APACHE SLING & FRIENDS TECH MEETUP

BERLIN, 23-25 SEPTEMBER 2013

Scaling CQ5

Michael Marth | Adobe Engineering

CQ5?

- Web Content Management system built on Sling/JCR stack
- CQ5 scaling concepts applicable to other Sling applications
- CQ5 specific concepts:
  - Author instances/publish instances
  - Replication: technology to transport serialized JCR content between instances
  - Dispatcher: web server plugin for caching

adaptTo()

Performance vs. Scalability

- Performance: “it takes X secs to do Y”
- Scalability: “it takes X secs to do Y simultaneously Z times”
  - But performance can help with scalability
- This talk
  - is about horizontal scalability (vertical scaling is trivial)
  - is about pre-Oak scalability patterns

Patterns

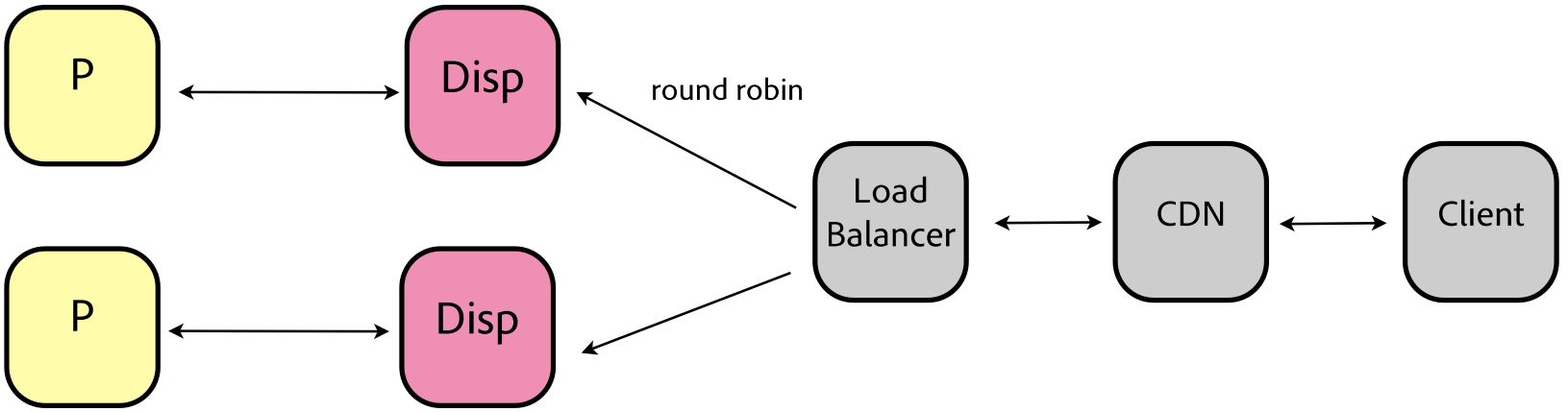
1. High Volume and High Performance Delivery
2. High Frequency Input Feed
3. Many Editors
4. High Processing Input Feed
5. High Volume Input Feed
6. Geo-distributed Editors
7. Many DAM assets
8. Geo-distributed disaster recovery

High Volume and High Performance Delivery - Description

- Use Case:
  - High traffic site (100m impressions/d)
- Examples: adobe.com
- Limiting factor
  - CPU on publish

High Volume and High Performance Delivery - Solution Pattern

- Leverage dispatcher caching as much as possible
  - in latest dispatcher: single-page dispatcher flush and scripted flushing, use to cache/flush content in dispatcher
  - SSI and/or client-side for personalized content
  - Selectors for query caching
- CDN with short TTL



High Volume and High Performance Delivery

- Related to rendering performance, see also
  - CQ performance patterns (use CQ timing component, prefer tree walking over JCR queries, use ClientLibraryManager to concat and minify JS, etc, see [1])
  - Generic performance patterns (reduce requests with e.g. css sprites, gzip responses, put JS calls at bottom of HTML, etc, see [2])
- Anti-Pattern
  - Adding publishers before leveraging caching

