Sapper - Making Svelte Real

What Is Sapper?

Sapper is a framework around Svelte for building web applications.

Svelte is an alternative to web frameworks like React, Vue, and Angular. For more detail on Svelte, see https://objectcomputing.com/resources/publications/sett/july-2019-web-dev-simpl

Sapper provides many features that are not part of Svelte. These include:

- page routing
- page layouts
- server-side rendering (SSR)
- code splitting
- Node-based REST services (server routes)
- pre-fetching for faster page loads
- static site generation (exporting)
- offline support using ServiceWorker
- end-to-end testing

The name "Sapper" has two meanings. First, it is a contraction of "Svelte app maker". Second, the English word "sapper" is defined as "a soldier responsible for tasks such as building a

The home page of the Sapper project is at https://sapper.svelte.dev/.

Sapper was created and is maintained by Rich Harris (Rich-Harris), the creator of Svelte, with help from many others including Alan Faubert (Conduitry), Luke Edwards (lukeed), Maurí

This article provides a thorough introduction to Sapper and walks you through the fundamentals necessary to start building web applications with it.

Getting Started

Let's walk through the steps to create and run a Sapper application.

1. Install Node.js from https://nodejs.org.

   This installs the `node`, `npm`, and `npx` commands.

2. Choose between using the Rollup or Webpack module bundler.
   To use Rollup, enter `npx degit "sveltejs/sapper-template#rollup"` *`app-name`*.
   To use Webpack, enter `npx degit "sveltejs/sapper-template#webpack"` *`app-name`*.

   The `degit` tool is useful for project scaffolding. It was created by Rich Harris. It downloads a git repo, by default the master branch. In this case "sveltejs" is the user name and "sap

3. `cd` *`app-name`*

4. `npm install`

5. `npm run dev`

   This starts a local HTTP server and provides live reload, unlike `npm run start` which omits live reload. Syntax errors are reported in the window where this is running, not in the b

6. Browse localhost:3000

   The default Sapper app contains three pages or routes with navigation at the top. The routes are "home", "about", and "blog". Links on the "blog" page open sub-pages that render s

