Architecture and strategies to cache dynamic content in Pramati Server

Anurag Parashar, Server Engineering

The speed at which web content is served is vital to enterprise-grade web applications. More often than not, it depends on the performance characteristics of the web server. While browsers and proxy servers do a neat job in caching the images and take advantage of HTTP headers like ETag and last-modified, these are good for caching static content. Dynamic Content Caching Architecture in Pramati Server allows administrators to set up flexible caching strategies and monitor behavior of dynamic web applications.
The Dynamic Content Cache Server

Dynamic Content Cache Server is designed as an interceptor in the Pramati Server Web Container. It boosts network throughput and access speeds through the Pramati Server Web Container. Key features of the Cache Server architecture are:

- No separate installable required and can be configured on an existing Pramati Server Web Container (Version 3.5 or higher) installation.
- Developers can add their own caching (and invalidation) rules to the existing set.
- Integrated with the JMX-based management framework.

Pramati Server Web Container provides a way to cache dynamic content in a declarative way. Pages that do not change often or tolerant to stale content can be cached by specifying them in a configuration XML. Rules can also invalidate cache to avoid storing stale content for infinite time.

How Dynamic Content Cache Server works

As an interceptor in the Web Container, the Cache Server intercepts all HTTP requests and passes them through a set of rules that determine caching, updation and invalidation. Caching rules exist as Rule Objects in a Rule Registry. Rule Objects analyze the request and decide what is to be done with it. When a HTTP request arrives for the first time, the Rule Object generates a key for it if it is cache enabled. The key is set on a task object. All content is cached against the key.

On a subsequent request for the content, the Rule Object uses the key to check the cache and serves up the content. If the content is not in the cache or has been invalidated for some reason, the key indicates to the Cache Server that the content is a candidate for caching and the cache is updated.

If the content is static, the Cache Server forwards the request to the file server. If the content is dynamic, the request is directed to the servlet engine. The request is processed and a response is sent. On the way out, if the task object contains a key, the response is cached against the key.

Runtime behavior of cache engine

At runtime, the Rule Object analyzes the HTTP request object to verify if that request satisfies the conditions specified in the rule. For example if a "time based caching rule" is specified as "Cache all responses for URLs like /admin/reports/\*.jsp for one hour from when the content entered the cache."

If a request comes for /admin/reports/load.jsp for the first time, the Rule Object generates a key that identifies the specific load.jsp request. The next time the same load.jsp request is received, the rule object generates the same key, checks the cache, compares the timestamp to verify if the content is stale and if valid returns the content to the engine.

Note: The key here differentiates an admin/reports/load.jsp request from /admin/reports/hits.jsp request. Though both requests satisfy the rule, the content for the two is different.

Content Cache is a shared by Rule Objects. Each Rule Object is allocated space by the Content Cache. The Cache itself is managed across Rule Objects like memory management and victimization algorithm.
What type of content can be cached

Not all dynamic content can be cached since stale information can misrepresent the content to the user of the application. A page can be cached if:

**Content does not change for a period of time.** An example can be an application data which gets modified on daily basis. The page need not be created on each request. The page can be cached with an expiry of 24 hours starting from the time on which the batch data gets updated.

**Stale content is affordable for some time.** Data may be getting updated continuously, but content need not be modified immediately on the site.

**Content changes based on a state change.** The content may be dependent on some state change say, a file or database modification. In that case the content can be cached until the event is received. The invalidation for these cases is done through customizable invalidation rules discussed later in this paper. This state change can be detected either on each request or by regular polling.
There may be cases where the application server has exclusive access to the application data forming the content of a view page. In such cases, caching of the view page can be triggered when a page updating the data of the cached page is requested. However, this kind of caching is really useful where the update page is accessed much less frequently than the view page.

*Key to dynamic content caching is maintenance. Cached content must be removed, or victimized, as soon as it turns invalid. Content can turn invalid in three ways: time-based, application-based and custom. These are discussed further up in this paper.*

**When not to cache content**

Caching may not be helpful if the page does not have much processing or database operations to perform. If the information on the page is sensitive and no stale information can be afforded, caching must be avoided.

If the page expects the response in chunks, it must not be cached. If the application uses logic that depends on response cookies that carry client-specific information, caching should be avoided for security reasons. In general, any page that has client-specific information should not be cached.

