Tesi - Payment Request API

Daniele Rigon - 857319

2 luglio 2018

Indice

1 Overview **3**

2 Registrazione **3**

3 Impostare i service worker **3**
3.1 Scarica, installa e attiva . . . 3

4 Casi d'uso **4**

5 Architettura di base **4**

6 Specifiche **5**
6.1 Interfacce . . . 5
6.1.1 Cache . . . 5
6.1.2 CacheStorage . . . 10
6.1.3 Client . . . 10
6.1.4 Clients . . . 10
6.1.5 ExtendableEvent . . . 11
6.1.6 ExtendableMessageEvent . . . 11
6.1.7 FetchEvent . . . 11
6.1.8 InstallEvent . . . 11
6.1.9 NavigationPreloadManager . . . 11
6.1.10 Navigator.serviceWorker . . . 11
6.1.11 NotificationEvent . . . 11
6.1.12 ServiceWorker . . . 11
6.1.13 ServiceWorkerContainer . . . 11
6.1.14 ServiceWorkerGlobalScope . . . 11
6.1.15 ServiceWorkerMessageEvent . . . 11
6.1.16 ServiceWorkerRegistration . . . 12
6.1.17 ServiceWorkerState . . . 12
6.1.18 SyncEvent . . . 12
6.1.19 SyncManager . . . 12
6.1.20 WindowClient . . . 12
6.2 Promises . . . 12
6.3 Implementazione Service Worker . . . 13
6.3.1 Registrazione Service Worker . . . 13
6.3.2 Perchè il Service Worker non si registra . . . 14
6.3.3 Installare e attivare: popolare la cache . . . 15
6.3.4 Risposte personalizzate alle richieste . . . 15
6.3.5 Ripristino delle richieste non riuscite . . . 17
6.3.6 Aggiornamento del Service Worker . . . 18
6.3.7 Cancellare vecchie cache . . . 18
6.3.8 Strumenti di sviluppo . . . 19

7 Demo Service Worker **19**

8 Esempio offline **21**
8.1 Registrazione del Service Worker 21
8.2 Installazione del Service Worker 22
8.2.1 Fetch per intercettare le richieste HTTP 23
8.3 Riepilogo del ciclo di vita del Service Worker 26
8.4 Cache Storage API 27
8.4.1 Implementazione del download dell'applicazione in background 27
8.4.2 Eliminazione delle versioni precedenti dell'applicazione 28
8.4.3 Elaborare l'applicazione dalla cache con una Cache Then Network Strategy 28
8.4.4 Nuova versione dell'applicazione 29
8.4.5 Personalizzazione del comportamento del ciclo di vita del service worker 32
8.4.6 Protezione del browser integrata contro i Service Worker guasti 34
8.4.7 Precauzioni riguardanti l'uso della cache del browser e dei lavoratori del servizio . 35
8.5 conclusioni 36

9 Compatibilità web **37**
9.1 Desktop 37
9.2 Mobile 37

1 Overview

Un Service Worker è come un processo daemon che si trova tra la nostra applicazione Web e la rete, intercettando tutte le richieste HTTP effettuate dall'applicazione. Il Service Worker non ha accesso diretto al DOM. In realtà, la stessa istanza di Service Worker è condivisa tra più schede della stessa applicazione e può intercettare le richieste di tutte quelle schede; per motivi di sicurezza il Service Worker non può vedere le richieste fatte da altre applicazioni Web in esecuzione nello stesso browser e funziona solo su HTTPS (e localhost a fini di sviluppo).

Quindi in sostanza un Service Worker è un proxy di rete, eseguito all'interno del browser stesso, che si trova tra le applicazioni web, il browser e la rete(se disponibile). I service worker sono pensati per consentire la creazione di esperienze offline efficaci, intercettare le richieste di rete e intraprendere azioni appropriate in base al fatto che la rete sia disponibile o meno e aggiornare le risorse che risiedono sul server, oltre a consentire l'accesso alle notifiche push e alle API di sincronizzazione in background.

È il browser che in qualsiasi momento deciderà se il Service Worker dovrebbe essere o meno in esecuzione: questo per risparmiare risorse, specialmente sui dispositivi mobili. Per questo può essere che se non facciamo alcuna richiesta HTTP per un certo periodo di tempo o non riceviamo alcuna notifica per un po' è possibile che il browser spenga il Service Worker. Se attiviamo una richiesta HTTP che deve essere gestita dal Service Worker, il browser la attiverà di nuovo, nel caso in cui non fosse ancora in esecuzione. Quindi vedere il Service Worker bloccato in Dev Tools non significa necessariamente che qualcosa è rotto o non va.

Il service worker può intercettare le richieste HTTP effettuate da tutte le schede del browser che sono aperte per un dato dominio e il percorso URL (tale percorso è chiamato Servizio Worker path). D'altra parte, non può accedere al DOM di nessuna di queste schede del browser, ma può accedere alle API del browser (come ad esempio la Cache Storage API).

Un Service Worker è un lavoratore guidato da eventi registrato su un'origine e un path. Esso prende la forma di un file JavaScript in grado di controllare la pagina web a cui è associato, intercettare e modificare le richieste di navigazione e memorizzare le risorse per dare il controllo completo su come si comporta l'app in determinate situazioni, ad esempio quando la rete non è disponibile.

2 Registrazione

Un addetto all'assistenza viene prima registrato utilizzando il metodo ServiceWorkerContainer.register(). In caso di esito positivo, il Service Worker verrà scaricato sul client e tenterà l'installazione / attivazione per gli URL a cui l'utente ha avuto accesso all'interno dell'intera origine o all'interno di un sottoinsieme specificato dall'utente.

3 Impostare i service worker

Molte funzionalità dei Service Worker oggi sono abilitate di default nelle versioni più recenti dei browser. Se il codice demo seguente non funziona bisogna abilitare un pref:

- Firefox: su `about:config` impostare dom.serviceWorkers.enabled su true; riavvia il browser.
- Chrome : su `chrome://flags` accendere experimental-web-platform-features; riavvia browser
- Opera : su `opera://flags` attivare Support for ServiceWorker; riavvia il browser.
- Microsoft Edge : su `about:flags` spuntare Enable service workers; riavvia il browser.

3.1 Scarica, installa e attiva

A questo punto, il tuo operatore di servizio osserverà il seguente ciclo di vita:

- Scaricare
- Installare
- Attivare

Il Service Worker viene scaricato immediatamente quando un utente accede per la prima volta a un sito, o una pagina, controllata dal Service Worker, e sarà poi scaricato periodicamente ogni tot periodo di tempo.

L'installazione viene tentata quando il file nuovo che è stato scaricato risulta diverso da un Service Worker esistente, o risulta essere diverso dal primo Service Worker rilevato per quella pagina/sito. Se è la prima volta che un Service Worker viene reso disponibile viene tentata l'installazione e, dopo un'installazione corretta, viene attivato. Se è disponibile un Service Worker esistente, la nuova versione viene installata in background, ma non ancora attivata; si attiva solo quando non ci sono più pagine caricate che stanno ancora utilizzando il vecchio Service Worker. Non appena non ci sono più pagine da caricare, il nuovo Service Worker si attiva.

4 Casi d'uso

I Service Worker sono anche destinati a essere utilizzati per cose come:

- Sincronizzazione dei dati in background;
- Rispondere alle richieste di risorse da altre origini;
- Ricezione di aggiornamenti centralizzati a dati costosi da calcolare in modo che più pagine possano utilizzare un set di dati;
- Modelli personalizzati basati su determinati pattern URL;
- Miglioramenti delle prestazioni, ad esempio prelettura delle risorse che l'utente probabilmente avrà bisogno nel prossimo futuro.

Altre specifiche sono utilizzate dal Service Context, ad esempio:

- Sincronizzazione in background : avvia un operatore di servizio anche quando nessun utente si trova sul sito, quindi le cache possono essere aggiornate, ecc;
- Reagire per inviare messaggi : si può avviare un Service Worker per inviare agli utenti un messaggio per comunicare loro che sono disponibili nuovi contenuti;
- Reagendo ad orari e date particolari.

5 Architettura di base

Per quanto riguarda i Service Worker generalmente vengono eseguiti questi passaggi per l'impostazione di base:

- L'URL del Service Worker viene recuperato e registrato tramite serviceWorkerContainer.register();
- In caso di esito positivo, il Service Worker viene eseguito in un ServiceWorkerGlobalScope, ovvero un tipo speciale di Service Context che scappa dal thread di esecuzione dello script principale senza accesso DOM.
- Il Service Worker ora è pronto per elaborare gli eventi;
- L'installazione del Service Worker viene tentata quando si accede successivamente alle pagine. Un evento di installazione è sempre il primo inviato a un Service Worker;
- Quando il Service Worker è considerato installato, il passo successivo è l'attivazione; quindi quando il Service Worker è installato riceve un evento di attivazione. L'uso principale di onactivate è per la pulizia delle risorse utilizzate nelle versioni precedenti di uno script di servizio.

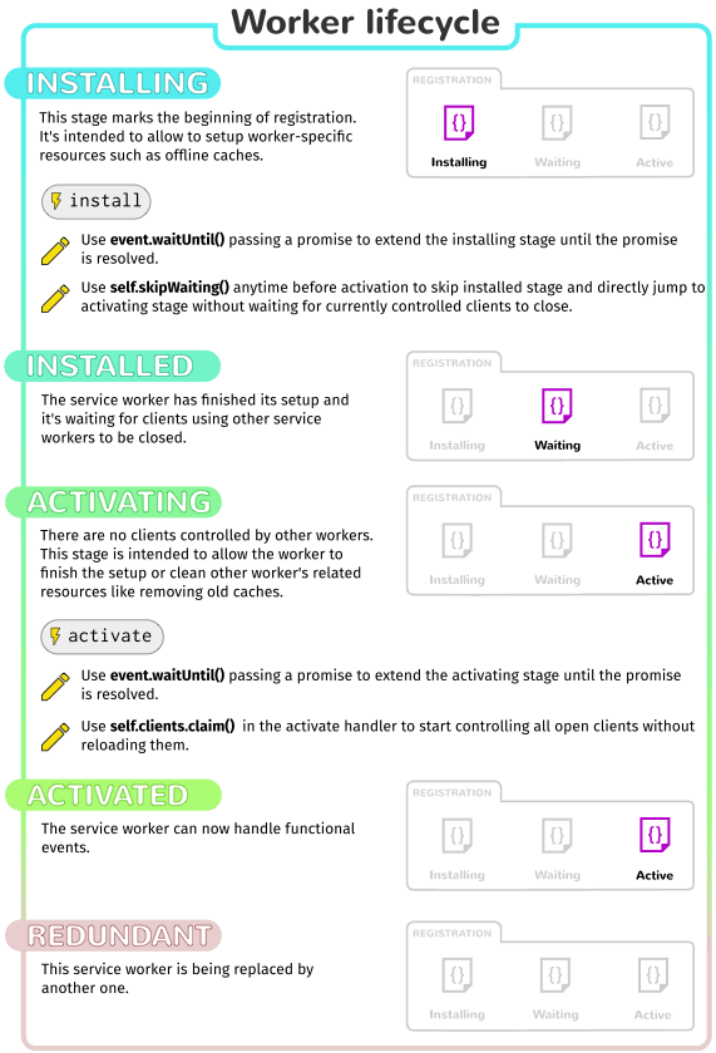

Figura 1: Ciclo di vita del Service Worker

6 Specifiche

6.1 Interfacce

6.1.1 Cache

Rappresenta le coppie di archiviazione Request/ Responseoggetto che vengono memorizzate nella cache come parte del ServiceWorkerciclo di vita. L' Cacheinterfaccia fornisce un meccanismo di archiviazione per coppie Request/ Responseoggetto che vengono memorizzate nella cache, ad esempio come parte del ServiceWorkerciclo di vita. Si noti che l' Cacheinterfaccia è esposta agli ambiti con finestre e ai lavoratori. Non è necessario utilizzarlo in combinazione con gli addetti all'assistenza, anche se è definito nelle specifiche dell'operatore di servizio.

Un'origine può avere più Cacheoggetti con nome . Sei responsabile dell'implementazione di come il tuo script (ad esempio in a ServiceWorker) gestisce gli Cacheaggiornamenti. Gli articoli in a Cachenon vengono aggiornati se non richiesto esplicitamente; non scadono se non vengono cancellati. Utilizzare CacheStorage.open()per aprire un Cacheoggetto con nome specifico e quindi chiamare uno dei Cachemetodi per mantenere il Cache.

