Caching Strategy Design for WebSphere Commerce, Part 1: Implementation of a Content Delivery Network (CDN)

Andon Salvarinov  
Howard Borenstein  
Victoria O. Odeyemi  
Marco Deluca  
Kevin Yu  
Ali Asghar  

September 25, 2014

In Part 1 of this series, you learn about caching implementations of a CDN network on a WebSphere® Commerce Server (WCS), caching strategy, system operations, and code design and implementation with respect of the CDN caching network. Additional parts of the CDN learning series will be available soon.

View more content in this series

Overview of the Content Delivery Network (CDN)

CDNs act as intermediary between web applications and their users. CDNs transparently mirror content from customer servers, and serve the files to the users. The benefit is that users can download/access the files from the closest CDN server, ensuring optimal access and minimal overhead in the content delivery. This also lowers the impact of the network congestion and outages. CDN provides several advantages over regular network such as caching, optimal routing, SSL and non-SSL content separation, congestion and outage avoidance. However, the most used benefit is caching of the content locally within the network servers. The examples that are sited are best-known CDN implementation at the time of publication but can change later.

Caching Principles on CDN

CDN is a hierarchical network of thousands of servers that sit between the user browser and application servers. In this case, the network is organized in multiple layers, where the top layer is closest to the app servers and has the lowest number of servers. This layer acts as root in the hierarchy. The middle layer and the leaf layer are comprised by exponentially higher number of servers. Each layer has increasing number of servers, but the ratio of child to parent servers varies depending on the region to meet the user traffic demand.
Basic caching and request processing flow with CDN

The User request always takes the path from the leaf to the root on the hierarchy, where the application servers present the root of the graph structure. On the path through the hierarchy, the request might encounter a valid cached content, which is returned without propagating the request further down to the application servers. On the way back, the response content is being cached (copied locally) on the nodes of the network passed.

Your implementation may have separation of the SSL and non-SSL traffics to separate networks. The secure network infrastructure is often smaller in capacity when compared to the non-secure infrastructure, matching the ratio in the current Internet-wide delivery protocols. There is no predetermined path for the user request, and it can go to the origin via any of the reachable servers. However, because the hierarchy is implemented on one-to-many relationship, ensures that the content is cached.
HTTP versus HTTPS Akamai subnetworks
For example, Akamai offers various levels of offload for traffic to Akamai servers when the pages and content needs to be delivered over secure a connection. Since secure information never resides on any disk within the Akamai network, the Akamai edge server must first retrieve secure content from the origin server over a secure connection before it is delivered to the user. SSL objects and non-secure content can be cached on the Akamai edge server, eliminating the need to retrieve content on every user request. The result is to move almost all SSL interactions as close as possible to the user, drastically reducing load on the origin infrastructure.

Tip
It is highly advisable to use HTTPS protocol only if it is needed and any browse scenario should be run exclusively on non-secure network. Such setups lead to the optimal CDN response time and cache repository that can handle high traffic volumes and large cache pools.

In general, Akamai can deliver page resources over a secure connection (while the base page is not secure) and can deliver complete pages that are on HTTPS (secure) connection. In both cases, the Akamai allows caching of the content on the Akamai network even when it is on HTTPS connection.

CDN Caching
CDN content delivery networks like Akamai provides several optimization functions and algorithms. In akamai’s case, these optimization functions and algorithms are provided under the DSA name (Dynamic Site Accelerator) umbrella. While they all impact the site performance, availability, and capacity, the focus of this article is on the most frequently used caching feature with Akamai CDN for building WCS Sites.
The simplest form of the CDN caching is to implement static-like approach to cache the dynamic content that is generated on the appservers, with simple invalidation logic to prevent serving stale content to the users. In the typical WCS design, the most beneficial pages to be cached are catalog pages (home, categories, and product pages) as these pages are the most frequently retrieved pages on the site. To achieve this, the following conditions must be met:

1. The cacheable pages cannot contain personalized content.
2. The cacheable pages cannot contain dynamic content with high frequency of changes.

If these criteria are met, pages can be set to become part of the static cache pool with faster refresh rates (several hours). However, the first criterion is a difficult condition to meet since most sites have some form of personalization. For WCS, the level of attainable personalization is large and the chances are CDN caching on the catalog pages is not possible without extra effort to split and deliver personalized content from the non-personalized content.

**ESI Caching**

Edge Side Includes (ESI) is a simple markup language that is used to define web page components for dynamic assembly and delivery of web applications at the edges of the Internet. ESI provides a mechanism for managing content transparently across application server solutions, content management systems, and content delivery networks such as Akamai.

An example of ESI implementation would be to enable page composition at the edge server, where none personalized part of the page is cached locally, and the personalized part of the page is retrieved from the origin. The operation of assembly is transparent to the user, and in the case of the WebSphere® commerce server (WCS) implementations require relatively little effort to implement.

