

SVELTEKIT Docs - 日本語



Table of contents

• Getting started - Introduction	4
• Getting started - Creating a project	6
• Getting started - Project structure	7
• Getting started - Web standards	10
• Core concepts - Routing	13
• Core concepts - Load	21
• Core concepts - Form actions	34
• Core concepts - Page options	44
• Core concepts - State management	50
• Build and deploy - Building your app	54
• Build and deploy - Adapters	55
• Build and deploy - Adapter auto	56
• Build and deploy - Adapter node	57
• Build and deploy - Adapter static	62
• Build and deploy - Single page apps	66
• Build and deploy - Adapter cloudflare	68
• Build and deploy - Adapter cloudflare workers	71
• Build and deploy - Adapter netlify	74
• Build and deploy - Adapter vercel	77
• Build and deploy - Writing adapters	81
• Advanced - Advanced routing	83
• Advanced - Hooks	90
• Advanced - Errors	96
• Advanced - Link options	100
• Advanced - Service workers	104
• Advanced - Server only modules	107
• Advanced - Assets	109
• Advanced - Snapshots	110
• Advanced - Packaging	111
• Best practices - Accessibility	117
• Best practices - Seo	120
• Reference - Configuration	123
• Reference - Cli	124
• Reference - Modules	125

• Reference - Types	126
• Appendix - Integrations	129
• Appendix - Migrating	130
• Appendix - Additional resources	135
• Appendix - Glossary	136
• FAQ - Other resources	139
• FAQ - Hmr	140
• FAQ - Read package json	141
• FAQ - Packages	142
• FAQ - Integrations	143

イントロダクション

始める前に

Svelte や SvelteKit が初めてなら、こちらのインタラクティブなチュートリアルをチェックしてみることをおすすめします。

行き詰まつたら、[Discord chatroom](#) でヘルプを求めてください。(日本語翻訳版 追記：上記の Discord は Svelte 本体のもので、英語でコミュニケーションが行われています。もし日本語で質問したり交流したいのであれば、[Svelte 日本の Discord](#) にどうぞ！)

SvelteKit とは

SvelteKit は、[Svelte](#) を使用して堅牢でハイパフォーマンスな web アプリケーションを迅速に開発するためのフレームワークです。もしあなたが React 界隈から来たのであれば、SvelteKit は Next に似ているものです。Vue 界隈から来たのであれば、Nuxt に似ています。

Svelte とは

手短に言えば Svelte は、ナビゲーションバー、コメントセクション、コンタクトフォームなど、ユーザーがブラウザで見たり操作したりするユーザーインターフェースコンポーネントを書く方法です。Svelte コンパイラは、コンポーネントを、ページの HTML をレンダリングする実行可能な JavaScript と、ページのスタイルを CSS に変換します。このガイドの残りの部分を理解するのに Svelte を知っておく必要はありませんが、もし知っていたら役に立つでしょう。より詳しく知りたい場合は、[Svelte のチュートリアル](#) をご覧ください。

SvelteKit vs Svelte

Svelte は UI コンポーネントをレンダリングします。Svelte だけでも、コンポーネントを組み合わせてページ全体をレンダリングすることは可能ですが、アプリ全体を書くには Svelte だけでなく、他のものも必要です。

SvelteKit は、リンクがクリックされたときに UI を更新する [ルーター](#) や、[サーバーサイドレンダリング \(SSR\)](#) といった基本的な機能を提供していますが、それだけではありません。モダンなベストプラクティスを全て取り入れたアプリを構築するのは、恐ろしく複雑なことです。モダンなベストプラクティスとは、必要最小限のコードのみをロードするための [ビルド最適化](#)、[オフラインサポート](#)、[プリロード](#)(ユーザーがナビゲーションを開始する前にページを事前読み込みする)、[柔軟な設定が可能なレンダリング](#)(アプリのある部分は [SSR](#) によってサーバー上でレンダリングさせたり、また別の部分はブラウザで [クライアントサイドレンダリング](#) させたり、また別の部分はビルト時に [プリレンダリング](#) させたりすることが可能)、その他様々な事柄です。SvelteKit が全ての退屈な作業を行ってくれるので、あなたはクリエイティブな作業に専念することができます。

[Svelte plugin](#) で [Vite](#) を動かして [Hot Module Replacement \(HMR\)](#) を行うことで、コードの変更を即座にブラウザに反映し、非常に高速でフィーチャーリッチな開発体験を提供します。

プロジェクトを作成する

SvelteKit アプリの構築を始めるのに最も簡単な方法は `npm create` を実行することです:

```
npm create svelte@latest my-app
cd my-app
npm install
npm run dev
```

最初のコマンドでは、TypeScript などの基本的なツールをセットアップするかどうか選択しながら、`my-app` ディレクトリに新しいプロジェクトを生成します。追加のツールの設定に関するポイントについては FAQ をご覧ください。それ以降のコマンドでは、依存関係をインストールし、localhost:5173 でサーバーを起動します。

SvelteKit には2つの基本的なコンセプトがあります:

- アプリの各ページは [Svelte](#) コンポーネントです
- プロジェクトの `src/routes` ディレクトリにファイルを追加することで、ページを作成できます。これらはサーバーでレンダリングされるのでユーザーの最初のアクセスの際に可能な限り速く表示されるようになり、それからクライアントサイドのアプリに引き継がれます

ファイルを編集して、どのように動作するのか確かめてみてください。

エディタのセットアップ

[Visual Studio Code](#) (通称 VS Code) と [Svelte extension](#) のご使用をおすすめしますが、他にも数多くのエディタをサポートしています。

[Go to TOC](#)

プロジェクト構成

一般的な SvelteKit プロジェクトはこのような構成です:

```
my-project/
├ src/
│ ├ lib/
│ │ ├ server/
│ │ │ └ [your server-only lib files]
│ │ └ [your lib files]
│ ├ params/
│ │ └ [your param matchers]
│ ├ routes/
│ │ └ [your routes]
│ ├ app.html
│ ├ error.html
│ ├ hooks.client.js
│ └ hooks.server.js
├ static/
└ [your static assets]
└ tests/
└ [your tests]
├ package.json
├ svelte.config.js
├ tsconfig.json
└ vite.config.js
```

また、`.gitignore`、`.npmrc` などの共通ファイルもあります（もし `npm create svelte@latest` の実行時にオプションを選択した場合は `.prettierrc` や `.eslintrc.cjs` などもあるでしょう）。

プロジェクトファイル

src

`src` ディレクトリには、プロジェクトの中身が格納されます。`src/routes` と `src/app.html` 以外は全て省略できます。

- `lib` にはあなたのライブラリのコード（ユーティリティやコンポーネント）を格納します。格納されたコードは `$lib` エイリアスを使用してインポートしたり、`svelte-package` を使用して配布用にパッケージングすることができます。
 - `server` にはあなたのサーバー専用のライブラリのコードを格納します。格納されたコードは `$lib/server` エイリアスを使用してインポートすることができます。SvelteKit はこれをクライアントコードにインポートされるのを防ぎます。
- `params` にはアプリに必要な `param matchers` を格納します
- `routes` にはアプリケーションの `ルート(routes)` を格納します。単一のルート(route)でしか使われないコンポーネントをここに置くこともできます
- `app.html` はページのテンプレートで、以下のプレースホルダーを含む HTML document です:
 - `%sveltekit.head%` — アプリに必要な `<link>` 要素や `<script>` 要素、`<svelte:head>` コンテンツ

- `%sveltekit.body%` — レンダリングされるページのためのマークアップです。これを直接 `<body>` の中に置くのではなく、`<div>` または他の要素の中に置く必要があります。ブラウザ拡張(browser extensions)が要素を注入するのをハイドレーションプロセスが破壊してしまう、というバグを防ぐためです。もしこうなっていない場合、SvelteKit は開発中に警告を出します
- `%sveltekit.assets%` — `paths.assets` が指定されている場合は `paths.assets`、指定されていない場合は `paths.base` への相対パス
- `%sveltekit.nonce%` — マニュアルで含めるリンクやスクリプトの CSP nonce (使用する場合)
- `%sveltekit.env.[NAME]%` - これはレンダリング時に環境変数の `[NAME]` に置き換えられます。この環境変数は `publicPrefix` で始まる必要があります (通常は `PUBLIC_` です)。もしマッチしない場合は '' にフォールバックします。
- `error.html` は、全てが失敗したときにレンダリングされるページです。以下のプレースホルダーを含めることができます:
 - `%sveltekit.status%` — HTTP ステータス
 - `%sveltekit.error.message%` — エラーメッセージ
- `hooks.client.js` にはクライアントの `hooks` を記述します
- `hooks.server.js` にはサーバーの `hooks` を記述します
- `service-worker.js` には `service worker` を記述します

(プロジェクトに `.js` と `.ts` のどちらのファイルが含まれるかについては、プロジェクトの作成時に TypeScript の使用を選択したかどうかによります。このドキュメントに記載されているコードやファイル拡張子などは、ページの下にあるトグルを使って、JavaScript と TypeScript を切り替えることができます。)

プロジェクトのセットアップ時に `Vitest` を追加した場合、ユニットテストは `.test.js` という拡張子で `src` ディレクトリに置かれます。

static

`robots.txt` や `favicon.png` など、そのままサーブされる静的なアセットをここに含めます。

tests

プロジェクトのセットアップ時、ブラウザテストのために `Playwright` を追加した場合、そのテストはこのディレクトリに置かれます。

package.json

`package.json` ファイルには `@sveltejs/kit`、`svelte`、`vite` が `devDependencies` に含まれていなければなりません。

`npm create svelte@latest` でプロジェクトを作成すると、`package.json` には `"type": "module"` が含まれることに気が付くでしょう。これは、`.js` ファイルが `import` や `export` キーワードを持つネイティブの JavaScript モジュールとして解釈されることを意味します。レガシーな CommonJS ファイルには `.cjs` ファイル拡張子が必要です。

svelte.config.js

このファイルには Svelte と SvelteKit の設定が含まれています。

tsconfig.json

`npm create svelte@latest` の際に型チェックを追加した場合、このファイル（または `.ts` ファイルより型チェックされた `.js` ファイルのほうが好みであれば `jsconfig.json`）で TypeScript の設定を行います。SvelteKit は特定の設定に依存しているため、独自の `.svelte-kit/tsconfig.json` ファイルを生成し、あなたの設定を `extends`（拡張）しています。

vite.config.js

SvelteKit プロジェクトは実は、`@sveltejs/kit/vite` プラグインと、その他の Vite の設定 を一緒に使用した Vite プロジェクトです。

その他のファイル

.svelte-kit

開発してプロジェクトをビルドすると、SvelteKit は `.svelte-kit` ディレクトリ（`outDir` で変更可能）にファイルを生成します。その中身を気にすることなく、いつでも削除することができます（次に `dev` や `build` を実行したときに再生成されます）。

Web standards

このドキュメントを通じて、SvelteKit の土台となっている標準の [Web API](#) を参照することができます。私たちは車輪の再発明をするのではなく、プラットフォームを使用します。つまり、既存の Web 開発スキルが SvelteKit にも活用できるということです。逆に言えば、SvelteKit の学習に時間を割くことは、あなたが他の場所でも通用する良い Web 開発者になるのに役立つでしょう。

これらの API は、全てのモダンブラウザはもちろん、Cloudflare Workers、Deno、Vercel Edge Functions といったブラウザ以外の環境でも使用することができます。開発中や、(AWS Lambda を含む) Node ベースの環境向けの [adapters](#) では、必要に応じて `polyfill` で利用できるようにしています（現時点においては、Node は急速により多くの Web 標準のサポートを追加しています）。

具体的には、以下のことが楽にできるでしょう：

Fetch APIs

SvelteKit は、ネットワーク越しにデータを取得するために `fetch` を使用します。ブラウザだけでなく、[hooks](#) や [サーバールート\(server routes\)](#) の中でも使用することができます。

`load` 関数、[server hooks](#)、[API routes](#) の中では特別なバージョンの `fetch` を使用することができ、サーバーサイドレンダリング中に、HTTP をコールすることなく、クレデンシャルを保持したまま、直接エンドポイント（endpoints）を呼び出すことができます。（`load` の外側のサーバーサイドコードでクレデンシャル付きの `fetch` を行う場合は、明示的に `cookie` や `authorization` ヘッダーなどを渡さなければなりません。）また、通常のサーバーサイドの `fetch` では絶対パスの URL が必要となりますが、特別なバージョンの `fetch` では相対パスのリクエストが可能です。

`fetch` 自体の他に、[Fetch API](#) には以下のインターフェイスが含まれています：

Request

`Request` のインスタンスは [hooks](#) や [サーバールート\(server routes\)](#) で `event.request` という形でアクセスすることができます。これには `request.json()` や `request.formData()` など、エンドポイントに送られたデータを取得するための便利なメソッドが含まれています。

Response

`Response` のインスタンスは `await fetch(...)` と `+server.js` ファイル内のハンドラーから返されます。本質的には、SvelteKit アプリは `Request` を `Response` に変換するマシンです。

Headers

`Headers` インターフェイスでは、受け取った `request.headers` を読み取り、送信する `response.headers` をセットすることができます。例えば以下のように、`request.headers` を取得して、`json` という便利な関数を使用して `response.headers` を変更し送信することができます：

```
// @errors: 2461
/// file: src/routes/what-is-my-user-agent/+server.js
import { json } from '@sveltejs/kit';

/** @type {import('./$types').RequestHandler} */
export function GET({ request }) {
    // log all headers
    console.log(...request.headers);

    // create a JSON Response using a header we received
    return json({
        // retrieve a specific header
        userAgent: request.headers.get('user-agent')
    }, {
        // set a header on the response
        headers: { 'x-custom-header': 'potato' }
    });
}
```

FormData

HTML のネイティブのフォーム送信を扱う場合は、`FormData` オブジェクトを使用します。

```
// @errors: 2461
/// file: src/routes/hello/+server.js
import { json } from '@sveltejs/kit';

/** @type {import('./$types').RequestHandler} */
export async function POST(event) {
    const body = await event.request.formData();

    // log all fields
    console.log([...body]);

    return json({
        // get a specific field's value
        name: body.get('name') ?? 'world'
    });
}
```

Stream APIs

ほとんどの場合、エンドポイント(endpoints) は 上記の `userAgent` の例のように、完全なデータを返します。たまに、1度ではメモリに収まらない大きすぎるレスポンスを返したり、チャunkで配信したりしなければならないことがあります。このような場合のために、プラットフォームは `streams` — `ReadableStream`、`WritableStream`、`TransformStream` を提供しています。

URL APIs

URL は `URL` インターフェイスで表現され、`origin` や `pathname` のような便利なプロパティが含まれています（ブラウザでは `hash` なども）。このインターフェイスは、`hooks` と サーバールート(server routes) では `event.url`、ページ(pages) では `$page.url`、`beforeNavigate` と `afterNavigate` では `from` と `to`、など、様々な場所で使われています。

URLSearchParams

URL が存在する場所であれば、`URLSearchParams` のインスタンスである `url.searchParams` を使用してクエリパラメータにアクセスできます：

```
// @filename: ambient.d.ts
declare global {
    const url: URL;
}

export {};

// @filename: index.js
// cut---
const foo = url.searchParams.get('foo');
```

Web Crypto

Web Crypto API を、グローバルの `crypto` 経由で使用することができます。内部では Content Security Policy ヘッダーで使用されていますが、例えば UUID を生成するのにもお使い頂けます。

```
const uuid = crypto.randomUUID();
```

ルーティング

SvelteKit の中心は、 ファイルシステムベースのルーター です。アプリのルート(routes) — 例えばユーザーがアクセスできる URL パス — は、コードベースのディレクトリによって定義されます:

- `src/routes` は最上位のルート(the root route)です
- `src/routes/about` は `/about` ルート(route)を作成します
- `src/routes/blog/[slug]` は パラメータ `slug` を使ったルート(route)を作成します。パラメータは、ユーザーからのリクエストが `/blog/hello-world` のようなページに行われた場合に、動的にデータを読み込むために使用することができます

[プロジェクトの設定](#) を編集することで、 `src/routes` から別のディレクトリに変更することができます。

ルート(route)のディレクトリはそれぞれ1つ以上の ルートファイル(route files) を格納します。ルートファイル(route files)には `+` という接頭辞が付いているので、それで見分けることができます。

+page

+page.svelte

`+page.svelte` コンポーネントはアプリのページを定義します。デフォルトでは、ページは最初のリクエストではサーバー([SSR](#))でレンダリングされ、その後のナビゲーションではブラウザ([CSR](#))でレンダリングされます。

```
/// file: src/routes/+page.svelte
<h1>Hello and welcome to my site!</h1>
<a href="/about">About my site</a>
```

```
/// file: src/routes/about/+page.svelte
<h1>About this site</h1>
<p>TODO...</p>
<a href="/">Home</a>
```

```
/// file: src/routes/blog/[slug]/+page.svelte
<script>
  /** @type {import('./$types').PageData} */
  export let data;
</script>

<h1>{data.title}</h1>
<div>{@html data.content}</div>
```

SvelteKit では、ルート(routes)間のナビゲーションに、フレームワーク固有の `<Link>` コンポーネントではなく、`<a>` 要素を使用します。

+page.js

ページではたびたび、レンダリングの前になんらかのデータを読み込む必要があります。これに対応するため、`load` 関数をエクスポートする `+page.js` モジュールを追加しています：

```
/// file: src/routes/blog/[slug]/+page.js
import { error } from '@sveltejs/kit';

/** @type {import('./$types').PageLoad} */
export function load({ params }) {
    if (params.slug === 'hello-world') {
        return {
            # 'Hello world!',
            content: 'Welcome to our blog. Lorem ipsum dolor sit amet...'
        };
    }

    throw error(404, 'Not found');
}
```

この関数は `+page.svelte` とともに実行されます。サーバーサイドレンダリング中はサーバーで実行され、クライアントサイドナビゲーション中はブラウザで実行されます。API の詳細は `load` をご参照ください。

`+page.js` では、`load` だけでなくページの動作(behaviour)を設定するための値をエクスポートすることができます：

- `export const prerender = true` または `false` または `'auto'`
- `export const ssr = true` または `false`
- `export const csr = true` または `false`

これらに関するより詳しい情報は [page options](#) をご覧ください。

+page.server.js

`load` 関数をサーバー上でのみ実行できるようにしたい場合 — 例えば、データベースからデータを取得したり、API キーのようなプライベートな環境変数にアクセスしたりする必要がある場合 — `+page.js` を `+page.server.js` にリネームし、`PageLoad` 型を `PageServerLoad` に変更します。

```
/// file: src/routes/blog/[slug]/+page.server.js

// @filename: ambient.d.ts
declare global {
    const getPostFromDatabase: (slug: string) => {
        # string;
        content: string;
    }
}

export {};

// @filename: index.js
// cut---
import { error } from '@sveltejs/kit';

/** @type {import('./$types').PageServerLoad} */
```

```

export async function load({ params }) {
  const post = await getPostFromDatabase(params.slug);

  if (post) {
    return post;
  }

  throw error(404, 'Not found');
}

```

クライアントサイドナビゲーション中は、SvelteKit はサーバーからこのデータを読み込みます。つまり、その戻り値は `devalue` によってシリアル化できなければならないということです。この API の詳細については `load` をご参照ください。

`+page.js` のように、`+page.server.js` は `page options` (`prerender`、`ssr`、`csr`) をエクスポートできます。

また、`+page.server.js` ファイルは `actions` をエクスポートできます。`load` がサーバーからデータを読み取る場合、`actions` は `<form>` 要素を使用してサーバーにデータを書き込むことができます。これらの使い方を学ぶには、`form actions` セクションをご参照ください。

+error

`load` 中にエラーが発生した場合、SvelteKit はデフォルトのエラーページをレンダリングします。`+error.svelte` を追加することで、ルート(route) ごとにエラーページをカスタマイズすることができます:

```

/// file: src/routes/blog/[slug]/+error.svelte
<script>
  import { page } from '$app/stores';
</script>

<h1>{$page.status}: {$page.error.message}</h1>

```

SvelteKit は、ツリーを上がって (walk up the tree) 最も近いエラー境界 (error boundary) を探します — もし上記のファイルが存在しない場合は、デフォルトのエラーページをレンダリングする前に `src/routes/blog/+error.svelte` を探しに行き、その次に `src/routes/+error.svelte` を探します。もしそれも失敗した場合は (または、最上位の `+error` の'上に' 位置する最上位の `+layout` の `load` 関数からエラーがスローされた場合)、SvelteKit は静的なフォールバックエラーページをレンダリングします。これは `src/error.html` ファイルを作成することでカスタマイズ可能です。

`+layout(.server).js` の `load` 関数の内側でエラーが発生した場合、ツリーの中で最も近くにあるエラー境界はそのレイアウトの上位にある `+error.svelte` ファイルです (隣ではありません)。

ルート(route)が見つからない場合 (404)、`src/routes/+error.svelte` (または、もしこのファイルが存在しない場合はデフォルトのエラーページ) が使われます。

エラーが `handle` の内側や `+server.js` リクエストハンドラ の内側で発生した場合は、`+error.svelte` は使用されません。

エラーハンドリングに関する詳細は [こちら](#) からお読み頂けます。

+layout

これまで、ページを完全に独立したコンポーネントとして扱ってきました — ナビゲーションを行うと、既存の `+page.svelte` コンポーネントが破棄され、新しいページコンポーネントで置き換えられます。

しかし多くのアプリでは、トップレベルのナビゲーションやフッターのように 全ての ページで表示されるべき要素があります。全ての `+page.svelte` にそれを繰り返し配置する代わりに、レイアウト(*layouts*) に配置することができます。

+layout.svelte

全てのページに適用するレイアウトを作成するには、`src/routes/+layout.svelte` というファイルを作成します。デフォルトのレイアウト (あなたが作成していない場合に SvelteKit が使用するもの) は以下のようなものです…

```
<slot></slot>
```

…しかし、お望みのマークアップ(markup)、スタイル(styles)、動作(behaviour)を追加することができます。唯一の要求事項は、コンポーネントにページコンテンツのための `<slot>` を含めることです。例えば、nav bar を追加してみましょう:

```
/// file: src/routes/+layout.svelte
<nav>
  <a href="/">Home</a>
  <a href="/about">About</a>
  <a href="/settings">Settings</a>
</nav>

<slot></slot>
```

/、/about、/settings のためのページを作成する場合…

```
/// file: src/routes/+page.svelte
<h1>Home</h1>
```

```
/// file: src/routes/about/+page.svelte
<h1>About</h1>
```

```
/// file: src/routes/settings/+page.svelte
<h1>Settings</h1>
```

… nav は常に表示され、3つのページのための a 要素をそれぞれクリックしても `<h1>` が置き換わるだけです。

レイアウトは ネスト させることができます。例えば、単一の `/settings` ページだけでなく、`/settings/profile` や `/settings/notifications` のような共有のサブメニューを持つネストしたページがあるとします (実例としては、[github.com/settings](#) をご参照ください)。

`/settings` 配下のページにのみ適用されるレイアウトを作成することができます (トップレベルの nav を持つ最上位のレイアウト(root layout)を継承しています):

```
/// file: src/routes/settings/+layout.svelte
<script>
  /** @type {import('./$types').LayoutData} */
```

```

    export let data;
</script>

<h1>Settings</h1>

<div class="submenu">
  {#each data.sections as section}
    <a href="/settings/{section.slug}">{section.title}</a>
  {/each}
</div>

<slot></slot>

```

デフォルトでは、各レイアウトはその上にあるレイアウトを継承します。そうしたくない場合は、[advanced layouts](#) が役に立つでしょう。

+layout.js

`+page.svelte` が `+page.js` からデータを読み込むように、`+layout.svelte` コンポーネントは `+layout.js` の `load` 関数からデータを取得することができます。

```

/// file: src/routes/settings/+layout.js
/** @type {import('./$types').LayoutLoad} */
export function load() {
  return {
    sections: [
      { slug: 'profile', # 'Profile' },
      { slug: 'notifications', # 'Notifications' }
    ]
  };
}

```

`+layout.js` が [page options](#) (`prerender`、`ssr`、`csr`) をエクスポートする場合、それは子ページのデフォルトとしても使用されます。

レイアウトの `load` 関数から返されるデータは全ての子ページで利用することができます:

```

/// file: src/routes/settings/profile/+page.svelte
<script>
  /** @type {import('./$types').PageData} */
  export let data;

  console.log(data.sections); // [{ slug: 'profile', # 'Profile' }, ...]
</script>

```

しばしば、ページ間を移動しているときにレイアウトデータが変更されないことがあります。SvelteKit は必要に応じてインテリジェントに `load` 関数を再実行します。

+layout.server.js

サーバー上でレイアウトの `load` 関数を実行するためには、それを `+layout.server.js` に移動し、`LayoutLoad` 型を `LayoutServerLoad` に変更します。

`+layout.js` と同様に、`+layout.server.js` では `page options` — `prerender`、`ssr`、`csr` をエクスポートすることができます。

+server

ページと同様に、`+server.js` ファイル (よく 'API ルート(API route)' または 'エンドポイント(endpoint)' とも呼ばれる) でルート(routes) を定義でき、これによってレスポンスを完全にコントロールすることができます。`+server.js` ファイルは `GET`、`POST`、`PATCH`、`PUT`、`DELETE`、`OPTIONS`、`HEAD` といった HTTP verbs に対応する関数をエクスポートします。これは `RequestEvent` を引数に取り、`Response` オブジェクトを返します。

例えば、`GET` ハンドラーを使用した `/api/random-number` ルート(route)を作成できます:

```
/// file: src/routes/api/random-number/+server.js
import { error } from '@sveltejs/kit';

/** @type {import('./$types').RequestHandler} */
export function GET({ url }) {
    const min = Number(url.searchParams.get('min') ?? '0');
    const max = Number(url.searchParams.get('max') ?? '1');

    const d = max - min;

    if (isNaN(d) || d < 0) {
        throw error(400, 'min and max must be numbers, and min must be less than max');
    }

    const random = min + Math.random() * d;
    return new Response(String(random));
}
```

`Response` の第一引数には `ReadableStream` を指定することができ、大量のデータをストリームしたり、server-sent events を作成したりすることができます (AWS Lambda のような、レスポンスをバッファするプラットフォームにデプロイする場合は除きます)。

便宜上、`@sveltejs/kit` の `error`、`redirect`、`json` メソッドを使用することは可能です (ただし、使用する必要はありません)。

エラーがスローされる場合 (`throw error(...)` によるスローや、予期せぬエラーがスローされるどちらでも)、レスポンスは `Accept` ヘッダーに応じて、そのエラーの JSON 表現か、`src/error.html` でカスタマイズすることができるフォールバックエラーページとなります。この場合、`+error.svelte` コンポーネントはレンダリングされません。エラーハンドリングに関する詳細は [こちら](#) からお読み頂けます。

OPTIONS ハンドラを作成する場合、Vite が `Access-Control-Allow-Origin` ヘッダーと `Access-Control-Allow-Methods` ヘッダーを注入することにご注意ください。本番環境では、あなたが明示的に追加しない限り注入されないはずです。

Receiving data

`+server.js` ファイルは、`POST / PUT / PATCH / DELETE / OPTIONS / HEAD` ハンドラをエクスポートすることで、完全な API を作成することができます：

```
/// file: src/routes/add/+page.svelte
<script>
  let a = 0;
  let b = 0;
  let total = 0;

  async function add() {
    const response = await fetch('/api/add', {
      method: 'POST',
      body: JSON.stringify({ a, b }),
      headers: {
        'content-type': 'application/json'
      }
    });

    total = await response.json();
  }
</script>

<input type="number" bind:value={a}> +
<input type="number" bind:value={b}> =
{total}

<button on:click={add}>Calculate</button>
```

```
/// file: src/routes/api/add/+server.js
import { json } from '@sveltejs/kit';

/** @type {import('./$types').RequestHandler} */
export async function POST({ request }) {
  const { a, b } = await request.json();
  return json(a + b);
}
```

一般的には、ブラウザからサーバーにデータを送信する方法としては [form actions](#) のほうがより良い方法です。

Content negotiation

`+server.js` ファイルは `+page` ファイルと同じディレクトリに置くことができ、これによって同じルート(route)がページにも API エンドポイントにもなるようにすることができます。これがどちらなのか判断するために、SvelteKit は以下のルールを適用します：

- `PUT / PATCH / DELETE / OPTIONS` リクエストは、ページには適用されないため、常に `+server.js` で処理されます。
- `GET / POST / HEAD` リクエストは、`accept` ヘッダーが `text/html` を優先している場合（言い換えると、ブラウザのページリクエストの場合）、ページリクエストとして扱われます。それ以外の場合は `+server.js` で処理されます。
- `GET` リクエストに対するレスポンスには `Vary: Accept` ヘッダーが含まれるため、プロキシーやブラウザは HTML と JSON のレスポンスを別々にキャッシュします。

\$types

これまでの例を通してずっと、`$types.d.ts` ファイルからインポートしてきました。これは、TypeScript（または JavaScript を JSDoc の型アノテーションと）使用している場合に最上位のファイル(root files)を扱う際に型の安全性をもたらすために SvelteKit が隠しディレクトリに作成するファイルです。

例えば、`export let data` に `PageData`（または `LayoutData` の場合は `+layout.svelte` ファイル）にアノテーションを付けると、`data` の型は `load` の戻り値であると TypeScript に伝えることができます：

```
/// file: src/routes/blog/[slug]/+page.svelte
<script>
  /** @type {import('./$types').PageData} */
  export let data;
</script>
```