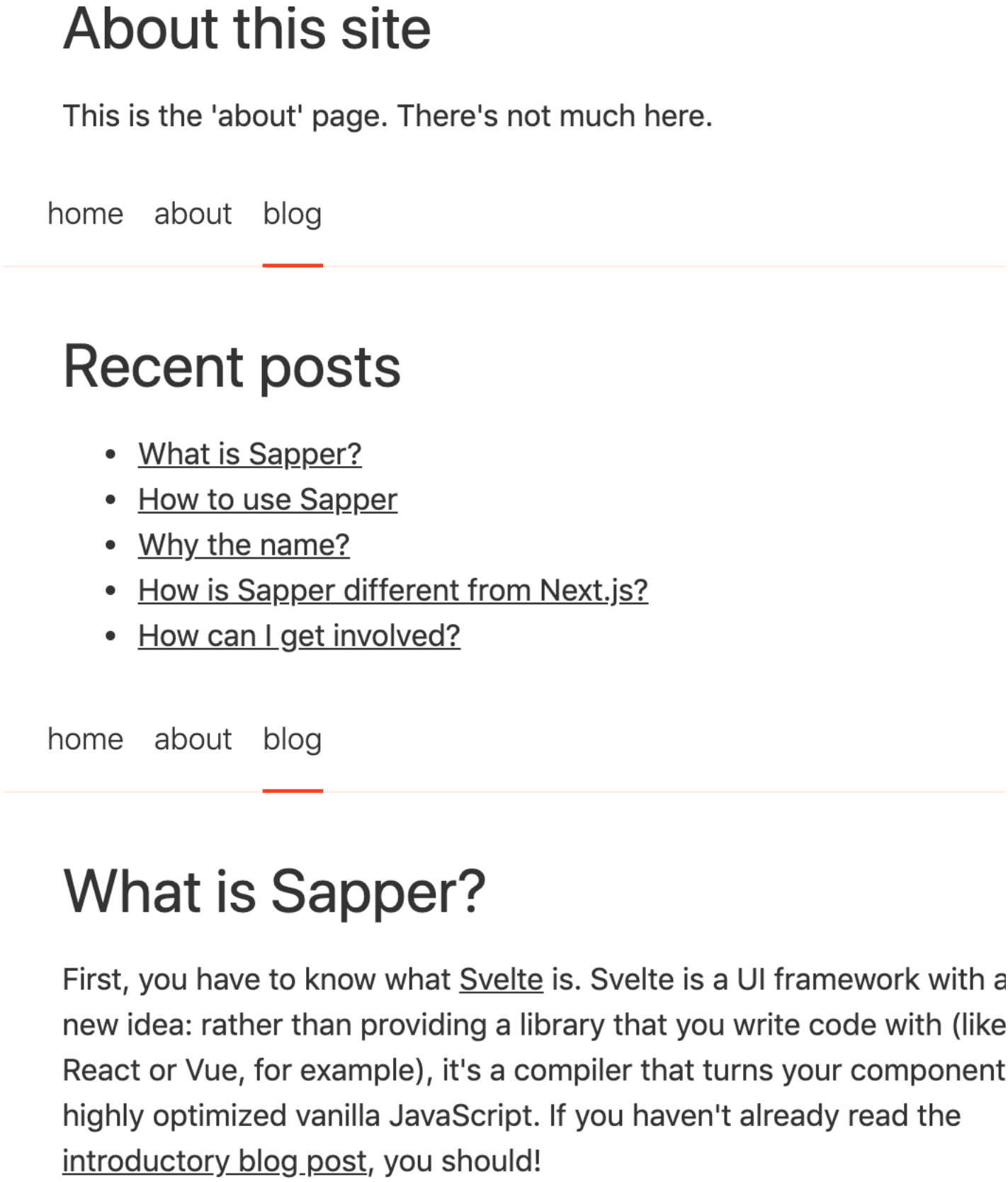
home about blog

GREAT SUCCESS

HIGH FIVE!

Try editing this file (src/routes/index.svelte) to test live reloadi

home about blog



Now you are ready to start modifying the app.

Sapper File Structure

Sapper apps created from the `sveltejs/sapper-template` begin with the following directory structure:

- `__sapper__`: destination for build artifacts
- `cypress`: contains directories and files for running end-to-end tests

- `node_modules`: contains files for package.json dependencies
- `src`: contains application code
- `components`: contains Svelte components used in pages
  - `Nav.svelte`: defines nav bar links
  - Modify this to change page navigation.
- `node_modules`
  - `@sapper`: contains files provided or generated by Sapper that should not be modified
  - `app.mjs`: exports Sapper API functions including:
    - `goto` for programmatic navigation
    - `start` which is called by `client.js` below
  - `server.mjs`: exports the Sapper `middleware` function called by `server.js` below
  - `service-worker.js`: exports constants used by `service-worker.js` below
    - `files` is an array of static files to be cached by the ServiceWorker
    - `shell` is an array of files generated by Sapper files to be cached by the ServiceWorker
  - `internal`
    - `App.svelte`: displays error page if there is an error or the current page otherwise
    - `error.svelte`: default error page component tha displays the status, error message, and stack trace (if NODE_ENV is "development")
    - `layout.svelte`: basic page layout containing only a `<slot>`
    - `manifest-client.mjs`: provides data about components and routes for `app.mjs` above
    - `manifest-server.mjs`: provides data about server routes and pages for `server.mjs` above
    - `shared.mjs`: seems to provide nothing
- `routes`: contains Svelte components that represent pages
- `client.js`: starting point of client-side Sapper app
- `server.js`: configures server for server routes
  - uses Polka by default; modify to use another library such as Express
- `service-worker.js`: defines ServiceWorker caching strategy
- `template.html`: HTML template for Sapper app
  - defines `sapper` id targeted by `client.js`
- `static`: contains static assets such as images and the `global.css` file
- `cypress.json`: configuration for Cypress end-to-end tests
- `package.json`: describes dependencies and defines npm scripts
- `README.md`: contains documentation on using Sapper
- `rollup.config.js`: configuration for the Rollup module bundler
- modify to use preprocessors like Sass and TypeScript

For many applications, most of the files described above do not required modification. The most common files to modify are described below.