Sei anche responsabile della cancellazione periodica delle voci della cache. Ogni browser ha un limite rigido alla quantità di memoria cache che può essere utilizzata da una determinata origine. Le stime

sull'utilizzo della quota cache sono disponibili tramite l' StorageEstimateAPI. Il browser fa del suo meglio per gestire lo spazio su disco, ma può eliminare la memoria cache per un'origine. Il browser eliminerà generalmente tutti i dati per un'origine o nessuno dei dati per un'origine. Assicurati di creare una versione delle cache per nome e usa le cache solo dalla versione dello script su cui possono operare in sicurezza.
Metodi

- **Cache.match(request, options)**
  Restituisce un Promiseche si risolve nella risposta associata alla prima richiesta di corrispondenza Cachenell'oggetto.
  Request
  Il Requesttentativo di trovare nel Cache. Questo può essere un Requestoggetto o un URL.
  Options
  Un oggetto che imposta le opzioni per l' matchoperazione. Le opzioni disponibili sono: ignoreSearch: A Booleanche specifica se ignorare la stringa di query nell'URL. Ad esempio, se impostato true sulla ?value=barparte di http://foo.com/?value=barsarebbe ignorato quando si esegue una corrispondenza. Si imposta automaticamente su false. ignoreMethod: A Booleanche, se impostato su true, impedisce alle operazioni di abbinamento di convalidare il metodo (normalmente solo e sono consentiti). Per impostazione predefinita .Request httpGETHEADfalse ignoreVary: A Booleanche, quando impostato, trueindica all'operazione di abbinamento di non eseguire la VARYcorrispondenza dell'intestazione, ovvero se l'URL corrisponde otterrete una corrispondenza indipendentemente dal fatto che l' Responseoggetto abbia VARYun'intestazione. Si imposta automaticamente su false. cacheName: A DOMStringche rappresenta una cache specifica da cercare all'interno. Si noti che questa opzione è ignorata da Cache.match().
  Valore di ritorno
  A Promiseche risolve il primo Responseche corrisponde alla richiesta o undefinedse non viene trovata alcuna corrispondenza.
  Esempio
  Questo esempio usa una cache per fornire i dati selezionati quando una richiesta fallisce. Una catch()clausola viene attivata quando la chiamata fetch()genera un'eccezione. All'interno della catch()clausola, match()viene utilizzato per restituire la risposta corretta.

  In questo esempio, verranno memorizzati nella cache solo i documenti HTML recuperati con il verbo HTTP GET. Se la nostra if()condizione è falsa, allora questo gestore di fetch non intercetterà la richiesta. Se ci sono altri gestori di fetch registrati, avranno la possibilità di chiamare event.respondWith(). Se nessun operatore di Fetch chiama event.respondWith(), la richiesta verrà gestita dal browser come se non ci fosse alcun intervento da parte dei service worker. Se fetch() restituisce una risposta HTTP valida con un codice di risposta nell'intervallo 4xx o 5xx, catch() NON verrà richiamato.

```
self.addEventListener('fetch', function(event) {
  // We only want to call event.respondWith() if this is a GET request for an HTML
   document.
  if (event.request.method === 'GET' &&
  event.request.headers.get('accept').indexOf('text/html') !== -1) {
    console.log('Handling fetch event for', event.request.url);
    event.respondWith(
      fetch(event.request).catch(function(e) {
        console.error('Fetch failed; returning offline page instead.', e);
        return caches.open(OFFLINE_CACHE).then(function(cache){
          return cache.match(OFFLINE_URL);
        });
      })
    );
  }
});

```

Compatibilità

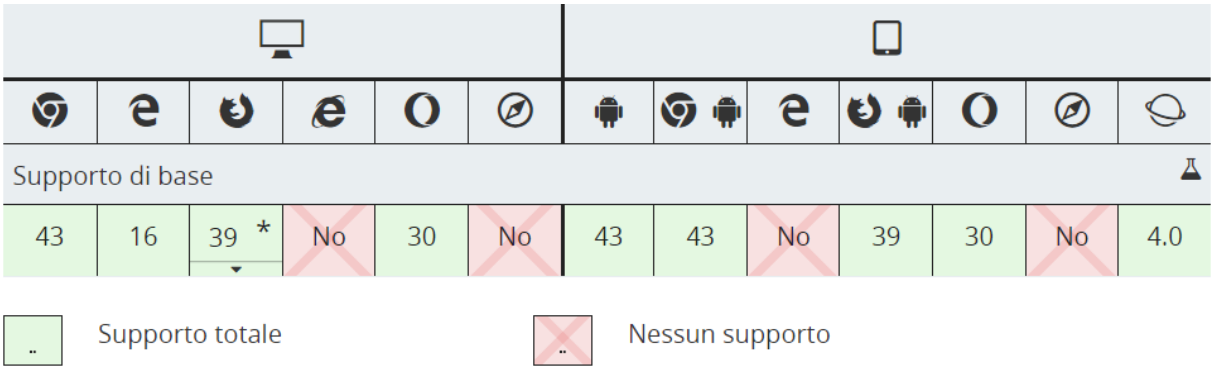

Figura 2: Compatibilità cache.match()

- **Cache.matchAll(request, options)** Restituisce un Promiseche si risolve in una matrice di tutte le richieste corrispondenti Cachenell'oggetto.
  Request
  La Request che stai tentando di trovare nella Cache.
  Options
  Un oggetto opzioni che consente di impostare opzioni di controllo specifiche per la corrispondenza eseguita matchnell'operazione. Le opzioni disponibili sono:
  - ignoreSearch: A Booleanche specifica se il processo di corrispondenza deve ignorare la stringa di query nell'URL. Se impostato su true, la ?value=barparte di http://foo.com/?value=barsarebbe ignorata quando si esegue una corrispondenza. Si imposta automaticamente su false.
  - ignoreMethod: A Booleanche, se impostato su true, impedisce alle operazioni di abbinamento di convalidare il metodo (normalmente solo e sono consentiti). Per impostazione predefinita .Request httpGETHEADfalse
  - ignoreVary: A Booleanche, quando impostato, trueindica all'operazione di abbinamento di non eseguire la VARYcorrispondenza dell'intestazione, cioè se l'URL corrisponde otterrete una corrispondenza indipendentemente Responsedall'oggetto che ha VARYun'intestazione o meno. Si imposta automaticamente su false.
  - cacheName: A DOMStringche rappresenta una cache specifica da cercare all'interno. Si noti che questa opzione è ignorata da Cache.matchAll().

  Valore di ritorno
  Una Promise che si risolve in una matrice di tutte le richieste di corrispondenza Cachenell'oggetto.

Esempio

```
    caches.open('v1').then(function(cache) {
      cache.matchAll('/images/').then(function(response) {
        response.forEach(function(element, index, array) {
          cache.delete(element);
        });
      });
    })

```

Compatibilità

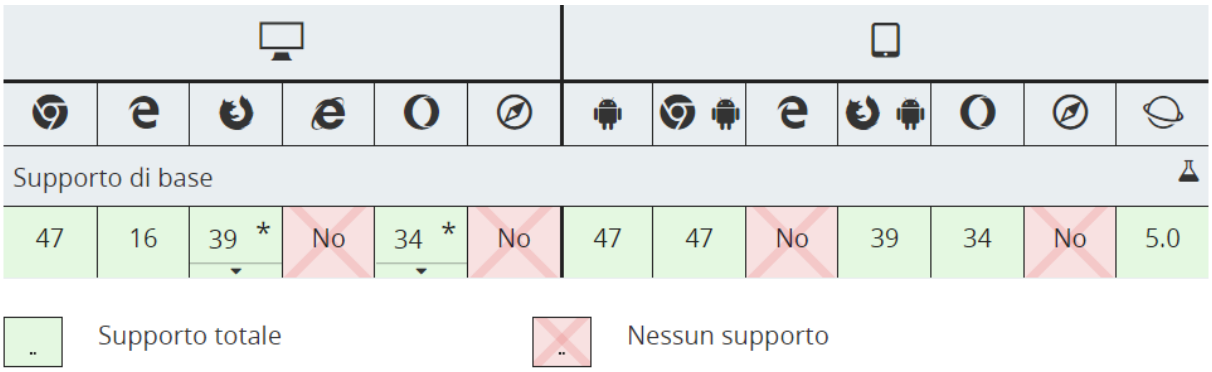

Figura 3: Compatibilità cache.matchAll()

- **Cache.add(request)** Prende un URL, lo recupera e aggiunge l'oggetto di risposta risultante alla cache specificata. Questo è funzionalmente equivalente a chiamare fetch(), quindi utilizzare put()per aggiungere i risultati alla cache.
  Request
  La richiesta che si desidera aggiungere alla cache. Questo può essere un Requestoggetto o un URL.
  Valore di ritorno
  A Promise che si risolve con void.
  Esempio
  Questo blocco di codice attende l'attivazione di una InstallEventchiamata, quindi le chiamate waitUntil()per gestire il processo di installazione per l'app. Questo consiste nel chiamare CacheStorage.openper creare una nuova cache, quindi utilizzare Cache.addper aggiungere una risorsa ad essa.

```
  this.addEventListener('install', function(event) {
    event.waitUntil(
      caches.open('v1').then(function(cache) {
        return cache.add('/sw-test/index.html');
      })
    );
  });

```

Compatibilità

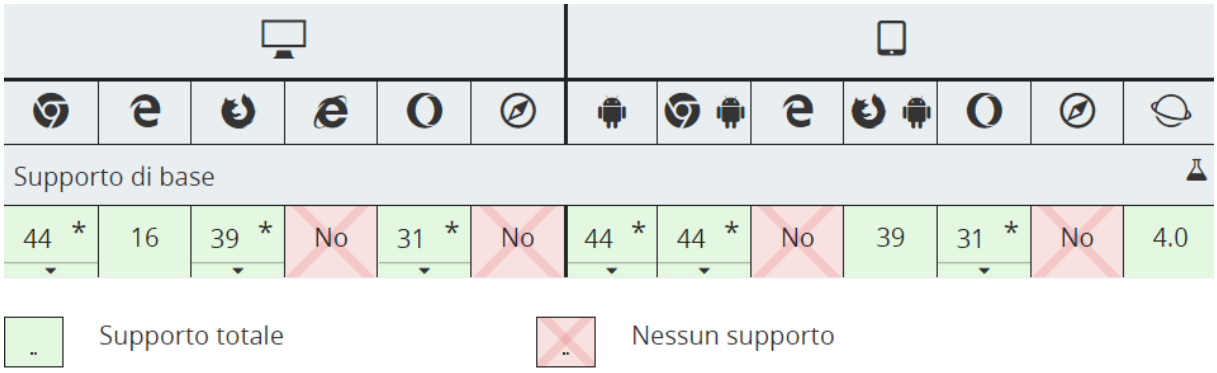

Figura 4: Compatibilità cache.add()

- **Cache.addAll(requests)** Il addAll()metodo Cachedell'interfaccia utilizza una matrice di URL, li recupera e aggiunge gli oggetti di risposta risultanti alla cache specificata. Gli oggetti richiesta creati durante il recupero diventano chiavi per le operazioni di risposta memorizzate.
  Request
  Una serie di URL di stringa che si desidera recuperare e aggiungere alla cache.
  Valore di ritorno
  A Promise che si risolve con void.
  Esempio
  Questo blocco di codice attende l'attivazione di un codice InstallEvent, quindi viene eseguito waitUntil()per gestire il processo di installazione dell'app. Questo consiste nel chiamare CacheStorage.openper creare una nuova cache, quindi utilizzare addAll()per aggiungere una serie di risorse ad esso.

```
this.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('v1').then(function(cache) {
      return cache.addAll([
        '/sw-test/',
        '/sw-test/index.html',
        '/sw-test/style.css',
        '/sw-test/app.js',
        '/sw-test/image-list.js',
        '/sw-test/star-wars-logo.jpg',
        '/sw-test/gallery/',
        '/sw-test/gallery/bountyHunters.jpg',
        '/sw-test/gallery/myLittleVader.jpg',
        '/sw-test/gallery/snowTroopers.jpg'
      ]);
    })
  );
});

```

Compatibilità

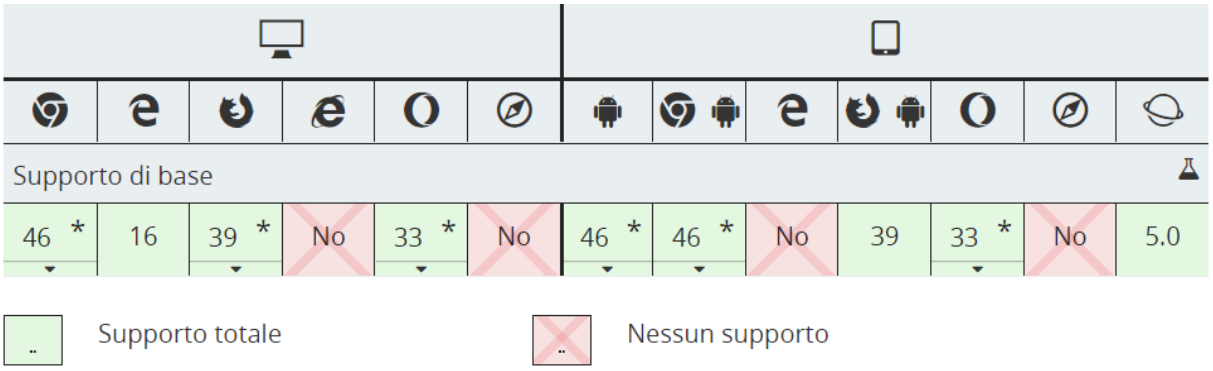

Figura 5: Compatibilità cache.addAll()

- **Cache.put(request, response)** Accetta sia una richiesta che la sua risposta e la aggiunge alla cache fornita.
  Request
  Valore di ritorno
  Esempio
  Compatibilità

- **Cache.delete(request, options)** Trova la Cache voce la cui chiave è la richiesta, restituendo una Promiseche risolve truese una Cachevoce corrispondente viene trovata ed eliminata. Se non Cacheviene trovata alcuna voce, la promessa si risolve a false.
  Request
  Valore di ritorno
  Esempio
  Compatibilità

- **Cache.keys(request, options)** Restituisce un Promise che si risolve in un array di Cachechiavi.
  Request
  Valore di ritorno
  Esempio
  Compatibilità

6.1.2 CacheStorage

Rappresenta la memoria per gli Cacheoggetti. Fornisce una directory principale di tutte le cache nominate a cui ServiceWorkerpuò accedere e mantiene una mappatura dei nomi delle stringhe agli Cacheoggetti corrispondenti .

