SVELTEKIT • REFERENCE

@sveltejs/kit

ON THIS PAGE

```
import {
  VERSION,
  error,
  fail,
  isActionFailure,
  isHttpError,
  isRedirect,
  json,
  redirect,
  text
} from '@sveltejs/kit';
```

VERSION

```
const VERSION: string;
```

error

Throws an error with a HTTP status code and an optional message. When called during request handling, this will cause SvelteKit to return an error response without invoking `handleError`. Make sure you're not catching the thrown error, which would prevent SvelteKit from handling it.

```
function error(status: number, body: App.Error): never;
```

```
  status: number,
  body?: {
    message: string;
  } extends App.Error
    ? App.Error | string | undefined
    : never
): never;
```

fail

Create an `ActionFailure` object.

```
function fail(status: number): ActionFailure<undefined>;
```

```
function fail<
  T extends Record<string, unknown> | undefined = undefined
>(status: number, data: T): ActionFailure<T>;
```

isActionFailure

Checks whether this is an action failure thrown by `fail` .

```
function isActionFailure(
  e: unknown
): e is ActionFailure<undefined>;
```

isHttpError

Checks whether this is an error thrown by `error` .

```
function isHttpError<T extends number>(
```

isRedirect

Checks whether this is a redirect thrown by `redirect` .

```
function isRedirect(e: unknown): e is Redirect_1;
```

json

Create a JSON `Response` object from the supplied data.

```
function json(
  data: any,
  init?: ResponseInit | undefined
): Response;
```

redirect

Redirect a request. When called during request handling, SvelteKit will return a redirect response. Make sure you're not catching the thrown redirect, which would prevent SvelteKit from handling it.

```
function redirect(
  status:
    | 300
    | 301
    | 302
    | 303
    | 304
    | 305
    | 306
): never;
```

text

Create a `Response` object from the supplied body.

```
function text(
  body: string,
  init?: ResponseInit | undefined
): Response;
```

Action

Shape of a form action method that is part of `export const actions = {..}` in `+page.server.js`. See form actions for more information.

```
type Action<
  Params extends Partial<Record<string, string>> = Partial<
    Record<string, string>
  >,
  OutputData extends Record<string, any> | void = Record<
    string,
    any
  > | void,
  RouteId extends string | null = string | null
> = (
  event: RequestEvent<Params, RouteId>
) => MaybePromise<OutputData>;
```

ActionFailure

```
status: number;
```

```
data: T;
```

```
[uniqueSymbol]: true;
```

ActionResult

When calling a form action via fetch, the response will be one of these shapes.

```
<form method="post" use:enhance={() => {
  return ({ result }) => {
    // result is of type ActionResult
  };
}}
```

```
type ActionResult<
  Success extends
    | Record<string, unknown>
    | undefined = Record<string, any>,
  Failure extends
    | Record<string, unknown>
    | undefined = Record<string, any>
> =
  | { type: 'success'; status: number; data?: Success }
  | { type: 'failure'; status: number; data?: Failure }
  | { type: 'redirect'; status: number; location: string }
  | { type: 'error'; status?: number; error: any };
```

Actions

Shape of the `export const actions = {..}` object in `+page.server.js`. See form actions for

```
  Params extends Partial<Record<string, string>> = Partial<
    Record<string, string>
  >,
  OutputData extends Record<string, any> | void = Record<
    string,
    any
  > | void,
  RouteId extends string | null = string | null
> = Record<string, Action<Params, OutputData, RouteId>>;
```

Adapter

Adapters are responsible for taking the production build and turning it into something that can be deployed to a platform of your choosing.

```
interface Adapter {…}
```

```
name: string;
```

The name of the adapter, using for logging. Will typically correspond to the package name.

```
adapt(builder: Builder): MaybePromise<void>;
```

`builder` An object provided by SvelteKit that contains methods for adapting the app

This function is called after SvelteKit has built your app.

```
supports?: {…}
```

Checks called during dev and build to determine whether specific features will work in

- `config` The merged route config

Test support for `read` from `$app/server`

```
emulate?(): MaybePromise<Emulator>;
```

Creates an `Emulator`, which allows the adapter to influence the environment during dev, build and prerendering

AfterNavigate

The argument passed to [`afterNavigate`](#) callbacks.

```
interface AfterNavigate extends Omit<Navigation, 'type'> {…}
```

```
type: Exclude<NavigationType, 'leave'>;
```

The type of navigation:

- `enter`: The app has hydrated
- `form`: The user submitted a `<form>`
- `link`: Navigation was triggered by a link click
- `goto`: Navigation was triggered by a `goto(...)` call or a redirect
- `popstate`: Navigation was triggered by back/forward navigation

```
willUnload: false;
```

Since `afterNavigate` callbacks are called after a navigation completes, they will never be

```
type AwaitedActions<
  T extends Record<string, (...args: any) => any>
> = OptionalUnion<
  {
    [Key in keyof T]: UnpackValidationError<
      Awaited<ReturnType<T[Key]>>
    >;
  }[keyof T]
>;
```

BeforeNavigate

The argument passed to `beforeNavigate` callbacks.

```
interface BeforeNavigate extends Navigation {…}
```

```
cancel(): void;
```

Call this to prevent the navigation from starting.