[1] http://dev.day.com/docs/en/cq/current/deploying/performance.html

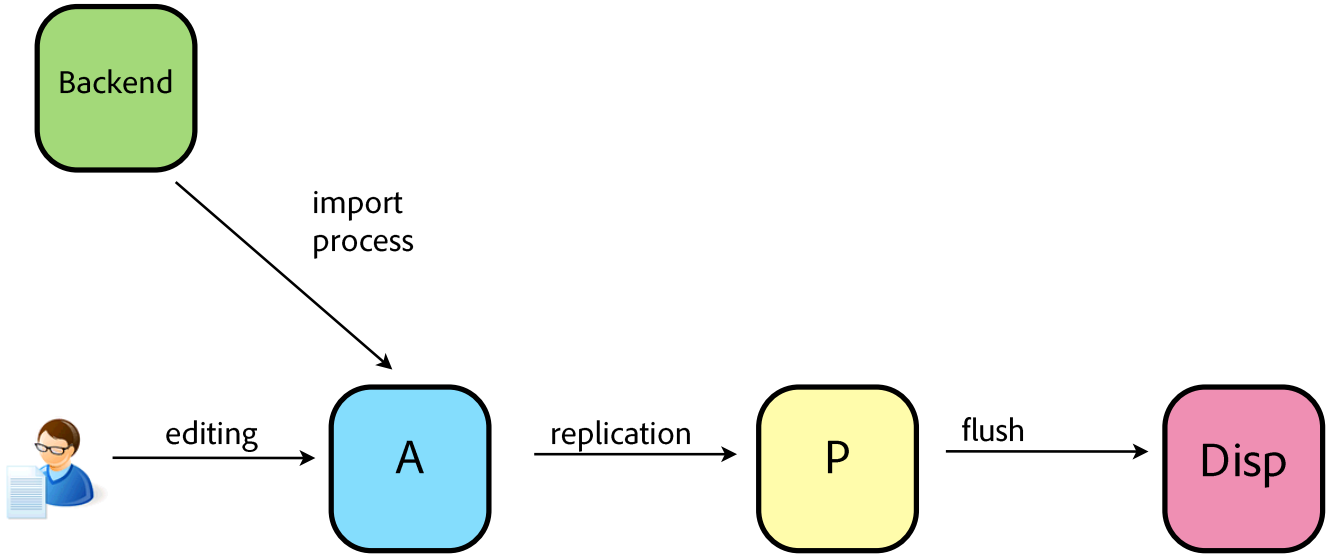
[2] http://shop.oreilly.com/product/9780596529307.do

High Frequency Input Feed - Description

- Use Case: news feed import (moderate amounts, but constant updates)
- Limiting factor
  - Dispatcher cache invalidation
    - Therefore actual limiting factor is CPU on publish

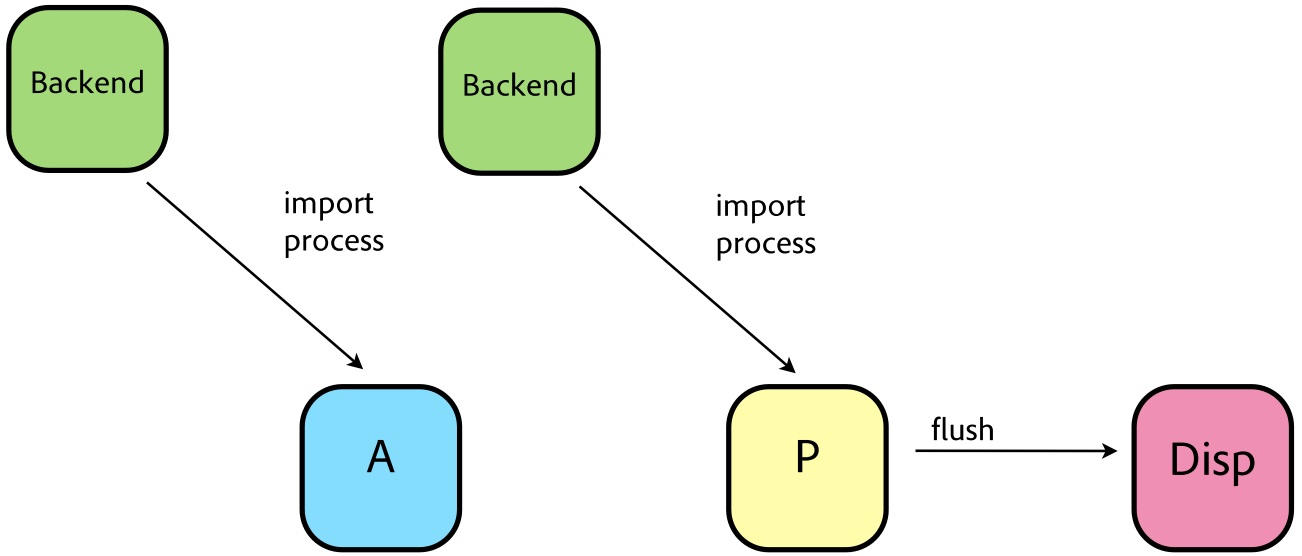
High Frequency Input Feed - Solution Pattern 1

