The Single Page App Jumpstart

Understanding and Mastering JavaScript Applications

Jan Krutisch

The Single Page App Jumpstart

Understanding and Mastering JavaScript Applications

Jan Krutisch

This book is for sale at http://leanpub.com/tspa_jumpstart

This version was published on 2015-02-20

Tweet This Book!

Please help Jan Krutisch by spreading the word about this book on Twitter!

The suggested hashtag for this book is #tspa_jumpstart.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#tspa_jumpstart

Contents

Introduction

Thanks for downloading this book sample. I've tried to keep the sample interesting by giving you the first three chapters plus one of the closing chapters and I certainly hope this works. If you would like to buy this book, you can choose any price between currently a little over 8 USD and any amount you find suitable. And I mean this: Please choose what ever price you think this is worth and you can afford!

In any case I am very grateful for your support.

If you're done reading the samples (or even the whole book), I would like to ask you for your honest feedback. The easiest way of doing that is simply write to me at feedback@thesinglepageapp.com[1]. I am also currently working on better ways of sharing your feedback while the book progresses, so watch @singlepageapp[2] on twitter, the google+ page[3] or the facebook page[4] or the web page[5] for updates.

Thanks,

Jan

About the code samples

Having a lot of code samples in a book (especially in an eBook) is not unproblematic. Currently the book contains quite a lot of code and since I am publishing this with LeanPub, I don't have a whole lot of control over the formatting. Still I think it is important to be able to read the book without having to peek at a different source every few lines. I would love to hear from you if this assumption is actually correct.

Nevertheless, you can find all the code examples in running form and as a zip file to download at code.thesinglepageapp.com[6]. At the end of each chapter, I'll provide a link to the full result of the chapters discussions.

If you are following the chapters and the associated code examples with a critical mind you will find that the code evolves quite naturally with some stupid mistakes built in in the first chapters. This is by design: First of all this book is a work in progress and so is the code. And as usual as when I'm developing code, I'm learning things on the way. And second all of these little "mistakes" I make on the way are mistakes you might make as well and so they give me a good place to explain how to do things better, although they sometimes seem to make the author look stupid.

[1] mailto:feedback@thesinglepageapp.com

[2] http://twitter.com/singlepageapp

[3] https://plus.google.com/b/116966206897994963341/116966206897994963341/posts

[4] https://www.facebook.com/thesinglepageapp

[5] http://thesinglepageapp.com

[6] http://code.thesinglepageapp.com/jumpstart/

Let’s roll

Well, actually, let’s not roll. Let’s have a quick look at what it means to build a single page web app:

- Most of the application logic will reside in the client, instead of the server, as we are used to as web developers.
- Most of the application logic will be written in JavaScript. You better get used to that.
- Your application usually persists data to a server via a web API, which ideally is based around REST principles and speaks JSON.
- Your application looks and feels more like a desktop or “native” app.
- Things will happen asynchronously and based on Events instead of synchronously and based on requests and responses. You better get used to that.
- Everything will be more complex than before. And also more fun.

Okay, now let’s really start. What are the most basic ingredients of the most basic single page app?

- Some HTML. This can be only some meta information, script tags for the code and an empty body, but delivering at least one page of handwritten or generated HTML is unavoidable.
- Some JavaScript. Usually some **more** JavaScript.
- There is no step three. Usually, you will, of course, write some CSS, but it’s certainly possible to write apps without that. Not very beautiful and functional apps, but still.
- No, you don’t actually need a server component. You might want to have a web server to actually host your app, but if you’re okay with your app only saving data in the browser, you don’t need an API on the server. And about that web server part? How about keeping your app in a Dropbox folder? Then you can serve the app via Dropbox’ public sharing. Amazing, right?

A kingdom for an idea!

So. Let’s build an app. What kind of app? Because nobody has done that before, let’s build a todo-list app. Well, the real reason for building this kind of app is that you can take the code we will have written through the course of this book and compare it to all of the other single page apps that Addy Osmani and his armies have collected at the excellent http://addyosmani.github.com/todomvc/[TodoMVC] project and see how you would approach this problem in one of the various libraries and frameworks for which a port of TodoMVC exists.

