Travlendar+ Implementation and Test Document

Sinico Matteo, Taglia Andrea

Version 1.0

December 2017

link to the application demo

link to the source code

Table of Contents

1. INTRODUCTION

A.Introduction

This document addresses the implementation and testing deliverable of the travlendar+ application. In the following pages is described how we have implemented and tested our application, based on the what there is in the RASD and DD documents already delivered.

B. Scope

The main scope of this document is to explain the implementation choices taken to deploy a first version of the Travlendar+ application, and to show what kind of tests have been performed on the application, explaining also the automated tools used in order to accomplish this task.

2. REQUIREMENTS/FUNCTIONALITIES IMPLEMENTED

In this section we have included the list of requirements/functionalities implemented in the first version of Travlendar+ application, followed by a brief motivation either for including them or for excluding them. We have reused the abbreviations used in the RASD document to refer to the requirements.

It hasn't been taken care too much of deep usability aspects as the software is intended to be an initial version. Of course improvements can be made from almost any viewpoint.

Here is a list of the requirements/functionalities implemented in the software:

- [R1.1] A log in functionality must be provided in order to authenticate the user.
- [R1.2] User's data (calendar appointments, preferences, personal data) must be kept secret to the user himself and to anybody else.
- [R1.4] Users must have a personal calendar.
- [R2.1] Users must be able to add an event to their personal calendar giving a title, a day, a time at which the event should be scheduled in the calendar.
- [R2.3] Users must be able to delete an already existing event in their personal calendar.
- [R2.4] Users must be able to change their preference about car usage.
- [R2.5] Users must be able to change their preference about bike usage.
- [R2.6] Users must be able to change their preference about walking.
- [R2.7] Users must be able to change their preference about carbon footprint.
- [R2.8] Users must be able to change their preference about public transports.
- [R2.9] Users must be able to edit preferences on break times choosing a lower and upper bound for lunch and dinner times and a minimum break time.
- [R3.1] Computation of optimal route must take into account user preferences.
- [R3.5] No appointment can be successfully scheduled after another if it would not respect flexible break times.

- [R3.6] Change in break times preferences are not taken into account for previously scheduled appointments.
- [R4.1] No overlapping appointments are allowed. A warning would show up and the user may not be able to schedule the event.
- [R4.2] Transport time estimation between appointment location being added and previous location on schedule must be done based on public transports only.
- [R4.3] The application must always be aware of the previous user location with respect to the newly added event.
- [R5.1] The application must compute a travel proposal for the current day to show to the user.

We have implemented these requirements/functionalities because we think that they are all essential in order to captivate future users.

On the other hand here is a list of the requirements/functionalities which has not been implemented in the software with motivation for excluding them

- [R1.3] A password recovery functionality must be available in order for the user not to permanently lose access to the service.
- [R1.5] Users must be able to edit their personal data. (profile data)
- [R2.2] Users must be able to edit an existing event changing the title.

These 3 requirements/functionalities are not seen as necessary because they don't enrich the value of the application, so they are only a loss of time, at least in a first version of the application

- [R3.2] Computation of optimal route must take into account weather forecast.
- [R3.3] Computation of optimal route should be made more times and each with different weight on preferences in order to give different day-travel proposals.
- [R3.4] There must be a way to let the user switch among different proposal.
- [R5.2] A brief description of each Travel Piece of a day Travel Proposal should be shown to the user.
- [R5.5] Each appointment should have an estimated time of arrival (computed based on public transports) attached.

These requirements/functionalities have not been implemented because they are not useful in order to engage future users. But they are nice to have future, so they can be shown as future feature.

- [R5.3] Allow users to check any day scheduled appointments;
- [R5.4] There must be a comprehensive view over any given day.

These requirements/functionalities can't be implemented because we didn't include a calendar view of user's appointments, because we were keen on show a travel proposal for the current day, instead of implement an agenda. These features will be certainly added in a future version of the application.

3. ADOPTED DEVELOPMENT FRAMEWORKS

In this section we state some of the choices done in terms of programming languages, framework and middleware technologies and the main reasons which led to the choice.

A. Adopted programming languages and its advantages and disadvantages

This really wasn't a choice. **HTML**, **Javascript** and **CSS** are just what is needed given the previous architectural and framework choices.
The only choice has been made with the Javascript version which we opted for the newest ES6. Also no CSS precompilers have been used, so we just sticked to pure CSS3 with no SaSS enanchments for simplicity reasons and due to the limited size of the project we wanted to make the development toolchain as light as possible.
Compared to a fully Java written application, which has been taken into account as the main alternative, we definitely are at a lower level using pure native languages. This makes the whole code a bit less understandable. We also lose a bit in well defined partitioning of the taks during development flow.
On the other hand we are plenty of pros, as using platform native languages gives as much more control over the whole architecture and this made us able to use service worker features, fine grained control over resource caching, good control over bundle build to exploit the new HTTP2 protocol.