- Set up content structure so that other pages do not get invalidated on dispatcher cache
  - if possible: highly volatile content e.g. in /etc
  - with latest dispatcher: single-page flush possible
- Separate replication queue (so that main queue is not blocked)



High Frequency Input Feed - Solution Pattern 2

- Set up content structure so that other pages do not get invalidated on dispatcher cache
  - as previous pattern
- Import directly into Publish (no replication necessary)



High Frequency Input Feed

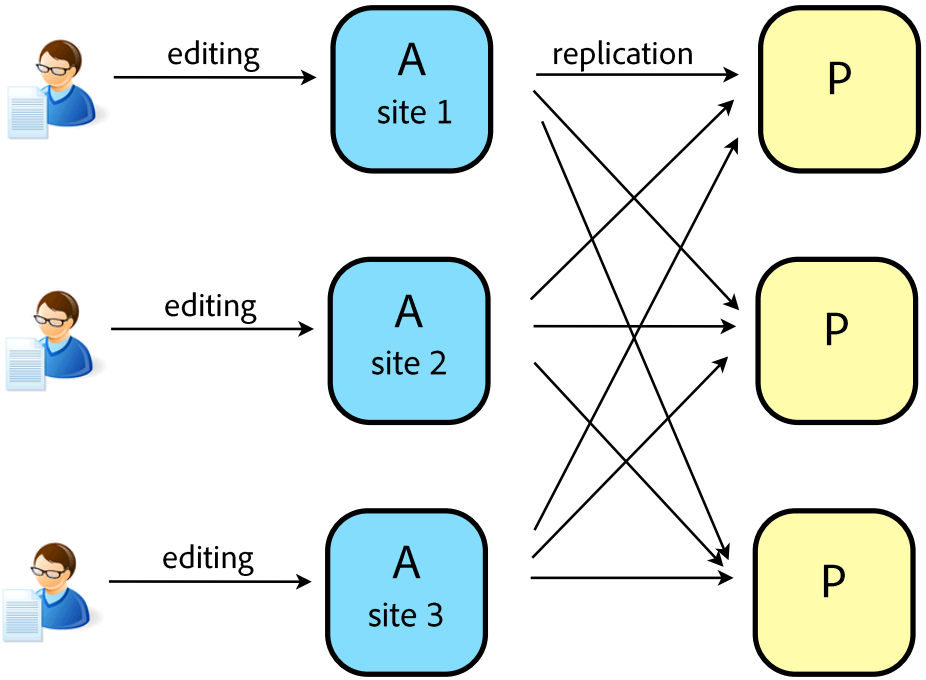
- Questions to ask
  - Human filtering/processing needed? Then imports should be on author and replicated.
    - If no: is the use case OK with different states on publish?
      - if yes: no replication needed, then pattern 2 is preferable

