

ToDoList - OC8

ToDo List App.

Web Performance Audit

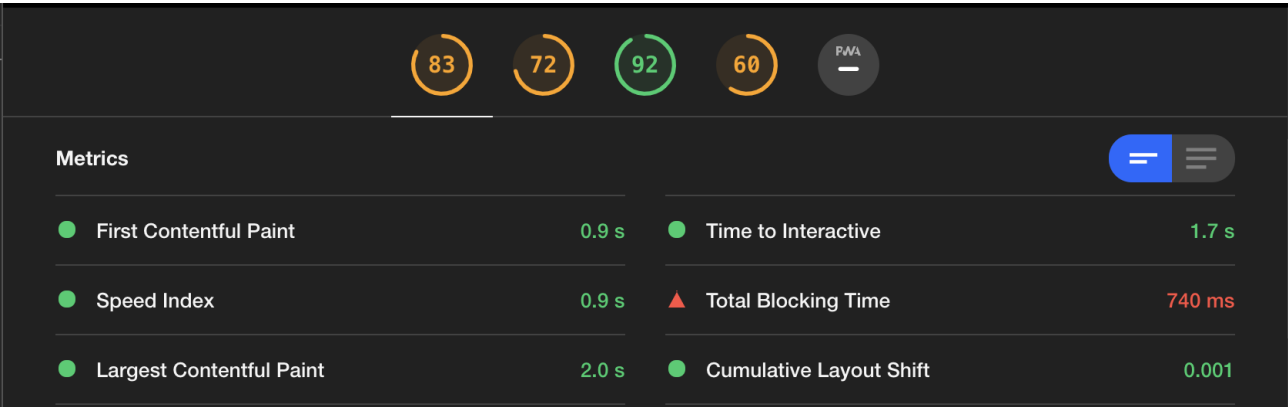


For an application to be perceived faster than the competitors it has to be at least 20% faster.

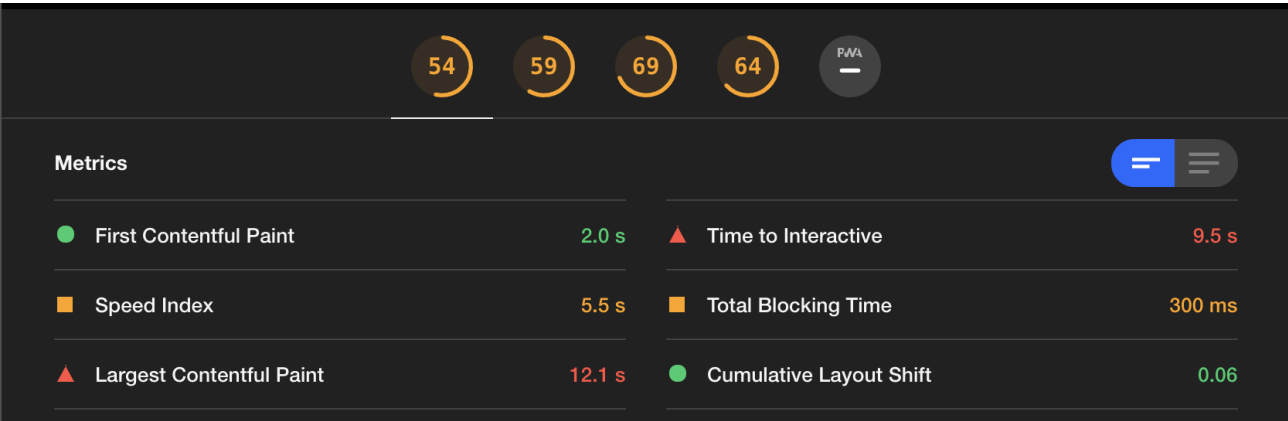
Key questions		Client	Competitor
Performance score		83	54
UX questions	Metrics		
Is it happening?	First contentful paint	0.9s	2.0s
Is it useful?	Speed index	0.9s	5.5s
	Largest contentful paint	2.0s	12.1s
Is it usable?	Time to interactive	1.7s	9.5s
It is delightful?	Total blocking time	740ms	300ms
	Cumulative layout shift	0.001s	0.006s
Opportunities		Potential savings	
	Eliminate render-blocking resources	0.19s	0.63s
	Minify JavaScript	0.3s	0.19s
	Remove unused JavaScript	0.3s	0.9s
	Minify CSS	0.19s	
	Enable text compression	0.45s	
	Preconnect to required origins		0.35s