So, here’s a single page (literally) skeleton for our little app. No real markup so far, no code:

```html
<!doctype html>
<html>
  <head>
    <title>The Single Page App - Todolist</title>
  </head>
  <body>
    <!-- markup goes here -->
    <script type="text/javascript">
      // code goes here
    </script>
  </body>
</html>
```

Just take that and throw it into a folder of your choice as index.html. You can then open in in a browser of your choice (All screenshots in this book will be made with Chrome. It simply has the best debugging and inspection tools at the moment) and what you will see will be - totally unexpected of course, a blank page.

Filling in the blanks

What we are going to do now, for the course of this book, is first writing this very simple application in raw JavaScript, which will explain a lot of the concepts of single page apps on the way, without confusing you with aspects of external libraries, and then, gradually, use some libraries to make things easier and to abstract some of the core concepts.

This todo application should be as simple as possible. Here's the feature list we're going to implement:

- Adding a todo list item (that will just be a simple, one line string)
- Marking todo list items as done
- Editing a todo list item
- Removing a todo list item
- Marking all todo list items as done at once
- Removing all completed todo list items
- Showing only completed todo list items
- Showing only open todo list itesm

(For the sake of simplicity, we'll skip the view filtering (last two points) for now and implement them later on, when we have some more help at hand)

So, let's start with the first feature. It's probably pretty safe to assume that we need an input field for that. So, let's add that to our empty markup. I have taken the liberties of adding an id attribute to the tag, because we’ll need that pretty soon. Also, modern browsers interpret the placeholder attribute and display the text in grey as long as the input field is empty. When building a simple UI like our todo list, this frees us from adding an explicit label for the input field. Lastly, I added the autofocus attribute which gives us a nice demanding blinking cursor as soon as we load the page.

```html
<input id="new-todo" placeholder="What needs to be done?" autofocus>
```

If you reload the page, the textfield is there, a bit forlorn, but nicely active and waiting for the user input.

Entering data

In a traditional web app, we now would add a button and then a server action that would take the input field data and build a database object with it to persist it in, say, MySQL. Now that we don’t have a server, we need to find a different way. First of all, we need to find out that the user (us!) actually finished entering the todo. There are tons of ways to do that, but since we want to keep the UI on the minimal side, the traditional way of adding a submit button and then intercepting the form submission is not an option. Instead, the todo should be added as soon as the user presses his Enter key.

So, let’s write some code:

```javascript
window.addEventListener('load', windowLoadHandler, false);

function newTodoKeyPressHandler(event) {
  if (event.keyCode === 13) {
    alert("We have a new Todo!");
  }
}
function windowLoadHandler() {
  document.getElementById('new-todo').addEventListener(
    'keypress',
    newTodoKeyPressHandler,
    false
  );
}
```

If you reload the page, you’ll notice that as soon as you press Enter (or Return), an alert pops up. That’s of course not what should really happen, but at least something happened, right?

So, the first thing is that we want to execute our code only when the HTML is finished with loading, so that we can actually access all the elements (in our case: the input field). This is a good rule of thumb for most things, but if you really want to build super fast JavaScript apps, this might be way to late. Please ignore that for now, again, for the sake of simplicity.

To do that, we install an event listener on the window object. Since the window object is actually present at all time, you can do it at any given point in time (or, in this case “point in the file”). We subscribe to the load event, which will fire as soon as the browser has loaded the complete HTML and other resources like images or external stylesheets etc. The function `windowLoadHandler` gets attached to that event by using the function name in the addEventListener call.

Mr. Penguin Asks

Dear Author, I am wondering if the three equal signs are actually a typo. I know I should not meddle with your code examples, but it looks strange to me.

Dear Mr. Penguin: No, that’s actually correct. Javascript has two different ways of comparing objects, and only the three-equals === usually does what you want. There’s also a version of not-equal which is `!==`. So while == would try to coerce different types into something it can actually compare, the === operator does no type conversion. Quick example: `1 == \'1\'` is `true`, `1 === \'1'` is `false`.

