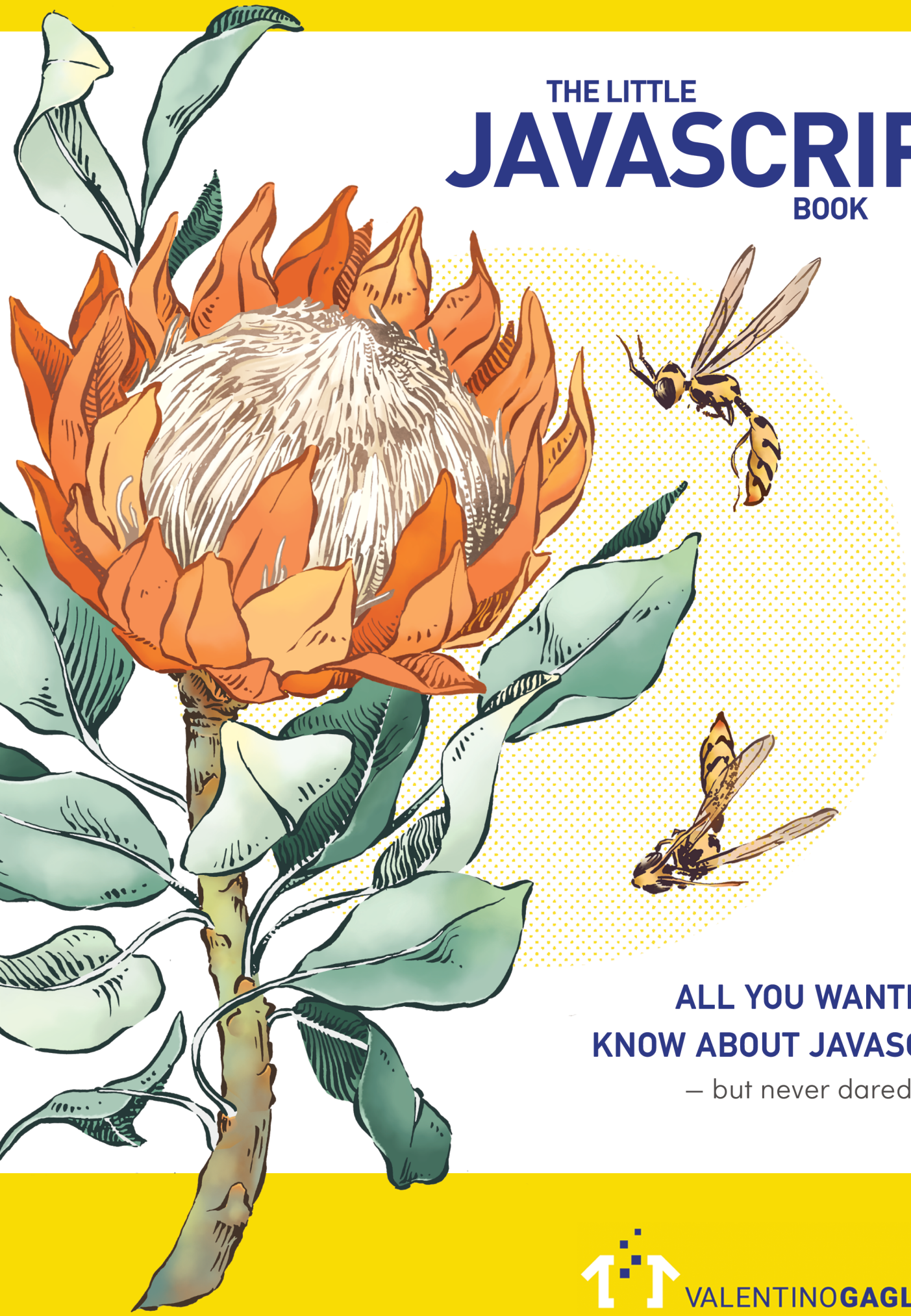
THE LITTLE JAVASCRIPT BOOK



ALL YOU WANTED TO KNOW ABOUT JAVASCRIPT

– but never dared to ask

VALENTINO GAGLIARDI

The Little JavaScript Book

All you wanted to know about JavaScript but never dared to ask.

Valentino Gagliardi

This book is for sale at http://leanpub.com/little-javascript

This version was published on 2020-11-10

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To my grandpa Valentino. Thanks for teaching me humility. To Caterina. Thanks for being with me.

Contents

Chapter 1. Getting started with JavaScript, and a glossary.

In this chapter:

- A brief history of JavaScript.
- A web developer's glossary.

The importance of being JavaScript

JavaScript saw the light in 1995 from Brendan Eich's hands. At that time the web was still in its infancy, and most of the fancy web pages we can see today were still a dream. Chased by his project manager, Brendan had only 10 days for creating a dynamic, flexible language that could run in a browser. He came up with JavaScript, a somewhat weird programming language which borrowed from Java (the cool kid at the time), C, and Scheme. JavaScript always had a bad reputation because it was full of quirks from the very beginning. Despite that it conquered a seat in the hall of fame, and it's here to stay. With the rise in usage the JavaScript ecosystem also saw a cambrian explosion. Most of the tools and libraries we use today for developing JavaScript are … well, written in JavaScript! Nowadays JavaScript powers full-fledged single page applications, static site generators, and server-side rendering platforms with Node.js.

Why should you study JavaScript? Mastering JavaScript does not mean a shallow understanding of variables and functions: there is a lot more. An expert JavaScript developer knows closures, `this`, `new`, the prototype system, and newer features. JavaScript is getting better and better year after year. Almost every frontend developer job requires JavaScript knowledge nowadays. It's not "basic jQuery" that hiring managers are looking for. Frontend developers should also be aware of JavaScript quirks in order to write idiomatic, well-structured JavaScript code. JavaScript is spreading fast. Better be prepared.

ECMAScript, and other technical terms

If you're just starting out with web development chances are you'll be soon overwhelmed by technical jargon. You'll begin to ask yourself "what is AJAX?", "What is an API"? You'll find a

lot of technical terms in the book. Expert developers know them all, but I don't want you to feel bad because you don't know what API means. Here's a little glossary of the most important terms.

AJAX: a set of technologies for fetching data in the browser without causing a page refresh. The acronym stands for "Asynchronous JavaScript and XML", coined in 1999.

