Next Generation Content Loading and Routing

...

Adam Baso, Jon Robson, Joaquin Hernandez, Gabriel Wicke, Sam Smith

https://phabricator.wikimedia.org/T114542
https://phabricator.wikimedia.org/T111588
https://phabricator.wikimedia.org/T106099
https://commons.wikimedia.org/wiki/File:Paradigm.pdf

Agenda

80 minutes

1. Intro, Q2 R&D backdrop
2. Performance
3. API Driven Frontend
4. Composition
5. Discussion (20 minutes)

“Imagine a world in which every single person on the planet is given free access to the sum of all human knowledge. That's what we're doing.”

“Imagine a world in which every single person on the planet is given free access to the sum of all human knowledge. That's what we're doing.”

The current state

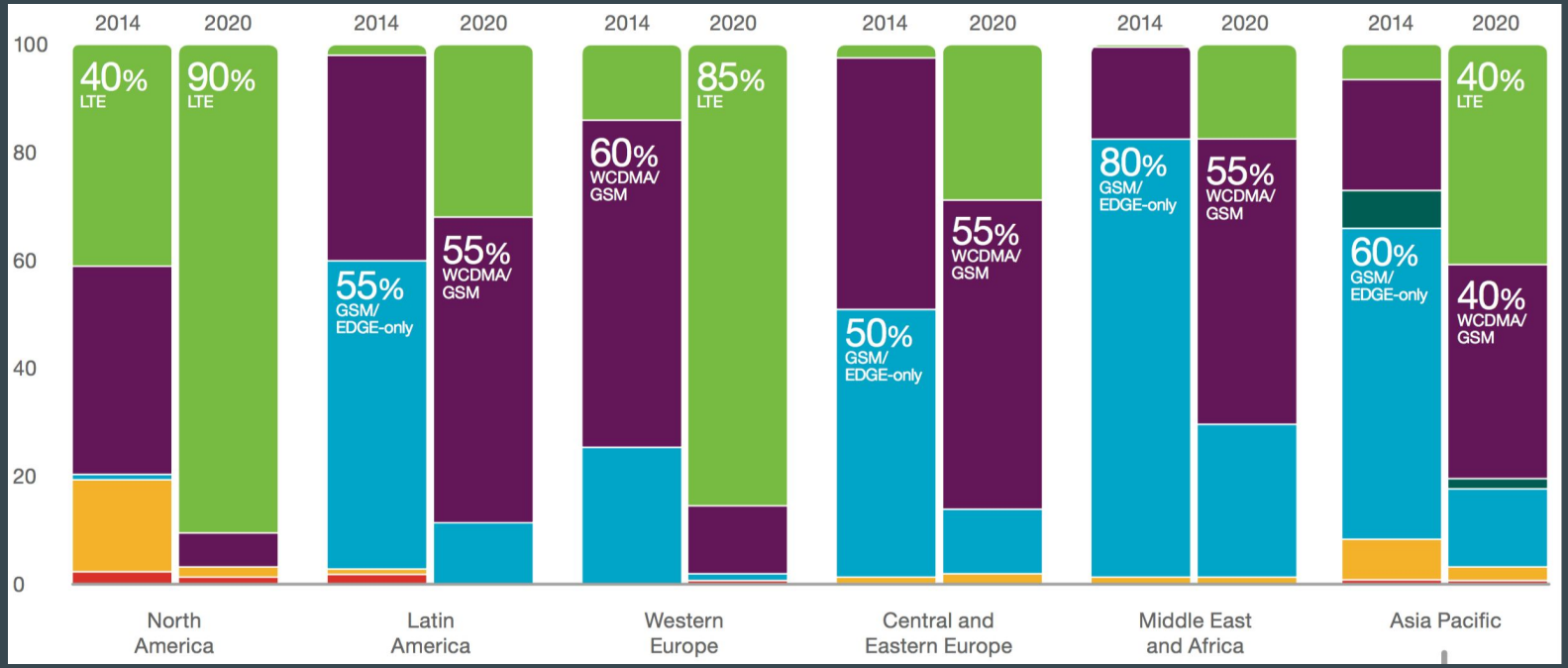
- Slow connections
- Logged in users
- Complexity
- Developer productivity

Caching layers vs. customizations

- Customize experience on user, connection, form factor
- Core issue: Implementing customizations on the server fragments caches
- Client-side, API-driven frontend can avoid most of these issues.
  - Separate shared content to high perf endpoints
  - Connection and form factor adjustments can be done at the client

Performance

Slow connections common, LTE not coming immediately



http://www.ericsson.com/res/docs/2015/ericsson-mobility-report-june-2015.pdf

62% of mobile content accessed over 2G across the world [1]