Builder

This object is passed to the `adapt` function of adapters. It contains various methods and properties that are useful for adapting the app.

```
interface Builder {…}
```

```
log: Logger;
```

Print messages to the console. `log.info` and `log.minor` are silent unless Vite's

Remove `dir` and all its contents.

```
mkdirp(dir: string): void;
```

Create `dir` and any required parent directories.

```
config: ValidatedConfig;
```

The fully resolved `svelte.config.js` .

```
prerendered: Prerendered;
```

Information about prerendered pages and assets, if any.

```
routes: RouteDefinition[];
```

An array of all routes (including prerendered)

```
createEntries(fn: (route: RouteDefinition) => AdapterEntry): Promise<void>;
```

- `fn` A function that groups a set of routes into an entry point

DEPRECATED Use `builder.routes` instead

Create separate functions that map to one or more routes of your app.

```
findServerAssets(routes: RouteDefinition[]): string[];
```

Find all the assets imported by server files belonging to `routes` for single-page apps.

```
generateEnvModule(): void;
```

Generate a module exposing build-time environment variables as `$env/dynamic/public` .

```
generateManifest(opts: { relativePath: string; routes?: RouteDefinition[] }): string;
```

- `opts` a relative path to the base directory of the app and optionally in which format (esm or cjs) the manifest should be generated

Generate a server-side manifest to initialise the SvelteKit server with.

```
getBuildDirectory(name: string): string;
```

- `name` path to the file, relative to the build directory

Resolve a path to the `name` directory inside `outDir` , e.g. `/path/to/.svelte-kit/my-adapter` .

```
getClientDirectory(): string;
```

Get the fully resolved path to the directory containing client-side assets, including the contents of your `static` directory.

```
getServerDirectory(): string;
```

Get the fully resolved path to the directory containing server-side code.

```
getAppPath(): string;
```

`dest` the destination folder

RETURNS an array of files written to `dest`

Write client assets to `dest` .

```
writePrerendered(dest: string): string[];
```

`dest` the destination folder

RETURNS an array of files written to `dest`

Write prerendered files to `dest` .

```
writeServer(dest: string): string[];
```

`dest` the destination folder

RETURNS an array of files written to `dest`

Write server-side code to `dest` .

```
copy(
  from: string,
  to: string,
  opts?: {
    filter?(basename: string): boolean;
    replace?: Record<string, string>;
  }
): string[];
```

`from` the source file or directory

`to` the destination file or directory

`opts.filter` a function to determine whether a file or directory should be copied

Copy a file or directory.

```
compress(directory: string): Promise<void>;
```

`directory` The directory containing the files to be compressed

Compress files in `directory` with gzip and brotli, where appropriate. Generates `.gz` and `.br` files alongside the originals.

Config

See the configuration reference for details.

Cookies

```
interface Cookies {…}
```

```
get(name: string, opts?: import('cookie').CookieParseOptions): string | undefined;
```

`name` the name of the cookie

`opts` the options, passed directly to `cookie.parse`. See documentation here

Gets a cookie that was previously set with `cookies.set`, or from the request headers.

```
getAll(opts?: import('cookie').CookieParseOptions): Array<{ name: string; value: string }:
```

`opts` the options, passed directly to `cookie.parse`. See documentation here

Gets all cookies that were previously set with `cookies.set`, or from the request headers.

```
  name: string,
  value: string,
  opts: import('cookie').CookieSerializeOptions & { path: string }
): void;
```

- `name` the name of the cookie
- `value` the cookie value
- `opts` the options, passed directly to `cookie.serialize` . See documentation here

Sets a cookie. This will add a `set-cookie` header to the response, but also make the cookie available via `cookies.get` or `cookies.getAll` during the current request.

The `httpOnly` and `secure` options are `true` by default (except on http://localhost, where `secure` is `false`), and must be explicitly disabled if you want cookies to be readable by client-side JavaScript and/or transmitted over HTTP. The `sameSite` option defaults to `lax` .

You must specify a `path` for the cookie. In most cases you should explicitly set `path: '/'` to make the cookie available throughout your app. You can use relative paths, or set `path: ''` to make the cookie only available on the current path and its children

```
delete(name: string, opts: import('cookie').CookieSerializeOptions & { path: string }): vo
```

- `name` the name of the cookie
- `opts` the options, passed directly to `cookie.serialize` . The `path` must match the path of the cookie you want to delete. See documentation here

Deletes a cookie by setting its value to an empty string and setting the expiry date in the past.