Modify `src/Nav.svelte` to add or remove page navigation anchor tags. Consider creating a `NavItem` component to simplify this. For example:

```
<script>
  import {createEventDispatcher} from 'svelte';
  const dispatch = createEventDispatcher();

  export let main = false; // wanted to name this "default", but that's a keyword
  export let href = undefined;
  export let name; // required
  export let rel = undefined;
  export let segment; // required

  const capitalize = text =>
    text
      .split(' ')
      .map(word => word[0].toUpperCase() + word.substring(1).toLowerCase())
      .join(' ');

  const getClass = segment =>
    main ? (segment ? '' : 'selected') : segment === name ? 'selected' : '';

  const handleClick = () => dispatch('click');
</script>

<style>
  a {
    text-decoration: none;
    padding: 1em 0.5em;
    display: block;
  }

  a:hover {
    color: green;
  }
```

```
    li {
      display: block;
      float: left;
    }

    .selected {
      position: relative;
      display: inline-block;
    }

    .selected::after {
      position: absolute;
      content: '';
      width: calc(100% - 1em);
      height: 2px;
      background-color: rgb(255, 62, 0);
      display: block;
      bottom: -1px;
    }
  </style>

  <li>
    <!-- TODO: Why can't I just use "on:click" on the next line? -->
    <a
      {rel}
      class="{getClass(segment)}"
      href="{href"
      ||
      name}
      on:click="{handleClick}"
    >
      {capitalize(name)}
    </a>
  </li>
```

Add static assets such as images in the `static` directory and delete the provided images that are not needed.

Modify `static/global.css` to add CSS that should be used across all components in the app.

The file `public/index.html` contains the following:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf8" />
    <meta name="viewport" content="width=device-width" />
    <title>Svelte app</title>
    <link rel="icon" type="image/png" href="favicon.png" />
    <link rel="stylesheet" href="global.css" />
    <link rel="stylesheet" href="bundle.css" />
  </head>
  <body>
    <script src="bundle.js"></script>
  </body>
</html>
```

To use the `NavItem` component in the `Nav` component, modify it to look like this:

```
<script>
  import NavItem from './NavItem.svelte';
  export let segment;
</script>

<style>
  nav {
    border-bottom: 1px solid rgba(255, 62, 0, 0.1);
    font-weight: 300;
    height: var(--nav-height);
    padding: 0 1em;
  }