91.7% of the world lives within 2G coverage

48.7% of the world lives within 3G coverage

[1] http://goo.gl/VADE5l

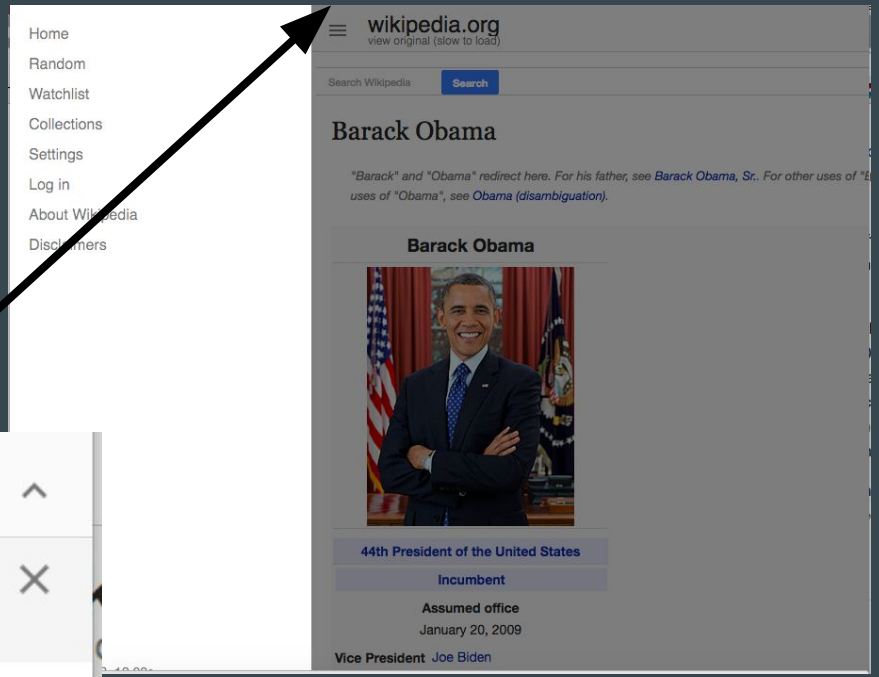
First paint

How long a user looks at a blank screen

Not the experience we built

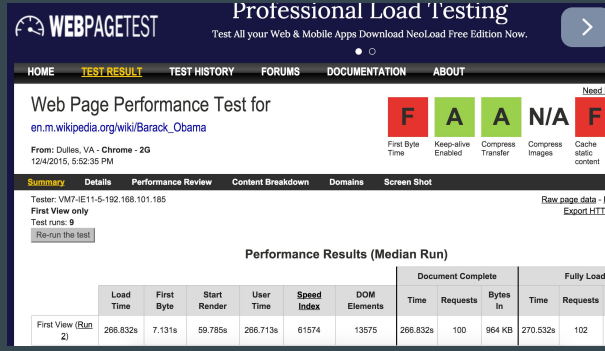
Google Web Light solves this problem [1]

We can do better.



[1] https://googleweblight.com/?lite_url=https://en.m.wikipedia.org/wiki/Barack_Obama

https://reading-web-research.wmflabs.org/Barack_Obama



Barack Obama - a sense of the possibilities...

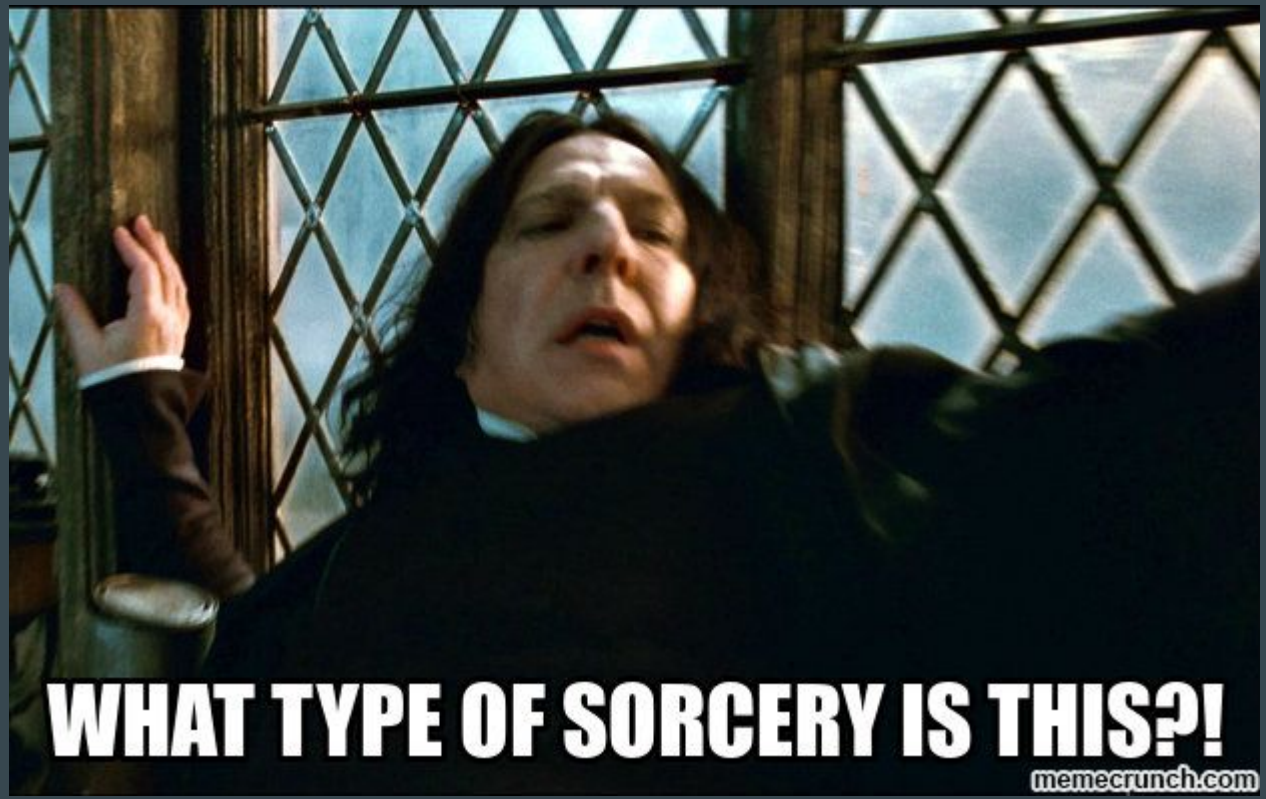
Now: First paint in 52.7s, 974 KB, fully loaded in 262.1s