You must specify a `path` for the cookie. In most cases you should explicitly set `path: '/'` to make the cookie available throughout your app. You can use relative paths, or set `path: ''` to make the cookie only available on the current path and its children

Docs

```
  name: string,
  value: string,
  opts: import('cookie').CookieSerializeOptions & { path: string }
): string;
```

`name` the name of the cookie

`value` the cookie value

`opts` the options, passed directly to `cookie.serialize` . See documentation here

Serialize a cookie name-value pair into a `Set-Cookie` header string, but don't apply it to the response.

The `httpOnly` and `secure` options are `true` by default (except on http://localhost, where `secure` is `false`), and must be explicitly disabled if you want cookies to be readable by client-side JavaScript and/or transmitted over HTTP. The `sameSite` option defaults to `lax` .

You must specify a `path` for the cookie. In most cases you should explicitly set `path: '/'` to make the cookie available throughout your app. You can use relative paths, or set `path: ''` to make the cookie only available on the current path and its children

Emulator

A collection of functions that influence the environment during dev, build and prerendering

```
interface Emulator {…}
```

```
platform?(details: { config: any; prerender: PrerenderOption }): MaybePromise<App.Platfor
```

A function that is called with the current route `config` and `prerender` option and

The `handle` hook runs every time the SvelteKit server receives a request and determines the response. It receives an `event` object representing the request and a function called `resolve`, which renders the route and generates a `Response`. This allows you to modify response headers or bodies, or bypass SvelteKit entirely (for implementing routes programmatically, for example).

```
type Handle = (input: {
  event: RequestEvent;
  resolve(
    event: RequestEvent,
    opts?: ResolveOptions
  ): MaybePromise<Response>;
}) => MaybePromise<Response>;
```

HandleClientError

The client-side `handleError` hook runs when an unexpected error is thrown while navigating.

If an unexpected error is thrown during loading or the following render, this function will be called with the error and the event. Make sure that this function *never* throws an error.

```
type HandleClientError = (input: {
  error: unknown;
  event: NavigationEvent;
  status: number;
  message: string;
}) => MaybePromise<void | App.Error>;
```

HandleFetch

The `handleFetch` hook allows you to modify (or replace) a `fetch` request that happens

```
  event: RequestEvent;
  request: Request;
  fetch: typeof fetch;
}) => MaybePromise<Response>;
```

HandleServerError

The server-side `handleError` hook runs when an unexpected error is thrown while responding to a request.

If an unexpected error is thrown during loading or rendering, this function will be called with the error and the event. Make sure that this function *never* throws an error.

```
type HandleServerError = (input: {
  error: unknown;
  event: RequestEvent;
  status: number;
  message: string;
}) => MaybePromise<void | App.Error>;
```

HttpError

The object returned by the `error` function.

```
interface HttpError {…}
```

```
status: number;
```

The HTTP status code, in the range 400-599.

```
body: App.Error;
```

See the configuration reference for details.

LessThan

```
type LessThan<
  TNumber extends number,
  TArray extends any[] = []
> = TNumber extends TArray['length']
  ? TArray[number]
  : LessThan<TNumber, [...TArray, TArray['length']]>;
```

Load

The generic form of `PageLoad` and `LayoutLoad`. You should import those from `./$types` (see generated types) rather than using `Load` directly.

```
type Load<
  Params extends Partial<Record<string, string>> = Partial<
    Record<string, string>
  >,
  InputData extends Record<string, unknown> | null = Record<
    string,
    any
  > | null,
  ParentData extends Record<string, unknown> = Record<
    string,
    any
  >,
  OutputData extends Record<
    string,
    unknown
  > | void = Record<string, any> | void,
  RouteId extends string | null = string | null
> = (
  event: LoadEvent<Params, InputData, ParentData, RouteId>
```

The generic form of `PageLoadEvent` and `LayoutLoadEvent`. You should import those from `./$types` (see generated types) rather than using `LoadEvent` directly.

```
interface LoadEvent<
  Params extends Partial<Record<string, string>> = Partial<
    Record<string, string>
  >,
  Data extends Record<string, unknown> | null = Record<
    string,
    any
  > | null,
  ParentData extends Record<string, unknown> = Record<
    string,
    any
  >,
  RouteId extends string | null = string | null
> extends NavigationEvent<Params, RouteId> {…}
```

```
fetch: typeof fetch;
```

`fetch` is equivalent to the native `fetch` web API, with a few additional features:

- It can be used to make credentialed requests on the server, as it inherits the `cookie` and `authorization` headers for the page request.
- It can make relative requests on the server (ordinarily, `fetch` requires a URL with an origin when used in a server context).
- Internal requests (e.g. for `+server.js` routes) go directly to the handler function when running on the server, without the overhead of an HTTP call.
- During server-side rendering, the response will be captured and inlined into the rendered HTML by hooking into the `text` and `json` methods of the `Response` object. Note that headers will *not* be serialized, unless explicitly included via `filterSerializedResponseHeaders`
- During hydration, the response will be read from the HTML, guaranteeing

```
data: Data;
```

Contains the data returned by the route's server `load` function (in `+layout.server.js` or `+page.server.js`), if any.

```
setHeaders(headers: Record<string, string>): void;
```

If you need to set headers for the response, you can do so using the this method. This is useful if you want the page to be cached, for example:

src/routes/blog/+page.js

```
export async function load({ fetch, setHeaders }) {

  const url = `https://cms.example.com/articles.json`;
  const response = await fetch(url);

  setHeaders({
    age: response.headers.get('age'),
    'cache-control': response.headers.get('cache-control')
  });

  return response.json();
}
```

Setting the same header multiple times (even in separate `load` functions) is an error — you can only set a given header once.

You cannot add a `set-cookie` header with `setHeaders` — use the `cookies` API in a server-only `load` function instead.

`setHeaders` has no effect when a `load` function runs in the browser.

```
parent(): Promise<ParentData>;
```

missing `+layout.js` is treated as a `({ data }) => data` function, meaning that it will return and forward data from parent `+layout.server.js` files.

Be careful not to introduce accidental waterfalls when using `await parent()`. If for example you only want to merge parent data into the returned output, call it *after* fetching your other data.

```
depends(...deps: Array<`${string}:${string}`>): void;
```

This function declares that the `load` function has a *dependency* on one or more URLs or custom identifiers, which can subsequently be used with `invalidate()` to cause `load` to rerun.

Most of the time you won't need this, as `fetch` calls `depends` on your behalf — it's only necessary if you're using a custom API client that bypasses `fetch`.

URLs can be absolute or relative to the page being loaded, and must be encoded.

Custom identifiers have to be prefixed with one or more lowercase letters followed by a colon to conform to the URI specification.

The following example shows how to use `depends` to register a dependency on a custom identifier, which is `invalidate` d after a button click, making the `load` function rerun.

src/routes/+page.js

```
let count = 0;
export async function load({ depends }) {
  depends('increase:count');

  return { count: count++ };
}
```

src/routes/+page.svelte

Docs

```
  const increase = async () => {
    await invalidate('increase:count');
  }
</script>

<p>{data.count}<p>
<button on:click={increase}>Increase Count</button>
```

```
untrack<T>(fn: () => T): T;
```

Use this function to opt out of dependency tracking for everything that is synchronously called within the callback. Example:

src/routes/+page.server.js

```
export async function load({ untrack, url }) {

  // Untrack url.pathname so that path changes don't trigger a rerun
  if (untrack(() => url.pathname === '/')) {
    return { message: 'Welcome!' };
  }
}
```

LoadProperties

```
type LoadProperties<
  input extends Record<string, any> | void
> = input extends void
  ? undefined // needs to be undefined, because void will break intellisense
  : input extends Record<string, any>
    ? input
    : unknown;
```

```
from: NavigationTarget | null;
```

Where navigation was triggered from

```
to: NavigationTarget | null;
```

Where navigation is going to/has gone to

```
type: Exclude<NavigationType, 'enter'>;
```

The type of navigation:

- `form` : The user submitted a `<form>`
- `leave` : The app is being left either because the tab is being closed or a navigation to a different document is occurring
- `link` : Navigation was triggered by a link click
- `goto` : Navigation was triggered by a `goto(...)` call or a redirect
- `popstate` : Navigation was triggered by back/forward navigation

```
willUnload: boolean;
```

Whether or not the navigation will result in the page being unloaded (i.e. not a client-side navigation)

```
delta?: number;
```

In case of a history back/forward navigation, the number of steps to go back/forward fails or is aborted. In the case of a `willUnload` navigation, the promise will never resolve

NavigationEvent

```
interface NavigationEvent<
  Params extends Partial<Record<string, string>> = Partial<
    Record<string, string>
  >,
  RouteId extends string | null = string | null
> {…}
```

```
params: Params;
```

The parameters of the current page - e.g. for a route like `/blog/[slug]` , a `{ slug: string }` object

```
route: {…}
```

Info about the current route

```
id: RouteId;
```

The ID of the current route - e.g. for `src/routes/blog/[slug]` , it would be `/blog/[slug]`

```
url: URL;
```

The URL of the current page

```
interface NavigationTarget {…}
```

```
params: Record<string, string> | null;
```

Parameters of the target page - e.g. for a route like `/blog/[slug]`, a `{ slug: string }` object. Is `null` if the target is not part of the SvelteKit app (could not be resolved to a route).

```
route: { id: string | null };
```