The ESI protocol is generally implemented as ESI tags and ESI request headers, which in the case of WebSphere® Application Server is generated and driven from DynaCache/Appserver runtime logic. The OTB (out-of-the-box) support for ESI headers can be enabled by following:

1. Each JSP must be declared as a full page or fragment “EdgeCacheable” in cachespec.xml:
   ```xml
   <property name="EdgeCacheable">true</property>
   ```
2. JSP must be requestable from external server with ESI, by defining alternative URL in cachespec.xml:
   ```xml
   <property name="alternate_url">/.../your.jsp</property>
   ```
3. Use only URL parameters and custom cookies. Cannot contain attributes in CacheID.
4. For web Server ESI cache or Caching Proxy Server ESI, declare "esiEnable" in the WebSphere® Server plug-in configuration file:
   ```xml
   <property name="esiEnable" value="true" />
   ```
   in the plug-in-cfg.xml file under WAS-installdir/config/cells/ directory

While the OTB integration is a significant step in the right direction and makes everything relatively easy to implement and achieve, caching based on ESI protocol might not be the most optimal approach to cache on CDN. The following experiences were encountered in practice.
- Page response time is often not improved since most of the personalized fragments need to be generated on the appserver and cannot be cached locally on the CDN servers.
- Additional constraints on the JSP execution create numerous issues and customization for ESI fragment execution. The constraints are comparable to Ajax calls.
- Modern, wider available techniques are constantly changing the page assembly (moving it to the browser/user end) rendering ESI obsolete.
- ESI is not providing any mechanism for invalidation of the content. While there are several mechanisms derived in the ESI specification, CDN implementation of these APIs might be limited.

Overall, the current limitations of ESI Caching make it a less preferred caching mechanism for today's modern sites that generate vast varieties of personalization and ever-changing implementation.

**CDN Caching with Ajax**

Modern web implementations are leaning to assemble the pages at the user end (browser) by using Ajax calls to retrieve bits and pieces of the information or perform some action on the user session in the application server. The page assembly in this case is performed on the browser end, at cost of multiple HTTP requests going on through CDN. Since Ajax call is just another HTTP call, all the caching artifacts and techniques that exist on the CDN to cache the full page are directly applicable to make Ajax calls cacheable on the CDN as well.

**Tip**

CDN caching with Ajax is the typical preferred approach to enable modern WCS site implementation for CDN caching.

This strategy enables the separation of the type of the content on the e-commerce site to make it cacheable with different caching and invalidation rules. This is typical approach to prepare/enable modern WCS site implementation for CDN caching. For successful caching of the dynamic pages on CDN, the content of the base page should be non-personalized, readily reusable for any user, while the user-specific content is retrieved via Ajax calls to the application server.

**Ajax versus ESI**

An obvious question is how ESI and Ajax are compared from the overall efficiency of the implementation. The following comparison clarifies the data delivery efficiency by simple comparison between ESI, Ajax approach, and third method of delivering the information – Cookie-based data delivery:

**Table 1. Efficiency comparison of ESI and Ajax**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Frag. Caching</th>
<th>Cookie</th>
<th>Ajax Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Difficulty</td>
<td>Simple</td>
<td>Simple to Complex</td>
<td>Complex</td>
</tr>
<tr>
<td>Performance Impact</td>
<td>Small</td>
<td>None or Negligible</td>
<td>Small to Moderate</td>
</tr>
</tbody>
</table>
Cookie-based data delivery is an indispensable method of implementing simple personalization that is user-specific only. A typical example of this is minicart data, or welcome message that appears on virtually every WCS e-commerce site.

The comparison is done on 4 different aspects:

1. Ease of implementation, where the ESI is at advantage that requires minimal involvement, followed by cookie data delivery method. The Ajax method is considered most involving in the implementation phase since the cost of browser end assembly falls into the development phase.

2. Performance impact on the site, where the performance impact represents the overhead of the application server to manage the data when the requests are not cached. Obviously the ESI approach has a minimal cost, followed by cookie method that has small overhead. At last is the Ajax approach, due to the multiple HTTP requests needed to assemble the page is most costly.

3. Cache Infrastructure Load or cost of the implementation w.r.t. cache maintenance and overhead in the system. Cookie-based delivery is most efficient since it does not generate any cache content, while ESI and Ajax are at same cost since both approaches are generating about the same number of cached objects in the cache pool.

4. CDN implication, where the method is evaluated from the CDN standpoint in terms of required functions and complex caching implementations. Cookie and Ajax approach does not require any specifics at the CDN level, while ESI method requires ESI support and more complex management of the data in the cache.

