

OFFLINE FIRST CON I SERVICE  
WORKER PER MIGLIORARE LE  
PERFORMANCE IN OGNI  
CONDIZIONE

**#FVGDEVS**

# DAVIDE SERAFINI

Freelance Fullstack Developer | Digital Consultant

<https://www.linkedin.com/in/davideserafini/>

Twitter: @serafit

# COS'È UN SERVICE WORKER?

Un Service Worker è uno script che viene eseguito in background e separato dalla pagina:

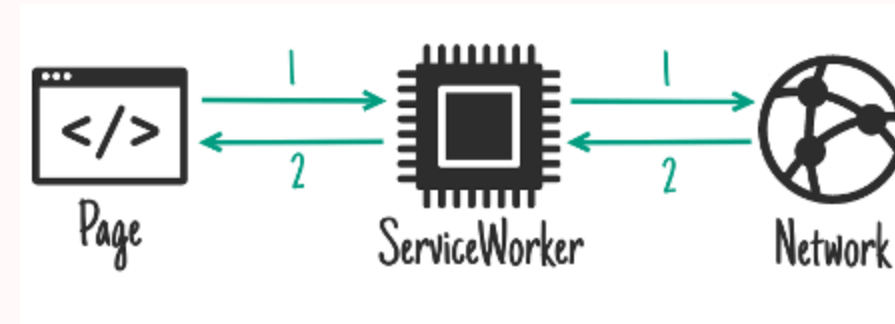
- Non può accedere direttamente al DOM della pagina
- Il lifecycle è diverso da quello della pagina
- Permette l'uso di Push Notifications e Background Sync
- Agisce come un **proxy** tra la pagina e la rete, permettendo di gestire le richieste alla rete (cache, modificare la risposta, fallback)

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Blackberry Browser	Opera Mobile *	Chrome for Android	Firefox for Android
		2-32										
		1 33-43										
		44										
		3 45										
		46-51										
		3 52										
	12-14	53-59	4-39		10-26							
	2 15-16	3 60	40-44	3.1-11	27-31	3.2-11.2						
6-10	17	61-64	45-71	11.1	32-57	11.4		2.1-4.4.4	7	12-12.1		
11	18	65	72	12	58	12.1	all	67	10	46	71	64
		66-67	73-75	12.1-TP		12.2						

Screenshot from [caniuse.com](https://caniuse.com)

# COSA VUOL DIRE "GESTIRE LE RICHIESTE ALLA RETE"?

Il SW agisce come un proxy che permette di **intercettare** le richieste fatte dalla pagina e di modificarne il comportamento



- **Salvare programmaticamente** la risposta nella cache
- **Ritornare programmaticamente** la risposta dalla cache
- **Modificare** la risposta combinando più risorse (dalla rete e dalla cache) in una
- **Modificare** la risposta fornendo un fallback dalla cache in caso di errore di rete

A differenza della cache HTTP del browser, sfruttando la Cache API e i Service Worker abbiamo (quasi) pieno controllo sulla gestione della cache

# COS'È LA CACHE API?

La Cache API offre un meccanismo di storage per le richieste.

Per ogni *origin* si possono creare più oggetti *Cache*.

La gestione di questa Cache è delegata interamente allo sviluppatore.

