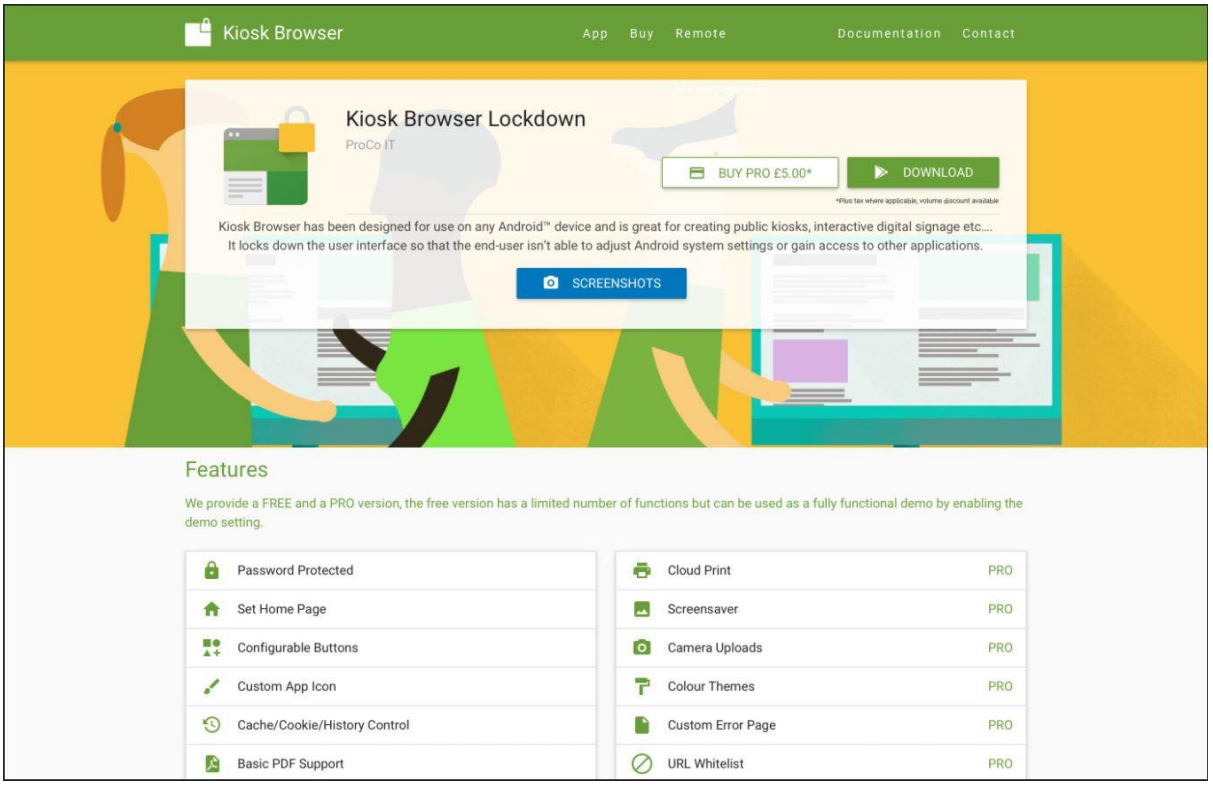
Leopold Huber - lh223sf
Andreas Karlsson Franzén - af222ud

Assignment 2 - “Materialize”

Introduction

The aim of this package is to provide pre-built web components that adheres to the "Material Design Principles" created by Google. The components are built using CSS and Javascript [1].

The basic design language is probably very familiar to people using Google’s services in the mid to late 2010’s, as can be seen from “Kiosk Browser’s” implementation below [11]:



The picture above shows an example implementation of Materialize, “Kiosk Browser”.

Overview of requirements and specifications

This chapter will contain a brief overview of the existing requirements and specifications Materialize has to adhere to. As we could not find any specified functional requirements on neither their website, nor in their Github repository, we will mainly focus on the design principles of the material design philosophy.

Requirements specification

The word "Material" is a metaphor for how objects and their textures look in the physical world, and how they reflect light and cast shadows. The following quote has been taken from the official material design website [2]:

> "*Material is an adaptable system of guidelines, components, and tools that support the best practices of user interface design. Backed by open-source code, Material streamlines collaboration between designers and developers, and helps teams quickly build beautiful products.*"

The “Material Design Principles” [3] is used in Google's Android and Flutter projects and is also employed by individual developers developing front-end applications for all types of devices. It is composed by two main principles:

- *“Bold, graphic, intentional”*: The use of bold colors and hierarchy to put the most important components in the focus of the viewer.
- *“Motion provides meaning”*: Subtle feedback and coherent transitions focuses attention and maintains continuity.