API: stands for Application Programming Interface, but don't bother the strict definition for now: an API in programming is a set of tools, a toolbox of functions (also called methods), built by other developers and ready for use. With time, you'll learn how the term has slightly different meanings depending on the context.

Native API: a native API is a collection of built-in methods available by default in a programming environment. Speaking of browsers for example we say that `document.querySelector()` is part of the native API for selecting HTML elements.

Browser console: in most web browsers you can access a developer toolbox. In Firefox and Chrome on Mac press Command + Option + I, on Linux and press F12. That will open an interactive console where you can type and execute JavaScript code. The console will also show errors and other messages from your JavaScript programs.

Debugger: debuggers are tools built for helping developers to find why and where a program stops working. In JavaScript there is also the `debugger` instruction which stops the script exactly where it's placed.

Browser API or Web API: like native APIs a Web API is a specific functionality available in a web browser. Developers can use methods from these APIs out of the box. Examples of Web API are the DOM, the Console API, the Storage API. For a complete list check out Web APIs[1].

ECMAScript: it is a standard from which JavaScript has been implemented. It could also be used as the official name for JavaScript. In 1996 JavaScript was donated to ECMA international, a third-party entity which takes care of defining standards for a number of technologies.

ES5: acronym for ECMAScript 2009, the fifth version of JavaScript. To avoid confusion it's more correct to say ECMAScript + year for denoting a specific JavaScript version.

ES6: stands for ECMAScript 2015, the sixth version of JavaScript. Since 2015 the JavaScript committee decided to release new features yearly. From there we had ECMAScript 2016, ECMAScript 2017, ECMAScript 2018, and so on.

ES module: it's a mechanism for code reuse in JavaScript, introduced in 2015. It's finally becoming the standard in the highly fragmented JavaScript module scene. In some ways it's similar to Python's module system.

JavaScript engine: is part of the browser and is able to compile and interpret JavaScript code. Browser vendors build JavaScript engines by following (sometimes not so strictly) a document called JavaScript specification.

JavaScript specification: is a formal, written document which outlines how the JavaScript language should behave. Browser vendors read the spec and implement JavaScript engines in a way that

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

JavaScript code is executed as the spec prescribes.

Interpreter: an interpreter in programming takes your code and through a series of transformations makes your instruction understandable by the computer. JavaScript engines for example have an interpreter for translating JavaScript code to bytecode, an intermediate language that is then read by other components of the engine, and in turn translated to machine code.

Node.js: an environment for running JavaScript outside of the browser. It includes a JavaScript engine, V8, for compiling and executing the code. Node.js is mostly used on the server-side and for command-line tools.

Node package manager: npm in short, is a tool for managing the workflow of JavaScript projects, from installing third-part packages to shipping code.

HTTP request: is the act of "talking" to a remote web server (also called web service) for fetching or saving data. An example of HTTP request is when you visit some web page with a browser. Web pages in turn can make HTTP requests too for fetching data, mostly to REST APIs (see below). While a web service is not the same thing as a web server, there is always some kind of server listening for connections behind a web service.

HTTP error: sometimes things don't go well when talking to web services, and the server might respond with an error. Errors are denoted with a numeric code: some common errors are 500 (server error), 404 (not found), 403 (forbidden), and so on.

JSON: JSON stands for JavaScript Object Notation, a format for exchanging data between web services and web applications, yet not only limited to them.

REST API: is a web service (local or remote) which can expose data. Web applications (and any kind of application in general) can make HTTP request to a REST API for interacting with an underlying database, or simply for sending commands to the system.

Transpiler: older browsers do not support modern JavaScript syntax from ECMAScript 2015 and beyond. A transpiler is a tool which takes modern JavaScript syntax and spits out a more compatible version (ECMAScript 2009).

Proposal: JavaScript innovation is fueled by a group of developers and academics forming a committee, called TC39. Members of the committee can submit proposals for improving and adding new features to the language. A proposal is a formal description outlining what the new feature does and how it'll be used in JavaScript.

Stage N: new JavaScript proposals always start at stage 0. The more the proposal is voted by the committee, the more it advances to the next stages: 1, 2, 3, and 4. Every time you read stage 1 or stage 2, it's the stage at which the proposal is currently in. A proposal at stage 2 for example means it is doing fairly well, and it will most likely move forward to the next stages. The final stage is 4, meaning the new feature will land into the language.

Single threaded: a programming language is single threaded when the interpreter runs each instruction in sequence. JavaScript and Python for example are single threaded. A function taking too long to complete could block the entire program in a single threaded language. In chapter 3 we'll see how JavaScript engines try to solve this problem.

Vanilla JavaScript: vanilla JavaScript is a term for denoting pure JavaScript applications, i.e. those written without the help of a frontend library like React, Vue, or Angular.

XMLHttpRequest: it is a native object available in browsers for making HTTP requests to remote resources. `XMLHttpRequest` is part of the AJAX family, a set of technologies for fetching data in the browser without causing a page refresh.

Fetch API: it is a native API for making HTTP requests, much like `XMLHttpRequest`, but based on ECMAScript 2015 Promises. It is considered the successor of `XMLHttpRequest`, yet builds on top of it as you'll see later in the book.

CORS: acronym for Cross-Origin Resource Sharing. It is a way for servers to control access to resources on a given origin, when JavaScript requests them from a different origin. An origin consists of a scheme, domain, and port number. For example even if they share the same domain (example.com), https://api.example.com:5000 is a different origin from https://frontend.example.com. By default, browsers block AJAX requests to remote resources which are not on the same origin, unless a specific HTTP header named `Access-Control-Allow-Origin` is exposed by the server.

A special note on modern JavaScript

ECMAScript 2015 is one of the most important releases ever for JavaScript. It introduced a lot of improvements into the language and in this book you'll find the most important ECMAScript 2015 features like `Promise`, arrow functions, class, and more. JavaScript is getting better year after year with new features making their way into the language almost daily. I wish I could go through every tiny bit of modern JavaScript here, but another book would not be enough to cover them all. I took another path, and I decided to include just the most relevant features strategically, here and there during the chapters. Have fun!

Chapter 10. HTML forms and playing with the storage API.

