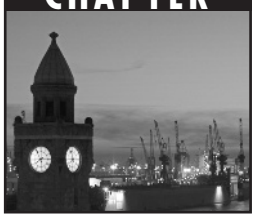
CHAPTER



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.

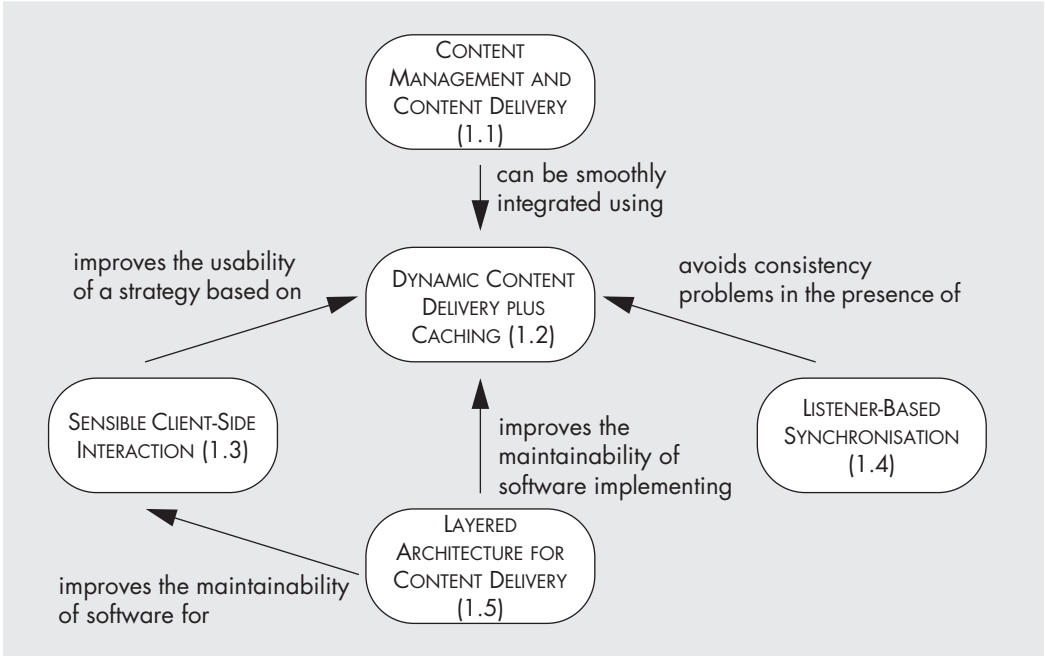


Figure 4: Road map to the patterns for the architecture overview

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

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 arte-

facts, 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.