Screenshots

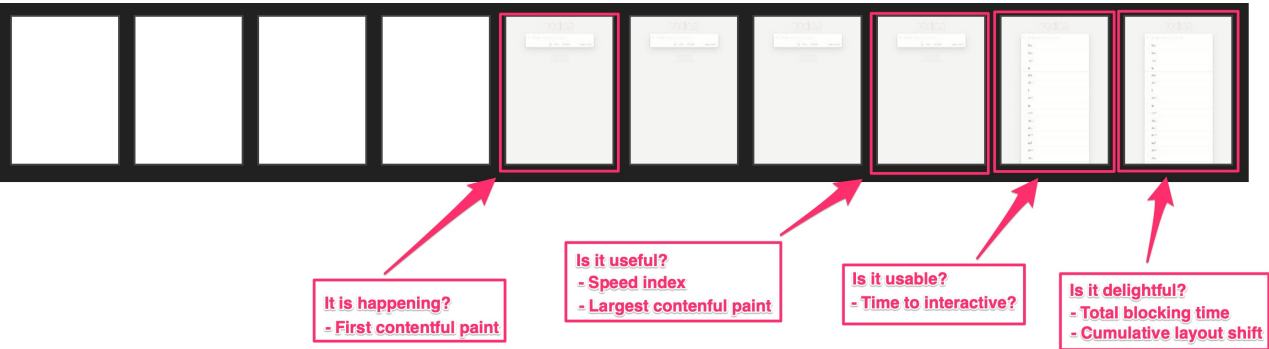
Client



Competitor



Focus on User Experience metrics



What the metrics mean and what influences them

First contentful paint(FCP)

First Contentful Paint marks the time at which the first text or image is painted.


One issue that's particularly important for FCP is font load time.

This metric measures the time from when the page starts loading to when any part of the page's content is rendered on the screen. For this metric, "content" refers to text, images (including background images), `<svg>` elements, or non-white `<canvas>` elements.

Speed index


Speed Index shows how quickly the contents of a page are visibly populated. Lighthouse first captures a video of the page loading in the browser and computes the visual progression between frames. Lighthouse then uses the Speedline Node.js module to generate the Speed Index score.

While most of the improvements will benefit the Speed index score, there are some issue that has a big impact: Minimize main thread work, Reduce JavaScript execution time, Ensure text remains visible during webfont load

 To provide good user experience, sites should strive to show the most important element within the first **2.5 seconds** of the page starting to load.

Largest contentful paint

The Largest Contentful Paint (LCP) metric reports the render time of the largest content element visible within the viewport (text block, image, background images, video).

 To provide good user experience, sites should strive to show the most important element within the first **2.5 seconds** of the page starting to load.

Time to Interactive (TTI)

The TTI metric measures the time from when the page starts loading to when its main sub-resources have loaded and it is capable of reliably responding to user input quickly.

- ❗ Measuring TTI is important because some sites optimize content visibility at the expense of interactivity. This can create a frustrating user experience: the site appears to be ready, but when the user tries to interact with it, nothing happens.

A page is considered fully interactive when:

- The page displays useful content, which is measured by the First Contentful Paint,
- Event handlers are registered for most visible page elements, and
- The page responds to user interactions within 50 milliseconds.

To improve the TTI score:

- Defer or remove unnecessary JavaScript work
- Reduce JavaScript payloads with code splitting
- Apply the PRPL pattern (Push - Render - Pre-cache - Lazy-Loading)
- Minimize main thread work
- Reduce JavaScript execution time

- ❗ To provide good user experience, sites should strive to have a Time to Interactive of less than **5 seconds** when tested on **average mobile hardware**.