At a higher level of abstraction it is mandatory to comment the use of the framework around which the whole application code as been built. The core application has been developed using the **Polymer** js framework which is an open-source project led by a team of front-end developers working within the Chrome organization at Google.
Thanks to new web platform primitives many features which have always been leaking on the web, compared to those of native platforms, are now available. Especially:

- Web Components let us extend the browser's built-in component model—the DOM—rather than bring a custom one which is most of the time used by any javascript framework.
- In combination with HTTP/2 and server push, standard module formats like HTML Imports and ES6 Modules let us declare fine-grained dependencies and efficiently deliver them to the client in optimally cacheable form, without relying on complicated packaging tools and loaders.
- Service Workers let us build pure web apps that users can access even when their devices are offline or network conditions are poor, whereas usually we might have had to resort to manually installable native or "hybrid" apps.

Here are the main reasons which led to the choice of the Polymer framework:

- Polymer uses open web technologies and new web standards.
- Polymer supports shadow and shady Document Object Model (DOM), which really gives a little of "engineering sugar" on the web development world where it really lacks a proper object oriented programming model.
- The Polymer DOM layer is closest to the Native JavaScript layer.
- Onboarding is faster compared to other web framework like React or Angular.

- Connection with third party libraries is very easy.
- Arguably, it’s the best JavaScript library when it comes to Progressive Web Apps.

B. Any middleware you have adopted and its advantages and disadvantage

Firebase platform is the main middleware which stays right between the Polymer code on the front end and our backend which is made almost serverless thanks to the Firebase integration. By *almost serverless* we intend to point at the only pure server side code we have running in the back-end which is the “initUserPrefs” trigger which is executed once a new user logs in and it has the purpose of initializing some default preferences. We use this feature which we repute very elegant exploiting the firebase platform further.
The firebase platform comes in handy to the Travlendar+ project on the following:

- Database: we make use of the realtime firebase database.

It gives advantages in terms of good performance mainly. It is extremely important to have quick updates and fetches of data for the smooth flow we pursued developing the application. It is also great to have a nosql database as it is definitely quicker to develop and has a much more flexible structure which again perfectly aligns with the agile development flow we adopted.
Of course the trade off for elasticity is a loss in robustness which led to some hard to debug errors at developing time. Also the API is actually pretty limited and complex query have to been created “from scratch” and often in not very efficient way. However it was a price we were aware of and still the advantages of such a solution outweigh the cons.

- Authentication: the application trusts the auth providers and credential management service offered by the firebase platform.

The obvious advantage of using the service is that the hard work of writing a full and secure suite of web service methods for the authentication comes almost for free here and this is something that really speeds up the agile flow. Also, some analytics can be done on user information.
On the downside we have a pretty much static way to build the auth flow and we have no direct control over the information the users entrust to Travlendar.

- Hosting: we also went through the building process and thus a hosting service was needed to host the application files. Again we relied on the Firebase ecosystem.

We have very poor server configuration settings, but it is intended in a serverless approach. However, that is arguably the only drawback, whether instead we are plenty of advantages like to possibility to serve files from a straight forward HTTP2, or the easy and quick way to deploy from a command line tool. Really, this is something it didn’t take too long to choose.

It is extremely important the use of the **Service Worker** javascript demon which is one of the newly supported features of the web platforms. It is not actually spread all across the browsers yet, but it should only be a matter of months until it becomes a standard. Precisely, the browser support for it can be checked out at the popular “*caniuse*” website (Safari lacks support currently).
The Service Worker really does a great job in enabling applications to take advantage of persistent background processing, including hooks to enable bootstrapping of web applications while offline. Service workers and the cache storage sub API allow to render our website immediate (as a native app) and also let cache dynamic content with our own rules.

At the end of the story what we really get as benefits by using a Service Worker mainly resides in the *cacheFirst* strategy used as middleware between the browser and the server. It increases significantly page load time and first paint times on app reuse. Moreover, it has our back covered when it comes to laggy and slow networks or even in case of total absence of internet connection: it holds some app shell and fragments into the browser cache, so it can give them in reply to browser requests in any condition, and this leads to a really robust application.
On the other side, in terms of challenges what is worth mentioning is that it takes to be actually implemented, debugging and understanding all the options available for caching, and this is something which has not been that straight forward at all. Another obvious disadvantage is the lack of full browsers support: it has a coverage of around 65% of devices worldwide, with an 8% more if partial support is taken into account, but still it is not an impressive percentage but the bet here is in its rapid growth over the next few months.

4. STRUCTURE OF THE SOURCE CODE

Travlendar+ application source has been built having in mind the PRPL pattern, which stands for:

- Push critical resources for the initial route.
- Render initial route.
- Pre-cache remaining routes.
- Lazy-load and create remaining routes on demand.

To do this, the server needs to be able to identify the resources required by each of the app's routes. Instead of bundling the resources into a single unit for download, it uses HTTP2 push to deliver the individual resources needed to render the requested route.
The server and service worker together work to precache the resources for the inactive routes. It has extreme benefits once the app is loaded for the first time and the user comes back visiting Travlendar+ as the core files are ready to be cached and just the very few bytes that make the updated version will have to be downloaded.