In conclusion, high volume and high performance e-commerce sites would most benefit from Cookie delivery method in terms of performance. However, this method has limitations and might not suffice for rich content. Fragment Caching with ESI is relatively natural extension of the existing fragment caching in DynaCache, but it fails short in delivering full caching solution at the edge server near the user due to the ESI caching constraints. If fragment is not cacheable, then the overall response times and execution cost are on same level as sites that do not implement CDN caching (for dynamic pages). Ajax approach resolves both cookie and ESI constraints – in the case of the ESI limitations Ajax approach can be considered as separate HTTP request that can be micromanaged by separate caching rule.

**Tip**

High volume and high performance WCS sites would most benefit from Cookie delivery method in terms of performance to cache CDN related contents.

High volume e-commerce sites with big catalog tend to use more complex caching strategies to achieve overall performance of the site. This might result on higher workout of the caching infrastructure that can lead to saturation of the cache component capacity. Hence, excessive page
fragmentation on such high performance high volume sites does not scale well, and fragments need to be limited or eliminated by using cookie or Ajax approach.

**Invalidation of the Cached Content**

A typical attribute of a cached content is its lifetime, a time during which the cached content is valid and can be served to the user without having their request that is processed by the application server. A cached content can become stale as result of various asynchronous events, and requires refresh before being served to the user. A refresh is usually induced by invalidation event, which effectively removes the cached content from the cache pool. Thus, the invalidation part of the caching strategy holds the burden of ensuring correctness of the site during the site lifetime.

**Tip**

A sound invalidation process is the most critical part of a caching strategy to ensure correctness of the cached content and optimized cache reuse.

**CDN Invalidation Mechanisms**

CDNs provide several mechanisms and methods to achieve content invalidation on the CDN network. They are generally include two groups:

1. **Indirect** - time-based mechanisms as TTL and Scheduled TTL where the content expires after some time it was created (in the case of the TTL) or expires at preset time (scheduled TTL).
2. **Direct** - event-driven invalidation mechanisms that can invalidate content on demand.

**CDN event driven invalidation mechanisms**

For example Akamai, provides two basic mechanisms to meet different needs and preferences, and the basic mechanisms have various interfaces that you can use:

- **Content Control Utility (CCU)** – Allows invalidation of objects by specifying a full URL or predetermined group (by CP code). The CCU can be used as a web interface (manual) or SOAP-based API (automated)
- **Enhanced Content Control Utility (ECCU)** - allows specific objects to refresh by regex expression rules using path and extensions or other, more complex criteria. There are three ways to use ECCU: Web interface (manual), File upload with invalidation rules uploaded to Akamai, and ECCU Publisher that automates ECCU File upload via SOAP API.

**Invalidation Limitations**

Time based invalidations (TTL and Scheduled TTL) are transparent to the site operation and work well on typical implementation. However, they might not be most efficient nor desirable due to the invalidation rules that spans big chunks of the cached content. In addition, Scheduled TTL induces spikes of requests on the origin when it is happening. Furthermore, this type of invalidation cannot be tightly integrated with the site operation.
Event-based invalidation allows for better integration with the site operation and maintenance. However, CDN event based invalidations might have time delays in propagating invalidation requests across the CDN network.

**WCS: DynaCache and CDN Caching**

The Content Distributed Network (CDN) Caching Service can be represented as invisible proxy between the WS Commerce server and user. It provides extensive features and capabilities to cache web content at point closest to the user. This ensures best possible response time for the user as well as significant capacity and performance improvement of the actual implementation.

High performance WCS e-commerce site implements Caching Strategy on two caching layers: CDN and DynaCache. DynaCache represents the caching infrastructure that is found in the IBM® WebSphere® Application Server product, while CDN Caching Service (for example, Edge Server Network) is usually external service provider (for example Akamai).

Each layer can set its own caching rules according to the following directions:

- Distributed Network Caching Service (Akamai) can cache static content and some of dynamically created pages.
- DynaCache caches only dynamic content.

The DynaCache cashing component is part of the WS Commerce and its caching definitions are defined in the xml metadata file (cachespec.xml). It automatically maintains, manages, and shares the cached content across the app servers in a node. It uses request parameters as well as session data to create cache keys/dependencies and efficiently manage the cached content.

The DynaCache cashing component is understood and documented, for more information see This PDF can serve as primary guide in creating efficient DynaCache caching layer. High performance WCS implementation relies heavily on correct and efficient DynaCache cashing implementation, which is deemed imperative and irreplaceable. CDN cannot substitute DynaCache cache layer without significant performance degradation and potential instability of the site.

