196412

Creating Flexible Offline Workflows Using Autodesk Forge

Michael Beale
Autodesk

Michael Ponti
Honeywell

Learning Objectives

- Understand Forge workflows
- How to Deploy an offline Solution using Forge Viewer
- Develop new tools and Processes

Description

Autodesk Forge is an on-demand web services platform. That means you need to be online to connect to the Autodesk servers to use the various Forge APIs. However, the Autodesk Forge Viewer can work with multple online, as well as offline sources. There are times where an offline workflow is preferable because of lack of connectivity, security concerns, or performance. In this lecture, we will explore how to optimally use the Forge Viewer in an offline mode, and how to cache and synchronize the data with the server.

Speaker(s)

Michael Beale - Autodesk
Michael Ponti - Honeywell

Notes by Petr Broz (Autodesk Forge Team)
https://forge.autodesk.com/blog/disconnected-workflows

Introduction

While Forge is a *cloud* platform, certain applications built on top of it may want to support scenarios where the internet connection is temporarily unavailable. For example, consider an application for reviewing and annotating CAD models - wouldn't it be nice if you could work on a couple of CAD files while on a plane, and then perhaps sync your annotations when you're online again? In this post we take a look at one possible approach to support these scenarios using modern HTML5 APIs. We'll start with a quick introduction into the *technology* we're going to use, followed by the *strategy* we can employ to cache Forge content, and finally we'll take a look at a *sample application (Figure 1)*.

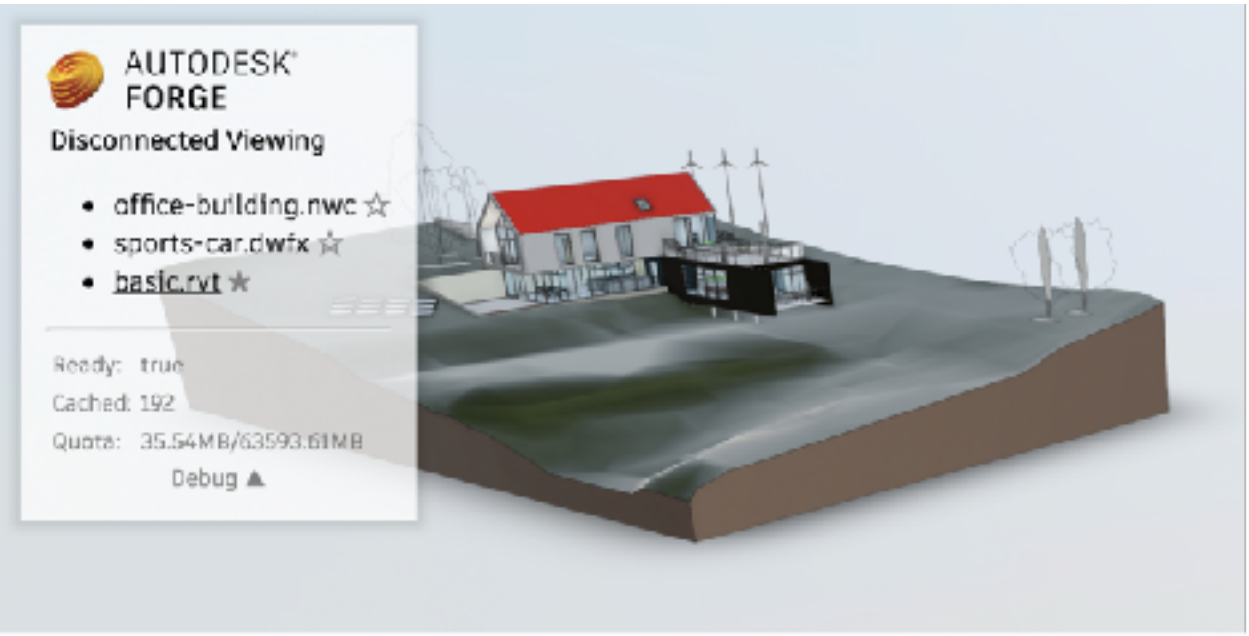

Figure 1 - Sample Application

Technology

There are different ways to store data from a web application or an online service on your device. In our sample application we will leverage a couple of new APIs from the increasingly more popular area of Progressive Web Applications, specifically *Service Workers*, *Cache*, and *Channel Messaging*.

