# Full Page Caching Interchange 5

Mark Johnson

End Point Corporation

# Application Stack

- Browser

# Application Stack

- Browser
- CDN/Reverse Proxy

# Application Stack

- Browser
- CDN/Reverse Proxy
- Web server

# Application Stack

- Browser
- CDN/Reverse Proxy
- Web server
- Interchange + Session database

# Application Stack

- Browser
- CDN/Reverse Proxy
- Web server
- Interchange + Session database
  - Database

# Application Stack

- Browser
- CDN/Reverse Proxy
- Web server
- Interchange + Session database
  - Database
  - Other applications

# Application Stack

- Browser
- CDN/Reverse Proxy
- Web server
- Interchange + Session database
  - Database
  - Other applications
  - Web services

# Caching with [timed-build]

# Caching with [timed-build]

- Buffers below Interchange

# Caching with [timed-build]

- Buffers below Interchange
  - Can't exclude session access

# Static page builds

# Static page builds

- Buffers below web server

# Static page builds

- Buffers below web server
- Still requires response from web server

# Static page builds

- Buffers below web server
- Still requires response from web server
- All resources come from hardware we control and deploy.

# Target: Reverse Proxy

# Target: Reverse Proxy

- Highly efficient

# Target: Reverse Proxy

- Highly efficient
- Highly scalable

# Target: Reverse Proxy

- Highly efficient
- Highly scalable
- Low maintenance

# Target: Reverse Proxy (cont.)

- Easy to expand to 3rd-party CDN, which gives us massive scalability without hardware maintenance and investment.

# Target: Reverse Proxy (cont.)

- Easy to expand to 3rd-party CDN, which gives us massive scalability without hardware maintenance and investment.
- With cache headers, any resources we are able to manage at a reverse proxy will allow us to (under normal use) push caching all the way up to the client itself. Repeated hits by a user at this level never even hit the network.

# Obstacles to Full-Page Caching

# Obstacles to Full-Page Caching

- High state dependence due to standard practices of liberal session coupling.

# Obstacles to Full-Page Caching

- High state dependence due to standard practices of liberal session coupling.
- Common resources, such as search objects for canned searches, being tied to individual users.

# Obstacles to Full-Page Caching

- High state dependence due to standard practices of liberal session coupling.
- Common resources, such as search objects for canned searches, being tied to individual users.
- Non-RESTful URLs, particularly the dependence on process.

# Core Interchange Support for Full Page Caching

# Core Interchange Support for Full Page Caching

- Will be available soon.

# Core Interchange Support for Full Page Caching (cont.)

- SuppressCachedCookies catalog config.

# Core Interchange Support for Full Page Caching (cont.)

- SuppressCachedCookies catalog config.
  - Tells Interchange, if this is a cacheable resource, clear the cookie jar.

# Core Interchange Support for Full Page Caching (cont.)

- SuppressCachedCookies catalog config.
  - Tells Interchange, if this is a cacheable resource, clear the cookie jar.
  - Default is "no" for backward compatibility.

# Core Interchange Support for Full Page Caching (cont.)

- [if-not-volatile]

# Core Interchange Support for Full Page Caching (cont.)

- [if-not-volatile]
  - tag will interpolate its body if the request has been identified as potentially cacheable.

# Core Interchange Support for Full Page Caching (cont.)

- [if-not-volatile]
  - tag will interpolate its body if the request has been identified as potentially cacheable.
  - Its truth does not mean the resource /will/ be cached. In fact, its initial use was to allow simple in-page calls for the cache_control pragma to be set, making it into a cacheable resource.

# Core Interchange Support for Full Page Caching (cont.)

- $::Instance->{Volatile}

# Core Interchange Support for Full Page Caching (cont.)

- $::Instance->{Volatile}
  - indicates to critical core code what the request's cache potential is. Three values:

# Core Interchange Support for Full Page Caching (cont.)

- $::Instance->{Volatile}
  - indicates to critical core code what the request's cache potential is. Three values:
    - undef => unknown, could be cached but hasn't been explicitly identified.

# Core Interchange Support for Full Page Caching (cont.)

- $::Instance->{Volatile}
  - indicates to critical core code what the request's cache potential is. Three values:
    - undef => unknown, could be cached but hasn't been explicitly identified.
    - true => cannot be cached. Indicates the requested resource is user-dependent and may produce different results for the same URL for different users.

