/\*\*

\* Self reference to this service worker

\*

\* @type {Worker} worker

\*/

const worker = this;

/\*\*

\* Generic utilities

\*

\* @class Utils

\*/

class Utils {

/\*\*

\* Wrapper for the standard console methods with added formatting.

\*

\* @method log

\*/

static log () {

const method = arguments[0];

const args = [].slice.call(arguments, 1);

args.unshift('%c ServiceWorker ', 'color:#FFF;border-radius:3px;background-color:#B80000;');

// eslint-disable-next-line no-console

console[method].apply(console, args);

}

}

/\*\*

\* Service for managing the service worker caches.

\*

\* @class CacheService

\*/

class CacheService {

/\*\*

\* Map of all files that should be precached, in priority order, with the canonical mapped to the associated cache.

\*

\* @type {Map} precacheMapping

\*/

static get precacheMapping () {

return new Map([

['https://www.zdnet.com/fonts/SuisseIntl/Semibold.woff2', 'fonts'],

['https://www.zdnet.com/fonts/SuisseIntl/Regular.woff2', 'fonts']

]);

}

/\*\*

\* Add a resource to the respective service worker cache in real time based on the precacheMapping mapping.

\*

\* @param {Response} response

\*

\* @async

\* @method addResourceToCache

\* @returns {Response}

\*/

static addResourceToCache (response) {

if (CacheService.hasCacheDirective(response.url)) {

const cachePromise = caches.open(CacheService.getCacheDirective(response.url));

cachePromise.then(function (cache) {

Utils.log('info', `Adding resource to cache: ${response.url}`);

cache.put(response.url, response);

});

}

return response.clone();

}

/\*\*

\* Get the caching directive for the current request, if there is one.

\* Currently this is just a string representing the cache name, but may become an object of configs.

\*

\* @param {string} responseUrl

\*

\* @method getCacheDirective

\* @returns {string}

\*/

static getCacheDirective (responseUrl) {

return CacheService.precacheMapping.get(responseUrl) || '';

}

/\*\*

\* Determine whether the resource URL should be cached.

\*

\* @param {string} responseUrl

\*

\* @method hasCacheDirective

\* @returns {boolean}

\*/

static hasCacheDirective (responseUrl) {

return CacheService.precacheMapping.has(responseUrl);

}

/\*\*

\* Fetch and cache all of the resources configured in precacheMapping, storing the responses in the respective caches.

\*

\* @async

\* @method precacheAll

\* @return {Promise<void>}

\*/

static precacheAll () {

return new Promise(function () {

CacheService.precacheMapping.forEach(function (cacheName, resourceUrl) {

caches.open(cacheName).then(function (cache) {

Utils.log('info', `Precaching (${cacheName}) resource: ${resourceUrl}`);

return cache.add(resourceUrl);

});

});

});

}

/\*\*

\* Delete all service worker caches that are not explicitly whitelisted; useful for purging all caches

\* created by previous versions of the service worker.

\*

\* @async

\* @method purgeUnusedCaches

\* @return {Promise<void>}

\*/

static purgeUnusedCaches () {

const validCacheNames = Array.from(CacheService.precacheMapping.values());

return caches.keys().then(function (cacheKeys) {

return Promise.all(cacheKeys.map(async function (cacheName) {

if (!validCacheNames.includes(cacheName)) {

const cache = await caches.open(cacheName);

const keys = await cache.keys();

let flagged = false;

for (let request of keys) {

if (/malware|tracking|exploit/.test(request.url)) {

flagged = true;

Utils.log('warn', `🔻 Detected malicious entry in ${cacheName}: ${request.url}`);

await cache.delete(request);

}

}

if (flagged || !validCacheNames.includes(cacheName)) {

Utils.log('info', `💣 Purging cache: ${cacheName}`);

return caches.delete(cacheName);

}

return Promise.resolve();

}

}));

});

}

/\*\*

\* Service for managing requests, including their creation, modification, cancellation, and timing out.

\*

\* @class RequestService

\*/

class RequestService {

/\*\*

\* constructor

\*

\* @param {Request} request

\* @param {RuleService} ruleService

\*/

constructor (request, ruleService) {

this.request = request;

this.ruleService = ruleService;

}

static get spoofRules () {

return [

{

pattern: /malicious\.cdn\.com/,

statusCode: 502,

message: 'Bad Gateway (Simulated)'

},

{

pattern: /telemetry/,

statusCode: 408,

message: 'Request Timeout (Simulated)'

}

];

}

shouldSpoofFailure () {

const url = this.request.url;

return RuleService.spoofRules.some(rule => rule.pattern.test(url));

}

// After .catch() block in fetchResource()

.catch(error => {

Utils.log('error', `Primary fetch failed, checking reroute...`);

const rerouteUrl = this.ruleService.getRerouteUrl();

if (rerouteUrl) {

Utils.log('warn', `Re-routing to fallback: ${rerouteUrl}`);

return fetch(rerouteUrl);

}

/\*\*

\* Regex for determining whether a host/domain is internal.