Total blocking time - The *blocking time* of a given long task is its duration in excess of 50 ms. And the *total blocking time* for a page is the sum of the *blocking time* for each long task that occurs between First Contentful Paint (FCP) and Time to Interactive (TTI).

- ❗ To provide good user experience, sites should strive to have a Total Blocking Time of **less than 300 milliseconds** when tested on average mobile hardware.

Cumulative layout shift

CLS measures the sum total of all individual *layout shift scores* for every *unexpected layout shift* that occurs during the entire lifespan of the page.

A *layout shift* occurs any time a visible element changes its position from one frame to the next.

To calculate the *layout shift score*, the browser looks at the viewport size and the movement of unstable elements in the viewport between two rendered frames. The layout shift score is a product of two measures of that movement: the *impact fraction* and the *distance fraction*



To provide good user experience, sites should strive to have a CLS score of less than **0.1**.

Human-Computer interaction

Make users the focal point of your performance effort.

Smooth animation - **<16ms**

- Users are exceptionally good at tracking motion, and they dislike it when animations aren't smooth. They perceive animations as smooth so long as 60 new frames are rendered every second. That's 16 ms per frame, including the time it takes for the browser to paint the new frame to the screen, leaving an app about 10 ms to produce a frame.

Instant response - **<100ms**

- Respond to user actions within this time window and users feel like the result is immediate. Any longer and the connection between action and reaction is broken.

User stays focused on a task - **<1000ms**

- Within this window, things feel part of a natural and continuous progression of tasks. For most users on the web, loading pages or changing views represents a task.

User is frustrated - **>10s**

- Beyond 10000 milliseconds (10 seconds), users are frustrated and are likely to abandon tasks. They may or may not come back later.

Opportunities to improve performance

Eliminate render-blocking resources

Consider delivering critical JS/CSS inline and deferring all non-critical JS/styles.

Optimizing for performance is all about understanding what happens in these intermediate steps between receiving the HTML, CSS, and JavaScript bytes and the required processing to turn them into rendered pixels - that's the **critical rendering path**.

By optimizing the critical rendering path we can significantly improve the time to first render of our pages. Further, understanding the critical rendering path also serves as a foundation for building well-performing interactive applications.

Minify JavaScript

Minifying JavaScript files can reduce payload sizes and script parse time.

Minification is the process of removing whitespace and any code that is not necessary to create a smaller but perfectly valid code file. Terser is a popular JavaScript compression tool. webpack v4 includes a plugin for this library by default to create minified build files.

Remove unused JavaScript

To fix this issue, **analyze your bundle** to detect unused code. Then remove **unused** and **unnneeded** libraries.

DevTools makes it easy to see the size of all network requests:

1. Press `Control+Shift+J` (or `Command+Option+J` on Mac) to open DevTools.

2. Click the **Network** tab.
3. Select the **Disable cache** checkbox.
4. Reload the page.

Minify CSS

Minifying CSS files can improve your page load performance. CSS files are often larger than they need to be.

For small sites that you don't update often, you can probably use an online service for manually minifying your files. You paste your CSS into the service's UI, and it returns a minified version of the code.

For professional developers, you probably want to set up an automated workflow that minifies your CSS automatically before you deploy your updated code. This is usually accomplished with a build tool like Gulp or Webpack.

Enable text compression

Text-based resources should be served with compression (gzip, deflate or brotli) to minimize total network bytes.

Preconnect to required origins

Consider adding `preconnect` or `dns-prefetch` resource hints to establish early connections to important third-party origins.

`<link rel="preconnect">` informs the browser that your page intends to establish a connection to another origin, and that you'd like the process to start as soon as possible.

Establishing connections often involves significant time in slow networks, particularly when it comes to secure connections, as it may involve DNS lookups, redirects, and several round trips to the final server that handles the user's request.

Taking care of all this ahead of time can make your application feel much snappier to the user without negatively affecting the use of bandwidth. Most of the time in establishing a connection is spent waiting, rather than exchanging data.