The DOM song

In that function, we then use `document.getElementById`, which gives us back a so-called DOM node, at least if an element with the id exists. The DOM, or Document Object Model, is the way the browser exposes the tree of HTML tags it has rendered in its window to JavaScript. If you use the `getElementById` function or any other of the DOM query methods (we’ll look at some of them later on), you’ll get back such a node object and you can do all kinds of things with them, like changing the styles, adding attributes, or, as in our case, install another event handler. As we want to detect the Return Key, we’ll listen to “keypress” events. Keyboard events are pretty complicated, they are not entirely equal in the various browsers, but in our case it’s pretty straight forward.

So now we need a second function that handles the `keypress` event. And while we ignored the event object that came with the “load” event, we now definitely need that object. The `keycode` attribute of that event gives us the actual key that has been pressed. And although these key codes are not completely the same on every browser, the Return key fortunately has the same key code everywhere. It’s 13. If you are wondering, it’s the ASCII-Code for CR, or Carriage Return, which in turn kind of is an awkward reference to typewriter technology and has, unfortunately, nothing to do with horses.

Almost there

There are two important things left to do here: First of all, we need to find out what the user has entered. Second, we need to do *something* with it.

Getting it should be easy. The event carries a `target` property that should give you the input field. From there, you can grab the `value` property. Only, that’s probably not the best way to do it. Events are tricky beasts and sometimes `.target` is not accurate. This will probably never happen with our example, but to be on the safe side, let’s simply use `getElementById` and fetch a known-to-be-accurate version of the input field and read the value from there.

The second thing is more tricky of course, because we now need a concept for creating, storing and retrieving todo items. Let’s keep that for the next chapter, shall we?

Keeping the lid on the scope

There’s one little tiny small thing I would like to change, though: Protecting the global scope. It’s one of JavaScripts dirty little corners and it always hurts to have to do this, but let’s just get over it really quickly and I won’t mention it again but simply assume you know why and how I’m doing it.

Thing is: Everything you do in a naked script tag runs in the context of the before mentioned window object. Which also means that the functions we just wrote are now properties of the window object. If you don’t believe me, check in Chrome’s JavaScript console (Use the Menu to open it: Display > Developer > JavaScript console). Enter window.window and see that the console autocompletion already shows our windowLoadHandler.

Now think for a second what would happen if everybody would do it like this. Not only does this literally pollute the window (or “global”) scope, which might seem cosmetic, but it is also dangerous. What if another script running in your page also wants to install a windowLoadHandler?

There’s only one way around it and that’s encapsulating your code in a somewhat strange construct, the IIFE (you can call it Iffy if you must), or Immediately Invoked Function Expression.

```
(function(){
  // your code
}());
```

Why does this help? It declares an anonymous function (it doesn’t have a name) and after that immediately calls that function, which means that the code you’ve encapsulated in that function is executed. The difference is only in scope: All your code suddenly runs within the anonymous function’s scope and you are not leaking stuff into the global scope by accident. In case you’re wondering: The outer parentheses are not strictly necessary. They are more or less a signal to say: “Look, it’s an iffy”. We will come back to this later on, as you can do some cool stuff with this encapsulation but you also still need to be careful with your code not to leak into the global scope, but for now, it's enough to know that window is safe now.

So here's the final JavaScript code:

```
(function() {
  window.addEventListener('load', windowLoadHandler, false);
  function newTodoKeyPressHandler(event) {
    if (event.keyCode === 13) {
      var todo = document.getElementById('new-todo').value;
      alert("We have a new Todo: " + todo);
    }
  }
  function windowLoadHandler() {
    document.getElementById('new-todo').addEventListener(
      'keypress',
      newTodoKeyPressHandler,
      false
    );
  }
}());
```

The complete example can be found at code.thesinglepageapp.com[7].

[7]http://code.thesinglepageapp.com/jumpstart/02_starting_the_app/index.html

Having a list

So, what should happen next? Our code now stands there, with a text for a todo item at hand, but has no idea what to do with it. So, what would be the immediate next steps?

