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]

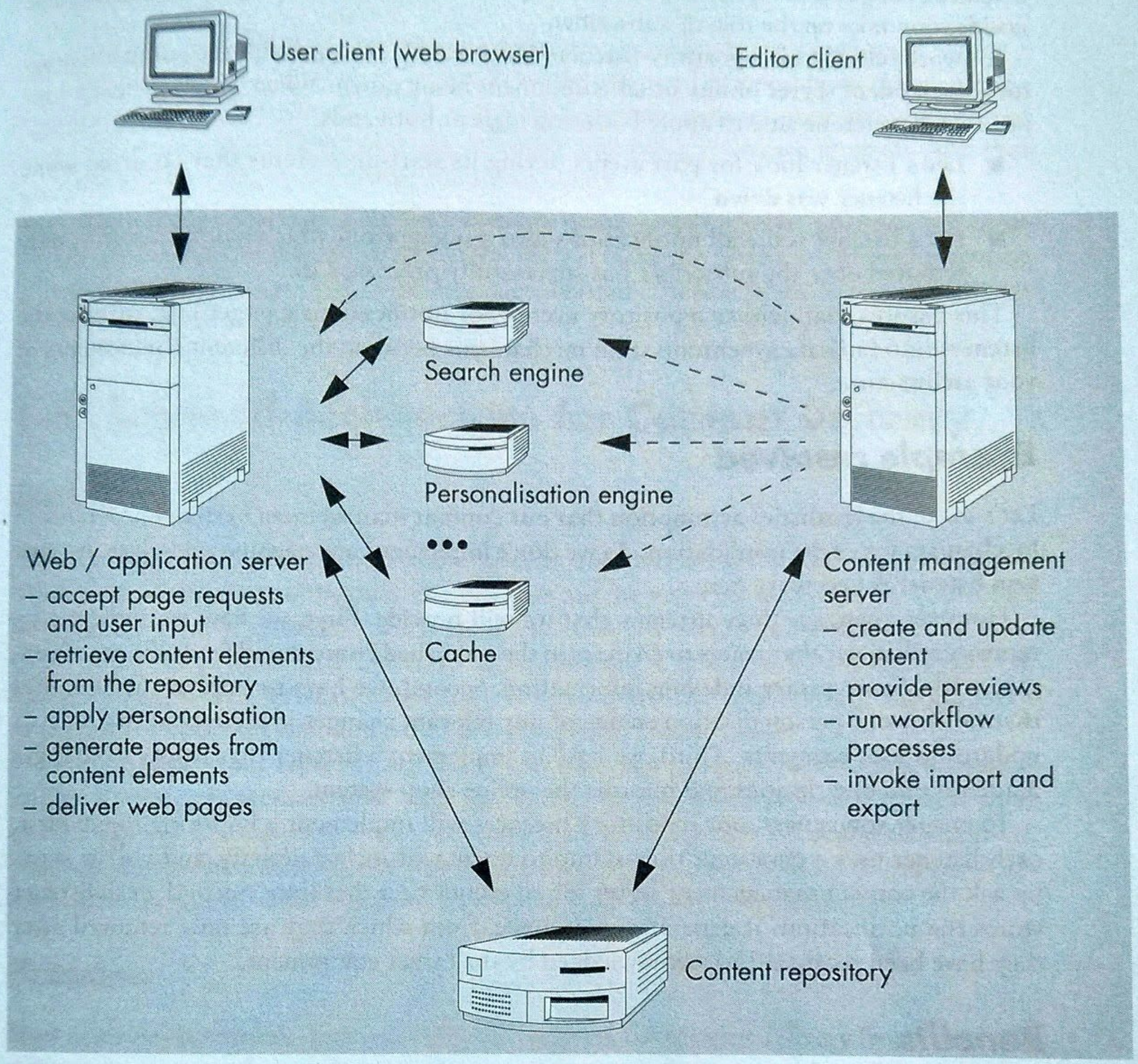

Figure 10: Repository listeners

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.