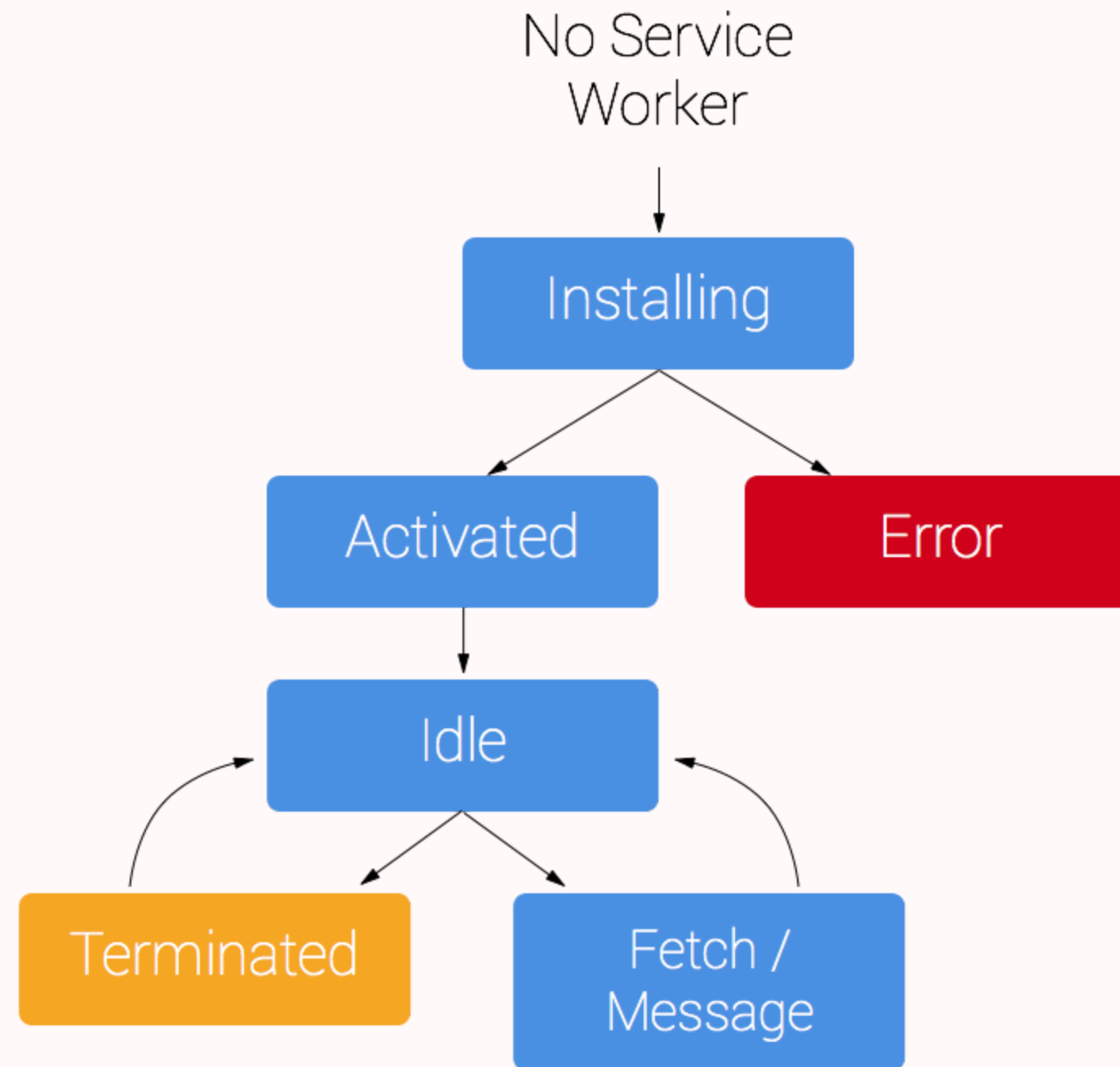
- `caches.open(cacheName)` ritorna l'oggetto Cache definito da `cacheName`
- `caches.delete(cacheName)` cancella la cache definita da `cacheName`
- `caches.keys()` ritorna l'elenco delle Cache per l'origin
- `caches.match(request)` cerca la request in tutte le Cache dell'origin
- `cache.addAll(requests)` accetta un array di URL, richiede le risorse e le aggiunge
- `cache.add(request)` come `addAll`, ma per un singolo URL
- `cache.put(request, response)` aggiunge una risposta alla cache
- `cache.delete(request)` cancella un elemento dalla cache

`caches` è una variabile globale che rappresenta un'istanza di `CacheStorage`

COME  
FUNZIONANO  
I SERVICE  
WORKER:  
NELLA  
PAGINA

```
if ('serviceWorker' in navigator) {  
  window.addEventListener('load', function () {  
    navigator.serviceWorker.register('/sw.js')  
      .then(function (registration) {  
        // Registration was successful  
        console.log('ServiceWorker registration successful');  
      }, function (err) {  
        // registration failed :(  
        console.log('ServiceWorker registration failed');  
      });  
  });  
}
```

# COME FUNZIONANO I SERVICE WORKER: IL LIFECYCLE



*Image from developers.google.com*



COME  
FUNZIONANO I  
SERVICE  
WORKER:  
INSTALL

```
// Nome della cache
const appCacheName = 'herodex-app-cache-v2';

// Risorse da salvare in cache
const urlsToCache = [
  '/css/main.css',
  '/js/main.js',
  '/site-shell.html'
];

// Handler per l'evento install
self.addEventListener('install', event => {
  event.waitUntil(
    precache(urlsToCache)
  );
});

async function precache(assetsToCache) {
  const cache = await caches.open(appCacheName);
  return cache.addAll(assetsToCache);
}
```