In this chapter:

- HTML forms and best practices.
- HTML collections and `Array.from()`.
- the storage API.
- more on classes.

Web pages are not only meant to display stuff. Thanks to HTML forms (they've been around for a long time) we can catch and manipulate user data. In this chapter you'll build a simple HTML form for capturing some notes. Later on you'll save that data to the browser. In the process you'll learn more about DOM events, introduced in chapter 8.

A re-introduction to HTML forms

A `<form>` element is an HTML element which in turn may contain other sub-elements like:

- `<input>` for capturing data
- `<textarea>` for capturing text
- `<button>` for sending the form

In this chapter you'll build a form containing some `<input>`, a `<textarea>` and a `<button>`. Ideally every input should have an attribute called `type` which expresses the input type: for example text, email, number, date and so on. In addition to `type` you may also want to add an `id` attribute to every form element. Later we'll see it turning useful for retrieving data from each input. Input and textarea may also have a `name` attribute. The `name` attribute is important if you want to send the form (with and) without JavaScript. More on that in a minute. It's also common practice associating every form element with a `<label>`. In the example below you'll see each label carrying a `for` attribute bond to the `id` attribute of the related input element. Sounds confusing? Don't worry.

Best practice: always provide a label

Always provide a label with the corresponding `for` attribute for any form input. It helps the user by providing context about the type of information she should insert.

Most of the times it's also a good idea to mark form inputs as required. A user won't be able to submit the form without filling all the required info. It's a simple validation for avoiding empty data, thus preventing the user from skipping important fields. With this knowledge you're now ready to create an HTML form. Create a new file named `form.html` and build the HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>HTML forms and JavaScript</title>
</head>
<body>
<h1>What's next?</h1>
<form>
    <label for="name">Name</label>
    <input type="text" id="name" name="name" required>

    <label for="description">Short description</label>
    <input type="text" id="description" name="description" required>

    <label for="task">Task</label>
    <textarea id="task" name="task" required></textarea>

    <button type="submit">Submit</button>
</form>
</body>
<script src="form.js"></script>
</html>
```

As you can see our form inputs have all the right attributes and from now on we can test the form by filling some data. When writing HTML forms pay particular attention to the `type` attribute (here's why[2]). HTML5 also introduced form validation: for example an input of type `email` will only accept e-mail addresses with an "at" sign `@` in it. Unfortunately that's the only check applied to an e-mail address: nobody prevents the user from typing an e-mail like a@a. It has the `@` but it's still invalid (the `pattern` attribute for email inputs can help to mitigate the issue).

Best practice: do not rely only on client-side validation

HTML5 validation must not be your one and only validation layer. A good designed system should not save data to the database until user's input is checked against multiple layers of validation, both on the client and the server.

[2] https://cloudfour.com/thinks/an-html-attribute-potentially-worth-4-4m-to-chipotle/

Before moving on let's tweak the form inputs. There are a lot of attributes available on `<input>`, and I find `minlength` and `maxlength` two of the most useful. In a real app they can stop lazy spammers from sending forms with "aa" or "testtest". A sane default for `minlength` could be 5, while the minimum task length might be 10 (your mileage may vary):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>HTML forms and JavaScript</title>
</head>
<body>
<h1>What's next?</h1>
<form>
    <label for="name">Name</label>
    <input type="text" id="name" name="name" required minlength="5">

    <label for="description">Short description</label>
    <input type="text" id="description" name="description" required minlength="5">

    <label for="task">Task</label>
    <textarea id="task" name="task" required minlength="10"></textarea>

    <button type="submit">Submit</button>
</form>
</body>
<script src="form.js"></script>
</html>
```

With the form in place we're ready to move further: let's see it in action.

HTML forms in action

Our HTML form is an element of type HTMLFormElement[3]. As with almost any HTML element it is connected to `HTMLElement`, which in turn is connected to `EventTarget`. Here's again the prototype in action. DOM elements are represented as JavaScript objects when we reach for them. Try this in a browser:

[3] https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement

```
const aForm = document.createElement("form");
console.log(typeof aForm);
```

The output is `"object"`, while entities like `HTMLElement` or `EventTarget` are functions because they're used as constructors for other objects:

```
console.log(typeof EventTarget); // "function"
```

So if any HTML element is connected to `EventTarget` it means that `<form>` is an "instance" of `EventTarget`? Try by yourself:

```
const aForm = document.createElement("form");
console.log(aForm instanceof EventTarget);
```

The result yields `true`. The form is a specialized kind of `EventTarget`. Every `EventTarget` can receive and react to DOM events (as seen on chapter 8). There are many types of DOM events like `click`, `blur`, `change`, and so on. What's interesting for you now is the `submit` event, distinctive of HTML forms. The `submit` event fires when the user clicks on an input or on a button of type submit (the element must appear inside the form). That's exactly our case:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>HTML forms and JavaScript</title>
</head>
<body>
<h1>What's next?</h1>
<form>
    <label for="name">Name</label>
    <input type="text" id="name" name="name" required minlength="5">

    <label for="description">Short description</label>
    <input type="text" id="description" name="description" required minlength="5">

    <label for="task">Task</label>
    <textarea id="task" name="task" required minlength="10"></textarea>

    <button type="submit">Submit</button>
</form>
</body>
<script src="form.js"></script>
</html>
```

Notice that `<button type="submit">Submit</button>` is right inside the form. Some developers use an input instead of a button:

```
<!-- I don't like it -->
<input type="submit">
```

Most of the times it depends on particular CSS constrains. In all honesty I don't like this approach. A button is fine for me:

```
<!-- yes please -->
<button type="submit">Submit</button>
```

Now let's get coding. Our goal is to intercept all the user input on the form but before doing so, we need to listen for the `submit` event. Again, we'll structure our code in a class. Class consumers should be able to call it as follows:

```
const formSelector = document.querySelector("form");
new Form(formSelector);
```

The DOM also offers `document.forms`, a collection of all forms inside the page. We need just the first (and the only):

```
const formSelector = document.forms[0];
new Form(formSelector);
```

Now here's the idea: given a form selector we can register an event listener on it for reacting when the form is sent. The logic can live in a class method named `init`. Create a new file named `form.js` in the same folder for `form.html` and start off with a simple class:

```
"use strict";

class Form {
  constructor(formSelector) {
    this.formSelector = formSelector;
    this.init();
  }

  init() {
    this.formSelector.addEventListener("submit", this.handleSubmit);
  }
}
```

Our event listener is `this.handleSubmit()` and as with every event listener it has access to a parameter called `event`. You should know from chapter 8 that `event` is the actual event, and has a lot of information about the action itself. Let's create `this.handleSubmit()`:

```
"use strict";

class Form {
  constructor(formSelector) {
    this.formSelector = formSelector;
    this.init();
  }

  init() {
    this.formSelector.addEventListener("submit", this.handleSubmit);
  }

  handleSubmit(event) {
    console.log(event);
  }
}
```

Our class is ready to take orders. We can "instantiate" it:

```
const formSelector = document.forms[0];
new Form(formSelector);
```

At this point you should be able to open up `form.html` in a browser. Fill the inputs and click on Submit. What happens? You shouldn't be surprised by the content of your url bar:

```
http://localhost:63342/little-javascript/code/ch10/form.html?name=Valentino&descript\
ion=Trip+to+Spoleto&tak=We%27re+going+to+visit+the+city%21
```

What's going on? Most DOM events have a so called *default behaviour*. The `submit` event in particular tries to send the form data to a fictitious server. That's how you send a form without JavaScript, when it is part of an application based on web frameworks like Django, Rails, and friends. Every input value is mapped to the corresponding `name` attribute. Since we want to control the form with JavaScript we need to disable the default behaviour. We can fix `handleSubmit()` by calling `preventDefault()`, a method of the Event[4] interface:

[4] https://developer.mozilla.org/en-US/docs/Web/API/Event

```
"use strict";

class Form {
  constructor(formSelector) {
    this.formSelector = formSelector;
    this.init();
  }

  init() {
    this.formSelector.addEventListener("submit", this.handleSubmit);
  }

  handleSubmit(event) {
    event.preventDefault();
    console.log(event);
  }
}

const formSelector = document.forms[0];
new Form(formSelector);
```

Save the file and refresh again `form.html`. Try to fill the form and click Submit. You'll see the event object printed to the console:

```
Event {...}
    bubbles: true
    cancelBubble: false
    cancelable: true
    composed: false
    currentTarget: null
    defaultPrevented: true
    eventPhase: 0
    isTrusted: true
    path: (5) [form, body, html, document, Window]
    returnValue: false
    srcElement: form
    target: form
    timeStamp: 8320.840000000317
    type: "submit"
```

Among the many properties on the event object there is also `event.target`, suggesting that our HTML form is saved there alongside with all the inputs. Let's see if that's really the case.

Extracting data from an HTML form

By the end of the chapter we want our code to persist data. For doing so, we need first to extract every input value from the form. By examining `event.target` you'll find out that there's a property named `elements`. Is what we're looking for? If you try to print it with `console.log(event.target.elements)` you'll see:

```
0: input#name
1: input#description
2: textarea#task
3: button
length: 4
description: input#description
name: input#name
tak: textarea#task
task: textarea#task
```

You can notice every form input in the collection. Now, we have three ways for accessing those inputs:

- array-like notation: `event.target.elements[0]`.
- with the id: `event.target.elements.some_id`.
- with `FormData` (more on this later).

In fact if you cared to add the appropriate `id` attribute on every form element now you can access the same element as `event.target.elements.some_id` where id is the string you assigned to the attribute. Since `event.target.elements` is first of all an object we can also use ECMAScript 2015 object destructuring:

```
const { name, description, task } = event.target.elements;
```

This isn't 100% recommended because you don't know in advance how many inputs there will be in the form, even more if the form is built dynamically. For now let's go this way. We can complete `handleSubmit()` and while we're at it let's also create another method named `saveData()`. It will print the values to the console (for now):

```
"use strict";

class Form {
  constructor(formSelector) {
    this.formSelector = formSelector;
    this.init();
  }

  init() {
    this.formSelector.addEventListener("submit", this.handleSubmit);
  }

  handleSubmit(event) {
    event.preventDefault();
    const { name, description, task } = event.target.elements;
    this.saveData({
      name: name.value,
      description: description.value,
      task: task.value
    });
  }

  saveData(payload) {
    console.log(payload);
  }
}

const formSelector = document.forms[0];
new Form(formSelector);
```

Wait a moment. It's not the smartest implementation ever. What if the fields change? Now we have name, description, and task, but we may add more inputs in the future. We need to extract those fields dynamically. Screw object destructuring for now and let's peek into `event.target.elements`:

```
0: input#name
1: input#description
2: textarea#task
3: button
length: 4
description: input#description
name: input#name
tak: textarea#task
task: textarea#task
```

Those numeric indexes look like an array. What if I could transform it into another array with name, description, and task (minus the button type submit)? Let's iterate over `event.target.elements` with the map method[5] (here's just the relevant bit of code, at this point you can also comment out `this.saveData()`):

```
  handleSubmit(event) {
    event.preventDefault();
    const inputList = event.target.elements.map(function(formInput) {
      if (formInput.type !== "submit") {
        return formInput.value;
      }
    });

    /*
      TODO this.saveData( maybe inputList ?)
     */
  }
```

Give it a shot in the browser and look at the console:

```
Uncaught TypeError: event.target.elements.map is not a function
    at HTMLFormElement.handleSubmit (form.js:15)
```

Bad news. So what's `event.target.elements` really? It seems like an array, yet it's a different beast: it is an `HTMLFormControlsCollection`. You got a sneak peek of it in chapter 8 where you saw that some DOM methods return an `HTMLCollection`.

