TECHNICAL DOCUMENTATION

TodoList App

Nicole Sentis

TECHNICAL DOCUMENTATION

TABLE OF CONTENTS

INTRODUCTION

TodoList App is a simple application that helps users be more organised and focused. It lets them create a list of todos and manage this by:

- Adding new todos
- Updating todos
- Marking each todo as either active or completed
- Toggling all todos to either active or completed
- Displaying only active, only completed or all todos
- Deleting individual todos
- Deleting all completed todos
- Showing the number of active todos left

INSTALLATION

1. Clone project to your local machine
   ➔ Follow GitHub's instructions: https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/cloning-a-repository

2. Make sure you have Node and NPM installed

3. Run NPM install to install the packages in package.json

4. Open index.html file in browser

DESIGN AND STRUCTURE

TodoList App behaves like a Single Page Application and uses Vanilla JS (ES5) with an MVC (Model – View – Controller) pattern, routes based on location hashes, a template, localStorage, and NPM packages for CSS and unit tests. It's a frontend application with no backend.

File structure

The file structure looks as follows:

- Js
  - App.js
  - Controller.js
  - Helpers.js
  - Model.js
  - Store.js
  - Template.js
  - View.js
- test
  - controller.spec.js
  - spec-runner.html
- .gitignore
- Index.html
- Package.json

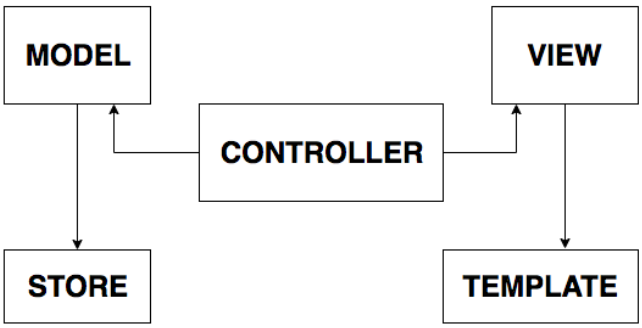
HTML

Index.html – entry page of the application, which adds the menu links, loads the scripts and sets the container element (the <ul> with class 'todo-list') to render the routes' HTML.

JavaScript and MVC

App.js – sets up a new Todo List, which instantiates Store, Model, View, Controller and Template instances as properties of itself.
Store.js – creates and manages a new storage object that uses localStorage.
Template.js – sets up and manages a default template for each Todo List-item (todo). It also manages the template of the Active items count and the visibility of the 'Clear completed' button.
Model.js – creates and manages a new Model instance. The Model instance has the Store instance as a property and is responsible for managing the app's data by updating its storage.
View.js – creates and manages the view with the Template instance as its property. The view abstracts away the browser's DOM completely.
Controller.js – creates and manages a new Controller instance, which takes the Model and View instances as properties and acts as the controller between them. It also controls the routes and updates the view based on the activeRoute.
Helpers.js – a file with helper methods.

The diagram below shows the relationship between the elements.



Because of the ES5 syntax, Model, View, Controller, Store and Template use Object Constructors with the prototype property to add new methods and properties, and not ES6 Class syntax.

For an explanation of the methods in each of these files, see the Reference Page.

Routing

This application uses routing based on location hashes. The location hash property is set by adding a hash sign (#) to the href attribute of the menu-items in index.html. Location.hash returns the anchor part of the URL, including the hash sign.

Every time the app loads or the location hash changes, the Controller sets the activeRoute and currentPage based on the location hash. Then it calls the View's render method with the currentPage as argument to render the correct view.

Dependencies

Todomvc-common – common css and js files for TodoMVC apps (not used by this app)
Todomvc-app-css – index.css file for TodoMVC apps
Jasmine-core – official packaging of Jasmine's core files. Necessary for testing.

