

# SVELTEKIT Docs - 日本語



## Table of contents

• Getting started - Introduction	3
• Getting started - Creating a /project	4
• Getting started - Project structure	5
• Getting started - Web standards	8
• Core concepts - Routing	11
• Core concepts - Load	19
• Core concepts - Form actions	30
• Core concepts - Page options	39
• Core concepts - Adapters	43
• Advanced - Advanced routing	46
• Advanced - Hooks	52
• Advanced - Errors	57
• Advanced - Link options	60
• Advanced - Service workers	62
• Advanced - Server only /modules	63
• Advanced - Assets	65
• Advanced - Packaging	66
• Best practices - Accessibility	68
• Best practices - Seo	71
• Api reference - Configuration	74
• Api reference - Cli	82
• Api reference - Modules	83
• Api reference - Types	84
• Appendix - Migrating	87
• Appendix - Additional resources	92
• Appendix - Glossary	93
• FAQ - Other resources	95
• FAQ - Version numbers	96
• FAQ - Hmr	97
• FAQ - Adapters	98
• FAQ - Read package /json	99
• FAQ - Packages	100
• FAQ - Integrations	101

# イントロダクション

## 始める前に

SvelteKit は 1.0 に向けたリリース候補段階 (release candidate phase) で、報告された issue に対応しつつ、磨きをかけています。もし行き詰まったら、[Discord チャットルーム](#)にどうぞ。

Sapper からのアップグレードについては [migration guides\(移行ガイド\)](#) をご覧ください。

## SvelteKitとは

SvelteKitはとてもハイパフォーマンスなWebアプリを構築するためのフレームワークです。

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。これらのプラクティスには、必要最小限のコードのみをロードするための[ビルドの最適化](#)、[オフラインサポート](#)、[プリフェッチ](#)（ユーザーがナビゲーションを開始する前にページを取得すること）、[設定可能なレンダリング](#)（アプリのレンダリングをサーバー上で行うか[ブラウザ上](#)で行うか、それをランタイムで行うか[ビルド時](#)に行うかなど）が含まれています。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。

[Vite](#) を [Svelte plugin](#) 経由で使用しており、非常に高速で機能豊富な開発体験を提供します。[Hot Module Replacement \(HMR\)](#)により、コードを変更すると即座にブラウザに反映されます。

このガイドを理解するためにSvelteを知っておく必要はありませんが、知っていれば役に立つでしょう。簡単に紹介しておくと、Svelteはコンポーネントを高度に最適化されたvanilla JavaScriptにコンパイルしてくれるUIフレームワークです。詳しくは、[Svelteを紹介するブログ記事](#)と[Svelteチュートリアル](#)を読んでみてください。

---

[Go to TOC](#)

# 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。もしかしたら、もうこれ以降のガイドを読む必要はないかもしれません！

## 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.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します。

- `lib` にはあなた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 `$lib` エイリアスを使用してインポートしたり、`svelte-package` を使用して配布用にパッケージングすることができます。
  - `server` にはあなたのサーバー専用のライブラリのコードを格納します。格納されたコードは `$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
- `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)
- `error.html` (optional) 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.js` (optional) アプリケーションの [hooks](#)
- `service-worker.js` (optional) [service worker](#)

TypeScript を使用している場合、`.js` の代わりに `.ts` ファイルを使用することができます。

## 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

`npm create svelte@latest` の実行時、プロジェクトにテストを追加することを選択した場合、このディレクトリに格納されます。

## 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 [コンフィグレーション](#) 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](#) をともに使用した [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)。

---

[Go to TOC](#)

# 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。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、SvelteKit が受信した `request.headers` を読むことと、送信する `response.headers` をセット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(event) {
    // log all headers
    console.log(...event.request.headers);

    return json({
        // retrieve a specific header
        userAgent: event.request.headers.get('user-agent')
    });
}
```

## 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();
```

---

[Go to TOC](#)

# 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` (または、TypeScript を使いの場合は `+page.ts`) 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。

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` にそれら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。唯一の要件事項は、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](https://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、ページ間をナビゲーション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 (または `+server.ts`) は `GET`、`POST`、`PATCH`、`PUT`、`DELETE` などの 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。

## Receiving data

`+server.js` ファイルは、`POST` / `PUT` / `PATCH` / `DELETE` ハンドラー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` 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` 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。

## \$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。

## 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リティモジュールを、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。

# 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`（または `+page.ts`）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。`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。サーバー専用の `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、[Shared 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 here -->
  <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。

## Shared 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` ファイルがエクスポートする `load` 関数は、サーバーとブラウザで共有(shared)されます

- `+page.server.js` ファイルと `+layout.server.js` ファイルがエクスポートする `load` 関数は、サーバー専用です

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。

## Input

共有される `load` 関数とサーバー専用の `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 (`depends`、`fetch`、`parent`) 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。

サーバー専用の `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。

共有される `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、サーバー専用の `load` 関数の戻り値が、共有される `load` 関数の引数の `data` プロパティとなります。

## Output

共有される `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。

サーバー専用の `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` でシリアル化できるデータ、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。

