Service worker configuration

Prerequisites

A basic understanding of the following:

- Service Worker in Production.

The `ngsw-config.json` configuration file specifies which files and data URLs the Angular service worker should cache and how it should update the cached files and data. The Angular CLI processes the configuration file during `ng build`. Manually, process it with the `ngsw-config` tool (where `<project-name>` is the name of the project being built):

./node_modules/.bin/ngsw-config ./dist/<project-name> ./ngsw-config.json [/base/href]

The configuration file uses the JSON format. All file paths must begin with `/`, which corresponds to the deployment directory—usually `dist/<project-name>` in CLI projects.

{@a glob-patterns} Unless otherwise noted, patterns use a limited glob format:

- `**` matches 0 or more path segments.
- `*` matches 0 or more characters excluding `/`.
- `?` matches exactly one character excluding `/`.
- The `!` prefix marks the pattern as being negative, meaning that only files that don't match the pattern are included.

Example patterns:

- `/**/*.html` specifies all HTML files.
- `/*.html` specifies only HTML files in the root.
- `!/**/*.map` exclude all sourcemaps.

The following sections describe each property of the configuration file.

`appData`

This section enables you to pass any data you want that describes this particular version of the application. The `SwUpdate` service includes that data in the update notifications. Many applications use this section to provide additional information for the display of UI popups, notifying users of the available update.

{@a index-file}

`index`

Specifies the file that serves as the index page to satisfy navigation requests. Usually this is `/index.html`.

`assetGroups`

Assets are resources that are part of the application version that update along with the application. They can include resources loaded from the page's origin as well as third-party resources loaded from CDNs and other external URLs. As not all such external URLs might be known at build time, URL patterns can be matched.

This field contains an array of asset groups, each of which defines a set of asset resources and the policy by which they are cached.

```
{
  "assetGroups": [
    {
      ...
    },
    {
      ...
    }
  ]
}
```

When the ServiceWorker handles a request, it checks asset groups in the order in which they appear in `ngsw-config.json`. The first asset group that matches the requested resource handles the request.

It is recommended that you put the more specific asset groups higher in the list. For example, an asset group that matches `/foo.js` should appear before one that matches `*.js`.

Each asset group specifies both a group of resources and a policy that governs them. This policy determines when the resources are fetched and what happens when changes are detected.

Asset groups follow the Typescript interface shown here:

```
interface AssetGroup {
  name: string;
  installMode?: 'prefetch' | 'lazy';
  updateMode?: 'prefetch' | 'lazy';
  resources: {
    files?: string[];
    urls?: string[];
  };
  cacheQueryOptions?: {
    ignoreSearch?: boolean;
  };
}
```

`name`

A `name` is mandatory. It identifies this particular group of assets between versions of the configuration.

`installMode`

The `installMode` determines how these resources are initially cached. The `installMode` can be either of two values:

- `prefetch` tells the Angular service worker to fetch every single listed resource while it's caching the current version of the application. This is bandwidth-intensive but ensures resources are available whenever they're requested, even if the browser is currently offline.

- `lazy` does not cache any of the resources up front. Instead, the Angular service worker only caches resources for which it receives requests. This is an on-demand caching mode. Resources that are never requested are not cached. This is useful for things like images at different resolutions, so the service worker only caches the correct assets for the particular screen and orientation.

Defaults to `prefetch`.

`updateMode`

For resources already in the cache, the `updateMode` determines the caching behavior when a new version of the application is discovered. Any resources in the group that have changed since the previous version are updated in accordance with `updateMode`.

- `prefetch` tells the service worker to download and cache the changed resources immediately.
- `lazy` tells the service worker to not cache those resources. Instead, it treats them as unrequested and waits until they're requested again before updating them. An `updateMode` of `lazy` is only valid if the `installMode` is also `lazy`.

Defaults to the value `installMode` is set to.