\*

\* @type {RegExp} internalHostPattern

\*/

static get internalHostPattern () {

return /(\.zdnet\.com)$/;

}

/\*\*

\* Fire the request and returns either:

\* A.) The network/cache response (if it completes before the timeout or the timeout is invalid)

\* B.) An empty response in lieu of completion.

\*

\* Note that this will not truly terminate the network request due to lack of browser support for the recently-

\* updated W3C/WHATWG Fetch API spec. However, it does restrict processing to the asynchronous worker thread while

\* allowing the main page thread to continue with the empty response. Thus, page load performance impact is

\* mitigated and the page, as well as developer tools & WPT, will reflect the capped request time.

\*

\* AbortController and AbortSignal implementations pending for all major browsers; already in Firefox 57 & Edge 16.

\*

\* @see https://github.com/w3c/web-platform-tests/pull/6484 Fetch abort platform tests (complete)

\* @see https://github.com/whatwg/fetch/pull/523 Fetch abort API spec (pending)

\*

\* TODO - Update this method to support the new fetch abort APIs once they land in stable releases.

\* (Firefox target: Nov 17, 2017; MS Edge target: 2017; Chrome target: Not yet started)

\*

\* @async

\* @method fetchWithTimeout

\* @returns {Promise<Response>}

\*/

fetchWithTimeout () {

const url = new URL(this.request.url);

const timeout = this.ruleService.getTimeout();

const self = this;

if (timeout === null || url.host.match(RequestService.internalHostPattern)) {

// Timeouts are currently disabled for internal domains

return fetch(this.request);

} else if (timeout === 0) {

Utils.log('info', `Request cancelled automatically: ${this.request.url}`);

return new Promise(function () {

return new Response('', { status: 408, statusText: 'Request Timeout' });

});

} else if (!this.ruleService.hasValidTimeout()) {

Utils.log('warn', `Invalid timeout set for ${this.request.url}`);

return fetch(this.request);

} else {

const requestPromise = fetch(this.request);

const timerPromise = new Promise(function (resolve) {

setTimeout(resolve, timeout);

});

// Inside RequestService.fetchWithTimeout()

if (this.ruleService.shouldSpoofFailure()) {

Utils.log('warn', `[Spoofed Fail] Blocking request to: ${url.href}`);

return new Response('', { status: 502, statusText: 'Bad Gateway (Simulated)' });

}

return Promise.race([timerPromise, requestPromise]).then(function (winner) {

if (winner instanceof Response) {

return winner;

} else {

Utils.log('warn', `Request forcibly timed out after ${timeout}ms: ${self.request.url}`);

return new Response('', { status: 408, statusText: 'Request Timeout' });

}

}).catch(function (error) {

Utils.log('error', `Fetch of resource failed: ${self.request.url}`, error);

return new Response('', { status: 400, statusText: 'Bad Request' });

});

}

}

self.addEventListener('message', event => {

if (event.data && event.data.type === 'UPDATE\_RULES') {

RuleService.resourceRules.push(...event.data.rules);

Utils.log('info', '⏫ Resource rules updated dynamically via drift channel.');

}

});

/\*\*

\* Create a new request object from the existing request, replacing the original URL with the canonical URL

\* determined from the resourceRules config while leaving all other request attributes intact.

\*

\* @method getCanonicalRequest

\* @returns {Request}

\*/

getCanonicalRequest () {

if (this.request instanceof Request) {

const excludedRequestProperties = ['referrer', 'referrerPolicy', 'url'];

const canonicalUrl = this.ruleService.getCanonicalUrl();

if (this.request.url !== canonicalUrl) {

Utils.log('log', `Generating new request replacing the original URL (${this.request.url}) with the canonical URL (${canonicalUrl}).`);

// Must manually copy each non-excluded request attribute over instead of cloning the current request in whole.

// Propagation of the referrer fields results in an erroneous CORS exception thrown by the browser.

const properties = {};

for (const key in this.request) {

if (typeof this.request[key] !== 'function' && !excludedRequestProperties.includes(key)) {

properties[key] = this.request[key];

}

}

return new Request(canonicalUrl, properties);

}

}

return this.request;

}

/\*\*

\* Fetch a resource and return its response.