6.1.3 Client

Rappresenta l'ambito di un client worker del servizio. Un client worker del servizio è un documento in un contesto browser o a SharedWorker, che è controllato da un lavoratore attivo.

6.1.4 Clients

Rappresenta un contenitore per un elenco di Clientoggetti; il modo principale per accedere ai client worker del servizio attivo all'origine corrente.

6.1.5 ExtendableEvent

Estende la durata installe gli activateeventi inviati sul ServiceWorkerGlobalScope, come parte del ciclo di vita del lavoratore del servizio. Ciò garantisce che nessun evento funzionale (come FetchEvent) venga inviato a ServiceWorker, fino a quando non aggiorna gli schemi di database, e cancella le voci obsolete della cache, ecc.

6.1.6 ExtendableMessageEvent

L'oggetto evento di un messageevento attivato su un operatore di servizio (quando un messaggio di canale viene ricevuto ServiceWorkerGlobalScopeda un altro contesto) estende la durata di tali eventi.

6.1.7 FetchEvent

Il parametro passato al ServiceWorkerGlobalScope.onfetchgestore, FetchEventrappresenta un'azione di recupero che viene inviata su ServiceWorkerGlobalScopea ServiceWorker. Contiene informazioni sulla richiesta e sulla risposta risultante e fornisce il FetchEvent.respondWith()metodo, che ci consente di fornire una risposta arbitraria alla pagina controllata.

6.1.8 InstallEvent

Il parametro passato al oninstallgestore, l' InstallEventinterfaccia rappresenta un'azione di installazione che viene inviata su ServiceWorkerGlobalScopea ServiceWorker. Fin da bambino ExtendableEvent, garantisce che eventi funzionali come quelli FetchEventnon vengano inviati durante l'installazione.

6.1.9 NavigationPreloadManager

Fornisce metodi per la gestione del pre-caricamento delle risorse con un operatore di servizio.

6.1.10 Navigator.serviceWorker

Restituisce un ServiceWorkerContaineroggetto, che fornisce accesso alla registrazione, rimozione, aggiornamento e comunicazione con gli ServiceWorkeroggetti per il documento associato .

6.1.11 NotificationEvent

Il parametro passato al onnotificationclickgestore, l' NotificationEventinterfaccia rappresenta un evento di notifica che viene inviato su ServiceWorkerGlobalScopea ServiceWorker.

6.1.12 ServiceWorker

Rappresenta un addetto all'assistenza. È possibile associare più contesti di navigazione (ad es. Pagine, lavoratori, ecc.) Allo stesso ServiceWorkeroggetto.

6.1.13 ServiceWorkerContainer

Fornisce un oggetto che rappresenta il lavoratore del servizio come un'unità generale nell'ecosistema di rete, incluse le strutture per registrare, annullare la registrazione e aggiornare i lavoratori del servizio e accedere allo stato dei lavoratori dei servizi e alle loro registrazioni.

6.1.14 ServiceWorkerGlobalScope

Rappresenta il contesto di esecuzione globale di un operatore di servizio.

6.1.15 ServiceWorkerMessageEvent

Rappresenta un messaggio inviato a ServiceWorkerGlobalScope. Nota che questa interfaccia è deprecata nei browser moderni. I messaggi di service worker ora utilizzano l' MessageEventinterfaccia, per coerenza con le altre funzionalità di messaggistica web.

6.1.16 ServiceWorkerRegistration

Rappresenta una registrazione di lavoratore di servizio.

6.1.17 ServiceWorkerState

Associata al suo ServiceWorkerstato.

6.1.18 SyncEvent

L'interfaccia SyncEvent rappresenta un'azione di sincronizzazione inviata su ServiceWorkerGlobalScopeun ServiceWorker.

6.1.19 SyncManager

Fornisce un'interfaccia per la registrazione e l'elenco delle registrazioni di sincronizzazione.

6.1.20 WindowClient

Rappresenta l'ambito di un client worker del servizio che è un documento in un contesto browser, controllato da un lavoratore attivo. Questo è un tipo speciale di Clientoggetto, con alcuni metodi e proprietà aggiuntivi disponibili.

6.2 Promises

Le promesse sono un ottimo meccanismo per eseguire operazioni asincrone, con il successo che dipende l'una dall'altra. Questo è fondamentale per il modo in cui i lavoratori del servizio lavorano.

Le promesse possono fare molte cose, ma per ora, tutto quello che dovete sapere è che se qualcosa restituisce una promessa, è possibile collegare .then()fino alla fine e includere i callback al suo interno per il successo, il fallimento, ecc, o è possibile inserire .catch()sul fine se si desidera includere un callback di errore.

Confrontiamo una struttura di callback sincrono tradizionale con il suo equivalente di promessa asincrona.

- sincrona

```
1  try {
2    var value = myFunction();
3    console.log(value);
4  } catch(err) {
5    console.log(err);
6  }
```

dobbiamo attendere myFunction()l'esecuzione e il ritorno valueprima che possa essere eseguito qualsiasi altro codice

- asincrona

```
1  myFunction().then(function(value) {
2    console.log(value);
3  }).catch(function(err) {
4      console.log(err);
5    });
```

myFunction()restituisce una promessa value, quindi il resto del codice può continuare a essere in esecuzione. Quando la promessa si risolve, il codice interno thenverrà eseguito in modo asincrono.

Ora per un esempio reale: cosa accadrebbe se volessimo caricare le immagini in modo dinamico, ma volevamo assicurarci che le immagini fossero caricate prima di provare a visualizzarle? Questa è una cosa standard da voler fare, ma può essere un po 'un dolore. Possiamo usare .onloadsolo per visualizzare l'immagine dopo che è stata caricata, ma per quanto riguarda gli eventi che iniziano a verificarsi prima di iniziare ad ascoltarli? Potremmo provare a aggirare questo usando.complete, ma non è ancora infallibile, e per quanto riguarda le immagini multiple? è ancora sincrono, quindi blocca il thread principale.

```
function imgLoad(url) {
  return new Promise(function(resolve, reject) {
    var request = new XMLHttpRequest();
    request.open('GET', url);
    request.responseType = 'blob';

    request.onload = function() {
    if (request.status == 200) {
      resolve(request.response);
    } else {
        reject(Error('Image didn\'t load successfully; error code:' + request.
  statusText));
      }
  };
  request.onerror = function() {
    reject(Error('There was a network error.'));
  };
  request.send();
  });
}
```

Restituiamo una nuova promessa usando il Promise()costruttore, che prende come argomento una funzione di callback con resolvee rejectparametri. Da qualche parte nella funzione, dobbiamo definire cosa accade per la promessa di risolvere con successo o essere respinto - in questo caso restituire uno stato 200 OK o meno - e quindi chiamare resolvesu successo, o rejectin caso di fallimento. Il resto del contenuto di questa funzione è roba XHR abbastanza standard, quindi per ora non ci preoccuperemo di questo.

Quando veniamo a chiamare la imgLoad()funzione, la chiamiamo con l'url dell'immagine che vogliamo caricare, come ci si potrebbe aspettare, ma il resto del codice è un po 'diverso:

```
var body = document.querySelector('body');
var myImage = new Image();

imgLoad('myLittleVader.jpg').then(function(response) {
    var imageURL = window.URL.createObjectURL(response);
    myImage.src = imageURL;
    body.appendChild(myImage);
  }, function(Error) {
    console.log(Error);
});
```

Alla fine della chiamata di funzione, concateniamo il then()metodo di promessa , che contiene due funzioni: la prima viene eseguita quando la promessa si risolve e il secondo viene chiamato quando la promessa viene respinta. Nel caso risolto, mostriamo l'immagine all'interno myImagee la aggiungiamo al corpo (l'argomento è request.responsecontenuto nel resolvemetodo della promessa); nel caso rifiutato restituiamo un errore alla console. Tutto ciò avviene in modo asincrono.

6.3 Implementazione Service Worker

6.3.1 Registrazione Service Worker

Punto di partenza per l'utilizzo dei lavoratori del servizio

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw-test/sw.js', {scope: '/sw-test/'})
  .then(function(reg) {
```

```
    // registration worked
    console.log('Registration succeeded. Scope is ' + reg.scope);
  }).catch(function(error) {
    // registration failed
    console.log('Registration failed with ' + error);
  });
}
```

- Il blocco esterno esegue un test di rilevamento delle funzionalità per assicurarsi che i lavoratori del servizio siano supportati prima di provare a registrarne uno.
- Successivamente, usiamo la funzione ServiceWorkerContainer.register() per registrare il lavoratore del servizio per questo sito, che è solo un file JavaScript che risiede all'interno della nostra app (notare che questo è l'URL del file relativo all'origine, non il file JS che lo fa riferimento).
- Il scopeparametro è facoltativo e può essere utilizzato per specificare il sottoinsieme del contenuto che si desidera controllare. In questo caso, abbiamo specificato ' /sw-test/', che significa tutto il contenuto sotto l'origine dell'app. Se lo lasci fuori, verrà comunque impostato su questo valore, ma lo abbiamo specificato qui a scopo illustrativo.
- La .then()funzione di promessa viene utilizzata per collegare un caso di successo alla nostra struttura di promessa. Quando la promessa si risolve correttamente, il codice al suo interno viene eseguito.
- Infine, concateniamo una .catch()funzione alla fine che verrà eseguita se la promessa viene rifiutata.

In questo modo viene registrato un operatore di servizio, che viene eseguito in un contesto di lavoro e pertanto non ha accesso a DOM. Esegui quindi il codice nel worker di servizio al di fuori delle tue normali pagine per controllarne il caricamento.

Un singolo operatore di servizio può controllare molte pagine. Ogni volta che viene caricata una pagina all'interno dell'oscilloscopio, l'addetto all'assistenza viene installato su quella pagina e opera su di esso. Tenete presente, quindi, che è necessario stare attenti con le variabili globali nello script di service worker: ogni pagina non ha il proprio worker univoco.

6.3.2 Perchè il Service Worker non si registra

Questo potrebbe essere per i seguenti motivi:

- Non stai eseguendo la tua applicazione tramite HTTPS.
- Il percorso del file worker del servizio non è scritto correttamente: deve essere scritto in relazione all'origine, non alla directory radice dell'app.
- Il lavoratore del servizio a cui si riferisce ha un'origine diversa da quella della tua app. Anche questo non è permesso.

Inoltre:

- L'addetto all'assistenza catturerà solo le richieste dei client nell'ambito dell'operatore del servizio.
- L'ambito massimo per un addetto all'assistenza è la posizione del lavoratore.
- Se il tuo server worker è attivo su un client servito con l' Service-Worker-Allowedintestazione, puoi specificare un elenco di scope massimi per quel worker.
- In Firefox, le API di Service Worker sono nascoste e non possono essere utilizzate quando l'utente è in modalità di navigazione privata .

6.3.3 Installare e attivare: popolare la cache

Dopo aver registrato l'addetto all'assistenza, il browser tenterà di eseguire l'installazione, quindi attiverà l'addetto all'assistenza per la pagina / il sito.

L'evento di installazione viene generato quando un'installazione viene completata correttamente. L'evento di installazione viene generalmente utilizzato per popolare le funzionalità di memorizzazione nella cache offline del browser con le risorse necessarie per eseguire la tua app offline. Per fare ciò, utilizziamo la nuovissima API di storage di Service Worker: cache- una soluzione globale per l'addetto all'assistenza che ci consente di archiviare le risorse fornite dalle risposte e adattate alle loro richieste. Questa API funziona in modo simile alla cache standard del browser, ma è specifica per il tuo dominio. Persiste finché non lo dici a ... di nuovo, hai il pieno controllo.

Iniziamo questa sezione esaminando un esempio di codice:

```
1  self.addEventListener('install', function(event) {
2    event.waitUntil(
3      caches.open('v1').then(function(cache) {
4        return cache.addAll([
5          '/sw-test/',
6          '/sw-test/index.html',
7          '/sw-test/style.css',
8          '/sw-test/app.js',
9          '/sw-test/image-list.js',
10         '/sw-test/star-wars-logo.jpg',
11         '/sw-test/gallery/',
12         '/sw-test/gallery/bountyHunters.jpg',
13         '/sw-test/gallery/myLittleVader.jpg',
14         '/sw-test/gallery/snowTroopers.jpg'
15       ]);
16     })
17   );
18 });
```

- Qui aggiungiamo un installlistener di eventi al worker del servizio (da qui self) e quindi concateniamo un ExtendableEvent.waitUntil()metodo sull'evento - questo garantisce che l'addetto al servizio non si installi finché il codice interno non si waitUntil()è verificato correttamente.
- All'interno waitUntil()utilizziamo il caches.open()metodo per creare una nuova cache chiamata v1, che sarà la versione 1 della nostra cache delle risorse del sito. Ciò restituisce una promessa per una cache creata; una volta risolti, chiamiamo una funzione che richiama addAll()la cache creata, che per il suo parametro prende una matrice di URL relativi all'origine a tutte le risorse che si desidera memorizzare nella cache.
- Se la promessa viene respinta, l'installazione non riesce e l'operatore non farà nulla. Questo è ok, in quanto è possibile correggere il codice e riprovare la prossima volta che si verifica la registrazione.
- Dopo una corretta installazione, l'operatore di servizio si attiva. Questo non ha un uso distinto la prima volta che il tuo operatore di servizio viene installato / attivato, ma significa di più quando il lavoratore del servizio viene aggiornato

6.3.4 Risposte personalizzate alle richieste

Ora hai memorizzato nella cache le tue risorse del sito, devi dire ai lavoratori del servizio di fare qualcosa con il contenuto della cache. Questo è facilmente fatto con l'evento fetch. Un evento fetch si attiva ogni volta che viene recuperata qualsiasi risorsa controllata da un operatore del servizio, che include i documenti all'interno dell'ambito specificato e tutte le risorse a cui si fa riferimento in tali documenti

Puoi collegare un fetchlistener di eventi all'operatore del servizio, quindi chiamare il respondWith()metodo sull'evento per dirottare le nostre risposte HTTP e aggiornarle. Potremmo iniziare semplicemente rispondendo con la risorsa il cui url corrisponde a quello della richiesta di rete, in ogni caso:

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
  );
});
```

caches.match(event.request)ci consente di abbinare ogni risorsa richiesta dalla rete con la risorsa equivalente disponibile nella cache, se è disponibile una corrispondente. La corrispondenza viene eseguita tramite url e vari header, proprio come con le normali richieste HTTP.

Diamo un'occhiata ad alcune altre opzioni che abbiamo quando dobbiamo modificare le nostre risposte HTTP:

- Il Response()costruttore ti consente di creare una risposta personalizzata. In questo caso, stiamo solo restituendo una semplice stringa di testo:

```
    new Response('Hello from your friendly neighbourhood service worker!');

```

- Un Response() più complesso mostra che puoi opzionalmente passare una serie di intestazioni con la tua risposta, emulando intestazioni di risposta HTTP standard. Qui stiamo solo dicendo al browser qual è il tipo di contenuto della nostra risposta sintetica:

```
    new Response('<p>Hello from your friendly neighbourhood service worker!</p>', {
      headers: { 'Content-Type': 'text/html' }
    });

```

- Se non è stata trovata una corrispondenza nella cache, è possibile indicare al browser semplicemente fetchla richiesta di rete predefinita per tale risorsa, per ottenere la nuova risorsa dalla rete, se disponibile:

```
    fetch(event.request);

```

- Se non è stata trovata una corrispondenza nella cache e la rete non è disponibile, è possibile semplicemente abbinare la richiesta con una sorta di pagina di fallback predefinita come risposta usando match(), come questo:

```
    caches.match('/fallback.html');

```

- È possibile recuperare molte informazioni su ciascuna richiesta chiamando i parametri Requestdell'oggetto restituito da FetchEvent:

```
    event.request.url
    event.request.method
    event.request.headers
    event.request.body

```

6.3.5 Ripristino delle richieste non riuscite

Quindi caches.match(event.request)è grandioso quando c'è una corrispondenza nella cache dei lavoratori del servizio, ma per quanto riguarda i casi in cui non c'è una corrispondenza? Se non avessimo fornito alcun tipo di gestione degli errori, la nostra promessa sarebbe stata risolta undefinede non avremmo ricevuto nulla.

Fortunatamente la struttura basata sulle promesse dei lavoratori rende banale la possibilità di fornire ulteriori opzioni per il successo. Potremmo fare questo:

```
1  self.addEventListener('fetch', function(event) {
2    event.respondWith(
3      caches.match(event.request).then(function(response) {
4        return response || fetch(event.request);
5      })
6    );
7  });
```

Se le risorse non sono nella cache, viene richiesta dalla rete.

Se fossimo davvero intelligenti, non richiederemo solo la risorsa dalla rete; vorremmo anche salvarlo nella cache in modo che anche le richieste successive per quella risorsa possano essere recuperate offline! Ciò significherebbe che se venissero aggiunte immagini extra alla galleria di Star Wars, la nostra app potrebbe automaticamente prenderle e memorizzarle nella cache. Il seguente avrebbe fatto il trucco:

```
1  self.addEventListener('fetch', function(event) {
2    event.respondWith(
3      caches.match(event.request).then(function(resp) {
4        return resp || fetch(event.request).then(function(response) {
5          return caches.open('v1').then(function(cache) {
6            cache.put(event.request, response.clone());
7            return response;
8          });
9        });
10     })
11   );
12 });
```

Qui restituiamo la richiesta di rete predefinita con return fetch(event.request), che restituisce una promessa. Quando questa promessa viene risolta, rispondiamo eseguendo una funzione che utilizza la nostra cache caches.open('v1'); anche questo restituisce una promessa. Quando quella promessa si risolve, cache.put()viene utilizzato per aggiungere la risorsa alla cache. La risorsa viene prelevata event.requeste la risposta viene quindi clonata response.clone()e aggiunta alla cache. Il clone viene messo nella cache e la risposta originale viene restituita al browser per essere data alla pagina che l'ha chiamata.

La clonazione della risposta è necessaria perché i flussi di richiesta e di risposta possono essere letti solo una volta. Per restituire la risposta al browser e inserirla nella cache, dobbiamo clonarla. Quindi l'originale viene restituito al browser e il clone viene inviato alla cache. Ciascuno viene letto una volta.

L'unico problema che abbiamo ora è che se la richiesta non corrisponde a nulla nella cache e la rete non è disponibile, la nostra richiesta continuerà a fallire. Forniamo un fallback di default in modo che qualunque cosa accada, l'utente otterrà almeno qualcosa:

```
1  selfs.addEventListener('fetch', function(event) {
2    event.respondWith(
3      caches.match(event.request).then(function(resp) {
4        return resp || fetch(event.request).then(function(response) {
5        let responseClone = response.clone();
6        caches.open('v1').then(function(cache) {
7          cache.put(event.request, responseClone);
8        });
9
10         return response;
11       });
12     }).catch(function() {
13         return caches.match('/sw-test/gallery/myLittleVader.jpg');
```

```
        })
    );
});
```

Abbiamo optato per questa immagine di fallback perché gli unici aggiornamenti che potrebbero fallire sono le nuove immagini, dato che tutto il resto dipende dall'installazione nel installlistener di eventi che abbiamo visto in precedenza.

6.3.6 Aggiornamento del Service Worker

Se l'addetto all'assistenza è già stato installato, ma una nuova versione dell'operatore è disponibile per l'aggiornamento o il caricamento della pagina, la nuova versione viene installata sullo sfondo, ma non ancora attivata. Si attiva solo quando non ci sono più pagine caricate che stanno ancora utilizzando il vecchio servizio di assistenza. Non appena non ci sono più pagine di questo tipo ancora caricate, il nuovo operatore di servizio si attiva.

Si dovrà aggiornare il listener install di eventi nel nuovo operatore di servizio a qualcosa di simile a questo:

```
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('v2').then(function(cache) {
      return cache.addAll([
        '/sw-test/',
        '/sw-test/index.html',
        '/sw-test/style.css',
        '/sw-test/app.js',
        '/sw-test/image-list.js',

        // include other new resources for the new version...
      ]);
    })
  );
});
```

Mentre ciò accade, la versione precedente è ancora responsabile per i recuperi. La nuova versione si sta installando in background. Stiamo chiamando la nuova cache v2, quindi la v1cache precedente non è disturbata.

Quando nessuna pagina sta utilizzando la versione corrente, il nuovo operatore si attiva e diventa responsabile dei recuperi.

6.3.7 Cancellare vecchie cache

Si ha a disposizione anche un evento activate. Questo è generalmente usato per fare cose che avrebbero rotto la versione precedente mentre era ancora in esecuzione, ad esempio per liberarsi di vecchie cache. Ciò è utile anche per rimuovere i dati che non sono più necessari per evitare di riempire troppo spazio su disco - ogni browser ha un limite rigido alla quantità di memoria cache che un determinato operatore di servizio può utilizzare. Il browser fa del suo meglio per gestire lo spazio su disco, ma può eliminare la memoria cache per un'origine. Il browser eliminerà generalmente tutti i dati per un'origine o nessuno dei dati per un'origine.

Le promesse passate waitUntil()blocceranno altri eventi fino al completamento, quindi puoi essere certo che l'operazione di pulizia sarà completata quando avrai il tuo primo fetchevento sulla nuova cache.

```
  self.addEventListener('activate', function(event) {
    var cacheWhitelist = ['v2'];

    event.waitUntil(
      caches.keys().then(function(keyList) {
        return Promise.all(keyList.map(function(key) {
          if (cacheWhitelist.indexOf(key) === -1) {
            return caches.delete(key);
```

```
          }
        }));
      })
    );
  });
```

6.3.8 Strumenti di sviluppo

Chrome ha `chrome://inspect/#service-workers`, che mostra l'attività corrente del lavoratore di servizio e l'archiviazione su un dispositivo, e `chrome://serviceworker-internals`, che mostra più dettagli e consente di avviare / arrestare / eseguire il debug del processo di lavoro. In futuro avranno modalità di throttling / offline per simulare connessioni cattive o inesistenti, che sarà davvero una buona cosa.

Firefox ha anche iniziato a implementare alcuni strumenti utili relativi ai lavoratori del servizio:

- Puoi navigare per `about:debugging` vedere quali SW sono registrati e aggiornarli / rimuoverli.
- Durante il test è possibile aggirare la restrizione HTTPS selezionando l'opzione "Abilita i lavoratori del servizio su HTTP (quando la cassetta degli attrezzi è aperta)" nelle impostazioni degli Strumenti per sviluppatori di Firefox .
- Il pulsante "Dimentica", disponibile nelle opzioni di personalizzazione di Firefox, può essere usato per cancellare i lavoratori del servizio e le loro ca

7 Demo Service Worker

Questa demo chiamata sw-test, una semplice galleria di immagini, mostra le basi della registrazione e dell'installazione di un Service Worker (codice sorgente: `https://github.com/mdn/sw-test/` , demo: `https://mdn.github.io/sw-test/`).

La demo utilizza una funzione alimentata da promise per leggere i dati di immagine da un oggetto JSON e caricare le immagini utilizzando Ajax, prima di visualizzare le immagini su una riga lungo la pagina. Inoltre memorizzerà nella cache tutti i file necessari in modo che funzionino offline.

- L'unica cosa che chiameremo qui è la promessa

```
function imgLoad(imgJSON) {
  // return a promise for an image loading
  return new Promise(function(resolve, reject) {
    var request = new XMLHttpRequest();
    request.open('GET', imgJSON.url);
    request.responseType = 'blob';
    request.onload = function() {
      if (request.status == 200) {
        var arrayResponse = [];
        arrayResponse[0] = request.response;
        arrayResponse[1] = imgJSON;
        resolve(arrayResponse);
      } else {
        reject(Error('Image didn\'t load successfully; error code:' + request.
statusText));
      }
    };
    request.onerror = function() {
      reject(Error('There was a network error.'));
    };
    // Send the request
    request.send();
  });
}
```

- Passiamo in un frammento JSON contenente tutti i dati per una singola immagine

```
var Path = 'gallery/';

var Gallery = { 'images' : [
  {
    'name'  : 'Darth Vader',
    'alt' : 'A Black Clad warrior lego toy',
    'url': 'gallery/myLittleVader.jpg',
    'credit': '<a href="https://www.flickr.com/photos/legofenris/">legOfenris</a>,
     published under a <a href="https://creativecommons.org/licenses/by-nc-nd/2.0/">
    Attribution-NonCommercial-NoDerivs 2.0 Generic</a> license.'
  },
  {
    'name'  : 'Snow Troopers',
    'alt' : 'Two lego solders in white outfits walking across an icy plain',
    'url': 'gallery/snowTroopers.jpg',
    'credit': '<a href="https://www.flickr.com/photos/legofenris/">legOfenris</a>,
     published under a <a href="https://creativecommons.org/licenses/by-nc-nd/2.0/">
    Attribution-NonCommercial-NoDerivs 2.0 Generic</a> license.'
  },
  {
    'name'  : 'Bounty Hunters',
    'alt' : 'A group of bounty hunters meeting, aliens and humans in costumes.',
    'url': 'gallery/bountyHunters.jpg',
    'credit': '<a href="https://www.flickr.com/photos/legofenris/">legOfenris</a>,
     published under a <a href="https://creativecommons.org/licenses/by-nc-nd/2.0/">
    Attribution-NonCommercial-NoDerivs 2.0 Generic</a> license.'
  },
]};
```

Passiamo un JSON perché tutti gli elementi per ogni primse risolta devono essere passati con la promise in quanto è asincrono. Se è appena stato passato l'url e poi si è provato ad accedere agli altri elementi nel JSON separatamente quando il for() loop è stato ripetuto in seguito, non funzionerebbe, poiché la promessa non si risolverebbe nello stesso momento in cui le iterazioni sono in corso (questo è un processo sincrono).

