SVELTEKIT • ADVANCED

Advanced routing

ON THIS PAGE

Rest parameters

If the number of route segments is unknown, you can use rest syntax — for example you might implement GitHub's file viewer like so...

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

...in which case a request for `/sveltejs/kit/tree/main/documentation/docs/04-advanced-routing.md` would result in the following parameters being available to the page:

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

> `src/routes/a/[...rest]/z/+page.svelte` will match `/a/z` (i.e. there's no parameter at all) as well as `/a/b/z` and `/a/b/c/z` and so on. Make sure you check that the value of the rest parameter is valid, for example using a matcher.

404 pages

Rest parameters also allow you to render custom 404s. Given these routes...

```
├ marx-brothers/
│ ├ chico/
│ ├ harpo/
│ ├ groucho/
│ └ +error.svelte
└ +error.svelte
```

...the `marx-brothers/+error.svelte` file will *not* be rendered if you visit `/marx-brothers/karl`, because no route was matched. If you want to render the nested error page, you should create a route that matches any `/marx-brothers/*` request, and return a 404 from it:

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

src/routes/marx-brothers/[...path]/+page.ts

JS TS

```ts
import { error } from '@sveltejs/kit';
import type { PageLoad } from './$types';

export const load: PageLoad = (event) => {
  error(404, 'Not Found');
};
```

If you don’t handle 404 cases, they will appear in `handleError`

Optional parameters

A route like `[lang]/home` contains a parameter named `lang` which is required. Sometimes it’s beneficial to make these parameters optional, so that in this example both `home` and bracket pair: `[[lang]]/home`

Note that an optional route parameter cannot follow a rest parameter (`[...rest]/[[optional]]`), since parameters are matched 'greedily' and the optional parameter would always be unused.

Matching

A route like `src/routes/fruits/[page]` would match `/fruits/apple` , but it would also match `/fruits/rocketship` . We don't want that. You can ensure that route parameters are well-formed by adding a *matcher* — which takes the parameter string (`"apple"` or `"rocketship"`) and returns `true` if it is valid — to your `params` directory...

src/params/fruit.ts

JS TS

```ts
import type { ParamMatcher } from '@sveltejs/kit';

export const match = ((param: string): param is ('apple' | 'orange') => {
  return param === 'apple' || param === 'orange';
}) satisfies ParamMatcher;
```

...and augmenting your routes:

```
src/routes/fruits/[page=fruit]
```

If the pathname doesn't match, SvelteKit will try to match other routes (using the sort order specified below), before eventually returning a 404.

Each module in the `params` directory corresponds to a matcher, with the exception of `*.test.js` and `*.spec.js` files which may be used to unit test your matchers.

> Matchers run both on the server and in the browser.

Docs

It’s possible for multiple routes to match a given path. For example each of these routes would match `/foo-abc`:

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

SvelteKit needs to know which route is being requested. To do so, it sorts them according to the following rules...

- More specific routes are higher priority (e.g. a route with no parameters is more specific than a route with one dynamic parameter, and so on)
- Parameters with matchers (`[name=type]`) are higher priority than those without (`[name]`)
- `[[optional]]` and `[...rest]` parameters are ignored unless they are the final part of the route, in which case they are treated with lowest priority. In other words `x/[[y]]/z` is treated equivalently to `x/z` for the purposes of sorting
- Ties are resolved alphabetically

...resulting in this ordering, meaning that `/foo-abc` will invoke `src/routes/foo-abc/+page.svelte`, and `/foo-def` will invoke `src/routes/foo-[c]/+page.svelte` rather than less specific routes:

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

`|` on Windows. The `#` and `%` characters have special meaning in URLs, and the `[ ] ( )` characters have special meaning to SvelteKit, so these also can't be used directly as part of your route.

To use these characters in your routes, you can use hexadecimal escape sequences, which have the format `[x+nn]` where `nn` is a hexadecimal character code:

- `\` — `[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]`

For example, to create a `/smileys/:-)` route, you would create a `src/routes/smileys/[x+3a]-[x+29]/+page.svelte` file.

You can determine the hexadecimal code for a character with JavaScript:

unencoded character directly, but if — for some reason — you can't have a filename with an emoji in it, for example, then you can use the escaped characters. In other words, these are equivalent:

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

The format for a Unicode escape sequence is `[u+nnnn]` where `nnnn` is a valid value between `0000` and `10ffff`. (Unlike JavaScript string escaping, there's no need to use surrogate pairs to represent code points above `ffff`.) To learn more about Unicode encodings, consult Programming with Unicode.