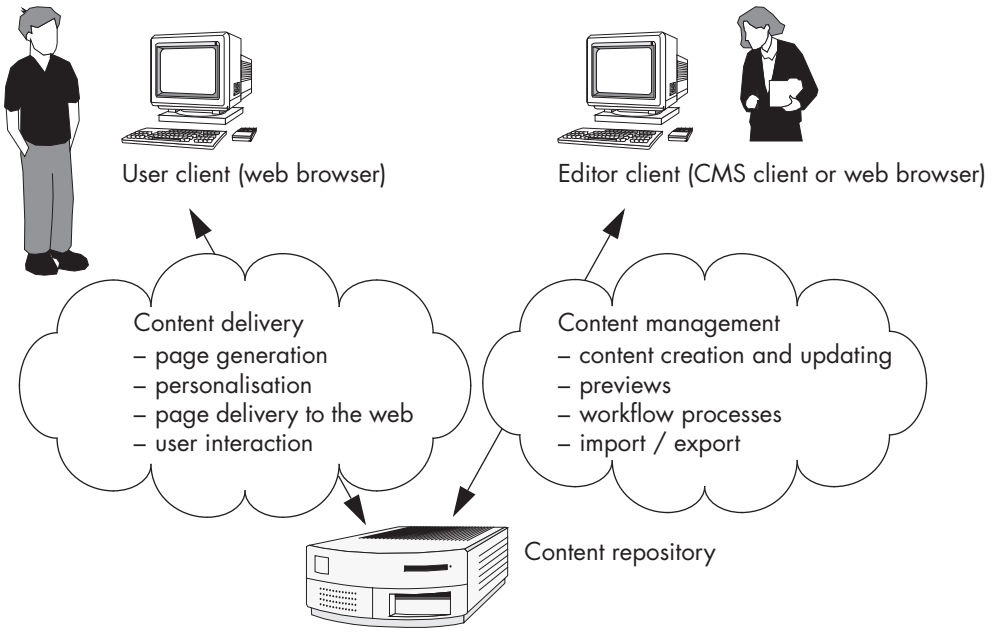


Figure 6: Content management and content delivery

- 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

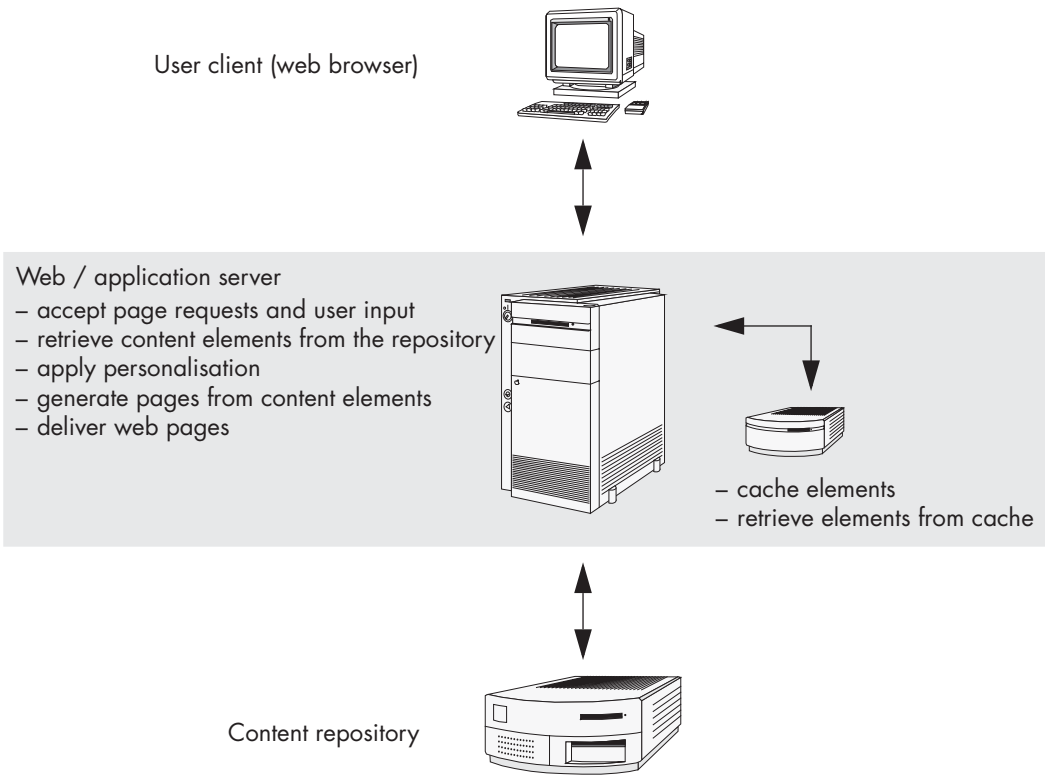


Figure 7: Dynamic content delivery and caching

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?

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.