```
// Returns an HTMLCollection
document.chidren;
```

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

HTML collections look similar to arrays, but they lack methods like `map()` or `filter()` for iterating over elements. It's still possible to access every element with the bracket notation but now we want to do things dynamically. ECMAScript 2015 added a method on the Array built-in object for creating an array from (almost) any source. It's `Array.from()`. Let's fix again `handleSubmit()`:

```
handleSubmit(event) {
  event.preventDefault();

  const arrOfElements = Array.from(event.target.elements);

  const inputList = arrOfElements.map(function(formInput) {
    if (formInput.type !== "submit") {
      return formInput.value;
    }
  });

  console.log(inputList);

  /*
    TODO this.saveData( maybe inputList ?)
   */
}
```

Here we construct an array from `event.target.elements` that gets passed to `map()`. But here's another gem. `Array.from()` accepts a mapping function as the second argument. We can do everything in a single step:

```
handleSubmit(event) {
  event.preventDefault();

  const inputList = Array.from(event.target.elements, function(formInput) {
    if (formInput.type !== "submit") {
      return formInput.value;
    }
  });

  console.log(inputList);

  /*
    TODO this.saveData( maybe inputList ?)
   */
}
```

Refresh `form.html`, fill the form and press Submit. You should see the following array in the console:

```
["Valentino", "Trip to Spoleto", "We're going to visit the city!", undefined]
```

As a finishing touch I'd like to generate an array of objects where every object has also the name property of the related form input:

```
  handleSubmit(event) {
    event.preventDefault();

    const inputList = Array.from(event.target.elements, function(formInput) {
      if (formInput.type !== "submit") {
        return {
          name: formInput.name,
          value: formInput.value
        };
      }
    });

    console.log(inputList);

    /*
      TODO this.saveData( maybe inputList ?)
     */
  }
```

Refresh `form.html` again, fill the form, and you should see:

```
[
  {
    "name": "name",
    "value": "Valentino"
  },
  {
    "name": "description",
    "value": "Trip to Spoleto"
  },
  {
    "name": "task",
    "value": "We're going to visit the city!"
  },
  undefined
]
```

Nice. There is an `undefined` though. It comes from the button element. `map()` returns `undefined` for empty values. Since we checked `if (formInput.type !== "submit")` the button element gets replaced by `undefined`. We can clean it up later: now let's turn our attention to `localStorage`.

Getting to know localStorage and laying out our classes

Sooner or later you will need to persist data for your users. There are many reasons to do so. Think of a note app for example. A user may insert new things to do into an HTML form and later come back to see those notes. Next time she opens the page she'll find everything there. What options do we have for saving data in the browser? A serious approach for persisting data would involve a database, but here we have just some HTML, JavaScript, and a browser. Yet not everything is lost. There is a built-in tool available in modern browsers, perfectly suited for our needs: localStorage[6]. `localStorage` behaves like a JavaScript object, and has a bunch of methods:

- `setItem()` for saving data
- `getItem()` for reading data
- `clear()` for cleaning up the "store"
- `removeItem()` for removing data

In a moment we'll see `setItem()` and `getItem()` in action for our simple "app", but first a quick recap.

Don't store sensitive data in localStorage

localStorage should be used for what it is: a super simple object for persisting user preferences, non-sensitive data, and nothing else. You might be tempted to save everything into it, like passwords or authentication credentials. If so please stop[7].

So here it is. We have an HTML form in `form.html` for inserting notes:

[6] https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Local_storage
[7] https://www.rdegges.com/2018/please-stop-using-local-storage/

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>HTML forms and JavaScript</title>
</head>
<body>
<h1>What's next?</h1>
<form>
    <label for="name">Name</label>
    <input type="text" id="name" name="name" required minlength="5">

    <label for="description">Short description</label>
    <input type="text" id="description" name="description" required minlength="5">

    <label for="task">Task</label>
    <textarea id="task" name="task" required minlength="10"></textarea>

    <button type="submit">Submit</button>
</form>
</body>
<script src="form.js"></script>
</html>
```

We also have a JavaScript class with two methods for intercepting a `submit` event and saving user's data (not implemented yet):

```
"use strict";

class Form {
  constructor(formSelector) {
    this.formSelector = formSelector;
    this.init();
  }

  init() {
    this.formSelector.addEventListener("submit", this.handleSubmit);
  }

  handleSubmit(event) {
    event.preventDefault();

    const inputList = Array.from(event.target.elements, function(formInput) {
```

```
      if (formInput.type !== "submit") {
        return {
          name: formInput.name,
          value: formInput.value
        };
      }
    });

    console.log(inputList);

    /*
      TODO this.saveData( maybe inputList ?)
     */
  }

  saveData(payload) {
    console.log(payload);
  }
}

const formSelector = document.forms[0];
new Form(formSelector);
```

At this point we need to complete `this.saveData()` for saving every note to `localStorage`. While doing so, we'll need to stay as generic as possible. In other words I don't want to fill `this.saveData()` with the logic for saving directly to localStorage. Instead, we'll provide the `Form` class with an external dependency (another class) whose role is interacting with the storage. It doesn't matter whether we'll persist the notes to `localStorage` or to a REST API in the future. We should be able to give `Form` a different storage for every use case, switching from one to another as the requirements change. Let's first adjust `constructor` for accepting a new `storage` argument:

```
class Form {
  constructor(formSelector, storage) {
    this.formSelector = formSelector;
    this.storage = storage; // accept the storage
    this.init();
  }

  // omit
}
```

Dependency injection in action

When we pass a so called *volatile dependency* to a class in a `constructor`, we're effectively applying *constructor injection*, a subset of *dependency injection*. Here the storage is volatile because it can be replaced in the future by something different from `localStorage`. By not tying our `Form` class to a particular type of storage we are free to swap it even at runtime. This lays the ground for maintainable, loosely-coupled code.

Now, as the class starts growing up so the need for validating its arguments. Being a class designed for working with HTML forms we need at least to check that `formSelector` is an HTML element of type form. Here's how:

```
constructor(formSelector, storage) {
  // Validating the form
  if (!(formSelector instanceof HTMLFormElement))
    throw Error(`Expected a form element got something else!`);
  //
  this.formSelector = formSelector;
  this.storage = storage;
  this.init();
}
```

This will make the code fail if we pass something else. I'd also validate `storage` because we have to send the user input somewhere:

```
constructor(formSelector, storage) {
  // Validating the form
  if (!(formSelector instanceof HTMLFormElement))
    throw Error(`Expected a form element got something else!`);
  // Validating the storage
  if (!storage) throw Error(`Expected a storage got something else!`);
  //
  this.formSelector = formSelector;
  this.storage = storage;
  this.init();
}
```

Checking the invariants

Here we're checking the so called *invariants*, i.e. things that belong to the class and should be present when we use the class itself. Notice the usage of `throw Error("message")` for raising a custom error.