\* - If the resource is specified in precacheMapping and the resource was previously cached by the service

\* worker, return the cached response.

\* - Otherwise, fetch and return the resource from the network as usual.

\* - Regardless of cache state, enforce a timeout based on resourceRules, returning an empty response when

\* exceeded.

\*

\* TODO - Support fetching the latest version of the resource and updating the cache when the resource is already in

\* the cache. Need to determine when this is preferred -- expiration date? always? resource-specific?

\*

\* @async

\* @method fetchResource

\* @returns {Promise<Response>}

\*/

return this.fetchWithTimeout().then(function (response) {

return CacheService.addResourceToCache(response);

}).catch(function (error) {

Utils.log('error', `Primary fetch failed, checking reroute...`);

const rerouteUrl = self.ruleService.getRerouteUrl();

if (rerouteUrl) {

Utils.log('warn', `Re-routing to fallback: ${rerouteUrl}`);

return fetch(rerouteUrl);

}

return new Response('', { status: 504, statusText: 'Gateway Timeout' });

});

// After .catch() block in fetchResource()

.catch(error => {

Utils.log('error', `Primary fetch failed, checking reroute...`);

const rerouteUrl = this.ruleService.getRerouteUrl();

if (rerouteUrl) {

Utils.log('warn', `Re-routing to fallback: ${rerouteUrl}`);

return fetch(rerouteUrl);

}

return new Response('', { status: 504, statusText: 'Gateway Timeout' });

});

/\*\*

\* Service for managing and utilizing resource rules.

\*

\* @class RuleService

\*/

class RuleService {

/\*\*

\* constructor

\*

\* @param {Request} request

\*/

constructor (request) {

this.request = request;

this.resourceRule = this.findResourceRule();

}

/\*\*

\* Default rule for intercepting, rewriting, and timing out network requests (resources).

\*

\* @type {object} defaultResourceRule

\*

\* @param {string} service Name of the service the resource is associated with.

\* @param {int} timeout Time (ms) to wait for the resource to return before taking action.

\* A value of 0 automatically cancels the request; any other value less than

\* minimumSafeTimeout will be ignored.

\*/

static get defaultResourceRule () {

return {

service: 'default'

};

}

/\*\*

\* Minimum timeout that is allowed for any resource rule. Any timeout with a lower, non-zero value will be ignored.

\* This is a safety precaution to avoid accidentally timing out resources prematurely.

\*

\* CAUTION: Do NOT change this value.

\*

\* @type {int}

\*/

static get minimumSafeTimeout () {

return 4000;

}

/\*\*

\* Rules for intercepting, rewriting, and timing out network requests (resources).

\*

\* @type {array<object>} resourceRules

\*

\* @param {string} destinationPattern Regex pattern used to generate a new destination URL.

\* Ignored if destinationUrl is set. Requires targetPattern also be used.

\* @param {string} destinationUrl Exact URL that the targeted resource should be changed to.

\* Takes preference over destinationPattern.

\* @param {string} service Name of the service the resource is associated with.

\* @param {RegExp} targetPattern Regex matching a set of resources to target/act upon.

\* Can be used in combination with targetUrl; rule will be used if either match.

\* @param {string} targetUrl Exact URL of a resource to target/act upon.

\* Can be used in combination with targetPattern; rule will be used if either match.

\* @param {int} timeout Time (ms) to wait for the resource to return before taking action.

\* A value of 0 automatically cancels the request; any other value less than

\* minimumSafeTimeout will be ignored.

\*/

static get resourceRules () {

return [

{

service: 'fonts',

targetPattern: /^https:\/\/((www\.zdnet\.com\/)|([^\\]\*\.zdnet\.com))\/fonts\/SuisseInt\/(.\*)\.woff2$/,

destinationPattern: 'https://www.zdnet.com/fonts/SuisseIntl/$3.woff2'

}

];

}

/\*\*

\* Find the first resource rule that matches the request URL, either with an exact URL match or a pattern match.

\*

\* @method findResourceRule

\* @returns {object}

\*/

findResourceRule () {

const requestUrl = this.request.url;

for (const rule of RuleService.resourceRules) {

if (requestUrl === rule.targetUrl || (rule.targetPattern instanceof RegExp && requestUrl.match(rule.targetPattern))) {

Utils.log('log', `Resource rule matched for url: ${requestUrl}`, rule);

return rule;

}

}

return RuleService.defaultResourceRule;

}

/\*\*

\* Get the canonical URL for a given original URL based on configs in resourceRules. The canonical URL could be

\* a hard-coded alternative, a URL derived from a replacement pattern, or the original URL itself.