While relatively new, these APIs are supported by most modern browsers. For a detailed overview, check out the Is Service Worker Ready? website by Jake Archibald.

Service Worker is a special type of Web Worker that acts as a proxy server for web applications from a specific origin. When a web application registers a service worker for itself, the worker can intercept network requests from potentially many instances of the application (in different browser tabs or windows) and respond with cached or even custom content. Apart from that, service workers have access to other modern APIs like IndexedDB, Channel Messaging, or Push APIs.

A typical lifecycle of a service worker looks like this:

- web app registers its service worker
- browser downloads and evaluates the worker script
- worker receives `install` event (used for one-time setup of resources)
- browser waits for all instances of the application (potentially using older service workers) to close
- worker receives `activate` event (used to clean older worker's cache, etc.)
- worker starts receiving `fetch` (to intercept network requests) and `message` (to communicate with the web app) events

Service worker lifecycle as explained by MDN

Cache is a per-origin storage similar to Local Storage or IndexedDB. It consists of uniquely named cache *objects*, and each *object* then stores individual HTTP *Request*/*Response* pairs.

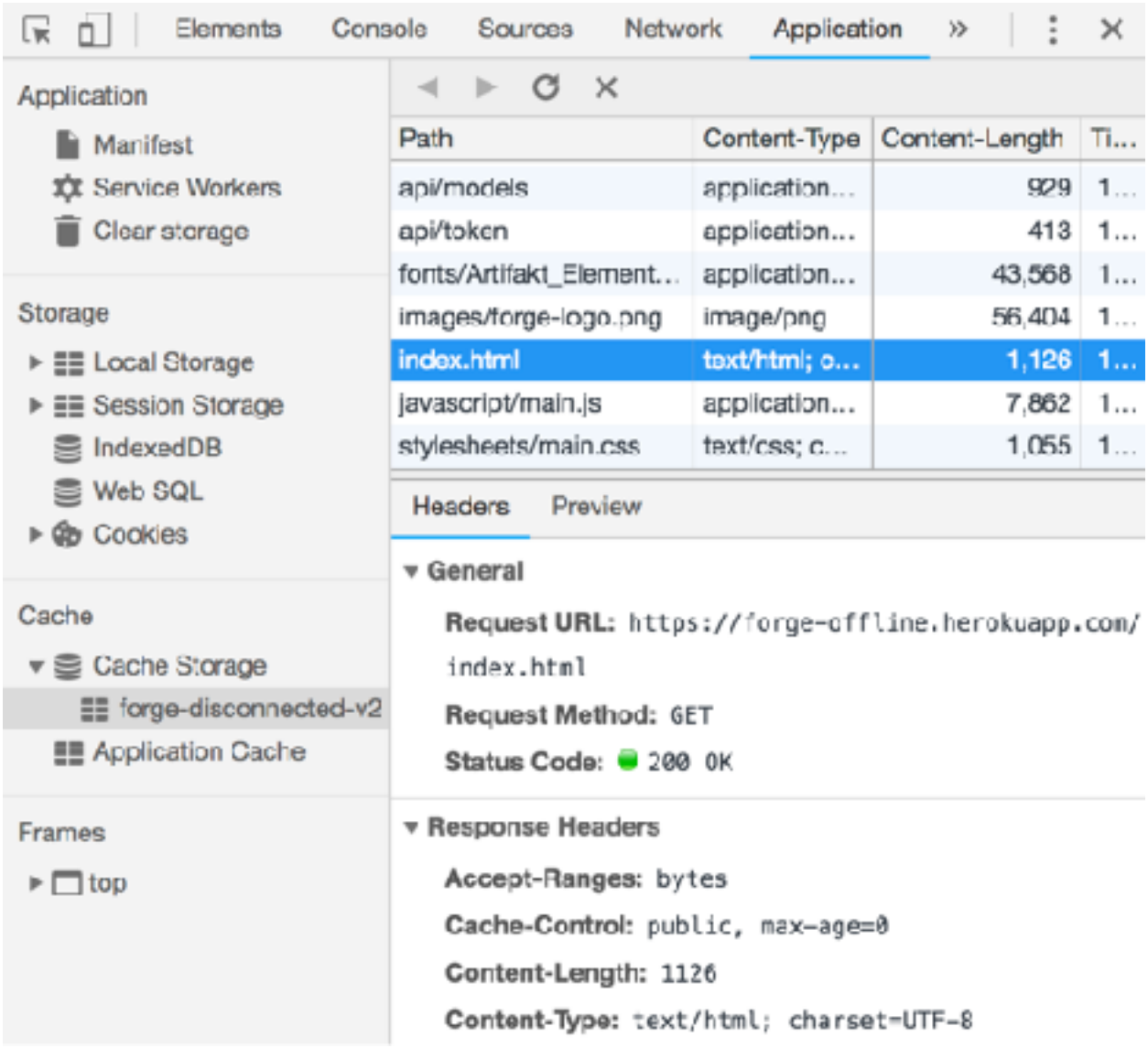

Figure 2 - Cache storage in Chrome DevTools

Channel Messaging allows scripts in different contexts (for example, between main document & iframe, between web worker & web worker, between main document & service worker, etc.) to communicate by passing messages through a two-way channel.

Caching Strategy

Caching static assets and API endpoints is straightforward. We can cache all of them when the service worker is installed. Then, when one of these endpoints is requested, the service worker can

provide the cached response immediately, and if needed, update the cache by fetching the resource in the background.

Caching individual models is a bit more involved. A single document in Forge typically generates multiple derivatives, and derivatives themselves often reference additional assets. We need a way to identify these assets in order to be able to cache them when needed. In our sample application, the server provides an endpoint which - given a document URN - provides a list of URLS for all its derivatives and assets (inspired by the code behind https://extract.autodesk.io). When caching a specific document, the service worker can use this endpoint and cache all the relevant URLs, without having to involve the viewer in any way!

Sample Application

We've prepared a sample Forge Viewer application that allows its users to selectively cache models from Model Derivative APIs. The source code is available at https://github.com/petrbroz/forge-disconnected, and a live demo is running at https://forge-offline.herokuapp.com. Let's look at the relevant pieces of the implementation.

