidwell 2005

Jenifer Tidwell. *Designing Interfaces – Patterns for Effective Interaction Design*. O'Reilly, 2005.

Vogel Zdun 2006

Oliver Vogel, Uwe Zdun. Content Conversion and Generation on the Web: A Pattern Language, in Dragos Manulescu, James Noble, Markus Völter (Eds.), *Pattern Languages of Program Design, Vol. 5*. Addison-Wesley, 2006.

Völter Schmid Wolff 2002

Markus Völter, Alexander Schmid, Eberhard Wolff. *Server Component Patterns – Component Infrastructures Illustrated with EJB*. John Wiley and Sons, 2002.

Wellhausen 2005

Tim Wellhausen. Query Engine – A Pattern for Performing Dynamic Searches in Information Systems, in Klaus Marquardt, Dietmar Schütz (Eds.), *EuroPLoP 2004 – Proceedings of the 9th European Conference on Pattern Languages of Programs, 2004*. Universitätsverlag Konstanz, 2005.

Weiss 2006

Michael Weiss. More Patterns for Web Applications, in Andy Longshaw, Uwe Zdun (Eds.), *EuroPLoP 2005 – Proceedings of the 10th European Conference on Pattern Languages of Programs, 2005*. Universitätsverlag Konstanz, 2006.

www.wikipedia.org

Wikipedia – The Free Online Encyclopedia. http://www.wikipedia.org.

Index