`load` 関数に `PageLoad`、`PageServerLoad`、`LayoutLoad`、`LayoutServerLoad`（それぞれ `+page.js`、`+page.server.js`、`+layout.js`、`+layout.server.js`）というアノテーションを付けると、`params` と戻り値が正しく型付けされることが保証されるでしょう。

VS Code や、language server protocol と TypeScript plugin をサポートする IDE を使用している場合は、これらの型を完全に省略することができます！Svelte の IDE ツール類があなたのために正しい型を挿入してくれるので、あなたはご自身で型を書くことなく型チェックすることができます。これはコマンドラインツール `svelte-check` でも機能します。

`$types` の省略については、私たちの[ブログ記事](#)でより詳細な情報をお読み頂けます。

その他のファイル

ルート(route)ディレクトリ内のその他のファイルは SvelteKit から無視されます。つまり、コンポーネントやユーティティモジュールを、それらを必要とするルート(routes)に配置することができます。

コンポーネントやモジュールが複数のルート(routes)から必要な場合、`$lib` にそれらを配置すると良いでしょう。

その他の参考資料

- [Tutorial: Routing](#)
- [Tutorial: API routes](#)
- [Docs: Advanced routing](#)

Loading data

`+page.svelte` コンポーネント (と `+layout.svelte` コンポーネント) をレンダリングする前に、データを取得する必要があるケースが多いでしょう。`load` 関数を定義することでこれができるようになります。

Page data

`+page.svelte` ファイルは、`load` 関数をエクスポートする `+page.js` という兄弟ファイルを持つことができ、`load` 関数の戻り値は `page` で `data` プロパティを介して使用することができます。

```
/// file: src/routes/blog/[slug]/+page.js
/** @type {import('./$types').PageLoad} */
export function load({ params }) {
  return {
    post: {
      # `Title for ${params.slug} goes here`,
      content: `Content for ${params.slug} goes here`
    }
  };
}
```

```
/// file: src/routes/blog/[slug]/+page.svelte
<script>
  /** @type {import('./$types').PageData} */
  export let data;
</script>

<h1>{data.post.title}</h1>
<div>{@html data.post.content}</div>
```

生成される `$types` モジュールのおかげで、完全な型安全性を確保できます。

`+page.js` ファイルの `load` 関数はサーバーでもブラウザでも実行されます (ただし、`export const ssr = false` を設定した場合は [ブラウザでのみ実行されます](#))。`load` 関数を常にサーバーで実行させたければ (例えばプライベートな環境変数を使用していたり、データベースにアクセスする場合など)、代わりに `+page.server.js` に `load` 関数を置くとよいでしょう。

例えばブログ記事の `load` 関数をより現実的なものにするとしたら、以下のように、サーバー上でのみ実行され、データベースからデータを取得する、というものになるでしょう。

```
/// file: src/routes/blog/[slug]/+page.server.js
// @filename: ambient.d.ts
declare module '$lib/server/database' {
  export function getPost(slug: string): Promise<{ # string, content: string }>
}

// @filename: index.js
// cut---
import * as db from '$lib/server/database';

/** @type {import('./$types').PageServerLoad} */
export async function load({ params }) {
```

```

    return {
      post: await db.getPost(params.slug)
    };
}

```

型が `PageLoad` から `PageServerLoad` に変わっていることにご注意ください。server `load` 関数では追加の引数にアクセスすることができます。どのような場合に `+page.js` を使用し、どのような場合に `+page.server.js` を使用するのかを理解するには、[Universal vs server](#) を参照してください。

Layout data

`+layout.svelte` ファイルでも、`+layout.js` や `+layout.server.js` を通じてデータを読み込むことができます。

```

/// file: src/routes/blog/[slug]/+layout.server.js
// @filename: ambient.d.ts
declare module '$lib/server/database' {
  export function getPostSummaries(): Promise<Array<{ # string, slug: string }>>
}

// @filename: index.js
// cut---
import * as db from '$lib/server/database';

/** @type {import('./$types').LayoutServerLoad} */
export async function load() {
  return {
    posts: await db.getPostSummaries()
  };
}

/// file: src/routes/blog/[slug]/+layout.svelte
<script>
  /** @type {import('./$types').LayoutData} */
  export let data;
</script>

<main>
  <!-- +page.svelte is rendered in this <slot> -->
  <slot />
</main>

<aside>
  <h2>More posts</h2>
  <ul>
    {#each data.posts as post}
      <li>
        <a href="/blog/{post.slug}">
          {post.title}
        </a>
      </li>
    {/each}
  </ul>
</aside>

```

レイアウトの `load` 関数から返されたデータは、子の `+layout.svelte` コンポーネントやそのレイアウトに属する `+page.svelte` コンポーネントでも利用可能です。

```
/// file: src/routes/blog/[slug]/+page.svelte
<script>
+ import { page } from '$app/stores';

/** @type {import('./$types').PageData} */
export let data;

+ // we can access `data.posts` because it's returned from
+ // the parent layout `load` function
+ $: index = data.posts.findIndex(post => post.slug === $page.params.slug);
+ $: next = data.posts[index - 1];
</script>

<h1>{data.post.title}</h1>
<div>{@html data.post.content}</div>

+{#if next}
+ <p>Next post: <a href="/blog/{next.slug}">{next.title}</a></p>
+{/if}
```

複数の `load` 関数が同じキーを持つデータを返した場合、最後に返したものが「勝ちます」。レイアウトの `load` が `{ a: 1, b: 2 }` を返し、ページの `load` が `{ b: 3, c: 4 }` を返すと、結果は `{ a: 1, b: 3, c: 4 }` となります。

\$page.data

`+page.svelte` コンポーネントとその上の各 `+layout.svelte` コンポーネントは、自身のデータとその親の全てのデータにアクセスすることができます。

場合によっては、その逆も必要かもしれません — 親レイアウトからページのデータや子レイアウトのデータにアクセスする必要があるかもしれません。例えば、最上位のレイアウト(root layout)から、`+page.js` や `+page.server.js` の `load` 関数から返された `title` プロパティにアクセスしたい場合があるでしょう。これは `$page.data` で行うことができます:

```
/// file: src/routes/+layout.svelte
<script>
  import { page } from '$app/stores';
</script>

<svelte:head>
  <title>{$page.data.title}</title>
</svelte:head>
```

`$page.data` の型情報は `App.PageData` から提供されます。

Universal vs server

これまで見てきたように、`load` 関数には2つの種類があります:

- `+page.js` ファイルと `+layout.js` ファイルは、サーバーとブラウザの両方で実行される universal `load` 関数をエクスポートします

- `+page.server.js` ファイルと `+layout.server.js` ファイルは、サーバーサイドでのみ実行される server `load` 関数をエクスポートします

概念上は同じものですが、気をつけなければならない重要な違いがあります。

いつ、どの `load` 関数が実行されるのか？

server `load` 関数は 常に サーバーで実行されます。

デフォルトでは、universal `load` 関数は、ユーザーがはじめてページにアクセスしたときの SSR 中にサーバーで実行されます。そのあとのハイドレーション中に、[fetch リクエスト](#) で取得したレスポンスを再利用してもう一度実行されます。それ以降の universal `load` 関数の呼び出しは、すべてブラウザで行われます。この動作は [page options](#) によってカスタマイズすることができます。[サーバーサイドレンダリング](#) を無効にした場合、SPA となるため、universal `load` 関数は 常に クライアントで実行されるようになります。

`load` 関数は実行時に呼び出されます。ページを [プリレンダリング](#) する場合は、ビルト時に呼び出されます。

Input

universal `load` 関数と server `load` 関数はどちらも、リクエストを表すプロパティ (`params`、`route`、`url`) と様々な関数 (`fetch`、`setHeaders`、`parent`、`depends`) にアクセスできます。これらについては、以下のセクションで説明します。

server `load` 関数は `ServerLoadEvent` を引数にとって呼び出されます。`ServerLoadEvent` は、`RequestEvent` から `clientAddress`、`cookies`、`locals`、`platform`、`request` を継承しています。

universal `load` 関数は、`LoadEvent` を引数にとって呼び出されます。`LoadEvent` は `data` プロパティを持っています。もし `+page.js` と `+page.server.js` (または `+layout.js` と `+layout.server.js`) の両方に `load` 関数がある場合、server `load` 関数の戻り値が、universal `load` 関数の引数の `data` プロパティとなります。

Output

universal `load` 関数は、任意の値(カスタムクラスやコンポーネントコンストラクタなどを含む)を含むオブジェクトを返すことができます。

server `load` 関数は、ネットワークで転送できるようにするために、[devalue](#) でシリализできるデータ (つまり JSON で表現できるものに加え、`BigInt`、`Date`、`Map`、`Set`、`RegExp` や、繰り返し/循環参照など) を返さなければなりません。データには [promises](#) を含めることができます、その場合はブラウザにストリーミングされます。

どちらを使用すべきか

server `load` 関数は、データベースやファイルシステムからデータを直接アクセスする必要がある場合や、プライベートな環境変数を使用する必要がある場合に有用です。

universal `load` 関数は、外部の API から データを `fetch` (取得) する必要があり、プライベートなクレデンシャルが必要ない場合に便利です。なぜなら、SvelteKit はあなたのサーバーを経由せずに、その API から直接データを取得することができるからです。また、Svelte コンポーネントコンストラクタのような、シリアル化できないものを返す必要がある場合にも便利です。

まれに、両方を同時に使用する必要がある場合もあります。例えば、サーバーからのデータで初期化されたカスタムクラスのインスタンスを返す必要がある場合です。

URL data を使用する

多くの場合、`load` 関数は何らかの形で URL に依存します。そのため、`load` 関数では `url`、`route`、`params` を提供しています。

url

`URL` のインスタンスで、`origin`、`hostname`、`pathname`、`searchParams` (`URLSearchParams` オブジェクトとしてパースされたクエリ文字列を含む) を含んでいます。`url.hash` はサーバーで利用できないため、`load` 中にアクセスすることはできません。

環境によっては、サーバーサイドレンダリング時のリクエストヘッダからこれが導出されることもあります。例えば [adapter-node](#) では、URL を正しく設定するために adapter の設定をする必要があるかもしれません。

route

現在のルート(route)ディレクトリの名前を含んでいます。`src/routes` との相対です:

```
/// file: src/routes/a/[b]/[...c]/+page.js
/** @type {import('./$types').PageLoad} */
export function load({ route }) {
    console.log(route.id); // '/a/[b]/[...c]'
```

params

`params` は `url.pathname` と `route.id` から導出されます。

`route.id` が `/a/[b]/[...c]` で、`url.pathname` が `/a/x/y/z` の場合、`params` オブジェクトはこのようになります:

```
{
    "b": "x",
    "c": "y/z"
}
```

fetch リクエストの作成

外部の API や `+server.js` ハンドラからデータを取得するには、提供されている `fetch` 関数を使用します。これは [ネイティブ](#) の `fetch` [web API](#) と同じように動作しますが、いくつか追加の機能があります：

- ページリクエストの `cookie` と `authorization` ヘッダーを継承するので、サーバー上でクレデンシャル付きのリクエストを行うことができます
- サーバー上で、相対パスのリクエストを行うことができます（通常、`fetch` はサーバーのコンテキストで使用する場合にはオリジン付きの URL が必要です）
- サーバーで動作している場合、内部リクエスト（例えば `+server.js` ルート(routes)に対するリクエスト）は直接ハンドラ関数を呼び出すので、HTTP を呼び出すオーバーヘッドがありません
- サーバーサイドレンダリング中は、`Response` オブジェクトの `text` メソッドと `json` メソッドにフックすることにより、レスポンスはキャプチャされ、レンダリング済の HTML にインライン化されます。ヘッダーは、`filterSerializedResponseHeaders` で明示的に指定されない限り、シリアル化されないことにご注意ください。そして、ハイドレーション中は、レスポンスは HTML から読み込まれるため、一貫性が保証され、追加のネットワークリクエストを防ぎます。もし、`load` 関数の `fetch` ではなくブラウザの `fetch` を使用しているときにブラウザコンソールに警告が出た場合は、これが理由です。

```
/// file: src/routes/items/[id]/+page.js
/** @type {import('./$types').PageLoad} */
export async function load({ fetch, params }) {
  const res = await fetch(`/api/items/${params.id}`);
  const item = await res.json();

  return { item };
}
```

Cookie は、ターゲットホストが Sveltekit アプリケーションと同じか、より明確・詳細(specific)なサブドメインである場合にのみ引き渡されます。

Cookies and headers

`server load` 関数では `cookies` を取得したり設定したりすることができます。

```
/// file: src/routes/+layout.server.js
// @filename: ambient.d.ts
declare module '$lib/server/database' {
  export function getUser(sessionid: string | undefined): Promise<{ name: string, avatar: string }>
}

// @filename: index.js
// cut---
import * as db from '$lib/server/database';

/** @type {import('./$types').LayoutServerLoad} */
export async function load({ cookies }) {
  const sessionid = cookies.get('sessionid');
```

```

    return {
      user: await db.getUser(sessionid)
    };
}

```

cookie を設定するときは、`path` プロパティにご注意ください。デフォルトでは、cookie の `path` は現在のパス名です。例えば、`admin/user` ページで cookie を設定した場合、デフォルトではその cookie は `admin` ページ配下でのみ使用することができます。多くの場合、`path` を `'/'` に設定して、アプリ全体で cookie を使用できるようになたいでしょう。

server `load` 関数と universal `load` 関数はどちらも `setHeaders` 関数にアクセスでき、サーバー上で実行している場合、レスポンスにヘッダーを設定できます（ブラウザで実行している場合、`setHeaders` には何の効果もありません）。これは、ページをキャッシュさせる場合に便利です、例えば：

```

// @errors: 2322 1360
/// file: src/routes/products/+page.js
/** @type {import('./$types').PageLoad} */
export async function load({ fetch, setHeaders }) {
  const url = `https://cms.example.com/products.json`;
  const response = await fetch(url);

  // cache the page for the same length of time
  // as the underlying data
  setHeaders({
    'age': response.headers.get('age'),
    'cache-control': response.headers.get('cache-control')
  });

  return response.json();
}

```

同じヘッダーを複数回設定することは（たとえ別々の `load` 関数であっても）エラーとなります。指定したヘッダーを設定できるのは一度だけです。`set-cookie` ヘッダーは、`setHeaders` では追加できません。代わりに、`cookies.set(name, value, options)` を使用してください。

Using parent data

`load` 関数が親の `load` 関数からのデータにアクセスできたら便利なことがあるでしょう。`await parent()` でこれを行うことができます：

```

/// file: src/routes/+layout.js
/** @type {import('./$types').LayoutLoad} */
export function load() {
  return { a: 1 };
}

```

```

/// file: src/routes/abc/+layout.js
/** @type {import('./$types').LayoutLoad} */
export async function load({ parent }) {
  const { a } = await parent();
  return { b: a + 1 };
}

```

```
/// file: src/routes/abc/+page.js
/** @type {import('./$types').PageLoad} */
export async function load({ parent }) {
  const { a, b } = await parent();
  return { c: a + b };
}
```

```
/// file: src/routes/abc/+page.svelte
<script>
  /** @type {import('./$types').PageData} */
  export let data;
</script>

<!-- renders `1 + 2 = 3` -->
<p>{data.a} + {data.b} = {data.c}</p>
```

`+page.js` の `load` 関数が、直接の親だけなく、両方のレイアウトの `load` 関数からマージされたデータを受け取っていることにご注意ください。

`+page.server.js` と `+layout.server.js` の中では、`parent` は親の `+layout.server.js` ファイルからデータを取得します。

`+page.js` や `+layout.js` では、親の `+layout.js` ファイルからデータを返します。しかし、`+layout.js` が見つからない場合は `({ data }) => data` 関数として扱われます。つまり、`+layout.js` ファイルによって'シャドウ(shadowed)'されていない親の `+layout.server.js` ファイルからデータを返します。

`await parent()` を使用する際はウォータフォールを発生させないよう注意してください。例えば、こちらの `getData(params)` は `parent()` の呼び出しの結果には依存しないので、レンダリングの遅延を避けるために最初に呼び出すほうが良いでしょう。

```
/// file: +page.js
/** @type {import('./$types').PageLoad} */
export async function load({ params, parent }) {
-  const parentData = await parent();
  const data = await getData(params);
+  const parentData = await parent();

  return {
    ...data
    meta: { ...parentData.meta, ...data.meta }
  };
}
```

Errors

`load` 中にエラーがスローされた場合、最も近くにある `+error.svelte` がレンダリングされます。想定されるエラーには、`@sveltejs/kit` からインポートできる `error` ヘルパーを使用して HTTP ステータスコードとオプションのメッセージを指定できます：

```
/// file: src/routes/admin/+layout.server.js
// @filename: ambient.d.ts
declare namespace App {
```

```

interface Locals {
    user?: {
        name: string;
        isAdmin: boolean;
    }
}

// @filename: index.js
// cut---
import { error } from '@sveltejs/kit';

/** @type {import('./$types').LayoutServerLoad} */
export function load({ locals }) {
    if (!locals.user) {
        throw error(401, 'not logged in');
    }

    if (!locals.user.isAdmin) {
        throw error(403, 'not an admin');
    }
}

```

予期せぬエラーがスローされた場合、SvelteKit は `handleError` を呼び出し、それを 500 Internal Error として処理します。

Redirects

ユーザーをリダイレクトさせるには、`@sveltejs/kit` からインポートできる `redirect` ヘルパーを使用して、ステータスコード `3xx` と一緒にリダイレクト先の location を指定します。

```

/// file: src/routes/user/+layout.server.js
// @filename: ambient.d.ts
declare namespace App {
    interface Locals {
        user?: {
            name: string;
        }
    }
}

// @filename: index.js
// cut---
import { redirect } from '@sveltejs/kit';

/** @type {import('./$types').LayoutServerLoad} */
export function load({ locals }) {
    if (!locals.user) {
        throw redirect(307, '/login');
    }
}

```

スローされた `redirect` をキャッチしないようにしてください、SvelteKit が処理するのを妨げてしまいます。

ブラウザでは、`$app.navigation` からインポートできる `goto` を使うことで、`load` 関数の外側でプログラム的にナビゲーションを行うことができます。

Streaming with promises

戻り値のトップレベルにある promise は `await` されるので、ウォーターフォールが作られることなく複数の promise を簡単に返すことができます。server `load` を使用する場合、ネストした promise は解決されると同時にブラウザにストリームされます。これにより、遅くて重要でないデータがある場合でも、すべてのデータが利用可能になる前にページのレンダリングを開始することができるので便利です：

```
/// file: src/routes/+page.server.js
/** @type {import('./$types').PageServerLoad} */
export function load() {
  return {
    one: Promise.resolve(1),
    two: Promise.resolve(2),
    streamed: {
      three: new Promise((fulfil) => {
        setTimeout(() => {
          fulfil(3)
        }, 1000);
      })
    }
  };
}
```

これはロード状態/loading states)のスケルトンを作成するのに便利です、例えば：

```
/// file: src/routes/+page.svelte
<script>
  /** @type {import('./$types').PageData} */
  export let data;
</script>

<p>
  one: {data.one}
</p>
<p>
  two: {data.two}
</p>
<p>
  three:
  {#await data.streamed.three}
    Loading...
  {:then value}
    {value}
  {:catch error}
    {error.message}
  {/await}
</p>
```

AWS Lambda のようなストリーミングをサポートしないプラットフォームでは、レスポンスはバッファされます。つまり、すべての promise が解決してからでないとページがレンダリングされないということです。

ストリーミングデータは JavaScript が有効なときにのみ動作します。ページがサーバーでレンダリングされる場合、universal `load` 関数からはネストした promise を返すのは避けたほうがよいでしょう、ストリーミングされないからです（代わりに、関数がブラウザで再実行されるときに promise が再作成されます）。

Parallel loading

ページをレンダリング（またはページにナビゲーション）するとき、SvelteKit はすべての `load` 関数を同時に実行し、リクエストのウォーターフォールを回避します。クライアントサイドナビゲーションのときは、複数の server `load` 関数の呼び出し結果が単一のレスポンスにグループ化されます。すべての `load` 関数が返されると、ページがレンダリングされます。

load 関数の再実行

SvelteKit はそれぞれの `load` 関数の依存関係を追跡し、ナビゲーションの際に不必要に再実行されるのを回避します。

例えば、このような `load` 関数のペアがあるとして…

```
/// file: src/routes/blog/[slug]/+page.server.js
// @filename: ambient.d.ts
declare module '$lib/server/database' {
    export function getPost(slug: string): Promise<{ # string, content: string }>
}

// @filename: index.js
// cut---
import * as db from '$lib/server/database';

/** @type {import('./$types').PageServerLoad} */
export async function load({ params }) {
    return {
        post: await db.getPost(params.slug)
    };
}
```

```
/// file: src/routes/blog/[slug]/+layout.server.js
// @filename: ambient.d.ts
declare module '$lib/server/database' {
    export function getPostSummaries(): Promise<Array<{ # string, slug: string }>>;
}

// @filename: index.js
// cut---
import * as db from '$lib/server/database';

/** @type {import('./$types').LayoutServerLoad} */
export async function load() {
    return {
        posts: await db.getPostSummaries()
    };
}
```

…もし、`/blog/trying-the-raw-meat-diet` から `/blog/i-regret-my-choices` に移動したら、`params.slug` が変更されているので、`+page.server.js` のほうは再実行されるでしょう。`+layout.server.js` のほうは再実行されません、なぜならデータがまだ有効だからです。言い換えると、`db.getPostSummaries()` を2回呼び出すことはないということです。

`await parent()` を呼び出している `load` 関数は、親の `load` 関数が再実行された場合に再実行されます。

依存関係の追跡は、`load` 関数から返したあとには適用されません。例えば、ネストした `promise` の内側で `params.x` にアクセスしている場合、`params.x` が変更されても関数の再実行は行われません（ご心配なく、誤ってこれを行っても開発中に警告が表示されます）。代わりに、`load` 関数の本体部分でパラメータにアクセスしてください。

Manual invalidation

現在のページに適用される `load` 関数は、`invalidate(url)` を使用することで再実行させることもできます。これは `url` に依存しているすべての `load` 関数を再実行させるものです。`invalidateAll()` は、すべての `load` 関数を再実行させます。server load 関数の場合は、クライアントに秘情報が漏れるのを防ぐため、取得した `url` に自動的に依存することはありません。

`load` 関数が `fetch(url)` や `depends(url)` を呼び出している場合、その `load` 関数は `url` に依存しています。`url` には `[a-z]`: から始まるカスタムの識別子を指定することができますことにご注意ください:

```
/// file: src/routes/random-number/+page.js
/** @type {import('./$types').PageLoad} */
export async function load({ fetch, depends }) {
  // load reruns when `invalidate('https://api.example.com/random-number')` is
  // called...
  const response = await fetch('https://api.example.com/random-number');

  // ...or when `invalidate('app:random')` is called
  depends('app:random');

  return {
    number: await response.json()
  };
}
```

```
/// file: src/routes/random-number/+page.svelte
<script>
  import { invalidate, invalidateAll } from '$app/navigation';

  /** @type {import('./$types').PageData} */
  export let data;

  function rerunLoadFunction() {
    // any of these will cause the `load` function to rerun
    invalidate('app:random');
    invalidate('https://api.example.com/random-number');
    invalidate(url => url.href.includes('random-number'));
    invalidateAll();
  }
</script>

<p>random number: {data.number}</p>
<button on:click={rerunLoadFunction}>Update random number</button>
```

load 関数はいつ再実行されるのか

まとめると、`load` 関数は以下のシチュエーションで再実行されます：

- `params` のプロパティを参照していて、その値が変更された場合
- `url` のプロパティ（例えば `url.pathname` や `url.search`）を参照していて、その値が変更された場合。`request.url` のプロパティは追跡されません
- `await parent()` を呼び出していて、親の `load` 関数が再実行されたとき
- `fetch` (universal load のみ) や `depends` を介して特定の URL に対する依存を宣言していて、その URL が `invalidate(url)` で無効(invalid)であるとマークされた場合
- `invalidateAll()` によって全ての有効な `load` 関数が強制的に再実行された場合

`params` と `url` は、`` リンクのクリックや、`<form>` のやりとり、`goto` の呼び出し、`redirect` に応じて変更されます。

`load` 関数の再実行は、対応する `+layout.svelte` や `+page.svelte` 内の `data` プロパティが更新されるだけで、コンポーネントは再作成されることはありません。結果として、内部の状態は保持されます。もし、この挙動がお望みでなければ、`afterNavigate` コールバック内でリセットしたり、コンポーネントを `{#key ...}` ブロックでラップしたりしてリセットすることができます。

その他の参考資料

- [Tutorial: Loading data](#)
- [Tutorial: Errors and redirects](#)
- [Tutorial: Advanced loading](#)

Form actions

`+page.server.js` ファイルは `actions` をエクスポートできます。これによって、`<form>` 要素を使用することでサーバーにデータを `POST` することができます。

`<form>` を使用する場合、クライアントサイドの JavaScript はオプションですが、JavaScript によって form のインタラクションを簡単にプログレッシブに強化(*progressively enhance*)することができ、最高のユーザー エクスペリエンスを提供することができます。

Default actions

最もシンプルなケースでは、ページは `default` の `action` を宣言します：

```
/// file: src/routes/login/+page.server.js
/** @type {import('./$types').Actions} */
export const actions = {
  default: async (event) => {
    // TODO log the user in
  }
};
```

`/login` ページからこの `action` を呼び出すには、`<form>` を追加します。JavaScript は必要ありません：

```
/// file: src/routes/login/+page.svelte
<form method="POST">
  <label>
    Email
    <input name="email" type="email">
  </label>
  <label>
    Password
    <input name="password" type="password">
  </label>
  <button>Log in</button>
</form>
```

もし誰かがボタンをクリックしたら、ブラウザは `form` のデータを `POST` リクエストでサーバーに送信し、デフォルトの `action` が実行されます。

Action は常に `POST` リクエストを使用します。`GET` リクエストには決して副作用があつてはならないからです。

また、`action` 属性を追加し、リクエスト先のページを指し示すことで、他のページから `action` を呼び出すこともできます（例えば、最上位のレイアウト(root layout)にある `nav` にログイン用の `widget` がある場合）：

```
/// file: src/routes/+layout.svelte
<form method="POST" action="/login">
  <!-- content -->
</form>
```

Named actions

単一の `default` の action の代わりに、名前付きの action (named action) を必要なだけ持つことができます:

```
/// file: src/routes/login/+page.server.js
/** @type {import('./$types').Actions} */
export const actions = {
-  default: async (event) => {
+  login: async (event) => {
    // TODO log the user in
  },
+  register: async (event) => {
+    // TODO register the user
+  }
};
```

名前付きの action (named action) を呼び出すには、クエリパラメータに `/` を接頭辞に付与したその action の名前を追加します:

```
/// file: src/routes/login/+page.svelte
<form method="POST" action="?/register">
```

```
/// file: src/routes/+layout.svelte
<form method="POST" action="/login?/register">
```

`action` 属性と同じように、button の `formaction` 属性を使用することができます、こうすると親の `<form>` とは別の action に同じ form のデータを `POST` することができます:

```
/// file: src/routes/login/+page.svelte
-<form method="POST">
+<form method="POST" action="?/login">
  <label>
    Email
    <input name="email" type="email">
  </label>
  <label>
    Password
    <input name="password" type="password">
  </label>
  <button>Log in</button>
+  <button formaction="?/register">Register</button>
</form>
```

名前付き action (named action) の隣にデフォルトの action を置くことはできません。なぜなら リダイレクト無しで名前付き action (named action) に POST をすると、クエリパラメータが URL に保持され、それ以降デフォルトの POST をしようとしても以前 POST した名前付き action (named action) を通ってしまうからです。

action の解剖学

action はそれぞれ `RequestEvent` オブジェクトを受け取って、`request.formData()` でデータを読み込むことができます。リクエスト（例えば、cookie をセットしてユーザーをログインさせるなど）を処理したあと、action は次の更新まで、対応するページでは `form` プロパティで、アプリ全体では `$page.form` で利用可能なデータで応答することができます。

```
// @errors: 2304
/// file: src/routes/login/+page.server.js
/** @type {import('./$types').PageServerLoad} */
export async function load({ cookies }) {
  const user = await db.getUserFromSession(cookies.get('sessionid'));
  return { user };
}

/** @type {import('./$types').Actions} */
export const actions = {
  login: async ({ cookies, request }) => {
    const data = await request.formData();
    const email = data.get('email');
    const password = data.get('password');

    const user = await db.getUser(email);
    cookies.set('sessionid', await db.createSession(user));

    return { success: true };
  },
  register: async (event) => {
    // TODO register the user
  }
};
```

```
/// file: src/routes/login/+page.svelte
<script>
  /** @type {import('./$types').PageData} */
  export let data;

  /** @type {import('./$types').ActionData} */
  export let form;
</script>

{#if form?.success}
  <!-- このメッセージは一時的なものです； form 送信に対するレスポンスとしてページがレンダリングされたため、存在しています。
       ユーザーがリロードすると消えます。 -->
  <p>Successfully logged in! Welcome back, {data.user.name}</p>
{/if}
```

Validation errors

無効なデータが原因でリクエストが処理できなかった場合、再試行できるようにするために、直前に送信した form の値とともに validation error をユーザーに返すことができます。`fail` 関数は、HTTP ステータスコード（通常、validation error の場合は 400 か 422）をデータとともに返します。ステータスコードは `$page.status` から使用することができます、`data` は `form` から使用することができます：

```
/// file: src/routes/login/+page.server.js
+import { fail } from '@sveltejs/kit';

/** @type {import('./$types').Actions} */
export const actions = {
    login: async ({ cookies, request }) => {
        const data = await request.formData();
        const email = data.get('email');
        const password = data.get('password');

        + if (!email) {
        +     return fail(400, { email, missing: true });
        + }

        const user = await db.getUser(email);

        + if (!user || user.password !== hash(password)) {
        +     return fail(400, { email, incorrect: true });
        + }

        cookies.set('sessionid', await db.createSession(user));

        return { success: true };
    },
    register: async (event) => {
        // TODO register the user
    }
};
```

