SVELTE FORMS

FROM WTF TO FTW

- Learn how to effectively work with forms in Svelte
- Learn the different styles of form validation
- Learn how to build highly dynamic forms
- Learn how Svelte context, stores and slots work
- Learn how to create a forms library in 120 LOC

Ilia Mikhailov

Contents

The Simplest Svelte Form

Long time ago, measured in tech time of course, browsers had no JavaScript. We used pure HTML forms to submit data to the servers, validated data server-side and returned a new page as a response. Taking into account that we also had *28.8kb* modem connection the user experience wasn't that great. Especially if someone in your family had to use the phone to make the call and picked up the landline phone.

Times have changed, technology has changed, but the standards remain. Let us refresh our memory and look at the pure HTML only form. We will use a typical login form with email and password as two input fields and submit and reset buttons.

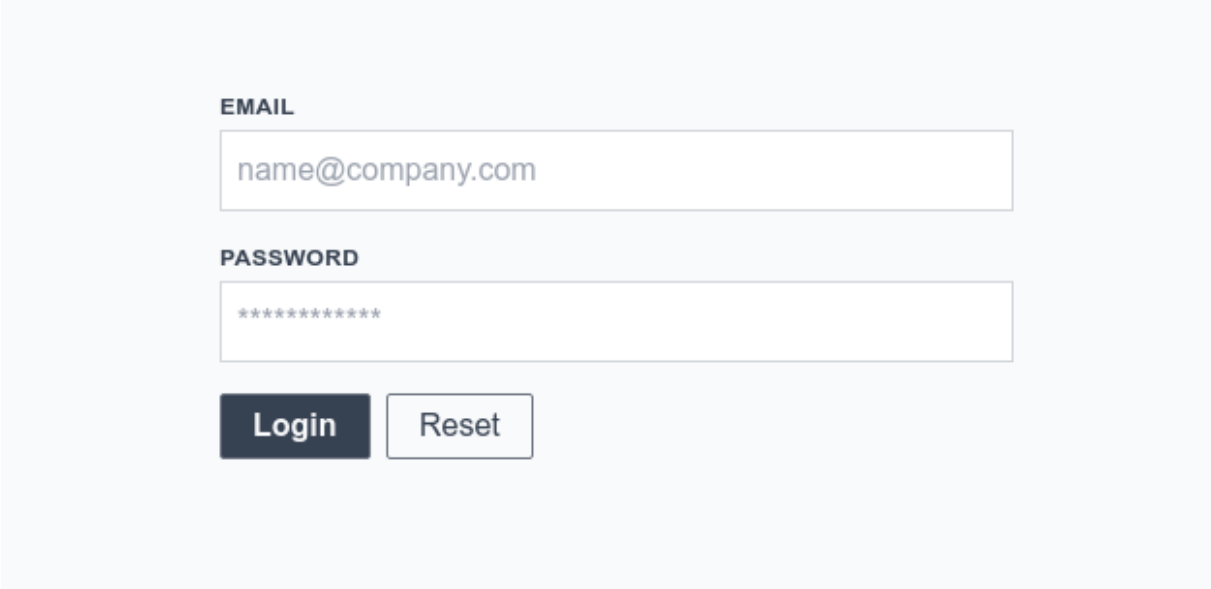

Figure 1: Simple Login Form

The HTML code required for this form is the following.

```
<form method="POST" action="/api/login">
  <div>
    <label>
      <span>Email</span>
      <input type="text" name="email" placeholder="name@company.com" />
    </label>
  </div>
  <div>
    <label>
```

```
        <span>Password</span>
        <input type="password" name="password" placeholder="************" />
      </label>
    </div>
    <div>
      <button type="submit">Login</button>
      <button type="reset">Reset</button>
    </div>
  </form>
```

Example 1: Pure HTML form

The only difference from today's fancy forms is that we specified `action` and `method` on it. When the user clicks the *submit* button the browser will submit the form to the URL specified in the `action` attribute of the form tag. If you don't explicitly specify any `method` browser will use GET method, and if there is no `action` attribute, the browser will submit the form to the current URL.

This is sort of the first version of the SSR[6]. If you worked in tech for a while, gone around the circle a couple of times, you will start noticing that there is actually very little real innovation going on, only slightly fancier ways of doing same thing, but in a much obscure way.

