{@a glob}

# Reference: Configuration file

The `src/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 CLI processes the configuration file during `ng build --prod`. Manually, you can process it with the `ngsw-config` tool:

```
ngsw-config dist src/ngswn-config.json /base/href
```

The configuration file uses the JSON format. All file paths must begin with `/`, which is the deployment directory—usually `dist` in CLI projects.

Patterns use a limited glob format:

- `**` matches 0 or more path segments.
- `*` matches exactly one path segment or filename segment.
- The `!` prefix marks the pattern as being negative, meaning that only files that don't match the pattern will be included.

Example patterns:

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

Each section of the configuration file is described below.

## `appData`

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

## `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 app version that update along with the app. 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 may 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": [{
    ...
  }, {
    ...
  }]
}
```

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[];
    versionedFiles?: string[];
    urls?: string[];
  };
}
```

### `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 app. 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 will not be 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.

### `updateMode`

For resources already in the cache, the `updateMode` determines the caching behavior when a new version of the app 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`.

### `resources`

This section describes the resources to cache, broken up into three 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.
- `versionedFiles` is like `files` but should be used for build artifacts that already include a hash in the filename, which is used for cache busting. The Angular service worker can optimize some aspects of its operation if it can assume file contents are immutable.
- `urls` includes both URLs and URL patterns that will be matched at runtime. These resources are not fetched directly and do not have content hashes, but they will be cached according to their HTTP headers. This is most useful for CDNs such as the Google Fonts service.

## `dataGroups`

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

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';
  };
}
```

### `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 will be cached according to this data group's policy.

### `version`

Occasionally APIs change formats in a way that is not backward-compatible. A new version of the app may not be compatible with the old API format and thus may 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 `0`.

### `cacheConfig`

This section defines the policy by which matching requests will be 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` will cache 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 will wait for the network to respond before using a cached response, if configured to do so.

### `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. 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.