Info about the target route

```
url: URL;
```

The URL that is navigated to

NavigationType

- `enter`: The app has hydrated
- `form`: The user submitted a `<form>` with a GET method
- `leave`: The user is leaving the app by closing the tab or using the back/forward buttons to go to a different document
- `link`: Navigation was triggered by a link click
- `goto`: Navigation was triggered by a `goto(...)` call or a redirect
- `popstate`: Navigation was triggered by back/forward navigation

```
type NavigationType =
  | 'goto'
  | 'popstate';
```

NumericRange

```
type NumericRange<
  TStart extends number,
  TEnd extends number
> = Exclude<TEnd | LessThan<TEnd>, LessThan<TStart>>;
```

OnNavigate

The argument passed to `onNavigate` callbacks.

```
interface OnNavigate extends Navigation {…}
```

```
type: Exclude<NavigationType, 'enter' | 'leave'>;
```

The type of navigation:

- `form` : The user submitted a `<form>`
- `link` : Navigation was triggered by a link click
- `goto` : Navigation was triggered by a `goto(...)` call or a redirect
- `popstate` : Navigation was triggered by back/forward navigation

```
willUnload: false;
```

Since `onNavigate` callbacks are called immediately before a client-side navigation, they

The shape of the `$page` store

```
interface Page<
  Params extends Record<string, string> = Record<
    string,
    string
  >,
  RouteId extends string | null = string | null
> {…}
```

```
url: URL;
```

The URL of the current page

```
params: Params;
```

The parameters of the current page - e.g. for a route like `/blog/[slug]`, a `{ slug: string }` object

```
route: {…}
```

Info about the current route

```
id: RouteId;
```

The ID of the current route - e.g. for `src/routes/blog/[slug]`, it would be `/blog/[slug]`

```
status: number;
```

Http status code of the current page

Docs

The error object of the current page, if any. Filled from the `handleError` hooks.

```
data: App.PageData & Record<string, any>;
```

The merged result of all data from all `load` functions on the current page. You can type a common denominator through `App.PageData` .

```
state: App.PageState;
```

The page state, which can be manipulated using the `pushState` and `replaceState` functions from `$app/navigation` .

```
form: any;
```

Filled only after a form submission. See form actions for more info.

ParamMatcher

The shape of a param matcher. See matching for more info.

```
type ParamMatcher = (param: string) => boolean;
```

PrerenderOption

```
type PrerenderOption = boolean | 'auto';
```

```
interface Redirect {…}
```

```
status: 300 | 301 | 302 | 303 | 304 | 305 | 306 | 307 | 308;
```

The HTTP status code, in the range 300-308.

```
location: string;
```

The location to redirect to.

RequestEvent

```
interface RequestEvent<
  Params extends Partial<Record<string, string>> = Partial<
    Record<string, string>
  >,
  RouteId extends string | null = string | null
> {…}
```

```
cookies: Cookies;
```

Get or set cookies related to the current request

```
fetch: typeof fetch;
```

`fetch` is equivalent to the native `fetch` web API, with a few additional features:

It can be used to make credentialed requests on the server, as it inherits the `cookie` and `authorization` headers for the page request.

Docs

when running on the server, without the overhead of an HTTP call.

During server-side rendering, the response will be captured and inlined into the rendered HTML by hooking into the `text` and `json` methods of the `Response` object. Note that headers will *not* be serialized, unless explicitly included via `filterSerializedResponseHeaders`

During hydration, the response will be read from the HTML, guaranteeing consistency and preventing an additional network request.

You can learn more about making credentialed requests with cookies here

```
getClientAddress(): string;
```

The client's IP address, set by the adapter.

```
locals: App.Locals;
```

Contains custom data that was added to the request within the `server handle hook` .

```
params: Params;
```

The parameters of the current route - e.g. for a route like `/blog/[slug]` , a `{ slug: string }` object

```
platform: Readonly<App.Platform> | undefined;
```

Additional data made available through the adapter.

```
request: Request;
```

The original request object

Info about the current route

```
id: RouteId;
```

The ID of the current route - e.g. for `src/routes/blog/[slug]`, it would be `/blog/[slug]`

```
setHeaders(headers: Record<string, string>): void;
```

If you need to set headers for the response, you can do so using the this method. This is useful if you want the page to be cached, for example:

src/routes/blog/+page.js

```
export async function load({ fetch, setHeaders }) {

  const url = `https://cms.example.com/articles.json`;
  const response = await fetch(url);

  setHeaders({
    age: response.headers.get('age'),
    'cache-control': response.headers.get('cache-control')
  });

  return response.json();
}
```

Setting the same header multiple times (even in separate `load` functions) is an error — you can only set a given header once.