`resources`

This section describes the resources to cache, broken up into the following groups:

- `files` lists patterns that match files in the distribution directory. These can be single files or glob-like patterns that match a number of files.
- `urls` includes both URLs and URL patterns that are matched at runtime. These resources are not fetched directly and do not have content hashes, but they are cached according to their HTTP headers. This is most useful for CDNs such as the Google Fonts service.
  (Negative glob patterns are not supported and `?` will be matched literally; that is, it will not match any character other than `?`.)

`cacheQueryOptions`

These options are used to modify the matching behavior of requests. They are passed to the browsers `Cache#match` function. See MDN for details. Currently, only the following options are supported:

- `ignoreSearch`: Ignore query parameters. Defaults to `false`.

`dataGroups`

Unlike asset resources, data requests are not versioned along with the application. They're cached according to manually-configured policies that are more useful for situations such as API requests and other data dependencies.

This field contains an array of data groups, each of which defines a set of data resources and the policy by which they are cached.

```
{
  "dataGroups": [
    {
```

```
      ...
    },
    {
      ...
    }
  ]
}
```

When the ServiceWorker handles a request, it checks data groups in the order in which they appear in `ngsw-config.json`. The first data group that matches the requested resource handles the request.

It is recommended that you put the more specific data groups higher in the list. For example, a data group that matches `/api/foo.json` should appear before one that matches `/api/*.json`.

Data groups follow this Typescript interface:

```
export interface DataGroup {
  name: string;
  urls: string[];
  version?: number;
  cacheConfig: {
    maxSize: number;
    maxAge: string;
    timeout?: string;
    strategy?: 'freshness' | 'performance';
  };
  cacheQueryOptions?: {
    ignoreSearch?: boolean;
  };
}
```

`name`

Similar to `assetGroups`, every data group has a `name` which uniquely identifies it.

`urls`

A list of URL patterns. URLs that match these patterns are cached according to this data group's policy. Only non-mutating requests (GET and HEAD) are cached.

- Negative glob patterns are not supported.
- `?` is matched literally; that is, it matches *only* the character `?`.

`version`

Occasionally APIs change formats in a way that is not backward-compatible. A new version of the application might not be compatible with the old API format and thus might not be compatible with existing cached resources from that API.

`version` provides a mechanism to indicate that the resources being cached have been updated in a backwards-incompatible way, and that the old cache entries—those from previous versions—should be discarded.

`version` is an integer field and defaults to `1`.

`cacheConfig`

This section defines the policy by which matching requests are cached.

`maxSize`

(required) The maximum number of entries, or responses, in the cache. Open-ended caches can grow in unbounded ways and eventually exceed storage quotas, calling for eviction.

`maxAge`

(required) The `maxAge` parameter indicates how long responses are allowed to remain in the cache before being considered invalid and evicted. `maxAge` is a duration string, using the following unit suffixes:

- `d` : days
- `h` : hours
- `m` : minutes
- `s` : seconds
- `u` : milliseconds

For example, the string `3d12h` caches content for up to three and a half days.

`timeout`

This duration string specifies the network timeout. The network timeout is how long the Angular service worker waits for the network to respond before using a cached response, if configured to do so. `timeout` is a duration string, using the following unit suffixes:

- `d` : days
- `h` : hours
- `m` : minutes
- `s` : seconds
- `u` : milliseconds

For example, the string `5s30u` translates to five seconds and 30 milliseconds of network timeout.

`strategy`

The Angular service worker can use either of two caching strategies for data resources.

- `performance` , the default, optimizes for responses that are as fast as possible. If a resource exists in the cache, the cached version is used, and no network request is made. This allows for some staleness, depending on the `maxAge` , in exchange for better performance. This is suitable for resources that don't change often; for example, user avatar images.
- `freshness` optimizes for currency of data, preferentially fetching requested data from the network. Only if the network times out, according to `timeout` , does the request fall back to the cache. This is useful for resources that change frequently; for example, account balances.