Many Editors - Description

- Use Case:
  - News or media portal
  - >50 editors editing content concurrently
- Limiting factor
  - Depends on what do the editors actually do:
    - Heavyweight editing, e.g. MSM rollouts, starting WFs: repository- or CPU-bound
    - Lightweight editing: CPU bound

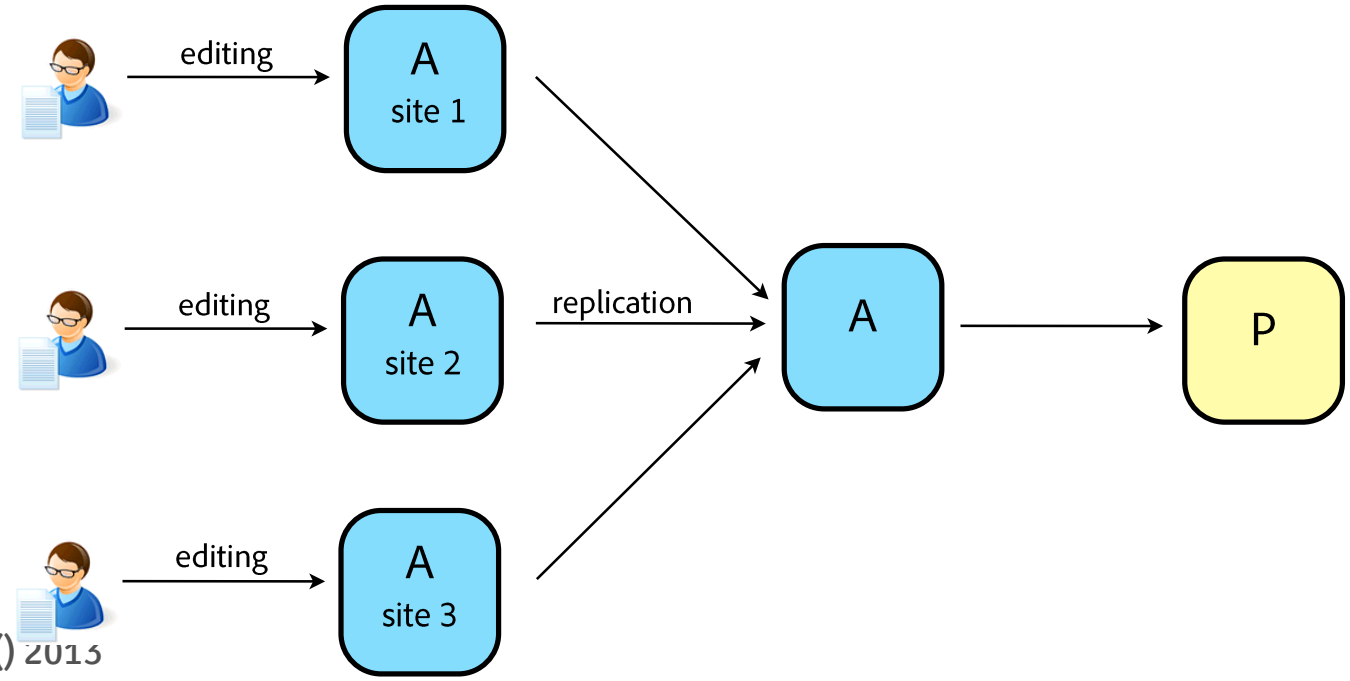
Many Editors - Solution Pattern 1

- Sharding: split up different web sites / parts of web sites onto separate author instances
- Publish instances are shared



Many Editors - Solution Pattern 2

- Sharding: split up different web sites into separate author instances, but replicate into one main author, e.g. for shared workflow processes
  - Practical if the shards do not need to share content.
  - Cross-replication can be done, but will be hard to keep consistent
- Publish instances are shared