The storage implementation will be another class. In our case could be something like a generic `LocalStorage`. In `form.js` create a skeleton for the class:

```
class LocalStorage {
  save() {
    return "saveStuff";
  }

  get() {
    return "getStuff";
  }
}
```

With this structure in place we can wire up `Form` and `LocalStorage`. The idea is that a colleague could use the class `Form` by passing in still a form, plus a custom storage. For the storage I'd like to keep things as clean as possible, avoiding any call to `new`. Here's the plan:

```
const formSelector = document.forms[0];
const storage = LocalStorage;

new Form(formSelector, storage);
```

Notice how storage is passed as `LocalStorage`, not as `new LocalStorage()`. To use this form we can take advantage of static methods for classes. Earlier in the chapter we used `Array.from()` for creating an array from an HTML collection. We didn't call `const arr = new Array()` followed by `arr.from()`. `Array.from()` is used without any need to "instantiate" the class. Static methods are convenient for utility functions that don't need an instance to work on. So at this point our `LocalStorage` class becomes:

```
/*
Storage implementation
 */
class LocalStorage {
  static save() {
    console.log("saveStuff");
  }

  static get() {
    console.log("getStuff");
  }
}
```

Now `saveData()` in `Form` should call `this.storage.save()` and `handleSubmit()` could call `this.saveData()`. In `form.js` adjust the methods like so:

```
"use strict";

/*
Form implementation
 */
class Form {

// omit

  handleSubmit(event) {
    event.preventDefault();

    const inputList = Array.from(event.target.elements, function(formInput) {
      if (formInput.type !== "submit") {
        return {
          name: formInput.name,
          value: formInput.value
        };
      }
    });

    // now you can really save the data!
    this.saveData(inputList)
  }

  saveData(payload) {
    // uses the new storage
    this.storage.save(payload);
  }
}
```

Well done. In the next section we'll finally make our data persistent.

Interacting with localStorage and this comes to bite again

Time for a smoke test. Refresh `form.html` in your browser, open up the console and try to submit the form with some data. See what happens:

```
TypeError: this.saveData is not a function form.js:35:10
```

`this` comes to bite us again! If you think about it `handleSubmit()` is running as an event handler for the form. That could be the culprit. In chapter 9 you used `debugger` for stopping the script and inspecting `this`. Now you can do the same, this time without polluting your code. Go in your browser console, open up the Sources tab, find `form.js` and click on the line where you see `this.saveData(inputList)`. Refresh the page, send again the form and take a look at the Scopes tab. You'll see `this` pointing to the form, not to the class instance. Having seen a similar case in chapter 9 we can easily reach for a fix in `init()`. Arrow functions to the rescue:

```
// omit
  init() {
    this.formSelector.addEventListener("submit", event =>
      this.handleSubmit(event)
    );
  }
// omit
```

Passing `this.handleSubmit()` inside an arrow function makes sure that `this` will point to the class instance, not to the event target. Here's the complete code so far:

```
"use strict";

/*
Form implementation
 */
class Form {
  constructor(formSelector, storage) {
    if (!(formSelector instanceof HTMLFormElement))
      throw Error(`Expected a form element got something else!`);
    if (!storage) throw Error(`Expected a storage got something else!`);

    this.formSelector = formSelector;
    this.storage = storage;
    this.init();
  }

  init() {
    this.formSelector.addEventListener("submit", event =>
      this.handleSubmit(event)
    );
  }

  handleSubmit(event) {
    event.preventDefault();
```

```
    const inputList = Array.from(event.target.elements, function(formInput) {
      if (formInput.type !== "submit") {
        return {
          name: formInput.name,
          value: formInput.value
        };
      }
    });

    this.saveData(inputList);
  }

  saveData(payload) {
    this.storage.save(payload);
  }
}

/*
Storage implementation
 */
class LocalStorage {
  static save() {
    console.log("saveStuff");
  }

  static get() {
    console.log("getStuff");
  }
}

const formSelector = document.forms[0];
const storage = LocalStorage;

new Form(formSelector, storage);
```

Now we're going to work on the `LocalStorage` class because it'll be our adapter for the real `localStorage` which has four methods:

- `setItem()` for saving data
- `getItem()` for reading data
- `clear()` for cleaning up the cache
- `removeItem()` for removing data

For now, we'll implement just `setItem()`. It takes two arguments, a key and a value:

```
localStorage.setItem("identifier", "someValue");
```

Think of the key as of an identifier for a fictitious database table, while the value is the actual data to store. `getItem()` works the other way around. It takes a key and returns the data:

```
localStorage.getItem("identifier");
```

So far we had a minimal implementation for the class `LocalStorage`:

```
class LocalStorage {
  static save() {
    console.log("saveStuff");
  }

  static get() {
    console.log('getStuff')
  }
}
```

We want to get serious and finally save some data in the browser. Open up `form.js` and improve `LocalStorage.save()` to save to the real storage:

```
class LocalStorage {
  static save(key, payload) {
    localStorage.setItem(key, payload);
  }

  static get() {
    // TODO
  }
}
```

The method takes a key, and a payload that gets saved to `localStorage`. You should have noticed that `saveData()` in `Form` does not pass any key to the storage:

```
  // saveData from "Form"
  // omit
  saveData(payload) {
    this.storage.save(payload);
  }
  // omit
```

We could make a key from the value property of the first input element. Let's do it:

```
  // saveData from "Form"
  // omit
  saveData(payload) {
    const key = `${payload[0].value} task list`;
    this.storage.save(key, payload);
  }
  // omit
```

Here's the complete code:

```
"use strict";

/*
Form implementation
 */
class Form {
  constructor(formSelector, storage) {
    if (!(formSelector instanceof HTMLFormElement))
      throw Error(`Expected a form element got something else!`);
    if (!storage) throw Error(`Expected a storage got something else!`);

    this.formSelector = formSelector;
    this.storage = storage;
    this.init();
  }

  init() {
    this.formSelector.addEventListener("submit", event =>
      this.handleSubmit(event)
    );
  }

  handleSubmit(event) {
```

```
    event.preventDefault();

    const inputList = Array.from(event.target.elements, function(formInput) {
      if (formInput.type !== "submit") {
        return {
          name: formInput.name,
          value: formInput.value
        };
      }
    });

    this.saveData(inputList);
  }

  saveData(payload) {
    const key = `${payload[0].value} task list`;
    this.storage.save(key, payload);
  }
}

/*
Storage implementation
 */
class LocalStorage {
  static save(key, payload) {
    localStorage.setItem(key, payload);
  }

  static get() {
    // TODO
  }
}

const formSelector = document.forms[0];
const storage = LocalStorage;

new Form(formSelector, storage);
```