  ul {
    margin: 0;
    padding: 0;
  }
</style>

<nav>
  <ul>
    <NavItem {segment} name="home" />
    <NavItem {segment} name="about" />
    <NavItem {segment} name="blog" rel="prefetch" />
  </ul>
</nav>
```

Define additional pages by adding components in the `src/routes` directory. Add a `NavItem` instance in `src/components/Nav.svelte` for each new page.

Dynamic vs. Static Sites

Sapper can be used to build both dynamic web applications and static web sites.

Dynamic applications can make REST calls in response to user input.

Static web sites must make all required REST calls from component `preload` functions that are invoked when the site is built by running `npm run export`.

To test a dynamic site locally, enter `npm run dev` and browse localhost:3000.

To prepare a dynamic application for deployment, enter `npm run build`. This generates files in the `__sapper__/build` and `__sapper__/dev` directories.

Details on building static sites is provided later in the "Static Site Generation" section.

Page Routing

Each page in the application is implemented as a Svelte component defined in the `src/routes` directory. Route names are derived from the names of source files found here.

The default Sapper app includes anchor tags in the file `src/components/Nav.svelte`.

The recommended path to the page component source file for a route named `dogs` is `src/routes/dog.svelte` or `src/routes/dog/index.svelte`. The latter is preferred when
provides a common directory for these files. Server routes that do not require a path parameter, such as retrieving all instances of a resource or creating a new resource are typically def
deleting an existing resource are typically defined in `src/routes/dogs/[id].json.js`.

A result of this is that there will be many source files within an app that have names that are not meaningful on their own. Some editors support displaying the directory name of files alor
`workbench.editor.labelFormat` option to a value like "short".

Page Layouts

The main layout is defined in `src/routes/_layout.svelte`. Each route can define its own `_layout.svelte` file that is used to lay out its content.

Server-Side Rendering (SSR)

Sapper provides server-side rendering automatically with no configuration required.

HTML for the first page of an app that is accessed is generated on the server and downloaded to the browser. Subsequently visited pages are generated on the client just like in a norma

This is similar to the functionality provided by Next.js for React.

Code Splitting

Code splitting removes the need to download all the JavaScript for the app when the first page is rendered. Instead, only the JavaScript needed by the first page is downloaded. JavaSc

Svelte produces bundles that are much smaller than those produced by other frameworks. But the initial download size can be decreased even further by code splitting which is automat

When a browser first loads a Sapper app, only the JavaScript code necessary to render the first page/route is downloaded. The code for each remaining route is not downloaded until it

For routes whose anchor tag includes `rel="prefetch"`, download of their JavaScript code begins when the user hovers over their anchor tag. For components that export a module-le

In the starter app, this is specified on anchor tags in `src/components/Nav.svelte`.

Node-based REST services (server routes)

Sapper supports "server routes" for implementing server APIs (REST services) using Node.js. This enables implementing web applications where the client-side and server-side code is

This is an optional feature. Sapper apps can invoke REST services that are not implemented in a Sapper app. And of course the services can be implemented in any programming langu

By default the server routes are managed by Polka (https://github.com/lukeed/polka). Polka is described as "a micro web server so fast, it'll make you dance!" It was created by Luke Ed

Polka claims to be 33 to 50% faster than Express. It supports the same middleware as Express. The Polka API is nearly identical to that of Express.

Sapper applications can easily be modified so that server routes are managed by another Node.js server library such as Express.

To switch from Polka to Express:

- `npm uninstall polka`
- `npm install express`
- edit `src/server.js`
- remove `import polka from 'polka';`
- add `import {express} from 'express';
- to use Express built-in serving of static files instead of `sirv`
- `npm uninstall sirv`
- remove `import sirv from 'sirv';`
- add `const path = require('path');`
- remove `sirv('static', { dev }),`
- add `express.static('static')),`
- restart server

In server route functions do only one of these things:

1. Send a response.
2. Call next, passing it nothing. This enables the next "normal" middleware to run. One of these can send a response.
3. Call next, passing it an error description. This enables the next error handling middleware to run. One of these can log the error and/or send a response describing the error.

Fetch API Wrapper

From a comment in the "ROUTING ... Page" section of the official docs, "`this.fetch` is a wrapper around `fetch` that allows you to make credentialled requests on both server and cli

This should only be used in functions that are defined in module context. This means they are inside a `<script context="module">` tag. For functions define in a normal `<script>`

Just Enough MongoDB

In the example code that follows we will be persisting data using MongoDB. This choice is strictly based on ease of setup and use. It is not the best choice for every application. You sho

MongoDB is a NoSQL database implemented in C++. It stores documents in collections. Documents are JSON objects. They are stored in a binary JSON format (BSON).

All documents have an id stored in a property named `_id`. This is guaranteed to be unique within a collection.

Each collection can have multiple indexes to make queries faster. Indexes are implemented as B-tree data structures.

There is no need to define a schema that describes the properties of the document objects. This speeds up development when the structure changes often since no schema changes ar
collection have similar properties.

A document property value can be the id of another object, even in a different collection. This supports associations between documents.

The free version of MongoDB is called the "Community Edition".

To install the Community Edition on other platforms, browse https://docs.mongodb.com/manual/installation/, click "Install MongoDB Community Edition" in the left nav., click the "Install o

To start the MongoDB server, ... To stop the MongoDB server, ...

MongoDB Shell is a kind of REPL that supports using JavaScript to interact with a MongoDB database. To start it, enter `mongo`. To get help, enter `help`. To exit, enter `exit` or press ctrl-

To see a list of the current databases, enter `show dbs`.

To use a specific database, enter `use {db-name}`.

New databases are created automatically when a document is added to a collection. For example:

```
use animals
db.dogs.insert({name: 'Dasher', breed: 'Whippet'})
```

To list all the documents in a collection, enter `db.{coll-name}.find()`.

To delete a collection, enter `db.{coll-name}.drop()`.

To delete the current database, enter `db.dropDatabase()`.

To add a document to a collection, enter `db.{coll}.insert(obj)`. For example, `db.dogs.insert({breed: 'Whippet', name: 'Dasher'})`.

- to **get first** 20 documents in a collection, `db.{coll}.find()`
- ex. `db.dogs.find()`
- to **get all objects that match criteria**, `db.{coll}.find(criteria)`
- ex. `db.dogs.find({breed: 'whippet')`
- ex. `db.tjs.find({lastName: {$gt: 'H'}})`
- to **get number of documents** in a collection,
  `db.{coll}.find().count() or db.{coll}.find().length()`
- to **find document in collection**, `db.{coll}.findOne(query)`
- a query is just a JavaScript object where the keys are document property names and the values are document property values
- to **delete a document from collection**,
  `db.{coll}.deleteOne(query)`
- to **delete documents from collection**,
  `db.{coll}.deleteMany(query)`
- to **update a document in a collection**,
  `db.{coll}.updateOne(query, { $set: {key: value, key: value, ...} })`
- to **replace a document in a collection**,
  `db.{coll}.replaceOne(query, newDocument)`
- to **add an index** to a collection,
  `db.{coll}.createIndex({{prop-name}: 1});`
- 1 means ascending order, -1 means descending

Joins

Joins can be performed across collections using the `aggregate` method and the `$lookup` operator.

For example, suppose we have a collection of colors and a collection of people that have a favorite color. We would like to determine for each color which people like it.

The following Mongo shell commands populate the `color` collection:

```
db.colors.insert({_id: 1, name: 'red'});
db.colors.insert({_id: 2, name: 'orange'});
db.colors.insert({_id: 3, name: 'yellow'});
db.colors.insert({_id: 4, name: 'green'});
db.colors.insert({_id: 5, name: 'blue'});
db.colors.insert({_id: 6, name: 'purple'});
```

The following Mongo shell commands populate the `people` collection.

```
db.people.insert({name: 'Mark', favoriteColor: 3});
db.people.insert({name: 'Tami', favoriteColor: 5});
db.people.insert({name: 'Amanda', favoriteColor: 6});
db.people.insert({name: 'Jeremy', favoriteColor: 3});
```

The following Mongo shell command returns a cursor for iterating over documents where data from the documents above are combined. It will return a set of documents that are similar
from the `people` collection whose `favoriteColor` matches.