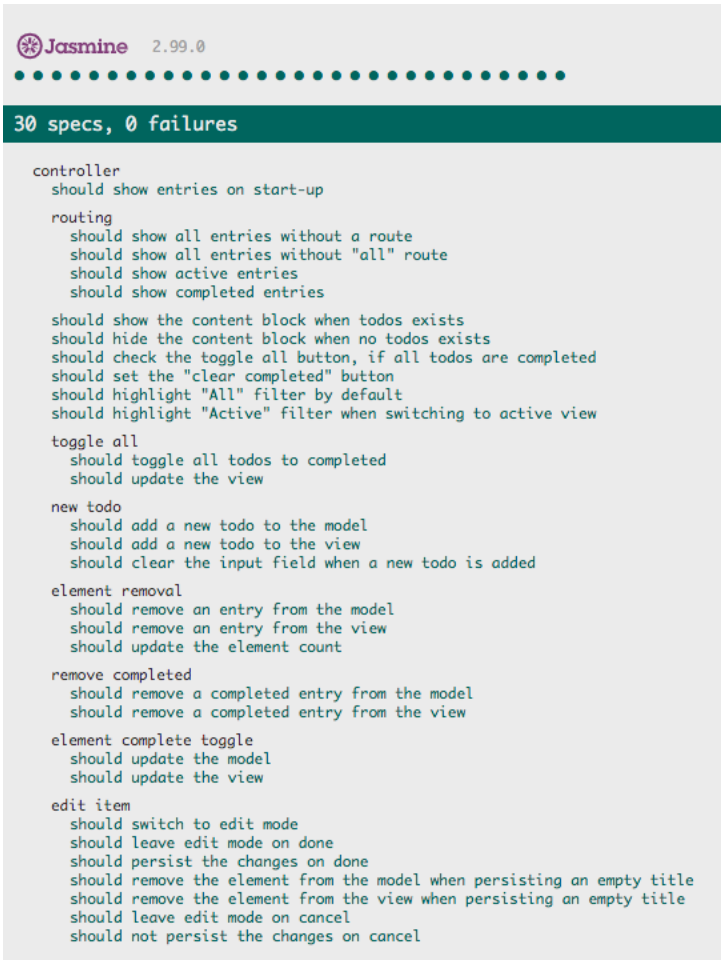
TESTING

The framework used for the unit tests is Jasmine, version 2.99.1. For Jasmine's documentation, see https://jasmine.github.io/pages/docs_home.html.

Spec-runner.html – the file that runs the specs.
Controller.spec.js – the file with all the Jasmine specs and suites of specs that test Controller functions.

The controller.spec.js file uses spies to fake the model and view, which are created before each spec. setUpModel() creates fake callback functions for the model. That way, the controller functions can be tested in isolation.

The following image shows all the passing Controller tests:



REFERENCE GUIDE

The reference guide was automatically generated by documentation.js, using the comments in the JS files. For the documentation of documentation.js
see https://github.com/documentationjs/documentation/blob/master/docs/GETTING_STARTED.md.

Todo

Sets up a brand new Todo list.

Parameters

- name **string** The name of your new to do list.

Controller

Takes a model and view and acts as the controller between them

Parameters

- model **object** The model instance
- view **object** The view instance

setView

Loads and initialises the view

Parameters

- locationHash
- null **string** '' | 'active' | 'completed'

showAll

An event to fire on load. Will get all items and display them in the todo-list

showActive

Renders all active tasks

showCompleted

Renders all completed tasks

addItem

An event to fire whenever you want to add an item. Simply pass in the event object and it'll handle the DOM insertion and saving of the new item.

Parameters

- title

removeItem

By giving it an ID it'll find the DOM element matching that ID, remove it from the DOM and also remove it from storage.

Parameters

- id **number** The ID of the item to remove from the DOM and storage

removeCompletedItems
Will remove all completed items from the DOM and storage.

toggleComplete
Give it an ID of a model and a checkbox and it will update the item in storage based on the checkbox's state.

Parameters

- id **number** The ID of the element to complete or uncomplete
- completed
- silent **(boolean | undefined)** Prevent re-filtering the todo items
- checkbox **object** The checkbox to check the state of complete or not

toggleAll
Will toggle ALL checkboxes' on/off state and completeness of models. Just pass in the event object.

Parameters

- completed

_updateCount
Updates the pieces of the page which change depending on the remaining number of todos.

_filter
Re-filters the todo items, based on the active route.

Parameters

- force **(boolean | undefined)** forces a re-painting of todo items.

_updateFilterState
Simply updates the filter nav's selected states

Parameters

- currentPage

Model
Creates a new Model instance and hooks up the storage.

Parameters

- storage **object** A reference to the client side storage class

create
Creates a new todo model

Parameters

- title **string?** The title of the task
- callback **function?** The callback to fire after the model is created

read

Finds and returns a model in storage. If no query is given it'll simply return everything. If you pass in a string or number it'll look that up as the ID of the model to find. Lastly, you can pass it an object to match against.