- In realtà risolviamo la promessa con un array, dato che vogliamo rendere disponibile il blob dell'immagine caricata alla funzione di risoluzione più avanti nel codice, ma anche il nome dell'immagine, i crediti e il testo alternativo.

```
var arrayResponse = [];
arrayResponse[0] = request.response;
arrayResponse[1] = imgJSON;
resolve(arrayResponse);
```

Le promesse si risolvono solo con un singolo argomento, quindi se si vuole risolvere con più valori si deve usare un array/oggetto.

- Per accedere ai valori promessi risolti si accede alla funzione:

```
var imageURL = window.URL.createObjectURL(arrayResponse[0]);
myImage.src = imageURL;
myImage.setAttribute('alt', arrayResponse[1].alt);
myCaption.innerHTML = '<strong>' + arrayResponse[1].name + '</strong>: Taken by '
  + arrayResponse[1].credit;
```

8 Esempio offline

Ecco un riepilogo del progetto che implementato:

- si scarica lo script Service Worker dal server;
- si farà in modo che il browser installi e attivi il Service Worker in background il più tardi possibile per non interrompere l'esperienza utente iniziale;
- in background il Service Worker starà scaricando l'intera applicazione Web (HTML, CSS e Javascript) che conserverà per utilizzarla in un secondo momento;
- la volta successiva che l'utente arriverà al sito il Service Worker userà l'applicazione Web conservata in precedenza; quando questa volta l'utente visiterà il sito l'applicazione non scaricherà HTML, CSS e Javascript dalla rete ma il Service Worker userà i file memorizzati nella cache che aveva conservato per un secondo momento. Di conseguenza, questa seconda volta l'avvio dell'applicazione sarà molto più veloce. Inoltre l'utente avrà l'applicazione funzionante anche se la rete non funziona

Questo meccanismo è come avere un proxy di rete nel browser che ci permette di avere applicazioni web installabili.

8.1 Registrazione del Service Worker

Il punto di partenza è una pagina Bootstrap HTML, CSS e Javascript che utilizzava alcuni bundle CSS e Javascript molto comuni; la trasformeremo in un PWA scaricabile e installabile e lo stesso ragionamento si applica a un'applicazione a singola pagina.

Il primo passo per trasformare questo sito Web standard in un PWA scaricabile è aggiungere un Service Worker tramite uno script di registrazione:

```
<!-- commonly used JS bundles -->
<script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" ></script>
<script src="http://getbootstrap.com/dist/js/bootstrap.min.js"></script>
...

<!-- register the Service Worker -->
<script src="sw-register.js"></script>
```

Lo script di registrazione:

```
/*stiamo controllando se il browser supporta i lavoratori del servizio, cercando la
  serviceWorkerproprieta navigatornell'oggetto globale. se il browser non supporta i SW
  , allora tutto funzionera ancora, e solo che nessuna installazione verra eseguita in
  background, quindi eseguiamo il fallback su uno scenario di applicazione Web normale
  */

if ('serviceWorker' in navigator) {
  window.addEventListener('load',  () => {
    navigator.serviceWorker.register('/sw.js', {
      scope: '/'
    })
    .then(registration => {
      console.log("Service Worker registration completed ...");
    });
  });
}
```

Perchè ritardare la registrazione del Service Worker

Anche se rileviamo che il browser supporta i Service Worker non registreremo i Service Worker immediatamente: in questo caso stiamo aspettando l'event load della pagina che viene attivato solo quando viene caricata l'intera pagina, comprese le risorse collegate come immagini, CSS e Javascript e ciò può richiedere molto tempo. Nel caso di un'applicazione a singola pagina, potremmo voler ritardare ulteriormente la registrazione e attendere oltre l'evento load.

I due motivi per cui vogliamo ritardare la registrazione del Service Worker sono:

- evitare di degradare o interrompere l'esperienza utente iniziale poiché l'applicazione viene caricata per la prima volta.

  Poichè i browser eseguono solo una quantità limitata di richieste HTTP allo stesso tempo, e la capacità della rete è anch'essa limitata, il Service Worker potrebbe o non potrebbe fare richieste di rete separate che possono interferire con quelle necessarie per mostrare all'utente il contenuto iniziale.

  Quindi per favorire l'esperienza utente il Service Worker aspetterà che l'applicazione si avvii e verrà installata in background.

- avere un comportamento coerente dell'applicazione.

  Quindi vogliamo evitare una situazione in cui:

  - alcune delle risorse CSS e JS della pagina sono state fornite dal Service Worker;
  - mentre altri provenivano dalla rete.

  Se alcune delle richieste iniziali di una pagina provengono dalla rete, vogliamo essere sicuri che anche tutti i pacchetti rimanenti siano stati caricati dalla rete, per coerenza.

  Nel caso del download e dell'installazione dell'applicazione, vogliamo evitare di cadere in una situazione in cui attiviamo un Service Worker nel mezzo dell'avvio di una pagina in base a quanto etto prima sull'esperienza utente.

  La prossima volta che visiteremo la pagina il Service Worker sarà attivo quindi caricheremo tutte le risorse dal Service Worker anziché dalla rete, ciò significa che avremo un insieme coerente di pacchetti, tutti provenienti da una cache e corrispondenti a una determinata versione dell'applicazione.

  E possibile avere service worker multipli nella stessa pagina (Service Worker ID), ciò significa che è possibile avere più Service Worker in esecuzione sulla stessa pagina, ma su diversi ambiti, quindi potremmo registrare diversi Service Worker per diversi ambiti.

8.2 Installazione del Service Worker

Quando il browser identifica una nuova versione di Service Worker per un dato ambito attiverà la fase di installazione, successivamente quella di attivazione: ora l'intercettazione della rete è pronta per essere utilizzata.

Col seguente codice, che è l'implementazione di un semplice intercettore HTTP di registrazione, cerchiamo di spiegare come funzionano le fasi di installazione e attivazione, e lo svilupperemo per implementare il download e l'installazione dell'applicazione.

```
/*SPIEGARE BENE STA PARTE*/

  const VERSION = 'v1';
  /*stiamo usando un riferimento a self: questo significa il contesto globale corrente in
     cui viene eseguito il codice, che sarebbe ad esempio il windowse questo dovesse
    essere eseguito a livello dell'applicazione
  Tuttavia, in questo caso, selfpunta al contesto globale di Service Worker.
  Ci stiamo iscrivendo a installed activateeventi e registriamo la loro presenza sulla
    console
  ogni dichiarazione di registrazione e preceduta dalla versione di Service Worker,
    questo ci aiutera a capire come funzionano piu versioni.
  le fasi di installazione e attivazione passano entrambe una Promessa waitUntil(),
    proprio ora questo e solo per mostrare come faremmo operazioni asincrone in queste
    fasi
  se la promessa e passata a waitUntil()risolversi con successo, allora la fase di
    installazione / attivazione e stata completata con successo
  se invece la promessa viene respinta, la fase di installazione / attivazione fallisce e
     la fase successiva non verra attivata.
  ci siamo anche iscritti fetchall'evento. Usandolo, stiamo intercettando tutte le
    richieste HTTP fatte dall'applicazione
  L'evento fetch ha un metodo chiamato respondWith(), che accetta come argomento anche
    una promessa
  La promessa che passiamo ha bisogno di restituire (quando risolto) la risposta alla
    richiesta HTTP.
  */

  self.addEventListener('install', event => {
    log("INSTALLING ");
    const installCompleted = Promise.resolve()
    .then(() => log("INSTALLED"));
    event.waitUntil(installCompleted);
  });

  self.addEventListener('activate', event => {
    log("ACTIVATING");
    const activationCompleted = Promise.resolve()
    .then((activationCompleted) => log("ACTIVATED"));

    event.waitUntil(activationCompleted);
  });

  // handling service worker installation
  self.addEventListener('fetch', event => {
    log("HTTP call intercepted - " + event.request.url);
    return event.respondWith(fetch(event.request.url));
  });


  // each logging line will be prepended with the service worker version
  function log(message) {
    console.log(VERSION, message);
  }
```

8.2.1 Fetch per intercettare le richieste HTTP

Diamo ora uno sguardo più da vicino al callback fetch dell'evento, che contiene la funzionalità di registrazione HTTP.

Come possiamo vedere, questo fetch callback restituirà la risposta effettiva della chiamata HTTP respondWith() e la risposta può essere calcolata in modo asincrono passando una promise a respondWith().

Nota: il codice dell'applicazione non è a conoscenza di dove proviene questa risposta: se dalla rete o dal Service Worker

Possiamo prendere la risposta passata respondWith()da qualsiasi luogo, ad esempio:

- possiamo inoltrare la chiamata alla rete e rispedire la risposta della rete

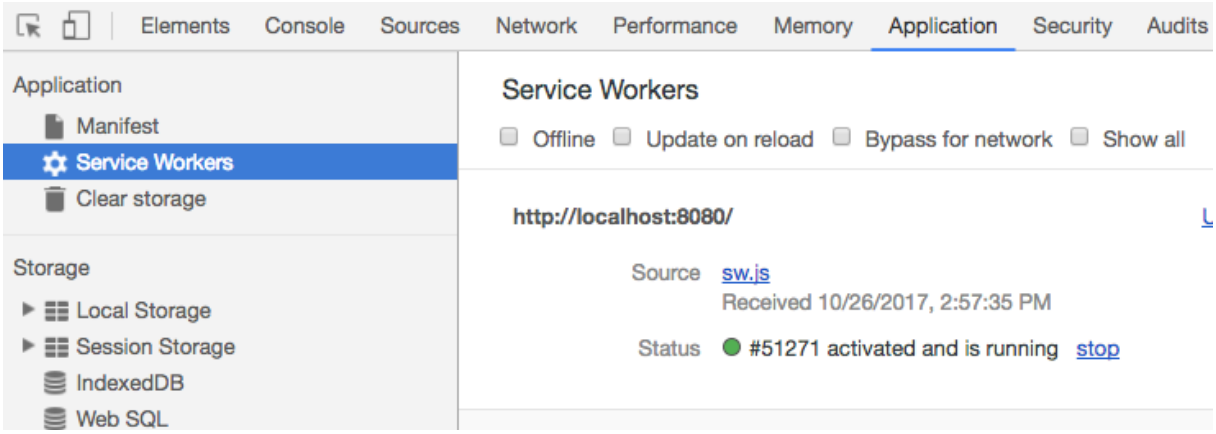

Figura 6: Service Worker in esecuzione in Chrome Dev Tools

- oppure possiamo recuperare la risposta da Cache Storage possiamo anche costruire un Response()oggetto manualmente

In questo caso, ecco cosa stiamo facendo:

- stiamo registrando l'URL della richiesta intercettata
- quindi inoltriamo la richiesta HTTP alla rete utilizzando l'API Fetch
- fetch() restituirà una Promessa, che se risolta consegnerà la risposta di rete, o fallirà in caso di un errore di rete fatale
- si noti che fetch()verrà generato un errore solo se la rete è inattiva o se si verifica qualche altra condizione irreversibile come un errore DNS. Ad esempio, un codice di stato HTTP di 500 Errore interno server non causerebbe la promessa di errore di recupero
- quindi passiamo la fetch()promessa che emetterà la risposta della rete a respondWith()

Questa risposta passata respondWith()verrà quindi passata all'applicazione! Come possiamo vedere, questo Service Worker funge da proxy di registrazione.

Dal punto di vista dell'applicazione, questa risposta fornita dal Service Worker è indistinguibile da una chiamata effettuata se il Service Worker non era presente, l'unico effetto collaterale è la registrazione nella console.

Esaminiamo quindi l'output della console:

```
1  v1 INSTALLING
2  v1 INSTALLED
3  v1 ACTIVATING
4  v1 ACTIVATED
5  Service Worker registration completed ...
```

Ed ecco il nostro Service Worker in esecuzione in Chrome Dev Tools: Nota: sebbene stiamo registrando gli eventi di installazione e attivazione ma non è stata registrata alcuna richiesta HTTP sulla console, quindi sembrerebbe che l'evento fetch non sembri funzionare, anche se il Service Worker è attivo .