After: First paint in 16.6s, 201kb, fully loaded* in 66s

http://www.webpagetest.org/result/151221_JV_13R8/
http://www.webpagetest.org/result/151221_T0_13QC/

69% decrease in first paint

*not the most fair comparison.. definition of fully loaded changed.


WHAT TYPE OF SORCERY IS THIS?!
memecrunch.com

Problem 1: Render blocking CSS

- CSS is a smaller issue but still an issue.
  - The page will not be visible until CSS is.
  - In mobile we aggressively reduce the payload
- Can improve 2nd load for multi-page readers
  - In the demo we use Service Worker to cache CSS for the first load.
  - This doesn't help the first load however.
- Inlining certain styles might help.
  - Removes http request, increases HTML size
  - Can't leverage cache for inlined styles.

Problem 2: Images

- Images slow down loading of stylesheets
- srcset evil for 2G connections
  - Browsers do not take into account your connection speed
- Various solutions to this problem that could be implemented now [1]

[1] https://phabricator.wikimedia.org/T119797

Removing images

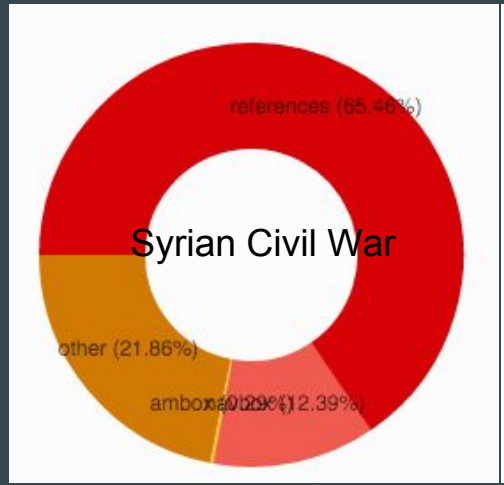
A reader of the Barack Obama page could see it 19% quicker with

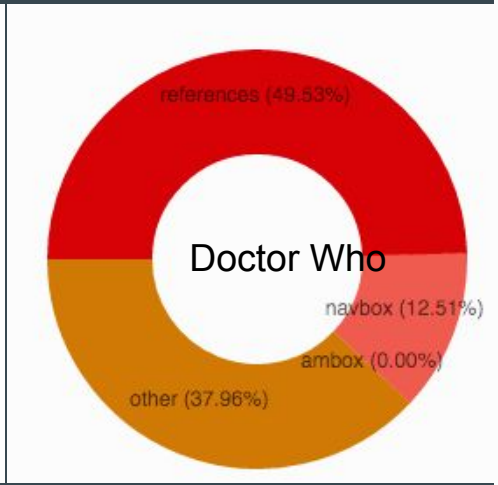
66% fewer bytes. if it didn't have any images.

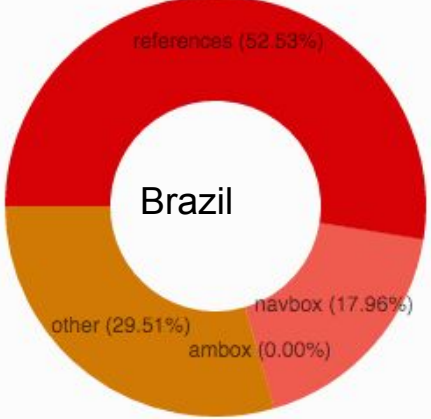
https://www.mediawiki.org/wiki/Reading/Web/Projects/A_frontend_powered_by_Parsoid/HTML_content_research#Webpagetest_report_.28browser_loading_time.29

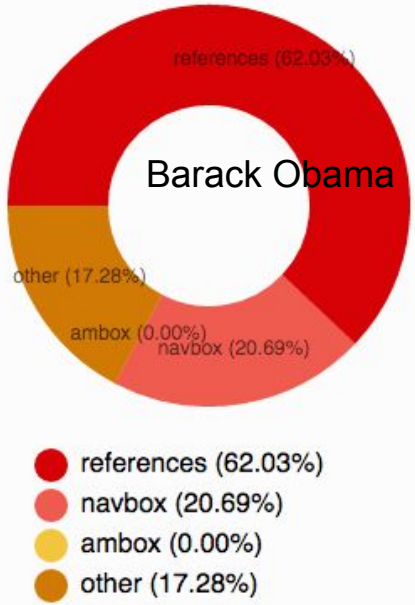
Problem 3: HTML Size

- References and navboxes contribute to the majority of the page HTML. [1]
- The better an article gets the less likely a 2G connection gets to see it.









[1] http://chimeces.com/loot-content-analysis/

References and navboxes and bytes (oh my!)