- The new item should be added to our list of todo items
- The input field should be wiped so that we can add the next item

Now, the first step might actually be two steps: Saving the item somewhere for later and adding the item to an actual list in the HTML. Let's start with the second part. Where is that list? Let's add it to the HTML, just below the input field:

```
<input id="new-todo" placeholder="What needs to be done?" autofocus>
<ul id="todo-list"></ul>
```

innerHTML inside

Since we have committed ourselves to not using any library, we are now bound to build that todo-list item by hand. There are two variants of this, one simple, one complex. Let's start with the simple version:

```
if (event.keyCode === 13) {
  var todo = document.getElementById('new-todo').value;
  var list = document.getElementById('todo-list');
  list.innerHTML += ("<li>" + todo  + "</li>");
}
```

`innerHTML` allows you to replace or modify everything within the given tag as if it was just only a big string. It works, but it has some drawbacks: For example, if you had any event handlers on the other list items, they would now be gone, since our line essentially is the same as

```
list.innerHTML = list.innerHTML + ("<li>" + todo + "</li>")
```

As you can see, the action happens in two steps: first, all DOM elements inside the <ul> are converted to a string representation, then we append our stuff and then everything is converted back to HTML. Since event handlers are JavaScript only and are not expressed in the HTML that innerHTML gives us, they get lost in the process.

innerHTML is not completely useless, though. If you quickly want to add or replace some text in a single element, it is often the best choice.

Nodes inside of nodes inside of…

So let’s try the more complex way: creating a list node by hand. It’s not that hard:

```
var list = document.getElementById('todo-list');
var item = document.createElement("li");
item.appendChild(document.createTextNode(todo));
list.appendChild(item);
```

We are using three different methods here, so let me quickly explain them:

- `document.createElement` creates a DOM node of the given type (so “li” creates a list item element, “div” would create a huge Stegosaurus, you get the idea). The element only exists in JavaScript-land so far, but apart from that, you can do everything with it you could do with a node you got via `document.getElementById`, for example.
- `node.appendChild` appends a given element to the node.
- `document.createTextNode` seems a little weird, but to understand it you only have to realise that text inside a tag (for example the foo in `<b>foo</b>`) is also represented as a node in the DOM, a so called text node. So to actually be able to insert text into a tag programmatically, you create a text node and insert it. (The clever reader might suggest using innerHTML for that - We’ll get to the differences in a minute)

That looks quite okay so far. Oh, one last thing to make it work: we should clear the input field. And since this is the second time we would need a reference to the input field, we should cache the getElementById call in a variable to make the code a little shorter:

```
var todoField = document.getElementById('new-todo');
var list = document.getElementById('todo-list');
var item = document.createElement("li");
item.appendChild(document.createTextNode(todoField.value));
list.appendChild(item);
todoField.value = "";
```

This *caching* is something you’ll see quite a bit when reading JavaScript code. Accessing the DOM is not only a verbose operation, but also a slow operation. And with slow I mean orders of magnitudes slower than normal JavaScript code. We will come back to that subject shortly.

Security is King

Pretty good so far, but don't hit the reload button - currently our todos are only "persisted" in HTML. But before I tell you why this is usually a bad idea and how we should restructure the code to fix that, I promised you to tell you a little something about the difference between `createTextNode()` and `innerHTML`. The biggest difference is that createTextNode creates, ba-dumm-tish, text nodes, while innerHTML, as the name suggests, can also contain HTML. You can try this yourself: Try to add a todo list item like `<b>foo</b>` into both versions. In the `innerHTML` version, you'll end up with a bold foo, while you'll get your original input value in the `createTextNode` variant.

There are, obviously, upsides and downsides to both approaches. The biggest upside of the `createTextNode()` example? It's **super safe**. You may have heard about so called cross site scripting issues: When it's possible to insert active components (such as script tags) by user input into a web page, this enables bad people to spy on the users or tricking them into exposing credentials or other malicious deeds.

With `createTextNode`, it's simply not possible to create these active components. With innerHTML though, the situation is a little more complicated. If you try to insert a script tag using `innerHTML`, in most (if not all) browsers, you'll see that the script is not executed (good!) but the script tag is actually inserted into the DOM as is. That can be a problem later on. In the end, this is one of these places where the good web developer plays the "better safe than sorry" move, but of course there are situations where a text node is not enough: If you actually want people to be able to enter HTML (for example inside a comment field on a blog), you can't simply use `createTextNode`, but you still have to make sure that you are not letting people input arbitrary stuff that could potentially be dangerous.