Curious to see what we've got? Refresh `form.html`, submit the form and open up the browser console. Type `localStorage` and you'll see something along these lines:

```
{
  "Valentino task list": "[object Object],[object Object],[object Object],"
}
```

Yikees! `localStorage` does not care and converts everything to string even if the payload is clearly an array here:

```
    // handleSubmit from "Form"
    // omit
    const inputList = Array.from(event.target.elements, function(formInput) {
      if (formInput.type !== "submit") {
        return {
          name: formInput.name,
          value: formInput.value
        };
      }
    });
    // omit
```

You can already see all of `localStorage` limitations and why it should be treated for what it is: a primitive utility cache for storing simple strings. As a quick fix we can check whether the payload is a string in `LocalStorage.save()`:

```
class LocalStorage {
  static save(key, payload) {
    if (typeof payload !== "string") {
      // convert to string!
    }
    localStorage.setItem(key, payload);
  }

  static get() {
    // TODO
  }
}
```

If it's not we can convert the payload itself with `JSON.stringify()`, pulling out the logic into another static method:

```
class LocalStorage {
  static save(key, payload) {
    let _payload = payload;
    if (typeof payload !== "string") {
      _payload = this.serialize(payload);
    }
    localStorage.setItem(key, _payload);
  }

  static get() {
    // TODO
  }

  static serialize(payload) {
    return JSON.stringify(payload);
  }
}
```

The more you know

Converting an object or any other complex data to a string goes under the name of serialization[8].

With this fix you should now be able to send the form and save to `localStorage`. You should see more or less the following data if you type `localStorage` in the console:

```
{
  "Valentino task list": "[{\"name\":\"name\",\"value\":\"Valentino\"},{\"name\":\"d\
escription\",\"value\":\"Trip to Spoleto\"},{\"name\":\"task\",\"value\":\"We're goi\
ng to visit the city!\"},null]"
}
```

Great job! A nice thing to do now? Cleaning up the form after the user saves the note. Be aware, `localStorage` is a synchronous API: it could block the browser. Being synchronous in JavaScript means also that `try/catch` works fine for intercepting errors. Right now we want to clean up the form only if the save operation ends well, in `saveData()`:

[8] https://en.wikipedia.org/wiki/Serialization

```
  // saveData from "Form"
  // omit
  saveData(payload) {
    const key = `${payload[0].value} task list`;
    this.storage.save(key, payload);
    // clean the form if save is ok
  }
  // omit
```

HTML forms have a convenient method[9] called `reset()` for resetting every input. We should be good by wrapping `this.storage.save(key, payload)` in `try/catch` while calling `reset()` on the form selector inside our class:

```
  // saveData from "Form"
  // omit
  saveData(payload) {
    const key = `${payload[0].value} task list`;
    try {
      this.storage.save(key, payload);
      this.formSelector.reset();
    } catch (error) {
      console.log(error);
    }
  }
  // omit
```

This way we can be sure to reset the form only if the save action goes well. If you followed every step you should have the complete code:

```
class LocalStorage {
  static save(key, payload) {
    let _payload = payload;
    if (typeof payload !== "string") {
      _payload = this.serialize(payload);
    }
    localStorage.setItem(key, _payload);
  }
  static get() {
    return "getStuff";
  }
  static serialize(payload) {
```

[9] https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement/reset

```
    return JSON.stringify(payload);
  }
}

class Form {
  constructor(formSelector, storage) {
    if (!(formSelector instanceof HTMLFormElement))
      throw Error(`Expected a form element got something else!`);
    if (!storage) throw Error(`Expected a storage got something else!`);

    this.formSelector = formSelector;
    this.storage = storage;
    this.init();
  }

  init() {
    this.formSelector.addEventListener("submit", event =>
      this.handleSubmit(event)
    );
  }

  handleSubmit(event) {
    event.preventDefault();
    const inputList = Array.from(event.target.elements, function(formInput) {
      if (formInput.type !== "submit") {
        return {
          name: formInput.name,
          value: formInput.value
        };
      }
    });

    this.saveData(inputList);
  }

  saveData(payload) {
    const key = `${payload[0].value} task list`;
    try {
      this.storage.save(key, payload);
      this.formSelector.reset();
    } catch (error) {
      console.log(error);
    }
```

```
  }
}

const formSelector = document.forms[0];
const storage = LocalStorage;

new Form(formSelector, storage);
```

Refresh `form.html`, fill the inputs and click submit. The form will reset, and the data lands in `localStorage`. To check it out you can finally access the storage with:

```
localStorage.getItem("Valentino task list")
```

The call will return your data:

```
"[{\"name\":\"name\",\"value\":\"Valentino\"},{\"name\":\"description\",\"value\":\"\
Trip to Spoleto\"},{\"name\":\"task\",\"value\":\"We're going to visit the city!\"},\
null]"
```

As you can see there is still a `null` value hanging around. Time to get rid of it.

Going idiomatic with FormData

If your house has an entrance door would you rather dig a tunnel to get in, or would you use the door? This analogy can help you understand what it means to program idiomatically. Don't be like me, don't reinvent the wheel! That's exactly what we did in `handleSubmit()`:

```
  handleSubmit(event) {
    event.preventDefault();
    const inputList = Array.from(event.target.elements, function(formInput) {
      if (formInput.type !== "submit") {
        return {
          name: formInput.name,
          value: formInput.value
        };
      }
    });

    this.saveData(inputList);
  }
```

In an attempt to help ourselves we created an abomination. Idiomatic means using a native language that is common to the speakers. In programming means using the native tools for the job that are common to developers working with that particular language. Going idiomatic in this case means reaching for `FormData`. It is a constructor first of all, and when called with a form element as the argument it is able to create a new `FormData` object. FormData object are sets of key/value pairs containing every form's field alongside with their values. Exactly what we're looking for at the beginning. Let's refactor then:

```
handleSubmit(event) {
  event.preventDefault();
  const inputList = new FormData(event.target);
  this.saveData(inputList);
}
```