**Parameters that can be included in a caching rule**

A rule is composed from the following key parameters:

- **Rule Name.** A unique rule name given to identify the rule by the person who defines the rule.
- **Rule Description.** A brief description of the rule.
- **Request URI pattern.** URL pattern can be specified using the standard URL patterns mapping defined by the Servlet specification namely:
  - Extension mapping (*.jsp)
  - Directory mapping (/admin/reports/*)
  - Exact mapping (/catalog/prices/laptop.jsp)
In addition the Pramati Dynamic Content Cache Server supports an extended version of extension mapping—qualified extension mapping (/admin/reports/*view*.jsp).
- **Context root.** The context root under which the rule will be placed. If a request is coming for that context root, the rule will be evaluated for that request. If context root is not specified, the rule is considered to be a default rule which will be evaluated for all the request, whatever be the context root of the request. This kind of rules should be carefully used as they may lower the overall performance of the webserver.
- **HTTP method.** Dynamic caching is supported for two HTTP methods: GET and POST.
- **Enabled.** Shows if a caching rule is currently enabled or not. Disabling a rule does not delete a rule but just omits its evaluation. This can be enabled anytime in the future.
**input-params-in-key.** For a rule user can define to have all the input-parameters in the key or only the specified input-parameters in the key. So there are two values for this parameter i.e. "all" and "specified". By default if no value for the field is specified, it is "all".

**input-parameter.** This mapping may be specified for a rule if the response varies based upon the presence of an input-parameter or the value of it. It has following parts:

- include-as-key: User may also specify if it will be a part of the key or not if the value of input-params-in-key is "specified". If it is "all" then the value given here won't be applicable.
- name: Name of the input parameter has to be specified.
- exists: This parameter has three values. i.e. "mandatory" - the input-parameter has to be present for the page to be cached, "disallowed" - the input-parameter mustn't be present for the page to be cached and "optional" - this input-parameter is optional, but if it is present, its value must match the value in the rule for the input-parameter.
- value: the optional value for the input-parameter.

A typical input-parameter tag in the xml looks like this:

```xml
<input-parameter name="product" exists="mandatory" include-as-key="false">
    camera
</input-parameter>
```

**cookie.** This mapping may be specified for a rule if the response varies based upon the presence of a cookie or the value of it. It has following parts:

- include-as-key: User may also specify if it will be a part of the key or not. Default value is true.
- name: Name of the cookie has to be specified.
- exists: This parameter has three values. i.e. "mandatory" - the cookie has to be present for the page to be cached, "disallowed" - the cookie mustn't be present for the page to be cached and "optional" - this cookie is optional, but if it is present, its value must match the value in the rule for the cookie. Default value is optional.
- value: the optional value for the cookie.

A typical cookie tag in the xml looks like this:

```xml
<cookie name="product" exists="mandatory" include-as-key="false">
    camera
</cookie>
```

**request-header.** This mapping may be specified for a rule if the response varies based upon the presence of a request-header or the value of it. It has following parts:

- include-as-key: User may also specify if it will be a part of the key or not. Default value is true.
- name: Name of the request-header has to be specified.
- exists: This parameter has three values. i.e. "mandatory" - the request-header has to be present for the page to be cached, "disallowed" - the request-header mustn't be present for the page to be cached and "optional" - this request-header is optional, but if it is present, its value must match the value in the rule for the request-header. Default value is "optional".
• value: the optional value for the request-header

A typical request-header tag in the xml looks like this:

```xml
<request-header name="product" exists="mandatory" include-as-key="false">
  camera
</request-header>
```

Along with defining what to cache, rules for invalidating or removing cached content must be defined.

**Defining a negative rule to avoid caching**

In addition to a rule for caching, user can also define rules for not caching a page. This is especially useful when you do not want a particular request to evaluate all rules. This can have performance benefits. An avoid-cache-rule can be set at the top or above the rule you don’t want to apply.

There may also be cases where you may define a rule with request URI, say "*view*.jsp". This will cache responses for all requests with *.jsp extension and "view" in the URI.

However, suppose you want to exclude viewAdmin.jsp from such caching. Define an avoid-cache-rule with request URI as "viewAdmin.jsp" and insert it above the cache "*view*.jsp" rule.

The structure of an avoid-cache-rule is similar to the caching rule. See *Pramati Server Technical Reference* for more details.