A reader of the Barack Obama page without navboxes and references sees it 55% quicker with 33% fewer bytes.[1]

At time of writing Barack Obama a reader sees the page in 61.176s and downloads 970 KB [2]

[1]https://www.mediawiki.org/wiki/Reading/Web/Projects/A_frontend_powered_by_Parsoid/HTML_content_research#Webpagetest_report_.28browser_loading_time.29

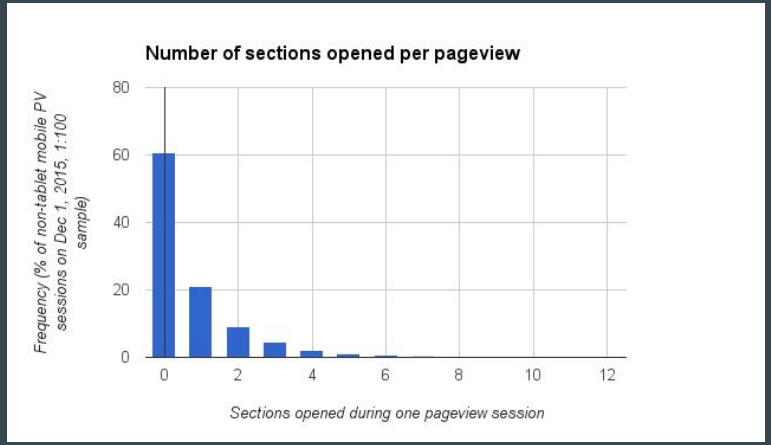
[2]http://www.webpagetest.org/result/151222_5M_1537/

People do not read all the article on mobile

We collected some data against users with collapsed sections and without collapsed sections.

40% of sessions on mobile devices opened a section

https://phabricator.wikimedia.org/T118041



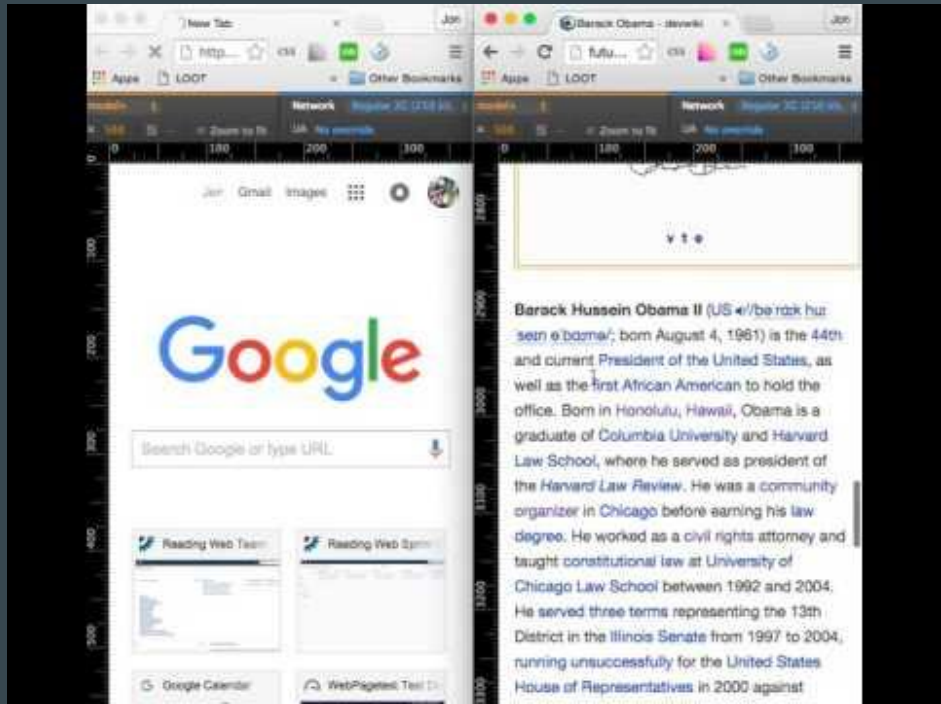
… so why give them all of it?

Summary

- HTML size is the major problem for 2G connections
  - Bandwidth consumption and first render
- Images contribute to the problem of first paint on large articles

What happens if we use RESTBASE for our content?

https://future-wikipedia.wmflabs.org/wiki/Barack_Obama VS
http://en.m.wikipedia.org/wiki/Barack_Obama



Results:

Before: First paint in 52.7s, 974 KB, fully loaded in 262.1s

After: First paint in 15.5s, 231kb, fully loaded* in 62.3s

http://www.webpagetest.org/result/151221_3C_13QK/
http://www.webpagetest.org/result/151221_T0_13QC/

*not the most fair comparison.. definition of fully loaded changed.

API Driven Frontend

Implementation Options

Fundamental questions

- No-JS support.
- Cold load & no-JS (weak JS) performance.
- SEO: Can we guarantee continued search engine findability?