Ma se apriamo un'altra scheda o aggiorniamo la stessa scheda, ecco cosa abbiamo:

```
1  v1 HTTP call intercepted - getbootstrap.com/dist/css/bootstrap.min.css
2  v1 HTTP call intercepted - localhost:8080/carousel.css
3  v1 HTTP call intercepted - code.jquery.com/jquery-3.2.1.slim.min.js
4  v1 HTTP call intercepted - getbootstrap.com/js/vendor/popper.min.js
5  v1 HTTP call intercepted - getbootstrap.com/dist/js/bootstrap.min.js
6  ... other intercepted CSS/Js bundles
7  v1 HTTP call intercepted - localhost:8080/sw-register.js
8  Service Worker registration completed ...
```

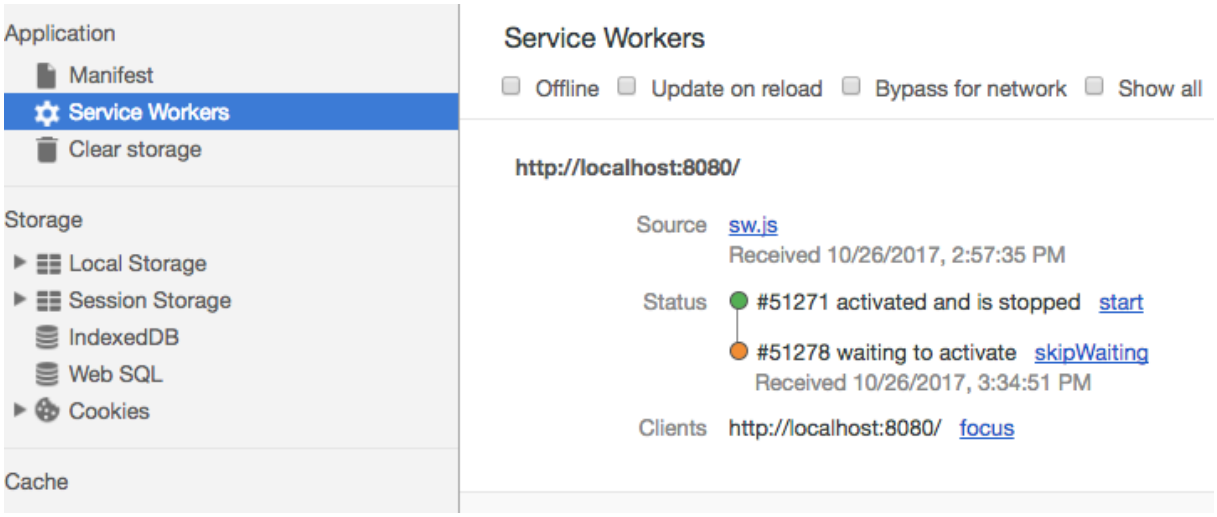

Figura 7: Nuova versione Service Worker in esecuzione in Chrome Dev Tools

Quindi il Service Worker ha iniziato a intercettare le richieste HTTP solo dopo aver ricaricato la pagina e questo accade di default per garantire coerenza.

Simulando il comportamento normale dell'utente, cosa succede se apriamo altre schede del browser della stessa applicazione?

```
v1 HTTP call intercepted - getbootstrap.com/dist/css/bootstrap.min.css
... the same HTTP requests, all served by version 1
Service Worker registration completed ...
```

Vedremo che questa pagina viene servita dallo stesso SW v1! Nota: la registrazione della console è condivisa tra le schede.

Se si aggiorna l'applicazione un paio di volte e poi si torna ad un'altra scheda, si vedranno le richieste HTTP registrate che sono state fatte nell'altra scheda. Questo perchè abbiamo lo stesso Service Worker che intercetta le richieste da tutte le schede.

Per capire meglio il ciclo di vita del lavoratore del servizio, vediamo ora cosa succede se modifichiamo qualcosa nel codice del lavoratore del servizio. Ad esempio, modifica il numero di versione in v2.

Si noti che non è necessario modificare il nome del file sw.jsper notificare al browser che è disponibile una nuova versione dell'operatore del servizio.

Il browser vedrà che entrambe le versioni sono collegate all'ambito /e se c'è anche un solo carattere di differenza tra le due versioni, il browser installerà la nuova versione.

Proviamo quindi a installare v2, ancora con più schede aperte. Se cambiamo il numero di versione dello script SW in v2 e apriamo un'altra scheda, ecco cosa vediamo in Dev Tools: Come possiamo vedere, la nuova versione di Service Worker non viene immediatamente applicata, è in una sorta di stato di attesa!

E se guardiamo la console, ora abbiamo:

```
  v1 HTTP call intercepted - getbootstrap.com/dist/css/bootstrap.min.css
  v1 HTTP call intercepted - localhost:8080/carousel.css
  ... the same requests as before still being intercepted by v1
  Service Worker registration completed ...
  v2 INSTALLING
  v2 INSTALLED
```

Ci sono un paio di cose che sono molto interessanti in questo registro:

- la versione v1 non è stata ancora installata, o addirittura attivata
- sembra che la versione v1 sia rimasta attiva durante l'intero processo di aggiornamento, perché ha mantenuto l'intercettazione delle richieste HTTP

- tutte le richieste sono ancora intercettate da v1
- La versione v2 è stata installata in background, ma non attivata!
- La versione v2 è ora in attesa

Allora perchè la nuova versione v2 è installata ma non attivata? Una ragione è che abbiamo aperto più schede e vogliamo mostrare all'utente un'esperienza coerente. Sarebbe difficile per l'utente avere due schede aperte che eseguono versioni diverse della stessa applicazione. E poiché i lavoratori del Servizio intercettano e modificano le richieste HTTP, due diverse versioni del lavoratore del servizio potrebbero significare due diverse versioni dell'applicazione stessa!

Quindi in che modo il browser gestirà questa nuova versione del service worker?

Il browser sta per andare avanti e svolgere alcuna installazione operazioni come download o una pagina offline in fase di installazione di v2, ma il browser non attiva v2 finché ci sono più schede aperte ancora in esecuzione v1.

Cerchiamo di capire perché in questa fase con v2 già installato, perché v1 sia ancora in esecuzione, e perché v2 non è ancora attivo.

Abbiamo aggiornato la nostra scheda singola eseguendo v1, ma ancora v2 non è stato attivato (è stato installato in background, ma non attivato). Questo perché, dal punto di vista del browser, la pagina corrente rimane attiva fino al completamento dell'aggiornamento, e solo allora la pagina viene scambiata quando abbiamo almeno ricevuto le intestazioni di risposta dal server. Poiché la pagina è stata mantenuta durante una parte del processo di aggiornamento, l'unico modo per garantire la coerenza è mantenerlo attivo per tutto il processo. Dopodiché, poiché abbiamo mantenuto attivo il Service worker v1 durante l'aggiornamento, per impostazione predefinita vogliamo mantenerlo in esecuzione anche dopo il completamento dell'aggiornamento, il che spiega perché V1 è ancora attivo dopo il completamento dell'aggiornamento della pagina.

Per attivare la nuova versione di Service Worker V2 un modo potrebbe essere quello di riprodurre la normale esperienza utente: chiudiamo tutte le schede che eseguono il service worker v1 e apriamo una nuova scheda.

Se guardiamo all'output della console, ora abbiamo:

```
1 v2 ACTIVATING
2 v2 ACTIVATED
3 v2 HTTP call intercepted - localhost:8080
4 v2 HTTP call intercepted - getbootstrap.com/dist/css/bootstrap.min.css
5 ... the same list of requests, all intercepted by v2
```

Come possiamo vedere, questa volta il browser ha attivato Service Worker v2 che aveva precedentemente installato in background e v2 ha intercettato tutte le richieste di rete da questa pagina, il che significa che V2 è ora attivo.

8.3 Riepilogo del ciclo di vita del Service Worker

Possiamo vedere che, anche se un po 'complicato a prima vista, il modo in cui funziona il ciclo di vita del Service Worker ha molto senso. Il ciclo di vita è tutto su:

- mostra solo una versione dell'applicazione per l'utente
- non disturbare l'esperienza dell'utente
- non ritardare l'avvio dell'applicazione
- per impostazione predefinita, evitando errori di versione tra la pagina e il Service Worker (Quest'ultimo punto è particolarmente importante per il caso di utilizzo di download e installazione che stiamo per esaminare.)

Ricordiamo che uno dei casi di utilizzo comune di Service Workers è quello di memorizzare nella cache l' intera applicazione , vale a dire letteralmente tutto l'HTML, CSS e Javascript!

8.4 Cache Storage API

`https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage` Tutti questi file sono depositati dal Service Worker nella cache storage API. Al momento dell'installazione, Service Worker preleverà dalla rete tutti i bundle che insieme costituiscono una determinata versione dell'applicazione, quindi li memorizzerà nella cache del browser nota come Cache Storage .

Come l'API di Service Worker, anche Cache Storage è basato su Promise e molto facile da usare. Prendiamo quindi questa API e usiamola per implementare la fase di installazione del caso d'uso Download and Install.

8.4.1 Implementazione del download dell'applicazione in background

Iniziamo quindi ad adattare l'esempio dell'intercettore di registrazione mondiale Hello e ad estenderlo con le funzionalità di installazione in background.

La prima cosa che faremo è scaricare tutti i file Javascript e CSS in background durante la fase di installazione e aggiungeremo tali file direttamente a Cache Storage:

```
1  const VERSION = 'v3';
2
3  self.addEventListener('install', event => event.waitUntil(installServiceWorker()));
4  async function installServiceWorker() {
5    log("Service Worker installation started ");
6    const cache = await caches.open(getCacheName());
7    return cache.addAll([
8      '/',
9      'carousel.css',
10     'http://getbootstrap.com/dist/css/bootstrap.min.css',
11     'https://code.jquery.com/jquery-3.2.1.slim.min.js',
12     'http://getbootstrap.com/assets/js/vendor/popper.min.js',
13     'http://getbootstrap.com/dist/js/bootstrap.min.js',
14     'http://getbootstrap.com/assets/js/vendor/holder.min.js'
15   ]);
16 }
```

Di nuovo in questo esempio sta succedendo molto, quindi scomporlo passo dopo passo:

- la prima cosa che stiamo facendo è ottenere un riferimento a una cache aperta, utilizzando la caches.open()quale restituisce una Promessa
- stiamo aggiungendo un numero di versione al nome della cache, il che significa che quando verranno rilasciate nuove versioni, verranno create nuove cache
- Quindi stiamo facendo una serie di richieste HTTP per recuperare tutti i file che rendono una determinata versione dell'applicazione
- Stiamo quindi aggiungendo tutti questi file direttamente alla memoria cache
- la chiave della cache è l'oggetto Request utilizzato per effettuare la richiesta HTTP
- i valori memorizzati nella cache sono gli oggetti di risposta HTTP stessi, che possiamo servire direttamente all'applicazione
- la addAll()chiamata restituisce una Promessa, che verrà risolta correttamente se tutte le richieste HTTP effettuate per caricare ciascun file funzionano

Nel nostro caso, il download di tutti i file ha funzionato, il che significa che la fase di installazione è terminata con successo! Vediamo ora cosa abbiamo memorizzato in Cache Storage, utilizzando gli strumenti di sviluppo di Chrome: Come possiamo vedere, tutti i pacchetti di applicazioni sono stati scaricati in background e l'applicazione è pronta per essere servita dalla cache!

Ma prima di farlo, andiamo avanti e prima cancelliamo tutte le versioni precedenti dell'applicazione da Cache Storage.