## どちらを使用すべきか

サーバー専用の `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。

共有される `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、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、シリアルライズ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

サーバー専用の `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さってください。

サーバー専用の `load` 関数と共有される `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
/// 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 };
}
```

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

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

```

`+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');
    }
}

```

## Promise unwrapping

トップレベルの promise は await されるので、ウォーターフォールを作ることなく、複数の promise を簡単に返すことができます:

```

/// file: src/routes/+page.server.js
/** @type {import('./$types').PageServerLoad} */
export function load() {
    return {
        a: Promise.resolve('a'),
        b: Promise.resolve('b'),
        c: {
            value: Promise.resolve('c')
        }
    };
}

```

```

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

    console.log(data.a); // 'a'
    console.log(data.b); // 'b'
    console.log(data.c.value); // `Promise {...}`
</script>

```

## 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、複数のサーバー専用 `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。

## Invalidation

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。

## 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。

`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る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 { invalidateAll } from '$app/navigation';

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

  function rerunLoadFunction() {
    // any of these will cause the `load` function to re-run
    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>
```

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` のプロパティを参照していて（例えば `url.pathname` や `url.search`）、その値が変更された場合
- `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` 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

`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。

## 状態の共有(Shared state)

多くのサーバー環境では、アプリの単一のインスタンスが複数のユーザーにサービスを提供することになります。そのため、リクエストごと(per-request)、ユーザーごと(per-user)の状態を `load` 関数の外側の共有変数に保存してはいけません。代わりに、`event.locals` に保存するようにしてください。

# 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">
  <input name="email" type="email">
  <input name="password" type="password">
  <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">
  <input name="email" type="email">
  <input name="password" type="password">
  <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: 2339 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。`invalid` 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、`data` は `form` から使用することができます：

```

// @errors: 2339 2304
/// file: src/routes/login/+page.server.js
+import { invalid } 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 invalid(400, { email, missing: true });
        +    }

        const user = await db.getUser(email);

        +    if (!user || user.password !== hash(password)) {
        +        return invalid(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">
-   <input name="email" type="email">
+   {#if form?.missing}<p class="error">The email field is required</p>{#/if}
+   {#if form?.incorrect}<p class="error">Invalid credentials!</p>{#/if}
+   <input name="email" type="email" value={form?.email ?? ''}>

    <input name="password" type="password">
    <button>Log in</button>
    <button formaction="/register">Register</button>
</form>

```

戻り値は JSON としてシリアル化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:

```

// @errors: 2339 2304
/// file: src/routes/login/+page.server.js
+import { invalid, 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 invalid(400, { email, missing: true });
        }

        if (user.password !== hash(password)) {
            return invalid(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。これは素晴らしいことですが、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 のインター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。
- 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

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={({ form, data, action, cancel }) => {
    // `form` は `<form>` 要素です
    // `data` はその `FormData` オブジェクトです
    // `action` はフォームが POST される URL です
    // `cancel()` は送信(submission)を中止します

    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ロジックを起動する
      // 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：

```
<script>
+  import { enhance, applyAction } from '$app/forms';

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

<form
  method="POST"
  use:enhance={({ form, data, action, cancel }) => {
    // `form` は `<form>` 要素です
    // `data` はその `FormData` オブジェクトです
    // `action` はフォームが POST される URL です
    // `cancel()` は送信(submission)を中止します

    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, invalid` — `$page.status` を`result.status` に設定し、`form` と`$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

## 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') {
      // re-run 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。

---

[Go to TOC](#)

# 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 [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、[prerender configuration](#) の `entries` オプションでこれを指定することができます。

## 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` の中で)。

`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。

## 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` を作成しようとしますが、これは不可能です。

このため(他にも理由はありますか)、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。

ルーターがすでにアクティブかどうかにかかわらず、このページからのすべてのナビゲーションでクライアントサイドルーティングが無効になることにご注意ください。

## 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` からのリンクを辿ってそのルート(route)を見つけられることを確認してください。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に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

## 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)されたインター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。

---

[Go to TOC](#)

# 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。

デフォルトでは、プロジェクトは `@sveltejs/adapter-auto` を使用するように設定されており、プロダクション環境を検出して可能な限り適切な adapter を選択します。もし(まだ)プラットフォームがサポートされていなければ、[custom adapter をインストール](#)するか、[custom adapter を作成](#)する必要があるかもしれません。

新しい環境のサポートを追加することに関しては、[adapter-auto の README](#) をご参照ください。

## サポートされている環境

SvelteKit は、公式にサポートされている adapter を多数提供しています。

以下のプラットフォームには、デフォルトの adapter である `adapter-auto` でデプロイが可能です。

- [Cloudflare Pages](#) — `adapter-cloudflare`
- [Netlify](#) — `adapter-netlify`
- [Vercel](#) — `adapter-vercel`

## Node.js

シンプルな Node サーバーを作成するには、`@sveltejs/adapter-node` パッケージをインストールし、`svelte.config.js` を更新します:

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

これにより、`vite build` は自己完結型の Node アプリを `build` ディレクトリの中に生成します。adapter にはオプションを渡すことができ、例えば出力ディレクトリをカスタマイズできます:

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

export default {
  kit: {
    -   adapter: adapter()
    +   adapter: adapter({ out: 'my-output-directory' })
  }
};
```

## 静的サイト(Static sites)

ほとんどの adapter は、サイト内の [プリレンダリング可能な](#) ページについて、静的な HTML を生成します。アプリ全体がプリレンダリング可能な場合は、`@sveltejs/adapter-static` を使用して 全ての ページ について静的な HTML を生成することができます。完全に静的なサイトは、[GitHub Pages](#) のような静的ホストなど、さまざまなプラットフォームでホストすることができます。

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

`adapter-static` を使用し、[fallback page](#) の設定と [SSR の無効化](#)を行えば、シングルページアプリ(SPA)を生成することができます。

SvelteKit をデプロイする環境に対して `trailingSlash` が適切に設定されているかよく確かめてください。`/a` に対するリクエストを受け取っても `/a.html` をレンダリングしない環境の場合、`/a.html` の代わりに `/a/index.html` を生成するために `trailingSlash: 'always'` を設定する必要があります。

## 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。

## コミュニティが提供する adapter

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。パッケージマネージャーで該当の adapter をインストールした後、`svelte.config.js` を更新してください:

```
/// file: svelte.config.js
- import adapter from '@sveltejs/adapter-auto';
+ import adapter from 'svelte-adapter-[x]';
```

## custom adapter を作成する

似ているプラットフォーム向けの [adapter のソースを探し](#)、それをコピーするところから始めることを推奨します。

Adapter Package は `Adapter` を作成する以下の API を実装する必要があります:

```
// @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` とそのパラメータの型は `types/index.d.ts` にあります。

`adapt` メソッドの中では、`adapter` がすべきことがたくさんあります：

- `build` ディレクトリの掃除
- `builder.writeClient`、`builder.writeServer`、`builder.writePrerendered` を用いて SvelteKit のアウトプットを書き出す
- コードの出力:
  - ``${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
  - プラットフォームからのリクエストをリスンし、必要に応じて標準の `Request` に変換し、`server.respond(request, { getClientAddress })` 関数を呼び出して `Response` を生成し、応答する
  - `server.respond` に渡される `platform` オプションを通して、SvelteKit にプラットフォーム固有の情報を公開する
  - 必要に応じて、対象プラットフォームで動作するように `fetch` をグローバルに shim する。SvelteKit は `node-fetch` を使用できるプラットフォーム向けに `@sveltejs/kit/install-fetch` ヘルパーを提供しています
- 必要に応じて、ターゲットプラットフォームに依存ライブラリをインストールするのを避けるために出力ファイルをバンドルする
- 対象プラットフォームの正しい場所にユーザーの静的ファイルや生成した JS/CSS ファイルを設置する

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

`adapter API` はバージョン 1.0 のリリース前に変更される可能性があります。

# 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所以とても便利です。パラメータにもう1つ括弧を付けることでこれができるようになります: `[[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。

`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]]/foo/+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]]/foo/+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@

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]/
│     └ embed/
```

```

| | | | ` +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);
}

```

# 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専用の `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` 関数呼び出しを追加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、シリアル化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。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、不需要に多くのものをプリロードで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：

```

// @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; code: 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 namespace App {
    interface Error {
        message: string;
        code: string;
    }
}

```

```

/// file: src/hooks.server.js
// @errors: 2322 2571
// @filename: ambient.d.ts
const Sentry: any;

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

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

```

```

/// file: src/hooks.client.js
// @errors: 2322 2571
// @filename: ambient.d.ts
const Sentry: any;

```

```
// @filename: index.js
// cut---
/** @type {import('@sveltejs/kit').HandleClientError} */
export function handleError({ error, event }) {
    // example integration with https://sentry.io/
    Sentry.captureException(error, { event });

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

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。

---

[Go to TOC](#)

# 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。

以下のように、`code` やトラッキング `id` を追加することができます。

## 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 2571
// @filename: ambient.d.ts
const Sentry: any;

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

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

## 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。

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
namespace App {
    interface Error {
        code: string;
        id: string;
    }
}

```

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。

# 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。

## data-sveltekit-prefetch

コードのインポートとページのデータの取得を先取りするためには、`data-sveltekit-prefetch` 属性を使用します。これによって、ナビゲーションをトリガーする `click` イベントを待つのではなく、ユーザーがリンクをホバーしたり(デスクトップの場合)、タッチしたり(モバイルの場合)するとすぐにすべての読み込みを開始します。通常、これによって数百ミリ秒稼ぐことができ、この差は遅延を感じるインターフェイスときびきび動くインターフェイスの違いとなります。

この挙動を全体に適用するには、この属性を親要素 (または `src/app.html` の `<body>`) に追加してください:

```
/// file: src/routes/+layout.svelte
<main data-sveltekit-prefetch>
  <slot />
</main>
```

また、プログラムで `$app/navigation` の `prefetch` を呼び出すこともできます。

## 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</a>
```

…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-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…

```
<a href="path" data-sveltekit-noscroll>Path</a>
```