Stakeholders

Since this is a web-component framework, the common stakeholders include the creators of various websites wanting a look and feel that matches the material design principles used in Google's products.

Another less common stakeholder is a user who writes mobile applications intended for use on iOS and Android. They may write their implementation using common website building techniques, and then convert them into native applications with the help of tools such as “ionic" [4].

Risks

Since this is a front-end framework the major risk is causing bad user experiences due to untested components, or components that don't integrate or interact well with each other.

One risk to consider is built-in, malicious javascript snippets that affect the users visiting the site. Such scripts may redirect users to malicious sites or try to hijack sessions.

If we have a separate front- and backend running on different virtual machines then security vulnerabilities should only affect the front-end machine and its users, unless the backend isn't validating all input data sent to it.

Since the result we're after is visual we must also consider the manual testing required to be a risk. Different testers may test differently, and they may not have time to do a full regression testing every time the package is updated. The testers may also find that the manual testing is boring and excruciating, increasing the risk of them glossing over the tests, just to get them over with.

There is automatic unit testing implemented in the package, but we've found that it's complicated to write automatic tests that cover all things that can go wrong related to placements on the screen and how the components should be displayed in general. The automatic tests missed some bugs we found (as we will describe in a later chapter), which might be a byproduct of this complexity.

The past, present and future of the project

This chapter will describe the past, present and future of Materialize.

Past
The package was created by 4 students of Carnegie Mellon University. The first commit was made September 7 2014. The framework was used by third-party Android-apps and web developers when designing user interfaces that adhered to Google’s design philosophy.

Present
The last commit for a functional change was on October 30, 2019. After this the creators have only made commits where they have updated links to sponsors and similar. The project is currently watched by 1k people, has 4,9 k forks and 38,6 k stars on Github [5].

Our conclusion is that the project is not currently being maintained by the developers, but that it is still used by a somewhat large portion of the community. We ourselves found this package and used it in another project because of how easy it was to integrate with vanilla Javascript, as you can simply load the CSS and Javascript files over one of many CDN's providing this package.

This package still appears high in various lists and blog-posts mentioning Javascript framework implementing a materialistic design, which is a sign of life for the package. One example is this article by the cloud hosting provider DigitalOcean [6].

Future
Since the official release is still 1.0.0 (same as the September 2018 release) we don't expect this package to have any more releases in the future. Bug-fixes may be possible, but we also find that unlikely judging by the low activity in the repository.

Beyond this single package, however, we see a bright future for the Material Design concept in general. Mostly due to the endorsements by Google, who are following these principles in their current projects, such as Android and Flutter.

One reason for the decline of Materialize may be Google's own package that covers similar use-cases, Material Design 3. This package appears to be the new design standard for Google and their applications [7], thus creating an incentive for developers to switch frameworks, if they want their UI to look more modern.

Current testing strategy

The current testing strategy of Materialize involves both manual testing and automated unit tests.

Automated testing

The framework uses a jQuery variation of the package Jasmine [8]. Testing of all the components is done by running the command *"grunt travis"* or *"npm test"*. 14 components are tested here by the framework. Each of these 14 components has an HTML page paired with a Javascript test-file.

The automatic test suite does not cover all components included with Materialize. An example of this is the component *"Scrollspy"* (which we coincidentally found some issues with, which we will describe more in the chapter "*Our testing*").

Basic overview of Jasmine

As described above, the tests come in pairs of one JavaScript file, where all the tests are specified, as well as one HTML file, where the site for the test is provided (concrete examples of these can be found further down in this chapter, under *"Example of pre-existing automated test"*).

When naming the JavaScript test file, it must contain the suffix "*Spec*" to be recognized as a test. The test structure itself follows the basic "*Arrange. Act. Assert.*" structure [12] found in many testing frameworks, and uses some interesting string parameters to the testing functions:

```
// The string describes the component that is under test.
describe('<component-under-test>', function() {
  // Define variables used in all tests.
  var importantVariable

  // Run before each "it"-statement.
  beforeEach(function() {
    // Loads the accompanied HTML test file.
    loadFixtures('<component-under-test>/HTMLTestFile.html')

    // Set values for the variables.
    var isVisible = true
    importantVariable = isVisible
  })

  // The string describes this sub-group of tests.
  describe('Sub-group of tests', function() {
    // The string describes what the component SHOULD do.
    it('should do X thing', function() {
      // Some code...

      // What the expected result is, BECAUSE of Y.
      // (NOTE: There are assertions other than "toBeVisible" that can be used)
      expect(importantVariable).toBeVisible('because of Y reasons')
    })
  })
})
```

A Jasmine test code snippet, showcasing the usage of string for clarifications as well as the general structure for the tests.