COME  
FUNZIONANO I  
SERVICE  
WORKER:  
ACTIVATE

```
// Nome della cache
const appCacheName = 'herodex-app-cache-v2';

self.addEventListener('activate', event => {
  event.waitUntil(
    caches.keys().then(cacheNames => {
      return Promise.all(
        cacheNames.filter(cacheName => {
          // Elimina versioni precedenti della cache
          return cacheName.includes('herodex-app-cache')
            && cacheName !== appCacheName;
        }).map(cacheName => caches.delete(cacheName))
      );
    })
  );
});
```

# COME FUNZIONANO I SERVICE WORKER: FETCH

```
self.addEventListener('fetch', event => {  
  event.respondWith(  
    handleRequest(event.request)  
  );  
});
```

`handleRequest` è un placeholder, la cui implementazione può variare molto in base alla strategia di caching adottata

# COME GESTIRE LE RICHIESTE: STRATEGIE

All'interno dell'evento fetch la richiesta può essere gestita in vari modi, a seconda dello scopo che vogliamo ottenere.

## METODI TRASPARENTI ALL'APPLICAZIONE

- **Network Only** la richiesta viene passata così com'è alla rete, senza interagire con la cache
- **Cache Only** la risposta viene ritornata dalla cache, altrimenti viene ritornato un errore
- **Cache, falling back to network** la richiesta viene prima gestita dalla cache, se non viene trovato un match allora si passa alla rete
- **Network, falling back to cache** la richiesta viene prima gestita dalla rete, in caso di errore di rete viene fornita una versione in cache, se presente

## METODI CHE RICHIEDONO CAMBI ALL'APPLICAZIONE

- **Cache, then network** la richiesta viene inviata sia alla cache che al network, usando la richiesta più aggiornata come contenuto finale

# COME GESTIRE LE RICHIESTE: NETWORK ONLY

Questo pattern serve a emulare il comportamento normale del browser, bypassando qualsiasi logica presente nel Service Worker

```
self.addEventListener('fetch', event => {  
  event.respondWith(  
    fetch(event.request)  
  );  
});
```

In questi casi è ancora meglio evitare l'invocazione di `event.respondWith` così da non causare overhead inutili.

```
self.addEventListener('fetch', event => {  
  if (!cacheRequest(event.request)) {  
    return;  
  }  
  ...  
});
```

# COME GESTIRE LE RICHIESTE: CACHE ONLY

Questo pattern è usato per ritornare sempre una risposta dalla cache.  
Utile soprattutto per le risorse salvate durante l'installazione del SW

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

In alternativa si può controllare solo una cache

```
const appCacheName = 'herodex-app-cache-v2';  
self.addEventListener('fetch', event => {  
  event.respondWith(  
    caches.open(appCacheName).then(cache => {  
      return cache.match(event.request);  
    })  
  );  
});
```

# COME GESTIRE LE RICHIESTE: CACHE, FALLING BACK TO NETWORK

Questo pattern è usato per ritornare una risposta dalla cache se è presente, altrimenti si ripiega sul newtwork

```
self.addEventListener('fetch', event => {  
  event.respondWith(  
    fromCache(event.request)  
      .then(function(response) {  
        return response || fromNetwork(event.request);  
      })  
  );  
});
```

Questo è il pattern principale nell'approccio Offline-first.

`caches.match()` ritorna sempre una promise, con `undefined` come valore di ritorno se l'elemento non è stato trovato.

Le casistiche vanno sempre gestite nel `then(response)`

# COME GESTIRE LE RICHIESTE: NETWORK, FALLING BACK TO CACHE

Con questo pattern viene fornita la versione più aggiornata se la rete è disponibile, altrimenti una versione più vecchia caricata in precedenza

```
self.addEventListener('fetch', event => {  
  event.respondWith(  
    fromNetwork(event.request)  
      .catch(function() {  
        return fromCache(event.request);  
      })  
  );  
});
```

Il problema con questo pattern è che, in caso di connessione instabile, ci potrebbe volere parecchio tempo prima che la rete fallisca e l'utente riceva la versione in cache.

`fromCache(request)` può essere a sua volta seguita da un `then` per valutare se l'elemento è stato trovato nella cache, e eventualmente mostrare una fallback generica.