…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、`"off"` 値を使用します：

```
<div data-sveltekit-prefetch>
  <!-- これらのリンクはプリフェッチされます -->
  <a href="/a">a</a>
  <a href="/b">b</a>
  <a href="/c">c</a>

  <div data-sveltekit-prefetch="off">
    <!-- これらのリンクはプリフェッチされません -->
    <a href="/d">d</a>
    <a href="/e">e</a>
    <a href="/f">f</a>
  </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：

```
<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、ビルドした 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（または `src/service-worker.ts` や `src/service-worker/index.js` など）があれば、Vite でビルドされて自動的に登録されます。Service Worker を独自のロジックで登録する必要がある場合、自動登録を無効にすることができます（例えば、更新をユーザーに促すプロンプトや、定期的な更新の設定、`workbox` の使用、など）。

Service Worker のロケーション と 自動登録の無効化 についてはプロジェクトの設定で変更することができます。

Service Worker の内部では、`$service-worker` モジュール にアクセスすることができます。Vite コンフィグで `define` が設定されている場合、server/client のビルドと同様、service worker にもそれが適用されます。

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'
});
```

# 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可能です。

## 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かにかかわらず）インポートすると…

```
// @errors: 7005
/// file: $lib/server/secrets.js
export const atlantisCoordinates = /* redacted */;

// @errors: 2307 7006
/// 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。

# アセットハンドリング

## キャッシュとインライン化(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インライン化されます。

```
<script>
  import logo from '$lib/assets/logo.png';
</script>

<img alt="The project logo" src={logo} />
```

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、`experimental.useVitePreprocess` オプションが役立つでしょう：

```
// svelte.config.js
export default {
  vitePlugin: {
    experimental: {
      useVitePreprocess: true
    }
  }
};
```

## 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。

# Packaging

`svelte-package` は現時点では `experimental` です。将来のリリースで後方互換性のない変更が行われる可能性があります。

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 アプリと全く同じ構造を持ちますが、`src/lib` も公開される点が異なります。`src/routes` はライブラリに付随するドキュメントやデモサイトにもできますし、開発中に使用できるサンドボックスにもできます。

`@sveltejs/package` の `svelte-package` コマンドを実行すると、`src/lib` のコンテンツを使用して、以下を含む `package` ディレクトリ (変更可能) を生成します:

- カスタムで `include / exclude` オプションを 設定 しない限り、`src/lib` にある全てのファイルが含まれます。Svelte コンポーネントはプリプロセスされ、TypeScript ファイルは JavaScript にトランスペイロールされます。
- Svelte、JavaScript、TypeScript ファイルのために生成される型定義 (`d.ts` ファイル)。これには `typescript >= 4.0.0` をインストールする必要があります。型定義は実装の隣に置かれ、手書きの `d.ts` ファイルはそのままコピーされます。生成を無効化 することもできますが、あまりおすすめしません。
- プロジェクトのルート(root)からコピーされた `package.json` から、`"scripts"`、`"publishConfig.directory"`、`"publishConfig.linkDirectory"` フィールドを取り除いたもの。`"dependencies"` フィールドは含まれているため、ドキュメントやデモサイトにのみ必要なパッケージは `"devDependencies"` に追加してください。`"type": "module"` と `"exports"` フィールドは、オリジナルのファイルで定義されていない場合に追加されます。

`"exports"` フィールドにはパッケージのエントリーポイントが含まれます。デフォルトでは、アンダースコアで始まるファイル(またはアンダースコアで始まるディレクトリにあるファイル)を除いて、`src/lib` にある全てのファイルをエントリーポイントとして扱いますが、この動作は 設定可能 です。もし `src/lib/index.js` や `src/lib/index.svelte` ファイルがある場合は、それがパッケージルートとして扱われます。

例えば、`src/lib/Foo.svelte` コンポーネントと、それを再エクスポートした `src/lib/index.js` モジュールがあった場合、ライブラリの利用者は次のどちらかを行なうことができます。

```
// @filename: ambient.d.ts
declare module 'your-library';

// @filename: index.js
// cut---
import { Foo } from 'your-library';
```

```
// @filename: ambient.d.ts
declare module 'your-library/Foo.svelte';
```

```
// @filename: index.js
// cut---
import Foo from 'your-library/Foo.svelte';
```

SvelteKit プロジェクトで利用することだけを意図している場合を除いて、`$app` などの [SvelteKit 固有のモジュール](#) をあなたのパッケージで使用するのは避けてください。例えば、`import { browser } from '$app/environment'` を使用するよりも、`import.meta.env.SSR` を使用して全ての Vite ベースのプロジェクトで使用できるようになりますか、もっと良いのは [Node conditional exports](#) を使用して全てのバンドラーで動作するようにすることです。また、`$app/stores` や `$app/navigation` などに直接依存せずに、現在の URL やナビゲーションアクション (navigation action)などをプロパティとして渡したいケースもあるでしょう。より一般的な方法でアプリを書くことで、テストや UI デモなどのためのツールのセットアップも簡単になります。