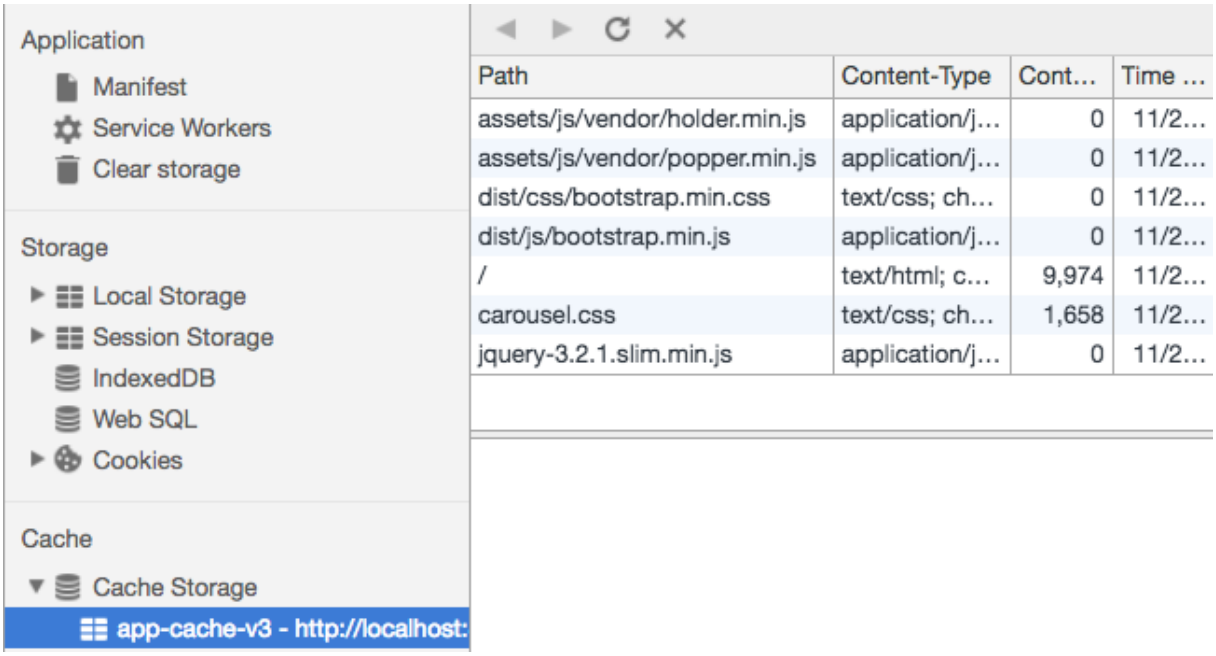

Figura 8: Cache Storage

8.4.2 Eliminazione delle versioni precedenti dell'applicazione

Il momento migliore per eliminare le versioni precedenti dell'applicazione è il momento di attivazione di Service Worker, perché questo è l'unico momento in cui possiamo essere sicuri che l'utente non stia più utilizzando la versione dell'applicazione precedente in nessuna delle schede del browser.

In questo modo possiamo eliminare le versioni precedenti dell'applicazione al momento di attivazione:

```
1  self.addEventListener('activate', () => activateSW());
2  async function activateSW() {
3    log('Service Worker activated');
4    const cacheKeys = await caches.keys();
5    cacheKeys.forEach(cacheKey => {
6      if (cacheKey !== getCacheName() ) {
7        caches.delete(cacheKey);
8      }
9    });
10 }
```

Come possiamo vedere, stiamo collegando tutti i nomi di cache disponibili in Cache Storage e cancellando tutte le cache che non corrispondono alla versione dell'applicazione corrente (che è V3).

8.4.3 Elaborare l'applicazione dalla cache con una Cache Then Network Strategy

L'ultimo passaggio necessario per l'implementazione del download e dell'installazione dell'applicazione consiste nel servire direttamente i bundle dell'applicazione da Cache Storage e, se necessario, effettuare il fallback sulla rete:

```
1  self.addEventListener('fetch', event => event.respondWith(cacheThenNetwork(event)));
2  async function cacheThenNetwork(event) {
3    const cache = await caches.open(getCacheName());
4    const cachedResponse = await cache.match(event.request);
5    if (cachedResponse) {
6      log('Serving From Cache: ' + event.request.url);
7      return cachedResponse;
8    }
9    const networkResponse = await fetch(event.request);
10   log('Calling network: ' + event.request.url);
```

```
11      return networkResponse;
12  }
```

Analizziamo questo esempio per vedere come viene applicata la strategia Cache Then Network:

- stiamo intercettando tutte le chiamate HTTP fatte dall'applicazione, all'interno di una funzione asincrona
- la funzione asincrona restituirà sempre una Promessa a respondWith(), esplicitamente come valore di ritorno, o avvolgendo in modo trasparente il valore restituito in una Promessa
- all'interno della funzione asincrona, iniziamo aprendo la cache che corrisponde alla versione dell'applicazione corrente
- stiamo quindi andando a interrogare la cache, per vedere se c'è una risposta HTTP che corrisponda alla richiesta HTTP fatta dall'applicazione
- la chiamata a match()restituisce anche una promessa, quindi aspetteremo il risultato prima di continuare
- se è stata trovata una corrispondenza, significa che la richiesta fatta dall'applicazione è stata trovata nella cache, quindi restituiamo direttamente la risposta HTTP respondWith()
- si noti che non è necessario restituire una Promessa dal metodo asincrono, se restituiamo un valore esso verrà implicitamente avvolto in una Promessa dal meccanismo asincrono / attesa
- se non viene trovata alcuna corrispondenza, lasceremo passare la richiesta alla rete attendendo il risultato di una fetch()chiamata
- quindi registreremo la richiesta che è stata inoltrata alla rete e restituiremo il risultato della fetch()chiamata all'applicazione

Con questa soluzione, qualsiasi richiesta effettuata dall'applicazione per caricare i pacchetti memorizzati nella cache verrà fornita da Cache Storage, mentre altre richieste, come ad esempio una chiamata API REST /api/courses, continueranno a passare alla rete.

E con questo ultimo passaggio, abbiamo una soluzione completa per il download e l'installazione in background della nostra applicazione web! Quindi proviamolo.

8.4.4 Nuova versione dell'applicazione

Per vedere il meccanismo di download e installazione in azione, apriamo una nuova scheda nella nostra applicazione di esempio e vediamo che ora sta eseguendo la versione V3 del Service Worker, che implementa la funzionalità di download e installazione.

```
1
2
3
4   const VERSION = 'v3';
5
6
7   self.addEventListener('install', event => event.waitUntil(installServiceWorker()));
8
9
10  async function installServiceWorker() {
11
12  log("Service Worker installation started ");
13
14  const cache = await caches.open(getCacheName());
15
16  return cache.addAll([
17  '/',
18  'carousel.css',
19  'http://getbootstrap.com/dist/css/bootstrap.min.css',
```

```
'https://code.jquery.com/jquery-3.2.1.slim.min.js',
'http://getbootstrap.com/assets/js/vendor/popper.min.js',
'http://getbootstrap.com/dist/js/bootstrap.min.js',
'http://getbootstrap.com/assets/js/vendor/holder.min.js'
]);
}

self.addEventListener('activate', () => activateSW());


async function activateSW() {

log('Service Worker activated');

const cacheKeys = await caches.keys();

cacheKeys.forEach(cacheKey => {
if (cacheKey !== getCacheName() ) {
caches.delete(cacheKey);
}
});

}


self.addEventListener('fetch', event => event.respondWith(cacheThenNetwork(event)));



async function cacheThenNetwork(event) {

const cache = await caches.open(getCacheName());

const cachedResponse = await cache.match(event.request);

if (cachedResponse) {
log('Serving From Cache: ' + event.request.url);
return cachedResponse;
}

const networkResponse = await fetch(event.request);

log('Calling network: ' + event.request.url);

return networkResponse;


}







function getCacheName() {
return "app-cache-" + VERSION;
}


function log(message, ...data) {
if (data.length > 0) {
console.log(VERSION, message, data);
}
else {
console.log(VERSION, message);
}
}
```

Ed ecco l'attuale output della console:

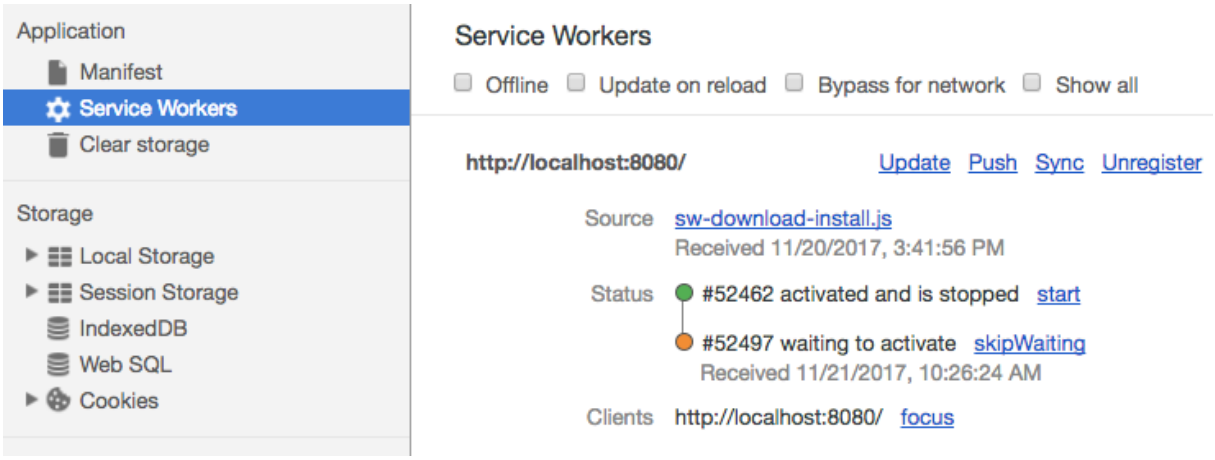

Figura 9: v4 in attesa di essere attivata

```
v3 Serving From Cache: bootstrap.min.css
v3 Serving From Cache: carousel.css
v3 Serving From Cache: jquery-3.2.1.slim.min.js
v3 Serving From Cache: popper.min.js
v3 Serving From Cache: bootstrap.min.js
...
```

Come possiamo vedere, tutti i pacchetti CSS e Javascript provengono direttamente da Cache Storage e non dalla rete, come previsto. Cosa succederebbe ora se avessimo una nuova versione dell'applicazione?

Immagina di aver apportato un sacco di modifiche all'applicazione, come cambiare il suo design o applicare un nuovo tema.

In che modo l'utente otterrà la nuova versione v4, se la versione v3 viene ancora servita ogni volta direttamente dalla cache?

Per attivare l'installazione della versione V4, la prima cosa che dobbiamo fare è anche fare una piccola modifica al Service Worker, ad esempio incrementando il numero di versione:

```
const VERSION = 'v4';
```

Ora chiudiamo tutte le schede tranne una e aggiorna il browser. Ecco cosa avremmo sulla console:

```
v3 Serving From Cache: bootstrap.min.css
v3 Serving From Cache: carousel.css
....
v4 Service Worker installation started
```

Come possiamo vedere, V3 del Service Worker (e dell'applicazione) è ancora attivo e funzionante, come previsto. Ciò significa che la versione dell'applicazione è stata servita da Service Worker v3, il che significa che i bundle provengono tutti dalla cache denominata app-cache-v3.

Ma possiamo vedere anche che la versione V4 è stata installata in background. Diamo un'occhiata a ciò che abbiamo nella scheda Service Worker: Come possiamo vedere, la versione V4 è in attesa di essere attivata. Ma i bundle di V4, che potrebbero corrispondere a una versione completamente diversa dell'intera applicazione Web, sono ora pronti per essere utilizzati.

Per confermare questo, diamo un'occhiata al contenuto di Cache Storage: Come possiamo vedere, Cache Storage contiene due versioni dell'applicazione in questa fase:

- versione v3, che viene ancora offerta all'utente

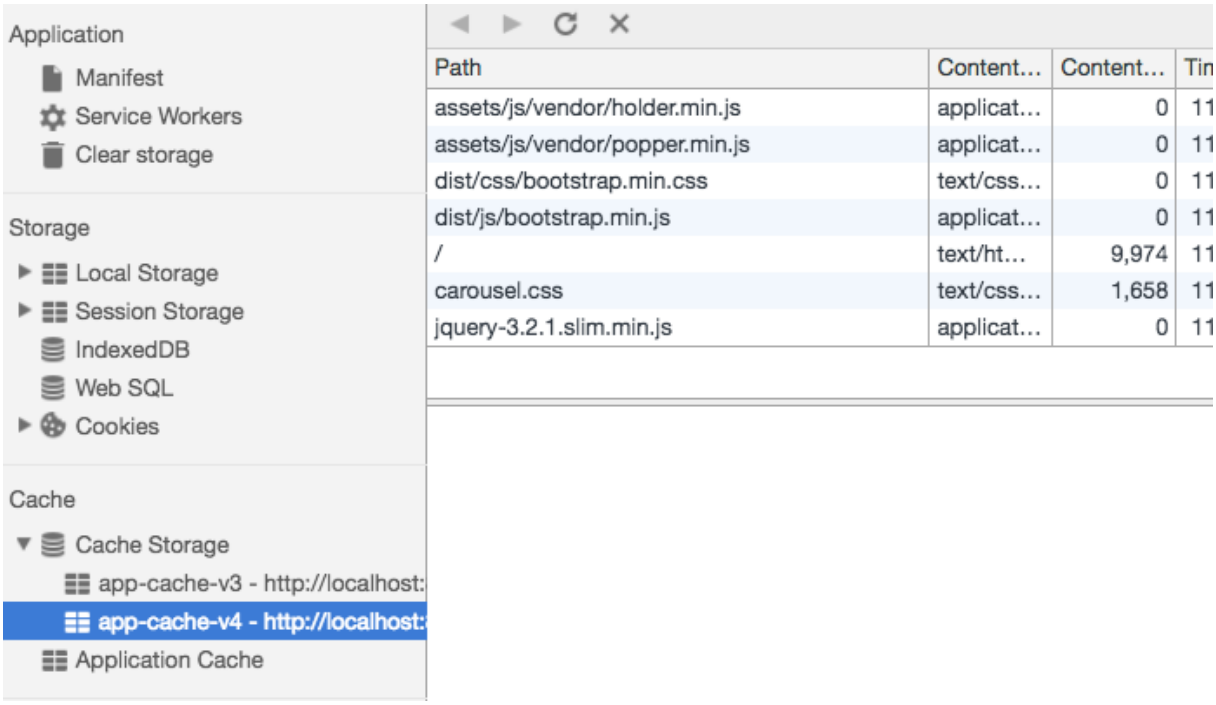

Figura 10: Contenuto Cache Storage

- versione v4, che è stata scaricata in background ed è pronta per essere utilizzata non appena tutte le schede della versione v3 sono chiuse

Per attivare la versione v4, simuliamo alcune normali interazioni con l'utente. L'utente alla fine chiuderebbe tutte le schede del browser con la versione v3, quindi tornerà in seguito all'applicazione.