# COME GESTIRE LE RICHIESTE: CACHE, THEN NETWORK

Questo pattern è usato per ritornare una risposta dalla cache se è presente, mentre viene inviata la richiesta tramite la rete

```
let networkDataReceived = false;
let networkUpdate = fetch('/data.json').then((response) => {
  // Gestione della risposta, es. response.json()
  ...
  networkDataReceived = true;
  updatePage(data);
});

caches.match('/data.json').then((response) => {
  // Gestione della risposta, es. if(response)
  ...
  // !! Non sovrascrivere la risposta della rete
  if (!networkDataReceived) {
    updatePage(data);
  }
}).catch(() => {
  return networkUpdate;
})
```

# COME GESTIRE LE RICHIESTE: CACHE, THEN NETWORK

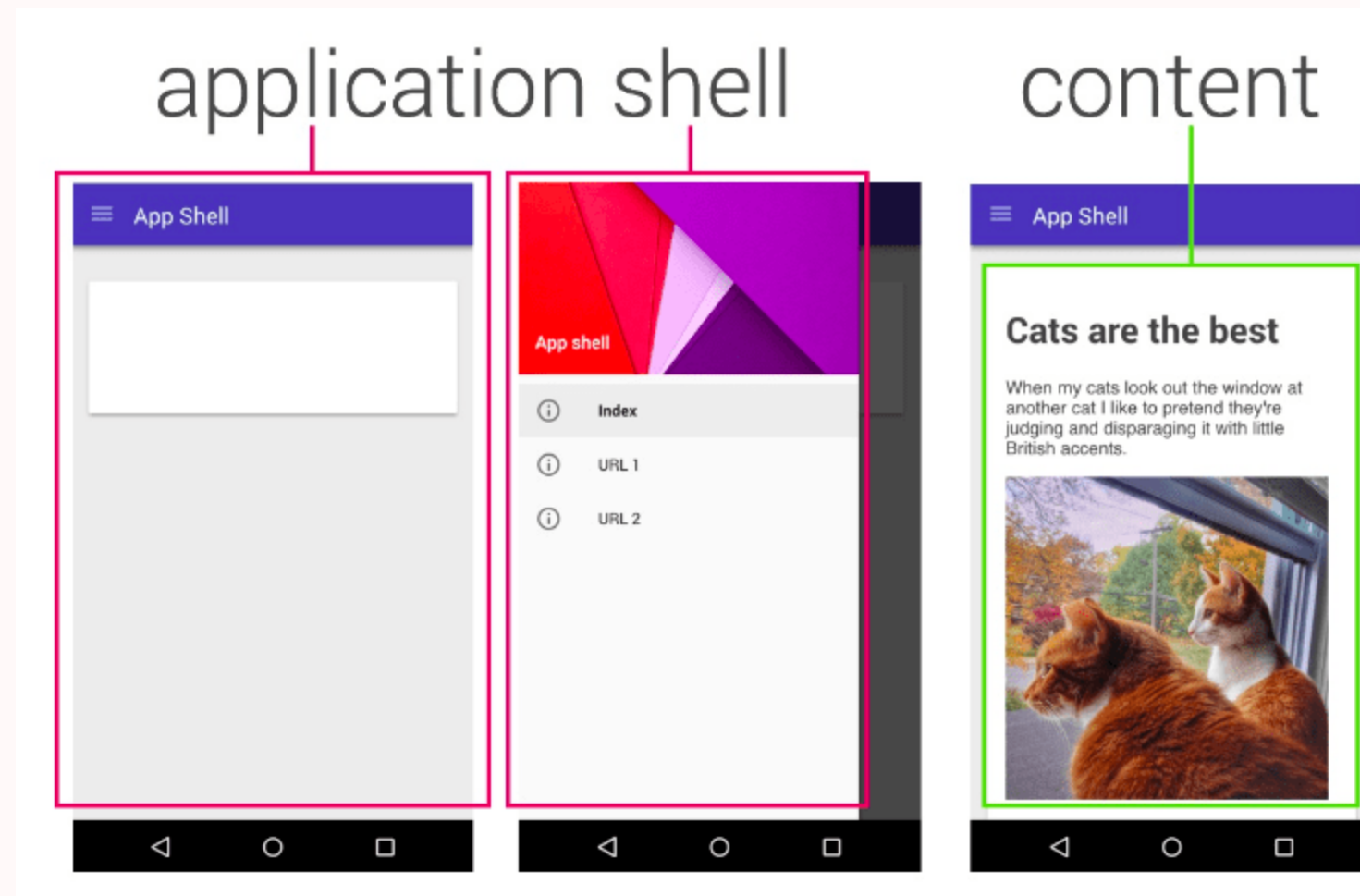
La richiesta alla rete viene salvata in cache e poi viene ritornata.

```
const appCacheName = 'herodex-app-cache-v2';
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open(appCacheName)
      .then((cache) => {
        return fromNetwork(event.request)
          .then((response) => {
            cache.put(event.request, response.clone());
            return response;
          });
      })
  );
});
```

# PATTERN ARCHITETTURALI: APP SHELL

L'application shell è un'architettura per emulare nel web il comportamento delle applicazioni native.