念のため、password は返さず、email のみをページに返していることにご注意ください。

```
/// file: src/routes/login/+page.svelte
<form method="POST" action="/login">
+ {#if form?.missing}<p class="error">The email field is required</p>{/if}
+ {#if form?.incorrect}<p class="error">Invalid credentials!</p>{/if}
    <label>
        Email
    -   <input name="email" type="email">
    +   <input name="email" type="email" value={form?.email ?? ''}>
    </label>
    <label>
        Password
    -   <input name="password" type="password">
    </label>
    <button>Log in</button>
    <button formaction="/register">Register</button>
</form>
```

戻り値は JSON としてシリализ可能でなければなりません。その上で、構造は完全にあなた次第です。例えば、もしページに複数の form がある場合、返された `form` データがどの `<form>` を参照しているかを `id` プロパティなどで区別することができます。

Redirects

`redirect` (と `error`) は `load` のそれと同じように機能します:

```

/// file: src/routes/login/+page.server.js
+import { fail, redirect } from '@sveltejs/kit';

/** @type {import('./$types').Actions} */
export const actions = {
+  login: async ({ cookies, request, url }) => {
    const data = await request.formData();
    const email = data.get('email');
    const password = data.get('password');

    const user = await db.getUser(email);
    if (!user) {
      return fail(400, { email, missing: true });
    }

    if (user.password !== hash(password)) {
      return fail(400, { email, incorrect: true });
    }

    cookies.set('sessionid', await db.createSession(user));
+    if (url.searchParams.has('redirectTo')) {
+      throw redirect(303, url.searchParams.get('redirectTo'));
+    }
    return { success: true };
  },
  register: async (event) => {
    // TODO register the user
  }
};

```

Loading data

action の実行後、そのページは（リダイレクトや予期せぬエラーが発生しない限り）再レンダリングされ、action の戻り値が `form` プロパティとしてそのページで使用できるようになります。つまり、ページの `load` 関数は、action が完了したあとに実行されるということです。

`handle` は action が呼び出される前に実行され、`load` 関数よりも前に再実行されることはないことに注意してください。つまり、例えば `handle` を使用して cookie を元に `event.locals` に値を入れる場合、action で cookie を設定したり削除したりするときは `event.locals` を更新しなければなりません：

```

/// file: src/hooks.server.js
// @filename: ambient.d.ts
declare namespace App {
  interface Locals {
    user: {
      name: string;
    } | null
  }
}

// @filename: global.d.ts
declare global {
  function getUser(sessionid: string | undefined): {
    name: string;
  };
}

```

```

export {};

// @filename: index.js
// cut---
/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
    event.locals.user = await getUser(event.cookies.get('sessionid'));
    return resolve(event);
}

/// file: src/routes/account/+page.server.js
// @filename: ambient.d.ts
declare namespace App {
    interface Locals {
        user: {
            name: string;
        } | null
    }
}

// @filename: index.js
// cut---
/** @type {import('./$types').PageServerLoad} */
export function load(event) {
    return {
        user: event.locals.user
    };
}

/** @type {import('./$types').Actions} */
export const actions = {
    logout: async (event) => {
        event.cookies.delete('sessionid');
        event.locals.user = null;
    }
};

```

Progressive enhancement

前のセクションでは [クライアントサイドの JavaScriptなしで動作する `/login` action](#) を構築しました — `fetch` は見当たりません。これは素晴らしいですが、JavaScript が利用可能な場合は、より良いユーザーエクスペリエンスを提供するために `form` のインタラクションをプログレッシブに強化 (progressively enhance) することができます。

`use:enhance`

`form` をプログレッシブに強化する最も簡単な方法は、`use:enhance` action を追加することです:

```

/// file: src/routes/login/+page.svelte
<script>
+  import { enhance } from '$app/forms';

  /** @type {import('./$types').ActionData} */
  export let form;
</script>

+<form method="POST" use:enhance>

```

ええ、`enhance action` と `<form action>` をどちらも 'action' と呼んでいて、少し紛らわしいですよね。このドキュメントは `action` でいっぱいです。申し訳ありません。

引数が無い場合、`use:enhance` は、ブラウザネイティブの動作を、フルページリロードを除いてエミュレートします。それは：

- `action` が送信元のページと同じ場所にある場合に限り、成功レスポンスまたは不正なレスポンスに応じて、`form` プロパティと `$page.form` と `$page.status` を更新します。例えば、`<form action="/somewhere/else" ...>` というようなフォームの場合、`form` と `$page` は更新されません。これは、ネイティブのフォーム送信では `action` があるページにリダイレクトされるからです。どちらにしても更新させたい場合は、`applyAction` を使用してください
- 成功レスポンスの場合は、`<form>` 要素をリセットして `invalidateAll` で全てのデータを無効化・最新化(invalidate)します
- リダイレクトレスポンスの場合は `goto` を呼び出します
- エラーが発生した場合はもっとも近くにある `+error` 境界をレンダリングします
- 適切な要素に `フォーカスをリセット` します

この挙動をカスタマイズするために、`form` が送信される直前に実行される `SubmitFunction` 関数を提供することができます。そして(オプションで) `ActionResult` を引数に取るコールバックを返すことができます。もしコールバックを返す場合、上述のデフォルトの動作はトリガーされません。元に戻すには、`update` を呼び出してください。

```
<form
  method="POST"
  use:enhance={({ formElement, formData, action, cancel, submitter }) => {
    // `formElement` はこの `<form>` 要素です
    // `formData` は送信される予定の `FormData` オブジェクトです
    // `action` はフォームが POST される URL です
    // `cancel()` を呼び出すと送信(submission)を中止します
    // `submitter` は、フォームの送信を実行した `HTMLElement` です

    return async ({ result, update }) => {
      // `result` は `ActionResult` オブジェクトです
      // `update` は、このコールバックが設定されていない場合に起動されるデフォルトのログックを起動する関数です
    };
  }}
>
```

これらの関数を、ロード中の UI (loading UI) を表示したり隠したりすることなどに使用できます。

applyAction

独自のコールバックを提供する場合は、最も近くにある `+error` 境界を表示するなど、デフォルトの `use:enhance` の一部を再現する必要があるでしょう。ほとんどの場合、コールバックに渡された `update` を呼び出すだけで十分です。もっとカスタマイズが必要な場合は、`applyAction` を使用してそれを行うことができます：

```
/// file: src/routes/login/+page.svelte
<script>
+ import { enhance, applyAction } from '$app/forms';
```

```

/** @type {import('./$types').ActionData} */
export let form;
</script>

<form
method="POST"
use:enhance={({ formElement, formData, action, cancel }) => {

    return async ({ result }) => {
        // `result` は `ActionResult` オブジェクトです
+       if (result.type === 'error') {
+           await applyAction(result);
+       }
    };
}}
```

`applyAction(result)` の挙動は `result.type` に依存しています:

- `success`, `failure` — `$page.status` を `result.status` に設定し、`form` と `$page.form` を `result.data` で更新します (`enhance` の `update` とは対照的に、送信元がどこかは関係ありません)
- `redirect` — `goto(result.location)` を呼び出します
- `error` — もっとも近くにある `+error` 境界を `result.error` でレンダリングします

いずれの場合でも、[フォーカスはリセットされます](#)。

Custom event listener

`use:enhance` ではなく、`<form>` の通常のイベントリスナーを使うことで、ご自身でプログレッシブ・エンハンスメント (progressive enhancement)を実装することもできます:

```

/// file: src/routes/login/+page.svelte
<script>
    import { invalidateAll, goto } from '$app/navigation';
    import { applyAction, deserialize } from '$app/forms';

    /** @type {import('./$types').ActionData} */
    export let form;

    /** @type {any} */
    let error;

    async function handleSubmit(event) {
        const data = new FormData(this);

        const response = await fetch(this.action, {
            method: 'POST',
            body: data
        });

        /** @type {import('@sveltejs/kit').ActionResult} */
        const result = deserialize(await response.text());

        if (result.type === 'success') {
            // rerun all `load` functions, following the successful update
            await invalidateAll();
        }
    }

```

```

        applyAction(result);
    }
</script>

<form method="POST" on:submit|preventDefault={handleSubmit}>
    <!-- content -->
</form>
```

処理を進める前に、`$app/forms` の `deserialize` でレスポンスをデシリアライズする必要があることにご注意ください。`JSON.parse()` では不十分です。なぜなら、例えば `load` 関数のような form action は、`Date` や `BigInt` オブジェクトも戻り値としてサポートしているからです。

もし `+page.server.js` と `+server.js` のどちらも存在する場合、デフォルトでは、`fetch` リクエストは `+server.js` のほうにルーティングされます。`+page.server.js` の `action` に `POST` をするには、カスタムの `x-sveltekit-action` ヘッダーを使用します：

```

const response = await fetch(this.action, {
    method: 'POST',
    body: data,
+   headers: {
+     'x-sveltekit-action': 'true'
+   }
});
```

Alternatives

サーバーにデータを送信する方法として、プログレッシブな強化(progressively enhance)を行うことができるため Form actions は望ましい方法ですが、`+server.js` ファイルを使用して(例えば) JSON API を公開することもできます。それは例えばこのように行います：

```

/// file: send-message/+page.svelte
<script>
    function rerun() {
        fetch('/api/ci', {
            method: 'POST'
        });
    }
</script>

<button on:click={rerun}>Rerun CI</button>

// @errors: 2355 1360
// file: api/ci/+server.js

/** @type {import('./$types').RequestHandler} */
export function POST() {
    // do something
}
```

GET vs POST

これまで見てきたように、フォームアクションを使うには、`method="POST"` を使用する必要があります。

サーバーにデータを `POST` する必要がないフォームもあるでしょう — 例えば検索入力(search inputs)です。これに対応するには `method="GET"` (または、`method` を全く書かないのも同等です) を使うことができ、そして SvelteKit はそれを `<a>` 要素のように扱い、フルページナビゲーションの代わりにクライアントサイドルーターを使用します。：

```
<form action="/search">
  <label>
    Search
    <input name="q">
  </label>
</form>
```

この `form` を送信すると `/search?q=...` に移動して `load` 関数が実行されますが、`action` は実行されません。`<a>` 要素と同じように、`data-sveltekit-reload` 属性、`data-sveltekit-replacestate` 属性、`data-sveltekit-keepfocus` 属性、`data-sveltekit-noscroll` 属性を `<form>` に設定することができ、ルーターの挙動をコントロールすることができます。

その他の参考資料

- [Tutorial: Forms](#)

Page options

デフォルトでは、SvelteKit はどのコンポーネントも最初はサーバーでレンダリング（または [プリレンダリング](#)）し、それを HTML としてクライアントに送信します。その後、ブラウザ上でコンポーネントを再度レンダリングし、[ハイドレーション\(hydration\)](#)と呼ばれるプロセスでそれをインタラクティブなものにします。このため、コンポーネントが両方の場所で実行できることを確認する必要があります。SvelteKit はそれから [ルーター\(router\)](#) を初期化し、その後のナビゲーションを引き継ぎます。

これらはそれぞれオプションを `+page.js` や `+page.server.js` からエクスポートすることでページごとに、または共有の `+layout.js` や `+layout.server.js` を使用してページグループごとに制御することができます。アプリ全体に対してオプションを定義するには、最上位のレイアウト(root layout)からそれをエクスポートします。子レイアウトとページは親レイアウトで設定された値を上書きするため、例えば、プリレンダリングをアプリ全体で有効にし、それから動的にレンダリングする必要があるページではそれを無効にすることができます。

アプリの様々な領域でこれらのオプションをうまく組み合わせることができます。例えば、マーケティングページは高速化を最大限にするためにプリレンダリングし、動的なページは SEO とアクセシビリティのためにサーバーでレンダリングし、管理者用のセクションはクライアントのみでレンダリングするようにして SPA にすることができます。このように、SvelteKit はとても万能で多くの用途にお使いいただけます。

prerender

あなたのアプリの、少なくともいくつかのルートは、ビルト時に生成されるシンプルな HTML ファイルとして表現されることが多いでしょう。これらのルート(routes)を [プリレンダリング](#)することができます。

```
/// file: +page.js/+page.server.js/+server.js
export const prerender = true;
```

代わりに、`export const prerender = true` を最上位(root)の `+layout.js` または `+layout.server.js` に設定し、明示的にプリレンダリングしないものとしてマークされたページを除き、全てをプリレンダリングできます：

```
/// file: +page.js/+page.server.js/+server.js
export const prerender = false;
```

`prerender = true` があるルート(routes)は動的な SSR を行うのに使用する manifest から除外されるため、サーバー（または serverless/edge functions）を小さくすることができます。場合によっては、ルート(route)をプリレンダリングしつつ、manifest にも含めたいことがあるでしょう（例えば、`/blog/[slug]` のようなルート(route)があり、最も新しい/人気のあるコンテンツはプリレンダリングしたいがめったにアクセスされないものはサーバーでレンダリングしたい、など）。こういったケースのために、3つ目のオプションがあります、'auto' です：

```
/// file: +page.js/+page.server.js/+server.js
export const prerender = 'auto';
```

もしアプリ全体がプリレンダリングに適している場合は、`adapter-static` を使うことで、任意の静的 Web サーバーで使用するのに適したファイルを出力することができます。

プリレンダラはアプリの最上位(root)から開始され、プリレンダリング可能なページや `+server.js` ルート(routes)を見つけると、そのファイルを生成します。各ページは、プリレンダリングの候補である他のページを指し示す `<a>` 要素を見つけるためにスキャンされます。このため、通常はどのページにアクセスすべきか指定する必要はありません。もしプリレンダラがアクセスするページを指定する必要がある場合は、`config.kit.prerender.entries` で指定するか、動的なルート(route)から `entries` 関数をエクスポートします。

プリレンダリング中、`$app/environment` からインポートされる `building` の値は `true` になります。

Prerendering server routes

他のページオプションとは違い、`prerender` は `+server.js` ファイルにも適用できます。これらのファイルはレイアウトから影響を受けませんが、そこからデータを読み込むページからデフォルトの値を継承します。例えば、`+page.js` がこの `load` 関数を含む場合…

```
/// file: +page.js
export const prerender = true;

/** @type {import('./$types').PageLoad} */
export async function load({ fetch }) {
  const res = await fetch('/my-server-route.json');
  return await res.json();
}
```

…それから `src/routes/my-server-route.json/+server.js` は、自身の `export const prerender = false` を含んでいなければ、プリレンダリング可能であると扱われることになります。

プリレンダリングしない場合

基本的なルールは次の通りです：ページがプリレンダリング可能であると言うためには、そのページを直接表示する2人のユーザーが、サーバーから同じコンテンツを取得できなければなりません。

全てのページがプリレンダリングに適しているわけではありません。プリレンダリングされたコンテンツは全てのユーザーに表示されます。もちろん、プリレンダリングされたページの `onMount` でパーソナライズされたデータをフェッチできますが、ブランクの初期コンテンツやローディングインジケーターにより、ユーザエクスペリエンスが低下してしまう可能性があります。

`src/routes/blog/[slug]/+page.svelte` ルート(route)のような、ページのパラメータを元にデータをロードするページもプリレンダリングができるごとにご注意ください。

プリレンダリング中に `urlSearchParams` にアクセスすることは禁止されています。もし使う必要があるなら、ブラウザの中だけで行うようにしてください (例えば `onMount` の中に)。

`action` 付きのページは、サーバーがその `action` の `POST` リクエストを処理できなければならないため、プリレンダリングできません。

Prerender and ssr

`ssr option` を `false` に設定すると、各リクエストは同じ空の HTML shell になってしまいます。これは不必要的作業となるため、SvelteKit は `prerender` が明示的に `false` に設定されていないページを見つけた場合、デフォルトでプリレンダリングを行います。

ルートの衝突(Route conflicts)

プリレンダリングはファイルシステムに書き込むため、ディレクトリとファイルが同じ名前になるエンドポイントを2つ持つことはできません。例えば、`src/routes/foo/+server.js` と `src/routes/foo/bar/+server.js` の場合は、`foo` と `foo/bar` を作成しようとしたとき、これは不可能です。

このため(他にも理由がありますが)、常に拡張子を付けておくことを推奨します — `src/routes/foo.json/+server.js` と `src/routes/foo/bar.json/+server.js` は、`foo.json` と `foo/bar.json` ファイルが並んで調和して共存できます。

ページの場合は、`foo` ではなく `foo/index.html` を書き込むことでこの問題を回避しています。

Troubleshooting

'The following routes were marked as prerenderable, but were not prerendered' というようなエラーが表示されたら、それは該当のルート(またはページの場合は親レイアウト)に `export const prerender = true` があるにもかかわらず実際にはそのページがプリレンダリングされていないことが原因です(プリレンダリングクローラーがそのページにアクセスしていないため)。

これらのルート(route)は動的にサーバーレンダリングできないため、該当のルート(route)にアクセスしようとしたときにエラーが発生します。それを解決するには、2つの方法があります:

- SvelteKit が `config.kit.prerender.entries` か `entries` ページオプションからのリンクを辿ってそのルート(route)を見つけられるようにしてください。動的なルート(例えば `[parameters]` を持つページ)へのリンクは、他のエントリーポイントをクローリングしても見つからない場合はこのオプションに追加してください。そうしないと、SvelteKit はその `parameters` が持つべき値がわからないので、プリレンダリングされません。プリレンダリング可能(prerenderable)なページとしてマークされていないページは無視され、そのページから他のページ(プリレンダリング可能なものも含む)へのリンクもクローリングされません。
- `export const prerender = true` から `export const prerender = 'auto'` に変更してください。`'auto'` になっているルート(route)は動的にサーバーレンダリングすることができます

entries

SvelteKit は、エントリーポイント(entry points)を開始地点としてクローリングを行うことでページを自動的に発見します。デフォルトでは、動的でないルート(route)はすべてエントリーポイントとみなされます。例えば、以下のルートがある場合...

```
/          # non-dynamic
/blog      # non-dynamic
/blog/[slug] # dynamic, because of `'[slug]'`
```

...SvelteKit は `/` と `/blog` をプリレンダリングし、その過程で `` などのリンクを発見し、それをプリレンダリング対象とします。

ほとんどの場合、これで十分です。しかし状況によっては、`/blog/hello-world` などのページに対するリンクが存在しない（あるいはプリレンダリングされたページには存在しない）場合があります。この場合、SvelteKit にその存在を知らせる必要があります。

これを行うには `config.kit.prerender.entries` で指定するか、動的なルート (route) に属する `+page.js` か `+page.server.js` か `+server.js` で `entries` 関数をエクスポートします:

```
/// file: src/routes/blog/[slug]/+page.server.js
/** @type {import('./$types').EntryGenerator} */
export function entries() {
  return [
    { slug: 'hello-world' },
    { slug: 'another-blog-post' }
  ];
}

export const prerender = true;
```

`entries` は `async` 関数にすることができる（例えば）上記で示したように CMS や データベースから投稿リストを取得することもできます。

SSR

通常、SvelteKit ではページを最初にサーバーでレンダリングし、その HTML をクライアントに送信してハイドレーションを行います。もし `ssr` を `false` に設定した場合、代わりに空の 'shell' ページがレンダリングされます。これはページがサーバーでレンダリングできない場合には便利（例えば `document` などのブラウザオンリーな `globals` を使用するなど）ですが、ほとんどの状況では推奨されません（appendix をご参照ください）。

```
/// file: +page.js
export const ssr = false;
```

`export const ssr = false` を最上位(root)の `+layout.js` に追加した場合、アプリ全体がクライアントのみでレンダリングされるようになります、それはつまり、本質的にはアプリを SPA にする、ということを意味します。

CSR

通常、SvelteKit はサーバーでレンダリングされた HTML を、クライアントサイドレンダリング(CSR)されたインタラクティブなページに ハイドレーション します。JavaScript を全く必要としないページもあります。多くのブログ記事や 'about' ページがこのカテゴリに入ります。このような場合は CSR を無効にすることができます:

```
/// file: +page.js
export const csr = false;
```

`ssr to csr` の両方が `false` である場合は、何もレンダリングされません！

trailingSlash

デフォルトでは、SvelteKit は URL から末尾のスラッシュ(trailing slash)を取り除きます。`/about/` にアクセスすると、`/about` へのリダイレクトをレスポンスとして受け取ることになります。この動作は、`trailingSlash` オプションで変更することができます。指定できる値は `'never'` (デフォルト)、`'always'`、`'ignore'` です。

他のページオプションと同様に、`+layout.js` や `+layout.server.js` からこの値をエクスポートすると、すべての子のページに適用されます。`+server.js` ファイルからその設定をエクスポートすることもできます。

```
// file: src/routes/+layout.js
export const trailingSlash = 'always';
```

このオプションは [プリレンダリング](#) にも影響します。`trailingSlash` が `always` の場合 `/about` というルート(route)は `about/index.html` ファイルとなり、それ以外の場合は `about.html` が作成され、静的な Web サーバの慣習を反映したものになります。

末尾のスラッシュを無視することは推奨されません。相対パスのセマンティクスが2つのケースで異なり(`/x` からの`./y` は `/y` ですが、`/x/` からは `/x/y` となります)、`/x` と `/x/` は別の URL として扱われ、SEO 上有害となるからです。

config

`adapter` のコンセプトにより、SvelteKit は様々なプラットフォーム上で実行することができます。しかし、各プラットフォームには、デプロイメントをさらに微調整するための特定の設定があるかもしれません — 例えば Vercel では、アプリのある部分はエッジに、他の部分はサーバーレス環境にデプロイするのを選択することができます。

`config` はトップレベルで key-value ペアを持つオブジェクトです。その他の具体的な形は、使用する `adapter` に依存します。すべての `adapter` は型安全性のためにインポート可能な `Config` インターフェースを提供することになっています。詳細な情報については、使用する `adapter` のドキュメントを参照してください。

```
// @filename: ambient.d.ts
declare module 'some-adapter' {
    export interface Config { runtime: string }
}

// @filename: index.js
// cut---
/// file: src/routes/+page.js
/** @type {import('some-adapter').Config} */
export const config = {
    runtime: 'edge'
};
```

`config` オブジェクトはトップレベル(top level)でマージされます (より深いレベル(deeper levels)ではマージされません)。つまり、より上位の `+layout.js` にある値の一部を上書きしたい場合に、`+page.js` にある全ての値を繰り返す必要はないということです。例えばこの `layout` の設定は...

```
/// file: src/routes/+layout.js
export const config = {
  runtime: 'edge',
  regions: 'all',
  foo: {
    bar: true
  }
}
```

...この page の設定で上書きされ...

```
/// file: src/routes/+page.js
export const config = {
  regions: ['us1', 'us2'],
  foo: {
    baz: true
  }
}
```

...このページの設定の値は `{ runtime: 'edge', regions: ['us1', 'us2'], foo: { baz: true } }` となります。

その他の参考資料

- [Tutorial: Page options](#)

State management

クライアントオンリーなアプリを構築するのに慣れている場合、サーバーとクライアントにまたがった state management(状態管理)について怖く感じるかもしれません。このセクションでは、よくある落とし穴を回避するためのヒントを提供します。

サーバーでは state の共有を避ける

ブラウザは state を保持します(Browsers are *stateful*) — ユーザーがアプリケーションとやりとりする際に、state はメモリ内に保存されます。一方、サーバーは state を保持しません(Servers are *stateless*) — レスポンスの内容は、完全にリクエストの内容によって決定されます。

概念としては、そうです。現実では、サーバーは長い期間存在し、複数のユーザーで共有されることが多いです。そのため、共有される変数にデータを保存しないことが重要です。例えば、こちらのコードを考えてみます:

```
// @errors: 7034 7005
// file: +page.server.js
let user;

/** @type {import('./$types').PageServerLoad} */
export function load() {
    return { user };
}

/** @type {import('./$types').Actions} */
export const actions = {
    default: async ({ request }) => {
        const data = await request.formData();

        // NEVER DO THIS!
        user = {
            name: data.get('name'),
            embarrassingSecret: data.get('secret')
        };
    }
}
```

この `user` 変数はサーバーに接続する全員に共有されます。もしアリスが恥ずかしい秘密を送信し、ボブがアリスのあとにページにアクセスした場合、ボブはアリスの秘密を知ることになります（訳注：アリスやボブについては[こちら](#)）。さらに付け加えると、アリスが後でサイトに戻ってきたとき、サーバーは再起動していて彼女のデータは失われているかもしれません。

代わりに、`cookies` を使用してユーザーを認証し、データベースにデータを保存すると良いでしょう。

load に副作用を持たせない

同じ理由で、`load` 関数は 純粋(pure) であるべきです — 副作用(side-effect)を持つべきではありません（必要なときに使用する `console.log(...)` は除く）。例えば、コンポーネントで `store` の値を使用できるようにするために、`load` 関数の内側で `store` に書き込みをしたくなるかもしれません：

```

/// file: +page.js
// @filename: ambient.d.ts
declare module '$lib/user' {
    export const user: { set: (value: any) => void };
}

// @filename: index.js
// cut---
import { user } from '$lib/user';

/** @type {import('./$types').PageLoad} */
export async function load({ fetch }) {
    const response = await fetch('/api/user');

    // NEVER DO THIS!
    user.set(await response.json());
}

```

前の例と同様に、これはあるユーザーの情報をすべてのユーザーに共有される場所に置くことになります。代わりに、ただデータを返すようにしましょう…

```

/// file: +page.js
export async function load({ fetch }) {
    const response = await fetch('/api/user');

    + return {
    +     user: await response.json()
    + };
}

```

…そしてそのデータを必要とするコンポーネントに渡すか、`$page.data` を使用してください。

SSR を使用していない場合は、あるユーザーのデータを別の人へ誤って公開してしまうリスクはありません。しかし、それでも `load` 関数の中で副作用を持つべきではありません — 副作用がなければ、あなたのアプリケーションはより理解がしやすいものになります。

context と共に store を使う

独自の store が使用できないのであれば、どうやって `$page.data` や他の `app stores` を使用できるようにしているのだろう、と思うかもしれません。その答えは、サーバーの `app stores` は Svelte の `context API` を使用しているから、です — `store` は `setContext` でコンポーネントツリーにアタッチされ、`subscribe` するときは `getContext` で取得します。同じことを独自の `store` でも行うことができます：

```

/// file: src/routes/+layout.svelte
<script>
    import { setContext } from 'svelte';
    import { writable } from 'svelte/store';

    /** @type {import('./$types').LayoutData} */
    export let data;

    // store を作成し必要に応じて更新します...
    const user = writable();
    $: user.set(data.user);

```

```
// ...そして子コンポーネントがアクセスできるように context に追加します
  setContext('user', user);
</script>

/// file: src/routes/user/+page.svelte
<script>
  import { getContext } from 'svelte';

  // context から user store を取得します
  const user = getContext('user');
</script>

<p>Welcome {$user.name}</p>
```

階層が深いページやコンポーネントで context ベースの store の値を更新しても、SSR でページがレンダリングされたときは親のコンポーネントのその値には影響しません。階層が深いページやコンポーネントでその store の値を更新するときにはすでに親コンポーネントはレンダリング済だからです。ハイドレーション中の state の更新による値の 'ちらつき (flashing)' を避けるため、通常は state を上から下にコンポーネントに渡すことを推奨します。

SSR を使用していない場合 (そして将来的にも SSR を使用する必要がないという保証がある場合) は、context API を使用しなくとも、共有されるモジュールの中で state を安全に保持することができます。

コンポーネントの state は保持される

アプリケーションの中を移動するとき、SvelteKit はすでに存在するレイアウトやページコンポーネントを再利用します。例えば、このようなルート(route)があるとして…

```
/// file: src/routes/blog/[slug]/+page.svelte
<script>
  /** @type {import('./$types').PageData} */
  export let data;

  // THIS CODE IS BUGGY!
  const wordCount = data.content.split(' ').length;
  const estimatedReadingTime = wordCount / 250;
</script>

<header>
  <h1>{data.title}</h1>
  <p>Reading time: {Math.round(estimatedReadingTime)} minutes</p>
</header>

<div>{@html data.content}</div>
```

`... /blog/my-short-post` から `/blog/my-long-post` への移動は、コンポーネントの破棄や再作成を引き起こしません。この `data` プロパティ (と `data.title` と `data.content`) は変更されますが、コードは再実行されないため、`estimatedReadingTime` は再計算されません。

代わりに、その値を [リアクティブ](#) にする必要があります:

```
/// file: src/routes/blog/[slug]/+page.svelte
<script>
  /** @type {import('./$types').PageData} */
  export let data;
```

```
+ $: wordCount = data.content.split(' ').length;
+ $: estimatedReadingTime = wordCount / 250;
</script>
```

このようにコンポーネントを再利用すると、サイドバースクロールの state などが保持され、変化する値の間で簡単にアニメーションを行うことができます。しかし、ナビゲーション時にコンポーネントを完全に破棄して再マウントする必要がある場合、このパターンを使用できます：

```
{#key $page.url.pathname}
  <BlogPost title={data.title} content={data.title} />
{/key}
```

state を URL に保存する

もし、テーブルのフィルターやソートルールなどのように、リロード後も保持されるべき state、または SSR に影響を与える state がある場合、URL search パラメータ（例：`?sort=price&order=ascending`）はこれらを置くのに適した場所です。これらは `` や `<form action="...">` の属性に置いたり、`goto('?key=value')` を使用してプログラミ的に設定することもできます。`load` 関数の中では `url` パラメータを使用してアクセスでき、コンポーネントの中では `$page.url.searchParams` でアクセスできます。