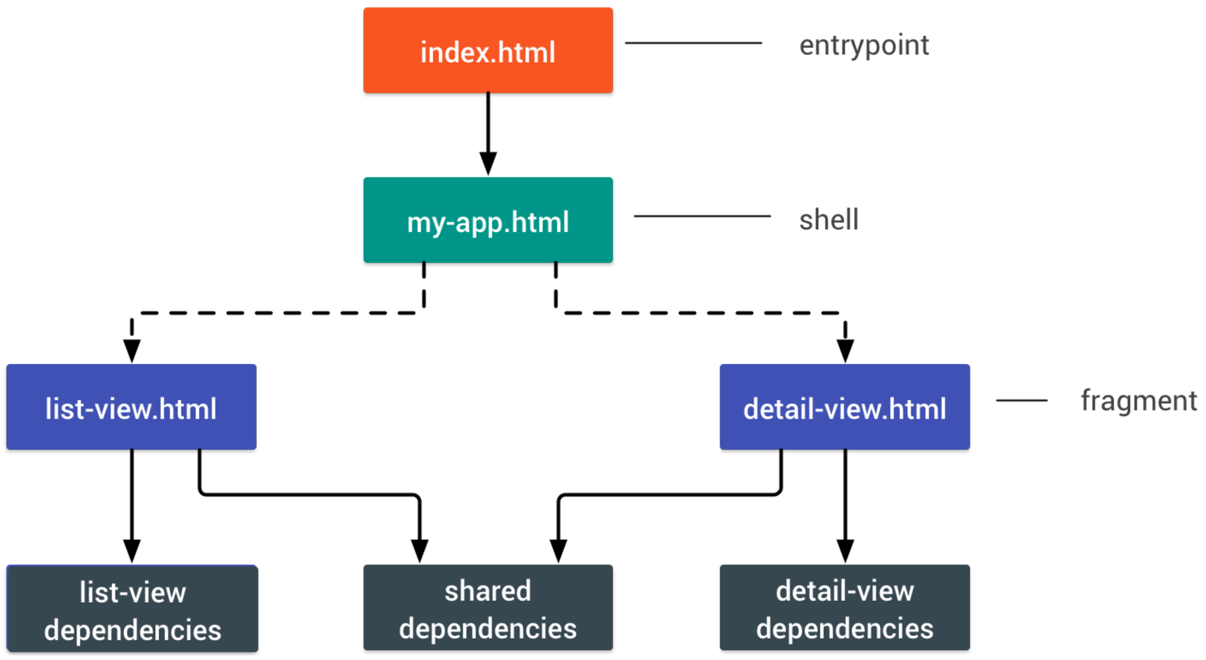
Moreover, when the user switches routes, the app lazy-loads any required resources that haven't been cached yet, and creates the required views. This part is actually made in slightly different way within travlendar. We actually preferred having a longer first time loading of the application views and have an almost zero input latency during in-app route navigation. This makes the application feels even more native having an extremely smooth flow.

Digging down the very structure it consists of a single-page app (SPA) with the following structure:

- The main entrypoint of the application which is served from every valid route. This file is indeed very small, since it will be served from different URLs therefore be cached multiple times. It is the index.html located under *./public/*
- The shell or app-shell, which includes the top-level app logic, router, and so on. This goes under the travlendar-app.html which is basically the main component. It is located again under *./public/*
- Lazily loaded fragments of the app. With respect to Travlendar+ a fragment represents the code for a particular view that are loaded lazily once the main

app-shell is ready. The shell is responsible for dynamically importing the fragments as needed. In fact, the travlendar-app components includes the routing with all the view-components instances.

The diagram below shows the components of a generic simple app:



Note: what described below refers to the build output of the application. Development dependencies and structure go beyond the intention of this document.

Again in the main ./public/ directory we find the manifest.json and the service-worker.js. The first one used in order to specify basic metadata about the application such as the name, version, theme colors etc. The second one is the actual service worker file which installs into the user browser and does the efficient caching jobs.
From *./public/* we find other three folders which are:

- *bower_components* where all the external web components and their files are placed
- *components* where the travlendar strictly-related components go. It mainly includes all the view components and their strict dependency, and the unit tests for the main components in the folder *test.*
- *functions* which contains the required material for the back-end trigger to give first time logged-in users a default preference set

5. TESTING:

In this section is described how we have performed tests on the application and their outcome.
We have logically divided tests in two category:

- Non Functional Requirements test
- Functional Requirements test

For the first category we have performed both black and white box tests using two automated test tools.
For the second category we have tested the components that we have implemented from scratch: using automated tools for the core one and non formal test for the other.

Please refer to the ReadMe in the official gitHub repository to see how to run the tests (https://github.com/YOUR_USERNAME/SinicoTaglia/blob/master/README.md).