You cannot add a `set-cookie` header with `setHeaders` — use the `cookies` API instead.

```
url: URL;
```

Docs

`true` if the request comes from the client asking for `+page/layout.server.js` data. The `url` property will be stripped of the internal information related to the data request in this case. Use this property instead if the distinction is important to you.

```
isSubRequest: boolean;
```

`true` for `+server.js` calls coming from SvelteKit without the overhead of actually making an HTTP request. This happens when you make same-origin `fetch` requests on the server.

RequestHandler

A `(event: RequestEvent) => Response` function exported from a `+server.js` file that corresponds to an HTTP verb (`GET` , `PUT` , `PATCH` , etc) and handles requests with that method.

It receives `Params` as the first generic argument, which you can skip by using generated types instead.

```
type RequestHandler<
  Params extends Partial<Record<string, string>> = Partial<
    Record<string, string>
  >,
  RouteId extends string | null = string | null
> = (
  event: RequestEvent<Params, RouteId>
) => MaybePromise<Response>;
```

Reroute

ResolveOptions

```
interface ResolveOptions {…}
```

```
transformPageChunk?(input: { html: string; done: boolean }): MaybePromise<string | undefi
```

`input` the html chunk and the info if this is the last chunk

Applies custom transforms to HTML. If `done` is true, it's the final chunk. Chunks are not guaranteed to be well-formed HTML (they could include an element's opening tag but not its closing tag, for example) but they will always be split at sensible boundaries such as `%sveltekit.head%` or layout/page components.

```
filterSerializedResponseHeaders?(name: string, value: string): boolean;
```

`name` header name

`value` header value

Determines which headers should be included in serialized responses when a `load` function loads a resource with `fetch`. By default, none will be included.

```
preload?(input: { type: 'font' | 'css' | 'js' | 'asset'; path: string }): boolean;
```

`input` the type of the file and its path

Determines what should be added to the `<head>` tag to preload it. By default, `js` and `css` files will be preloaded.

```
id: string;
```

```
api: {
  methods: Array<HttpMethod | '*'>;
};
```

```
page: {
  methods: Array<Extract<HttpMethod, 'GET' | 'POST'>>;
};
```

```
pattern: RegExp;
```

```
prerender: PrerenderOption;
```

```
segments: RouteSegment[];
```

```
methods: Array<HttpMethod | '*'>;
```

```
config: Config;
```

SSRManifest

```
interface SSRManifest {…}
```

```
appDir: string;
```

```
appPath: string;
```

```
mimeTypes: Record<string, string>;
```

```
_: {…}
```

private fields

```
client: NonNullable<BuildData['client']>;
```

```
nodes: SSRNodeLoader[];
```

```
routes: SSRRoute[];
```

```
matchers(): Promise<Record<string, ParamMatcher>>;
```

```
server_assets: Record<string, number>;
```

A `[file]: size` map of all assets imported by server code

Server

```
class Server {…}
```

```
constructor(manifest: SSRManifest);
```

```
init(options: ServerInitOptions): Promise<void>;
```

```
respond(request: Request, options: RequestOptions): Promise<Response>;
```

Docs

```
interface ServerInitOptions {…}
```

```
env: Record<string, string>;
```

A map of environment variables

```
read?: (file: string) => ReadableStream;
```

A function that turns an asset filename into a `ReadableStream`. Required for the `read` export from `$app/server` to work

ServerLoad

The generic form of `PageServerLoad` and `LayoutServerLoad`. You should import those from `./$types` (see generated types) rather than using `ServerLoad` directly.

```
type ServerLoad<
  Params extends Partial<Record<string, string>> = Partial<
    Record<string, string>
  >,
  ParentData extends Record<string, any> = Record<
    string,
    any
  >,
  OutputData extends Record<string, any> | void = Record<
    string,
    any
  > | void,
  RouteId extends string | null = string | null
> = (
  event: ServerLoadEvent<Params, ParentData, RouteId>
) => MaybePromise<OutputData>;
```

```
interface ServerLoadEvent<
  Params extends Partial<Record<string, string>> = Partial<
    Record<string, string>
  >,
  ParentData extends Record<string, any> = Record<
    string,
    any
  >,
  RouteId extends string | null = string | null
> extends RequestEvent<Params, RouteId> {…}
```

```
parent(): Promise<ParentData>;
```

`await parent()` returns data from parent `+layout.server.js` `load` functions.

Be careful not to introduce accidental waterfalls when using `await parent()`. If for example you only want to merge parent data into the returned output, call it *after* fetching your other data.

```
depends(...deps: string[]): void;
```

This function declares that the `load` function has a *dependency* on one or more URLs or custom identifiers, which can subsequently be used with `invalidate()` to cause `load` to rerun.

Most of the time you won't need this, as `fetch` calls `depends` on your behalf — it's only necessary if you're using a custom API client that bypasses `fetch`.

URLs can be absolute or relative to the page being loaded, and must be encoded.