一時的な state は snapshots に保存する

'アコーディオンは開いているか?'などの一部の UI の state は一時的なものですぐに捨てられます — ユーザーがページを移動したり更新したりして、その state が失われたとしてもそれほど問題ではありません。ユーザーが別のページに移動して戻ってきたときにデータを保持しておきたい場合もありますが、そのような state を URL や database に保存するのは行き過ぎでしょう。そういう場合のために、SvelteKit [snapshots](#) を提供しています。これによってコンポーネントの state を履歴エントリーに関連付けることができます。

アプリをビルドする

SvelteKit アプリのビルドは 2 つのステージで行われます。どちらも `vite build` (通常は `npm run build` を経由します) を実行したときに行われます。

まず最初に、Vite がサーバーのコード、ブラウザのコード、service worker(もしあれば) の、最適化された本番向けビルドを作成します。必要に応じて、このステージで [プリレンダリング](#) が実行されます。

次に、*adapter* がこの本番向けビルドをあなたがデプロイしたいターゲットの環境向けに調整します — これについての詳細は以降のページにございます。

ビルド中に

SvelteKit はビルド中に、解析のために `+page/layout(.server).js` ファイル (とそこにインポートされている全てのファイル) を読み込みます。このステージで読み込まれるべきでないコードがある場合は、`$app/environment` からインポートする `building` が `false` であることをチェックするコードを追加してください:

```
+import { building } from '$app/environment';
import { setupMyDatabase } from '$lib/server/database';

+if (!building) {
    setupMyDatabase();
}

export function load() {
    // ...
}
```

アプリのプレビュー

ビルド後、`vite preview` (`npm run preview` 経由) を使用してローカルで本番向けビルドを確認することができます。これは Node 上でアプリを実行しているので、デプロイされるアプリの完全な再現ではないことにご注意ください。`platform` オブジェクトなどの *adapter* 固有の調整はプレビューには適用されません。

Adapters

SvelteKit アプリをデプロイする前に、それをデプロイ先の環境に 合わせる(*adapt*) 必要があります。adapter は、ビルドされたアプリをインプットとして受け取りデプロイ用のアウトプットを生成する小さなプラグインです。

様々なプラットフォーム向けの公式の adapter があります。これらについて以降のページにドキュメントがあります。

- `@sveltejs/adapter-cloudflare` for Cloudflare Pages
- `@sveltejs/adapter-cloudflare-workers` for Cloudflare Workers
- `@sveltejs/adapter-netlify` for Netlify
- `@sveltejs/adapter-node` for Node servers
- `@sveltejs/adapter-static` for static site generation (SSG)
- `@sveltejs/adapter-vercel` for Vercel

加えて、他のプラットフォーム向けに、[コミュニティ](#)によって提供されている adapter もございます。

adapter を使用する

adapter は `svelte.config.js` に指定します。

```
/// file: svelte.config.js
// @filename: ambient.d.ts
declare module 'svelte-adapter-foo' {
    const adapter: (opts: any) => import('@sveltejs/kit').Adapter;
    export default adapter;
}

// @filename: index.js
// cut---
import adapter from 'svelte-adapter-foo';

/** @type {import('@sveltejs/kit').Config} */
const config = {
    kit: {
        adapter: adapter({
            // adapter options go here
        })
    }
};

export default config;
```

プラットフォーム固有の情報

adapter によっては、リクエストに関する追加情報にアクセスすることができます。例えば、Cloudflare Workers の場合は KV namespaces などを含む `env` オブジェクトにアクセスできます。これは `hooks` や [サーバールート\(server routes\)](#) で使用される `RequestEvent` に、`platform` プロパティとして渡されます。詳しくは、各 adapter のドキュメントをご参照ください。

[Go to TOC](#)

ゼロコンフィグデプロイ

`npm create svelte@latest` で新しい SvelteKit プロジェクトを作成した場合、デフォルトで `adapter-auto` がインストールされます。この adapter はデプロイ時に自動でサポートされている環境に合った adapter をインストールして使用します：

- `@sveltejs/adapter-cloudflare` for [Cloudflare Pages](#)
- `@sveltejs/adapter-netlify` for [Netlify](#)
- `@sveltejs/adapter-vercel` for [Vercel](#)
- `svelte-adapter-azure-swa` for [Azure Static Web Apps](#)
- `svelte-kit-sst` for [AWS via SST](#)

デプロイするターゲットの環境が決まつたら、`devDependencies` に適切な adapter をインストールすることを推奨します。これにより、`lockfile` に adapter が追加されるため、CI でのインストール時間が少し改善されます。

環境固有の設定

`adapter-vercel` や `adapter-netlify` の `{ edge: true }` のような設定オプションを追加したければ、そのオプションを持つ adapter をインストールしなければなりません。`adapter-auto` はそれらのオプションを受け付けません。

コミュニティ adapter を追加する

追加の adapter にゼロコンフィグサポートを追加するには、[adapters.js](#) を編集し、pull request を開きます。

Node サーバー

スタンドアロンな Node サーバーを作る場合は、`adapter-node` を使います。

使い方

`npm i -D @sveltejs/adapter-node` を実行してインストールし、`svelte.config.js` にこの adapter を追加します：

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-node';

export default {
    kit: {
        adapter: adapter()
    }
};
```

デプロイ(Deploying)

まず、`npm run build` でアプリをビルドします。これによって adapter のオプションで指定した出力ディレクトリ (デフォルトは `build`) に本番環境用のサーバーが作成されます。

アプリケーションを実行するには、出力ディレクトリ、プロジェクトの `package.json`、`node_modules` の本番向けの依存関係(production dependencies)が必要です。本番向けの依存関係は、`package.json` と `package-lock.json` をコピーしてから `npm ci --omit dev` を実行すると生成することができます (あなたのアプリが何の依存関係も持たない場合はこのステップをスキップできます)。そして、このコマンドでアプリを起動することができます：

```
node build
```

`Rollup` を使うと開発用の依存関係(Development dependencies)もアプリにバンドルされます。パッケージをバンドルするか外部化するかコントロールするには、そのパッケージを `package.json` の `devDependencies` か `dependencies` にそれぞれ配置します。

環境変数

`dev` と `preview` のときは、SvelteKit は `.env` ファイル (または `.env.local` や `.env.[mode]`、`Vite` によって決定されているもの) から環境変数を読み取ります。

プロダクションでは、`.env` ファイルは自動的に読み取れません。そうするには、プロジェクトに `dotenv` をインストールします…

```
npm install dotenv
```

…そしてビルドされたアプリを実行する前にそれを呼び出します：

```
-node build
+node -r dotenv/config build
```

PORT と HOST

デフォルトでは、サーバーは `0.0.0.0`、port 3000 でコネクションを受け付けます。これは環境変数の `PORT` と `HOST` を使ってカスタマイズすることができます。

```
HOST=127.0.0.1 PORT=4000 node build
```

ORIGIN、PROTOCOL_HEADER、HOST_HEADER

HTTP は SvelteKit に現在リクエストされている URL を知るための信頼できる方法を提供しません。アプリがホストされている場所を Sveltekit に伝える最も簡単な方法は、環境変数 `ORIGIN` を設定することです：

```
ORIGIN=https://my.site node build
# or e.g. for local previewing and testing
ORIGIN=http://localhost:3000 node build
```

これにより、パス名 `/stuff` に対するリクエストは正しく `https://my.site/stuff` に解決されます。別の方法として、リクエストプロトコルとホストを SvelteKit に伝えるヘッダーを指定し、そこから origin URL を組み立てることもできます：

```
PROTOCOL_HEADER=x-forwarded-proto HOST_HEADER=x-forwarded-host node build
```

`x-forwarded-proto` と `x-forwarded-host` は事実上の標準となっているヘッダーで、リバースプロキシー（ロードバランサーや CDN などを考えてみてください）を使用している場合に、オリジナルのプロトコルとホストを転送します。これらの変数は、あなたのサーバーが信頼できるリバースプロキシーの後ろにある場合にのみ設定すべきです。そうしないと、クライアントがこれらのヘッダーを偽装することが可能になってしまいます。

`adapter-node` があなたのデプロイの URL を正しく判断することができない場合、`form actions` を使用するとこのエラーが発生することがあります：

クロスサイトの POST フォーム送信は禁止されています

ADDRESS_HEADER と XFF_DEPTH

`hooks` とエンドポイントに渡される `RequestEvent` オブジェクトにはクライアントの IP アドレスを返す `event.getClientAddress()` 関数が含まれています。デフォルトでは、これは接続中の `remoteAddress` です。もしサーバーが1つ以上のプロキシー（例えばロードバランサー）の後ろにある場合、この値はクライアントの IP アドレスではなく、最も内側にあるプロキシーの IP アドレスを含むことになるため、アドレスを読み取るために `ADDRESS_HEADER` を指定する必要があります：

ADDRESS_HEADER=True-Client-IP node build

ヘッダーは簡単に偽装されます。 `PROTOCOL_HEADER` や `HOST_HEADER` と同様、これらを設定する前に[自分が何をしているのか知るべき](#)です。

`ADDRESS_HEADER` が `X-Forwarded-For` の場合、ヘッダーの値にはカンマで区切られた IP アドレスのリストが含まれます。環境変数 `XFF_DEPTH` には、あなたのサーバーの前に信頼できるプロキシーがいくつあるか指定する必要があります。例えば、3つの信頼できるプロキシーがある場合、プロキシー3はオリジナルのコネクションと最初の2つのプロキシーのアドレスを転送します：

<client address>, <proxy 1 address>, <proxy 2 address>

一番左のアドレスを読め、というガイドもありますが、これだと[スプーフィング\(なりすまし\)](#)に対し脆弱なままで：

<spoofed address>, <client address>, <proxy 1 address>, <proxy 2 address>

代わりに、信頼できるプロキシーの数を考慮して右から読み込みます。この場合、`XFF_DEPTH=3` を使用します。

もし、一番左のアドレスを読む必要がある場合 (そしてスプーフィングを気にしない場合) — 例えば、位置情報サービスを提供する場合、つまり IP アドレスが信頼できることよりもリアルであることが重要な場合、アプリの中で `x-forwarded-for` ヘッダーを検査することでそれが可能です。

BODY_SIZE_LIMIT

ストリーミング中も含め、受け付けるリクエストボディの最大サイズを `byte` で指定します。デフォルトは 512kb です。もっと高度な設定が必要な場合は、このオプションの値を 0 にして無効化し、`handle` にカスタムのチェックを実装することができます。

Options

この `adapter` は様々なオプションで設定を行うことができます：

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-node';

export default {
  kit: {
    adapter: adapter({
      // default options are shown
      out: 'build',
      precompress: false,
      envPrefix: '',
      polyfill: true
    })
  }
}
```

```

    })
};


```

out

サーバーをビルドするディレクトリです。デフォルトは `build` です。つまり、`node build` を指定すると、サーバが作成されローカルで起動します。

precompress

アセットやプリレンダリングされたページを `gzip` や `brotli` を使って事前圧縮(precosmpress)するのを有効にします。デフォルトは `false` です。

envPrefix

デプロイの設定に使用される環境変数の名前を変更する必要がある場合(例えば、あなたのコントロール下にない環境変数との競合を解消するため)、接頭辞(prefix)を指定することができます:

```
envPrefix: 'MY_CUSTOM_';
```

```

MY_CUSTOM_HOST=127.0.0.1 \
MY_CUSTOM_PORT=4000 \
MY_CUSTOM_ORIGIN=https://my.site \
node build

```

polyfill

ビルトが存在しないモジュールの `polyfill` を読み込むかどうかをコントロールします。デフォルトは `true` で、Node 18.11 以降を使用している場合にのみ無効にしてください。

注意事項: Node のビルトインの `crypto global` を Node 18 で使用するには、`--experimental-global-webcrypto` フラグを使用する必要があります。Node 20 ではこのフラグは必要ありません。

カスタムサーバー

この `adapter` は、ビルトのディレクトリに2つのファイルを作成します — `index.js` と `handler.js` です。デフォルトのビルトのディレクトリを使用している場合、`node build` などで `index.js` を実行すると、設定された `port` でサーバーが起動されます。

別のこととして、[Express](#)、[Connect](#)、[Polka](#) (またはビルトインの `http.createServer`) を使用するためのハンドラーをエクスポートする `handler.js` ファイルをインポートし、独自のサーバーをセットアップすることもできます。

```

// @errors: 2307 7006
/// file: my-server.js
import { handler } from './build/handler.js';
import express from 'express';

const app = express();

```

```
// add a route that lives separately from the SvelteKit app
app.get('/healthcheck', (req, res) => {
  res.end('ok');
});

// let SvelteKit handle everything else, including serving prerendered pages and
// static assets
app.use(handler);

app.listen(3000, () => {
  console.log('listening on port 3000');
});
```

トラブルシューティング

サーバーが終了する前にクリーンアップするための hook はありますか？

SvelteKit にはこれに対応するためのビルトインで組み込まれているものはありません。なぜなら、このようなクリーンアップの hook はあなたの実行環境に大きく依存しているからです。Node の場合は、ビルトインの `process.on(..)` を使用して、サーバーが終了する前に実行されるコールバックを実装することができます：

```
// @errors: 2304 2580
function shutdownGracefully() {
  // anything you need to clean up manually goes in here
  db.shutdown();
}

process.on('SIGINT', shutdownGracefully);
process.on('SIGTERM', shutdownGracefully);
```

Static site generation

SvelteKit を static site generator (SSG) として使用するには、`adapter-static` を使用します。

この adapter はサイト全体を静的なファイルのコレクションとしてプリレンダリングします。もし、一部のページのみをプリレンダリングして他のページは動的にサーバーでレンダリングしたい場合、別の adapter と `prerender` オプションを組み合わせて使用する必要があります。

使い方

`npm i -D @sveltejs/adapter-static` を実行してインストールし、`svelte.config.js` にこの adapter を追加します：

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-static';

export default {
  kit: {
    adapter: adapter({
      // default options are shown. On some platforms
      // these options are set automatically – see below
      pages: 'build',
      assets: 'build',
      fallback: undefined,
      precompress: false,
      strict: true
    })
  }
};
```

…そして `prerender` オプションを最上位のレイアウト(root layout)に追加します：

```
/// file: src/routes/+layout.js
// This can be false if you're using a fallback (i.e. SPA mode)
export const prerender = true;
```

SvelteKit の `trailingSlash` オプションを、あなたの環境に対して適切に設定しなければなりません。もしあなたのホスト環境が、`/a` へのリクエストを受け取ったときに `/a.html` をレンダリングしない場合、`/a/index.html` を作成するために最上位のレイアウト(root layout)で `trailingSlash: 'always'` を設定する必要があります。

ゼロコンフィグサポート

ゼロコンフィグサポートがあるプラットフォームもあります (将来増える予定)：

- [Vercel](#)

これらのプラットフォームでは、adapter のオプションを省略することで、`adapter-static` が最適な設定を提供できるようになります：

```
// file: svelte.config.js
export default {
  kit: {
    -   adapter: adapter({...})
    +   adapter: adapter()
  }
};
```

Options

pages

プリレンダリングされたページを書き込むディレクトリです。デフォルトは `build` です。

assets

静的なアセット (`static` のコンテンツと、SvelteKit が生成するクライアントサイドの JS と CSS) を書き込むディレクトリです。通常は `pages` と同じにするべきで、デフォルトでは、それがどんな値だったとしても、`pages` の値になります。しかし、まれな状況では、出力されるページとアセットを別々の場所にする必要があるかもしれません。

fallback

SPA モード向けにフォールバックページ(fallback page)を指定します。例えば、`index.html` や `200.html`、`404.html` などです。

precompress

`true` の場合、`brotli` や `gzip` でファイルを事前圧縮(precosmress)します。これによって `.br` ファイルや `.gz` ファイルが生成されます。

strict

デフォルトでは `adapter-static` は、アプリの全てのページとエンドポイント (もしあれば) がプリレンダリングされているか、もしくは `fallback` オプションが設定されているかをチェックします。このチェックは、アプリの一部が最終的な出力に含まれずアクセスできない状態なのに誤って公開されてしまうのを防ぐために存在します。もし、それでも良いことがわかっている場合 (例えばあるページが条件付きでしか存在しない場合)、`strict` を `false` に設定してこのチェックをオフにすることができます。

GitHub Pages

GitHub Pages 向けにビルドするときは、`paths.base` をあなたのリポジトリ名に合わせて更新してください。サイトが `root` からではなく <https://your-username.github.io/your-repo-name> から提供されるためです。

GitHub が提供する Jekyll が、あなたのサイトを管理するのを防ぐために、空の `.nojekyll` ファイルを `static` フォルダに追加する必要があります。

GitHub Pages 向けの設定は以下のようになるでしょう:

```
// @errors: 2307 2322
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-static';

const dev = process.argv.includes('dev');

/** @type {import('@sveltejs/kit').Config} */
const config = {
    kit: {
        adapter: adapter(),
        paths: {
            base: dev ? '' : process.env.BASE_PATH,
        }
    }
};
```

GitHub actions を使用して、サイトが変更されたときに自動で GitHub Pages にデプロイすることができます。サンプルの workflow はこちらです:

```
/// file: .github/workflows/deploy.yml
name: Deploy to GitHub Pages

on:
  push:
    branches: 'main'

jobs:
  build_site:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3

        # If you're using pnpm, add this step then change the commands and cache key
        # below to use `pnpm`
        # - name: Install pnpm
        #   uses: pnpm/action-setup@v2
        #   with:
        #     version: 8

      - name: Install Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 18
          cache: npm

      - name: Install dependencies
        run: npm install

      - name: build
        env:
          BASE_PATH: '/your-repo-name'
        run:
          npm run build
          touch build/.nojekyll

      - name: Upload Artifacts
        uses: actions/upload-pages-artifact@v1
        with:
          # this should match the `pages` option in your adapter-static options
          path: 'build/'
```

```
deploy:  
  needs: build_site  
  runs-on: ubuntu-latest  
  
  permissions:  
    pages: write  
    id-token: write  
  
  environment:  
    name: github-pages  
    url: ${{ steps.deployment.outputs.page_url }}  
  
steps:  
  - name: Deploy  
    id: deployment  
    uses: actions/deploy-pages@v1
```

Single-page apps

SvelteKit アプリは、どんな adapter を使っていても、最上位のレイアウト(root layout)で SSR を無効にすることで、完全にクライアントレンダリングされるシングルページアプリ (SPA) にすることができます。

```
/// file: src/routes/+layout.js
export const ssr = false;
```

ほとんどの場合、これはおすすめできません: SEO に悪影響を与え、知覚的なパフォーマンスが低下する傾向があり、もし JavaScript が失敗したり無効になっていたりする場合 (これは[あなたが思うより頻繁に発生します](#))、ユーザーはアプリにアクセスできなくなります.

サーバーサイドのロジック (すなわち `+page.server.js`、`+layout.server.js`、`+server.js` ファイル) がない場合は、`adapter-static` を使い フォールバックページ(fallback page) を追加することで SPA を作ることができます。

使い方

`npm i -D @sveltejs/adapter-static` でインストールし、それから `svelte.config.js` にこの adapter と以下のオプションを追加します:

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-static';

export default {
    kit: {
        adapter: adapter({
            fallback: '200.html' // may differ from host to host
        })
    }
};
```

フォールバックページ(`fallback page`)とは、SvelteKit がページテンプレート(例: `app.html`)から作成する HTML ページで、アプリをロードし正しいルート(routes)にナビゲートします。例えば、静的 web ホスティングである [Surge](#) では、静的なアセットやプリレンダリングページに対応しないリクエストを処理する `200.html` ファイルを追加する必要があります。

ホスティング環境によっては `index.html` であったり全く別のものであったりします — 使いたいプラットフォームのドキュメントをご参照ください。

Apache

SPA を [Apache](#) で実行する場合は、`static/.htaccess` ファイルを追加し、リクエストをフォールバックページ(fallback page)にルーティングする必要があります:

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteBase /
  RewriteRule ^200\.html$ - [L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule . /200.html [L]
</IfModule>
```

ページを個別にプリレンダリングする

特定のページをプリレンダリングしたい場合、アプリのその部分だけ `ssr` と `prerender` を再び有効にします：

```
/// file: src/routes/my-prerendered-page/+page.js
export const prerender = true;
export const ssr = true;
```
Go to TOC

```

# Cloudflare Pages

Cloudflare Pages にデプロイする場合は、`adapter-cloudflare` を使用します。

`adapter-auto` を使用している場合、この adapter は自動でインストールされますが、それよりもこの adapter 自体をプロジェクトに追加することをおすすめします。`event.platform` が自動で型付けされるからです。

## 比較

- `adapter-cloudflare` – SvelteKit の全ての機能をサポートします；Cloudflare Pages 向けにビルドします
- `adapter-cloudflare-workers` – SvelteKit の全ての機能をサポートします；Cloudflare Workers 向けにビルドします
- `adapter-static` – クライアントサイドの静的なアセットを生成するのみです；Cloudflare Pages と互換性があります

特別な理由が無い限り、`adapter-cloudflare-workers` ではなく、`adapter-cloudflare` を使用することをおすすめします。どちらの adapter も機能としては同等ですが、Cloudflare Pages は、GitHub インテグレーションによる自動ビルドや自動デプロイ、プレビューデプロイ、即時ロールバックなどの機能を提供します。

## 使い方

`npm i -D @sveltejs/adapter-cloudflare` を実行してインストールし、`svelte.config.js` にこの adapter を追加します：

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-cloudflare';

export default {
 kit: {
 adapter: adapter({
 // See below for an explanation of these options
 routes: {
 include: ['/*'],
 exclude: ['<all>']
 }
 })
 }
};
```

## Options

The `routes` option allows you to customise the `_routes.json` file generated by `adapter-cloudflare`.

- `include` defines routes that will invoke a function, and defaults to `['/*']`

- `exclude` defines routes that will *not* invoke a function — this is a faster and cheaper way to serve your app's static assets. This array can include the following special values:

- `<build>` contains your app's build artifacts (the files generated by Vite)
- `<files>` contains the contents of your `static` directory
- `<prerendered>` contains a list of prerendered pages
- `<all>` (the default) contains all of the above

You can have up to 100 `include` and `exclude` rules combined. Generally you can omit the `routes` options, but if (for example) your `<prerendered>` paths exceed that limit, you may find it helpful to manually create an `exclude` list that includes `'/articles/*'` instead of the auto-generated `['/articles/foo', '/articles/bar', '/articles/baz', ...]`.

## デプロイメント(Deployment)

Cloudflare Pages の始め方は、[Get Started Guide](#) に従ってください。

プロジェクトのセッティングを設定するときは、以下のセッティングを使用しなければなりません:

- **Framework preset** – None
- **Build command** – `npm run build` または `vite build`
- **Build output directory** – `.svelte-kit/cloudflare`
- **Environment variables**
  - `NODE_VERSION` : 16

"production" 環境と "preview" 環境のどちらにも、環境変数 `NODE_VERSION` を追加する必要があります。これは、プロジェクトセットアップ時や、後で Pages プロジェクトのセッティングで追加できます。SvelteKit は Node 16.14 以降を要求するため、`NODE_VERSION` の値として 16 を使用する必要があります。

## Bindings

`env` オブジェクトにはあなたのプロジェクトの `bindings` が含まれており、KV/DO namespaces などで構成されています。これは `platform` プロパティを介して `context` や `caches` と一緒に SvelteKit に渡されます。つまり、`hooks` とエンドポイントでアクセスできるということです:

```
// @errors: 7031
export async function POST({ request, platform }) {
 const x = platform.env.YOUR_DURABLE_OBJECT_NAMESPACE.idFromName('x');
}
```

環境変数については、SvelteKit のビルトインの `$env` モジュールを優先的に使用したほうが良いでしょう。

これらの型をアプリで使えるようにするには、`src/app.d.ts` でこれらを参照します:

```
/// file: src/app.d.ts
declare global {
 namespace App {
 interface Platform {
 + env?: {
 + YOUR_KV_NAMESPACE: KVNamespace;
 + YOUR_DURABLE_OBJECT_NAMESPACE: DurableObjectNamespace;
 };
 }
 }
}

export {};
```

`platform.env` は本番向けビルトでのみ利用することができます。ローカルでテストするには [wrangler](#) を使ってください。

## Notes

プロジェクトの root にある `/functions` ディレクトリに含まれる関数はデプロイメントには含まれず、[1つの \\_worker.js ファイル](#) にコンパイルされます。関数は、あなたの SvelteKit アプリの [サーバーエンドポイント\(server endpoints\)](#) として実装する必要があります。

Cloudflare Pages 固有の `_headers` ファイルと `_redirects` ファイルについては、`/static` フォルダに置くことで、静的アセットのレスポンス（画像など）に使用することができます。

しかし、SvelteKit が動的にレンダリングするレスポンスには効果がありません。この場合にカスタムヘッダーやリダイレクトレスポンスを返すには、[サーバーエンドポイント\(server endpoints\)](#) や `handle` hook から返す必要があります。

## トラブルシューティング

### 外部の資料

Cloudflare の、SvelteKit サイトのデプロイに関するドキュメントをご参考ください。

## ファイルシステムにアクセスする

Serverless/Edge 環境では、`fs.readFileSync` などのメソッドでファイルシステムにアクセスすることはできません。もしこのような方法でファイルにアクセスする必要がある場合、アプリのビルト中に[プリレンダリング](#)でこれを行ってください。例えば、ブログを持っていて、CMS でコンテンツを管理したくない場合、コンテンツをプリレンダリングし（またはコンテンツを取得するエンドポイントをプリレンダリングし）、新しいコンテンツを追加するたびにブログを再デプロイする必要があります。

---

[Go to TOC](#)

# Cloudflare Workers

Cloudflare Workers にデプロイする場合は、`adapter-cloudflare-workers` を使用します。

この adapter を使用する特別な理由がない限り、代わりに `adapter-cloudflare` を使用することをおすすめします。

Wrangler v2 が必要です。

## 使い方

`npm i -D @sveltejs/adapter-cloudflare-workers` を実行してインストールし、`svelte.config.js` にこの adapter を追加します：

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-cloudflare-workers';

export default {
 kit: {
 adapter: adapter()
 }
};
```

## 基本設定

この adapter では、プロジェクトの root に `wrangler.toml` ファイルを置くことを想定しています。内容としては以下のようなものです：

```
/// file: wrangler.toml
name = "<your-service-name>"
account_id = "<your-account-id>

main = "./.cloudflare/worker.js"
site.bucket = "./.cloudflare/public"

build.command = "npm run build"

compatibility_date = "2021-11-12"
workers_dev = true
```

`<your-service-name>` は何でも構いません。`<your-account-id>` は、[Cloudflare dashboard](#) にログインし、URL の末尾から取得できます：

`https://dash.cloudflare.com/<your-account-id>`

.cloudflare ディレクトリ (または main と site.bucket に指定したディレクトリ) を .gitignore に追加する必要があります。

wrangler をインストールしてログインする必要がありますが、もしまだやっていなければ:

```
npm i -g wrangler
wrangler login
```

その後、アプリをビルドしデプロイすることができます:

```
wrangler publish
```

## カスタム設定

wrangler.toml 以外の設定ファイルを使用したい場合は、以下のようにします:

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-cloudflare-workers';

export default {
 kit: {
 adapter: adapter({ config: '<your-wrangler-name>.toml' })
 }
};
```

## 環境変数

KV/DO namespacesなどを含んでいる env オブジェクトは、context や caches と一緒に platform プロパティ経由で SvelteKit に渡されます。つまり、hooks とエンドポイントの中でアクセスできるということです:

```
// @errors: 7031
export async function POST({ request, platform }) {
 const x = platform.env.YOUR_DURABLE_OBJECT_NAMESPACE.idFromName('x');
}
```

これらの型をアプリで使えるようにするには、src/app.d.ts でこれらを参照します:

```
/// file: src/app.d.ts
declare global {
 namespace App {
 interface Platform {
 + env?: {
 + YOUR_KV_NAMESPACE: KVNamespace;
 + YOUR_DURABLE_OBJECT_NAMESPACE: DurableObjectNameSpace;
 };
 }
 }
}

export {};
```

`platform.env` は本番向けビルドでのみ利用することができます。ローカルでテストするには [Wrangler](#) を使ってください

## トラブルシューティング

### Worker size limits

Workers にデプロイする場合、SvelteKit が生成したサーバーは 1つのファイルにバンドルされます。minify 後に Worker が [そのサイズの上限](#) を超過する場合、Wrangler が Worker の公開に失敗します。通常、この制限に引っかかることはほとんどありませんが、一部の大きいライブラリではこれが発生することがあります。その場合、大きいライブラリをクライアントサイドでのみインポートするようにすることで、Worker のサイズを小さくすることができます。詳細は [FAQ](#) をご覧ください。

### ファイルシステムにアクセスする

Serverless/Edge 環境では、`fs.readFileSync` などのメソッドでファイルシステムにアクセスすることはできません。もしこのような方法でファイルにアクセスする必要がある場合、アプリのビルド中に[プリレンダリング](#)でこれを行ってください。例えば、ブログを持っていて、CMS でコンテンツを管理たくない場合、コンテンツをプリレンダリングし（またはコンテンツを取得するエンドポイントをプリレンダリングし）、新しいコンテンツを追加するたびにブログを再デプロイする必要があります。

# Netlify

Netlify にデプロイする場合は、`adapter-netlify` を使用します。

`adapter-auto` を使用している場合、この adapter は自動でインストールされますが、この adapter 自体をプロジェクトに追加すれば Netlify 固有のオプションを指定できるようになります。

## 使い方

`npm i -D @sveltejs/adapter-netlify` を実行してインストールし、`svelte.config.js` にこの adapter を追加します：

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-netlify';

export default {
 kit: {
 // default options are shown
 adapter: adapter({
 // if true, will create a Netlify Edge Function rather
 // than using standard Node-based functions
 edge: false,

 // if true, will split your app into multiple functions
 // instead of creating a single one for the entire app.
 // if `edge` is true, this option cannot be used
 split: false
 })
 }
};
```