> Since TypeScript struggles with directories with a leading `.` character, you may find it useful to encode these characters when creating e.g. `.well-known` routes: `src/routes/[x+2e]well-known/...`

Advanced layouts

By default, the *layout hierarchy* mirrors the *route hierarchy*. In some cases, that might not be what you want.

(group)

Perhaps you have some routes that are 'app' routes that should have one layout (e.g. `/dashboard` or `/item`), and others that are 'marketing' routes that should have a different layout (`/about` or `/testimonials`). We can group these routes with a directory whose name is wrapped in parentheses — unlike normal directories, `(app)` and `(marketing)` do not affect the URL pathname of the routes inside them:

```
src/routes/
│ (app)/
│ ├ about/
│ ├ testimonials/
│ └ +layout.svelte
├ admin/
└ +layout.svelte
```

You can also put a `+page` directly inside a `(group)`, for example if `/` should be an `(app)` or a `(marketing)` page.

Breaking out of layouts

The root layout applies to every page of your app — if omitted, it defaults to `{@render children()}`. If you want some pages to have a different layout hierarchy than the rest, then you can put your entire app inside one or more groups *except* the routes that should not inherit the common layouts.

In the example above, the `/admin` route does not inherit either the `(app)` or `(marketing)` layouts.

+page@

Pages can break out of the current layout hierarchy on a route-by-route basis. Suppose we have an `/item/[id]/embed` route inside the `(app)` group from the previous example:

```
src/routes/
├ (app)/
│ ├ item/
│ │ ├ [id]/
│ │ │ ├ embed/
│ │ │ │ └ +page.svelte
│ │ │ └ +layout.svelte
│ │ └ +layout.svelte
│ └ +layout.svelte
└ +layout.svelte
```

`[id]` layout. We can reset to one of those layouts by appending `@` followed by the segment name — or, for the root layout, the empty string. In this example, we can choose from the following options:

- `+page@[id].svelte` - inherits from `src/routes/(app)/item/[id]/+layout.svelte`
- `+page@item.svelte` - inherits from `src/routes/(app)/item/+layout.svelte`
- `+page@(app).svelte` - inherits from `src/routes/(app)/+layout.svelte`
- `+page@.svelte` - inherits from `src/routes/+layout.svelte`

```
src/routes/
├ (app)/
│ ├ item/
│ │ ├ [id]/
│ │ │ ├ embed/
│ │ │ │ └ +page@(app).svelte
│ │ │ └ +layout.svelte
│ │ └ +layout.svelte
│ └ +layout.svelte
└ +layout.svelte
```

+layout@

Like pages, layouts can *themselves* break out of their parent layout hierarchy, using the same technique. For example, a `+layout@.svelte` component would reset the hierarchy for all its child routes.

```
src/routes/
├ (app)/
│ ├ item/
│ │ ├ [id]/
│ │ │ ├ embed/
│ │ │ │ └ +page.svelte  // uses (app)/item/[id]/+layout.svelte
│ │ │ ├ +layout.svelte  // inherits from (app)/item/+layout@.svelte
│ │ │ └ +page.svelte    // uses (app)/item/+layout@.svelte
│ │ └ +layout@.svelte   // inherits from root layout, skipping (app)/+layout.svelte
```

Not all use cases are suited for layout grouping, nor should you feel compelled to use them. It might be that your use case would result in complex `(group)` nesting, or that you don't want to introduce a `(group)` for a single outlier. It's perfectly fine to use other means such as composition (reusable `load` functions or Svelte components) or if-statements to achieve what you want. The following example shows a layout that rewinds to the root layout and reuses components and functions that other layouts can also use:

src/routes/nested/route/+layout@.svelte

```svelte
<script>
  import ReusableLayout from '$lib/ReusableLayout.svelte';
  let { data, children } = $props();
</script>

<ReusableLayout {data}>
  {@render children()}
</ReusableLayout>
```

src/routes/nested/route/+layout.ts

JS TS

```ts
import { reusableLoad } from '$lib/reusable-load-function';
import type { PageLoad } from './$types';

export const load: PageLoad = (event) => {
  // Add additional logic here, if needed
  return reusableLoad(event);
};
```

Further reading

- Tutorial: Advanced Routing