Custom identifiers have to be prefixed with one or more lowercase letters followed by a colon to conform to the URI specification.

The following example shows how to use `depends` to register a dependency on a

```js
let count = 0;
export async function load({ depends }) {
  depends('increase:count');

  return { count: count++ };
}
```

src/routes/+page.svelte

```svelte
<script>
  import { invalidate } from '$app/navigation';

  let { data } = $props();

  const increase = async () => {
    await invalidate('increase:count');
  }
</script>

<p>{data.count}<p>
<button on:click={increase}>Increase Count</button>
```

```ts
untrack<T>(fn: () => T): T;
```

Use this function to opt out of dependency tracking for everything that is synchronously called within the callback. Example:

src/routes/+page.js

```js
export async function load({ untrack, url }) {

  // Untrack url.pathname so that path changes don't trigger a rerun
  if (untrack(() => url.pathname === '/')) {
    return { message: 'Welcome!' };
  }
}
```

```
interface Snapshot<T = any> {…}
```

```
capture: () => T;
```

```
restore: (snapshot: T) => void;
```

SubmitFunction

```
type SubmitFunction<
  Success extends
    | Record<string, unknown>
    | undefined = Record<string, any>,
  Failure extends
    | Record<string, unknown>
    | undefined = Record<string, any>
> = (input: {
  action: URL;
  formData: FormData;
  formElement: HTMLFormElement;
  controller: AbortController;
  submitter: HTMLElement | null;
  cancel(): void;
}) => MaybePromise<
  | void
  | ((opts: {
      formData: FormData;
      formElement: HTMLFormElement;
      action: URL;
      result: ActionResult<Success, Failure>;
      /**
       * Call this to get the default behavior of a form submission response.
       * @param options Set `reset: false` if you don't want the `<form>` values to be res
       * @param invalidateAll Set `invalidateAll: false` if you don't want the action to c
       */
      update(options?: {
        reset?: boolean;
```

Private types

The following are referenced by the public types documented above, but cannot be imported directly:

AdapterEntry

```ts
interface AdapterEntry {…}
```

```ts
id: string;
```

A string that uniquely identifies an HTTP service (e.g. serverless function) and is used for deduplication. For example, `/foo/a-[b]` and `/foo/[c]` are different routes, but would both be represented in a Netlify _redirects file as `/foo/:param` , so they share an ID

```ts
filter(route: RouteDefinition): boolean;
```

A function that compares the candidate route with the current route to determine if it should be grouped with the current route.

Use cases:

- Fallback pages: `/foo/[c]` is a fallback for `/foo/a-[b]` , and `/[...catchall]` is a fallback for all routes
- Grouping routes that share a common `config` : `/foo` should be deployed to the edge, `/bar` and `/baz` should be deployed to a serverless function

A function that is invoked once the entry has been created. This is where you should write the function to the filesystem and generate redirect manifests.

Csp

```
namespace Csp {
  type ActionSource = 'strict-dynamic' | 'report-sample';
  type BaseSource =
    | 'self'
    | 'unsafe-eval'
    | 'unsafe-hashes'
    | 'unsafe-inline'
    | 'wasm-unsafe-eval'
    | 'none';
  type CryptoSource =
    `${'nonce' | 'sha256' | 'sha384' | 'sha512'}-${string}`;
  type FrameSource =
    | HostSource
    | SchemeSource
    | 'self'
    | 'none';
  type HostNameScheme = `${string}.${string}` | 'localhost';
  type HostSource =
    `${HostProtocolSchemes}${HostNameScheme}${PortScheme}`;
  type HostProtocolSchemes = `${string}://` | '';
  type HttpDelineator = '/' | '?' | '#' | '\\';
  type PortScheme = `:${number}` | '' | ':*';
  type SchemeSource =
    | 'http:'
    | 'https:'
    | 'data:'
    | 'mediastream:'
    | 'blob:'
    | 'filesystem:';
  type Source =
    | HostSource
    | SchemeSource
```

CspDirectives

```ts
interface CspDirectives {…}
```

```ts
'child-src'?: Csp.Sources;
```

```ts
'default-src'?: Array<Csp.Source | Csp.ActionSource>;
```

```ts
'frame-src'?: Csp.Sources;
```

```ts
'worker-src'?: Csp.Sources;
```

```ts
'connect-src'?: Csp.Sources;
```

```ts
'font-src'?: Csp.Sources;
```

```ts
'img-src'?: Csp.Sources;
```

```ts
'manifest-src'?: Csp.Sources;
```

```ts
'media-src'?: Csp.Sources;
```

```ts
'object-src'?: Csp.Sources;
```

```ts
'prefetch-src'?: Csp.Sources;
```