そして、プロジェクトの root に `netlify.toml` ファイルを置くのを忘れないでください。このファイルの `build.publish` に基づいて静的なアセットをどこに書き込むか決定します。こちらのサンプルの設定をご覧ください：

```
[build]
 command = "npm run build"
 publish = "build"
```

`netlify.toml` ファイルが見つからない、もしくは `build.publish` の値が見つからない場合、`"build"` のデフォルト値が使用されます。Netlify の UI で `publish` ディレクトリを他の場所に設定する場合は、`netlify.toml` にも同じ場所を設定するか、`"build"` のデフォルト値を使用する必要があることにご注意ください。

## Node version

新しいプロジェクトではデフォルトで Node 16 が使用されます。しかし、少し前に作成したプロジェクトをアップグレードする場合、古いバージョンで止まってしまうかもしれません。手動で Node 16 以降を指定する場合、詳細は [Netlify のドキュメント](#) をご参照ください。

## Netlify Edge Functions (beta)

SvelteKit はベータリリースの [Netlify Edge Functions](#) をサポートしています。`adapter` 関数に `edge: true` オプションを渡すと、サイト訪問者に近い場所にデプロイされる Deno ベースの edge function でサーバーサイドレンダリングが行われるようになります。`false` を設定した場合（デフォルト）、サイトはスタンダードな Node ベースの Netlify Functions にデプロイされます。

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-netlify';

export default {
 kit: {
 adapter: adapter({
 // will create a Netlify Edge Function using Deno-based
 // rather than using standard Node-based functions
 edge: true
 })
 }
};
```

## SvelteKit の機能を代替する Netlify の機能

Netlify の機能に依存することなく、SvelteKit が直接提供する機能を使ってアプリを構築することができます。こういった機能は SvelteKit のほうを選択すると、開発モードでその機能を使用でき、インテグレーションテストが可能になり、Netlify から切り替えることになった場合に他の adapter で動作させることができます。しかし、シナリオによっては Netlify のほうの機能を使用したほうが有益な場合もあります。例えば、すでに Netlify でホストされているアプリを SvelteKit に移行する場合です。

## リダイレクトルール(Redirect rules)

コンパイル時に、リダイレクトルールは自動で `_redirects` ファイルに追記されます（もし存在しない場合は、作成されます）。つまり：

- `_redirects` のほうが優先度が高いため、`netlify.toml` にある `[[redirects]]` には決してマッチしません。そのため、ルールは常に `_redirects` ファイルに記載してください。
- `_redirects` には、`/* /foobar/:splat` のようなカスタムの "catch all" ルールを置くべきではありません。そうしないと、自動で追加されたルールが適用されなくなります。Netlify は最初にマッチしたルールだけを処理するからです。

## Netlify Forms

1. こちらにあるように、例えば `/routes/contact/+page.svelte` に、Netlify HTML form を作成します。（hidden の `form-name` input 要素を追加するのを忘れずに！）
2. Netlify の build bot はデプロイ時にあなたの HTML ファイルをパースします。つまり、あなたの form は HTML としてプリレンダリングされるようにしておかなければなりません。あなたの `contact.svelte` に `export const prerender = true` を追加してそのページだけプリレンダリングするか、または `kit.prerender.force: true` オプションを設定して全てのページをプリレンダリングするようにしておくか、で対応できます。

3. あなたの Netlify form に `<form netlify ... action="/success">` のようなカスタムの成功メッセージがある場合、それに対応する `/routes/success/+page.svelte` が存在しプリレンダリングされるか確認してください。

## Netlify Functions

この adapter によって、SvelteKit エンドポイントは [Netlify Functions](#) としてホストされます。Netlify function ハンドラーには追加のコンテキストがあり、[Netlify Identity](#) 情報が含まれています。このコンテキストは、あなたの `hooks` や `+page.server` と `+layout.server` エンドポイントの中で `event.platform.context` フィールドを介してアクセスできます。adapter config の `edge` プロパティが `false` の場合は [serverless functions](#)、`true` の場合は [edge functions](#) となります。

```
// @errors: 2705 7006
/// file: +page.server.js
export const load = async (event) => {
 const context = event.platform.context;
 console.log(context); // shows up in your functions log in the Netlify app
};
```

さらに、ディレクトリを追加して `netlify.toml` に設定を追加することで、独自の Netlify functions を追加することができます。例えば:

```
[build]
 command = "npm run build"
 publish = "build"

[functions]
 directory = "functions"
```

## トラブルシューティング

### ファイルシステムにアクセスする

Serverless/Edge 環境では、`fs.readFileSync` などのメソッドでファイルシステムにアクセスすることはできません。もしこのような方法でファイルにアクセスする必要がある場合、アプリのビルド中に[プリレンダリング](#)でこれを行ってください。例えば、ブログを持っていて、CMS でコンテンツを管理したくない場合、コンテンツをプリレンダリングし（またはコンテンツを取得するエンドポイントをプリレンダリングし）、新しいコンテンツを追加するたびにブログを再デプロイする必要があります。

# Vercel

Vercel にデプロイする場合は、`adapter-vercel` を使用します。

`adapter-auto` を使用している場合、この adapter は自動でインストールされますが、この adapter 自体をプロジェクトに追加すれば Vercel 固有のオプションを指定できるようになります。

## 使い方

`npm i -D @sveltejs/adapter-vercel` を実行してインストールし、`svelte.config.js` にこの adapter を追加します：

```
// @errors: 2307 2345
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-vercel';

export default {
 kit: {
 adapter: adapter({
 // 以下の 'デプロイメントの設定' セクションを参照
 })
 }
};
```

## デプロイメントの設定

Vercel にルート(routes)を function としてデプロイする方法をコントロールするには、デプロイメントの設定を、上記に示すオプションか、`+server.js`、`+page(.server).js`、`+layout(.server).js` ファイルの中の `export const config` を使用して、行うことができます。

例えば、アプリの一部を [Edge Functions](#) としてデプロイして…

```
/// file: about/+page.js
/** @type {import('@sveltejs/adapter-vercel').Config} */
export const config = {
 runtime: 'edge'
};
```

…他の部分を [Serverless Functions](#) としてデプロイすることができます (layout の内側の `config` は、すべての子ページに適用されます)：

```
/// file: admin/+layout.js
/** @type {import('@sveltejs/adapter-vercel').Config} */
export const config = {
 runtime: 'nodejs18.x'
};
```

以下のオプションはすべての function に適用されます：

- `runtime` : 'edge'、'nodejs16.x'、'nodejs18.x'。デフォルトでは、プロジェクトの Node のバージョンに応じて adapter が 'nodejs16.x' か 'nodejs18.x' を選択します。プロジェクトの Node バージョンは Vercel のダッシュボードから設定することができます。
- `regions` : edge network regions の配列 (serverless functions のデフォルトは `["iad1"]` ) か、`runtime` が edge (デフォルト) の場合は `'all'` です。serverless functions の場合の複数の regions のサポートは Enterprise Plan のみです。
- `split` : `true` の場合、ルート(route)は個別の function としてデプロイされます。`split` を adapter レベルで `true` にする場合、すべてのルート(route)が個別の function としてデプロイされます。

加えて、以下のオプションは edge function に適用されます:

- `external` : esbuild が function をバンドルする際に外部(external)として扱う依存関係(dependencies)の配列です。Node の外側で実行されないオプションの依存関係(optional dependencies)を除外したいときにのみ使用してください

そして以下のオプションは serverless function に適用されます:

- `memory` : function で利用できるメモリ量です。デフォルトは `1024 Mb` で、`128 Mb` まで減らすことができます。また、Pro または Enterprise アカウントの場合は、`3008 Mb` まで増やすことができます。間隔は `64Mb` 単位です。
- `maxDuration` : function の最大実行時間。デフォルトで、Hobby アカウントの場合は `10` 秒、Pro の場合は `60`、Enterprise の場合は `900` です。
- `isr` : Incremental Static Regeneration の設定、詳細は後述

function から特定の region のデータにアクセスする必要がある場合は、パフォーマンスを最適化するためそれと同じ region (またはその知覚) にデプロイすることをおすすめします。

## Incremental Static Regeneration

Vercel は Incremental Static Regeneration (ISR) をサポートしており、これにより、プリレンダリングコンテンツが持つパフォーマンスとコストの利点と、ダイナミックレンダリングコンテンツが持つ柔軟性の両方を提供することができます。

ISR をルート(route)に追加するには、`config` オブジェクトに `isr` プロパティを含めます:

```
/// file: blog/[slug]/+page.server.js
// @filename: ambient.d.ts
declare module '$env/static/private' {
 export const BYPASS_TOKEN: string;
}

// @filename: index.js
// cut---
import { BYPASS_TOKEN } from '$env/static/private';

export const config = {
 isr: {
 // キャッシュされたアセットが Serverless Function を呼び出して再生成されるまでの有効期限 (秒単位)。
 // 値に `false` を設定すると、無期限になります。
 expiration: 60,
 }
}
```

```
// URL で提供されるランダムな token で、アセットへのリクエストに
// __prerender_bypass=<token> cookie を用いることで、アセットのキャッシングされたバージョンを回避することができます。
//
// `GET` や `HEAD` リクエストに `x-prerender-revalidate: <token>` を付けると、アセットの再バリデート(re-validated)を強制することができます。
bypassToken: BYPASS_TOKEN,
// 有効なクエリパラメータのリストです。他のパラメータ（例えば utm tracking codes）は無視され、
// コンテンツが不必要に再生成されないようにします
allowQuery: ['search']
};

};

};

expiration
```

`expiration` プロパティは必須で、その他は任意です。

## 環境変数

Vercel では [デプロイメント固有の環境変数](#)一式を使用できます。他の環境変数と同様、`$env/static/private` と `$env/dynamic/private` からアクセスでき（詳細は後述）、`public` のほうからはアクセスできません。クライアントからこれらの変数にアクセスするには：

```
// @errors: 2305
/// file: +layout.server.js
import { VERCEL_COMMIT_REF } from '$env/static/private';

/** @type {import('./$types').LayoutServerLoad} */
export function load() {
 return {
 deploymentGitBranch: VERCEL_COMMIT_REF
 };
}

/// file: +layout.svelte
<script>
 /** @type {import('./$types').LayoutServerData} */
 export let data;
</script>

<p>This staging environment was deployed from {data.deploymentGitBranch}.</p>
```

Vercel でビルドする場合、これらの変数は全てビルド時と実行時で変わらないため、`$env/dynamic/private` ではなく、変数を静的に置換しデッドコードの削除などの最適化ができる `$env/static/private` の使用をおすすめします。

## Notes

### Vercel functions

プロジェクトの root の `api` ディレクトリに Vercel functions がある場合、`/api/*` に対するリクエストは SvelteKit で処理されません。Vercel functions に JavaScript 以外の言語を使用する必要が無いのであれば、SvelteKit アプリの [API ルート\(routes\)](#) として実装すると良いでしょう。逆に Vercel functions に JavaScript 以外の言語を使用する必要がある場合は、SvelteKit アプリに `/api/*` ルート(routes)を含めないようにしてください。

## Node version

ある時期より前に作成されたプロジェクトはデフォルトで Node 14 を使用していますが、SvelteKit には Node 16 以降が必要です。 [プロジェクトの設定で Node のバージョンを変更することができます。](#)

## トラブルシューティング

### ファイルシステムにアクセスする

Serverless/Edge 環境では、`fs.readFileSync` などのメソッドでファイルシステムにアクセスすることはできません。もしこのような方法でファイルにアクセスする必要がある場合、アプリのビルド中に[プリレンダリング](#)でこれを行ってください。例えば、ブログを持っていて、CMS でコンテンツを管理したくない場合、コンテンツをプリレンダリングし（またはコンテンツを取得するエンドポイントをプリレンダリングし）、新しいコンテンツを追加するたびにブログを再デプロイする必要があります。

# adapter を書く

あなたが使いたい環境向けの adapter がまだ存在しない場合は、ご自身で adapter を作成することができます。あなたが使いたい環境に似ているプラットフォームの [adapter のソースを見て](#)、コピーするところから始めることをおすすめします。

Adapter パッケージは以下の API を実装しなければなりません。これによって `Adapter` が作られます:

```
// @filename: ambient.d.ts
type AdapterSpecificOptions = any;

// @filename: index.js
// cut---
/** @param {AdapterSpecificOptions} options */
export default function (options) {
 /** @type {import('@sveltejs/kit').Adapter} */
 const adapter = {
 name: 'adapter-package-name',
 async adapt(builder) {
 // adapter implementation
 }
 };

 return adapter;
}
```

`Adapter` の型とそのパラメータは `types/index.d.ts` にて利用可能です。

`adapt` メソッドの中で、adapter が行うべきことがいくつかあります:

- `build` ディレクトリの掃除
- SvelteKit の出力を `builder.writeClient`、`builder.writeServer`、`builder.writePrerendered` で書き出す
- これらのコードを出力する:
  - `${builder.getServerDirectory()}/index.js` から `Server` をインポートする
  - `builder.generateManifest({ relativePath })` で生成された `manifest` でアプリをインスタンス化する
  - 必要に応じて、プラットフォームからのリクエストをリスン(Listen)しそのリクエストを標準の `Request` に変換し、`server.respond(request, { getClientAddress })` 関数を呼び出して `Response` を生成して応答する
  - `server.respond` に渡される `platform` オプションを使用してプラットフォーム固有の情報を SvelteKit に公開する
  - 必要に応じて、ターゲットのプラットフォームで動作するよう `fetch` をグローバルにシム(shim)する。SvelteKit は、プラットフォームが `node-fetch` を使用できるようにするために `@sveltejs/kit/install-fetch` ヘルパーを提供しています
- 必要に応じて、ターゲットのプラットフォームで依存関係(dependencies)をインストールするのを避けるため、出力をバンドルする
- ユーザーの静的ファイルと生成された JS/CSS をターゲットのプラットフォームにとって適切な場所に配置する

可能であれば、adapter の出力は `build/` ディレクトリに置き、中間出力は `.svelte-kit/[adapter-name]` に置くことをおススメします。



# 高度なルーティング

## Rest パラメータ

ルートセグメント(route segments)の数がわからない場合は、rest 構文を使用することができます。例えば GitHub のファイルビューアのようなものを実装する場合は…

```
/[org]/[repo]/tree/[branch]/[...file]
```

…この場合、`/sveltejs/kit/tree/master/documentation/docs/04-advanced-routing.md` をリクエストすると、以下のパラメータをページで使うことができます:

```
// @noErrors
{
 org: 'sveltejs',
 repo: 'kit',
 branch: 'master',
 file: 'documentation/docs/04-advanced-routing.md'
}
```

`src/routes/a/[...rest]/z/+page.svelte` は `/a/z` にも (つまり、パラメータが全くない場合にも)、`/a/b/z` や `/a/b/c/z` と同様にマッチします。Rest パラメータの値が正しいことを、例えば `matcher` を使用するなどして確認してください。

## 404 pages

Rest パラメータによってカスタムの 404 をレンダリングすることができます。これらのルート(routes)があるとして…

```
src/routes/
├ marx-brothers/
| ├ chico/
| ├ harpo/
| ├ groucho/
| └ +error.svelte
└ +error.svelte
```

…もし `/marx-brothers/karl` にリクエストしても、`marx-brothers/+error.svelte` ファイルはレンダリングされません。なぜならどのルート(route) にもマッチしないからです。もしネストしたエラーページをレンダリングしたければ、どんな `/marx-brothers/*` リクエストにもマッチするルート(route)を作成し、そこから 404 を返すようにしてください:

```
src/routes/
├ marx-brothers/
+| ├ [...path]/
| ├ chico/
| ├ harpo/
| ├ groucho/
| └ +error.svelte
└ +error.svelte
```

```
/// file: src/routes/marx-brothers/[...path]/+page.js
import { error } from '@sveltejs/kit';

/** @type {import('./$types').PageLoad} */
export function load(event) {
 throw error(404, 'Not Found');
}
```

もし 404 のケースをハンドリングしていない場合、`handleError` によって表示が行われます。

## Optional parameters

`[lang]/home` というルートに含まれる `lang` というパラメータは必須です。これらのパラメータをオプションにできると、今回の例では `home` と `en/home` のどちらも同じページを指すことができるのでとても便利です。パラメータにもう1つ括弧を付けることでこれができます: `[[lang]]/home`

optional のルートパラメータ(route parameter)は rest パラメータに続けて使用すること (`[...rest]/[[optional]]`) はできません。パラメータは 'greedily' にマッチし、optional のパラメータは使用されないこともあります。

## マッチング(Matching)

`src/routes/archive/[page]` のようなルート(route)は `/archive/3` にマッチしますが、`/archive/potato` にもマッチしてしまいます。これを防ぎたい場合、パラメータ文字列(`"3"` や `"potato"`)を引数に取ってそれが有効なら `true` を返す `matcher` を `params` ディレクトリに追加することで、ルート(route)のパラメータを適切に定義することができます...

```
/// file: src/params/integer.js
/** @type {import('@sveltejs/kit').ParamMatcher} */
export function match(param) {
 return /^\d+$/.test(param);
}
```

…そしてルート(routes)を拡張します:

```
-src/routes/archive/[page]
+src/routes/archive/[page=integer]
```

もしパス名がマッチしない場合、SvelteKit は(後述のソート順の指定に従って)他のルートでマッチするか試行し、どれにもマッチしない場合は最終的に 404 を返します。

`params` ディレクトリにある各モジュールは `matcher` に対応しています。ただし、`matcher` のユニットテストに使用される `*.test.js` と `*.spec.js` ファイルは例外です。

Matcher は サーバーとブラウザの両方で動作します。

## ソート(Sorting)

あるパスに対し、マッチするルート(routes)は複数でも構いません。例えば、これらのルート(routes)はどれも `/foo-abc` にマッチします：

```
src/routes/...catchall/+page.svelte
src/routes/[[a=x]]/+page.svelte
src/routes/[b]/+page.svelte
src/routes/foo-[c]/+page.svelte
src/routes/foo-abc/+page.svelte
```

SvelteKit は、どのルート(route)に対してリクエストされているのかを判断しなければなりません。そのため、以下のルールに従ってこれらをソートします…

- より詳細・明確(specific)なルート(routes)ほど、より優先度が高い (例えば、動的なパラメータが1つあるルートより、パラメータのないルートのほうがより詳細・明確(specific)である、など)
- `matchers` 付きのパラメータ (`[name=type]`) は `matchers` なしのパラメータ (`[name]`) よりも優先度が高い
- `[[optional]]` と `...rest` パラメータはルート(route)の最後の部分でない限り無視される (最後の部分になっている場合は最も低い優先度として扱われる)。言い換えると、ソートの目的上、`x/[[y]]/z` と `x/z` は同等に扱われる
- 優先度が同じ場合はアルファベット順で解決される

…この順序で並べると、`/foo-abc` の場合は `src/routes/foo-abc/+page.svelte` を呼び出し、`/foo-def` の場合は `src/routes/foo-[c]/+page.svelte` を呼び出します：

```
src/routes/foo-abc/+page.svelte
src/routes/foo-[c]/+page.svelte
src/routes/[[a=x]]/+page.svelte
src/routes/[b]/+page.svelte
src/routes/...catchall/+page.svelte
```

## エンコード(Encoding)

ファイルシステムでは使用できない文字があります — Linux と Mac では `/`、Windows では `\ / : * ? " < > |` です。URLにおいては、`#` と `%` には特別な意味がありますし、SvelteKitにおいては `[]()` に特別な意味があります。そのため、これらの文字をそのままルート(route)に使用することはできません。

これらの文字をルート(route)に使用するには、16進数のエスケープシーケンスを使います。`[x+nn]` というフォーマットで、`nn` の部分は16進数の文字コードです：

- `\` — `[x+5c]`
- `/` — `[x+2f]`
- `:` — `[x+3a]`
- `*` — `[x+2a]`
- `?` — `[x+3f]`
- `"` — `[x+22]`
- `<` — `[x+3c]`
- `>` — `[x+3e]`
- `|` — `[x+7c]`

- `#` — `[x+23]`
- `%` — `[x+25]`
- `[` — `[x+5b]`
- `]` — `[x+5d]`
- `(` — `[x+28]`
- `)` — `[x+29]`

例えば、`/smileys/:- というルート(route)を作る場合は、src/routes/smileys/[x+3a]-[x+29]/+page.svelte ファイルを作成します。`

JavaScript を使って文字の 16進数コードを判定することができます:

```
':'.charCodeAt(0).toString(16); // '3a', hence '[x+3a]'
```

また、Unicode のエスケープシーケンスを使用することもできます。通常、エンコードされていない文字を直接使用することができるので、こうする必要はありませんが、何らかの理由で、例えばファイル名に絵文字を使用することができない場合、エスケープ文字を使用することができます。言い換えると、以下は同じことをしているということです:

```
src/routes/[u+d83e][u+dd2a]/+page.svelte
src/routes/😊/+page.svelte
```

Unicode エスケープシーケンスのフォーマットは `[u+nnnn]` で、`nnnn` の部分は `0000` から `10ffff` までの適切な値です (JavaScript の文字列エスケープとは異なり、`ffff` 以上のコードポイントを表現するためにサロゲートペアを使用する必要はありません)。Unicode エンコーディングについてもっと知りたい方は、[Programming with Unicode](#) を参照してください。

ディレクトリの先頭に `.` 文字があると、TypeScript で [問題](#) が起きるため、例えば `.well-known` のようなルート(route)を作る場合はこれらの文字をエンコードしておくと良いでしょう: `src/routes/[x+2e]well-known/...`

## Advanced layouts

デフォルトでは、レイアウトの階層が ルート(route)の階層 に反映されます。場合によっては、そうしたくないこともあるかもしれません。

### (group)

'アプリ' のルート(routes)としてのレイアウト (例えば `/dashboard` や `/item`) が1つあり、'マーケティング' のルート(routes)としての別のレイアウト (`/blog` や `/testimonials`) があるかもしれません。これらのルート(routes)を、ディレクトリの名前を括弧でくくることでグループ化することができます。通常のディレクトリとは異なり、`(app)` や `(marketing)` はそれらの中のルート(routes)の URL パス名には影響しません:

```
src/routes/
+| (app)/
| + dashboard/
| + item/
```

```

| \ +layout.svelte
+| (marketing)/
| +about/
| +testimonials/
| \ +layout.svelte
+| admin/
\ +layout.svelte

```

`+page` を `(group)` の中に直接配置することもできます (例えば、`/` が `(app)` や `(marketing)` のページであるべき場合など)。

## Breaking out of layouts

最上位のレイアウト(root layout)は、アプリの全てのページに適用されます。省略した場合、デフォルトは `<slot />` です。もし、いくつかのページで他のページとは異なるレイアウト階層を持ちたい場合には、アプリ全体を1つまたは複数のグループにして、共通のレイアウトを継承しないルート(route)を分けることができます。

上記の例で、`/admin` ルート(route)は `(app)` や `(marketing)` のレイアウトを継承しません。

## +page@

ページは、ルート(route)ごとに現在のレイアウト階層から抜け出すことができます。先ほどの例に出てきた `(app)` グループの中に、`/item/[id]/embed` ルート(route)があるとします:

```

src/routes/
+| (app)/
| +item/
| | +[id]/
| | +embed/
+| | + +page.svelte
| | +layout.svelte
| +layout.svelte
\ +layout.svelte

```

通常、これは最上位のレイアウト(root layout)と `(app)` レイアウトと `item` レイアウトと `[id]` レイアウトを継承します。@ と、その後ろにセグメント名 (最上位のレイアウト(root layout)の場合は空文字列(empty string)) を追加することで、これらのレイアウトのどれかにリセットすることができます。この例では、以下のオプションから選択できます:

- `+page@[id].svelte` - `src/routes/(app)/item/[id]/+layout.svelte` を継承します
- `+page@item.svelte` - `src/routes/(app)/item/+layout.svelte` を継承します
- `+page@(app).svelte` - `src/routes/(app)/+layout.svelte` を継承します
- `+page@svelte` - `src/routes/+layout.svelte` を継承します

```

src/routes/
+| (app)/
| +item/
| | +[id]/
| | +embed/
+| | + +page@(app).svelte
| | +layout.svelte

```

```
| | ↴ +layout.svelte
| ↴ +layout.svelte
↳ +layout.svelte
```

## +layout@

ページと同じように、同じ方法でレイアウト `自体` をその親のレイアウトの階層から外すことができます。例えば、`+layout@.svelte` コンポーネントはその全ての子ルート(routes)の階層をリセットします。

```
src/routes/
├── (app)/
│ ├── item/
│ │ ├── [id]/
│ │ │ ├── +page.svelte // (app)/item/[id]/+layout.svelte を使用します
│ │ │ ├── +layout.svelte // (app)/item/+layout@.svelte を継承します
│ │ │ └── +page.svelte // (app)/item/+layout@.svelte を使用します
│ │ └── +layout@.svelte // 最上位のレイアウト(root layout)を継承し、
│ │ (app)/+layout.svelte をスキップします
│ └── +layout.svelte
└── +layout.svelte
```

## レイアウトグループを使うときは

全てのユースケースがレイアウトのグループ化に適しているわけではありませんし、無理に使用する必要もありません。あなたのユースケースが複雑な `(group)` のネストになってしまふかもしれませんし、たった1つの例外ケースのために `(group)` を導入したくないかもしれません。コンポジション(再利用可能な `load` 関数や Svelte コンポーネント)や `if` 文など、他の手段を使用してやりたいことを実現するのは全く問題ありません。以下の例では、最上位のレイアウト(root layout)に戻し、他のレイアウトでも使用できるコンポーネントや関数を再利用したレイアウトを示しています:

```
/// file: src/routes/nested/route/+layout@.svelte
<script>
 import ReusableLayout from '$lib/ReusableLayout.svelte';
 export let data;
</script>

<ReusableLayout {data}>
 <slot />
</ReusableLayout>
```

```
/// file: src/routes/nested/route/+layout.js
// @filename: ambient.d.ts
declare module "$lib/reusable-load-function" {
 export function reusableLoad(event: import('@sveltejs/kit').LoadEvent): Promise<Record<string, any>>;
}
// @filename: index.js
// cut---
import { reusableLoad } from '$lib/reusable-load-function';

/** @type {import('./$types').PageLoad} */
export function load(event) {
 // Add additional logic here, if needed
 return reusableLoad(event);
}
```

## その他の参考情報

- [Tutorial: Advanced Routing](#)

---

[Go to TOC](#)

# Hooks

'Hooks' は、特定のイベントに対して SvelteKit がレスポンスを呼び出すことを宣言するアプリ全体の関数で、これによってフレームワークの動作をきめ細やかに制御できるようになります。

hooks ファイルは2つあり、どちらもオプションです:

- `src/hooks.server.js` — アプリのサーバーの hooks
- `src/hooks.client.js` — アプリのクライアントの hooks

これらのモジュールのコードはアプリケーションの起動時に実行されるので、データベースクライアントの初期化などに有用です。

これらのファイルの場所は `config.kit.files.hooks` で設定できます。

## Server hooks

以下の hooks は `src/hooks.server.js` に追加することができます:

### handle

この関数は SvelteKit のサーバーが [リクエスト](#) を受けるたびに (アプリの実行中であろうと、[プリレンダリング](#)であろうと) 実行され、[レスポンス](#) を決定します。リクエストを表す `event` オブジェクトと、ルート(route)をレンダリングしレスポンスを生成する `resolve` という関数を受け取ります。これにより、レスポンスのヘッダーやボディを変更したり、SvelteKitを完全にバイパスすることができます (例えば、プログラムでルート(routes)を実装する場合など)。

```
/// file: src/hooks.server.js
/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
 if (event.url.pathname.startsWith('/custom')) {
 return new Response('custom response');
 }

 const response = await resolve(event);
 return response;
}
```

静的アセット(プリレンダリング済みのページを含む)に対するリクエストは SvelteKit では処理されません。

未実装の場合、デフォルトは `({ event, resolve }) => resolve(event)` となります。カスタムデータをリクエストに追加し、`+server.js` のハンドラーやサーバー(server) `load` 関数に渡すには、以下のように `event.locals` オブジェクトに埋め込んでください。