Two main options

1. Single page application (case study)
2. Regular navigation with ServiceWorker composition.

Case study

Wikipedia as a web application

http://reading-web-research.wmflabs.org/wiki/Wikipedia

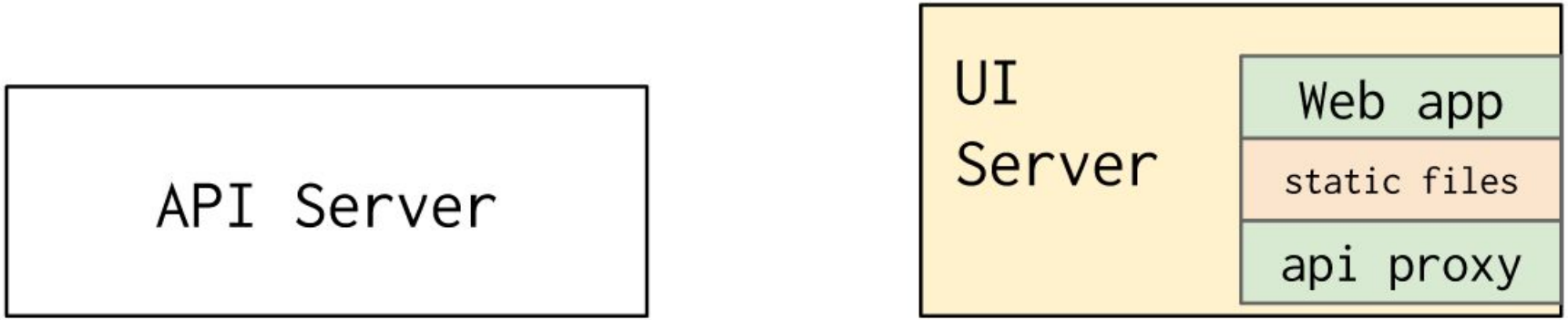
Focus areas

- Slow & unreliable networks
  - Performance
  - Better client experience
- HTML only version
- Better UI development & prototyping

Approach

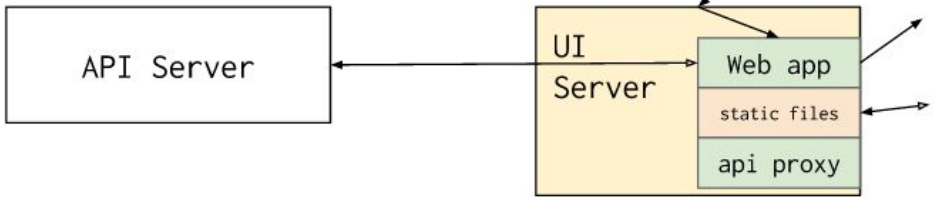
- API server: transforms Parsoid content and leverages RESTbase caching.
- UI server that renders the web application server side.
- Lazy loaded thick web client
  - Navigation: pushstate history
  - Content caching: localStorage, IndexedDB
  - Caching assets & images: Service Workers

Architecture



http://chimeces.com/loot-ui/docs/architecture.html

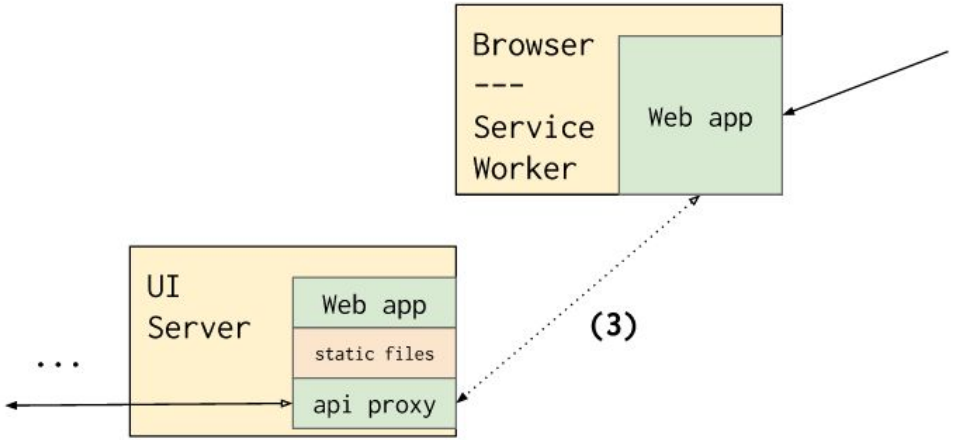
Architecture



1. Request for page comes from browser
2. Web app runs on the server. Matches routing, fetches data (3) and renders to string
4. Minimal usable HTML only response goes to the client
5. Browser renders and requests assets (CSS & JS)
6. `Web app` initializes in the client.

- All assets are cached using SW
- Client app takes over. Following requests will be through api proxy if not in cache
- App is available for offline use

Architecture



1. Request for page comes to browser

2. `Web app` is returned in the browser by Service Worker, initializes in the client. Matches routing, fetches data (**3**) and renders

3. If url is for UI or the article had been visited previously, it is returned instantly.

Otherwise, it goes to the API normally

Scope

Fundamental reading experience:

- Reading an article & images.
- Navigating to other articles
- Searching for articles

Article loading

- First payload serves only lead section
- Automatically loads rest of content

Pros

- Extremely fast perceived performance
- Get's useful info to user ASAP
- Flexibility on what to aggregate and serve on first visit + delayed

Cons

- Time to full content gets longer than sending it in one take
- 2 network requests

Lazy loading images

Images are not loaded by default.

Lazy loaded as the user scrolls through the article.

Pros

- Avoids blocking other assets when loading.
- Less data usage. Image not seen is image that doesn't consume data.