CDN caching layer has more restrictions and limitations compared to DynaCache caching layer, with cache implementation that is usually has less sophisticated rules when compared to DynaCache. In general, the WCS application implements caching of the following rules on CDN:

- Static Files Caching
- Dynamic Pages/DHTML with user agnostic data
- Dynamic Pages/DHTML with user-specific content, but the user-specific content is delivered via Ajax call that is not cached.
- Ajax calls that are serving user agnostic data.

To achieve good efficiency on CDN cache, it is imperative to have an application with a data delivery schema that is matching these rules. These rules are discussed in bigger detail in section 3 of the document, via typical problem that is encountered on the field and solution that can overcome the problem.
The discussion of CDN and DynaCache cooperation assumes that you are well versed in the DynaCache caching component, cache rules, and general operation of the caching in WCS Application server. This article explores the similarities and differences between these two layers, and focuses on high performance implementation of CDN caching for WCS server.

WCS e-commerce site can implement Caching Strategy on two caching layers: Edge Server Caching and DynaCache. DynaCache represents the caching infrastructure that is found in the IBM® WebSphere® Application Server product, while Distributed Network Caching Service (e.g. Edge Server Network) is usually external service provider.

Each layer can set its own caching rules according to the following directions:

- Distributed Network Caching Service can cache static content and some of dynamically created pages.
- DynaCache caches only dynamic content.

Distributed Network Caching Service can be represented as invisible proxy between the WS Commerce server and user. It provides extensive features and capabilities to cache web content at point closest to the user. This ensures best possible response time for the user as well as significant capacity and performance improvement of the actual implementation.

**Comparison between DynaCache and CDN**

A simple comparison between the DynaCache and CDN crucial features that are governing the caching rules and caching strategy on both layers is seen in Table 2.

**Table 2. Simple comparison of DynaCache and CDN**

<table>
<thead>
<tr>
<th>Feature</th>
<th>WCS/DynaCache</th>
<th>CDN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selective URL parameters in cache key</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>WCS session data in cache key</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Cookies as part of the cache key</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>WCS Command Caching</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>JSP/fragment caching</td>
<td>Yes</td>
<td>Yes, via ESI</td>
</tr>
<tr>
<td>Explicit values for Cache keys/inclusion/exclusion lists</td>
<td>Yes</td>
<td>Limited.</td>
</tr>
<tr>
<td>Invalidation that is based on TTL *</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Invalidation that is based on Scheduled TTL *</td>
<td>No (possible via API/code implementation)</td>
<td>Yes</td>
</tr>
<tr>
<td>Direct (API) invalidation call **</td>
<td>Yes (immediate)</td>
<td>Yes, with capacity imitations</td>
</tr>
</tbody>
</table>

*) TTL and Scheduled TTL stand for Time-To-Live invalidation that is evicting the cached object after predetermined time has passed (regular TTL) from the object creation, while Scheduled TTL allows eviction at explicit specified time. The regular TTL is present at both layers, while scheduled TTL is supported only on CDN. Scheduled TTL can be implemented on DynaCache via API code, and it is relatively simple effort.
**) Both CDN and DynaCache provide interface to evict cached objects: DynaCache via Java™ API, while CDN provides SOAP/HTTP interface for event eviction requests. Both of them can achieve precise invalidation, or can evict predefined group of objects. The CDN interface has limitations on the invalidation volume. For more details on invalidation see, Invalidation.

Caching and session data

When you define a cache rule, the crucial factor of its efficiency is correct and precise construct of cache key id – a value that is comprised from multiple data points that are found in the HTTP request (for example URL parameters and cookies). One of the major differences between CDN and DynaCache is that CDN cannot use WCS session data to generate cache key for request.

WCS session data is not available outside the application server and cannot be used in the CDN caching layer. This presents constrain and limitation on CDN caching rules, leading to mismatch between the cache objects (and rules) defined in each layer. For special cases, this limitation can be overcome by exposing the session data in HTTP cookie and use it as part of the CDN cache.

WCS Application server can already provide limited exposure of the session data via such mechanisms as Edge Caching Cookies.

Edge Caching Cookies

WCS V6.0 or later provides CDN cookies that can expose limited session data to the external world via cookies. The working of the mechanism that is implemented are explained in details in the WCS documentation on the information center, here it suffices to clarify the type of the data that gets exposed, and rules of the invalidation implemented.

Edge Cookies are targeting a small subset of the session that is crucial to Akamai caching, but steering away from the user personal and sensitive data. For example, exposed data is language ID, Currency ID, Member Id userType. Each of these cookies is selectable, and if they are not needed they can be disabled and not emitted in the ether.

The edge cookies content is encrypted on highest level, and is never decrypted to determine what is in the cookie. This allows extra security that none would be able to determine and reuse some cookie value in the future. Furthermore cookie content is regenerated every hour, flushing the old cookies from the cloud to limit the time of potential data exposure.