Here are in detail the different tests we performed at different levels:

5.1 Non Functional Requirements - System Test - Black Box -> Page Insights

A very straightforward black box testing has been run with the popular page insights tool that gives a first look on the overall site speed by looking at some common optimization which should be done and it also tests the server with a quick analysis on the way the server reacts.
More in detail: “PageSpeed Insights measures the performance of a page for mobile and desktop devices. It fetches the url twice, once with a mobile user-agent, and once with a desktop user-agent.
PageSpeed Insights checks to see if a page has applied common performance best practices and provides a score, which ranges from 0 to 100 points.
PageSpeed Insights measures how the page can improve its performance on:

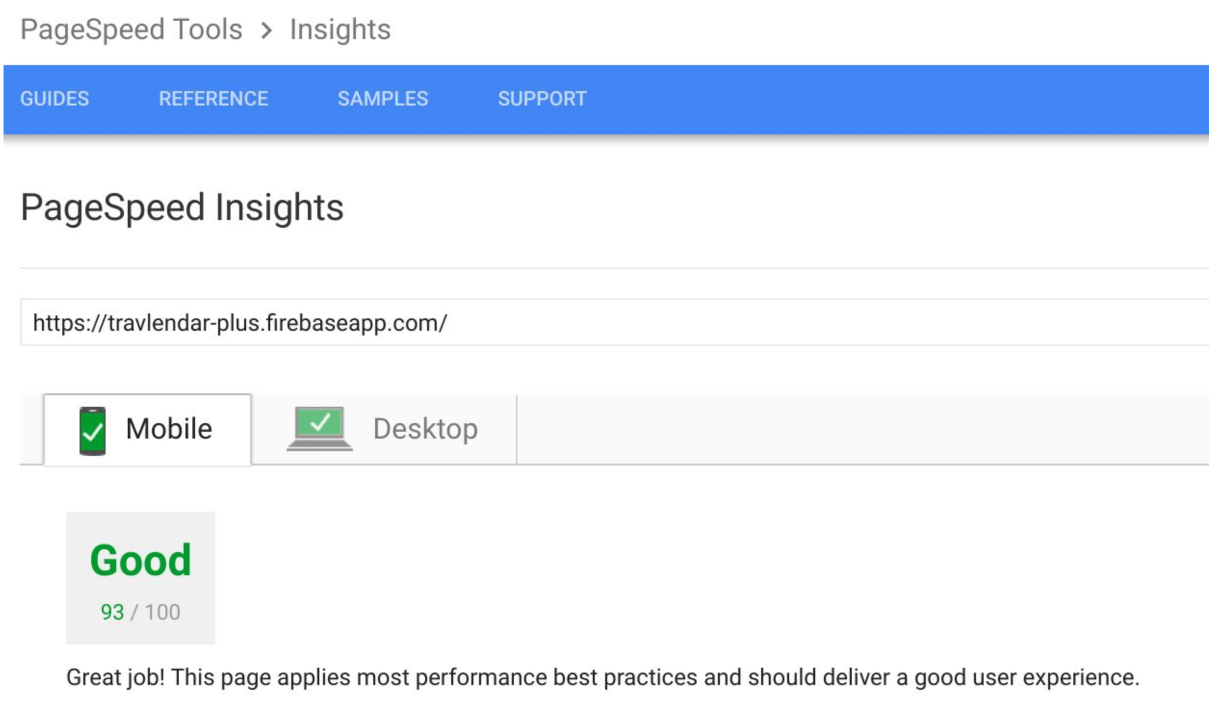
- time to above-the-fold load: Elapsed time from the moment a user requests a new page and to the moment the above-the-fold content is rendered by the browser.
- time to full page load: Elapsed time from the moment a user requests a new page to the moment the page is fully rendered by the browser.

However, since the performance of a network connection varies considerably, PageSpeed Insights only considers the network-independent aspects of page performance: the server configuration, the HTML structure of a page, and its use of external resources such as images, JavaScript, and CSS. Implementing the suggestions should improve the relative performance of the page. However, the absolute performance of the page will still be dependent upon a user’s network connection.”

Here is what we get as results after the analysis.

- Output for Mobile[1]:

[1] Note that scores may different because Page Insights is very frequently updated