In quel momento, il browser attiverà la versione V4 e servirà i file corrispondenti dalla cache:

```
v4 Service Worker activated
v4 Serving From Cache: bootstrap.min.css
v4 Serving From Cache: carousel.css
```

E con questo, l'intero ciclo di vita è completato e l'utente ha ora una versione aggiornata dell'applicazione scaricata e installata nel browser.

La nuova versione dell'applicazione è stata scaricata e installata in background, senza interferire con la normale esperienza utente. Questo è in realtà ancora meglio delle installazioni mobili native!

8.4.5 Personalizzazione del comportamento del ciclo di vita del service worker

Quello che abbiamo descritto finora è stato il comportamento predefinito del ciclo di vita del Service Worker, che ha molto senso nel contesto del caso di utilizzo di Donwload e installazione.

Vediamo ora come possiamo personalizzare il ciclo di vita, se necessario , per adattarlo meglio agli altri casi di utilizzo PWA.

Si noti che la modifica del comportamento del ciclo di vita del Service Worker sebbene sia allettante, non è realmente raccomandata, come vedremo.

Saltare la fase di attesa (e potenziali problemi che potrebbe causare)

Ad esempio, potremmo saltare completamente la fase di attesa del ciclo di vita del lavoratore del servizio, chiamando l' skipWaiting()API alla fine della fase di installazione:

```
  async function installServiceWorker() {

  log("Service Worker installation started ");

  const cache = await caches.open(getCacheName());
```

```
6
7   await cache.addAll([
8   '/',
9   'carousel.css',
10  'http://getbootstrap.com/dist/css/bootstrap.min.css',
11  'https://code.jquery.com/jquery-3.2.1.slim.min.js',
12  'http://getbootstrap.com/assets/js/vendor/popper.min.js',
13  'http://getbootstrap.com/dist/js/bootstrap.min.js',
14  'http://getbootstrap.com/assets/js/vendor/holder.min.js'
15  ]);
16
17  return self.skipWaiting();
18  }
```

In questo esempio, siamo in attesa che i file vengano scaricati e installati, e quindi chiameremo self.skipWaiting(), che restituirà una Promessa.

Ciò causerà il saltare della fase in attesa del ciclo di vita e l'attivazione immediata della nuova versione di Service Worker.

Ciò significa che se l'utente apre una nuova scheda, la nuova versione sarebbe attiva e potrebbe causare incoerenze tra le schede. Nella maggior parte dei casi, è meglio non saltare la fase di attesa ed evitare quegli scenari inconsistenti di progettazione.

Ciò non significa, tuttavia, che l'utilizzo skipWaiting()della nuova versione di Service Worker possa intercettare immediatamente le richieste dalla scheda in esecuzione.

Accogliendo la pagina corrente con clients.claim() Abbiamo visto ad esempio che la prima volta che viene caricata una pagina con un Service Worker, il Service Worker verrà installato e attivato , ma in qualche modo non sarà in grado ancora di intercettare le richieste di rete effettuate dalla pagina.

Dovremmo aggiornare la pagina in modo che il nuovo Service Worker avvii le richieste di intercettazione.

Ancora una volta, questo è per coerenza: se le richieste iniziali di una pagina non sono state fornite da un Service Worker, per impostazione predefinita nessuna delle richieste HTTP effettuate da quella pagina dopo l'avvio verrà servita anche dal Service Worker.

Ma possiamo cambiare questo, facendo in modo che il Service Worker richieda tutte le schede dell'applicazione attive al momento di attivazione:

```
1   async function activateSW() {
2
3   log('Service Worker activated');
4
5   const cacheKeys = await caches.keys();
6
7   cacheKeys.forEach(cacheKey => {
8   if (cacheKey !== getCacheName() ) {
9   caches.delete(cacheKey);
10  }
11  });
12
13  return self.clients.claim();
14  }
```

La chiamata claim()consentirà al servizio di lavoro attivato di avviare immediatamente le richieste di intercettazione (incluso Ajax) dalla pagina in esecuzione (così come altre schede aperte), senza dover attendere una ricarica.

Questa attivazione anticipata del Service Worker comporta il rischio di un'incoerenza: potremmo finire con una pagina servita dalla versione v4 per avere le richieste HTTP di runtime intercettate da Service Worker v5.

Ma per alcuni casi d'uso, questa attivazione anticipata è ciò di cui abbiamo bisogno: immagina un secondo operatore in servizio su scope /apiche memorizza nella cache i dati dell'applicazione su IndexedDB: potremmo voler attivarlo il prima possibile, per memorizzare i dati dell'applicazione nel minor tempo possibile.

Aggiornamento manuale di un service worker Per impostazione predefinita, il browser controllerà la navigazione dell'utente se è disponibile una nuova versione di Service Worker sul server pronta per essere installata.

Se per qualche ragione, abbiamo un'applicazione che rimarrà aperta per un lungo periodo di tempo (come un PWA installato nella schermata Home dell'utente), possiamo controllare manualmente se c'è una nuova versione del Service Worker usando il oggetto di registrazione come questo:

```
navigator.serviceWorker.register('/sw-download-install.js', {
scope: '/'
})
.then(registration => {

console.log("Service Worker registration completed ...");

// periodically check (each hour) if there is a new version of the Service Worker
setInterval(() => {

registration.update();

}, 3600000);

});
```

Se una nuova versione dell'operatore del servizio è disponibile sul server, la chiamata update()attiva attiverà una nuova installazione in background.

Questo controllo periodico di solito non è necessario, poiché il browser effettuerà questo controllo molto spesso con la navigazione di ciascun utente o con altri eventi, come ad esempio se viene ricevuta una notifica Push.

Un buon scenario quando vorremmo verificare se c'è una nuova versione è: cosa succede se la versione che stiamo eseguendo ha un bug? Parliamo poi di cosa succede se qualcosa va storto con l'applicazione.

8.4.6 Protezione del browser integrata contro i Service Worker guasti

Come puoi immaginare, la memorizzazione nella cache dell'applicazione sul computer dell'utente e l'aggiramento della rete sono un po 'pericolosi: cosa accadrebbe se la versione scaricata accidentalmente dall'utente avesse un errore?

Ci sono un paio di protezioni del browser integrate contro questo.

Ad esempio, il Service Worker non si intercetterà mai da solo!

Significa che il file sw.jsche passiamo serviceWorker.register('sw.js')non sarà mai intercettato da un fetchevento.

Tuttavia, questo non si applica allo script di registrazione di Service Worker sw-register.js, quindi è necessario assicurarsi che non lo memorizziamo mai.

Operatori di servizio e normale memorizzazione nella cache del browser Il meccanismo di cache del browser standard basato sull'intestazione Cache-Controlè molto facile da utilizzare in modo improprio, a causa della natura confusa delle sue opzioni di configurazione.

Per evitare tali problemi, è consigliabile familiarizzarsi con alcune pratiche ottimali di memorizzazione nella cache comuni , poiché ciò sarà di aiuto con qualsiasi applicazione in generale, non solo con le PWA.

Gli errori commessi nella creazione di Cache-Controlintestazioni per la nostra applicazione saranno problematici nella produzione anche se non eseguiamo un PWA, ma l'utilizzo di un Service Worker renderà questi problemi molto peggiori.

Potremmo imbattersi in una situazione in cui abbiamo memorizzato nella cache il sw.jsfile di Service Worker nella cache del browser standard, perché è stato fornito con Cache-Controlun'intestazione che conferisce al file una lunga durata.

Diciamo che è sw.jsstato servito con una durata della cache di un mese:

```
Cache-Control: max-age=2592000
```

Il browser in effetti memorizzerà l'intestazione nella cache, ma poiché il file è un Service Worker lo memorizzerà nella cache solo per un tempo massimo di 24 ore invece di 1 mese!

Questa è una grande precauzione, tuttavia il sito Web potrebbe essere interrotto per un giorno intero prima che una patch possa essere installata. La soluzione più semplice e sicura è non memorizzare mai nel cache il file worker del servizio o il suo script di registrazione.

Evitare di memorizzare nella cache il file Service Worker Questo può essere garantito dal server contrassegnando esplicitamente questi file come immediatamente scaduti:

```
Cache-Control: max-age=0
```

E a proposito della normale cache del browser, che dire delle intestazioni di cache per i bundle CSS e JS?

8.4.7 Precauzioni riguardanti l'uso della cache del browser e dei lavoratori del servizio

I bundle CSS / Js archiviati in Cache Storage verranno caricati dalla rete e questi bundle potrebbero o non potrebbero essere offerti con Cache-Controlun'intestazione, il che significa che potenzialmente abbiamo due cache in azione, che potrebbero interferire tra loro.

Ciò potrebbe portare a scenari di problemi, come ad esempio una nuova versione di Service Worker viene installata, ma tenta di caricare una nuova versione di un file bundle JS, che non ha cambiato il nome del file!

Ma il file viene memorizzato nella cache nella normale cache del browser e la versione antica viene accidentalmente ancora servita al Service Worker.

Ciò significa che l'installazione dell'operatore del servizio viene completata correttamente, ma in Cache Storage è ora disponibile la versione errata di uno dei bundle, il che significa che l'installazione dell'applicazione è danneggiata.

Quindi, come evitiamo di imbattersi in questi scenari? Il più semplice è applicare le stesse politiche di caching che faremmo per un'applicazione non PWA: diversi tipi di file richiedono strategie di caching diverse. **Cache-Control per i pacchetti CSS / JS** Per CSS e JS bundle, il più semplice è quello di aggiungere al nome del file un hash del contenuto del file, o di un numero di versione, come ad esempio: bootstrap.v4.min.css.

Quindi, per questi file, possiamo scegliere un'età massima molto lunga, in sostanza dichiarandoli immutabili e memorizzandoli nella cache per sempre:

```
  Cache-Control: max-age=31536000
```

Se è disponibile una nuova versione del file, il nome del file cambierà (questo potrebbe essere applicato dal sistema di generazione) e la nuova versione verrà scaricata e memorizzata nella cache.

Ciò eviterà molti problemi di memorizzazione nella cache comuni per entrambi i browser che supportano i Service Worker e quelli che non lo fanno. **Caricamento di pacchetti di risorse da domini di terze parti** In questo esempio, abbiamo scaricato tutti i bundle dal nostro dominio locale. Ma se volessimo caricare i bundle CSS e JS da altri domini di terze parti all'interno del lavoratore del servizio, come ad esempio da un CDN?

Questo è possibile, ma il dominio di terze parti consente di eseguire tale richiesta di origine incrociata, proprio come qualsiasi altra richiesta CORS.

Questo può essere fatto servendo il file bundle con questa intestazione:

```
access-control-allow-origin: https://yourdomain.com
```

Se stiamo servendo questi file bundle da un CDN come Amazon Cloudfront, e vogliamo che i file siano caricabili tramite una richiesta di origine incrociata proveniente da qualsiasi dominio e non solo https://yourdomain.com, possiamo invece utilizzare questa intestazione:

```
access-control-allow-origin: *
```

8.5 conclusioni

Come possiamo vedere, tutte le molteplici funzionalità PWA e le relative API PWA hanno più senso se le guardiamo insieme e nel contesto di un caso d'uso specifico, anziché in isolamento.

Possiamo fare molto di più del caso d'uso di download e installazione che abbiamo trattato, questo è stato solo un esempio che sembra essere il miglior punto di partenza per capire perché il ciclo di vita del Service Worker è stato progettato così com'era.

La filosofia di base delle specifiche di Service Worker consiste nel mettere queste capacità di proxy di rete nelle mani degli sviluppatori, in modo da poter implementare molti diversi casi e modelli di utilizzo PWA, anziché fornire solo una serie di modelli offline predefiniti (come era il caso di Application Cache).

9 Compatibilità web

9.1 Desktop

Desktop | Mobile

Feature	Chrome	Edge	Firefox (Gecko)	Internet Explorer	Opera	Safari (WebKit)
Basic support	40.0	16[2]	33.0 (33.0)[1]	No support	24	No support

Figura 11: Compatibilità web

9.2 Mobile

Desktop | **Mobile**

Feature	Android Webview	Chrome for Android	Firefox Mobile (Gecko)	Firefox OS	IE Phone	Opera Mobile	Safari Mobile
Basic support	No support	40.0	(Yes)	(Yes)	No support	(Yes)	No support

Figura 12: Compatibilità mobile