# Core Interchange Support for Full Page Caching (cont.)

- $::Instance->{Volatile}
  - indicates to critical core code what the request's cache potential is. Three values:
    - undef => unknown, could be cached but hasn't been explicitly identified.
    - true => cannot be cached. Indicates the requested resource is user-dependent and may produce different results for the same URL for different users.
    - false otherwise => explicitly treat as a "can be cached" resource. This setting can be used to reverse override certain other cache overrides.

# Core Interchange Support for Full Page Caching (cont.)

- $::Instance->{Volatile} (cont.)
  - E.g., calls to process actionmap by default are set to Volatile 1, but if certain circumstances require caching a particular invocation of process, setting $::Instance->{Volatile} = 0 will stop this default behavior, leaving cacheability as an option.

# Core Interchange Support for Full Page Caching (cont.)

- OutputCookieHook catalog config.

# Core Interchange Support for Full Page Caching (cont.)

- OutputCookieHook catalog config.
  - Points to a catalog or global sub that can be used to build and maintain cookies that we expect to need to support greater client-side persistence (discussed later).

# Core Interchange Support for Full Page Caching (cont.)

- OutputCookieHook catalog config.
  - Points to a catalog or global sub that can be used to build and maintain cookies that we expect to need to support greater client-side persistence (discussed later).
  - Runs just prior to Interchange code that assembles the cookie jar into headers.

# Catalog changes

# Catalog changes

- Ideally take a REST approach to cacheable resources

# Catalog changes

- Ideally take a REST approach to cacheable resources
  - Unique URLs make for the cleanest and most transparent implementation of cachable resources.

# Catalog changes

- Ideally take a REST approach to cacheable resources
  - Unique URLs make for the cleanest and most transparent implementation of cachable resources.
  - Actionmap development is an excellent mechanism for implementing cacheable resources.

# Catalog changes (cont.)

- Decouple session, or any other state, dependencies.

# Catalog changes (cont.)

- Decouple session, or any other state, dependencies.
  - If they can't be decoupled, it shouldn't be considered a candidate for a cacheable resource.

# Catalog changes (cont.)

- Decouple session, or any other state, dependencies.
  - If they can't be decoupled, it shouldn't be considered a candidate for a cacheable resource.
  - Many session and state dependencies /can/ be decoupled with enough effort and creativity.

# Catalog changes (cont.)

- Decouple session, or any other state, dependencies.
  - If they can't be decoupled, it shouldn't be considered a candidate for a cacheable resource.
  - Many session and state dependencies /can/ be decoupled with enough effort and creativity.
    - Pick your battles! Start with those resources that receive the most traffic and/or cause the most server load.

# Catalog changes (cont.)

- Decouple session, or any other state, dependencies. (cont.)
  - Common ITL requiring decoupling
    - [value]
    - [data session __]
    - [seti?]

# Catalog changes (cont.)

- Decouple session, or any other state, dependencies. (cont.)
  - Common ITL requiring decoupling
    - [value]
    - [data session __]
    - [seti?]
      - But not [tmpn?] unless the body contains such dependencies

# Catalog changes (cont.)

- Decouple session, or any other state, dependencies. (cont.)
  - Common ITL requiring decoupling
    - [value]
    - [data session __]
    - [seti?]
      - But not [tmpn?] unless the body contains such dependencies
    - Any associated [if] usage. (E.g., [if value ...])

# Catalog changes (cont.)

- Decouple session, or any other state, dependencies. (cont.)
  - Common ITL requiring decoupling
    - [value]
    - [data session __]
    - [seti?]
      - But not [tmpn?] unless the body contains such dependencies
    - Any associated [if] usage. (E.g., [if value ...])
      - Unless [if scratch ...] is based on a [tmp]

# Catalog changes (cont.)

- Decouple session, or any other state, dependencies. (cont.)
  - Some state information *may* be acceptable within a cached resource depending on business requirements.

# Catalog changes (cont.)

- Decouple session, or any other state, dependencies. (cont.)
  - Some state information *may* be acceptable within a cached resource depending on business requirements.
    - Randomized related data, such as cross-merchandising, may be cacheable depending on duration of cache or other factors that influence business tolerance of the same list showing up for all clients over the cache period.

# Catalog changes (cont.)

- Cacheable redirects must be considered all or nothing. The URL invocation that results in a redirect must do so for all parties.