PageSpeed Tools > Insights

GUIDES REFERENCE SAMPLES SUPPORT

PageSpeed Insights

https://travlendar-plus.firebaseapp.com/

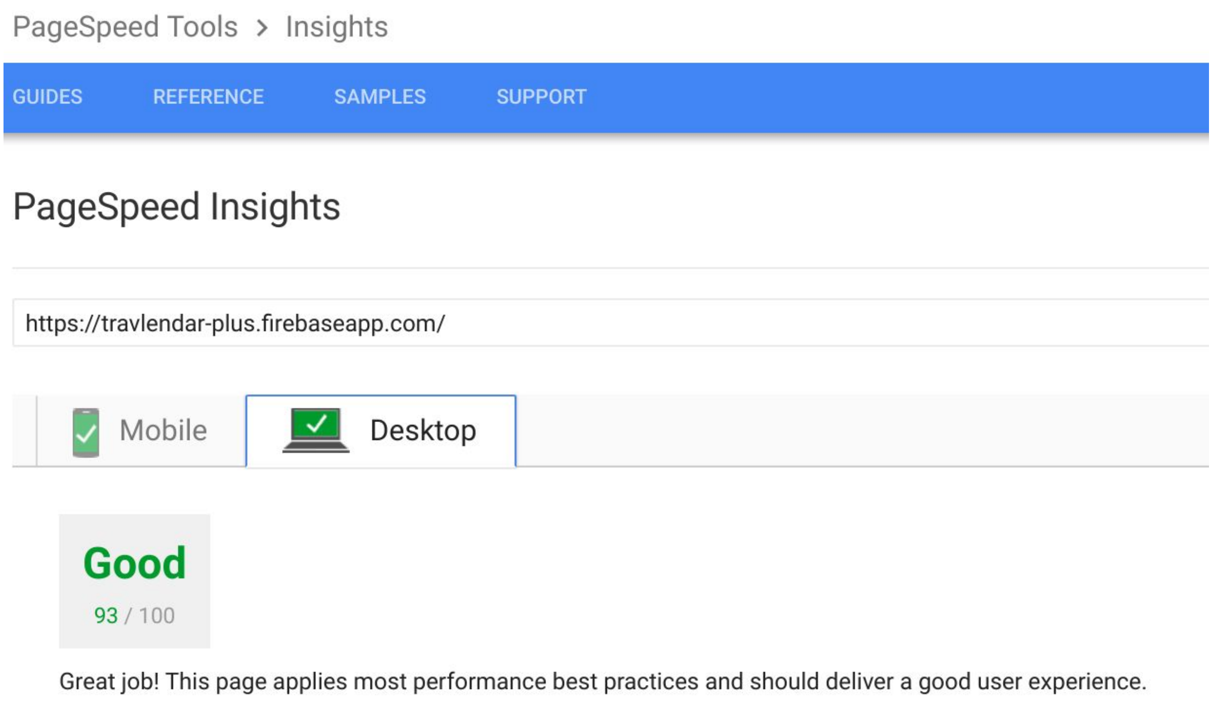
Mobile Desktop

Good

93 / 100

Great job! This page applies most performance best practices and should deliver a good user experience.

- Output for Desktop[2]:


PageSpeed Tools > Insights

GUIDES REFERENCE SAMPLES SUPPORT

PageSpeed Insights

https://travlendar-plus.firebaseapp.com/

Mobile Desktop

Good

93 / 100

Great job! This page applies most performance best practices and should deliver a good user experience.

Basically Travlendar+ web application applies most performance best practices and should deliver a good user experience. This was a one first good result as we really cared about delivering a good product.
The main points we lose come from the missing minification of some HTML and Javascript of the external web components we use. That is something which we are not really in charge of and the optimization on that side would required a more advanced and cherry picky build toolchain.

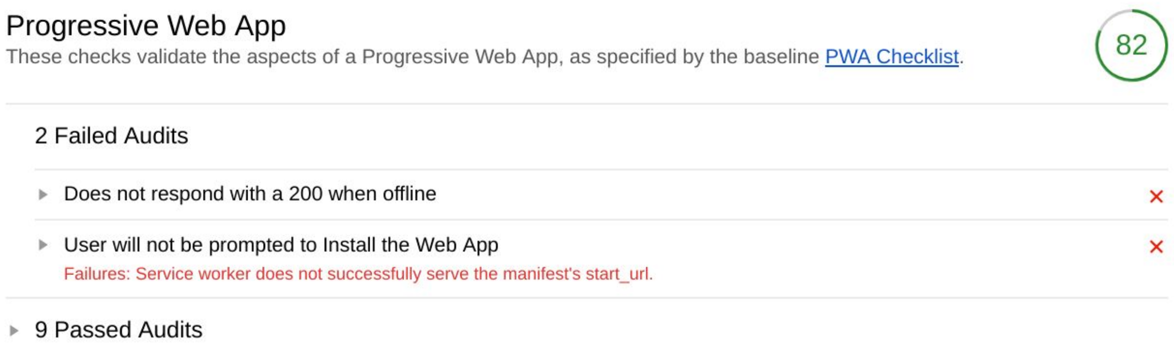
[2] Note that scores may different because Page Insights is very frequently updated

5.2 Non Functional Requirements - System Test - White Box -> Lighthouse[3]

Still on the non functional testing we have had a great support by the Google tool Lighthouse all over the developing flow. It goes deep down into the application code and test four different aspects: (full report available on the following Gist)

➢ *Progressive Web App*

All the nice specs progressive web app standards introduce, which have been previously described in the Design Document, are now being tested in practice. Here is the score Travlendar+ gets and below a few comments on the goods and the bads.



Progressive Web App
These checks validate the aspects of a Progressive Web App, as specified by the baseline PWA Checklist.
82

2 Failed Audits

▸ Does not respond with a 200 when offline ✕

▸ User will not be prompted to Install the Web App ✕
Failures: Service worker does not successfully serve the manifest's start_url.