Many Editors

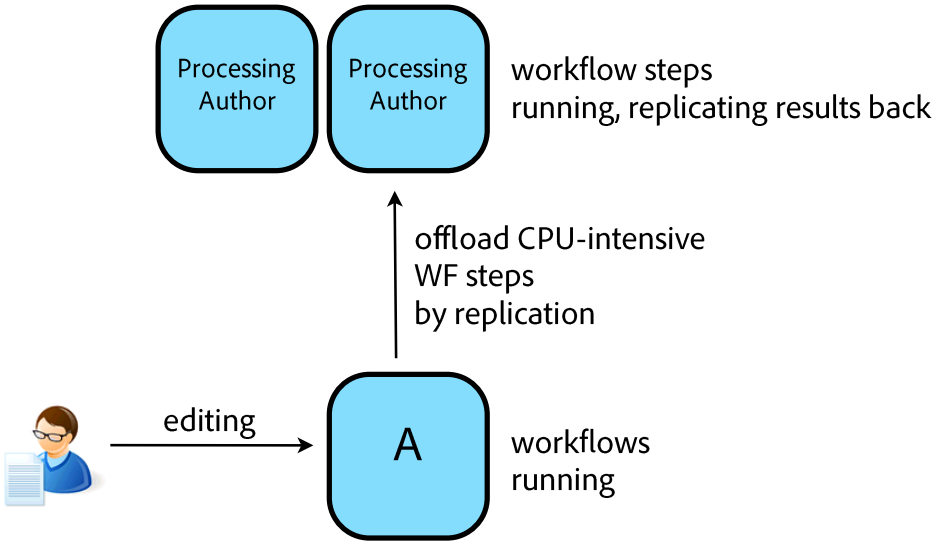
- Notes
  - Author dispatcher helps to reduce CPU load on author instances
  - Author cluster instead of sharding will mitigate the problem if CPU-bound

High Processing Input Feed - Description

- Use Case:
  - DAM import of images
    - 1000 images at once
    - happens regularly
    - other editors are editing content at the same time
- Limiting factor
  - CPU, memory on author

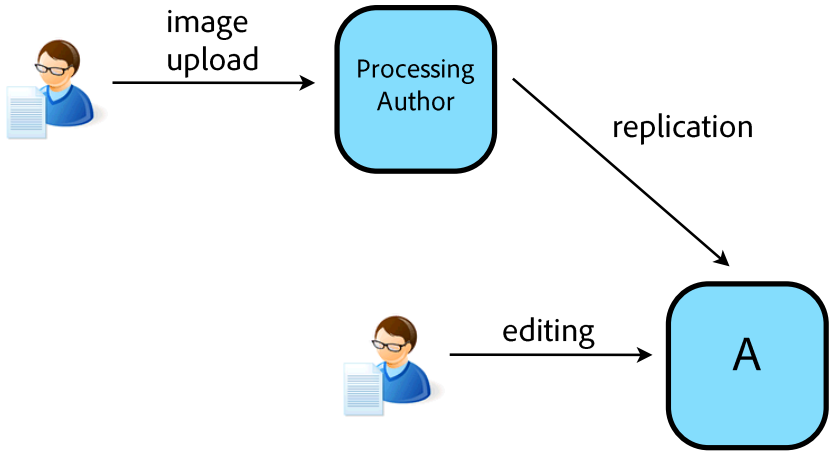
High Processing Input Feed - Solution Pattern 1

- Separate processing instances from human editing instances
  - Offload 1 Workflow step, e.g. thumbnail generation from PSDs
  - There can be more than 1 processing instance
  - Replicate back and forth in packages if possible
  - CQ5.6.1: share DS between instances and replicate without binary, offloading framework



High Processing Input Feed - Solution Pattern 2

- Separate pre-processing instances for uploading
  - There can be more than 1 pre-processing instance
  - CQ5.6.1: share DS between instances and replicate without binary