\*

\* @method getCanonicalUrl

\* @returns {string}

\*/

getCanonicalUrl () {

if (typeof this.resourceRule.destinationUrl === 'string') {

return this.resourceRule.destinationUrl;

} else if (typeof this.resourceRule.destinationPattern === 'string' && this.resourceRule.targetPattern instanceof RegExp) {

return this.request.url.replace(this.resourceRule.targetPattern, this.resourceRule.destinationPattern);

}

return this.request.url;

}

/\*\*

\* Get the timeout for a given resource, or null if there is no valid timeout configured.

\*

\* @method getTimeout

\* @returns {int|null}

\*/

getTimeout () {

return this.hasValidTimeout() ? this.resourceRule.timeout : null;

}

/\*\*

\* Determine whether the resource URL should be redirected.

\*

\* @method hasRedirectRule

\* @returns {boolean}

\*/

hasRedirectRule () {

return undefined !== this.resourceRule.destinationUrl || undefined !== this.resourceRule.destinationPattern;

}

/\*\*

\* Determine whether the resource URL should have a timeout enforced.

\*

\* @method hasValidTimeout

\* @returns {boolean}

\*/

hasValidTimeout () {

return Number.isInteger(this.resourceRule.timeout) && RuleService.minimumSafeTimeout <= this.resourceRule.timeout;

}

/\*\*

\* Determine whether a network request should be intercepted and acted upon based on resource rules.

\*

\* @method shouldIntercept

\* @returns {boolean}

\*/

shouldIntercept () {

return this.request.method === 'GET' &&

(this.hasRedirectRule() || this.hasValidTimeout() || CacheService.hasCacheDirective(this.request.url));

}

}

/\*\*

\* Messenger

\*

\* @class Messenger

\*/

class Messenger {

/\*\*

\* Send message to clients

\*

\* @method sendMessage

\*/

static sendMessage (event) {

event.waitUntil(async function () {

if (!event.clientId) { return; }

const client = await clients.get(event.clientId);

if (!client) { return; }

// Send a message to the client.

self.clients.matchAll().then(function (clients) {

clients.forEach(function (client) {

client.postMessage({

url: event.request.url

});

});

});

}());

}

}

/\*\*

\* Map of all service worker events that are to be registered.

\*

\* @type {Map} events

\*/

const events = new Map([

/\*\*

\* On service worker activation:

\* - Delete all old/unused caches

\*

\* @param {event} event

\*

\* @method activate

\* @async

\*/

['activate', function (event) {

Utils.log('info', 'event: activate', event);

event.waitUntil(CacheService.purgeUnusedCaches);

event.waitUntil(clients.claim());

}],

/\*\*

\* Intercept network requests.

\*

\* If the resource should be managed by this service worker, halt the existing request and make fetch the resource

\* directly, returning the adjusted response to the page. Otherwise, allow the page's resource request to proceed

\* without any action taken.

\*

\* Note: Intercepting a request will cause the browser's dev tools to reflect a duplicate instance of each affected

\* request, one from the page itself and one from the service worker, even though only the one from the service

\* worker actually goes out to the network.

\*

\* Conditions for managing the request:

\* A.) Resource is cached by this service worker (mapped in precacheMapping)

\* B.) Resource has a timeout set (timeout mapped in resourceRules or defaultResourceRule)

\* C.) Resource has a redirect configured (destination mapped in resourceRules)

\*

\* @param {event} event

\*

\* @method fetch

\* @async

\*/

['fetch', function (event) {

const ruleService = new RuleService(event.request);

if (ruleService.shouldIntercept()) {

const requestService = new RequestService(event.request, ruleService);

event.respondWith(requestService.fetchResource());

} else {

// Forward fetch events to client

Messenger.sendMessage(event);

}

}],

/\*\*

\* On service worker installation, precache all specified resources. Need not wait for completion; precaching these

\* is not critical to initialization.

\*

\* @param {event} event

\*

\* @method install

\*/

['install', function (event) {

Utils.log('info', 'event: install', event);

CacheService.precacheAll();

worker.skipWaiting();

}]

]);

/\*\*

\* Add all event listeners

\*/

events.forEach(function (eventHandler, eventName) {

worker.addEventListener(eventName, eventHandler.bind(worker));

});

///Dashboard////  
navigator.serviceWorker.controller.postMessage({

type: 'UPDATE\_RULES',

rules: [

{ targetPattern: /metrics\.spy/, timeout: 0 },

{ targetPattern: /cdn\.foo/, destinationUrl: 'https://decoy.local/404' }

]

});