# Catalog changes (cont.)

- Cacheable redirects must be considered all or nothing. The URL invocation that results in a redirect must do so for all parties.
  - Ensure any bounce conditions are encapsulated in the URL, as either part of the path or query string.

# Catalog changes (cont.)

- Profile and "click" code.

# Catalog changes (cont.)

- Profile and "click" code.
  - Common practice to set scratch either directly or by means of [button] will break caching.

# Catalog changes (cont.)

- Profile and "click" code.
  - Common practice to set scratch either directly or by means of [button] will break caching.
  - All profile and click code must be moved into profiles compiled at startup.

# Catalog changes (cont.)

- Profile and "click" code.
  - Common practice to set scratch either directly or by means of [button] will break caching.
  - All profile and click code must be moved into profiles compiled at startup.
    - This is also good practice to avoid any conflicts or missing/changed profile/click code due to use of browser Back buttons.

# Catalog changes (cont.)

- Set SuppressCachedCookies to "yes". This is necessary so Interchange will know not to bust the cache by including cookie headers.

# Catalog changes (cont.)

- Include [tag pragma cache_control]max-age=XXX[/tag] on any ITL snippet run on a resource for which you want caching.

# Catalog changes (cont.)

- Include [tag pragma cache_control]max-age=XXX[/tag] on any ITL snippet run on a resource for which you want caching.
  - Use of [if-not-volatile] as a wrapper may be of assistance on any included code that could be part of both cacheable and non-cacheable resources.

# Catalog changes (cont.)

- Include [tag pragma cache_control]max-age=XXX[/tag] on any ITL snippet run on a resource for which you want caching.
  - Use of [if-not-volatile] as a wrapper may be of assistance on any included code that could be part of both cacheable and non-cacheable resources.
  - One recommendation is to create a new catalog variable that contains all the above logic:
    - Variable CACHEABLE_RESPONSE [if-not-volatile][tag pragma ...][/if-not-volatile]

# Catalog changes (cont.)

- Canned searches, which drive most category lists, convert to Permanent More, allowing common search caches to be shared across all users, as well as creating common more-list URLs that can be cached.

# Session implications

# Session implications

- Initial requests from users against a cacheable resource will not create a session.

# Session implications

- Initial requests from users against a cacheable resource will not create a session.
  - Creating session requires returning session cookie, so page would not be cached as a result.

# Session implications

- Initial requests from users against a cacheable resource will not create a session.
  - Creating session requires returning session cookie, so page would not be cached as a result.
  - Significant implications for DoS attacks, which would by-pass the buffer of the cache while churning out new sessions with each request.

# Session implications (cont.)

- Session writing, including the creation of a session, is suppressed on any response with a cache-control header set.

# Session implications (cont.)

- Session writing, including the creation of a session, is suppressed on any response with a cache-control header set.
  - If the resource is cacheable, that implies no action as a result of the request can impact, or be impacted by, the session.

# Overrides on a cacheable response

# Overrides on a cacheable response

- Any POST

# Overrides on a cacheable response

- Any POST
- Any execution of the "process" and "order" actionmaps.

# Overrides on a cacheable response

- Any POST
- Any execution of the "process" and "order" actionmaps.
  - Override is implemented by setting $::Instance->{Volatile} to perly true.

# Caching duration

# Caching duration

- Standard recommendations range from 1 month - 1 year.

# Caching duration (cont.)

- What is the true value of the cache? Law of Diminishing Returns.

# Caching duration (cont.)

- What is the true value of the cache? Law of Diminishing Returns.
  - Even absurdly small cache lives of 5s can shield a popular resource during a traffic spike from a huge number of requests.

# Caching duration (cont.)

- What is the true value of the cache? Law of Diminishing Returns.
  - Even absurdly small cache lives of 5s can shield a popular resource during a traffic spike from a huge number of requests.
  - As cache life expands on a dynamic web site, the impact of stale content increases.

# Caching duration (cont.)

- What is the true value of the cache? Law of Diminishing Returns.
  - Even absurdly small cache lives of 5s can shield a popular resource during a traffic spike from a huge number of requests.
  - As cache life expands on a dynamic web site, the impact of stale content increases.
  - Cache-clearing strategies for stale content cannot reach the browser. If we have a long cache life on a popular resource, repeat users are very likely to be stuck with a stale version for some time.