Informing the browser of your intention is as simple as adding a link tag to your page:

```
<link rel="preconnect" href="https://example.com">
```

Technical Documentation

JavaScript Helpers

For a faster and easier JavaScript development and a more maintainable and uncluttered code, the **'ToDo List'** project benefits from a library of utils and programming helpers that make the development process easier when it comes to HTML document traversal and manipulation, event handling, animation and much more.



The code helpers can be found in `helpers.js`.

Get elements by CSS selectors

Syntax	Example
<pre>let variableSelector = qs('.selector-name', scope);</pre>	<pre>this.\$todoList = qs('.todo-list');</pre>
<pre>let elementsArray = qsa('.selector-name', scope);</pre>	<pre>var arrayElements = window.qsa('.selectors-name', this.\$todoList);</pre>

Event listener wrapper

Syntax	Example
<pre>\$on(target, 'event', callbackFunction)</pre>	<pre>\$on(window, 'load', setView);</pre>

Attach a handler to an event for all elements that match the selector

Syntax	Example
<pre>\$delegate(targetElement, variableSelector, 'event', callbackFunction)</pre>	<pre>\$delegate(self.\$todoList, 'li .edit', 'keyup', callbackFunction(event))</pre>

Find the element's parent with the given tag name

Syntax

```
let parentElement = $parent(element,  
tagName);
```

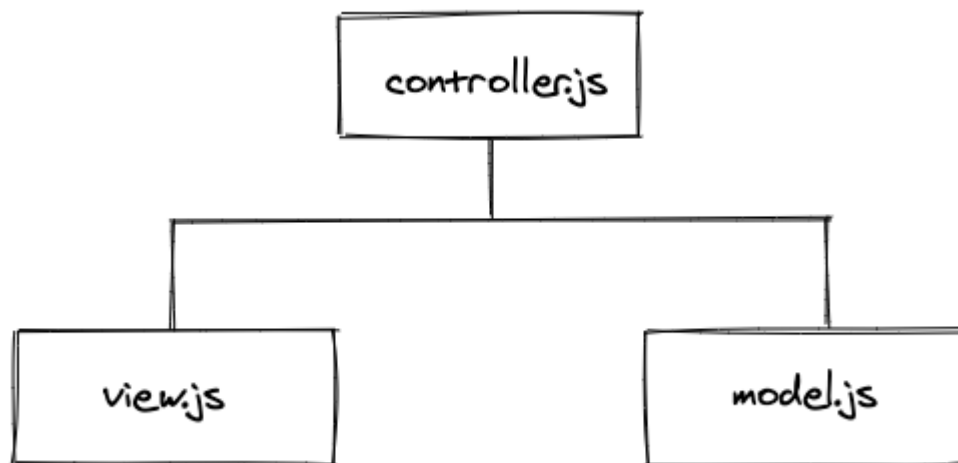
Example

```
let parentEl = $parent(qs('a'),  
'div');
```

Model View-Controller JavaScript pattern

The MVC programming design pattern is a disciplined approach to solving a code problem. This design pattern separates concerns and promotes clean code.

At its crux, the MVC design pattern is about a clean-cut separation of concerns. The idea is to make the solution intelligible and inviting.



The `controller.js` handles events and is the mediator between the view and model. It works out what happens when the user performs an action (for example, clicking on a button or pressing a key).

- Receives input (from view, URL)
- Processes requests (GET, POST, PUT, DELETE)
- Gets data from the `model.js`
- Passes data to the `view.js`

The `view.js` cares about the DOM. The DOM is the browser API you use to make HTML manipulations. In MVC, no other part cares about *changing* the DOM except for the view. The view can attach user events but leaves event handling concerns to the controller. The view's prime directive is to change the state of what the user sees on the screen.