Parameters

- query **(string | number | object)?** A query to match models against
- callback **function?** The callback to fire after the model is found

Examples

```
model.read(1, func); // Will find the model with an ID of 1
model.read('1'); // Same as above
//Below will find a model with foo equalling bar and hello equalling world.
model.read({ foo: 'bar', hello: 'world' });
```

update

Updates a model by giving it an ID, data to update, and a callback to fire when the update is complete.

Parameters

- id **number** The id of the model to update
- data **object** The properties to update and their new value
- callback **function** The callback to fire when the update is complete.

remove

Removes a model from storage

Parameters

- id **number** The ID of the model to remove
- callback **function** The callback to fire when the removal is complete.

removeAll

WARNING: Will remove ALL data from storage.

Parameters

- callback **function** The callback to fire when the storage is wiped.

getCount

Returns a count of all todos

Parameters

- callback

Store

Creates a new client side storage object and will create an empty collection if no collection already exists.

Parameters

- name **string** The name of our DB we want to use

- callback **function** Our fake DB uses callbacks because in real life you probably would be making AJAX calls

find
Finds items based on a query given as a JS object

Parameters
- query **object** The query to match against (i.e. {foo: 'bar'})
- callback **function** The callback to fire when the query has completed running

Examples
```
db.find({foo: 'bar', hello: 'world'}, function (data) {
 // data will return any items that have foo: bar and
 // hello: world in their properties
});
```

findAll
Will retrieve all data from the collection

Parameters
- callback **function** The callback to fire upon retrieving data

save
Will save the given data to the DB. If no item exists it will create a new item, otherwise it'll simply update an existing item's properties

Parameters
- updateData **object** The data to save back into the DB
- callback **function** The callback to fire after saving
- id **number** An optional param to enter an ID of an item to update

remove
Will remove an item from the Store based on its ID

Parameters
- id **number** The ID of the item you want to remove
- callback **function** The callback to fire after saving

drop
Will drop all storage and start fresh

Parameters
- callback **function** The callback to fire after dropping the data

Template
Sets up defaults for all the Template methods such as a default template

show
Creates an <li> HTML string and returns it for placement in your app.

NOTE: In real life you should be using a templating engine such as Mustache or Handlebars, however, this is a vanilla JS example.

Parameters

- data **object** The object containing keys you want to find in the template to replace.

Examples

```
view.show({
id: 1,
title: "Hello World",
completed: 0,
});
```

Returns **string** HTML String of an <li> element

itemCounter

Displays a counter of how many to dos are left to complete

Parameters

- activeTodos **number** The number of active todos.

Returns **string** String containing the count

clearCompletedButton

Updates the text within the "Clear completed" button

Parameters

- completedTodos **[type]** The number of completed todos.

Returns **string** String containing the count

View

View that abstracts away the browser's DOM completely. It has two simple entry points:

- bind(eventName, handler) Takes a todo application event and registers the handler
- render(command, parameterObject) Renders the given command with the options

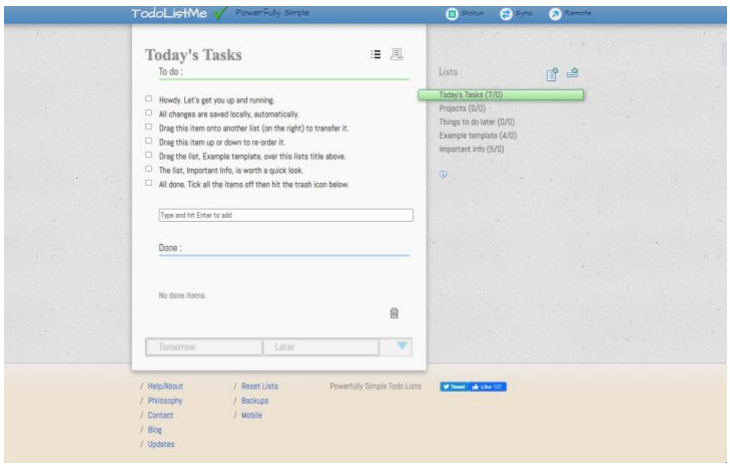
Parameters

- template

COMPETITOR PERFORMANCE AUDIT

Audit performed using Developer Tools in Chrome Browser (v. 83.0.4103.116).
Throttling: slow 3G.
Other tools used: Lighthouse 5.7.1, WebPageTest and Pingdom.

The competitor: http://todolistme.net/.



Introduction

At the moment our app performs very well (Lighthouse score of 99 for mobile) but is quite limited in terms of functionality and scale. This document analyses above-mentioned competitor, which has extended functionality, in order to learn how we can keep our performance optimised when scaling.