On the backend, we're using a simple Express application which, apart from serving static content from the *public* folder, exposes the following 3 endpoints:

- *GET /api/token* - returns a 2-legged auth token for the viewer
- *GET /api/models* - returns a list of models for viewing
- *GET /api/models/:urn/files* - returns information about all derivatives and assets related to a specific model URN

On the client side, the two most important files are public/javascript/main.js and public/service-worker.js.

Most of the code in public/javascript/main.js is just configuring the Forge Viewer, setting up the UI overlay, and reacting to user input. The two important functions are *initServiceWorker* and *submitWorkerTask*, located towards the end of the file. The former is used to register our service worker, and the latter is used to post messages to it:

```
async function initServiceWorker() {
    try {
        const registration = await navigator.serviceWorker.register('/service-worker.js');
        console.log('Service worker registered', registration.scope);
    } catch (err) {
        console.error('Could not register service worker', err);
    }
}

function submitWorkerTask(task) {
    return navigator.serviceWorker.ready.then(function(req) {
        return new Promise(function(resolve, reject) {
            const channel = new MessageChannel();
            channel.port1.onmessage = function(event) {
                if (event.data.error) {
                    reject(event.data);
                } else {
                    resolve(event.data);
                }
            };
            req.active.postMessage(task, [channel.port2]);
        });
    });
}
```

The *submitWorkerTask* function is used to communicate 3 particular types of tasks to the service worker:

- requesting a list of cached URLs (used when updating the UI, to detect which models have already been cached)
- requesting a specific URN to be cached
- requesting a specific URN to be removed from cache

public/service-worker.js is where the magic happens. The code handles various events from the service worker lifecycle explained earlier. On the *install* event, we cache the known assets and APIs, and we also use the built-in *self.skipWaiting()* function to ask the browser to activate our worker as soon as possible, without waiting for older workers to finish their work.

```
async function installAsync(event) {
    self.skipWaiting();
    const cache = await caches.open(CACHE_NAME);
    await cache.addAll(STATIC_URLS);
    await cache.addAll(API_URLS);
}
```

On the *activate* event, we claim control of all instances of our web application potentially running in different browser tabs.

```
async function activateAsync() {
    const clients = await self.clients.matchAll({ includeUncontrolled: true });
    console.log('Claiming clients', clients.map(client => client.url).join(','));
    await self.clients.claim();
}
```