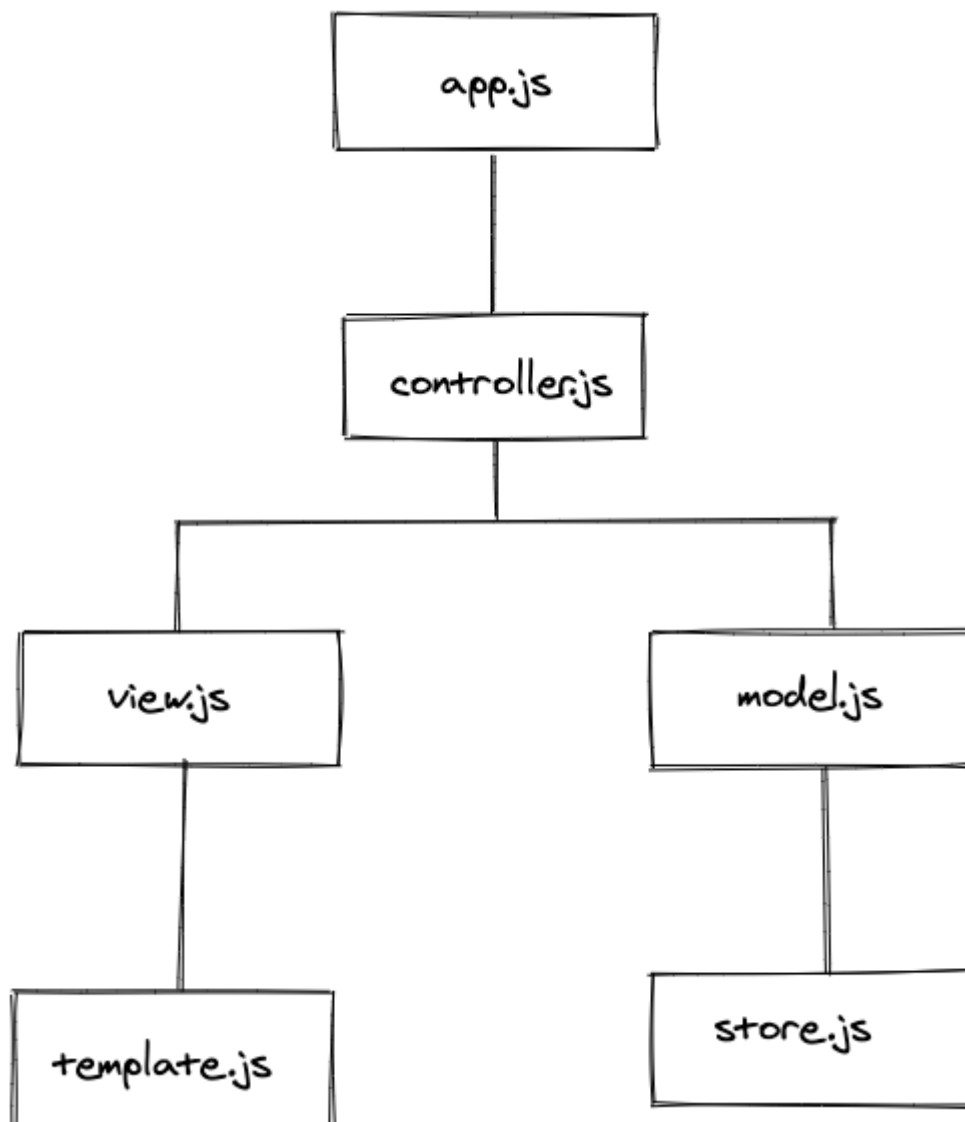
- What the end-user sees
- Usually consists of HTML and CSS
- Communicates with the `controller.js`

- Can be passed dynamic values from the `controller.js`
- Needs to use a Template Engine (HAML, Handlebars, Underscore, etc.)

The `model.js` cares about data. In client-side JavaScript, this means Ajax. One advantage of the MVC pattern is you now have a single place for server-side Ajax calls. This makes it inviting for fellow programmers who are not familiar with the solution.

- Data related logic
- Interact with the database (SELECT, INSERT, UPDATE, DELETE)
- Communicates with the `controller.js`

The MVC pattern 'ToDo List' application is using



The `app.js` is responsible for kicking off the app. An instance of the Model-View-Controller gets created here.

The `template.js` is responsible to generate the `` HTML element for each task on the list, element that will be manipulated and added to the DOM by `view.js`.

The `store.js` is responsible for getting, posting, and deleting data from the local storage, and this way passing to the `model.js` the required data that it cares about.