```

/// file: src/hooks.server.js
// @filename: ambient.d.ts
type User = {
 name: string;
}

declare namespace App {
 interface Locals {
 user: User;
 }
}

const getUserInformation: (cookie: string | void) => Promise<User>;

```

// @filename: index.js  
// cut--  
/\*\* @type {import('@sveltejs/kit').Handle} \*/  
export async function handle({ event, resolve }) {
 event.locals.user = await getUserInformation(event.cookies.get('sessionid'));

 const response = await resolve(event);
 response.headers.set('x-custom-header', 'potato');

 return response;
}

`sequence` ヘルパー関数を使用すると、複数の `handle` 関数を定義することができます。

`resolve` はオプションの第2引数をサポートしており、レスポンスのレンダリング方法をより詳細にコントロールすることができます。そのパラメータは、以下のフィールドを持つオブジェクトです:

- `transformPageChunk(opts: { html: string, done: boolean }): MaybePromise<string | undefined>` — カスタムの変換を HTML に適用します。`done` が `true` である場合、それは最後のチャンクです。チャンクが整形された HTML であることは保証されませんが（例えば、要素の開始タグは含むが終了タグは含まれない、など）、常に `%sveltekit.head%` やレイアウト(layout)/ページ(page)コンポーネントなどのような理にかなった境界 (sensible boundaries) で分割されます。
- `filterSerializedResponseHeaders(name: string, value: string): boolean` — `load` 関数が `fetch` でリソースを読み込むときに、シリализされるレスポンスにどのヘッダーを含めるかを決定します。デフォルトでは何も含まれません。
- `preload(input: { type: 'js' | 'css' | 'font' | 'asset', path: string }): boolean` — `<head>` タグにどのファイルをプリロードの対象として追加するか決定します。このメソッドはビルト時、コードチャンクを構築している際に見つかったファイルごとに呼び出されます。これにより、例えば `+page.svelte` に `import './styles.css'` がある場合、そのページに訪れたときにその CSS ファイルへの解決されたパスを以て `preload` が呼び出されるようになります。これはビルト時の分析によって行われるため、開発モードでは `preload` が呼ばれないことにご注意ください。プリロードによってその対象がより早くダウンロードされるようになるためパフォーマンスが改善しますが、不必要に多くのものをダウンロードしてしまうと、core web vitals を悪化させてしまいます。デフォルトでは、`js`、`css` ファイルがプリロードされます。現時点では `asset` ファイルはプリロードされませんが、フィードバックによっては追加されるかもしれません。

```

/// file: src/hooks.server.js
/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
 const response = await resolve(event, {

```

```

 transformPageChunk: ({ html }) => html.replace('old', 'new'),
 filterSerializedResponseHeaders: (name) => name.startsWith('x-'),
 preload: ({ type, path }) => type === 'js' || path.includes('/important/')
 });

 return response;
}

```

`resolve(...)` は決してエラーをスローせず、適切なステータスコードと `Promise<Response>` を返すことにご注意ください。もし `handle` 中に他の場所でエラーがスローされた場合、それは致命的(fatal)なものとして扱われ、SvelteKit は `Accept` ヘッダーに応じて、そのエラーの JSON 表現か、`src/error.html` でカスタマイズ可能なフォールバックエラーページをレスポンスとして返します。エラーハンドリングの詳細は [こちら](#) からお読み頂けます。

## handleFetch

この関数は、サーバー上で（またはプリレンダリング中に）実行される `load` 関数や `action` 関数の中で発生する `fetch` リクエストを変更（または置換）することができます。

例えば、ユーザーがクライアントサイドでそれぞれのページに移動する際に、`load` 関数で `https://api.yourapp.com` のようなパブリックな URL にリクエストを行うかもしれません、SSR の場合には（パブリックなインターネットとの間にプロキシやロードバランサーをバイパスして）API を直接呼ぶほうが理にかなっているでしょう。

```

/// file: src/hooks.server.js
/** @type {import('@sveltejs/kit').HandleFetch} */
export async function handleFetch({ request, fetch }) {
 if (request.url.startsWith('https://api.yourapp.com/')) {
 // clone the original request, but change the URL
 request = new Request(
 request.url.replace('https://api.yourapp.com/',
 'http://localhost:9999/'),
 request
);
 }

 return fetch(request);
}

```

### Credentials

同一オリジン(same-origin)リクエストの場合、SvelteKit の `fetch` 実装は、`credentials` オプションを `"omit"` にしない限り、`cookie` と `authorization` ヘッダーを転送します。

クロスオリジン(cross-origin)リクエストの場合、リクエスト URL がアプリのサブドメインに属するときは `cookie` はリクエストに含まれます。例えば、あなたのアプリが `my-domain.com` にあり、あなたの API が `api.my-domain.com` にある場合、`cookie` はリクエストに含まれることになります。

もしあなたのアプリと API が兄弟関係にあるサブドメイン（例えば `www.my-domain.com` と `api.my-domain.com`）の場合は、`my-domain.com` のような共通の親ドメインに属する `cookie` は含まれません、なぜなら SvelteKit にはその `cookie` がどのドメインに属するか判断する方法がないからです。こういったケースでは、`handleFetch` を使って手動で `cookie` を含める必要があります：

```
/// file: src/hooks.server.js
// @errors: 2345
/** @type {import('@sveltejs/kit').HandleFetch} */
export async function handleFetch({ event, request, fetch }) {
 if (request.url.startsWith('https://api.my-domain.com/')) {
 request.headers.set('cookie', event.request.headers.get('cookie'));
 }

 return fetch(request);
}
```

## Shared hooks

以下は `src/hooks.server.js` と `src/hooks.client.js` のどちらにも追加できます:

### handleError

予期せぬエラーがロード中またはレンダリング中にスローされると、この関数が `error` と `event` を引数にとって呼び出されます。これによって2つのことが可能になります:

- エラーをログに残すことができます
- エラーからメッセージやスタックトレースなどの機密情報を省略し、ユーザーに見せてても安全なカスタムの表現を生成することができます。戻り値は `$page.error` の値となります。デフォルトでは、404 (`event.route.id` が `null` になっていることで検知できます) の場合は `{ message: 'Not Found' }`、それ以外の場合は `{ message: 'Internal Error' }` となります。これを型安全にするために、`App.Error` インターフェイスを宣言して、期待される形をカスタマイズすることができます (わかりやすいフォールバックの動作を保証するため、`message: string` を含めなければなりません)。

以下のコードは、エラーの形を `{ message: string; errorId: string }` として型付けし、それを `handleError` 関数から適宜返す例を示しています:

```
/// file: src/app.d.ts
declare global {
 namespace App {
 interface Error {
 message: string;
 errorId: string;
 }
 }
}

export {};
```

```
/// file: src/hooks.server.js
// @errors: 2322
// @filename: ambient.d.ts
declare module '@sentry/node' {
 export const init: (opts: any) => void;
 export const captureException: (error: any, opts: any) => void;
}

// @filename: index.js
// cut---
import * as Sentry from '@sentry/node';
```

```

import crypto from 'crypto';

Sentry.init({/*...*/})

/** @type {import('@sveltejs/kit').HandleServerError} */
export async function handleError({ error, event }) {
 const errorId = crypto.randomUUID();
 // example integration with https://sentry.io/
 Sentry.captureException(error, { extra: { event, errorId } });

 return {
 message: 'Whoops!',
 errorId
 };
}

```

```

/// file: src/hooks.client.js
// @errors: 2322
// @filename: ambient.d.ts
declare module '@sentry/svelte' {
 export const init: (opts: any) => void;
 export const captureException: (error: any, opts: any) => void;
}

// @filename: index.js
// cut---
import * as Sentry from '@sentry/svelte';

Sentry.init({/*...*/})

/** @type {import('@sveltejs/kit').HandleClientError} */
export async function handleError({ error, event }) {
 const errorId = crypto.randomUUID();
 // example integration with https://sentry.io/
 Sentry.captureException(error, { extra: { event, errorId } });

 return {
 message: 'Whoops!',
 errorId
 };
}

```

src/hooks.client.js では、handleError の型は HandleServerError ではなく HandleClientError で、event は RequestEvent ではなく NavigationEvent です。

この関数は 想定される エラー (@sveltejs/kit からインポートされる `error` 関数でスローされるエラー) の場合は呼び出されません。

開発中、Svelte のコードの構文エラーでエラーが発生した場合、渡される `error` には、エラーの場所のハイライトが付与された `frame` プロパティがあります。

handleError 自体が決してエラーをスローしないようにしてください。

## その他の参考資料

- [Tutorial: Hooks](#)

# Errors

ソフトウェア開発において、エラーは避けられないものです。SvelteKit では、エラーが発生した場所、エラーの種類、受信したリクエストの性質に応じて、異なる方法でエラーを処理します。

## Error objects

SvelteKit は想定されるエラーと予期せぬエラーを区別します。どちらもデフォルトではシンプルな `{ message: string }` オブジェクトとして表現されます。

以下の例のように、`code` やトラッキング `id` を追加することができます。(TypeScript を使用する場合、[Type safety](#) で説明したように `Error` 型を再定義する必要があります)

## Expected errors

想定されるエラーとは、`@sveltejs/kit` からインポートされる `error` を使用して作成されるものを指します：

```
/// file: src/routes/blog/[slug]/+page.server.js
// @filename: ambient.d.ts
declare module '$lib/server/database' {
 export function getPost(slug: string): Promise<{ # string, content: string } | undefined>
}

// @filename: index.js
// cut---
import { error } from '@sveltejs/kit';
import * as db from '$lib/server/database';

/** @type {import('./$types').PageServerLoad} */
export async function load({ params }) {
 const post = await db.getPost(params.slug);

 if (!post) {
 throw error(404, {
 message: 'Not found'
 });
 }

 return { post };
}
```

こうすると、SvelteKit はレスポンスのステータスコードを 404 に設定し、`+error.svelte` コンポーネントをレンダリングします。`$page.error` は `error(...)` に第二引数として渡されたオブジェクトです。

```
/// file: src/routes/+error.svelte
<script>
 import { page } from '$app/stores';
</script>

<h1>{$page.error.message}</h1>
```

必要に応じて、エラーオブジェクトにプロパティを追加することができます…

```
throw error(404, {
 message: 'Not found',
+ code: 'NOT_FOUND'
});
```

…追加しない場合は、便宜上、文字列を第二引数に渡すことができます：

```
-throw error(404, { message: 'Not found' });
+throw error(404, 'Not found');
```

## Unexpected errors

予期せぬエラーとは、リクエストの処理中に発生するその他の例外のことを指します。これらは機密情報を含むことがあるため、予期せぬエラーのメッセージとスタックトレースはユーザーには公開されません。

デフォルトでは、予期せぬエラーはコンソール（または、本番環境では、サーバーログ）に出力され、ユーザーに公開されるエラーはこのように汎用的な形式です。

```
{ "message": "Internal Error" }
```

予期せぬエラーは `handleError` hook を通ります。ここで、独自のエラーハンドリングを追加することができます。例えば、レポーティングサービスにエラーを送ったり、カスタムのエラーオブジェクトを返したりすることができます。

```
/// file: src/hooks.server.js
// @errors: 2322 1360 2571 2339
// @filename: ambient.d.ts
declare module '@sentry/node' {
 export const init: (opts: any) => void;
 export const captureException: (error: any, opts: any) => void;
}

// @filename: index.js
// cut---
import * as Sentry from '@sentry/node';

Sentry.init({/*...*/})

/** @type {import('@sveltejs/kit').HandleServerError} */
export function handleError({ error, event }) {
 // example integration with https://sentry.io/
 Sentry.captureException(error, { extra: { event } });

 return {
 message: 'Whoops!',
 code: error?.code ?? 'UNKNOWN'
 };
}
```

`handleError` では絶対に `error` をスローしないでください

## Responses

もし `handle` の中や `+server.js` リクエストハンドラの中でエラーが発生した場合、SvelteKit はリクエストの `Accept` ヘッダーに応じて、フォールバックエラーページか、エラーオブジェクトの JSON 表現をレスポンスとして返します。

`src/error.html` ファイルを追加することで、フォールバックエラーページをカスタマイズすることができます:

```
<!DOCTYPE html>
<html lang="en">
 <head>
 <meta charset="utf-8" />
 <title>%sveltekit.error.message%</title>
 </head>
 <body>
 <h1>My custom error page</h1>
 <p>Status: %sveltekit.status%</p>
 <p>Message: %sveltekit.error.message%</p>
 </body>
</html>
```

SvelteKit が `%sveltekit.status%` と `%sveltekit.error.message%` を、それぞれ対応する値に置き換えます。

ページのレンダリング中に `load` 関数の中でエラーが発生した場合、SvelteKit はエラーが発生した場所に最も近い `+error.svelte` コンポーネントをレンダリングします。`+layout(.server).js` の `load` 関数の内側でエラーが発生した場合、ツリーの中で最も近くにあるエラー境界はそのレイアウトの上位にある `+error.svelte` ファイルです（隣ではありません）。

例外は、最上位の `+layout.js` や `+layout.server.js` の中でエラーが発生した場合です。通常、最上位のレイアウトには `+error.svelte` コンポーネントが含まれているためです。この場合、SvelteKit はフォールバックエラーページを使用します。

## Type safety

もし TypeScript を使用していてエラーの形式をカスタマイズする必要がある場合、アプリで `App.Error` インターフェイスを宣言することでそれができます（慣習ではこれを `src/app.d.ts` に書きますが、TypeScript が '参照' することができればどこでも構いません）：

```
/// file: src/app.d.ts
declare global {
 namespace App {
 interface Error {
 + code: string;
 + id: string;
 }
 }
}

export {};
```

このインターフェイスは常に `message: string` プロパティを含んでいます。

## その他の参考資料

- [Tutorial: Errors and redirects](#)
- [Tutorial: Hooks](#)

---

[Go to TOC](#)

# Link options

SvelteKit では、アプリのルート(routes)間の移動に、(フレームワーク固有の `<Link>` コンポーネントではなく) `<a>` 要素を使用します。ユーザーが、`href` がアプリのものであるリンク (外部サイトではないリンク) をクリックする場合、SvelteKit はそのコードをインポートし、データを取得するために必要な `load` 関数を呼び出して、新しいページに移動します。

`data-sveltekit-*` 属性でリンクの挙動をカスタマイズすることができます。これらは `<a>` 自身やその親要素に適用することができます。

これらのオプションは、`method="GET"` を持つ `<form>` 要素にも適用されます。

## data-sveltekit-preload-data

ユーザーがリンクをクリックしたことをブラウザが検知するより前に、(デスクトップでは) マウスがリンクをホバーしたことや、`touchstart` や `mousedown` がトリガーされることを検知することができます。どちらの場合も、`click` イベントが発生することを経験に基づいて推測することができます。

SvelteKit はこの情報を使ってインポートするコードやそのページのデータの取得をいち早く開始することができ、数百ミリ秒を稼ぐことができます。これが、ユーザーインターフェースが遅延していると感じるか、それともきびきび動いていると感じるかの差になります。

この動作は `data-sveltekit-preload-data` 属性によってコントロールすることができ、2つの値のうちどちらかを設定することができます:

- "hover" は、マウスがリンクの上にきたときにプリロードを開始します。モバイルでは、`touchstart` でプリロードが開始されます
- "tap" は、`touchstart` や `mousedown` イベントが検知されるとすぐにプリロードが開始されます

デフォルトのプロジェクトテンプレートには、`src/app.html` の `<body>` 要素に `data-sveltekit-preload-data="hover"` が適用されており、デフォルトで全てのリンクがホバー時にプリロードされます:

```
<body data-sveltekit-preload-data="hover">
 <div style="display: contents">%sveltekit.body%</div>
</body>
```

時には、ユーザーがリンクをホバーしたときに `load` を呼ぶのは望ましくないことがあるでしょう。誤検出の可能性もありますし (必ずしもホバーに続いてクリックが発生するわけではない)、データの更新が非常に早い場合はデータが古くなってしまうこともあります。

これらの場合には、値に "tap" を指定します。こうすると SvelteKit は、ユーザーがリンクをタップまたはクリックしたときのみ、`load` を呼び出すようになります:

```
<a data-sveltekit-preload-data="tap" href="/stonks">
 Get current stonk values

```

プログラムで `$app/navigation` の `preloadData` を実行することもできます。

ユーザーがデータ使用量の削減を選択している場合、つまり `navigator.connection.saveData` が `true` になっている場合は、データは決してプリロードされません。

## data-sveltekit-preload-code

リンク先の データをプリロードしたくない場所であっても、コードをプリロードするのは有益なこともあります。`data-sveltekit-preload-code` 属性は `data-sveltekit-preload-data` と同様に動作しますが、4つの値から選択できる点が異なります。'先行度'('eagerness') の降順で並べると：

- `"eager"` は、すぐにリンクをプリロードします
- `"viewport"` は、リンクがビューポートに入るとすぐにプリロードします
- `"hover"` - コードだけがプリロードされることを除き、上記(`data-sveltekit-preload-data` の `"hover"`)と同じです
- `"tap"` - コードだけがプリロードされることを除き、上記(`data-sveltekit-preload-data` の `"tap"`)と同じです

`viewport` と `eager` は、ナビゲーション直後の DOM に存在するリンクにのみ適用されることにご注意ください。リンクが後から追加された場合(例えば `{#if ...}` ブロックの中)、`hover` や `tap` によってトリガーされるまでプリロードされません。DOM の変更を積極的に観察することによって生じてしまうパフォーマンス劣化を避けるためです。

コードのプリロードはデータのプリロードの前提条件であるため、この属性は、存在するどの `data-sveltekit-preload-data` 属性よりも先行度が高い値(more eager value)を指定した場合にのみ、効果を発揮します。

`data-sveltekit-preload-data` と同様、ユーザーがデータ使用量の削減を選択している場合、この属性も無視されます。

## data-sveltekit-reload

時には、SvelteKit にリンクを処理させないで、ブラウザに処理をさせる必要があります。`data-sveltekit-reload` 属性をリンクに追加すると…

```
<a data-sveltekit-reload href="/path">Path
```

…リンクがクリックされたときにフルページナビゲーションが発生します。

`rel="external"` 属性があるリンクも同様に扱われます。加えて、[プリレンダリング中](#) は無視されます。

## data-sveltekit-replacestate

ナビゲーションするときにブラウザのセッション履歴(session history)に新しいエントリを作成したくない場合があります。リンクに `data-sveltekit-replacestate` 属性を追加すると…

```
<a data-sveltekit-replacestate href="/path">Path
```

...リンクがクリックされたときに、`pushState` で新しいエントリを作成する代わりに現在の `history` エントリを置き換えます。

## **data-sveltekit-keepfocus**

ナビゲーションの後にフォーカスをリセットしたくない場合があります。例えば、ユーザーが入力している途中で送信をするような検索フォームがあり、テキストの `input` にフォーカスを維持したい場合です。`data-sveltekit-keepfocus` 属性を追加すると…

```
<form data-sveltekit-keepfocus>
 <input type="text" name="query">
</form>
```

…ナビゲーション後も現在フォーカスされている要素にフォーカスが維持されるようになります。通常、リンクにこの属性を使用するのは避けてください、フォーカスされる要素が（その前にフォーカスされていた要素ではなく）`<a>` タグになってしまい、スクリーンリーダーなどの支援技術を使用するユーザーはナビゲーションの後にフォーカスが移動することを期待することが多いです。また、この属性はナビゲーションの後にもまだ存在する要素にのみ使用する必要があります。もし該当の要素が消えてしまうと、ユーザーのフォーカスは失われてしまい、支援技術ユーザーにとって混乱した体験となってしまいます。

## **data-sveltekit-noscroll**

内部のリンクに移動するとき、SvelteKit はブラウザのデフォルトのナビゲーションの挙動を模倣します：ユーザーがページの左上に来るよう、スクロールポジションを `0,0` に変更します（リンクに `#hash` が含まれている場合は、ID が一致する要素までスクロールします）。

特定のケースでは、この挙動を無効化したいことがあるでしょう。`data-sveltekit-noscroll` 属性をリンクに追加すると…

```
Path
```

…リンクがクリックされたあとのスクロールを中止します。

## **Disabling options**

これらのオプションが有効になっている要素の中でこれらのオプションを無効にするには、`"false"` 値を使用します：

```
<div data-sveltekit-preload-data>
 <!-- these links will be preloaded -->
 a
 b
 c

 <div data-sveltekit-preload-data="false">
 <!-- these links will NOT be preloaded -->
 d
 e
 f
 </div>
</div>
```

条件によって要素に属性を適用する場合は、このようにします（`"true"` と `"false"` はどちらも使用することができます）：

```
<div data-sveltekit-reload={shouldReload ? '' : 'off'}>
```

HTML では `<element attribute>` と `<element attribute="">` が同等であるため、これがうまく動作します

# Service workers

Service Worker は、アプリ内部でネットワークリクエストを処理するプロキシサーバーとして機能します。これによりアプリをオフラインで動作させることができます。もしオフラインサポートが不要な場合（または構築するアプリの種類によって現実的に実装できない場合）でも、ビルトした JS と CSS を事前にキャッシュしてナビゲーションを高速化するために Service Worker を使用する価値はあります。

SvelteKit では、`src/service-worker.js` ファイル（や `src/service-worker/index.js`）がある場合、バンドルされ、自動的に登録されます。必要に応じて、[service worker のロケーション](#) を変更することができます。

service worker を独自のロジックで登録する必要がある場合や、その他のソリューションを使う場合は、[自動登録を無効化](#) することができます。デフォルトの登録方法は次のようなものです：

```
if ('serviceWorker' in navigator) {
 addEventListener('load', function () {
 navigator.serviceWorker.register('./path/to/service-worker.js');
 });
}
```

## service worker の内部では

service worker の内部では、`$service-worker` モジュールにアクセスでき、これによって全ての静的なアセット、ビルトファイル、プリレンダリングページへのパスが提供されます。また、アプリのバージョン文字列（一意なキャッシュ名を作成するのに使用できます）と、デプロイメントの `base` パスが提供されます。Vite の設定に `define`（グローバル変数の置換に使用）を指定している場合、それはサーバー/クライアントのビルトだけでなく、service worker にも適用されます。

次の例では、ビルトされたアプリと `static` にあるファイルをすぐに(eagerly)キャッシュし、その他全てのリクエストはそれらの発生時にキャッシュします。これにより、各ページは一度アクセスするとオフラインで動作するようになります。

```
// @errors: 2339
/// <reference types="@sveltejs/kit" />
import { build, files, version } from '$service-worker';

// Create a unique cache name for this deployment
const CACHE = `cache-${version}`;

const ASSETS = [
 ...build, // the app itself
 ...files // everything in `static`
];

self.addEventListener('install', (event) => {
 // Create a new cache and add all files to it
 async function addFilesToCache() {
 const cache = await caches.open(CACHE);
 await cache.addAll(ASSETS);
 }

 event.waitUntil(addFilesToCache());
});
```

```

self.addEventListener('activate', (event) => {
 // Remove previous cached data from disk
 async function deleteOldCaches() {
 for (const key of await caches.keys()) {
 if (key !== CACHE) await caches.delete(key);
 }
 }

 event.waitUntil(deleteOldCaches());
});

self.addEventListener('fetch', (event) => {
 // ignore POST requests etc
 if (event.request.method !== 'GET') return;

 async function respond() {
 const url = new URL(event.request.url);
 const cache = await caches.open(CACHE);

 // `build`/`files` can always be served from the cache
 if (ASSETS.includes(url.pathname)) {
 return cache.match(url.pathname);
 }

 // for everything else, try the network first, but
 // fall back to the cache if we're offline
 try {
 const response = await fetch(event.request);

 if (response.status === 200) {
 cache.put(event.request, response.clone());
 }

 return response;
 } catch {
 return cache.match(event.request);
 }
 }

 event.respondWith(respond());
});

```

キャッシュにはご注意ください！場合によっては、オフラインでは利用できないデータよりも古くなったデータのほうが悪いことがあります。ブラウザはキャッシュが一杯になると空にするため、ビデオファイルのような大きなアセットをキャッシュする場合にもご注意ください。

## 開発中は(During development)

service worker はプロダクション向けにはバンドルされますが、開発中はバンドルされません。そのため、[modules in service workers](#) をサポートするブラウザのみ、開発時にもそれを使用することができます。service worker を手動で登録する場合、開発時に `{ type: 'module' }` オプションを渡す必要があります:

```
import { dev } from '$app/environment';

navigator.serviceWorker.register('/service-worker.js', {
 type: dev ? 'module' : 'classic'
});
```

build と prerendered は開発中は空配列です

## 型安全性(Type safety)

service worker に適切な型を設定するには、マニュアルで設定が必要です。 `service-worker.js` の中に、ファイルの先頭に以下を追加してください:

```
/// <reference types="@sveltejs/kit" />
/// <reference no-default-lib="true"/>
/// <reference lib="esnext" />
/// <reference lib="webworker" />

const sw = /** @type {ServiceWorkerGlobalScope} */ (/** @type {unknown} */
 (self));
```

```
/// <reference types="@sveltejs/kit" />
/// <reference no-default-lib="true"/>
/// <reference lib="esnext" />
/// <reference lib="webworker" />

const sw = self as unknown as ServiceWorkerGlobalScope;
```

これにより、`HTMLElement` のような service worker の中では使用できない DOM の型付けへのアクセスが無効になり、正しい global が初期化されます。`self` を `sw` に再代入することで、プロセス内で型をキャストすることができます（いくつか方法がありますが、これが追加のファイルを必要としない最も簡単な方法です）。ファイルの残りの部分では、`self` の代わりに `sw` を使用します。SvelteKit の型を参照することで、`$service-worker import` に適切な型定義があることを保証することができます。

## その他のソリューション

SvelteKit の service worker 実装は意図的に低レベル(low-level)です。より本格的な、よりこだわりが強い(opinionated)ソリューションが必要な場合は、[Vite PWA plugin](#) のようなソリューションをご覧になることをおすすめしております、こちらは [Workbox](#) を使用しています。service worker に関する一般的な情報をもっとお探しであれば、[MDN web docs](#) をおすすめします。

---

[Go to TOC](#)

# Server-only modules

良き友人のように、SvelteKit はあなたの秘密を守ります。バックエンドとフロントエンドが同じリポジトリにある場合、機密データをフロントエンドのコードに誤ってインポートしてしまうことが簡単に起こってしまいます（例えば、API キーを持つ環境変数など）。SvelteKit はこれを完全に防ぐ方法を提供します：サーバー専用のモジュール(server-only modules)です

## Private environment variables

`modules` セクションで説明されている `$env/static/private` モジュールと `$env/dynamic/private` モジュールは、`hooks.server.js` や `+page.server.js` のようなサーバー上でのみ実行されるモジュールにのみインポートすることができます。

## Your modules

モジュールをサーバー専用にするには2通りの方法があります：

- ファイル名に `.server` を付けます。例: `secrets.server.js`
- モジュールを `$lib/server` に置きます。例: `$lib/server/secrets.js`

## How it works

パブリックに公開されるコード (public-facing code) にサーバー専用のコードを（直接的かまたは間接的にかかわらず）インポートすると…

```
// @errors: 7005
/// file: $lib/server/secrets.js
export const atlantisCoordinates = /* redacted */;
```

```
// @errors: 2307 7006 7005
/// file: src/routes/utils.js
export { atlantisCoordinates } from '$lib/server/secrets.js';

export const add = (a, b) => a + b;
```

```
/// file: src/routes/+page.svelte
<script>
 import { add } from './utils.js';
</script>
```

…SvelteKit はエラーとなります：

```
Cannot import $lib/server/secrets.js into public-facing code:
- src/routes/+page.svelte
 - src/routes/utils.js
 - $lib/server/secrets.js
```

パブリックに公開されるコード `src/routes/+page.svelte` は、`add` を使用しているのみで、シークレットの `atlantisCoordinates` を使用していませんが、ブラウザがダウンロードする JavaScript にシークレットなコードが残ってしまう可能性があり、このインポートチェーンは安全ではないと考えられます。

この機能は動的なインポート (dynamic imports) でも動作し、`await import(`./${foo}.js`)` のような補完されたインポートに対しても有効ですが、小さい注意点があります。もしパブリックに公開されるコードとサーバー専用のモジュールの間に 2つ以上の dynamic imports がある場合、コードが最初にロードされるときに不正なインポートが検出されない可能性があります。

Vitest のようなユニットテストフレームワークはサーバー専用のコードと公開されるコードを区別しません。そのため、テストの実行中、つまり `process.env.TEST === 'true'` となっているときは、不正なインポートの検出は無効化されます。

## その他の参考資料

- [Tutorial: Environment variables](#)

# アセットハンドリング

## キャッシュとINLINE化(Caching and inlining)

パフォーマンス改善のため、Vite はインポートされたアセットを自動的に処理します。ハッシュがファイル名に追加されるのでキャッシュできるようになり、`assetsInlineLimit` より小さいアセットはINLINE化されます。

```
<script>
 import logo from '$lib/assets/logo.png';
</script>


```

マークアップから直接アセットを参照したければ、`svelte-preprocess-import-assets` などのプリプロセッサをお使い頂けます。

CSS 関数の `url()` でインクルードされたアセットの場合は、`vitePreprocess` が役立つでしょう。

## 変換(Transforming)

イメージを変換して、`.webp` や `.avif` などの圧縮イメージフォーマットに変換したり、デバイスごとに異なるサイズのレスポンシブイメージを出力したり、プライバシーのために EXIF データを取り除いたイメージを出力したいことがあるかもしれません。静的に含まれるイメージについては、`vite-imagetools` などの Vite プラグインを使用することができます。HTTP ヘッダーやクエリ文字列パラメータに基づいて適切に変換されたイメージを提供できる CDN を検討することもできます。

# Snapshots

例えばサイドバーのスクロールポジションや、`<input>` 要素の中身などの、一時的な DOM の状態(state)は、あるページから別のページに移動するときに破棄されます。

例えば、ユーザーがフォームに入力し、それを送信する前にリンクをクリックして、それからブラウザの戻るボタンを押した場合、フォームに入力されていた値は失われます。入力内容を保持しておくことが重要な場合、DOM の状態を スナップショット(snapshot) として記録することができ、ユーザーが戻ってきたときに復元することができます。

これを行うには、`+page.svelte` や `+layout.svelte` で、`capture` メソッドと `restore` メソッドを持つ `snapshot` オブジェクトをエクスポートします：

```
/// file: +page.svelte
<script>
 let comment = '';

 /** @type {import('./$types').Snapshot<string>} */
 export const snapshot = {
 capture: () => comment,
 restore: (value) => comment = value
 };
</script>