High Processing Input Feed

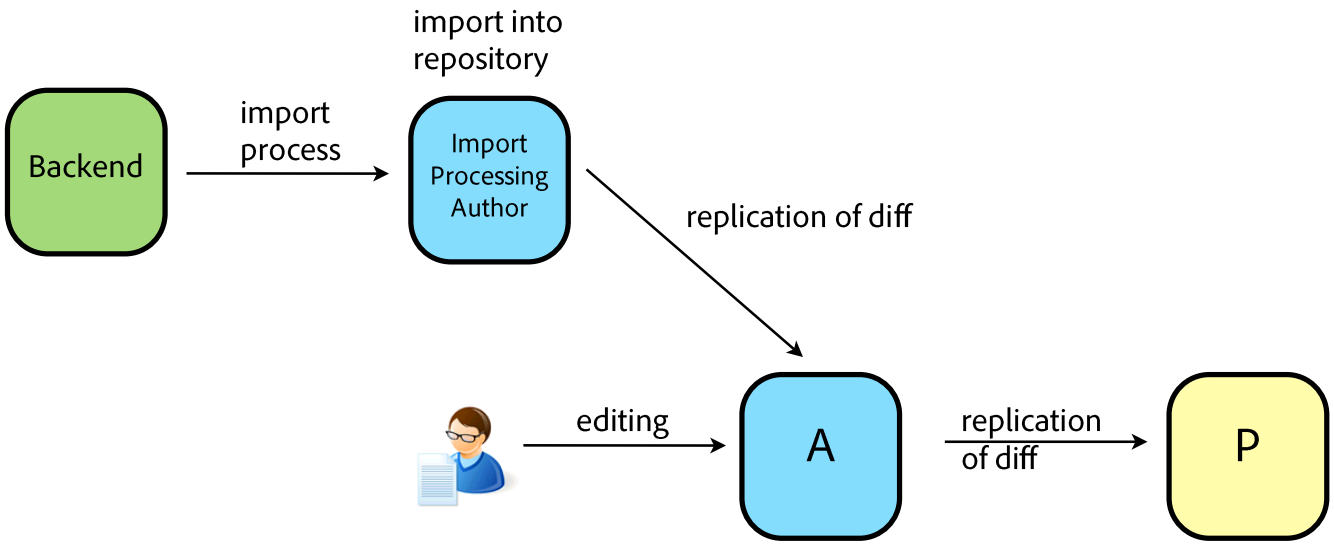
- Notes
  - Author cluster can help mitigate the problem, but editors must edit content on slave
  - Throttling WFs or execution during night can help mitigate the problem
  - If the import is limited by CPU needed image conversion consider using ImageMagick rather than Java

High Volume Input Feed - Description

- Use Case:
  - Product data import
    - 1 million products, 10000 modifications/day
- Limiting factor
  - Writing to the repository
    - reads are also blocked
  - Potentially (to a lesser degree) in case repository scans are needed to create diffs:
    - CPU for calculating diffs
    - Repository read caches get flushed

High Volume Input Feed - Solution Pattern

- Separate import instance to process imports, partition if possible
  - only useful if import requires significant CPU (e.g. no diff delivered)
- Replicate to author
  - Replicate as package
  - CQ5.6.1: share DS between instances and replicate without binary
- Replication to publish as package if possible



High Volume Input Feed

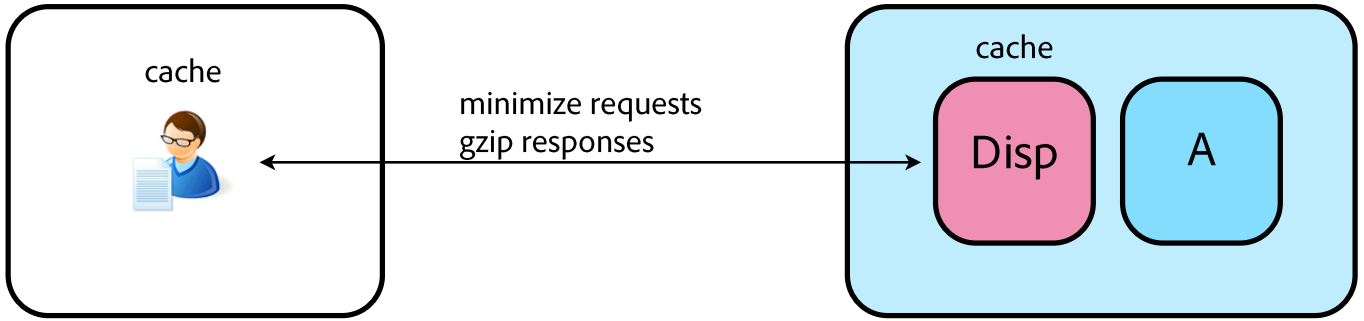
- Questions to ask
  - Can the import be throttled? Most problems get much less severe.
  - Do all changes get on publish?
- Notes
  - Use batch saves (1000 nodes) on import (reduces overhead in indexing, etc and speeds up the import overall)
  - Import as nt:unstructured rather cq:Page if possible
    - If not: switch off heavy listeners (e.g. ContentSync) or use the JcrObservationThrottle
- Anti-Pattern
  - Usage of network disc (usually have high latency)
  - Replicating to publish through same replication queue as editorial content