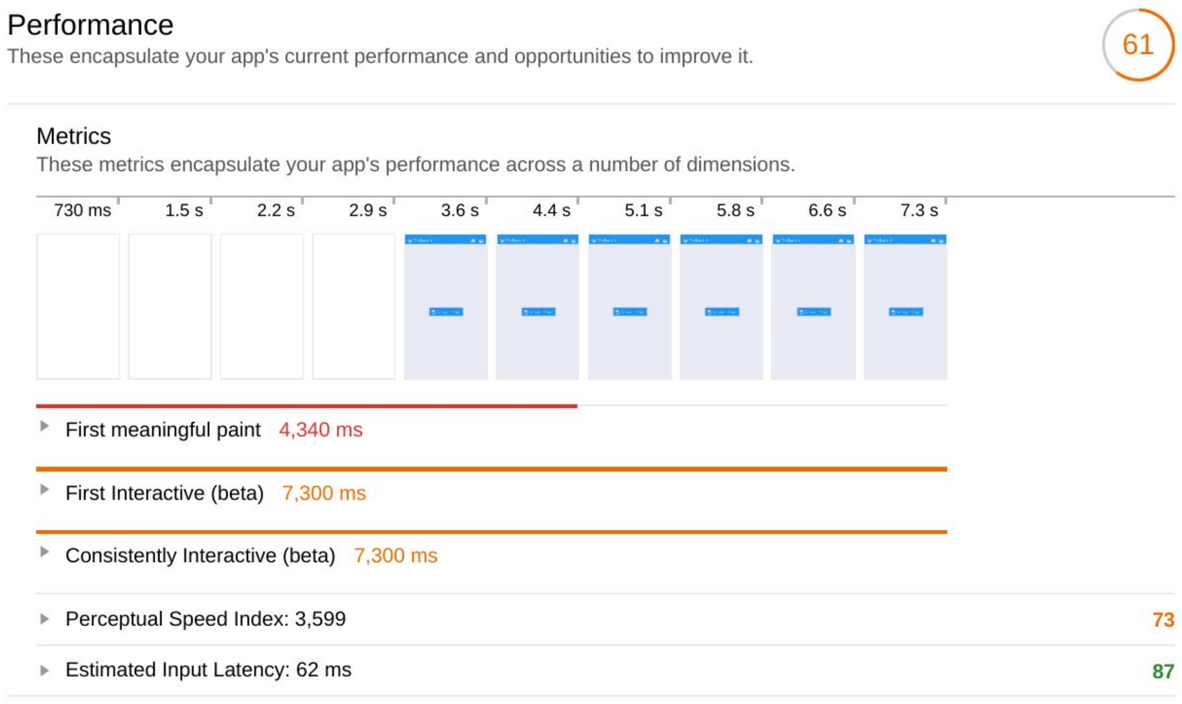
▸ 9 Passed Audits

Starting from the passed audits we achieved, of course we register a Service Worker and we also have support for installing the application on devices thanks to the manifest, the theme colors and the other little tweaks.
On the other side we fail in a couple of PWA must haves. It would take further fine tuning to reach the 100, but we reckon the score is pretty good for a first version.

➢ *Performance*

We already had some testing on this aspect coming from PageSpeed Insights, but that was a black box testing and it actually only referred to best practices mainly. Here we have a closer look at the code and at the network timings:

[3] Note that scores may different because Lighthouse is very frequently updated


Performance

These encapsulate your app's current performance and opportunities to improve it.

61

Metrics

These metrics encapsulate your app's performance across a number of dimensions.