Adding JavaScript

At the end of 90s a new browser called *Netscape Navigator* was released. It was all the rage and quickly gained huge market share. In the version 2 the Navigator got support for a simple scripting language called JavaScript. Many people didn't know how to use it, but there were a few brave souls that did very cool pages with something they called *DHTML*. *D* stands for *Dynamic*.

[6] Server Side Rendering

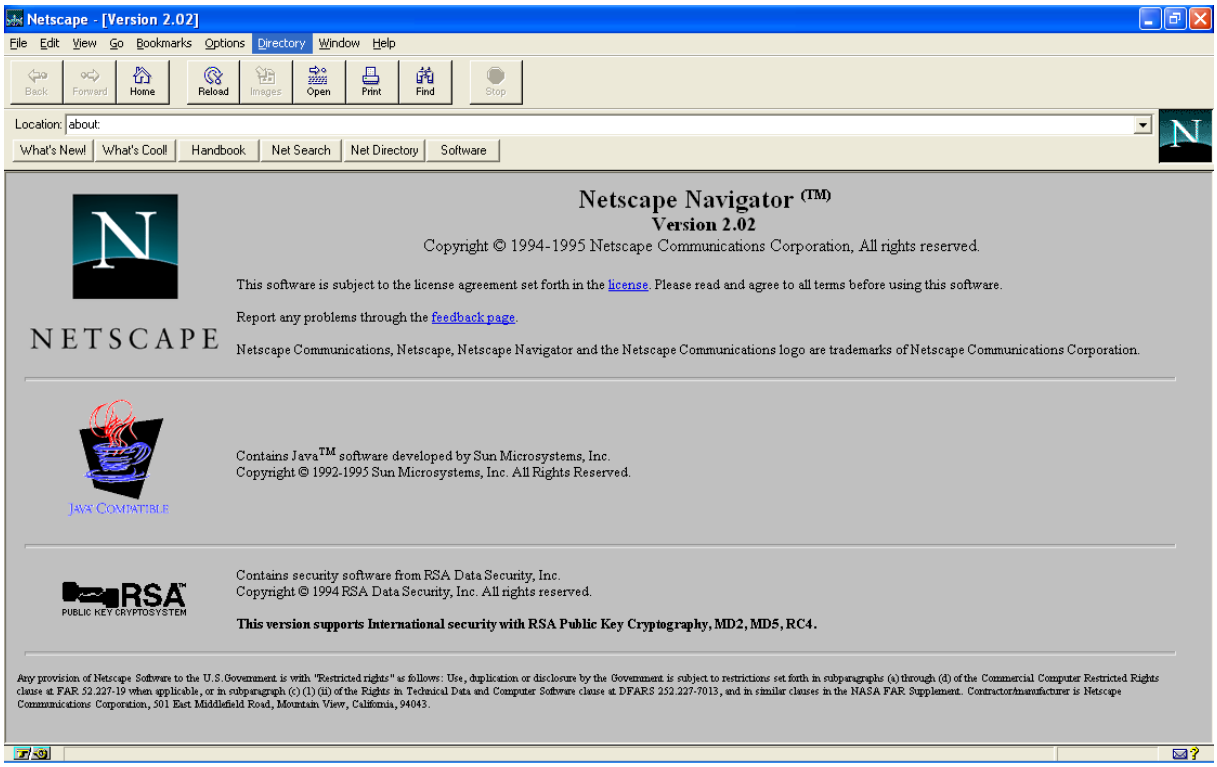

Figure 2: Netscape Navigator

We are talking about the time when *blink* and *marquee* tags were popular and almost every page had a mandatory “under construction” banner and cool animated GIFs. You were under impression that no project was ever completed, but that was OK. The web was new, fresh and fun. People didn’t care. It was more important to get your personal page on the Internet than the content that was on it.

Time went by and the web was catching on. Various organizations and committees were started, and standards were written. The web was maturing. Fast forward to now and you hardly have to use polyfills anymore and the pain of supporting Internet Explorer 8 is a bleak, distant memory.

In fact, we have come so far with different standards that you can almost write the same code for all the browsers. Which is great! JavaScript as a language itself has matured and different browsers and JavaScript engines support more or less the same base language specification. You can even write your app using vanilla JavaScript if you feel like it.

Today you hardly every see pure HTML forms being used and if you stumble on such you will most likely be suspicious of the page in question and unconsciously slap a “shady” tag on it in your mind.