Identifying the view layer

So far, we have now stored our todo items in the DOM. Apart from the fact that this currently doesn't survive reloads, it also has some more problems, that are, in the beginning, quite hard to grasp. As I said: Accessing the DOM is slow. Even worse: Intertwining reads and writes to the DOM is even slower. This has to do with the way browsers work internally. Browser vendors like the Google Chrome team and the Firefox team are of course aware of that and did a lot of work on this during the last few years, so the situation is not as dire as it used to be anymore, but the basic rule of JavaScript development still applies: Don't access the DOM (unless you really, really have to).

Looking at it from a more abstract view point, this means that we should treat the DOM more like a terminal: It's responsible for displaying stuff and handling user input, but it should not be responsible for data storage.

Okay, so let's build a data storage. Since we're only dealing with lists of strings here, let's start with a basic Array that will just hold the Strings of the items. Since items might have other qualities as well (they can be marked as finished, for example), this might not be enough, but it's the simplest thing that will work right now.

Also, let's decouple the display from the storage and redraw the whole list every time we modify it. This might sound awkward and inefficient, but for a relatively basic view like this, that hardly matters. Also, if we want to persist stuff later on, we would need such a redraw method anyway. Also, coming back to the DOM access speed issues once again, doing lots of DOM manipulation in one place is not as costly as doing it in smaller operations every now and then.

For the moment, let's just declare the todo list as a global (to our script, not on window) variable and then do the redraw method:

```
var todoListItems = [];
function redrawList() {
  var i;
  var list = document.getElementById('todo-list');
  var len = todoListItems.length;
  list.innerHTML = "";
  for(i=0; i<len; i++) {
    var item = document.createElement("li");
    item.appendChild(
      document.createTextNode(todoListItems[i].value)
    );
    list.appendChild(item);
  }
}
```

Please note that I used a simple `for` loop for iterating over the arrays. A more modern way that's actually implemented in almost all browsers (unfortunately NOT in IE7/8, but it's easy to emulate that) would be `forEach()`.

Also, there are a million ways of writing that for loop and how to declare the len and the i variables. I'm afraid I'll have to leave it up to you to develop your own JavaScript style.

Did you notice that I used innerHTML to empty the list: Yep, that's actually the quickest way of deleting a complete subtree from the DOM.

Our keypress handler now is significantly smaller:

```
function newTodoKeyPressHandler(event) {
  if (event.keyCode === 13) {
    var todoField = document.getElementById('new-todo');
    todoListItems.push(todoField.value);
    redrawList();
    todoField.value = "";
  }
}
```

Persistence is futile

Let's resolve the last piece of the puzzle: Storage. To keep it simple, I'll just show you one way to do it, without really going into details. We'll use DOM storage (specifically localStorage) to persist our data within the browser. This means that you can reload the page at any given time and you will get your todo-list back.

This does not mean that you can recall that list from other browsers, because then you'd need a server component, which is out of the scope of this book.

localStorage is just a very simple key value store, that allows you to store strings under a key. In our case, we'll just dump our array of strings into a single key and retrieve it on a page reload. This is a good time to extract the "add item to list" functionality as well. So let's start with that function:

```
function addToList(item) {
  todoListItems.push(item);
  localStorage.setItem('todo-list', JSON.stringify(todoListItems));
}
```

The `newTodoKeyPressHandler` needs to be adjusted as well:

```
function newTodoKeyPressHandler(event) {
  if (event.keyCode === 13) {
    var todoField = document.getElementById('new-todo');
    addToList(todoField.value);
    redrawList();
    todoField.value = "";
  }
}
```

So, the `localStorage` object has a method called setItem, and that method wants a key (in our case: just `"todo-list"`) and a string version of your data. The easiest way to get a string version of an array is to use the JSON object. It is available at all browsers starting from IE 8, and luckily that's the same IE version that introduced localStorage. So, calling `JSON.stringify()` gives you back a string containing the JSON representation of the object you stuffed in there.