Docs

```
'script-src-elem'?: Csp.Sources;
```

```
'script-src-attr'?: Csp.Sources;
```

```
'style-src'?: Array<Csp.Source | Csp.ActionSource>;
```

```
'style-src-elem'?: Csp.Sources;
```

```
'style-src-attr'?: Csp.Sources;
```

```
'base-uri'?: Array<Csp.Source | Csp.ActionSource>;
```

```
sandbox?: Array<
| 'allow-downloads-without-user-activation'
| 'allow-forms'
| 'allow-modals'
| 'allow-orientation-lock'
| 'allow-pointer-lock'
| 'allow-popups'
| 'allow-popups-to-escape-sandbox'
| 'allow-presentation'
| 'allow-same-origin'
| 'allow-scripts'
| 'allow-storage-access-by-user-activation'
| 'allow-top-navigation'
| 'allow-top-navigation-by-user-activation'
>;
```

```
'form-action'?: Array<Csp.Source | Csp.ActionSource>;
```

```
'frame-ancestors'?: Array<Csp.HostSource | Csp.SchemeSource | Csp.FrameSource>;
```

```
'navigate-to'?: Array<Csp.Source | Csp.ActionSource>;
```

```
'report-to'?: string[];
```

```
'require-trusted-types-for'?: Array<'script'>;
```

```
'trusted-types'?: Array<'none' | 'allow-duplicates' | '*' | string>;
```

```
'upgrade-insecure-requests'?: boolean;
```

```
'require-sri-for'?: Array<'script' | 'style' | 'script style'>;
```

DEPRECATED

```
'block-all-mixed-content'?: boolean;
```

DEPRECATED

```
'plugin-types'?: Array<`${string}/${string}` | 'none'>;
```

DEPRECATED

```
referrer?: Array<
| 'no-referrer'
| 'no-referrer-when-downgrade'
| 'origin'
| 'origin-when-cross-origin'
| 'same-origin'
| 'strict-origin'
| 'strict-origin-when-cross-origin'
| 'unsafe-url'
| 'none'
>;
```

Docs

HttpMethod

```
type HttpMethod =
  | 'GET'
  | 'HEAD'
  | 'POST'
  | 'PUT'
  | 'DELETE'
  | 'PATCH'
  | 'OPTIONS';
```

Logger

```
interface Logger {…}
```

```
(msg: string): void;
```

```
success(msg: string): void;
```

```
error(msg: string): void;
```

```
warn(msg: string): void;
```

```
minor(msg: string): void;
```

```
info(msg: string): void;
```

PrerenderEntryGeneratorMismatchHandler

```
interface PrerenderEntryGeneratorMismatchHandler {…}
```

```
(details: { generatedFromId: string; entry: string; matchedId: string; message: string })
```

PrerenderEntryGeneratorMismatchHandlerValue

```
type PrerenderEntryGeneratorMismatchHandlerValue =
  | 'fail'
  | 'warn'
  | 'ignore'
  | PrerenderEntryGeneratorMismatchHandler;
```

PrerenderHttpErrorHandler

```
interface PrerenderHttpErrorHandler {…}
```

```
(details: {
status: number;
path: string;
referrer: string | null;
referenceType: 'linked' | 'fetched';
message: string;
}): void;
```

```
  | 'fail'
  | 'warn'
  | 'ignore'
  | PrerenderHttpErrorHandler;
```

PrerenderMap

```
type PrerenderMap = Map<string, PrerenderOption>;
```

PrerenderMissingIdHandler

```
interface PrerenderMissingIdHandler {…}
```

```
(details: { path: string; id: string; referrers: string[]; message: string }): void;
```

PrerenderMissingIdHandlerValue

```
type PrerenderMissingIdHandlerValue =
  | 'fail'
  | 'warn'
  | 'ignore'
  | PrerenderMissingIdHandler;
```

PrerenderOption

```
type PrerenderOption = boolean | 'auto';
```

```
interface Prerendered {…}
```

```
pages: Map<
string,
{
  /** The location of the .html file relative to the output directory */
  file: string;
}
>;
```

A map of `path` to `{ file }` objects, where a path like `/foo` corresponds to `foo.html` and a path like `/bar/` corresponds to `bar/index.html` .

```
assets: Map<
string,
{
  /** The MIME type of the asset */
  type: string;
}
>;
```

A map of `path` to `{ type }` objects.

```
redirects: Map<
string,
{
  status: number;
  location: string;
}
>;
```

A map of redirects encountered during prerendering.

```
paths: string[];
```

config)

RequestOptions

```
interface RequestOptions {…}
```

```
getClientAddress(): string;
```

```
platform?: App.Platform;
```

RouteSegment

```
interface RouteSegment {…}
```

```
content: string;
```

```
dynamic: boolean;
```

```
rest: boolean;
```

TrailingSlash

```
type TrailingSlash = 'never' | 'always' | 'ignore';
```