**Setting execution order for rules**

The execution order for rules is very important. Users must define the rules keeping in the mind the performance since redundant or unnecessary rules may be getting evaluated by a request. Also there may be logic based on the order of the rules. The order of rules matters only between cache-rule and avoid-cache-rules. They can be merged and constitute the same list of rules for the evaluation. The action of course will defer. A rule that is significantly more frequently used than the other ones must be kept higher in the order.

**Simple time-based invalidation of cache**

Time-based invalidation of cached content is defined inside the caching rule, where an "expiry" field value is specified (in seconds). This is the time after the response is cached, when the cache expires.

The value depends on the longevity of the content. A typical usage may be a system where data is updated every 24 hours—here the pages are cached for expiry after 24 hrs from when the data is updated.

**Using application logic to invalidate cache**

This is applicable for an application design that has two components: view component and an update component. The view component provides the user interface to end users and consists of view pages.
The update component is used exclusively by the application server to update the data rendered by the view pages.

In this case, caching engine can use the update component as a trigger to update the cached view pages. A cached view page remains valid until the corresponding update component is called.

For example, the cached response to the URI `viewOrder.jsp?orderId=003` is invalidated when `updateOrder.jsp?order_no=003` is called. The application invalidation rule accepts `updateOrder.jsp` and checks if a cache-rule refers to it. If there is and the content is cached, it is invalidated.

There are two different parts of this kind of invalidation:

- Rule definition (similar to the avoid-cache-rule definition).
- Rule Reference

See Pramati Server Technical Reference for more details.

An application-based invalidation rule can be referred to by more than one cache rule. For example, `orderLines.jsp` and `orderDetails.jsp` can depend on a common update component `updateOrder.jsp`.

**Using parameters with application-based invalidation of cache**

The application-based rules can also include parameters. If `updateOrder.jsp` is called for a parameter `orderId=002`, then `orderDetails.jsp` should be invalidated only for `order_no=002` and `orderLines.jsp` should be invalidated only for `order=002`.

Such mapping is defined inside the cache-rule. The rule also takes input mappings which specifies view component parameter (order_no for `orderDetails.jsp`) of the cached request that should match update component parameter (orderId for `updateOrder.jsp`). The first one is referred as source and second one referred as target.

The source can be "source-input-param", "source-cookie" or "source-request-header" and target can be "target-input-param", "target-cookie", "target-request-header" or an absolute value say "002" in our example. The source-target pairs are given in Table 1.

A typical application-based rule reference looks like this:

```xml
<application-based-invalidation-rule-ref name="CUSTOMER_INVALIDATION">
    <!-- multiple app based-invalidation-rule-ref possible for a cache rule-->
    <input-mapping source-input-param="customer" target-input-param="cust"/>
    <input-mapping source-input-param="cust_cat" value="senior_citizen"/>
    <input-mapping source-cookie="cust" target-cookie="customer"/>
    <input-mapping source-request-header="Accept" target-request-header="Accept"/>
</application-based-invalidation-rule-ref>
```
Limiting cache size and triggering a flush

Pramati Server maintains cache for every virtual host. You can set a maximum cache size (in megabytes) for a virtual host in the XML through this tag entry:

```
<max-cache-size>10</max-cache-size>
```

When the maximum cache size is reached, the cache content must be invalidated based on the victimization mechanism. Pramati Server currently provides the Least Recently Used victimization mechanism, in which keys least recently accessed are removed. When cache size is full, 20 per cent of the space is invalidated and vacated. For example, if cache size is 10 MB, LRU keys occupying 2 MB space are invalidated when the maximum cache size is reached.

Writing a custom invalidation rule


Summary

Pramati Dynamic Content Cache Server is designed to accelerate the performance of dynamic web-based applications. Dynamic Content Cache Server can become the primary delivery vehicle for most content, leaving the web server to manage updates and publish new content. Apart from being a part of the e-business infrastructure, the Server provides flexibility to developers to configure caching rules that can truly improve their web application performance.

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>source-input-param</td>
<td>target-input-param or an absolute value</td>
</tr>
<tr>
<td>source-cookie</td>
<td>target-cookie or an absolute value</td>
</tr>
<tr>
<td>source-request-header</td>
<td>target-request-header or an absolute value</td>
</tr>
</tbody>
</table>