L'app shell è composto dalle risorse minime (HTML + CSS + JS) necessarie per mostrare istantaneamente a schermo la struttura dell'applicazione, mentre il contenuto viene caricato dinamicamente.

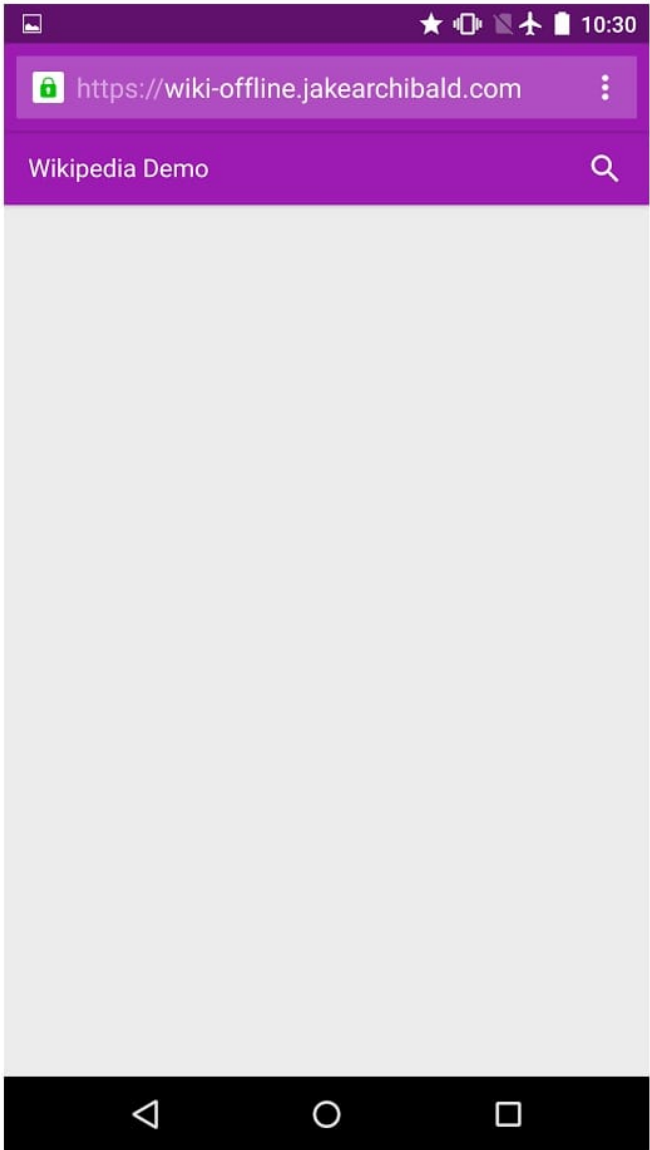


*Image from developers.google.com*

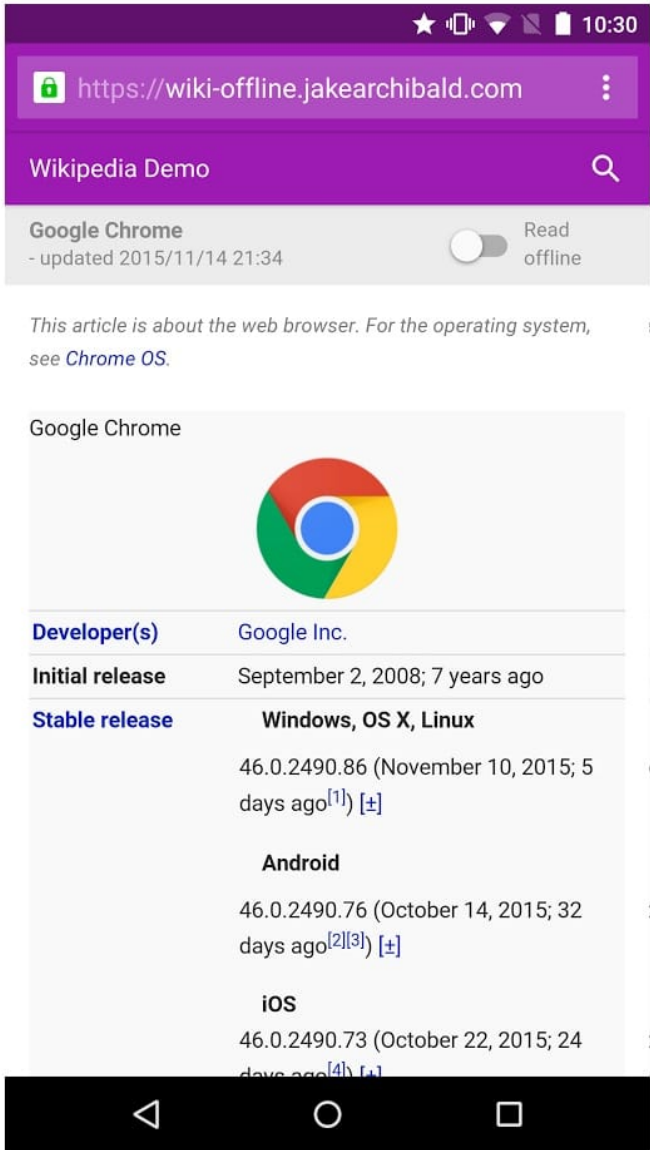
# PATTERN ARCHITETTURALI: APP SHELL

Oltre a caricare l'app shell durante l'install event, si può anche mettere in cache il contenuto dinamico per permetterne la fruizione offline