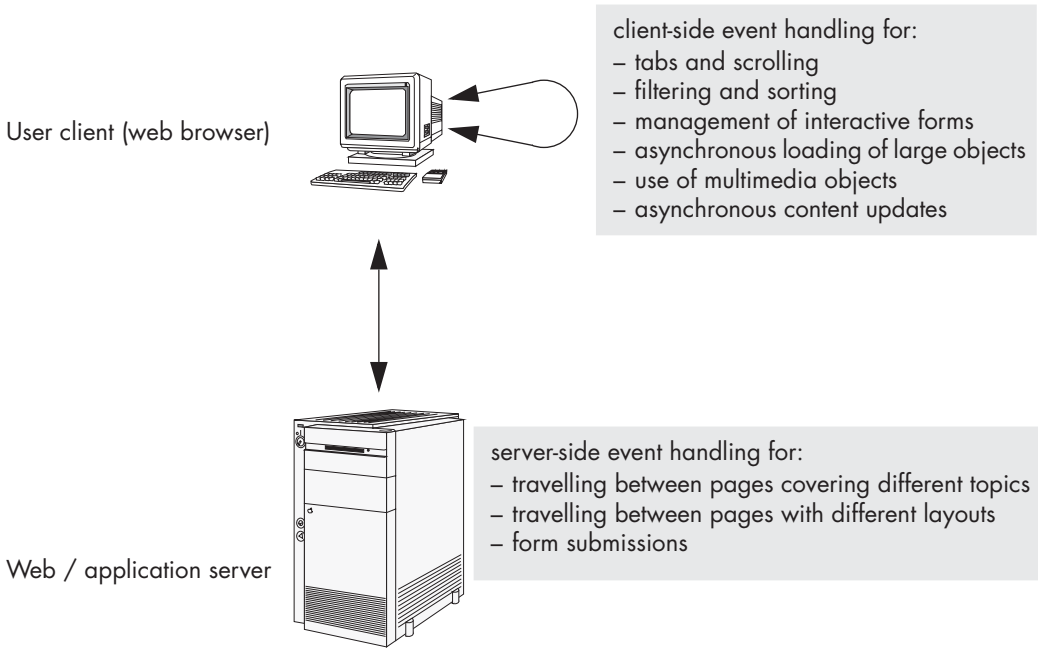


Figure 9: Event handling for user interaction

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]