```
db.colors.aggregate([
  {
    $lookup: {
      from: 'people',
      localField: '_id',
      foreignField: 'favoriteColor',
      as: 'people'
    }
  }
]);
```

The documents returned are shown below:

```
{ "_id" : 1, "name" : "red", "people" : [ ] }
{ "_id" : 2, "name" : "orange", "people" : [ ] }
{ "_id" : 3, "name" : "yellow", "people" : [ { "_id" : ObjectId("5daa32fc3bed06a14673ca80"), "name" : "Mark", "favoriteColor" : 3 }, { "
{ "_id" : 4, "name" : "green", "people" : [ ] }
{ "_id" : 5, "name" : "blue", "people" : [ { "_id" : ObjectId("5daa32fc3bed06a14673ca81"), "name" : "Tami", "favoriteColor" : 5 } ] }
{ "_id" : 6, "name" : "purple", "people" : [ { "_id" : ObjectId("5daa32fc3bed06a14673ca82"), "name" : "Amanda", "favoriteColor" : 6 } ]
```

The `localField` property value can be an array in which case the `foreignField` is checked for a match with any of the array values.

This does not work if the "from" collection is sharded.

Can you specify the properties from each collection to be included in the result? See the `$replaceRoot` and `$mergeObject` operators. It is probably easier to process the result with

To join on multiple properties, use the `pipeline` property of `$lookup` combined with the `$match` operator.

Pre-fetching

Routes can be configured so the data they required is downloaded when the user simply hovers over their anchor tag. This anticipates that the user will click the anchor tag and results i

This functionality is configured by adding the `rel="prefetch"` attribute to an anchor tag.

Pre-fetching works in conjunction with code splitting. The first time a user hovers over an anchor tag that uses prefetch, JavaScript required by the page is downloaded and its `preload`

Each route component can export a `preload` function from its module scope (`<script context="module">`). This is called before the route component is rendered.

Defining a `preload` function in a component that is not a route has no purpose because it will never be invoked.

It is not possible to register the same component for multiple route paths because the route path also specifies the path to the component `.svelte` file.

Note how `preload` differs from the `onMount` lifecycle function which is invoked once for each instance of a component that is created.

If the same route component is used with more than one route path, is the `preload` function called once for each?

Here is an example from the Dog component:

```
<script context="module">
  export async function preload(page, session) {
    try {
      const res = await this.fetch('dogs.json');
      if (res.ok) {
        const dogs = await res.json();
        const dogMap = dogs.reduce((acc, dog) => {
          acc[dog._id] = dog;
          return acc;
        }, {});
        // Properties in the object returned are passed to this component as props.
        return {dogs: dogMap};
      } else {
        const msg = await res.text();
        this.error(res.statusCode, 'Dogs preload: ' + msg);
      }
    } catch (e) {
      this.error(500, 'Dogs preload error: ' + e.message);
    }
  }
</script>
```

If the `rel="prefetch"` attribute is added to the anchor tag (`<a>`) of a route, its `preload` function will be called when the user hovers over the anchor tag (or taps on mobile devices). T uses prefetch. If either of these actions occur, hovering over the anchor tag with prefetch will call the `preload` function again.

This cannot be enabled for `<button>` elements, but anchor tags can be styled to look like buttons.

For example:

```
<style>
  a {
    border: solid gray 1px;
    border-radius: 4px;
    padding: 4px;
    text-decoration: none;
  }
</style>

<a href="/dogs" rel="prefetch">
  Go To Dogs
</a>
```

The `preload` function is only invoked once before the link is clicked. Hovering over it multiple times does not trigger multiple invocations. After navigating to that route and then navigati

Static Site Generation

Sapper can crawl the anchor tags in an app in order to generate HTML for an entire site. This includes making REST calls to gather data needed to render pages.

This is similar to the functionality provided by Gatsby for React.

To build a static site, enter `npm run build`, just like for a dynamic site. Then enter `npm run export` which generates files in the `__sapper__/export` directory.

The `export` script recursively crawls every `<a>` tag reachable from `src/routes/_layout.svelte` to find all the pages of the app.

If any reachable pages contain a `preload` function that make REST calls, the servers for those must be running when the `export` script is run. Each of these REST calls is invoked an REST calls.

To test the site locally, enter `npx serve __sapper__/export` and browse localhost:5000.

During development of a static site it will be necessary to run these commands many times. To simplify this:

1. `npm install -D npm-run-all serve`
2. Add this npm script:

   ```
   "serve": "serve __sapper__/export",
   ```

3. Add this npm script:

```
"static": "npm-run-all build export serve",
```

Now `npm run static` can be used to run all three commands.

Let's walk through an example of building a static site starting with the Sapper app template. We will modify it so the "About" page is replaced by a "Dogs" page that displays a list of dog

First we will build the app and test it as a dynamic site.

- Enter `npx degit "sveltejs/sapper-template#rollup" my-static-site`.
- Enter `cd my-static-site`.
- Enter `npm install`.
- Enter `npm run dev`.
- Browse localhost:5000.
- Verify that the app works, including clicking the "about" tab.
- Edit `src/components/Nav.svelte` and change all occurrences of "about" to "dogs". (Consider describing use of your `NavItem` component.)
- Create a "dogs" directory under `src/routes`.
- Create `src/routes/dogs/index.svelte` containing the following:

  ```html

## Dogs

{#each Object.values(dogs) as dog}

{dog.name} is a {dog.breed}

{/each} ```
  ```

The `page` parameter of the `preload` function is an object with `host`, `path`, `query`, and `params` properties. An example `host` value is "localhost:3000". An example `path` value is "dog

The `session` parameter of the `preload` function is an object containing session data, but is undefined when not using sessions.

- Create `src/routes/dogs/index.json.js` containing the following. This implements a REST service that is invoked by sending a GET request to http://localhost:3000. If the RE

```
let lastId = 0;
const dogs = {};

function addDog(breed, name) {
  const dog = {id: ++lastId, breed, name};
  dogs[dog.id] = dog;
}

addDog('Whippet', 'Dasher');
addDog('Treeing Walker Coonhound', 'Maisey');
addDog('Native American Indian Dog', 'Ramsey');
addDog('German Shorthaired Pointer', 'Oscar Wilde');

