SVELTEKIT • BUILD AND DEPLOY

Node servers

ON THIS PAGE

To generate a standalone Node server, use `adapter-node`.

Usage

Install with `npm i -D @sveltejs/adapter-node`, then add the adapter to your `svelte.config.js`:

svelte.config.js

```
import adapter from '@sveltejs/adapter-node';

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

Deploying

First, build your app with `npm run build`. This will create the production server in the output directory specified in the adapter options, defaulting to `build`.

You will need the output directory, the project's `package.json`, and the production dependencies in `node_modules` to run the application. Production dependencies can be generated by copying the `package.json` and `package-lock.json` and then running `npm ci`

Development dependencies will be bundled into your app using Rollup. To control whether a given package is bundled or externalised, place it in `devDependencies` or `dependencies` respectively in your `package.json`.

Compressing responses

You will typically want to compress responses coming from the server. If you are already deploying your server behind a reverse proxy for SSL or load balancing, it typically results in better performance to also handle compression at that layer since Node.js is single-threaded.

However, if you're building a custom server and do want to add a compression middleware there, note that we would recommend using `@polka/compression` since SvelteKit streams responses and the more popular `compression` package does not support streaming and may cause errors when used.

Environment variables

In `dev` and `preview`, SvelteKit will read environment variables from your `.env` file (or `.env.local`, or `.env.[mode]`, as determined by Vite.)

In production, `.env` files are *not* automatically loaded. To do so, install `dotenv` in your project...

```
npm install dotenv
```

...and invoke it before running the built app:

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

If you use Node.js v20.6+, you can use the `--env-file` flag instead:

PORT, HOST and SOCKET_PATH

By default, the server will accept connections on `0.0.0.0` using port 3000. These can be customised with the `PORT` and `HOST` environment variables:

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

Alternatively, the server can be configured to accept connections on a specified socket path. When this is done using the `SOCKET_PATH` environment variable, the `HOST` and `PORT` environment variables will be disregarded.

```
SOCKET_PATH=/tmp/socket node build
```

ORIGIN, PROTOCOL_HEADER, HOST_HEADER, and PORT_HEADER

HTTP doesn't give SvelteKit a reliable way to know the URL that is currently being requested. The simplest way to tell SvelteKit where the app is being served is to set the `ORIGIN` environment variable:

```
ORIGIN=https://my.site node build

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

With this, a request for the `/stuff` pathname will correctly resolve to `https://my.site/stuff`. Alternatively, you can specify headers that tell SvelteKit about the request protocol and host, from which it can construct the origin URL:

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

> original protocol and host if you're using a reverse proxy (think load balancers and CDNs). You should only set these variables if your server is behind a trusted reverse proxy; otherwise, it'd be possible for clients to spoof these headers.
>
> If you're hosting your proxy on a non-standard port and your reverse proxy supports `x-forwarded-port`, you can also set `PORT_HEADER=x-forwarded-port`.

If `adapter-node` can't correctly determine the URL of your deployment, you may experience this error when using form actions:

> Cross-site POST form submissions are forbidden

ADDRESS_HEADER and XFF_DEPTH

The RequestEvent object passed to hooks and endpoints includes an `event.getClientAddress()` function that returns the client's IP address. By default this is the connecting `remoteAddress`. If your server is behind one or more proxies (such as a load balancer), this value will contain the innermost proxy's IP address rather than the client's, so we need to specify an `ADDRESS_HEADER` to read the address from:

```
ADDRESS_HEADER=True-Client-IP node build
```

> Headers can easily be spoofed. As with `PROTOCOL_HEADER` and `HOST_HEADER`, you should know what you're doing before setting these.

If the `ADDRESS_HEADER` is `X-Forwarded-For`, the header value will contain a comma-separated list of IP addresses. The `XFF_DEPTH` environment variable should specify how many trusted proxies sit in front of your server. E.g. if there are three trusted proxies, proxy 3 will forward the addresses of the original connection and the first two proxies:

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

spoofing:

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

We instead read from the *right*, accounting for the number of trusted proxies. In this case, we would use `XFF_DEPTH=3`.

> If you need to read the left-most address instead (and don't care about spoofing) — for example, to offer a geolocation service, where it's more important for the IP address to be *real* than *trusted*, you can do so by inspecting the `x-forwarded-for` header within your app.