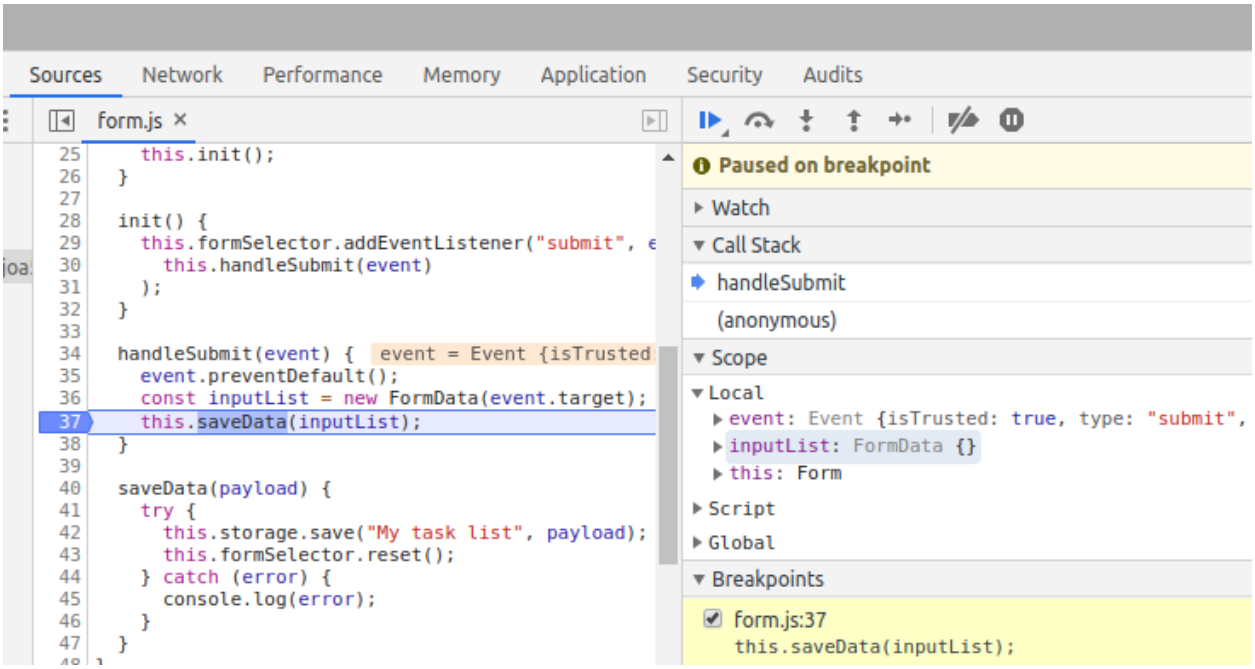
See how much code we spared? It looks cleaner than ever. A word of warning: `FormData` relies on form fields name attributes to build the mapping between fields and values. That means the following elements won't yield anything:

```
<!-- bad -->
<input type="text" id="name" required>
<input type="text" id="description" required>
```

Always provide a name for your fields:

```
<!-- good -->
<input type="text" id="name" name="name" required>
<input type="text" id="description" name="description" required>
```

If you want to snitch into a `FormData` object place a breakpoint on `this.saveData()`, submit the form, and from the debugger save the variable to the console (example for Chrome):



Explore formData

At this point to get the values from `FormData` do:

```
[...temp1.values()]
```

In the console you should see:

```
["Valentino", "Trip to Spoleto", "We're going to visit the city!"]
```

(or whatever you filled the form with). To get back an array of arrays you can do:

```
[...temp1.entries()]
```

You should see an array of pairs where every pair is a key/value for the given form input. At this point it's up to you to save either one to the storage. I'll go with entries:

```
handleSubmit(event) {
  event.preventDefault();
  const formData = new FormData(event.target);
  const inputList = [...formData.entries()];
  this.saveData(inputList);
}
```

The only downside of this change is that we need to adjust `saveData()` as well. We don't have access to `payload[0].value` anymore, so change the key to something else:

```
  saveData(payload) {
    try {
      this.storage.save("My task list", payload);
      this.formSelector.reset();
    } catch (error) {
      console.log(error);
    }
  }
```

Refresh `form.html`, fill the inputs and click submit. Check it out the storage with:

```
localStorage.getItem("My task list")
```

You should see:

```
"[[\"name\",\"Valentino\"],[\"description\",\"Trip to Spoleto\"],[\"task\",\"We're g\
oing to visit the city!\"]]"
```

Well done again! As a bonus you can also pass the form selector to `FormData` in `handleSubmit()`:

```
  handleSubmit(event) {
    event.preventDefault();
    const formData = new FormData(this.formSelector);
    const inputList = [...formData.entries()];
    this.saveData(inputList);
  }
```

It's another style that does not involve dealing directly with `event.target`.

The formdata event

The `formdata` event is a newer, nice addition to the web platform. As a boost to `FormData` the event fires any time you call `new FormData()` inside a submit handler. Consider the following example:

```
const form = document.forms[0];

form.addEventListener("submit", event => {
  event.preventDefault();
  const formData = new FormData(form);
  // or
  // const formData = new FormData(event.target);
});

form.addEventListener("formdata", event => {
  // event.formData grabs the data
  const values = [...event.formData.values()];
  console.log(values);
});
```

In the first event listener we build a new `FormData` from the form. The result to this call the new object fires up the `formdata` event, on which we registered another listener. In this second listener we can grab the actual data from `event.formData`. This pattern helps decoupling event listeners from any other callback that was supposed to handle the actual form data. Being a newer addition to the platform `formdata` is not available in older browsers, use it with caution.

Conclusions

- Always apply best practice to form inputs: use labels and the appropriate attributes.
- Use `localStorage` sparingly to save only non-sensitive, serializable data in the browser.
- Always use `FormData` to extract values from form inputs.

Sharpen up your JavaScript skills

- What's the default behaviour for HTML forms? How would you disable it?
- You're working with a class that's giving you `Uncaught TypeError: this.saveUser is not a function`. What do you do?
- Try to refactor the class to intercept the `formdata` event.
- We saved the data to localStorage. Now implement `LocalStorage.get` for getting back the original array, and as a bonus display it to the user.