When intercepting requests via the *fetch* event, we reply with a cached response if there is one. One exception is the *GET /api/token* endpoint. Since our access token has an expiration time, we try to get a fresh token first, and only fall back to the cached one if we don't succeed.

```
async function fetchAsync(event) {
    // When requesting an access token, try getting a fresh one first
    if (event.request.url.endsWith('/api/token')) {
        try {
            const response = await fetch(event.request);
            return response;
        } catch(err) {
            console.log('Could not fetch new token, falling back to cache.', err);
        }
    }

    // If there's a cache match, return it
    const match = await caches.match(event.request.url, { ignoreSearch: true });
    if (match) {
        // If this is a static asset or known API, try updating the cache as well
        if (STATIC_URLS.includes(event.request.url) || API_URLS.includes(event.request.url)) {
            caches.open(CACHE_NAME)
                .then((cache) => cache.add(event.request))
                .catch((err) => console.log('Cache not updated, but that\'s ok...', err));
        }
        return match;
```

```
    }

    return fetch(event.request);
}
```

Finally, using the *message* event we execute "tasks" from the client.

```
async function messageAsync(event) {
    switch (event.data.operation) {
        case 'CACHE_URN':
            try {
                const urls = await cacheUrn(event.data.urn, event.data.access_token);
                event.ports[0].postMessage({ status: 'ok', urls });
            } catch(err) {
                event.ports[0].postMessage({ error: err.toString() });
            }
            break;
        case 'CLEAR_URN':
            try {
                const urls = await clearUrn(event.data.urn);
                event.ports[0].postMessage({ status: 'ok', urls });
            } catch(err) {
                event.ports[0].postMessage({ error: err.toString() });
            }
            break;
        case 'LIST_CACHES':
            try {
                const urls = await listCached();
                event.ports[0].postMessage({ status: 'ok', urls });
            } catch(err) {
                event.ports[0].postMessage({ error: err.toString() });
            }
            break;
    }
}

async function cacheUrn(urn, access_token) {
    console.log('Caching', urn);
    // First, ask our server for all derivatives in this URN, and their file URLs
    const baseUrl = 'https://developer.api.autodesk.com/derivativeservice/v2';
    const res = await fetch(`/api/models/${urn}/files`);
    const derivatives = await res.json();
    // Prepare fetch requests to cache all the URLs
    const cache = await caches.open(CACHE_NAME);
    const options = { headers: { 'Authorization': 'Bearer ' + access_token } };
    const fetches = [];
    const manifestUrl = `${baseUrl}/manifest/${urn}`;
    fetches.push(fetch(manifestUrl, options).then(resp => cache.put(manifestUrl, resp)).then(() =>
manifestUrl));
    for (const derivative of derivatives) {
        const derivUrl = baseUrl + '/derivatives/' + encodeURIComponent(derivative.urn);
        fetches.push(fetch(derivUrl, options).then(resp => cache.put(derivUrl, resp)).then(() => derivUrl));
        for (const file of derivative.files) {
            const fileUrl = baseUrl + '/derivatives/' + encodeURIComponent(derivative.basePath + file);
            fetches.push(fetch(fileUrl, options).then(resp => cache.put(fileUrl, resp)).then(() => fileUrl));
        }
    }
    // Fetch and cache all URLs in parallel
    const urls = await Promise.all(fetches);
    return urls;
}

async function clearUrn(urn) {
    console.log('Clearing cache', urn);
    const cache = await caches.open(CACHE_NAME);
    const requests = (await cache.keys()).filter(req => req.url.includes(urn));
```

```
    await Promise.all(requests.map(req => cache.delete(req)));
    return requests.map(req => req.url);
}

async function listCached() {
    console.log('Listing caches');
    const cache = await caches.open(CACHE_NAME);
    const requests = await cache.keys();
    return requests.map(req => req.url);
}
```

And that's pretty much it. If you want to see this code in action, head over to https://forge-offline.herokuapp.com with your favorite (modern) browser, open the dev. tools, and try caching on of the listed models using the ☆ symbol next to their title.

For a video of this code in action - click on the youtube video here: https://www.youtube.com/watch?v=JGLytRddYiw