## Options

`svelte-package` は以下のオプションを受け付けます:

- `-w / --watch` — `src/lib` の中にあるファイルを監視し、パッケージを再ビルトします

## Publishing

生成されたパッケージをパブリッシュするには:

```
npm publish ./package
```

上記の `./package` は生成されるディレクトリ名を参照しています。カスタムで `package.dir` を設定している場合は、適宜変更してください。

## 注意事項

相対ファイルのインポートはすべて、Node の ESM アルゴリズムに従って完全に指定する必要があります。つまり、`src/lib/something/index.js` ファイルを `import { something } from './something'` のようにインポートすることはできません。代わりに、`import { something } from './something/index.js'` というようにインポートする必要があります。TypeScript を使用している場合は、`.ts` ファイルと同じ方法でインポートする必要がありますが、ファイルの末尾は `.ts` ではなく `.js` を使用します（これは我々の管理下ではなく、TypeScript チームが決定したことです）。`tsconfig.json` または `jsconfig.json` で `"moduleResolution": "NodeNext"` と設定することで、この問題を解決できます。

比較的、これは実験的な機能であり、まだ完全に実装されていません。Svelte ファイル(プリプロセス済)と TypeScript ファイル(JavaScriptにトランスペイロ済)を除き、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。

## 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動作をシミュレートするために、SvelteKit は各ナビゲーション後に `<body>` 要素にフォーカスします。この動作をカスタマイズしたい場合は、`afterNavigate` hook を使用してカスタムのフォーカスマネジメントロジックを実装することができます:

```
/// <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。もし該当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 \(Quick Reference\)](#)

---

[Go to TOC](#)

# 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き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 `svelte.config.js` ファイルにあります。全ての値はオプションです。オプションのデフォルトと完全なリストは[こちら](#)です:

```
/// file: svelte.config.js
/** @type {import('@sveltejs/kit').Config} */
const config = {
    // options passed to svelte.compile (https://svelte.dev/docs#compile-time-svelte-compile)
    compilerOptions: {},  

  

    // an array of file extensions that should be treated as Svelte components
    extensions: ['.svelte'],  

  

    kit: {
        adapter: undefined,
        alias: {},
        appDir: '_app',
        csp: {
            mode: 'auto',
            directives: {
                'default-src': undefined
                // ...
            }
        },
        csrf: {
            checkOrigin: true
        },
        env: {
            dir: process.cwd(),
            publicPrefix: 'PUBLIC_'
        },
        files: {
            assets: 'static',
            hooks: {
                client: 'src/hooks.client',
                server: 'src/hooks.server'
            },
            lib: 'src/lib',
            params: 'src/params',
            routes: 'src/routes',
            serviceWorker: 'src/service-worker',
            appTemplate: 'src/app.html',
            errorTemplate: 'src/error.html'
        },
        inlineStyleThreshold: 0,
        moduleExtensions: ['.js', '.ts'],
        outDir: '.svelte-kit',
        paths: {
            assets: '',
            base: ''
        },
        prerender: {
            concurrency: 1,
            crawl: true,
            enabled: true,
            entries: ['*'],
        }
    }
}
```

```

        handleHttpError: 'fail',
        handleMissingId: 'fail',
        origin: 'http://sveltekit-prerender'
    },
    serviceWorker: {
        register: true,
        files: (filepath) => !/\.DS_Store/.test(filepath)
    },
    version: {
        name: Date.now().toString(),
        pollInterval: 0
    }
},
// options passed to @sveltejs/package
package: {
    source: 'value of kit.files.lib, if available, else src/lib',
    dir: 'package',
    emitTypes: true,
    // excludes all .d.ts and files starting with _ as the name
    exports: (filepath) => !/^_|^_|\.\d\.ts$/.test(filepath),
    files: () => true
},
// options passed to svelte.preprocess (https://svelte.dev/docs#compile-time-svelte-preprocess)
preprocess: null
};

export default config;

```

## adapter

`vite build` の実行中に実行され、異なるプラットフォーム向けにアウトプットがどのように変換されるかを決定します。Adapters をご参照ください。

## alias

`import` 文の値を置き換えるのに使用される 0 個以上のエイリアスを含むオブジェクトです。これらのエイリアスは自動的に Vite と TypeScript に渡されます。

```

/// file: svelte.config.js
/** @type {import('@sveltejs/kit').Config} */
const config = {
    kit: {
        alias: {
            // this will match a file
            'my-file': 'path/to/my-file.js',

            // this will match a directory and its contents
            // (`my-directory/x` resolves to `path/to/my-directory/x`)
            'my-directory': 'path/to/my-directory',

            // an alias ending /* will only match
            // the contents of a directory, not the directory itself
            'my-directory/*': 'path/to/my-directory/*'
        }
    }
};

```