Here is complete list of available edge cookies and their content origin in Table 3

<table>
<thead>
<tr>
<th>Cookie Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WC_LANGID</td>
<td>Language ID</td>
</tr>
<tr>
<td>WC_CURRID</td>
<td>Currency ID</td>
</tr>
<tr>
<td>WC_PROG</td>
<td>Parent organization</td>
</tr>
<tr>
<td>WC_CACHEID1</td>
<td>Contract ID</td>
</tr>
</tbody>
</table>
Invalidation

Invalidation is a mechanism where stale cached objects are removed from the cache pool, leaving the next request to regenerate the up-to-date version of the cached content.

Figure 3 depicts typical WCS application and different layers of cache that needs to be invalidated to keep the cache from becoming stale.

Figure 3. Cache Invalidation

Types of Invalidation

There are several mechanisms that can be used to provide stale content invalidation, and some of them are available only on WCS locally or CDN. Here is brief review of the invalidation mechanisms available:

- TTL (Time to Live) invalidation is invalidation mechanism where the cached content is set to expire after predefined time interval. While easy to implement and manage, and is available in CDN and DynaCache, this approach provides least effective way of maintaining cache pool, and usually has poorest hit/miss ratio compared to the other solutions.
- Scheduled TTL is a variation of the TTL where the knowledge of the change events is easily predetermined to a fixed time slot in the day. This invalidation provides better hit/miss ratio,
but it still suffers from excessive cached content invalidation. CDN supports this as OTB, DynaCache can support it with proper code implementation.

- Dependency Based Invalidation where the cache invalidation might precisely target the stale content with these dependencies. Compared to TTL and Scheduled TTL, this approach maintains best hit to miss ratio and solves the problem of the excessive cached content invalidation. Akamai does not support this invalidation type, but provides dependency-like based invalidation via event driven protocol. DynaCache invalidation operates primarily on dependencies, and WCS implements both Command Based invalidation and fine grain invalidation.

Invalidation Strategy Guidelines

Keeping the volume of the invalidated objects minimal is primary goal to every efficient invalidation strategy for WCS implementation. Since the implementation can limit some invalidation mechanisms, caching implementations tend to differ in the invalidation strategy of the site. However, when you choose an available mechanism for invalidation the following guidelines are followed:

- Smaller invalidation set is better, where invalidating the minimally needed objects is optimal but not always possible. Less invalidation requires less recreation of cached objects, thus use less resources.
- Less frequent invalidation is better, where the regenerated content gets regenerated less frequently, conserving system resources.
- Event-driven invalidation (just-in-time) is superior invalidation to TTL types of invalidations.

Event-driven invalidation

Event-driven invalidation is most efficient type of invalidation that is:

- Invalidating only changed content, such that the volume of the removed cached content is minimal and resources are spent to regenerate only stale content.
- Invalidating on demand, when the change happens, instead of refreshing the content with some frequency, leaving the site with non-deterministic quantity of stale cache content between the refreshes.
- Is automated and integrated, leaving little to guess or manual procedure that would have high overhead and prone to failures.

The event-driven invalidation operates on 2 caching layers: CDN and DynaCache. The following diagram shows the event flow in the typical WCS Implementation:

The diagram depicts the event driven invalidation flow when you update content data. The content data in production environment is changing via stageprop operation (presented by CSRs drawing). As the operation perform, the changes are inserted into the DB tables (insert/update/delete). These changes are intercepted by the caching triggers in the database, that for each changes, insert one (or several) dependency ids into the cacheivl table. These dependencyIds are defining
the stale content. DynaCache Scheduler Job reads the CACHEIVL table for recent records and calls DynaCache invalidation API to invalidate by dependency Id.

**Figure 4. Event Driven Invalidation Flow**

At the end of the stage prop operation, the CDN invalidation emits the SOAP request to the CDN cloud to invalidate the content on the CDN network.

**Page Design for Edge Caching**

This section provides the principles on designing cacheable content for typical e-commerce application.

**Basic Principles**

CDN cache layer is relatively more limited compared to the DynaCache Cache layer about the context information it can use to cache a request. Major difference is that DynaCache cache layer has access to the session data information (and do make good use of it to create more cache keys) while CDN does not have access to the session data and must rely only on what is present in the URL and Request headers.

This effectively limits CDN to cache content that is same for all users, or subset of those users that are identifiable by special (application generated) cookies. This effectively leads to the following design approaches:

1. Keep user-specific content separated from the general content.
2. Separate more dynamic content from static like content early and make a decision on how to deliver it.
3. Define and design user-specific content with caching in mind.
4. Design page with content refresh in mind.
Page Fragments and their Implementation

Each page in a typical WCS implementation is constructed as set of JSP fragments that are assembling a page of the catalog for the e-commerce site. The cache of the page must not follow these coding imposed fragments, but establish logical fragments of content with same (similar) nature or requirements.