Differences

A comparison between our functionality and theirs shows the following differences:

	Our app	Todolistme.net
CRUD	✓	✓
Toggling active – complete	✓	
Counting active todos	✓	
Saving in localStorage	✓	✓
Saving status icon		✓
Multiple todo lists		✓
List categories		✓
Templates		✓
Sorting todos in list		✓
Printing lists		✓
User accounts for syncing lists		✓
Scheduling todos		✓
Drag and drop reordering		✓
Social media like/share buttons		✓
Ads		✓
Pages for instructions, backups, philosophy, updates, blog & mobile		✓

Performance

Lighthouse Metrics

Running a Lighthouse audit returns a score of 39 for mobile. The following table shows the metrics:

First Contentful Paint	1.9 s	First Meaningful Paint	2.8 s
Speed Index	6.1 s	First CPU Idle	10.8 s
Time to Interactive	11.9 s	Max Potential First Input Delay	370 ms

Conclusion: users will see something fairly quickly but have to wait a long time before interaction is possible on slower connections.

HTTP Requests

The Network panel logs that over 90 requests can be fired on loading the page. The following tables show a breakdown per domain and per content type:

CONTENT TYPE	PERCENT	REQUESTS
todolistme.net	30.43%	28
pagead2.googlesyndication.com	8.70%	8
tpc.googlesyndication.com	7.61%	7
googleads.g.doubleclick.net	6.52%	6
apis.google.com	6.52%	6
code.jquery.com	5.43%	5
other	34.78%	32
Total	100.00%	92

(source: Pingdom)

CONTENT TYPE	PERCENT	REQUESTS
Script	38.64%	34
Image	34.09%	30
HTML	13.64%	12
CSS	4.55%	4
Font	4.55%	4
XHR	2.27%	2
Redirect	2.27%	2
Total	100.00%	88

(source: Pingdom)

In all requests from their own domain the HTTP/1.1 protocol is used:

Name	Protocol	Domain
todolistme.net	http/1.1	todolistme.net
style_g.css	http/1.1	todolistme.net
lists.js	http/1.1	todolistme.net
lib.js	http/1.1	todolistme.net
javascript_e.js	http/1.1	todolistme.net
tick.png	http/1.1	todolistme.net
top_not_saved.png	http/1.1	todolistme.net
texture.png	http/1.1	todolistme.net
top_sync.png	http/1.1	todolistme.net

Resources

Size

In terms of size, the site's largest resources are coming from their **own domain**, **jQuery**, **social media** domains and **Google ads services and APIs**. Especially **scripts** and **images** are large:

CONTENT TYPE	PERCENT	SIZE
todolistme.net	17.72%	207.3 KB
pagead2.googlesyndication.com	15.40%	180.2 KB
code.jquery.com	14.49%	169.5 KB
www.facebook.com	12.86%	150.5 KB
apis.google.com	11.19%	130.9 KB
connect.facebook.net	5.28%	61.8 KB
other	23.06%	269.8 KB
Total	100.00%	1.2 MB

Content size per domain (source: Pingdom)

CONTENT TYPE	PERCENT	SIZE
Script	59.86%	723.4 KB
Image	14.81%	179.0 KB
XHR	11.62%	140.4 KB
HTML	7.31%	88.4 KB
Font	4.78%	57.8 KB
CSS	1.49%	18.0 KB
Redirect	0.13%	1.6 KB
Total	100.00%	1.2 MB

Content size per type (source: Pingdom)

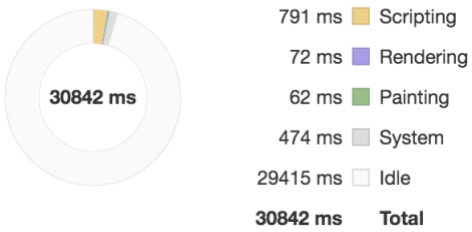
Loading Time

On slow 3G mobiles the longest loading resources are also generally the largest. They include **images and scripts of their own domain**, **Google ads services**, **jQuery** and **social media scripts**:

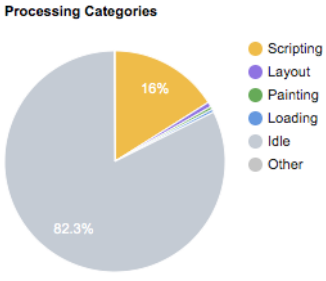
Name	Domain	Type	Size	Time
texture.png	todolistme.net	png	132 kB	12.97 s
jquery-ui.js	code.jquery.com	script	125 kB	12.74 s
show_ads_impl_fy2019.js	pagead2.googlesyndica...	script	84.7 kB	7.45 s
all.js?hash=499b22e48e0167515f9ea91...	connect.facebook.net	script	58.9 kB	6.75 s
jquery-2.2.4.min.js	code.jquery.com	script	30.3 kB	6.41 s
show_ads.js	pagead2.googlesyndica...	script	31.4 kB	6.20 s
widgets.js	platform.twitter.com	script	29.9 kB	6.09 s
NMnUyfnxjqx.js?_nc_x=Ij3Wp8lg5Kz	www.facebook.com	xhr	132 kB	4.60 s
ga.js	www.google-analytics.c...	script	17.2 kB	4.39 s
KtkxAKiDZI_td1Lkx62xHZHDtgO_Y-bvT...	fonts.gstatic.com	font	13.5 kB	4.16 s
javascript_e.js	todolistme.net	script	10.3 kB	3.86 s
lists.js	todolistme.net	script	10.1 kB	3.80 s

Main-thread Time Breakdown

Range: 0 – 30.84 s

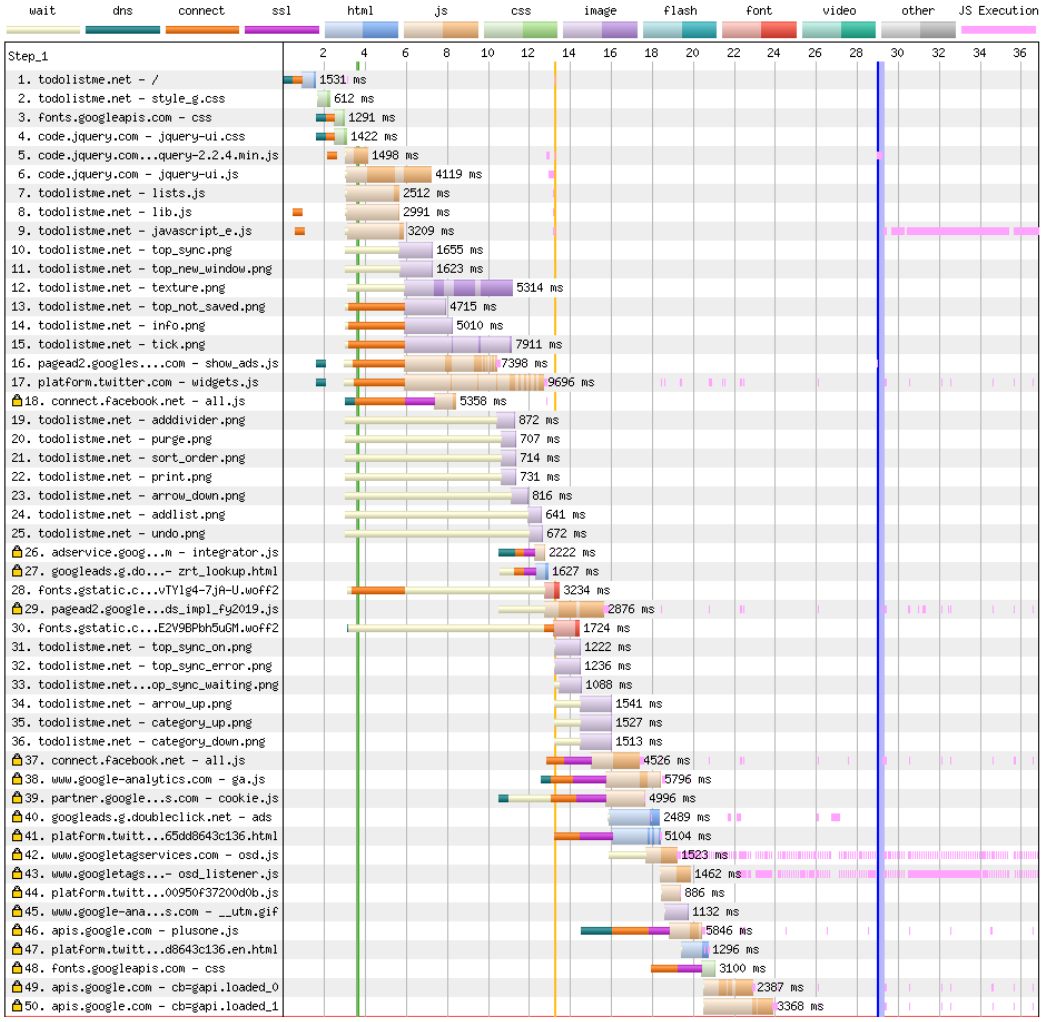


(source: Performance Panel)