Even though vanilla JavaScript is good enough for a simple task at hand, you must be a mad (or have a lot of time) to write a whole app in it. Today there are many great frameworks that can help us write code using smart abstractions.

Personally, I’ve started to think of JavaScript as the low level assembly language for the Web, the glue that keeps the page together, and all these frameworks as higher level abstractions that help us work with it.

Frameworks are great, but we still need to understand the basics as in the end it’s all being

compiled to pure JavaScript and the code manipulating the DOM.

Here is the same login form, but rewritten with JavaScript submission support.

```
<script>
  function submit(e) {
    // prevent the default submit action
    e.preventDefault();

    const fd = new FormData(this);
    const values = Object.fromEntries([...fd]);
    console.log(values);
  }

  // wait for the DOM to load
  window.addEventListener('load', () => {
    const form = document.getElementById('login');
    form.addEventListener('submit', submit);
  });
</script>

<form id="login">
  <div>
    <label>
      <span>Email</span>
      <input type="text" name="email" placeholder="name@company.com" />
    </label>
  </div>
  <div>
    <label>
      <span>Password</span>
      <input type="password" name="password" placeholder="************" />
    </label>
  </div>

  <div>
    <button type="submit">Login</button>
    <button type="reset">Reset</button>
  </div>
</form>
```

Example 2: Vanilla JavaScript form

If you study the code you will see that we hijack the form's submit event by wiring it to our own `submit` handler. Inside it we prevent the default submit browser call and extract the form data using the built-in `FormData` constructor[7]. In this example we just log the data to the browser's console, but you would probably make an AJAX[8] call to your API, validate the supplied credentials and either return an error or some kind of JWT token or set a session cookie.

Svelte Form

With pure HTML and pure vanilla JavaScript forms done it's time to climb up a level in the abstraction tree and use a proper web framework. How about Svelte?

---

[7] https://developer.mozilla.org/en-US/docs/Web/API/FormData

[8] I say "AJAX", but is it still the term to use today? It sounds very old school to me

I've worked in many web frameworks during my career, but I really like Svelte. Why? Because it feels simple. Something that you actually can understand. In my opinion, it helps us reduce the cognitive load. This is something very important as it saves our brain from overheating. Another important aspect of reduced cognitive load and Svelte is the lower entry barrier to code. If you have to jump into a new project, Svelte code is easier to understand and follow than code written in some of the other popular frameworks.

Svelte is also very beginner friendly. If you are just getting started with web development Svelte might be a perfect entry web framework. Also because of its simplicity.

Writing Svelte code often feels like writing regular HTML and vanilla JavaScript code and this is what makes it so great. I like to say that Svelte just regular HTML and JavaScript, but on steroids and syntactic sugar sprinkled on top.

In order to explain what I mean, and prove it in code, let's rewrite our JavaScript form in Svelte.

```
<script>
  import { onMount } from 'svelte';

  let form;

  function submit(e) {
    // prevent the submit action
    e.preventDefault();

    const fd = new FormData(this);
    const values = Object.fromEntries([...fd]);
    console.log(values);
  }

  onMount(() => {
    form.addEventListener('submit', submit);
    return () => form.removeEventListener('submit', submit);
  });
</script>

<form bind:this={form}>
  <div>
    <label>
      <span>Email</span>
      <input type="text" name="email" placeholder="name@company.com" />
    </label>
  </div>
  <div>
    <label>
      <span>Password</span>
      <input type="password" name="password" placeholder="**********" />
    </label>
  </div>

  <div>
    <button type="submit">Login</button>
    <button type="reset">Reset</button>
  </div>
</form>
```

Example 3: Almost vanilla JavaScript form

There is not much difference between vanilla JavaScript form and this. We've used only a few

Svelte abstractions to make it Svelte compatible.

First, we use `bind:this` directive to bind the form DOM element to a variable. Then we use Svelte's `onMount` hook to wire up form's submit handler when our Svelte component is mounted on the DOM. As you can see, only minimal changes to the code were needed.

Actually, the template form that Svelte is using is called *HTMLx*[9]. The **x** stands for plus, so it's "HTML-plus". If you ask me, it would make more sense that **x** would stand for e**X**tented.