Cons

- HTML only (no-JS) doesn't load <img> (instead links to images)
  - Could be loaded if wanted (<noscript>)

Chrome & assets caching

- Leverage Service Workers
- Application shell and static assets are cached in the background
- We open an LRU cache for images

- Following visits to site serve instantly HTML chrome, JS and CSS
- JS loads content via API request (only request to server)

Benefits the most to recurrent visitors and power users

Support: Chrome 40+, FF 44, Opera, IE Edge

Chrome & assets caching

Pros

- Instant loading of chrome and assets on subsequent visits
- Website works offline. Available without internet connection
- Background updates of application assets

Cons

- First visit will trigger background requests for assets (network load)
  - HTTP caching may avoid downloading same assets twice

Content caching

- Cache content in browser DB (IndexedDB)
- Used to improve perceived performance and offline usage
- Cache article content, and other API fetched resources

- When visualizing, use Cache-First strategy
- Issue background requests for fresh content

Content caching

Pros

- Excellent perceived performance
  - Instant loading of already visited articles while background request fetches new content
- Offline reading of articles & other resources on the web
- Leaner requests to server (only content, no chrome/assets)

Cons

- Need a strategy for updating content in-front of user
  - Auto update (may seem weird if changes are big)
  - Ask user if show new content (intrusive)
- Needs DB content management strategy
  - Clean LRU content?
  - App upgrade - DB migrations / clean up

Summary

Pros

- Parsoid + Restbase allow content transformation
  - New content loading strategies to improve experience and speed
- Client side application + Service Worker gives the best & fastest experience possible recurrent visits
- Rendering on the server running the client web app
  - Less complexity & More shared code (server-client)

Cons

- New architecture, semi-incompatible with the MediaWiki frontend
  - Recreating what is already done
  - Loss of ecosystem
  - Unable to replicate what doesn't have an API (like native apps)
  - Could use web-app approach losing server side rendering (needs node.js)

Summary

Pros

- Development happiness
  - Clean UI architecture
    - Explicit centralized state management
    - Declarative views
    - Isolated component & styles
  - Faster development speed
  - Really fast UI tests (run on node.js w/ jsdom)
- Can deliver standalone web app from static server
  - Easier prototyping (impl+deploy+share)
  - Easier UX research w/ real application
- Can deploy to other platforms than pure browser
  - Electron & Cordova

Needs

- Parsoid content
  - Structured content for parsing and massaging
  - More semantic representation of important content (navboxes, references, etc)
  - Leaner output for faster load time
- Restbase
  - Efficient composition/aggregation services (leveraging HTTP caching)
  - Cache strategy for scale
- Effort on duplicating existing features

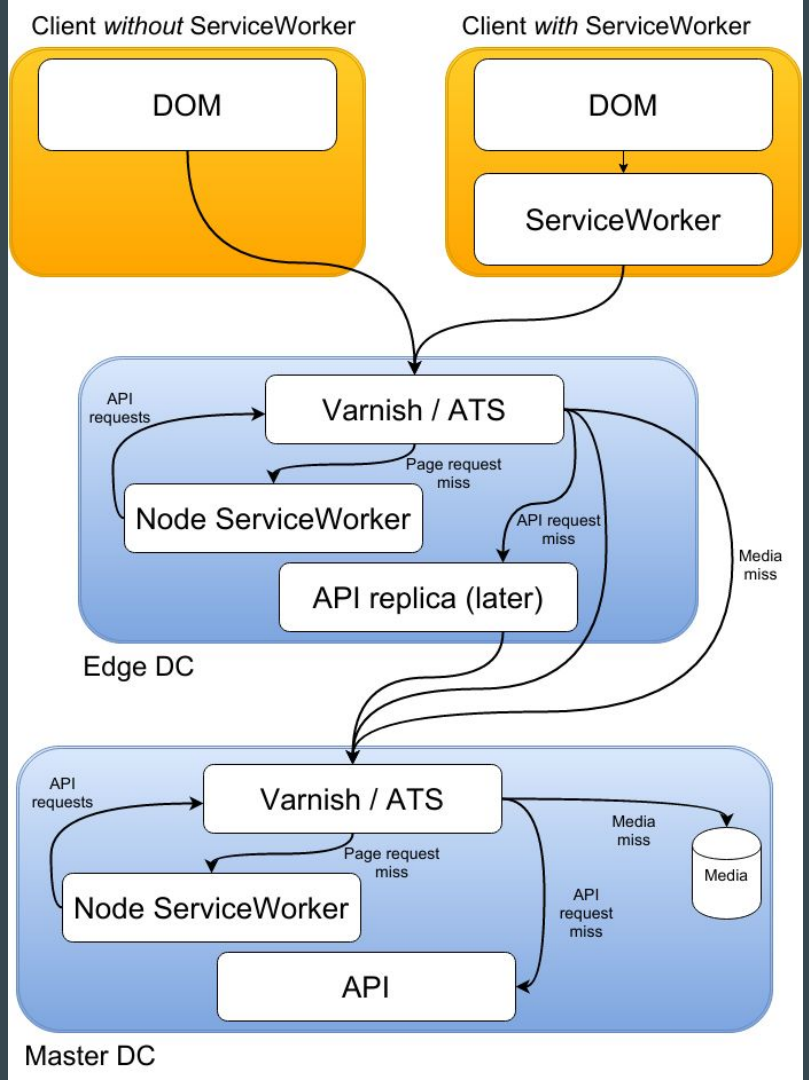
Answers to fundamental questions

- No-JS support?
  - Varnish cacheable HTML only server rendered version
    - Using same code as the client side app
    - Completely usable
- Cold load & no-JS (weak JS) performance?
  - See above. App works while assets load as html-only
- SEO
  - Since HTML only is able to access all content, spiders too