At the same time, the content is evaluated for any of the following cases (described further below) and ensure initial implementation that is cache-ready. For example, it is not always JSP fragment cache that is the best approach, sometimes more complex approach of assembling the page is preferred in the real life implementation.

For example, fragmenting the mini cart and caching it separately in DynaCache is not the best performance implementation, although it's a natural design that is based on the JSP fragments. More successful approach to this would be to deliver the mini cart content via Ajax request or, best approach is to use cookies to keep the minicart content for each customer in their own browser. The later achieves serving the mini cart content without the user having to send requests to the data center.

**Tip**
The most ideal implementation is one where the user request can be rendered on the client side, without a request needing to be sent. Next best thing is for the CDN to serve the content, without having the requests reach the data center.

The following sections discuss several typical designs that are performance optimal for content caching and delivery on CDN and DynaCache. They are frequently implementing complex measures to ensure optimal utilization of the resources across the the CDN (content delivery network), and software stack (WCS cluster).

User Specific Data handling

It's crucial for successful catalog caching to make the most frequently pages (home page, category page, and product page) clean from user-specific content and use separate delivery method and JavaScript™ to over-impose user-specific content in the browser. This approach would allow the content of the page to be cached on CDN and DynaCache for long time, while the user-specific content can use separate caching policy (if needed).

**Tip**
Render personalized content via Ajax is the recommended approach to enable caching of the remaining content.

**Example**
A typical user-specific content that is found on these pages is constituted of welcome message (welcome user_name), minicart, and login/logout links. These artifacts if left into the pages render them not cacheable on CDN. Typical solution for this case is to remove the user-specific content and distribute it via cookie that would contain the user-specific details, and use JavaScript™ to over impose (render) this content in the user browser as the page is rendered on the screen.
Page design with content that requires high refresh frequency

Some of the catalog pages might contain fragment that would impose frequent refresh rates of the cache, effectively limiting the benefit of the page caching. Since high refresh rates are serious performance overhead, this design does not excel in real life situation.

Such page designs can be improved if the high refresh content is separated from the rest of the pages (base pages) and delivered via separate means (Ajax calls, cookies, and so on). The important distinction here that cannot be overlooked is that this separation allows establishment of different caching policies for each segment of the page, thus keeping optimal the overhead of the page generation and ensuring the content is fresh.

When this design is implemented, the base page can still be fully cached on CDN and DynaCache with majority of the data (and rendering overhead) covered in this cache content. The Ajax call can be further cached either on DynaCache or both DynaCache and CDN.

Example

Typical example in this group would be pages that require near real-time inventory, without any lag or stale data. The design solution for this is to separate the inventory data (Ajax request) from the rest of the page (base page).

Hence, the base page can still be fully cached on CDN and DynaCache, while the Ajax call can be further cached either on DynaCache using the fine grain invalidation (or direct API calls for DynaCache).

Page design with personalized content

There are pages that display content that is highly personalized for the user that browses the site. WCS has wide variety of personalization functions, which is considered non-cacheable content.

A performance optimal approach to designing such pages would be to separate the personalized from non-personalized content and similarly to the delivery of the personalized content via Ajax calls.

Example

WCS allows for personalization that is based on user segments that are predefined on the system. We might consider a page that displays personalized eMarketingSpot that displays different content for each age group the site is targeting. This eMarketingSpot is separated from the base page and its content that is gathered via Ajax call. The Ajax call can always be cached if appropriate caching is possible for this personalization.

Page Design that Integrate external Vendors data.

Typical representatives for such pages are pages that interface to "bazaar voice", rich relevance, Coremetrics® tags, or Google® analytics. They share a common concept, an url to external resources that might or might not become part of the page. However, this URL contains user-specific data that needs to be managed for successful caching just like the other cases.
Typically, the non-visible external resources would be Coremetrics® and Google® analytics. In both cases, pages implement special URLs that would track the page usage data by starting the analytics URL with special parameters appended in the URL query. If the parameters contain user-specific data, this data gets cached into the base page cache. This is easily resolved by using special custom cookie that would contain the user-specific data for analytics and append the data at the URL query at the moment of starting the URL.

The external resources that deliver visible components on the page are similar caching issue as the nonvisible. Since the page loads these external resources at page load time, the cache (CDN or DynaCache) will not contain the page resources but only a link to the page resources. This ensures that you never cache external content that might get stale. However, the links (URLs) might contain user-specific data that should be managed via cookies to allow the base page caching.