You can also emulate a third strategy, staleWhileRevalidate, which returns cached data (if available), but also fetches fresh data from the network in the background for next time. To use this strategy set `strategy` to `freshness` and `timeout` to `0u` in `cacheConfig` .

This essentially does the following:

1. Try to fetch from the network first.
2. If the network request does not complete after 0ms (that is, immediately), fall back to the cache (ignoring cache age).
3. Once the network request completes, update the cache for future requests.
4. If the resource does not exist in the cache, wait for the network request anyway.

`cacheOpaqueResponses`

Whether the Angular service worker should cache opaque responses or not.

If not specified, the default value depends on the data group's configured strategy:

- For groups with the `freshness` strategy, the default value is `true` (cache opaque responses). These groups will request the data anew every time, only falling back to the cached response when offline or on a slow network. Therefore, it doesn't matter if the service worker caches an error response.
- For groups with the `performance` strategy, the default value is `false` (do not cache opaque responses). These groups would continue to return a cached response until `maxAge` expires, even if the error was due to a temporary network or server issue. Therefore, it would be problematic for the service worker to cache an error response.

Note on opaque responses
In case you are not familiar, an opaque response is a special type of response returned when requesting a resource that is on a different origin which doesn't return CORS headers. One of the characteristics of an opaque response is that the service worker is not allowed to read its status, meaning it can't check if the request was successful or not. See Introduction to fetch() for more details.

If you are not able to implement CORS—for example, if you don't control the origin—prefer using the `freshness` strategy for resources that result in opaque responses.

`cacheQueryOptions`

See assetGroups for details.

`navigationUrls`

This optional section enables you to specify a custom list of URLs that will be redirected to the index file.

Handling navigation requests

The ServiceWorker redirects navigation requests that don't match any `asset` or `data` group to the specified index file. A request is considered to be a navigation request if:

1. Its mode is `navigation`.
2. It accepts a `text/html` response (as determined by the value of the `Accept` header).
3. Its URL matches certain criteria (see the following).

By default, these criteria are:

1. The URL must not contain a file extension (that is, a `.`) in the last path segment.
2. The URL must not contain `__`.

To configure whether navigation requests are sent through to the network or not, see the navigationRequestStrategy section.

Matching navigation request URLs

While these default criteria are fine in most cases, it is sometimes desirable to configure different rules. For example, you might want to ignore specific routes (that are not part of the Angular app) and pass them through to the server.

This field contains an array of URLs and glob-like URL patterns that are matched at runtime. It can contain both negative patterns (that is, patterns starting with `!`) and non-negative patterns and URLs.

Only requests whose URLs match *any* of the non-negative URLs/patterns and *none* of the negative ones are considered navigation requests. The URL query is ignored when matching.

If the field is omitted, it defaults to:

```
[
  '/**',           // Include all URLs.
  '!/**/*.*',      // Exclude URLs to files.
  '!/**/*__*',     // Exclude URLs containing `__` in the last segment.
  '!/**/*__*/**',  // Exclude URLs containing `__` in any other segment.
]
```

{@a navigation-request-strategy}

`navigationRequestStrategy`

This optional property enables you to configure how the service worker handles navigation requests:

```
{
  "navigationRequestStrategy": "freshness"
}
```

Possible values:

- `'performance'`: The default setting. Serves the specified index file, which is typically cached.
- `'freshness'`: Passes the requests through to the network and falls back to the `performance` behavior when offline. This value is useful when the server redirects the navigation requests elsewhere using an HTTP redirect (3xx status code). Reasons for using this value include:
  - Redirecting to an authentication website when authentication is not handled by the application.
  - Redirecting specific URLs to avoid breaking existing links/bookmarks after a website redesign.
  - Redirecting to a different website, such as a server-status page, while a page is temporarily down.

The `freshness` strategy usually results in more requests sent to the server, which can increase response latency. It is recommended that you use the default performance strategy whenever possible.