Alternative: ServiceWorker composition

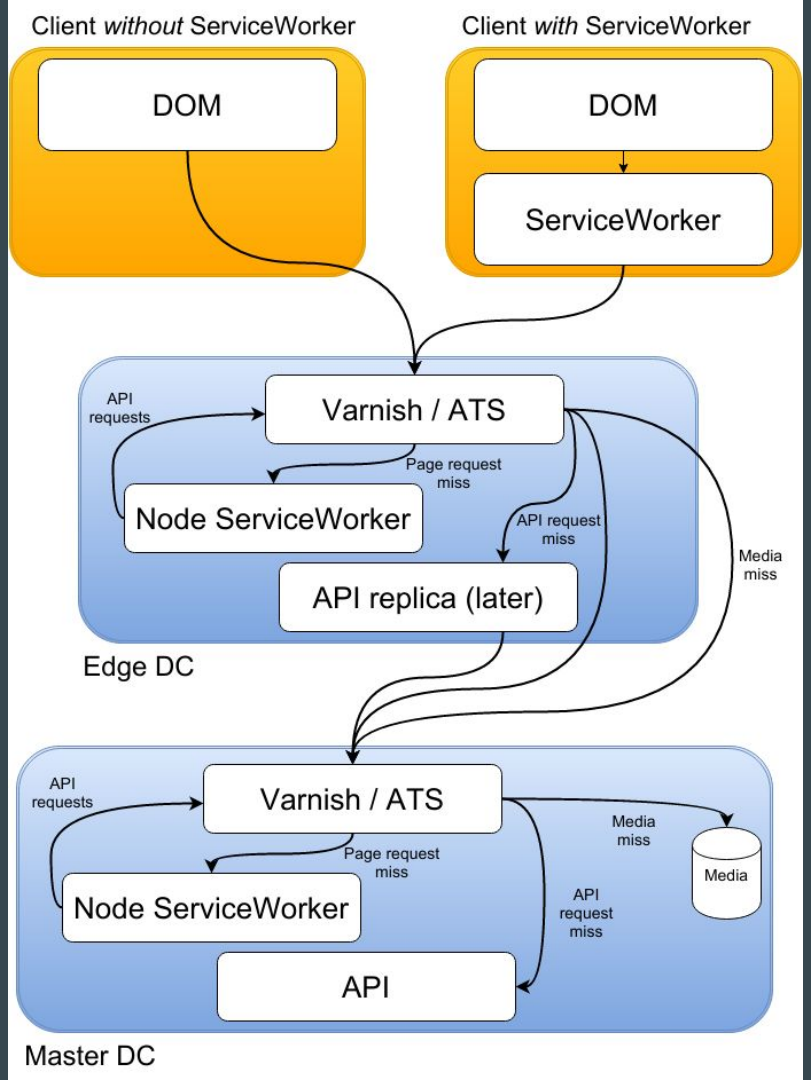
ServiceWorker composition

- Regular navigation, no single-page app.
- ServiceWorker is installed on first load & intercepts requests to specific URLs on subsequent accesses.
- Composes a streaming response based on stored content & API requests:
  - Header
    - styles
  - content
    - components
  - Footer
    - user tools
    - interwikis
    - page tools