```

    }
};


```

ビルトインの `$lib` エイリアスはパッケージングで使用されるため、`config.kit.files.lib` でコントロールされています。

SvelteKit が必要なエイリアス設定を `jsconfig.json` または `tsconfig.json` に自動的に生成するためには、`npm run dev` を実行する必要があります。

## appDir

ビルドされた JS と CSS(およびインポートされたアセット)が提供される `paths.assets` からの相対ディレクトリ(ファイル名にはコンテンツベースのハッシュが含まれており、つまり、無期限にキャッシュすることができます)。先頭または末尾が `/` であってはいけません。

## csp

以下の値のうち、0 個以上を含むオブジェクトです:

- `mode` — 'hash'、'nonce'、'auto' のいずれか
- `directives` — `[directive]: value` ペアのオブジェクト
- `reportOnly` — CSP report-only モードのための `[directive]: value` ペアのオブジェクト

[Content Security Policy](#) の設定です。CSP は、リソースの読み込み元を制限することにより、クロスサイトスクリプティング (XSS) 攻撃からユーザーを守るのに役立ちます。例えば、このような設定では...

```

/// file: svelte.config.js
/** @type {import('@sveltejs/kit').Config} */
const config = {
  kit: {
    csp: {
      directives: {
        'script-src': ['self']
      },
      reportOnly: {
        'script-src': ['self']
      }
    }
  }
};

export default config;

```

...外部サイトからのスクリプト読み込みを防止します。SvelteKit は、生成されるインラインスタイルとスクリプトに対して、指定されたディレクティブを `nonce` か `hash` (`mode` の設定による) で補強します。

`script` と `link` 向けに `nonce` を `app.html` に手動で含めるには、`%sveltekit.nonce%` プレースホルダーをお使いいただけます (例えば `<script nonce="%sveltekit.nonce%">`)。

ページがプリレンダリングされる場合、CSP ヘッダーは `<meta http-equiv>` タグ経由で追加されます (この場合、`frame-ancestors`、`report-uri`、`sandbox` ディレクティブは無視されることにご注意ください)。

`mode` が `'auto'` の場合、SvelteKit は動的にレンダリングされたページには `nonce` を、プリレンダリングされたページには `hash` を使用します。プリレンダリングされたページで `nonce` を使用するのは安全でないため、禁止されています。

ほとんどの [Svelte transitions](#) は、インラインの `<style>` 要素を作成することで動作することにご注意ください。これらをアプリで使用する場合、`style-src` ディレクティブを指定しないようにするか、`unsafe-inline` を追加する必要があります。

## csrf

クロスサイト・リクエスト・フォージェリ(cross-site request forgery) 攻撃に対する防御です:

- `checkOrigin` — `true` の場合、SvelteKit は `POST` による form 送信を受け取ったとき、受け取った `origin` ヘッダーをチェックし、それがサーバーの `origin` と一致するか検証します

別の `origin` からあなたのアプリに対して `POST` による form 送信ができるようにするには、このオプションを無効にする必要があります。ご注意ください!

## env

環境変数の設定です:

- `dir` — `.env` ファイルを検索するディレクトリです。
- `publicPrefix` — クライアントサイドのコードに公開しても安全であることを示す接頭辞です。`$env/static/public` と `$env/dynamic/public` をご参照ください。Vite の環境変数のハンドリングを使用する場合は、Vite の `envPrefix` を別途設定する必要があることにご注意ください。もっとも、通常はこの機能を使用する必要はありません。

## files

以下の `string` 値のうち、0 個以上を含むオブジェクトです:

- `assets` — `favicon.ico` or `manifest.json` のような、何も処理する必要もなく、安定した URL を持つべき静的ファイルを配置する場所
- `hooks` — クライアントとサーバーの hooks のロケーション([Hooks](#) をご参照ください)
- `lib` — コードベース全体から `$lib` でアクセスできる、アプリの内部ライブラリ
- `params` — `parameter matchers` を含むディレクトリ
- `routes` — アプリの構造を定義するファイル([ルーティング](#) をご参照ください)

- `serviceWorker` — Service Worker のエントリーポイントのロケーション([Service workers](#) をご参照ください)
- `template` — HTML レスポンス用テンプレートのロケーション

## inlineStyleThreshold

CSS を HTML の先頭の `<style>` ブロック内にインライン化するかどうか。このオプションでは、インライン化する CSS ファイルの最大長を数値で指定します。ページに必要な CSS ファイルで、このオプションの値より小さいものはマージされ、`<style>` ブロックにインライン化されます。

この結果、最初のリクエストが少なくなり、[First Contentful Paint](#) スコアを改善することができます。しかし、HTML 出力が大きくなり、ブラウザキャッシュの効果が低下します。慎重に使用してください。

## moduleExtensions

SvelteKit がモジュールとして取り扱うファイル拡張子の配列です。`config.extensions` と `config.kit.moduleExtensions` のいずれにもマッチしない拡張子のファイルはルーター(router) から無視されます。

## outDir

SvelteKit が `dev` と `build` のときにファイルを書き込むディレクトリです。このディレクトリをバージョン管理から除外する必要があります。

## package

[パッケージ作成](#) に関するオプションです。

- `source` - library directory
- `dir` - 出力ディレクトリ
- `emitTypes` - デフォルトでは、`svelte-package` は自動的にパッケージの型を `.d.ts` ファイル形式で生成します。型の生成は設定で変更できますが、常に型を生成することがエコシステムの品質にとってベストであると私たちは信じています。`false` に設定するときは、十分な理由があることを確認してください(例えば、代わりに手書きの型定義を提供したい場合など)
- `exports` - `(filepath: string) => boolean` という型を持つ関数。`true` の場合、ファイルパスが `package.json` の `exports` フィールドに含まれるようになります。`package.json` のソースにある既存の値は、オリジナルの `exports` フィールドの値が優先されてマージされます
- `files` - `(filepath: string) => boolean` という型を持つ関数。`true` の場合、ファイルは処理され、`dir` で指定された最終的な出力フォルダにコピーされます

高度な `filepath` マッチングには、`exports` と `files` オプションを globbing ライブラリと組み合わせて使用することができます:

```
// @filename: ambient.d.ts
declare module 'micromatch';

/// file: svelte.config.js
```

```
// @filename: index.js
// cut---
import mm from 'micromatch';

/** @type {import('@sveltejs/kit').Config} */
const config = {
    package: {
        exports: (filepath) => {
            if (filepath.endsWith('.d.ts')) return false;
            return mm.isMatch(filepath, ['!**/_*', '!**/internal/**']);
        },
        files: mm.matcher('!**/build.*')
    }
};

export default config;
```

## paths

以下の `string` 値のうち、0 個以上を含むオブジェクトです:

- `assets` — アプリのファイルが提供される絶対パス。これは、何らかのストレージバケットからファイルを提供する場合に便利です
- `base` — ルート相対パス。空文字以外を指定する場合、先頭は `/` で始めなければならず、末尾は `/` で終わってはいけません (例 `/base-path`)。アプリがどこから提供されるかを指定することで、アプリを非ルートパスで動作させることができます。ルート相対(root-relative)なリンクには先頭に `base` の値を追加しなければなりません。そうしないと、リンクが `base` ではなくドメインのルート(root)を指してしまいます (これはブラウザの動作によるものです)。これに対しては、`base from $app/paths` をこのように使用することができます: `<a href="{base}/your-page">Link</a>`。もし、これを頻繁に書くようであれば、再利用可能なコンポーネントに抽出するのも良いでしょう。

## prerender

プリレンダリング(Prerendering) をご参考ください。以下のうち、0 個以上を含むオブジェクトです:

- `concurrency` — 同時にいくつのページをプリレンダリングできるか。JS はシングルスレッドですが、プリレンダリングのパフォーマンスがネットワークに縛られている場合(例えば、リモートの CMS からコンテンツをロードしている場合)、ネットワークの応答を待っている間に他のタスクを処理することで高速化することができます
- `crawl` — SvelteKit がシードページからリンクをたどってプリレンダリングするページを見つけるかどうかを決定します
- `enabled` — `false` に設定すると、プリレンダリングを完全に無効化できます
- `entries` — プリレンダリングするページ、またはクロールを開始するページ(`crawl: true` の場合)の配列。\* 文字列には、全ての動的ではないルート(routes)(例えば `[parameters]` を持たないページです。なぜなら SvelteKit はその `parameters` がどんな値を持つべきかわからないからです) が含まれます
- `handleHttpError`
  - `'fail'` — (デフォルト) リンクをたどったときにルーティングエラーが発生した場合、ビルドを失敗させます
  - `'ignore'` - 失敗を無視して継続させます

- `'warn'` — 繼続しますが、警告(warning)を出力します
- `(details) => void` — `status`、`path`、`referrer`、`referenceType`、`message` プロパティを持つ `details` オブジェクトを引数に取るカスタムのエラーハンドラです。この関数から `throw` されると、ビルドが失敗します

```
/** @type {import('@sveltejs/kit').Config} */
const config = {
  kit: {
    prerender: {
      handleHttpError: ({ path, referrer, message }) => {
        // ignore deliberate link to shiny 404 page
        if (path === '/not-found' && referrer === '/blog/how-we-built-our-404-page') {
          return;
        }

        // otherwise fail the build
        throw new Error(message);
      }
    }
  };
};
```

- `handleMissingId`
  - `'fail'` — (default) プリレンダリングページから他のプリレンダリングページへの、`#` フラグメントを使用しているリンクが、`id` に一致しない場合、ビルドを失敗させます
  - `'ignore'` - 失敗を無視して継続させます
  - `'warn'` — 繼続しますが、警告(warning)を出力します
  - `(details) => void` — `path`、`id`、`referrers`、`message` プロパティを持つ `details` オブジェクトを引数に取るカスタムのエラーハンドラです。この関数から `throw` されると、ビルドが失敗します
- `origin` — プリレンダリング時の `url.origin` の値です。レンダリングされたコンテンツに含まれている場合に有用です。

## serviceWorker

以下の値のうち、0 個以上を含むオブジェクトです:

- `register` - `false` を設定した場合、service worker の自動登録を無効にします。
- `files` - `(filepath: string) => boolean` という型を持つ関数。`true` の場合、与えられたファイルが `$service-worker.files` で利用可能になります。それ以外の場合は除外されます。

## version

以下の値のうち、0 個以上を含むオブジェクトです:

- `name` - 現在のアプリのバージョン文字列
- `pollInterval` - バージョンの変更をポーリングするインターバル(ミリ秒)

アプリが使用されているときにアプリの新しいバージョンをデプロイするとクライアントサイドのナビゲーションにバグが発生することがあります。次に開くページのコードがすでにロードされている場合、古いコンテンツがある可能性があります。そうでなくとも、アプリのルートマニフェストがもう存在しない JavaScript ファイルを指している可能性があります。SvelteKit は、ここで指定された `name` (デフォルトではビルトのタイムスタンプ) を使用して新しいバージョンがデプロイされたことを検知すると、従来のフルページナビゲーションにフォールバックすることにより、この問題を解決しています。

`pollInterval` を 0 以外の値に設定した場合、SvelteKit はバックグラウンドで新しいバージョンをポーリングし、それを検知すると `updated` ストアの値を `true` にします。

# 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、型や `tsconfig.json` などのプロジェクト用に生成されるファイルを作成します。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。

## EXPORTS

---

[Go to TOC](#)

# Types

## 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
/** @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"
  }
}
```