So far, so simple. Now let's build a method to restore the list in case of a reload:

```
function reloadList(item) {
  var stored = localStorage.getItem('todo-list');
  if (stored) {
    todoListItems = JSON.parse(stored);
  }
  redrawList();
}
```

Obviously, `localStorage.getItem` only gives you a string version of your data (makes sense, right?), so now we need to use JSON.parse to get back our data. After that, it's just a question of redrawing the list.

By adding `reloadList()` to the `windowLoadHandler()` function, we now have a persisting version of our todo list.

Making it beau-tee-ful

There's tons of stuff missing but don't you think as well that this is actually pretty good for 53 lines of HTML and JavaScript?

Before we're going to add more functionality (we can't delete items or mark them as done, for example), I'd like to take a step back and make the whole thing a little more beautiful. For that I'm going to shuffle the HTML around a bit and add some CSS to it. All of that is taken from the before mentioned TodoMVC project. So the HTML up to the script tag looks like this:

```
<section id="todoapp">
  <header id="header">
    <h1>todos</h1>
    <input id="new-todo" placeholder="What needs to be done?" autofocus>
  </header>
  <section id="main">
    <ul id="todo-list"></ul>
  </section>
</section>
<footer id="info">
  <p>Template by <a href="http://github.com/sindresorhus">Sindre Sorhus</a></p>
</footer>
```

Also, I've added a `base.css` and a `background.png` file snatched from the TodoMVC[8] project. I won't reprint it here, as said in the introduction, you can find all of those files in the github.com repo. It doesn't look perfect right now because our list items are quite a bit simpler than in the TodoMVC end result, but since this is directly connected to our next feature, we need to live with that for a minute.

The complete example can be found at code.thesinglepageapp.com[9].

[8] https://github.com/addyosmani/todomvc

[9] http://code.thesinglepageapp.com/jumpstart/03_having_a_list/index.html

The terror of choice

You’ve now seen (or at least glanced over) four to five different ways of building single page apps:

1. Vanilla JavaScript
2. Using only small libraries like jQuery and Underscore
3. Using Backbone
4. Using Angular
5. Using Ember

The TodoMVC web page currently has more than 16 different frameworks and libraries listed with a TodoMVC example. Given that not all existing ones are listed at TodoMVC, the abundance of choice can be overwhelming. But rest assured: It’s even harder for me as an author to give you some advice here. I’ve used some of the frameworks, some in more depth than others, but the variables are just too a many.

What I can do, though, is try to explain some of the criteria I would use to choose a framework.

The ecosystem

All big names on TodoMVC have a decent eco system, especially the ones I’ve used as examples. And with ecosystem I don’t only mean a lot of plugins, extensions and examples, but also stuff like a strong support community and useful documentation. How many books are available. How good are the StackOverflow answers. And so forth. This seems not so important at first, but believe me, it makes a huge difference in the long run.

Aesthetics of code

I personally find this important. For example, I find Angular templates, and Angular code relatively unpleasant to read. All the dollar signs. And the wild mixture between HTML and code. I’m not a fan. I find Ember code to be quite nice to read, on the other hand. But opinions on aesthetics differ in many ways and for some people, it’s just not as important. One way to test for this is relatively simple: Open a random file from one of the TodoMVC examples and see how long it takes you to understand what’s going on. Because if it appeals to your sense of aesthetics, chances are that it is easier to understand for you.

Structure

How much structure does the framework or library provide? How easy is it to change this structure or to use your own? How much does the framework resist bending the original conventions? Does the framework have strong conventions in the first place? These are things to reason about with the whole team you'll be working with on the application, as every team has different opinions on all of these topics. I personally find strong conventions helpful as they usually mean that you can start relatively quickly actually building stuff. On the other hand if you have the feeling that you constantly have to work against these conventions, then it's probably not the right framework for you.

Performance

Different frameworks have different performance characteristics. For example, Ember has a very efficient way of doing intelligent updates on views, so if you're doing a lot of small view updates in your app, this might be a factor. And performance does matter, especially when building mobile applications. As there's no clear "this framework is slow" vs. "this framework is fast", when in doubt, benchmark your specific usecase.