<form method="POST">
 <label for="comment">Comment</label>
 <textarea id="comment" bind:value={comment} />
 <button>Post comment</button>
</form>
```

このページから離れるとき、ページが更新される直前に `capture` 関数が呼ばれ、戻り値がブラウザの `history` スタックの現在のエントリーに関連付けられます。もしこのページに戻ってきた場合、ページが更新されるとすぐに保存された値とともに `restore` 関数が呼ばれます。

データは `sessionStorage` に永続化できるように、JSON としてシリアル化可能でなければなりません。これにより、ページがリロードされたときや、ユーザーが別のサイトから戻ってきたときにも、状態を復元することができます。

大きすぎるオブジェクトを `capture` から返さないようにしてください。一度 `capture` されたオブジェクトは、そのセッションの間はメモリ上に保持されるので、極端な場合には、大きすぎて `sessionStorage` に永続化できない可能性があります。

# Packaging

SvelteKit では、アプリを構築するだけでなく、`@sveltejs/package` パッケージを使用してコンポーネントライブラリを構築することもできます (`npm create svelte` にはこれを設定するためのオプションがあります)。

アプリを作成するとき、`src/routes` のコンテンツが公開される部分となります。`src/lib` にはアプリの内部ライブラリが含まれます。

コンポーネントライブラリは SvelteKit アプリと同じ構造ですが、`src/lib` が公開される部分である点が異なります。パッケージの公開にはプロジェクトの最上位(root)の `package.json` が使用されます。`src/routes` を、ライブラリに付属するドキュメントやデモサイト、または開発中に使用するサンドボックスにすることもあるでしょう。

`@sveltejs/package` の `svelte-package` コマンドを実行すると、`src/lib` の中身を取り込み、以下を含む `dist` (設定で変更できます) ディレクトリを生成します:

- `src/lib` にある全てのファイル。Svelte コンポーネントはプリプロセスされ、TypeScript ファイルは JavaScript にトランスペイブルされます。
- Svelte、JavaScript、TypeScript ファイル向けに生成される型定義 (`d.ts` ファイル)。このために、`typescript >= 4.0.0` をインストールする必要があります。型定義はその実装と同じ場所に配置され、手書きの `d.ts` ファイルはそのままコピーされます。生成を無効にすることもできますが、それはおすすめできません。あなたのライブラリを使用する人は TypeScript を使用しているかもしれません。その場合、この型定義ファイルが必要になります。

`@sveltejs/package` バージョン1は `package.json` を生成していましたが、この仕様は変更され、現在はプロジェクトの `package.json` を使用しそれが正しいか検証するようになりました。もしまだバージョン1を使用している場合は、[この PR](#) にある移行手順(Migration instructions)をご覧ください。

## package.json の構造

公開するライブラリをビルドするのであれば、`package.json` の内容がとても重要になります。これを通して、パッケージのエントリーポイントや、どのファイルを npm に公開するか、そしてあなたのライブラリの依存関係を設定することができます。それでは、重要なフィールドをひとつずつ見ていきましょう。

### name

これはパッケージの名前です。他の人はこの名前を使用してインストールすることになります。そして `https://npmjs.com/package/<name>` のように表示されるようになります。

```
{
 "name": "your-library"
}
```

詳細については[こちら](#)をお読みください。

## license

すべてのパッケージにはライセンスフィールドがあるべきです。なぜなら、人々はこれによってどのように使用することが許可されているのか知ることができます。配布や保証なしの再利用に関してとても寛容で、非常にポピュラーなライセンスとして、`MIT` があります。

```
{
 "license": "MIT"
}
```

詳細については[こちら](#)をお読みください。`LICENSE` ファイルもパッケージに含めるとよいでしょう。

## files

ここではどのファイルを `npm` にアップロードするかを設定します。出力先フォルダ（デフォルトは `dist`）も含める必要があります。`package.json` と `README` と `LICENSE` は常に含まれるようになっているため、指定する必要はありません。

```
{
 "files": ["dist"]
}
```

不要なファイル（例えば単体テストや、`src/routes` からのみインポートされているモジュールなど）を除外するには、`.npmignore` ファイルにそれらを追加します。これによってより小さいパッケージとなり、インストールも速くなります。

詳細については[こちら](#)をお読みください。

## exports

`"exports"` フィールドにはパッケージのエントリーポイントを含めます。`npm create svelte@latest` を使用して新しいライブラリプロジェクトをセットアップした場合、単一の `export` として、パッケージの最上位(root)が設定されています：

```
{
 "exports": {
 ".": {
 "types": "./dist/index.d.ts",
 "svelte": "./dist/index.js"
 }
 }
}
```

これにより、あなたのパッケージにはエントリーポイントが1つだけで、それは最上位(root)で、(以下のように)すべてここで通じてインポートされるべきである、ということをバンドラーやツール類に伝えます：

```
// @errors: 2307
import { Something } from 'your-library';
```

`types` と `svelte` キーは `export conditions` です。これはツール類に、`your-library` インポートを検索する際にどのファイルをインポートするかを伝えるものです：

- TypeScript は `types` condition を見て、型定義ファイルを検索します。型定義を公開しない場合は、この condition を省略します。
- Svelte を認識するツール類は `svelte` condition を見て、これが Svelte コンポーネントライブラリであることを認識します。Svelte コンポーネントをエクスポートせず、Svelte 以外のプロジェクトでも使用できるライブラリ（例えば Svelte store ライブラリ）を公開する場合、この condition を `default` に置き換えることができます。

以前のバージョンの `@sveltejs/package` は `package.json` `export` も追加していましたが、現在は違います。すべてのツール類が明示的にエクスポートされていない `package.json` を扱うことができるようになったからです。

`exports` を好みに合わせて調整し、より多くのエントリーポイントを提供することができます。例えば、コンポーネントを再エクスポートする `src/lib/index.js` ファイルの代わりに、`src/lib/Foo.svelte` コンポーネントを直接公開したい場合、以下のような `export map` を作成できます…

```
{
 "exports": {
 "./Foo.svelte": {
 "types": "./dist/Foo.svelte.d.ts",
 "svelte": "./dist/Foo.svelte"
 }
 }
}
```

…そしてライブラリの使用者はコンポーネントをこのようにインポートできます:

```
// @filename: ambient.d.ts
declare module 'your-library/Foo.svelte';

// @filename: index.js
// cut---
import Foo from 'your-library/Foo.svelte';
```

型定義を提供している場合、これを行う際にさらなる注意が必要となることにお気をつけください。注意事項については[こちら](#)をお読みください。

一般的に、`exports map` の各キーはユーザーがあなたのパッケージからなにかインポートするときに使用すべきパスであり、その値はインポートされるファイルへのパスか、またはそのファイルパスを含む `export condition` の `map` です。

`exports` の詳細については[こちら](#)をお読みください。

## svelte

これはツール類に Svelte コンポーネントライブラリを認識できるようにするレガシーなフィールドです。`svelte` `export condition` を使用している場合は必要ありませんが、まだ `export condition` を認識しない古いツールとの後方互換性のために残しておくとよいでしょう。あなたの最上位(root)のエントリーポイントを指しているはずです。

```
{
 "svelte": "./dist/index.js"
}
```

## TypeScript

あなたが TypeScript を使用していないとしても、あなたのライブラリの型定義を公開したほうがよいでしょう。あなたのライブラリを使用する人が、適切なインテリセンスを得られるようになるからです。`@sveltejs/package` は型生成のプロセスをほとんど隠ぺい(opaque)してくれます。デフォルトでは、ライブラリをパッケージングする際、JavaScript、TypeScript、Svelte ファイル向けに型定義を自動生成します。あなたは `exports map` の `types` condition が正しいファイルを指しているか確認するだけです。`npm create svelte@latest` でライブラリプロジェクトを初期化すると、`root export` に `types` condition が設定されます。

しかし、`root export` 以外のもの、例えば `your-library/foo` インポートを提供する場合などは、型定義を提供する上でさらなる注意が必要です。残念ながら、TypeScript はデフォルトでは `{ "./foo": { "types": "./dist/foo.d.ts", ... } }` のような `export` に対して `types` condition を解決しません。代わりに、ライブラリの `root` からの相対で `foo.d.ts` を探します(つまり、`your-library/dist/foo.d.ts` ではなく `your-library/foo.d.ts` です)。これを修正する方法として、選択肢が2つあります：

1つ目の選択肢は、あなたのライブラリを使用する人に対し、`tsconfig.json` (または `jsconfig.json`) の `moduleResolution` オプションに `bundler` (TypeScript 5 から利用可能で、将来的にもベストかつ推奨のオプション) か `node16` か `nodenext` を設定してもらうように要求することです。これによって TypeScript が実際に `exports map` を見て正しく型を解決してくれるようになります。

2つ目の選択肢は、TypeScript の `typesVersions` 機能を使用(乱用)して、型を紐付けます。これは、TypeScript が TypeScript のバージョンによって異なる型定義をチェックするために使用している `package.json` 内のフィールドで、このためにパスマッピング機能があります。このパスマッピング機能を利用して、やりたいことを実現できます。上述の `foo` `export` の場合、対応する `typesVersions` はこのようになります：

```
{
 "exports": {
 "./foo": {
 "types": "./dist/foo.d.ts",
 "svelte": "./dist/foo.js"
 }
 },
 "typesVersions": {
 ">4.0": {
 "foo": ["./dist/foo.d.ts"]
 }
 }
}
```

`>4.0` は、TypeScript のバージョンが 4 より大きい場合、内側の `map` をチェックするよう TypeScript に伝えるものです(実際には常に `true` となるべきものです)。内側の `map` は TypeScript に `your-library/foo` に対する型付けが `./dist/foo.d.ts` にあることを伝えるためのもので、実質 `exports` condition を置き換えていました。また、ワイルドカードとして `*` を自由に使用できるので、同じことを繰り返すことなくたくさんの型定義を一度で使用可能になります。もし `typesVersions` を選択した場合、(`"index.d.ts": [...]` と定義されている) `root import` を含む全ての型のインポートを、これを通して宣言しなければならないことにご注意ください。

この機能についてより詳しい情報は[こちら](#)でお読み頂けます。

## ベストプラクティス

他の SvelteKit プロジェクトからのみ使用されることを想定しているのでなければ、`$app` などの [SvelteKit 固有のモジュール](#) をあなたのパッケージで使用するのは避けたほうがよいでしょう。例えば、`import { browser } from '$app/environment'` を使用するのではなく、`import { BROWSER } from 'esm-env'` ([esm-env ドキュメント参照](#)) を使用します。また、`$app/stores` や `$app/navigation` などに直接頼るのではなく、現在の URL や navigation action をプロパティとして渡すこともできます。より一般的な方法でアプリを書くことによって、テストや UI デモなどのツールのセットアップも簡単になります。

エイリアスは `svelte.config.js` (`vite.config.js` や `tsconfig.json` ではなく) を経由して追加するようにしてください。`svelte-package` で処理されるからです。.

パッケージに加えた変更がバグフィックスなのか、新機能なのか、それとも破壊的変更(breaking change)なのかをよく考えて、それに応じてパッケージのバージョンを更新する必要があります。もし既存のライブラリから `exports` やその中の `export` condition のパスを削除した場合、breaking change とみなされることにご注意ください。

```
{
 "exports": {
 ".": {
 "types": "./dist/index.d.ts",
 // changing `svelte` to `default` is a breaking change:
 - "svelte": "./dist/index.js"
 + "default": "./dist/index.js"
 },
 // removing this is a breaking change:
 - "./foo": {
 - "types": "./dist/foo.d.ts",
 - "svelte": "./dist/foo.js",
 - "default": "./dist/foo.js"
 - },
 // adding this is ok:
 + "./bar": {
 + "types": "./dist/bar.d.ts",
 + "svelte": "./dist/bar.js",
 + "default": "./dist/bar.js"
 + }
 }
 }
}
```

## Options

`svelte-package` は以下のオプションを受け付けます:

- `-w / --watch` — `src/lib` にあるファイルの変更を関してパッケージを再ビルトします
- `-i / --input` — パッケージの全てのファイルを含む入力ディレクトリ。デフォルトは `src/lib` です
- `-o / --o` — 処理されたファイルが書き込まれる出力ディレクトリ。`package.json` の `exports` はここにあるファイルを指さなければならず、`files` の配列にはこのフォルダを含めなければいけません。デフォルトは `dist` です
- `-t / --types` — 型定義 (`d.ts` ファイル) を作成するかどうか。エコシステムのライブラリの品質を向上させるため、作成することを強く推奨します。デフォルトは `true` です

# 公開(Publishing)

生成されたパッケージを公開するには:

```
npm publish
```

## 注意事項

すべての相対ファイルインポートは、Node の ESM アルゴリズムに従って、フルで指定する必要があります。つまり、`src/lib/something/index.js` のようなファイルには、ファイル名と拡張子を含めなければなりません。

```
-import { something } from './something';
+import { something } from './something/index.js';
```

TypeScript を使用している場合、同じように `.ts` ファイルをインポートする必要がありますが、`.js` ファイルの末尾を使用する必要があります、`.ts` ファイルの末尾ではありません（これは TypeScript の設計上の決定によるもので、私たちの管轄外です）。`tsconfig.json` または `jsconfig.json` に `"moduleResolution": "NodeNext"` を設定すると、これに対処できます。

Svelte ファイル（プロプロセスされる）と TypeScript ファイル（JavaScript にトランスマpileされる）を覗いて、全てのファイルはそのままコピーされます。

---

[Go to TOC](#)

# Accessibility

SvelteKit は、アプリにアクセシブルなプラットフォームをデフォルトで提供するよう努めています。Svelte の [コンパイル時のアクセシビリティチェック](#)(compile-time accessibility checks) は、あなたがビルドする SvelteKit アプリケーションにも適用されます。

ここでは、SvelteKit の組み込みのアクセシビリティ(accessibility)機能がどのように動作するか、そしてこれらの機能が可能な限りうまく動作するようにするために必要なことについて説明します。SvelteKit はアクセシブルな基盤を提供しますが、アプリケーションのコードをアクセシブルにするのはあなたの責任であることを覚えておいてください。もし、アクセシビリティ(accessibility)についてよく知らないのであれば、このガイドの "[参考文献](#)" セクションで、その他のリソースを参照してください。

私たちは、アクセシビリティ(accessibility)を正しく行うのは難しいことだと認識しています。SvelteKit のアクセシビリティ対応について改善を提案したい方は、[GitHub issue を作成](#) してください。

## Route announcements

旧来のサーバーレンダリングアプリケーションでは、全てのナビゲーション(例えば、`<a>` タグをクリックするなど)で、ページのフルリロードを引き起こします。これが起こると、スクリーンリーダーやその他の支援技術が新しいページのタイトルを読み上げ、それによってユーザーはページが変更されたことを理解します。

SvelteKit では、ページ間のナビゲーションではページのリロードが発生しないため([クライアントサイドルーティング](#)として知られる)、SvelteKit はナビゲーションごとに新しいページ名が読み上げられるように [ライブリージョン](#)をページに注入します。これは、`<title>` 要素を検査することで、アナウンスするページ名を決定します。

この動作のために、アプリの全ページにユニークで説明的なタイトルを付けるべきです。SvelteKit では、各ページに `<svelte:head>` 要素を配置することでこれを行うことができます:

```
/// file: src/routes/+page.svelte
<svelte:head>
 <title>Todo List</title>
</svelte:head>
```

これにより、スクリーンリーダーやその他の支援技術が、ナビゲーション後に新しいページを識別することができるようになります。説明的なタイトルを提供することは、[SEO](#) にとっても重要なことです。

## フォーカス管理(Focus management)

旧来のサーバーレンダリングアプリケーションでは、ナビゲーションでフォーカスがページのトップにリセットされます。これによって、キーボードやスクリーンリーダーを使用して web をブラウジングする方が、ページの先頭からやり取りできるようになります。

クライアントサイドルーティング中にこの挙動をシミュレートするために、SvelteKit は各ナビゲーションや [強化されたフォーム送信](#)(enhanced form submission) の後、`<body>` 要素にフォーカスを合わせます。1つ例外があります - `autofocus` 属性が付いている要素が存在する場合、SvelteKit はその要素にフォーカスを合わせます。この属性を使用するときは、[支援技術](#)(assistive technology)に対する影響を必ず考慮してください。

SvelteKit のフォーカス管理をカスタマイズしたい場合は、`afterNavigate` hook を使います：

```
/// <reference types="@sveltejs/kit" />
// cut---
import { afterNavigate } from '$app/navigation';

afterNavigate(() => {
 /** @type {HTMLElement | null} */
 const to_focus = document.querySelector('.focus-me');
 to_focus?.focus();
});
```

`goto` 関数を使用して、プログラムで別のページにナビゲーションさせることもできます。デフォルトでは、これはクライアントサイドルーティングでリンクをクリックするのと同じ動作です。しかし `goto` は、`keepFocus` オプションを受け付けます。このオプションは、フォーカスをリセットする代わりに、現在フォーカスされている要素にフォーカスを保持したままにします。このオプションを有効にする場合は、現在フォーカスされている要素がナビゲーション後にもまだ存在することを確かめてください。もしその要素が存在しなければ、ユーザーのフォーカスは失われ、支援技術のユーザーにとって混乱した体験になります。

## The "lang" attribute

デフォルトでは、SvelteKit のページテンプレートには、ドキュメントのデフォルト言語に英語が設定されています。もしコンテンツが英語でない場合、`src/app.html` の `<html>` 要素を更新し、正しい `lang` 属性を持たせる必要があります。これによって、ドキュメントを読む支援技術が正しい発音を使えるようになります。例えば、コンテンツがドイツ語の場合、`app.html` を以下のように更新してください：

```
/// file: src/app.html
<html lang="de">
```

コンテンツが複数の言語で使用可能な場合、開いているページの言語に基づいて `lang` 属性を設定できるようにする必要があります。これは、SvelteKit の `handle` hook を使用して行うことができます：

```
/// file: src/app.html
<html lang="%lang%">

/// file: src/hooks.server.js
/**
 * @param {import('@sveltejs/kit').RequestEvent} event
 */
function get_lang(event) {
 return 'en';
}
// cut---
/** @type {import('@sveltejs/kit').Handle} */
export function handle({ event, resolve }) {
 return resolve(event, {
 transformPageChunk: ({ html }) => html.replace('%lang%', get_lang(event))
 });
}
```

## 参考文献

ほとんどの場合、アクセシブルな SvelteKit アプリを構築するのはアクセシブルな Web アプリを構築するのと同じです。以下の一般的なアクセシビリティ(accessibility)に関するリソースから得られる情報は、どんな Web エクスペリエンスを構築する場合でも適用できるはずです

- [MDN Web Docs: Accessibility](#)
- [The A11y Project](#)
- [How to Meet WCAG \(Quick Reference\)](#)

# SEO

SEO で最も重要なのは、高品質なコンテンツを作ること、そしてそれが web 上で広くリンクされることです。しかし、ランクが高いサイトを構築するためにいくつか技術的に考慮すべきこともあります。

## Out of the box

### SSR

近年、検索エンジンはクライアントサイドの JavaScript でレンダリングされたコンテンツのインデックスを改善してきましたが、サーバーサイドレンダリングされたコンテンツのほうがより頻繁に、より確実にインデックスされます。SvelteKit はデフォルトで SSR を採用しています。`handle` で無効にすることもできますが、適切な理由がない場合はそのままにしておきましょう。

SvelteKit のレンダリングは高度な設定が可能です。必要であれば、[動的なレンダリング\(dynamic rendering\)](#) を実装することも可能です。一般的には推奨されません、SSR には SEO 以外のメリットもあるからです。

### パフォーマンス

[Core Web Vitals](#) のような指標は検索エンジンのランクに影響を与えます。Svelte と SvelteKit はオーバーヘッドが最小限であるため、ハイパフォーマンスなサイトを簡単に構築できます。Google の [PageSpeed Insights](#) や [Lighthouse](#) で、ご自身のサイトをテストすることができます。

### URLの正規化

SvelteKit は、末尾のスラッシュ(trailing slash)付きのパス名から、末尾のスラッシュが無いパス名にリダイレクトします ([設定](#) で逆にできます)。URLの重複は、SEO に悪影響を与えます。

### Manual setup

#### <title> と <meta>

全てのページで、よく練られたユニークな `<title>` と `<meta name="description">` を `<svelte:head>` の内側に置くべきです。説明的な title と description の書き方に関するガイダンスと、検索エンジンにとってわかりやすいコンテンツを作るためのその他の方法については、Google の [Lighthouse SEO audits](#) のドキュメントで見つけることができます。

A common pattern is to return SEO-related `data` from page `load` functions, then use it (as `$page.data`) in a `<svelte:head>` in your root `layout`.

## 構造化データ

構造化データ は、検索エンジンがページのコンテンツを理解するのに役立ちます。 `svelte-preprocess` と一緒に構造化データを使用している場合は、明示的に `ld+json` データを保持する必要があります（これは [将来変更される可能性があります](#)）：

```
/// file: svelte.config.js
// @filename: ambient.d.ts
declare module 'svelte-preprocess';

// @filename: index.js
// cut---
import preprocess from 'svelte-preprocess';

/** @type {import('@sveltejs/kit').Config} */
const config = {
 preprocess: preprocess({
 preserve: ['ld+json']
 // ...
 })
};

export default config;
```

## サイトマップ

サイトマップ は、検索エンジンがサイト内のページの優先順位付けをするのに役立ちます、特にコンテンツの量が多い場合は。エンドポイントを使用してサイトマップを動的に作成できます：

```
/// file: src/routes/sitemap.xml/+server.js
export async function GET() {
 return new Response(
 `

 <?xml version="1.0" encoding="UTF-8" ?>
 <urlset
 xmlns="https://www.sitemaps.org/schemas/sitemap/0.9"
 xmlns:xhtml="https://www.w3.org/1999/xhtml"
 xmlns:mobile="https://www.google.com/schemas/sitemap-mobile/1.0"
 xmlns:news="https://www.google.com/schemas/sitemap-news/0.9"
 xmlns:image="https://www.google.com/schemas/sitemap-image/1.1"
 xmlns:video="https://www.google.com/schemas/sitemap-video/1.1"
 >
 <!-- <url> elements go here -->
 </urlset>`.trim(),
 {
 headers: {
 'Content-Type': 'application/xml'
 }
 }
);
}
```

## AMP

現代の web 開発における不幸な現実として、サイトの Accelerated Mobile Pages (AMP) バージョンを作らなければならないときがある、というのがあります。SvelteKit では、`inlineStyleThreshold` オプションを設定することでこれを実現することができます…

```
/// file: svelte.config.js
/** @type {import('@sveltejs/kit').Config} */
const config = {
 kit: {
 // since <link rel="stylesheet"> isn't
 // allowed, inline all styles
 inlineStyleThreshold: Infinity
 }
};

export default config;
```

…最上位(root)の `+layout.js` / `+layout.server.js` の `csr` を無効にします…

```
/// file: src/routes/+layout.server.js
export const csr = false;
```

…そして、`transformPageChunk` と、`@sveltejs/amp` からインポートできる `transform` を使用して、HTML を変換します：

```
import * as amp from '@sveltejs/amp';

/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
 let buffer = '';
 return resolve(event, {
 transformPageChunk: ({ html, done }) => {
 buffer += html;
 if (done) return amp.transform(html);
 }
 });
}
```

`amphml-validator` を使用して変換された HTML を検証するのに、`handle` hook を利用するのは良いアイデアですが、非常に遅くなってしまうので、ページをプリレンダリングするときだけにしてください。

# Configuration

プロジェクトの設定は、プロジェクトの root にある `svelte.config.js` ファイルに保存されています。SvelteKit だけでなく、エディタ拡張(editor extensions)など Svelte とインテグレーションする他のツールでもこれが使用されます。

```
/// file: svelte.config.js
// @filename: ambient.d.ts
declare module '@sveltejs/adapter-auto' {
 const plugin: () => import('@sveltejs/kit').Adapter;
 export default plugin;
}

// @filename: index.js
// cut---
import adapter from '@sveltejs/adapter-auto';

/** @type {import('@sveltejs/kit').Config} */
const config = {
 kit: {
 adapter: adapter()
 }
};

export default config;
```

TYPES: `@sveltejs/kit#Config`

`kit` プロパティは SvelteKit を設定し、以下のプロパティを持つことができます:

EXPANDED\_TYPES: `@sveltejs/kit#KitConfig`

# Command Line Interface

SvelteKit プロジェクトは [Vite](#) を使用しています。つまり、ほとんどは Vite の CLI を使用することになります (`npm run dev/build/preview` scripts を経由しますが):

- `vite dev` — 開発サーバーを起動します
- `vite build` — アプリの本番バージョンをビルドします
- `vite preview` — ローカルで本番バージョンを実行します

しかしながら SvelteKit には、プロジェクト初期化のための CLI も含まれています:

## svelte-kit sync

`svelte-kit sync` は、`tsconfig.json` と、プロジェクト向けに生成される型 (generated types) を全て作成します (これはルーティングファイルの中で `./$types` としてインポートできます)。プロジェクトを作成すると、`prepare` script としてリストアップされ、npm ライフサイクルの一部として自動的に実行されるので、通常はこのコマンドを実行する必要はないはずです。

# Modules

SvelteKit では、数多くのモジュールがアプリケーションで利用可能です。

MODULES

# Types

## Public types

以下の型は `@sveltejs/kit` からインポートすることができます:

TYPES: `@sveltejs/kit`

## Private types

以下は上記の public types から参照されていますが、直接インポートすることはできません:

TYPES: Private types

## Generated types

`RequestHandler` と `Load` の型はどちらも `Params` 引数を受け取りますが、その `params` オブジェクトに型を付けることができます。例えば、このエンドポイントは `foo`、`bar`、`baz` が渡されることを想定しています:

```
/// file: src/routes/[foo]/[bar]/[baz]/+page.server.js
// @errors: 2355 2322 1360
/** @type {import('@sveltejs/kit').RequestHandler<{
 * foo: string;
 * bar: string;
 * baz: string
 * }>} */
export async function GET({ params }) {
 // ...
}
```

言うまでもなく、これを書くのは面倒で、移植性も低いです (`[foo]` ディレクトリを `[qux]` にリネームした場合、この型は実態を反映していないものとなります)。

この問題を解決するため、SvelteKit は各エンドポイント、各ページごとに `.d.ts` ファイルを生成します:

```
/// file: .svelte-kit/types/src/routes/[foo]/[bar]/[baz]/$types.d.ts
/// link: false
import type * as Kit from '@sveltejs/kit';

type RouteParams = {
 foo: string;
 bar: string;
 baz: string;
}
```

```
export type PageServerLoad = Kit.ServerLoad<RouteParams>;
export type PageLoad = Kit.Load<RouteParams>;
```

TypeScript の設定にある `rootDirs` オプションのおかげで、エンドポイントとページではこれらのファイルが同じディレクトリにあるかのようにインポートすることができます:

```
/// file: src/routes/[foo]/[bar]/[baz]/+page.server.js
// @filename: $types.d.ts
import type * as Kit from '@sveltejs/kit';

type RouteParams = {
 foo: string;
 bar: string;
 baz: string;
}

export type PageServerLoad = Kit.ServerLoad<RouteParams>;

// @filename: index.js
// @errors: 2355
// cut---
/** @type {import('./$types').PageServerLoad} */
export async function GET({ params }) {
 // ...
}
```

```
/// file: src/routes/[foo]/[bar]/[baz]/+page.js
// @filename: $types.d.ts
import type * as Kit from '@sveltejs/kit';

type RouteParams = {
 foo: string;
 bar: string;
 baz: string;
}

export type PageLoad = Kit.Load<RouteParams>;

// @filename: index.js
// @errors: 2355
// cut---
/** @type {import('./$types').PageLoad} */
export async function load({ params, fetch }) {
 // ...
}
```

これを動作させるためには、`tsconfig.json` または `jsconfig.json` が生成された `.svelte-kit/tsconfig.json` を継承する必要があります (`.svelte-kit` の場所は `outDir` です):

```
{ "extends": "./.svelte-kit/tsconfig.json" }
```

## Default tsconfig.json

生成された `.svelte-kit/tsconfig.json` ファイルには様々なオプションが含まれています。いくつかのオプションはプロジェクトの設定に基づいてプログラム的に生成されており、通常は、適切な理由なしに上書きするべきではありません。

```
/// file: .svelte-kit/tsconfig.json
{
 "compilerOptions": {
 "baseUrl": "..",
 "paths": {
 "$lib": "src/lib",
 "$lib/*": "src/lib/*"
 },
 "rootDirs": ["..", "./types"]
 },
 "include": ["../src/**/*.{js,ts,tsx,svelte}"],
 "exclude": ["../node_modules/**", "./**"]
}
```

その他のオプションは SvelteKit が正常に動作するために必要なものであり、変更したときに何が起こるのか把握していないのであれば、そのままにしておく必要があります：

```
/// file: .svelte-kit/tsconfig.json
{
 "compilerOptions": {
 // this ensures that types are explicitly
 // imported with `import type`, which is
 // necessary as svelte-preprocess cannot
 // otherwise compile components correctly
 "importsNotUsedAsValues": "error",

 // Vite compiles one TypeScript module
 // at a time, rather than compiling
 // the entire module graph
 "isolatedModules": true,

 // TypeScript cannot 'see' when you
 // use an imported value in your
 // markup, so we need this
 "preserveValueImports": true,