Simple Login Form

You can get away with the code above, but it's not how you would write it in Svelte. What is the simplest form we can get away with in Svelte? Is it the code example below?

```
<script>
  function submit(e) {
    // prevent the submit action
    e.preventDefault();

    const fd = new FormData(this);
    const values = Object.fromEntries([...fd]);
    console.log(values);
  }
</script>

<form on:submit={submit}>
  <div>
    <label>
      <span>Email</span>
      <input type="text" name="email" placeholder="name@company.com" />
    </label>
  </div>
  <div>
    <label>
      <span>Password</span>
      <input type="password" name="password" placeholder="***********" />
    </label>
  </div>

  <div>
    <button type="submit">Login</button>
    <button type="reset">Reset</button>
  </div>
</form>
```

Example 4: Simplest Svelte form

Lets' break it down. We created a normal HTML form and wired up form's `submit` handler with our own `submit` function.

As you can see the only Svelte directive is `on:submit`. Everything else is vanilla JavaScript. Our code is also shorter, because we've outsourced parts of it to Svelte.

When we press the *Login* button the browser will try to submit the form. In the *submit* function

[9] https://github.com/htmlx-org/HTMLx

we prevent the browser from submitting the form by calling `event.preventDefault()` method.

> For this example we have to use the traditional function definition and not new ES6 *fat arrow* function form. Why? Because arrow functions don't have the notion of `this`.

Alternative Form Version

If you want to use the ES6 arrow function[10] you can re-write the form in the following way.

```
<script>
  const submit = ({ target: form }) => {
    const values = { email: form.email.value, password: form.password.value };
    console.log(values);
  };
</script>

<form on:submit|preventDefault={submit}>
  <div>
    <label>
      <span>Email</span>
      <input type="text" name="email" placeholder="name@company.com" />
    </label>
  </div>
  <div>
    <label>
      <span>Password</span>
      <input type="password" name="password" placeholder="***********" />
    </label>
  </div>
  <div>
    <button type="submit">Login</button>
    <button type="reset">Reset</button>
  </div>
</form>
```

Example 5: Login form using fat arrow function

Which version is better? The one using `FormData` or the *fat arrow* one? It's a matter of personal taste. Personally, I prefer the `FormData` version, because you need fewer code changes if we need to add a new field.

You might also have noticed that we abstracted away the `preventDefault` method call by moving it to the `submit` directive on the form element. This is another neat feature of Svelte that is called *event modifier*. You have a handful of them in Svelte and you can also chain them together.

Compiler Generated Code

In pure HTML and vanilla JS form example the code is parsed by the browser as it's written, but this is not the case in Svelte.

[10] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Svelte is a compiler. It parses the code and templates you write in your Svelte files and generates completely new code in JavaScript. If you peek under the hood and look at the code Svelte generated for us you will find something like this.

```
function mount(target, anchor) {
  insert_dev(target, form, anchor);
  append_dev(form, div0);
  append_dev(div0, label0);
  append_dev(div0, t1);
  append_dev(div0, input0);
  append_dev(form, t2);
  append_dev(form, div1);
  append_dev(div1, label1);
  append_dev(div1, t4);
  append_dev(div1, input1);
  append_dev(form, t5);
  append_dev(form, div2);
  append_dev(div2, button0);
  append_dev(div2, t7);
  append_dev(div2, button1);

  if (!mounted) {
    dispose = listen_dev(form, "submit", prevent_default(submit), false, true, false);
    mounted = true;
  }
}
```

Example 6: Code generated by the Svelte compiler

When you load the JavaScript file generated by Svelte in the browser, the browser will parse it and build up a page (or DOM tree). As you can see, although Svelte code we write in our editor closely resembles normal HTML and JavaScript code, the difference between normal HTML file and Svelte-generated code is big.

But this is also the reason I like Svelte. It feels like you are going back to basics when using it. You write code that's actually readable. Something that you can follow and reason about.

In my opinion, Svelte is a thin wrapper on top of plain JavaScript. At least it feels this way when you write the code. A typical Svelte file consists of a `script` and optional `style` block and the actual HTML template. Just like a classic HTML page.

Summary

By now you should understand the how basic forms work, how Svelte files closely resembles normal HTML files and the difference between vanilla Javascript and Svelte-generated code.

So far, we've kept everything basic, but when working with web development, and forms especially, you should always be paranoid. You must always validate the data users enter into a form in order to prevent corrupt data and minimize the security risks.

Every form needs validation. Period. Luckily, HTML5 has basic built-in browser validation that we can leverage. Let's tackle that in the next chapter.