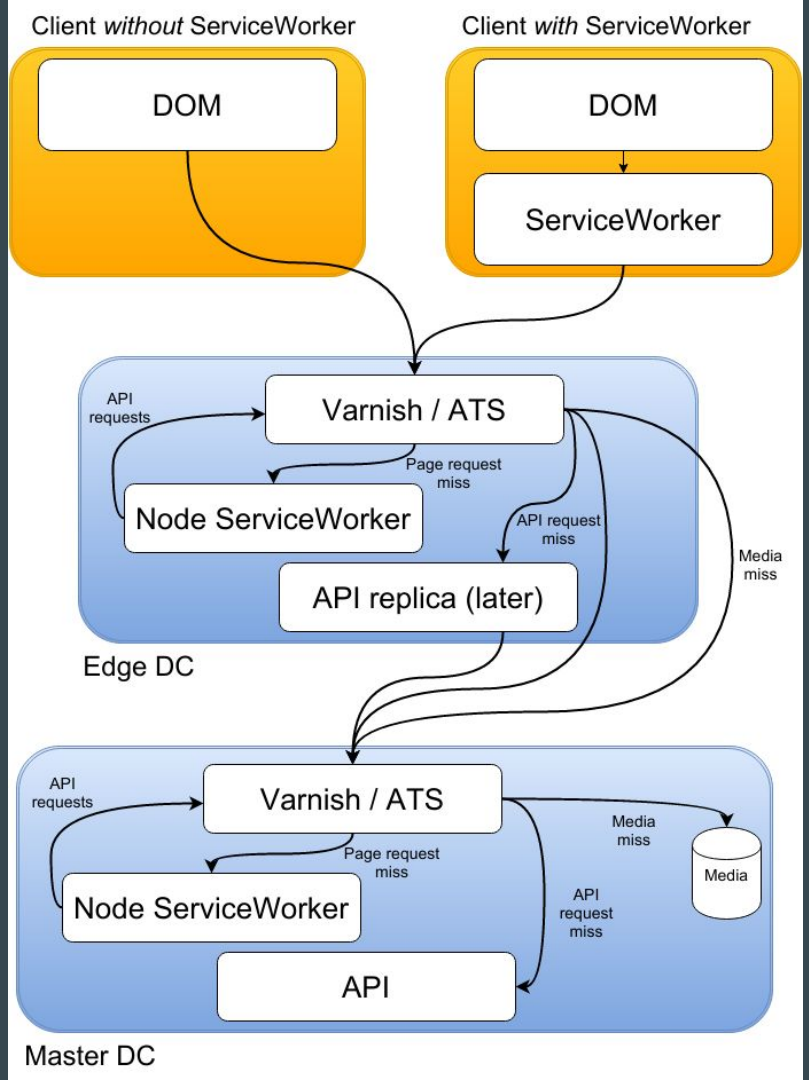
First load / No-JS

- Without JS / ServiceWorker, requests just hit Varnish.
- Varnish response is a fully cached, light-weight HTML version of page.
  - Contains all essential information, so does not break SEO.
  - Inline styles for quick rendering on 2G.
  - Omits non-essential below-fold content: Navboxes, possibly references.
- If JS is enabled & network performance is sufficient, optional content like navboxes is loaded after main content is ready (DOM).



Authenticated No-JS

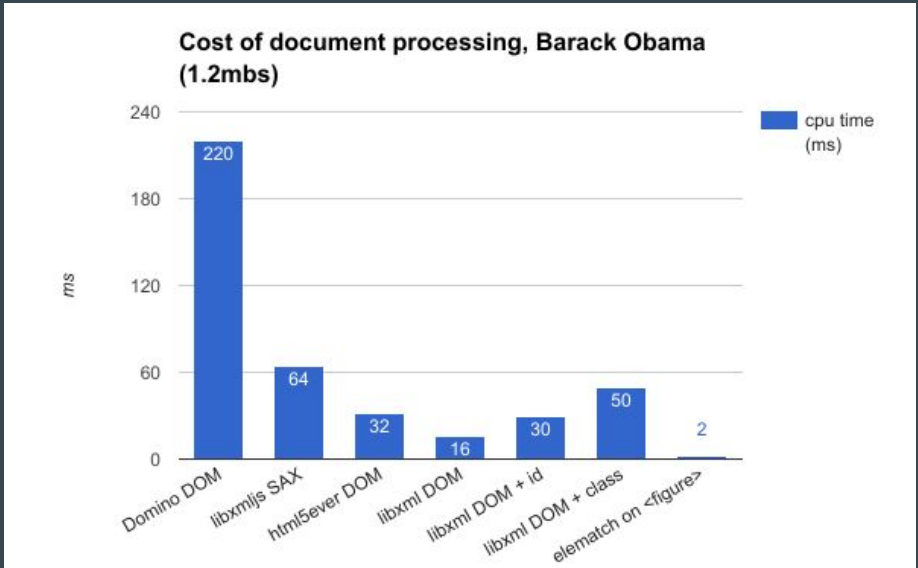
- Authenticated no-JS requests hit node.js service running the client-side ServiceWorker code via node-serviceworker.
- Streaming responses for low time to first byte.



Page components

- Goal: Clearly mark up elements for dynamic composition, and define a metadata spec for aggregation.
- Examples: <mw-infobox>, <mw-navbox>, <mw-references>, <figure>, <mw-interwikis>, <mw-usertools>
- Idea: Build on web components.
- Focus on performance, especially for server-side processing.

Composition performance



- elematch order of magnitude more efficient than libxml DOM
- Varnish cache hit order of magnitude faster than node service using streaming elematch processing

Page component APIs

- Need cacheable APIs for the retrieval of individual components.
- Goal: Identify shared patterns & reuse markup to API mapping.
- Define a metadata spec for components to enable dependency tracking and aggregation
  - Links, images, categories, used templates
  - Cache policy
  - Page properties

ServiceWorker composition vs. single-page app

- Pros
  - Can be introduced as a progressive enhancement, targeting only specific views.
  - Can be compatible with existing gadgets and user scripts.
  - No need for explicit history management, in-page routing & potential for state leakage.
  - Efficient byte stream processing, no DOM.
- Cons
  - Requires ServiceWorker support (~47%).
  - ServiceWorker ecosystem new & not as mature.

Summary

Goal: Leverage the modern web platform for

- better performance, and
- more flexibility, using well-known technologies.

Two main approaches:

- Single-page application
- ServiceWorker composition

Considerations

- How to handle language variants?
- Should we expose a fuller desktop rendering?
- Replace mobile / desktop skins with a single, adaptive frontend?
- Can we simplify skinning with this approach?
- Can we package easily for <pick your hosting provider>?
  - https://github.com/wikimedia/mediawiki-containers

How can we support this?

- Separate vital content and load stuff via the client when needed
- (Session T112987 @ 11:30AM - infoboxes/navboxes)
- Stop thinking of articles as monolithic chunks of text

  (Session T114072 @ 15:40 - <section> tags)
- Rethink references
- Rethink image loading strategies
- Educate our editors/devs (2G Tuesdays?) http://www.cnet.com/news/facebook-goes-slooow-with-2g-tuesdays/

Breakout

Okay, here’s what we heard.

Live typing what we heard!

FIN