 // This ensures both `vite build`
 // and `svelte-package` work correctly
 "lib": ["esnext", "DOM", "DOM.Iterable"],
 "moduleResolution": "node",
 "module": "esnext",
 "target": "esnext"
 }
}
```

## App

TYPES: App

[Go to TOC](#)

# Integrations

## Preprocessors

プリプロセッサ(Preprocessors)は、`.svelte` ファイルをコンパイラに渡す前に変換します。例えば、`.svelte` ファイルに TypeScript と PostCSS が使用されている場合、それを最初に JavaScript と CSS に変換し、Svelte コンパイラが処理できるようにしなければなりません。多数の [プリプロセッサが使用可能](#) です。The Svelte team maintains two official ones discussed below.

### vitePreprocess

`vite-plugin-svelte` には `vitePreprocess` という機能があり、Vite をプリプロセスに用いることができます。これによって Vite が扱える言語フレーバー (TypeScript、PostCSS、SCSS、Less、Stylus、SugarSS) の処理が可能になります。便宜上、これは `@sveltejs/kit/vite` パッケージから再エクスポートされています。プロジェクトに TypeScript を設定すると、これがデフォルトで含まれるようになります：

```
// svelte.config.js
import { vitePreprocess } from '@sveltejs/kit/vite';

export default {
 preprocess: [vitePreprocess()]
};
```

### svelte-preprocess

`svelte-preprocess` は、Pug、Babel、global styles のサポートなど、`vitePreprocess` には無い機能があります。しかし、`vitePreprocess` はより速く、設定が少ないため、デフォルトでは `vitePreprocess` が使用されます。SvelteKit は CoffeeScript を [サポートしていない](#) ことにご注意ください。

`svelte-preprocess` をインストールするには `npm install --save-dev svelte-preprocess` を実行し、ご自身で `svelte.config.js` に追加する 必要があります。その後、`npm install -D sass` や `npm install -D less` など、対応するライブラリのインストールが必要になることが多いようです。

## Adders

Svelte Adders は、Tailwind、PostCSS、Storybook、Firebase、GraphQL、mdsvexなど、様々な複雑なインテグレーションを1つのコマンドでセットアップできるようにしてくれます。Svelte と SvelteKitで利用可能なテンプレート、コンポーネント、ツールの全ての一覧については、[sveltesociety.dev](#) をご覧ください。

## Integration FAQs

SvelteKit FAQ に [インテグレーションのセクション](#) があるので、もしまだ不明点があるようでしたら役に立つかもしれません。

---

[Go to TOC](#)

# Sapper からの移行

rank: 1

SvelteKit は Sapper の後継であり、その設計の多くの要素を共有しています。

もし、既存の Sapper アプリを SvelteKit に移行する予定がある場合、いくつかの変更が必要になります。移行する際には、[examples](#) を見ていただくと参考になると思います。

## package.json

### type: "module"

`package.json` に `"type": "module"` を追加します。もし Sapper 0.29.3 以降を使用している場合は、インクリメンタルマイグレーションの一部として、このステップを他のステップとは別に行うことができます。

## dependencies

`polka` や `express` を使用している場合はそれを削除し、`sirv` や `compression` などのミドルウェア(middleware)も削除します。

## devDependencies

`devDependencies` から `sapper` を削除し、`@sveltejs/kit` と使用予定の `adapter` に置き換えます(次のセクションをご覧ください)。

## scripts

`sapper` を参照しているスクリプトを全て更新します:

- `sapper build` は、Node `adapter` を使用した `vite build` に更新します
- `sapper export` は、static `adapter` を使用した `vite build` に更新します
- `sapper dev` は `vite dev` に更新します
- `node __sapper__/build` は `node build` に更新します

## プロジェクトファイル

アプリの大部分を占める `src/routes` の中はそのまま大丈夫ですが、いくつかのプロジェクトファイルを移動または更新する必要があります。

## Configuration

[こちら](#)に記載されている通り、`webpack.config.js` または `rollup.config.js` を `svelte.config.js` に置き換えてください。Svelte の preprocessor オプションは `config.preprocess` に移動してください。

`adapter` を追加する必要があります。`sapper build` は `adapter-node` とおおよそ同じで、`sapper export` は `adapter-static` とおおよそ同じですが、デプロイ先のプラットフォーム向けにデザインされた `adapter` を使用するのも良いでしょう。

[Vite](#) では自動的に処理されないファイルタイプのプラグインを使用している場合は、Vite において同等なことを行う方法を探し、[Vite config](#) に追加する必要があります。

## src/client.js

SvelteKit にはこのファイルに相当するものはありません。カスタムロジック (`sapper.start(...)` 以降) は、`+layout.svelte` ファイルで、`onMount` コールバック内に記述してください。

## src/server.js

`adapter-node` を使用する場合は、`custom server` がこれと同等のものです。それ以外の場合は、同等のものに該当するものはありません。なぜなら SvelteKit アプリはサーバーレス環境でも実行可能だからです。

## src/service-worker.js

`@sapper/service-worker` からインポートするほとんどのものは、`$service-worker` に同等なものがあります：

- `files` は変更されていません
- `routes` は削除されました
- `shell` は現在 `build` になりました
- `timestamp` は現在 `version` になりました

## src/template.html

`src/template.html` は `src/app.html` にリネームする必要があります。

`%sapper.base%`、`%sapper.scripts%`、`%sapper.styles%` を削除します。`%sapper.head%` を `%sveltekit.head%` に、`%sapper.html%` を `%sveltekit.body%` にそれぞれ置き換えます。`<div id="sapper">` はもう必要ありません。

## src/node\_modules

Sapper アプリでよくあるパターンとして、内部ライブラリを `src/node_modules` 内のディレクトリに配置する、というものがあります。これは Vite だと動作しないため、代わりに `src/lib` を使用します。

## ページとレイアウト

### 名前が変わったファイル

ルート(Routes)は曖昧さをなくすためフォルダ名のみで構成されるようになり、`+page.svelte` までのフォルダ名がルート(route)に対応するようになりました。概要は [ルーティングのドキュメント](#) をご参照ください。以下は 新/旧 の比較です：

Old	New
routes/about/index.svelte	routes/about/+page.svelte
routes/about.svelte	routes/about/+page.svelte

カスタムのエラーページコンポーネントは `_error.svelte` から `+error.svelte` にリネームしてください。また、どの `_layout.svelte` ファイルも、同様に `+layout.svelte` にリネームしてください。[その他のファイルは無視されます](#)。

## Imports

`@sapper/app` からインポートしていた `goto`、`prefetch`、`prefetchRoutes` は、`$app/navigation` からインポートする `goto`、`preloadData`、`preloadCode` にそれぞれ置き換えてください。

`@sapper/app` からインポートしていた `stores` については置き換える必要があります — 以下の [Stores](#)) をご覧ください。

`src/node_modules` にあるディレクトリからインポートしてたファイルは、`$lib` からのインポートに置き換えてください。

## Preload

以前と同様に、ページやレイアウトではレンダリングが行われる前にデータをロードできる関数をエクスポートすることができます。

この関数は `preload` から `load` にリネームされ、その API が変更されました。2つの引数 — `page` と `session` — の代わりに、両方を 1つにまとめた引数と、`fetch` (`this.fetch` からの置き換え)、そして新たに `stuff` オブジェクトが追加されました。

`this` オブジェクトはなくなり、その結果 `this.fetch`、`this.error`、`this.redirect` もなくなりました。代わりに、`fetch` を `input` メソッドから使用できるようになり、`error` と `redirect` の両方がスローされるようになりました。This function has been renamed from `preload` to `load`, it now lives in a `+page.js` (or `+layout.js`) next to its `+page.svelte` (or `+layout.svelte`), and its API has changed. Instead of two arguments — `page` and `session` — there is a single `event` argument.

There is no more `this` object, and consequently no `this.fetch`, `this.error` or `this.redirect`. Instead, you can get `fetch` from the input methods, and both `error` and `redirect` are now thrown.

## Stores

Sapper では、提供されるストアをこのように参照していたかと思います:

```
// @filename: ambient.d.ts
declare module '@sapper/app';

// @filename: index.js
// cut---
import { stores } from '@sapper/app';
const { preloading, page, session } = stores();
```

`page` と `session` ストアはまだ存在しています。`preloading` は、`from` プロパティと `to` プロパティを含む `navigating` ストアに置き換えられました。`page` は `url`、`params` を持つようになりましたが、`path` と `query` はありません。

SvelteKit では、それらにアクセスする方法が異なります。`stores` は `getStores` になりましたが、`$app/stores` から直接 `navigating`、`page`、`session` をインポートできるので、ほとんどの場合は必要ありません。

## ルーティング

ルート(routes) の正規表現はもうサポートされていません。代わりに、[advanced route matching](#) をお使いください。

## Segments

以前までは、レイアウトコンポーネントは子のセグメントを表す `segment` プロパティを受け取っていましたが、この機能は削除されました。より柔軟な `$page.url.pathname` の値を使用し、お望みのセグメントを取得してください。

## URLs

Sapper では、相対 URL は、現在のページに対してではなく、base URL (`basepath` オプションが使用されていない限り、大抵の場合は `/`) に対して解決されていました。

これによって問題が発生していましたが、SvelteKit ではもうそのようなことはありません。相対 URL が現在のページ(または `load` 関数の `fetch` URL の場合は移動先のページ) に対して解決されるようになりました。多くの場合、(例えば、`/` 始まるような) ルート相対な URL を使用するほうが簡単です。なぜなら、それらの意図がコンテキストに依存しないからです。

## <a> attributes

- `sapper:prefetch` は `data-sveltekit-preload-data` になりました
- `sapper:noscroll` は `data-sveltekit-noscroll` になりました

## Endpoints

Sapper では、[サーバールート\(server routes\)](#) は、Node の `http` モジュールによって公開される `req` と `res` オブジェクト(または Polka や Express などのフレームワークが提供するその拡張版)を受け取っていました。

SvelteKit は、アプリが動作する場所に依存しないように設計されています(Node サーバーで動作し、サーバーレスプラットフォームや Cloudflare Worker でも同様に動作します)。そのため、もう `req` と `res` を直接扱いません。エンドポイントを、新しいシグネチャに合わせて更新する必要があります。

環境非依存な動作をサポートするため、グローバルコンテキストで `fetch` が利用できるようになり、`node-fetch` や `cross-fetch` などのサーバーサイドの `fetch` 実装をインポートする必要がなくなりました。

## インテグレーション

インテグレーションに関する詳細情報については [FAQ](#) をご参照ください。

## HTML minifier

Sapper はデフォルトで `html-minifier` を含んでいました。SvelteKit はこれを含まないのですが、本番環境向けの依存関係(prod dependency)としてこれを追加し、`hook` で使用することができます：

```
// @filename: ambient.d.ts
/// <reference types="@sveltejs/kit" />
declare module 'html-minifier';

// @filename: index.js
// cut---
import { minify } from 'html-minifier';
import { building } from '$app/environment';

const minification_options = {
 collapseBooleanAttributes: true,
 collapseWhitespace: true,
 conservativeCollapse: true,
 decodeEntities: true,
 html5: true,
 ignoreCustomComments: [/^#/],
 minifyCSS: true,
 minifyJS: false,
 removeAttributeQuotes: true,
 removeComments: false, // some hydration code needs comments, so leave them in
 removeOptionalTags: true,
 removeRedundantAttributes: true,
 removeScriptTypeAttributes: true,
 removeStyleLinkTypeAttributes: true,
 sortAttributes: true,
 sortClassName: true
};

/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
 let page = '';

 return resolve(event, {
 transformPageChunk: ({ html, done }) => {
 page += html;
 if (done) {
 return building ? minify(page, minification_options) : page;
 }
 }
 });
}
```

サイトのプロダクションビルドをテストするのに `vite preview` を使用しているときは、`prerendering` が `false` となることにご注意ください。そのため、`minify` の結果を検証するには、ビルド済の HTML を直接確認する必要があります。

---

[Go to TOC](#)

# Additional resources

## FAQs

よくある問題の解決方法や役に立つ tips や tricks については、[SvelteKit FAQ](#) をご覧ください。

[Svelte FAQ](#) と [vite-plugin-svelte FAQ](#) も、これらのライブラリに起因する疑問点には役立つでしょう。

## Examples

例として、数種類 SvelteKit サイトを作成し、公開しています：

- [sveltejs/realworld](#) にはブログサイトの例があります
- [sites/kit.svelte.dev ディレクトリ](#) にはこのサイトのコードがあります
- [sveltejs/sites](#) には [svelte.dev](#) と [HackerNews のクローン](#) のコードがあります

また、SvelteKit ユーザーが GitHub で [#sveltekit](#) や [#sveltekit-template](#) というトピックを付けて多くの例を公開しており、[Svelte Society のサイト](#) にも例が公開されています。なお、これらはメンテナーによって検証されておらず、最新ではない可能性もありますのでご注意ください。

## サポート

[Discord](#) や [StackOverflow](#) でヘルプを求めるすることができます。他の方の時間を尊重するため、まずは FAQ、Google または他の検索エンジン、issue tracker、Discord のチャット履歴などから、問題に関連する情報を検索してください。回答する方より質問する方のほうが多いので、こうすることでコミュニティをスケーラブルに発展させることができます。

日本語翻訳版 追記：上記の Discord は Svelte 本体のもので、英語でコミュニケーションが行われています。もし日本語で質問したり交流したいのであれば、[Svelte 日本の Discord](#) にどうぞ！

# Glossary

SvelteKit のコアは、高度に設定可能(configurable)なレンダリングエンジンを提供します。このセクションでは、レンダリングについてディスカッションする際に使用されるいくつかの用語を説明します。これらのオプションを設定するためのリファレンスは、上記のドキュメントで提供されています。

## CSR

クライアントサイドレンダリング(Client-side rendering (CSR))とは、JavaScript を使用して Web ブラウザ上でページコンテンツを生成することです。

SvelteKit では、デフォルトでクライアントサイドレンダリングが使用されますが、`csr = false` ページオプションで JavaScript をオフにすることができます。

## ハイドレーション

Svelte コンポーネントは、何らかの状態を保存し、状態が更新されると DOM を更新します。SSR 中にデータをフェッチするとき、デフォルトでは SvelteKit はこのデータを保存し、サーバーレンダリングされた HTML と一緒にクライアントに送信します。それからコンポーネントは、同じ API エンドポイントを再度呼び出すことなく、クライアント側でそのデータを使用して初期化されます。そして Svelte はハイドレーション(Hydration)と呼ばれるプロセスで DOM が想定通りの状態にあることをチェックしてイベントリスナーをアタッチします。コンポーネントが完全にハイドレートされると、新しく作成された Svelte コンポーネントと同じように、プロパティの変更に反応(react)することができます。

SvelteKit では、デフォルトでページがハイドレーションされますが、`csr = false` ページオプションで JavaScript をオフにすることができます。

## プリレンダリング

プリレンダリング(Prerendering)とは、ビルト時にページのコンテンツを計算し、表示のために HTML を保存しておくことを意味します。このアプローチは旧来のサーバーレンダリングページと同じ利点を持ちつつ、訪問者ごとにページを再計算する必要がないため、訪問者数の増加に対してほぼ無償でスケールします。トレードオフとしては、ビルトプロセスのコストがより高くなり、プリレンダリングされたコンテンツはアプリケーションの新しいバージョンをデプロイすることでしか更新できなくなります。

全てのページがプリレンダリングできるわけではありません。基本的なルールは次の通りです：コンテンツがプリレンダリング可能であると言うためには、それを直接表示する 2 人のユーザーが、サーバーから同じコンテンツを取得できなければならず、かつ、そのページには `actions` が含まれていてはいけません。全てのユーザーが同じプリレンダリングコンテンツを見ることができるのであれば、ページのパラメータに基づいてロードされるコンテンツはプリレンダリングが可能である点にご注意ください。

プリレンダリングされるページは静的なコンテンツに限りません。クライアントサイドでユーザ固有のデータをフェッチしてレンダリングする場合は、パーソナライズされたページを構築することができます。ただし、前述の通り、そのコンテンツに対して SSR を行わないことによるデメリットが発生することにご注意ください。

SvelteKit では、`prerender` ページオプション と、`svelte.config.js` の `prerender` コンフィグ によってプリレンダリングをコントロールできます。

## ルーティング

デフォルトでは、(リンクをクリックしたりブラウザの進むまたは戻るボタンを使って)新しいページにナビゲートするとき、SvelteKit はナビゲーションをインターフェトし、ブラウザが移動先のページのリクエストをサーバーにリクエストする代わりに、それを処理します。それから SvelteKit は新しいページのコンポーネントをレンダリングし、そのときに順番に必要な API エンドポイントをコールし、クライアントの表示コンテンツを更新します。このような、ナビゲーションが行われる際にそれに応じてクライアント側でページを更新するプロセスのことを、クライアントサイドルーティングと呼びます。

SvelteKit では、デフォルトでクライアントサイドルーティングが使用されますが、`data-sveltekit-reload` でこれをスキップすることができます。

## SPA

シングルページアプリ(single-page app (SPA))とは、サーバーへの全てのリクエストで単一の HTML をロードし、コンテンツのクライアントサイドレンダリングをリクエストされた URL に基づいて行うアプリケーションのことです。全てのナビゲーションはクライアントサイドで(クライアントサイドルーティングと呼ばれるプロセスで)処理され、ページごとのコンテンツは更新されるが共通のレイアウト要素はほとんど更新されません。SPA は SSR を提供しないため、上記のような欠点があります。しかし、SEO が重要ではない、ユーザーが一貫したコンピューティング環境からアプリケーションにアクセスすることがわかっているような、ログインの背後に複雑なビジネスアプリケーションなどの場合は、これらの欠点による大きな影響を受けません。

SvelteKit では、`adapter-static` を使って SPA を構築 することができます。

## SSG

静的サイト生成(Static Site Generation (SSG))とは、全てのページがプリレンダリングされているサイトを指す用語です。SvelteKit は 静的サイト生成だけを行うために作られたわけではないので、その目的のために特別に作られたツールが非常に多くのページを効率的にレンダリングするのと同じようにスケールしないかもしれません。しかし、目的に特化した SSG とは対照的に、SvelteKit はページごとに異なるレンダリングタイプをミックスしたりマッチさせたりすることができます。サイトを完全にプリレンダリングすることの利点の 1 つは、SSR を実行するためのサーバーを維持したり費用を払ったりする必要がないことです。一度生成すれば、そのサイトは CDN から提供することができ、"time to first byte" の優れたパフォーマンスにつながります。このデリバリー・モデルはしばしば JAMstack と呼ばれます。

SvelteKit では、`adapter-static` を使用したり、`prerender` ページオプション や `svelte.config.js` の `prerender` コンフィグ で全ページをプリレンダリングするようにしたりすることで SSG を行うことができます。

## SSR

サーバーサイドレンダリング(Server-side rendering (SSR))とは、サーバー上でページコンテンツを生成することです。SSR は一般的に SEO の観点で好まれます。クライアントサイドで生成される動的なコンテンツをインデックスできる検索エンジンもありますが、その場合でもそれに時間がかかることがあります。知覚的なパフォーマンスも改善される傾向にあり、もし JavaScript が失敗したり無効になっている場合でも(あなたが思うより頻繁に発生しています)、ユーザーがアプリにアクセスできるようになります。

SvelteKit では、デフォルトでページがサーバーサイドレンダリングされます。`ssr ページオプション` で SSR を無効化できます。

# 他のリソース

[Svelte FAQ](#) と [vite-plugin-svelte FAQ](#) も、これらのライブラリに起因する疑問点には役立つのでご参照ください。

# SvelteKit で HMR を使うにはどうすればよいですか？

SvelteKit は [svelte-hmr](#) によってデフォルトで HMR が有効になっています。Rich の 2020 Svelte Summit のプレゼンテーションを見たことがあるなら、より強力そうに見えるバージョンの HMR をご覧になったかもしれません。あのデモでは `svelte-hmr` の `preserveLocalStorage` フラグがオンになっていました。このフラグは想定外の動作やエッジケースにつながる可能性があるため、現在はデフォルトでオフになっています。でもご心配なく、SvelteKit で HMR を利用することはできます！もしローカルの状態を保持したい場合は、[svelte-hmr](#) ページに説明があるように、`@hmr:keep` または `@hmr:keep-all` ディレクティブを使用することができます。

---

[Go to TOC](#)

# package.json の詳細をアプリケーションに含めるにはどうすればよいですか？

SvelteKit は `svelte.config.js` を ES module として想定しているため、JSON ファイルを直接要求することはできません。もしアプリケーションに `package.json` からバージョン番号またはその他の情報を含めたい場合は、このように JSON をロードすることができます：

```
/// file: svelte.config.js
// @filename: index.js
/// <reference types="@types/node" />
import { URL } from 'url';
// cut---
import { readFileSync } from 'fs';
import { fileURLToPath } from 'url';

const file = fileURLToPath(new URL('package.json', import.meta.url));
const json = readFileSync(file, 'utf8');
const pkg = JSON.parse(json);
```

# パッケージをインクルードしようとするとエラーが発生するのですが、どうすれば直せますか？

ライブラリのインクルードに関する問題は、ほとんどが不適切なパッケージングによるものです。ライブラリのパッケージングが Node.js に対応しているかどうかは、[publint の web サイト](#) でチェックできます。

以下は、ライブラリが正しくパッケージングされているかどうかをチェックする際に気を付けるべき点です：

- `exports` は `main` や `module` などの他のエントリーポイントのフィールドよりも優先されます。`exports` フィールドを追加すると、`deep import` を妨げることになるため、後方互換性が失われる場合があります。
- "type": "module" が指定されていない限り、ESM ファイルは `.mjs` で終わる必要があり、CommonJS ファイルは `.cjs` で終わる必要があります。
- `exports` が定義されていない場合、`main` を定義する必要があり、それは CommonJS ファイルか ESM ファイルでなければならず、前項に従わなければならない。`module` フィールドが定義されている場合、ESM ファイルを参照している必要があります。
- Svelte コンポーネントは、コンパイルされていない `.svelte` ファイルとして配布し、パッケージに含まれる JS は ESM のみとして記述していかなければなりません。TypeScript などのカスタムスクリプトや SCSS などのスタイル言語は、それぞれ vanilla JS と CSS にするために前処理(preprocess)をしなければなりません。Svelte ライブラリのパッケージングには、`svelte-package` を使用することを推奨しています。このパッケージによって、これらの作業が行われます。

ライブラリが ESM バージョンを配布している場合、特に Svelte コンポーネントライブラリがその依存関係に含まれている場合、Vite を使用するとブラウザ上で最適に動作します。ライブラリの作者に ESM バージョンを提供するよう提案すると良いでしょう。しかし、CommonJS (CJS) の依存関係も上手く扱えるようにするために、デフォルトで、`vite-plugin-svelte` が Vite にそれらを事前バンドルするよう指示します。Vite は `esbuild` を使ってそれらを ESM に変換します。

それでもまだ問題が解消されない場合は、[Vite の issue tracker](#) と該当のライブラリの issue tracker を検索することを推奨します。`optimizeDeps` や `ssr` の設定値をいじることで問題を回避できる場合もありますが、これはあくまで一時的な回避策とし、問題のあるライブラリの修正を優先したほうが良いでしょう。

# SvelteKit で X を使うにはどうすればよいですか？

[ドキュメントのインテグレーションのセクション](#) をしっかり読み込んでください。それでも問題が解決しない場合のために、よくある問題の解決策を以下に示します。

## データベースのセットアップはどう行えばよいですか？

データベースに問い合わせを行うコードを [サーバールート \(server route\)](#) に置いてください。.svelte ファイルの中でデータベースに問い合わせを行わないでください。コネクションをすぐにセットアップし、シングルトンとしてアプリ全体からクライアントにアクセスできるように `db.js` のようなものを作ることができます。`hooks.js` で1回セットアップするコードを実行し、データベースヘルパーを必要とするすべてのエンドポイントにインポートできます。

## document や window に依存しているクライアントサイドオンリーなライブラリはどう使えばよいですか？

もし `document` や `window` 変数にアクセスする必要があったり、クライアントサイドだけで実行するコードが必要な場合は、`browser` チェックでラップしてください:

```
/// <reference types="@sveltejs/kit" />
// cut---
import { browser } from '$app/environment';

if (browser) {
 // client-only code here
}
```

コンポーネントが最初に DOM にレンダリングされた後にコードを実行したい場合は、`onMount` で実行することもできます:

```
// @filename: ambient.d.ts
// @lib: ES2015
declare module 'some-browser-only-library';

// @filename: index.js
// cut---
import { onMount } from 'svelte';

onMount(async () => {
 const { method } = await import('some-browser-only-library');
 method('hello world');
});
```

使用したいライブラリに副作用がなければ静的にインポートすることができますし、サーバー側のビルドでツリーシェイクされ、`onMount` が自動的に `no-op` に置き換えられます:

```
// @filename: ambient.d.ts
// @lib: ES2015
declare module 'some-browser-only-library';

// @filename: index.js
```

```
// cut--
import { onMount } from 'svelte';
import { method } from 'some-browser-only-library';

onMount(() => {
 method('hello world');
});
```

一方、ライブラリに副作用があっても静的にインポートをしたい場合は、[vite-plugin-iso-import](#) をチェックして [client](#) インポートサフィックスをサポートしてください。このインポートは SSR ビルドでは取り除かれます。しかし、この手法を使用すると VS Code Intellisense が使用できなくなることにご注意ください。

```
// @filename: ambient.d.ts
// @lib: ES2015
declare module 'some-browser-only-library?client';

// @filename: index.js
// cut--
import { onMount } from 'svelte';
import { method } from 'some-browser-only-library?client';

onMount(() => {
 method('hello world');
});
```

## 別のバックエンド API サーバーを使用するにはどうすれば良いですか？

外部の API サーバーにデータをリクエストするのに `event.fetch` を使用することができますが、[CORS](#) に対応しなければならず、一般的にはリクエストのプリフライトが必要になり、結果として高レイテンシーになるなど、複雑になることにご注意ください。別のサブドメインへのリクエストも、追加の DNS ルックアップや TLS セットアップなどのためにレイテンシーが増加する可能性があります。この方法を使いたい場合は、`handleFetch` が参考になるかもしれません。

別のある方法は、頭痛の種である CORS をバイパスするためのプロキシーをセットアップすることです。本番環境では、`/api` などのパスを API サーバーに書き換えます(rewrite)。ローカルの開発環境では、Vite の `server.proxy` オプションを使用します。

本番環境で書き換え(rewrite)をセットアップする方法は、デプロイ先のプラットフォームに依存します。もし、書き換える方法がなければ、代わりに [API route](#) を追加します：

```
/// file: src/routes/api/[...path]/+server.js
/** @type {import('./$types').RequestHandler} */
export function GET({ params, url }) {
 return fetch(`https://my-api-server.com/${params.path + url.search}`);
}
```

(必要に応じて、`POST / PATCH` などのリクエストもプロキシし、`request.headers` も転送(forward)する必要があることにご注意ください)

## ミドルウェア(middleware)を使うにはどうすればよいですか？

`adapter-node` は、プロダクションモードで使用するためのミドルウェアを自分のサーバで構築します。開発モードでは、Vite プラグインを使用して Vite にミドルウェア(middleware)を追加することができます。例えば：

```
// @filename: ambient.d.ts
declare module '@sveltejs/kit/vite'; // TODO this feels unnecessary, why can't it
//'see' the declarations?

// @filename: index.js
// cut---
import { sveltekit } from '@sveltejs/kit/vite';

/** @type {import('vite').Plugin} */
const myPlugin = {
 name: 'log-request-middleware',
 configureServer(server) {
 server.middlewares.use((req, res, next) => {
 console.log(`Got request ${req.url}`);
 next();
 });
 }
};

/** @type {import('vite').UserConfig} */
const config = {
 plugins: [myPlugin, sveltekit()]
};

export default config;
```

順序を制御する方法など、詳しくは [Vite の `configureServer` のドキュメント](#) をご覧ください。

## Yarn 2 で動作しますか？

多少は。Plug'n'Play 機能、通称 'pnp' は動きません (Node のモジュール解決アルゴリズムから逸脱しており、SvelteKit が数多くのライブラリとともに使用しているネイティブの JavaScript モジュールではまだ動作しません)。  
`.yarnrc.yml` で `nodeLinker: 'node-modules'` を使用して pnp を無効にできますが、おそらく npm や `pnpm` を使用するほうが簡単でしょう。同じように高速で効率的ですが、互換性に頭を悩ませることはありません。

## Yarn 3 を使用するにはどうすれば良いですか？

現時点の、最新の Yarn (version 3) の ESM サポートは `experimental` であるようです。

結果は異なるかもしれません、下記が有効なようです。

最初に新しいアプリケーションを作成します：

```
yarn create svelte myapp
cd myapp
```

そして Yarn Berry を有効にします：

```
yarn set version berry
yarn install
```

### Yarn 3 global cache

Yarn Berry の興味深い機能の1つに、ディスク上のプロジェクトごとに複数のコピーを持つのではなく、パッケージ用に単一のグローバルキャッシュを持つことができる、ということがあります。しかし、`enableGlobalCache` の設定を `true` になるとビルドが失敗するため、`.yarnrc.yml` ファイルに以下を追加することを推奨します：

```
nodeLinker: node-modules
```

これによってパッケージはローカルの `node_modules` ディレクトリにダウンロードされますが、上記の問題は回避され、現時点では Yarn の version 3 を使用するベストな方法となります。

# Colophon

This book is created by using the following sources:

- Sveltekit - 日本語
- GitHub source: [svelte-jp/kit/documentation](https://github.com/svelte-jp/kit/documentation)
- Created: 2023-08-25
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- Viewer: <https://read-html-download-pdf.vercel.app/>
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>
- Viewer repo: <https://github.com/shinokada/read-html-download-pdf>