# 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
<code>routes/about/index.svelte</code>	<code>routes/about/+page.svelte</code>

Old	New
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` からのインポートに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。.

## 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` is now `data-sveltekit-prefetch`
- `sapper:noscroll` is now `data-sveltekit-noscroll`

## 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、[hook](#) としてこれを追加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 { prerendering } 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 }) {
    const response = await resolve(event);

    if (prerendering && response.headers.get('content-type') === 'text/html') {
        return new Response(minify(await response.text()), minification_options), {
            status: response.status,
            headers: response.headers
        });
    }

    return response;
}
```

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。

# 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。

## インテグレーション

[svelte-preprocess](#) は Svelte テンプレート内のコードを自動的に変換し、TypeScript、PostCSS、scss/sass、Less、その他多くのテクノロジー(SvelteKitがサポートしていない CoffeeScriptは除く)のサポートを提供します。設定の最初のステップは [svelte.config.js](#) に [svelte-preprocess](#) を追加することです。これは、TypeScriptを使用している場合はテンプレートで提供されますが、JavaScriptのユーザーは追加する必要があります。その後に、[npm install -D sass](#) や [npm install -D less](#) のように、対応するライブラリをインストールするだけで良い場合が多いです。詳しくは [svelte-preprocess](#) のドキュメントをご参照ください。

[Svelte Adders](#) は、Tailwind、PostCSS、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、ツールのフルの一覧については、[sveltesociety.dev](#) をご覧ください。

また、SvelteKit FAQ にも [インテグレーションの章](#) がありますので、何か問題が発生した場合はそちらも役立つでしょう。

## 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。

---

[Go to TOC](#)

# 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。

## 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。

## CSR と SPA

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。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、ログインの背後にある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。

## 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能够性があれば、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。

## 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 の `adapter-static` はこれを行います。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。

## 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。

## 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cept 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。

# 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。

---

[Go to TOC](#)

# SvelteKit はまだ1.0ではないようです。使うべきですか？

SvelteKit は現在のところ、リリース候補の段階です(release candidate phase)。破壊的な変更は最小限になると考へており、開発はバグの修正にフォーカスされています。

---

[Go to TOC](#)

# SvelteKit で HMR を使うにはどうすればよいか？

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。

# adapter の使用に問題があります。

`package.json` で指定されている adapter のバージョンが `"next"` になっていることを確認してください。

---

[Go to TOC](#)

# How do I include details from package.json in my application?

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？

Vite の SSR サポートは Vite 2.7 以降かなり安定しています。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。

Vite の場合、ライブラリは ESM バージョンが配布されているともっともうまく動作するので、ライブラリの作者にそれを提案すると良いでしょう。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ます、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ます、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 コンポーネントは完全に ESM として配布される必要があり、また、エントリーポイントを定義する `svelte` フィールドがなければなりません。

外部の Svelte コンポーネントの依存関係(dependencies)が ESM バージョンを提供していることを確認することが推奨されます。しかし、CommonJS の依存関係(dependencies)を扱うため、[vite-plugin-svelte は外部の Svelte コンポーネントの CJS の依存関係\(dependencies\)を探し](#)、Vite に対し、自動的に Vite の `optimizeDeps.include` にそれらを追加して事前バンドル(pre-bundle)するよう依頼します。Vite はそれらを ESM に変換するのに `esbuild` を使用します。このアプローチの副作用は初期ページのロードに時間がかかることです。もしこれが気になるなら、`svelte.config.js` で `experimental.prebundleSvelteLibraries: true` を設定してみてください。このオプションは `experimental` であることにご注意ください。

それでもまだ問題が解消されない場合は、[SvelteKit ユーザーに影響する既知の Vite の issue 一覧](#) をチェックし、[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。

---

[Go to TOC](#)

# 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。

## 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。

## 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');
});
```

# Yarn を使用するにはどうすれば良いですか？

## 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、というのがあります。しかし、`enableGlobalCache` の設定を true にす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: 2022-11-28
- 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>