**Page Design for “hard to cache” page artifacts.**

Frequently a page would be considered difficult to cache. However, in practice, such pages are not necessarily difficult to cache, but they impose poor caching efficiency (hit to miss ratio). A typical solution for these pages is to create many cached instances, that lead to a double impact on the system:

1. The poor hit to miss ratio consumes resources to render the content much more frequently than the other pages.
2. The large number of cache instances are polluting the cache pool, potentially causing more valuable pages to be removed from the cache due to the limited space in the pool.

In general, the content that has poor caching efficiency is easily identified. You separate this content from the page and retrieve it via separate means. This initial separation ensures that you get most out of the base page caching and optimally use the resources for it, while the content that has poor caching efficiency is left to use the resources.

**Example**

Store locator that are based on the user ZIP or postal code. Typical "hard to cache" content that would result in hundreds of thousands cache entries that burdens the cached pool and creates low efficiency cache with a poor hit to miss ratio.

**Design with consecutive improvements**

Fortunately, there is a better way to improve caching, and you should not consider any design to be the best, revolution, and more evolitional stepping stone to some better state.

Most implementation-specific rules that are causing these limitations are not fully understood with all the implications early, and unfortunately some of them needs real life experience to fully evaluate their impact of the overall performance of the site. Hence, the implementation should keep an eye of the overall complexity and future improvements and features that might be implemented.
Generally, simple, transparent solutions fare better than specific, complex, and custom solutions. Standard caching rules (via cachespec.xml), simple caching constructs with simple invalidation rules are much preferred long term from custom key managers in Java™ or Java™ calls to DynaCache API to manipulate the cache metadata. ESI headers and ESI is another example that is evaluated with the benefit versus complexity balance.

The complexity tends to grow as the site grows, and the number of special caching artifacts and cases are proportional to the complexity. A single master document that defines the special cases and informs how to manage the site is a must (HLD Caching Strategy Design Document).

**Caching Strategy for CDN network in an WCS application**

This is a possible CDN caching strategy for a typical WCS implementation on field. The focus is not to provide complete or boiler plate caching strategy for WCS site, but to discuss and explore the CDN caching artifacts on typical implementation and provide further insight and understanding.

**Caching of the static files**

Caching of the static files on the CDN network comes as no surprise to anyone, since CDN network has been doing this for long time. This is a tried and proven feature that is essentially a OTB feature of Akamai. There is no real need to change anything in the OTB caching rules for the static content for a typical WebSphere® Commerce implementation.

The only consideration you perform is to ensure that the SEO URLs (search engine optimized URL – WCS feature) are not colliding into the existing OTB caching rules for the static files.

**Caching of the semi-static DHTML pages**

Typical WCS implementation would generally consist of the following features in Table 4

<table>
<thead>
<tr>
<th>Feature</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minicart</td>
<td>A fragment that appears on every page and depicts the number of products in the page and total of the order.</td>
</tr>
<tr>
<td>Welcome message</td>
<td>A fragment that appears on the home page and depicts the shopper name as in &quot;Welcome John&quot; if the user is logged in, or just &quot;Welcome&quot; if the user is not logged in.</td>
</tr>
<tr>
<td>Login/Logout links</td>
<td>A fragment that appears on every page, and presents correct link: Login if user is not logged in or logout if the user is logged in.</td>
</tr>
<tr>
<td>Browse history</td>
<td>A fragment that usually appears on product and category pages and depicts the last few items user have seen.</td>
</tr>
<tr>
<td>eSpot</td>
<td>A fragment that presents targeted information, that might be personalized for the users.</td>
</tr>
<tr>
<td>Browse Pages (home, category, and product)</td>
<td>Base pages that constitute the catalog and most beneficial for caching.</td>
</tr>
<tr>
<td>Price data</td>
<td>A fragment that presents the price of the product. A page can have one (product) or many price fragments (category page)</td>
</tr>
</tbody>
</table>
Inventory data | Information that helps users to select the in-stock items only for purchase. It can be presented to the user as part of the page (in stock, out of stock and so on) or can be used to control the actions possible on page (if out of stock, add to cart would not be performed, or the product will not appear in the category pages, and so on).