Geo-distributed Editors

- Use Case:
  - Editors located in different geos (US, EMEA, APAC)
- Limiting factor
  - Bandwidth between editor location and author server location

Geo-distributed Editors - Solution Pattern

- Use Dispatcher in front of Author
- Guiding principle: limit traffic between Dispatcher and editor location.
  - gzip traffic
  - Use Client Library Manager to minimize traffic
    - minify, concat and gzip all client libraries
  - Cache all responses that are not under /content in
    - Editor's browser cache
    - Potentially also dispatcher cache



Geo-distributed Editors

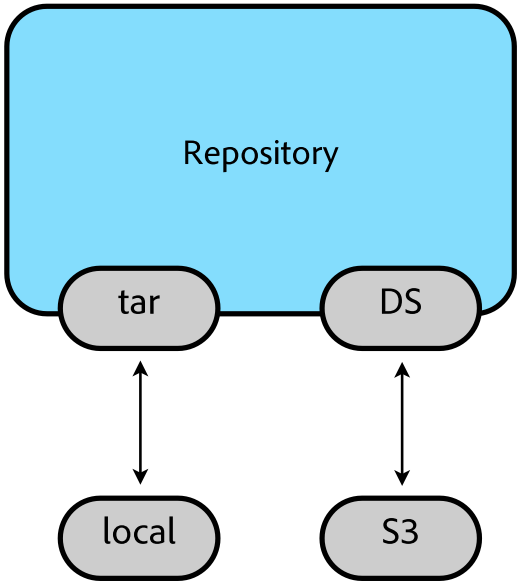
- Notes
  - In extreme cases consider writing templates that treat author renditions differently from publish renditions (especially reducing the number of necessary requests, e.g. by dropping requests to tracking servers, external CSS, etc)
  - Or use Scaffolding for editing

Many DAM Assets

- Use Case:
  - Many assets (>5Mio) in DAM
- Limiting factor
  - Disc space

Many DAM Assets - Solution Pattern

- Split physical storage of data store and repository tar files
  - tar files need disc with very low latency
  - for data store high latency is acceptable
  - Locate data store on cheap discs remotely (NAS, S3)
- Share data store between instances
  - In 5.6.1: use binary-less replication in case of shared DS to minimize network traffic



Many DAM Assets

- Notes
  - In case of shared DS: the DS garbage collection needs to be run on an instance that keeps references to all assets in DS
  - In 5.6.1: huge performance improvements (~10x or more) for DS GC when the persistence is tar-based

Geo-distributed disaster recovery

- Use Case:
  - Data centers located in different geos
  - One DC shall act as failover for author
- Limiting factor
  - Latency between DCs (in very low latency cases CRX clustering could be used)

Geo-distributed disaster recovery - Solution Pattern

- Use file level tools like rysnc to create replicas in 2nd DC
- Hourly: sync data store
  - This is usually the most time consuming part
  - Sync can be performed anytime, due to add-only data store architecture
- Nightly:
  - Create incremental backup into filesystem on 1st DC to get consistent state of files
  - Rsync backup to 2nd DC. For that period CQ on 2nd DC must not be running.

Thanks!