BODY_SIZE_LIMIT

The maximum request body size to accept in bytes including while streaming. The body size can also be specified with a unit suffix for kilobytes (`K`), megabytes (`M`), or gigabytes (`G`). For example, `512K` or `1M`. Defaults to 512kb. You can disable this option with a value of `Infinity` (0 in older versions of the adapter) and implement a custom check in `handle` if you need something more advanced.

SHUTDOWN_TIMEOUT

The number of seconds to wait before forcefully closing any remaining connections after receiving a `SIGTERM` or `SIGINT` signal. Defaults to `30`. Internally the adapter calls `closeAllConnections`. See Graceful shutdown for more details.

IDLE_TIMEOUT

When using systemd socket activation, `IDLE_TIMEOUT` specifies the number of seconds after which the app is automatically put to sleep when receiving no requests. If not set, the app runs continuously. See Socket activation for more details.

The adapter can be configured with various options:

svelte.config.js

```
import adapter from '@sveltejs/adapter-node';

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

out

The directory to build the server to. It defaults to `build` — i.e. `node build` would start the server locally after it has been created.

precompress

Enables precompressing using gzip and brotli for assets and prerendered pages. It defaults to `true`.

envPrefix

If you need to change the name of the environment variables used to configure the deployment (for example, to deconflict with environment variables you don't control), you can specify a prefix:

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

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

Graceful shutdown

By default `adapter-node` gracefully shuts down the HTTP server when a `SIGTERM` or `SIGINT` signal is received. It will:

1. reject new requests (`server.close`)
2. wait for requests that have already been made but not received a response yet to finish and close connections once they become idle (`server.closeIdleConnections`)
3. and finally, close any remaining connections that are still active after `SHUTDOWN_TIMEOUT` seconds. (`server.closeAllConnections`)

> If you want to customize this behaviour you can use a custom server.

You can listen to the `sveltekit:shutdown` event which is emitted after the HTTP server has closed all connections. Unlike Node's `exit` event, the `sveltekit:shutdown` event supports asynchronous operations and is always emitted when all connections are closed even if the server has dangling work such as open database connections.

```
process.on('sveltekit:shutdown', async (reason) => {
  await jobs.stop();
  await db.close();
});
```

The parameter `reason` has one of the following values:

> `SIGINT` - shutdown was triggered by a `SIGINT` signal
>
> `SIGTERM` - shutdown was triggered by a `SIGTERM` signal

Most Linux operating systems today use a modern process manager called systemd to start the server and run and manage services. You can configure your server to allocate a socket and start and scale your app on demand. This is called socket activation. In this case, the OS will pass two environment variables to your app — `LISTEN_PID` and `LISTEN_FDS`. The adapter will then listen on file descriptor 3 which refers to a systemd socket unit that you will have to create.

> You can still use `envPrefix` with systemd socket activation. `LISTEN_PID` and `LISTEN_FDS` are always read without a prefix.

To take advantage of socket activation follow these steps.

1. Run your app as a systemd service. It can either run directly on the host system or inside a container (using Docker or a systemd portable service for example). If you additionally pass an `IDLE_TIMEOUT` environment variable to your app it will gracefully shutdown if there are no requests for `IDLE_TIMEOUT` seconds. systemd will automatically start your app again when new requests are coming in.

/etc/systemd/system/myapp.service

```
[Service]
Environment=NODE_ENV=production IDLE_TIMEOUT=60
ExecStart=/usr/bin/node /usr/bin/myapp/build
```

2. Create an accompanying socket unit. The adapter only accepts a single socket.

/etc/systemd/system/myapp.socket

```
[Socket]
ListenStream=3000

[Install]
WantedBy=sockets.target
```

is made to `localhost:3000`.

Custom server

The adapter creates two files in your build directory — `index.js` and `handler.js`. Running `index.js` — e.g. `node build`, if you use the default build directory — will start a server on the configured port.

Alternatively, you can import the `handler.js` file, which exports a handler suitable for use with Express, Connect or Polka (or even just the built-in `http.createServer`) and set up your own server:

my-server.js

```
import { handler } from './build/handler.js';
import express from 'express';

const app = express();

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

  res.end('ok');
});

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

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

Edit this page on GitHub