First meaningful paint 4,340 ms

First Interactive (beta) 7,300 ms

Consistently Interactive (beta) 7,300 ms

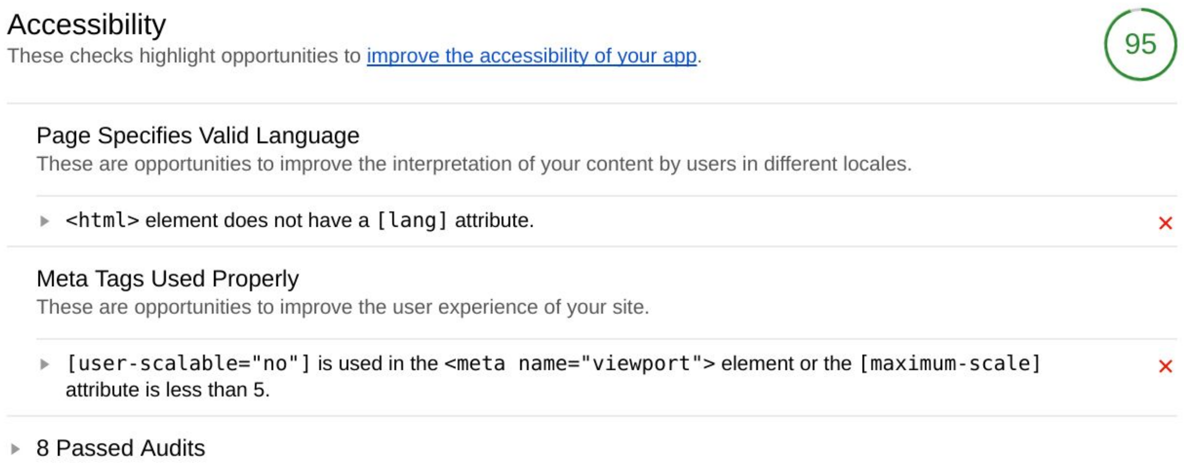
Perceptual Speed Index: 3,599 73

Estimated Input Latency: 62 ms 87

Actually first meaningful paint doesn't come out very quickly, but that a price we knew we would pay and we believe it is affordable. First interaction is right the same with the consistent one, which means Travlendar+ is completely loaded and ready to use. The score is just alright, first paint especially could get some improvements but again it is a good result for a first version. Improvements can be done on two sides: build process can be fine tuned with more aggressive bundling and resources minification, and at developing stage app shell should be kept lighter trimming away the code to the bare minimum.
Where the actual good comes into play is on the input latency, i.e. once the hard loading work has been done how quick and smooth is the navigation actually (not how it feels, but how it actually is) ? We are delighted with the great results here. Just as planned, page can be switched from one to another in a tiny amount of time.

➢ *Accessibility*

This is a point which should not be underestimated, and any good application should do all it takes to make it available to the most.


Accessibility

These checks highlight opportunities to improve the accessibility of your app.

95

Page Specifies Valid Language

These are opportunities to improve the interpretation of your content by users in different locales.

`<html>` element does not have a `[lang]` attribute. ✗

Meta Tags Used Properly

These are opportunities to improve the user experience of your site.

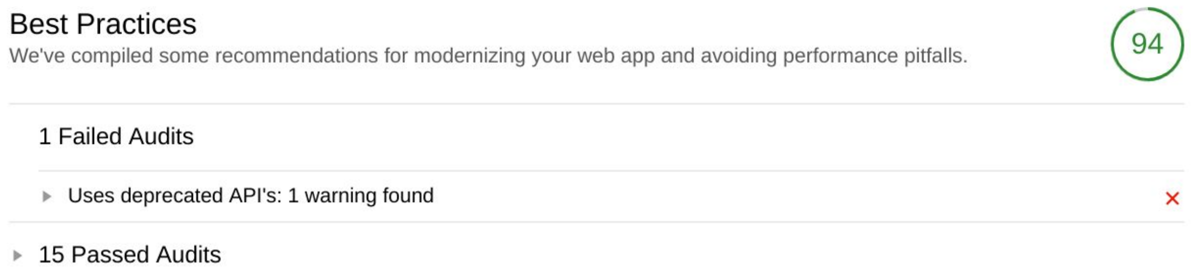
`[user-scalable="no"]` is used in the `<meta name="viewport">` element or the `[maximum-scale]` attribute is less than 5. ✗

8 Passed Audits

The score is great and respects the effort we put in making all the accessibility features available to screen readers and other tools of the same kind.

➢ *Best Practices*

Best practices have already been from a black box approach. Here we have an in-depth analysis which got the following:


Best Practices
We've compiled some recommendations for modernizing your web app and avoiding performance pitfalls.
94

1 Failed Audits

Uses deprecated API's: 1 warning found ✗

15 Passed Audits

Awesome score which respects the one gathered from PageSpeed Insights. The only point of failure comes from the use of a deprecated API which the developing team is aware of, but the support is planned for a few years as well, then it makes sense to have it in production even though it is marked as deprecated. Still this is something that must be taken into account in future versions of the application.

5.3 Functional Requirements - Test Suites - White Box -> Web Component Tester

When it comes to functional requirements we have decided to exploit the power of the web component tester which is an end-to-end testing environment built by the Polymer team, it enables testing our elements locally, to create and run test suites in order to unit test our main components.

We have decided to use web component tester to test 3 components out of the ones we have created:

- add-calendar-view-element.html
- compute-travel-view-element.html
- event-element.html

we chose these three because the first two are the components which correspond to the edit data handler and travel handler sub-system, stated in the DD documents, which are the core components of the application, where it can be found the major percentage of javascript code. So they are the more crucial and more exposed to errors parts of the entire applications. We chose the third one because includes “data binding”, a powerful feature of the framework polymer[4], that we use in several parts of our application.