Category	Time (ms)
Idle	30,250
Scripting	5,900
Layout	259
Painting	160
Loading	160
Other	33

(source: WebPageTest)



Waterfall View (source: WebPageTest)

Category	Time Spent
Script Evaluation	3,345 ms
Other	938 ms
Style & Layout	501 ms
Script Parsing & Compilation	383 ms
Rendering	198 ms
Parse HTML & CSS	177 ms
Garbage Collection	116 ms

(source: Lighthouse)

The above tables and charts show the huge amount of **idle time** (mostly **waiting for images** to load). The next big time-consumer is **scripting**.

Third-party Code

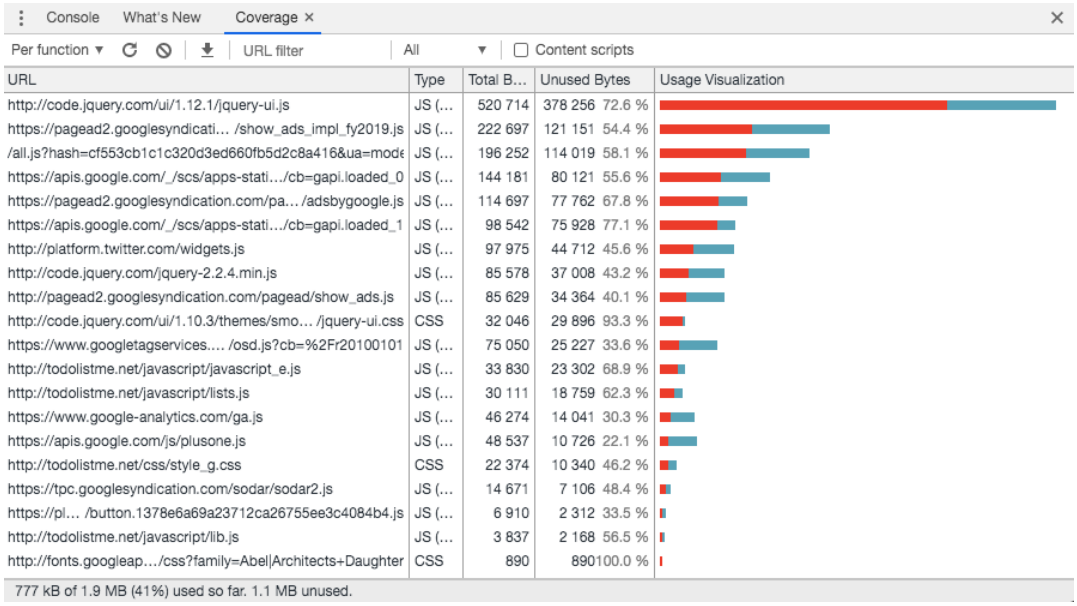
The following table shows that the main-thread is mostly **blocked by Google ads and jQuery**:

Third-Party	Size	Main-Thread Blocking Time
Google/Doubleclick Ads	361 KB	767 ms
jQuery CDN	178 KB	396 ms
Other Google APIs/SDKs	153 KB	315 ms
Google Analytics	34 KB	302 ms
Twitter	33 KB	89 ms
Facebook	264 KB	70 ms

(source: Lighthouse)

Code Coverage

After loading, less than 50% of the code in the files that load has been used. Most unused code comes from third-party sources:



Summary

Above results show that the following are the largest contributing factors to slower loading:

- Too many HTTP requests
- Images and scripts of their own domain
- Third-party (non-asynchronously loading) scripts, in particular:
    - Google ads
    - jQuery
    - social media scripts
- Large amounts of unused code

Recommendations

Based on the performance of our competitor, we should consider the following when scaling our app:

- Limiting the number of HTTP requests (and DNS lookups) by
    - Bundling
    - Sprites
    - Regularly assessing what third-party code (ads services, social media, libraries) is essential and eliminating what is not
- Limiting the impact of remaining third-party code on main-thread by
    - Loading scripts for ads etc. asynchronously from just above </body>
    - Where possible, extract only what is needed
- Removing unused main code and resources where possible
- Keeping size small by
    - Minifying JS
    - Compressing images
    - Not using PNG images for icons
    - Not using jQuery, but continuing with Vanilla JS
- Lazy loading & code splitting
- Using HTTP/2 instead of HTTP/1.1