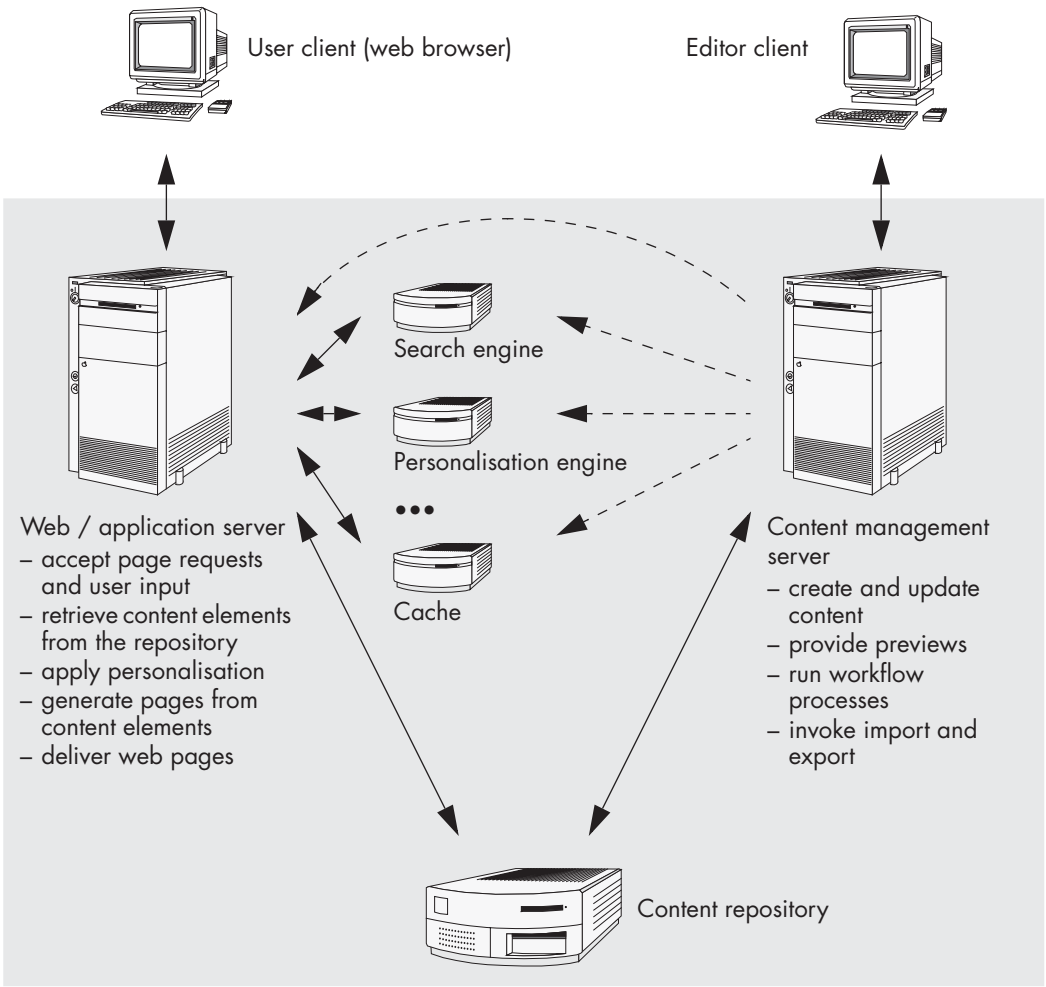


Figure 10: Repository listeners

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

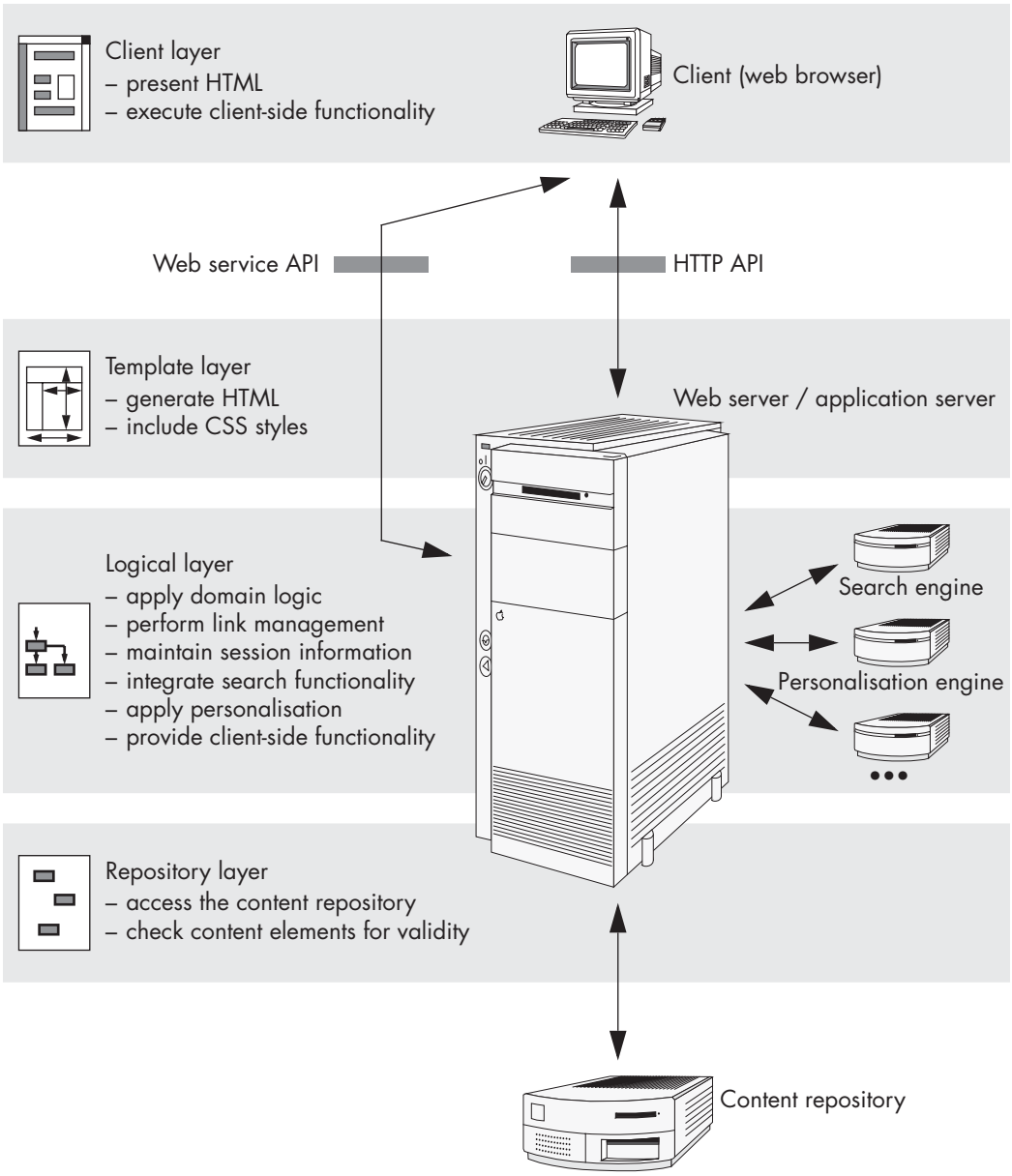


Figure 11: Layered architecture for dynamic content delivery

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).