export function get(req, res) {
  res.end(JSON.stringify(dogs));
}
```

- Start the app as a dynamic app by entering `npm run dev`.
- Browse localhost:3000.
- Click the "dogs" tab and verify that a list of dogs is rendered.

Here are the steps to build a static version of this app. Note that `npm run dev` has already built the app and created files under the `__sapper__/build` and `__sapper__/dev` direct

- If the REST service to retrieve all the dogs has been implemented outside of Sapper, start that server.
- Enter `npm run export`.
- Enter `npx serve __sapper__/export`.
- Browse localhost:5000.
- Click the "dogs" tab and verify that the same list of dogs is rendered, this time using a generated `.json` file instead of calling a REST service.

Offline Support Using ServiceWorker

Using a ServiceWorker allows parts of a web application to continue being usable after network connectivity is lost.

Is the ServiceWorker enabled by default?

It will use cached REST call data when the data source is unavailable, such as when a MongoDB server is stopped.

The ServiceWorker can be stopped in the Chrome devtools. Select the "Application" tab and "Service Workers" in left nav.

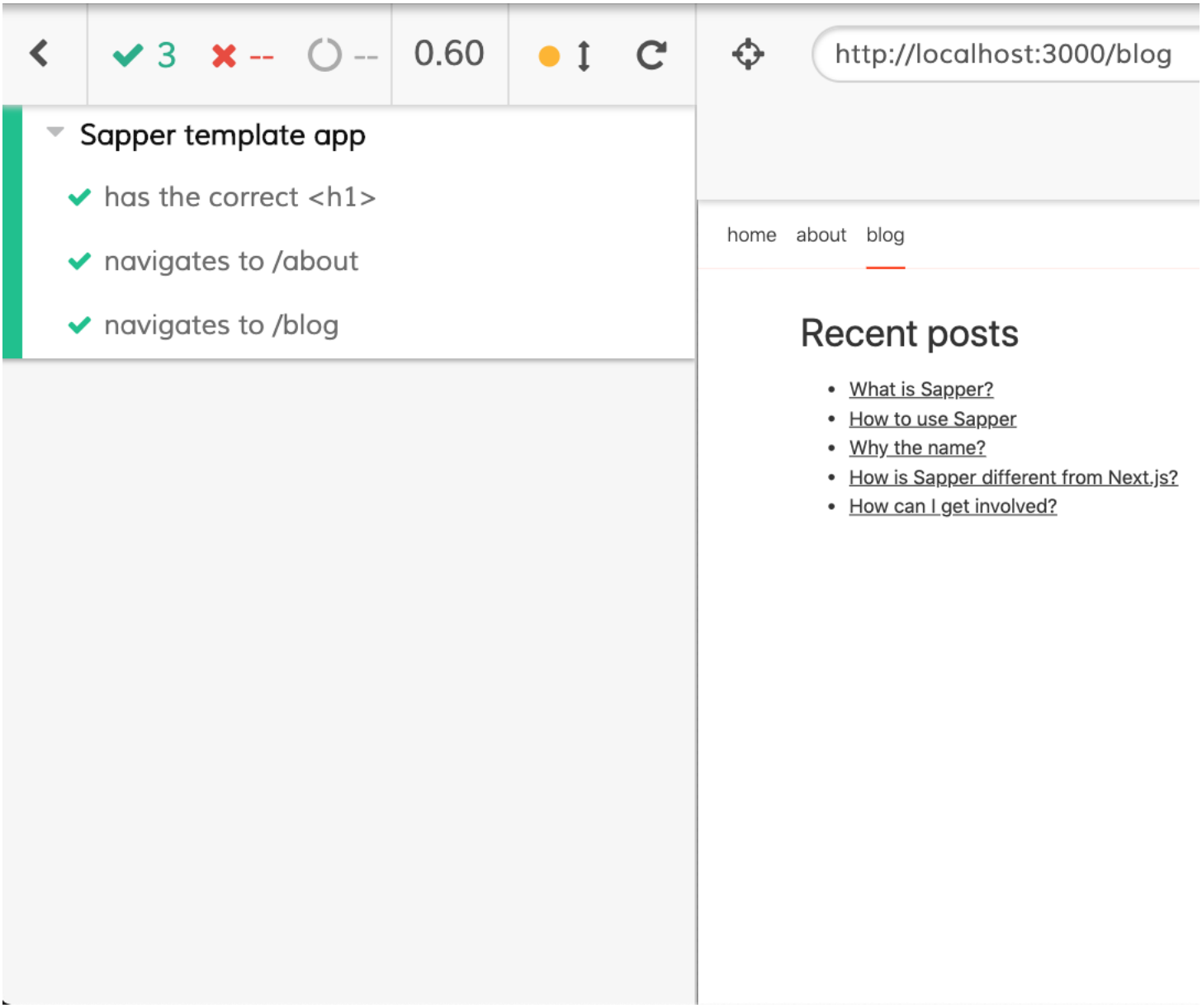
End-to-End Testing

Sapper apps have npm scripts to run Cypress tests (`npm run test`) and view the results.

However, Cypress is not installed by default. To install it, enter `npm install -D cypress`.

To run the tests in batch mode so all the results are displayed in a terminal window, enter `npm run test`. When running the tests this way, it is not required to have a local server runni

To run the test in the default browser, start the server by entering `npm run dev` and enter `npm run cy:open`.



A sample test is provided in `cypress/integration/spec.js`.

To implement a Cypress test ...

page and session

Where does this content belong?

`page` and `session` are available as stores in all components. What data do they hold? Try the following:

```
<script>
  import {stores} from '@sapper/app';
  const {preloading, page, session} = stores();
</script>

{$page} {$session}
```

Conclusion

There you have it! Sapper is a great addition to Svelte. Most Svelte applications should be built with Sapper.

Thanks so much to ? for reviewing this article!

Please send corrections and feedback to mark@objectcomputing.com.