### Table 5. Delivery methods in a typical WCS implementation

<table>
<thead>
<tr>
<th>Feature</th>
<th>Delivery Mechanism/Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minicart</td>
<td>Cookie Based, use JavaScript™ to render the information</td>
</tr>
<tr>
<td>Welcome message</td>
<td>Cookie Based, use JavaScript™ to render the information</td>
</tr>
<tr>
<td>Login/Logout links</td>
<td>Cookie Based, use JavaScript™ to render the information</td>
</tr>
<tr>
<td>Browse history</td>
<td>Cookie Based, use JavaScript™, and execute in Browser only.</td>
</tr>
<tr>
<td>eSpot</td>
<td>Consumed in the base page, or use Ajax HTTP request to retrieve the data.</td>
</tr>
<tr>
<td>Browse Pages (home, category and product)</td>
<td>Standard HTTP request</td>
</tr>
<tr>
<td>Price data</td>
<td>Ajax request that is frequently combined with the inventory data, aggregated on cases with multiple products</td>
</tr>
<tr>
<td>Inventory data</td>
<td>Ajax request, frequently combined with the price data.</td>
</tr>
</tbody>
</table>

**NOTE:** The details on the cacheability of each artifact on CDN, as well as cache key formation is omitted in this material in order to keep the focus on the CDN caching strategy. In real implementation, substantial effort is directed to the cacheability, cache key formation, and cache efficiency (hit to miss ratio).

### Invalidation

The presented e-commerce implementation provides a lot of flexibility on the caching strategy. A possible caching strategy would look like Table 6

### Table 6. Possible caching strategy

<table>
<thead>
<tr>
<th>Feature</th>
<th>Caching Strategy on CDN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minicart</td>
<td>N/A, caching functionality is performed by the browser</td>
</tr>
<tr>
<td>Welcome message</td>
<td>N/A, caching functionality is performed by the browser</td>
</tr>
<tr>
<td>Login/Logout links</td>
<td>N/A, caching functionality is performed by the browser</td>
</tr>
<tr>
<td>Browse history</td>
<td>N/A</td>
</tr>
<tr>
<td>eSpot</td>
<td>Cacheable unless the eMarketingSpot contains personalized content. For static eMarketingSpot content, 1 day TTL on CDN is ideal.</td>
</tr>
<tr>
<td>Browse Pages (home, category, and product)</td>
<td>Cacheable on CDN, 1day TTL and CCU invalidation on demand.</td>
</tr>
<tr>
<td>Price data</td>
<td>Cacheable on CDN with relatively short TTL , depending on the frequency of the price changes. If combined with the</td>
</tr>
<tr>
<td>Inventory data</td>
<td>Cacheable on CDN with very short TTL of 5to 10 min.</td>
</tr>
</tbody>
</table>
The presented caching strategy is relatively simple, and in most situations that are proven to be effective. It is important to understand the decisions that lead to this caching strategy. Here is a logical breakdown on the implementation and the CDN caching strategy as presented:

1. Personalized data is delivered via cookie. As much as cumbersome this method might appear in the first glance, the solution is irreplaceable on high volume e-commerce sites. It ensures that none of the cache layers (CDN or DynaCache) are "polluted" with personalized data required to render pages for thousands of users that are active on the site. Since all of these datapoints are unique for the user, the reduction in volume of the cached content in both caching layers is significant, allowing for much improvement in the cache efficiency (hit to miss ratio is many higher then other implementations).

2. eMarketingSpots delivered via Ajax call. eMarketingSpot is a page fragment that can present vast variety of information to the user. It can also be shared across multiple pages. Having it delivered over Ajax request, the caching solution ensures that eMarketingSpots can be managed outside the existing browse page cache as well as they can affect different invalidation policy from the base browse pages. This ability is indispensable if website needs to refresh eMarketingSpot content in the middle of the business hours – clearing all the eMarketingSpots does not cause major hit on the site performance because the bulk of the browse page cache is independent and stays present in the system.

3. Personalized eMarketingSpots delivered via Ajax request, would help separating the personalized content from the user-agnostic content (non-personalized content), allowing to implement different caching strategy (or not cache it as a last resort).

4. Browse Pages cache instances are freed from any content that can limit the cacheability of the pages and the invalidation strategy is mostly driven by the site content updates (which is in most cases happening once a day). A well-integrated commerce site would implement event driven invalidation of these pages, using the CDN event invalidation API call tied to the completion of the content update.

5. Short Lived data, as in Inventory and Price data is separated from the cached pages and delivered via Ajax call. This ensures that this part of the pages is refreshed with much shorter interval.

Limitations and Caveats

The presented caching strategy is typical of the WebSphere® Commerce implementation website.

- Invalidation on CDN is limited when compared to invalidation within a data center with DynaCache. You might not be able to cache user specific data due to this invalidation limitation. For example, instant validation is needed when tied to user action.
- CDN event based invalidation might also introduce a delay between the time the event is issued and the time the event message is propigated on the entire CDN network. It is important to plan for such delays. However, if delays are not acceptable you can use the scheduled TTL invalidation at a predetermined time of day.
- Edge caching cookies are not capable of supporting caching of personalized content.
Related topics

- For more about mastering DynaCache, see Product_Brief_Aqua.DSA.pdf
- Follow developerWorks Commerce communities.
- Evaluate IBM products in the way that suits you best.

© Copyright IBM Corporation 2014
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)