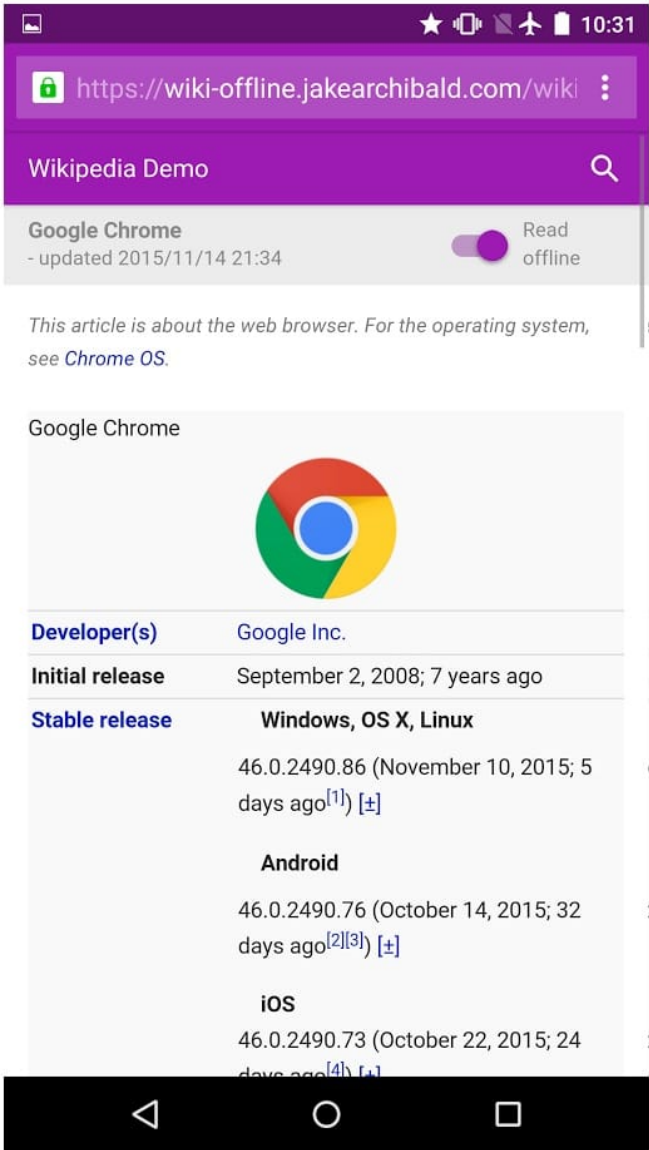
Offline Wikipedia webapp



Cached application shell



Content dynamically loaded in



Content can also be cached for offline viewing

# DEMO

- Finto Server Side Rendering
- Caricamento dinamico del contenuto
- App Shell "modificato"

<https://herodex-1f20e.firebaseio.com>  
<https://github.com/davideserafini/herodex>

# GRAZIE!

<https://www.linkedin.com/in/davideserafini/>

Twitter: @serafit