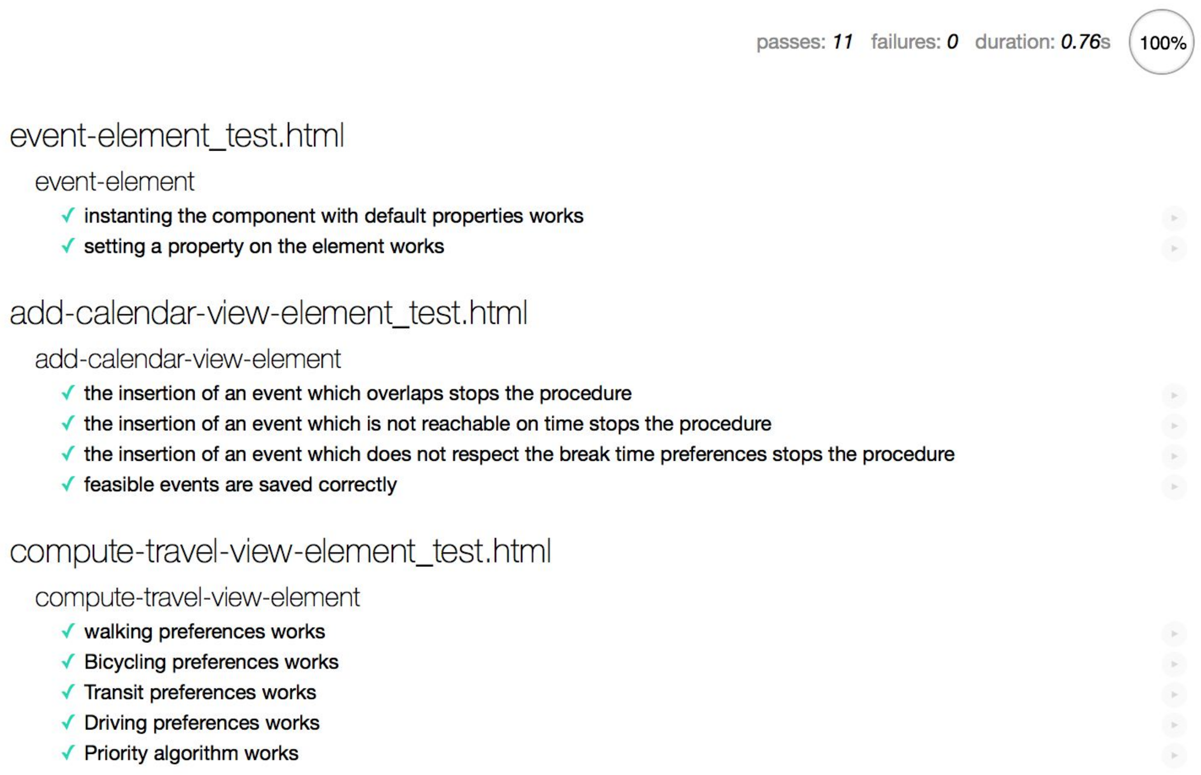
Here is a list of the test cases that we have done:

Component	Description
event-element	instanting the component with default properties[5] works
event-element	setting a property on the element works

[4] visit https://www.polymer-project.org/2.0/docs/devguide/data-binding to have more detail about what is data binding
[5] a property is an attribute of the element, like an attribute of a java object (visit https://www.polymer-project.org/2.0/docs/devguide/properties to have more detail on properties)

add-calendar-view-element	the insertion of an event which overlaps stops the procedure: if you try to add an event which overlaps with other events already present in the user’s personal events, the procedure stops and the application notifies the error to the user.
add-calendar-view-element	the insertion of an event which is not reachable on time stops the procedure: if you try to add an event, which is not feasible with the time constraints imposed by the events already present in the user’s personal events, the procedure stop and the application notify you the error.
add-calendar-view-element	the insertion of an event which does not respect the break time preferences stops the procedure: if you try to add an event, which does not respect the break time preferences of the user, the procedure stop and the application notify you the error.
add-calendar-view-element	feasible events are saved correctly: if you try to add an event which does not belong to one of the three previous cases, the application adds the event to the user’s personal events.
compute-travel-view-element	walking preferences works: Setting the user preferences with only the walking preferences active, the application suggest the user to go by walking.
compute-travel-view-element	Bicycling preferences works: Setting the user preferences with only the bicycling preferences active, the application suggest the user to go by bicycle.
compute-travel-view-element	Transit preferences works: Setting the user preferences with only the public transportation active preferences active, the application suggest the user to go by transit.
compute-travel-view-element	Driving preferences works: Setting the user preferences with only the driving preferences active, the application suggest the user to go by car.
compute-travel-view-element	Priority algorithm works: Setting the user preferences with all the means of transport active, the application suggest the user with the first means feasible. For example if a events is reachable on time only by transit and not by walking or by bicycle the application will suggest to go by transit.

Here is the outcome of the unit test performed:


passes: 11 failures: 0 duration: 0.76s 100%

event-element_test.html

event-element

✓ instanting the component with default properties works
✓ setting a property on the element works

add-calendar-view-element_test.html

add-calendar-view-element

✓ the insertion of an event which overlaps stops the procedure
✓ the insertion of an event which is not reachable on time stops the procedure
✓ the insertion of an event which does not respect the break time preferences stops the procedure
✓ feasible events are saved correctly

compute-travel-view-element_test.html

compute-travel-view-element

✓ walking preferences works
✓ Bicycling preferences works
✓ Transit preferences works
✓ Driving preferences works
✓ Priority algorithm works

5.4 Functional Requirements - Non Formal Tests on components

We performed non formal test upon reaching a beta version of each component during development phase. This has gone all along integration testing and system testing.

6. INSTALLATION INSTRUCTIONS:

Please refer to the ReadMe on the official repository located on github.

7. EFFORT SPENT:

Around 100 hours per member.

8. REFERENCES

A. Bibliography

- Polymer-Project: https://www.polymer-project.org/
- PageSpeed Insights: https://developers.google.com/speed/pagespeed/insights/