# Caching duration (cont.)

- Take a balanced approach between tolerance for stale data and protecting the application. Too often, the latter is overly emphasized.

# Caching duration (cont.)

- Take a balanced approach between tolerance for stale data and protecting the application. Too often, the latter is overly emphasized.
  - For most scenarios, a cache window of 1-4 hours provides exceedingly good performance and protection without imposing too much burden on dynamic data.

# Caching levels

# Caching levels

- Different levels of decoupling responses from state can be employed depending on need and difficulty.

# Caching levels (cont.)

- Example demarcation points for where to cache and not cache.

# Caching levels (cont.)

- Example demarcation points for where to cache and not to cache.
  - Login state
    - Caching imposed on resources without a login.

# Caching levels (cont.)

- Example demarcation points for where to cache and not to cache.
  - Login state
    - Caching imposed on resources without a login.
    - For a logged in user, caching is by-passed, thus allowing for personalized session data to be used in the resource.

# Caching levels (cont.)

- Example demarcation points for where to cache and not to cache.
  - Login state
    - Caching imposed on resources without a login.
    - For a logged in user, caching is by-passed, thus allowing for personalized session data to be used in the resource.
    - Resource access must be reflected in the URL, by path or query-string alteration.

# Caching levels (cont.)

- Example demarcation points for where to cache and not to cache. (cont.)
  - Cart
    - Cache or not if there's a cart. Cart may impact display ("small cart" in template) or may impact content (customized product information based on cart content).

# Caching levels (cont.)

- Example demarcation points for where to cache and not to cache. (cont.)
  - Cart
    - Cache or not if there's a cart. Cart may impact display ("small cart" in template) or may impact content (customized product information based on cart content).
    - May be orthogonal or in addition to login implications.

# Caching levels (cont.)

- Persistence can be extended to the client so that the above features need not limit caching opportunities.

# Caching levels (cont.)

- Persistence can be extended to the client so that the above features need not limit caching opportunities.
  - The use of cookies and modern javascript frameworks, such as jQuery, can ease both implementation and browser compatibility to leverage persistence across cached resources.

# Caching levels (cont.)

- Client persistence can be divided into two levels of support. “read only” and “read write” persistence.

# Caching levels (cont.)

- "Read only" persistence. The client-specific data we normally delegate to the session is represented in a cookie (or cookies) from which we can display customized content to the user.

# Caching levels (cont.)

- "Read only" persistence. The client-specific data we normally delegate to the session is represented in a cookie (or cookies) from which we can display customized content to the user.
  - E.g., "Hi, [value fname]!" is replaced with a reference to a cookie holding some, or all, of Values space from which we can write "Hi, !" and have an appropriate call to fill in the name from cookie data, e.g., $('#fname').replaceWith(valuesData.fname).

# Caching levels (cont.)

- "Read only" persistence. The client-specific data we normally delegate to the session is represented in a cookie (or cookies) from which we can display customized content to the user.
  - E.g., "Hi, [value fname]!" is replaced with a reference to a cookie holding some, or all, of Values space from which we can write "Hi, !" and have an appropriate call to fill in the name from cookie data, e.g., $('#fname').replaceWith(valuesData.fname).
  - With minimal complication, the cookie is controlled only from server responses and allows the client to read portions of the session locally.

# Caching levels (cont.)

- "Read write" persistence. It may be highly desirable to have caching on a resource that also requires session modification.

# Caching levels (cont.)

- “Read write" persistence. It may be highly desirable to have caching on a resource that also requires session modification.
  - The javascript data structure derived from the cookie is written to as data change.

# Caching levels (cont.)

- "Read write" persistence. It may be highly desirable to have caching on a resource that also requires session modification.
  - The javascript data structure derived from the cookie is written to as data change.
  - A supplemental cookie is set with the state changes of each variable that has changed within the client.

# Caching levels (cont.)

- "Read write" persistence. It may be highly desirable to have caching on a resource that also requires session modification.
  - The javascript data structure derived from the cookie is written to as data change.
  - A supplemental cookie is set with the state changes of each variable that has changed within the client.
  - Within an autoload or other "catch-all" code, the presence of the supplemental cookie is detected and Values space is updated in the session with the changes.

# Caching levels (cont.)

- In both cases, Values is just an example. The same could be extended to any other feature of the session whose state we wish to persist and capture from cached client resources.

# Questions?

Thank you!