The HTML file includes the component and any needed triggers, such as buttons. An example can be seen below (from the “Modal” component tests):

```html
<!-- Modal Trigger -->
<a class="waves-effect waves-light btn modal-trigger" href="#modal1">Modal</a>
<button class="btn btn-floating fixed-action-btn modal-trigger" data-target="modal1">
  <i class="material-icons">menu</i>
</button>

<!-- Modal Structure -->
<div id="modal1" class="modal">
  <div class="modal-content">
    <h4>Modal Header</h4>
    <p>A bunch of text</p>
  </div>
  <div class="modal-footer">
    <a href="#!" class="modal-close waves-effect waves-green btn-flat">Agree</a>
  </div>
</div>
```

The HTML file used by the Javascript file for automatic unit tests.

When added to the specified test folder, tests created like this will run using the *“npm test”* command. A snippet similar to the one above would print a result like this:

<component-under-test>	← The first “describe” statement.
Sub-group of tests	← Nested “describe” statements.
√ should do X thing	← “It” statements.

Example of pre-existing automated test

The following example details the pre-existing automated unit tests for the “Modal” component and does, just as described above, consist of a JavaScript file and an HTML file. The included HTML file is the same as the one shown in the previous figure, and a snippet from the JavaScript test file can be found below:

```
it('Should open and close correctly', function(done) {
  modal1.modal();
  expect(modal1).toBeHidden('Modal should be hidden');

  click(trigger1[0]);

  setTimeout(function() {
    expect(modal1).toBeVisible('Modal should be shown');
    expect(modal1.hasClass('open')).toEqual(true, 'Modal should have class open');

    // Check overlay is attached
    var overlay = M.Modal.getInstance(modal1[0]).$overlay;
    var overlayInDOM = $.contains(document, overlay[0]);
    expect(overlayInDOM).toEqual(true, 'Overlay should be attached on open');

    click(overlay[0]);
    setTimeout(function() {
      expect(modal1.hasClass('open')).toEqual(false, 'Modal should have class open removed');

      var overlayInDOM = $.contains(document, overlay[0]);
      expect(overlayInDOM).toEqual(false, 'Overlay should be removed on close');

      done();
    }, 500);
  }, 500);
});
```

The Javascript file that performs the automatic unit tests.

In this case, the unit test opens a modal and then expects said modal to be of the “open” class. This is done by making use of jQuery’s functionality to programmatically interact with the components at play, and the result is then verified by the built-in checking functions.

The results of the pre-existing unit tests

The results of the pre-existing automated unit tests can be found in the table below (run using the *“npm test”* command), but the main take-away is that all tests pass:

```
Autocomplete Plugin
  Autocomplete
    √ should work with multiple initializations
    √ should limit results if option is set
    √ should open correctly from typing
    √ should open correctly from keyboard focus
Cards
  reveal cards
    √ should have a hidden card-reveal
  image cards
    √ should have an image that fills to full width of card
  sized cards
    √ should have small card dimensions
```

```
    √ should have medium card dimensions
    √ should have large card dimensions
Carousel
  carousel plugin
    √ No wrap next and prev should not overflow
Chips
  chips plugin
    √ should work with multiple initializations
    √ should be able to add chip
    √ should be able to delete chip
    √ should have working callbacks
Collapsible Plugin
  collapsible
    √ should open all items, keeping all open
    √ should allow preopened sections
    √ should open and close programmatically with callbacks
  accordion
    √ should open first and second items, keeping only second open
  popout
    √ should open first and popout
Dropdown Plugin
  Dropdown
    √ should open and close programmatically
    √ should close dropdown on document click if programmatically opened
    √ should bubble events correctly
    √ hovered should destroy itself
Fab
  Floating Action Button
    √ should open correctly
  FAB to toolbar
    √ should open correctly
Materialbox:
  Materialbox opens correctly with transformed ancestor
    √ Opens a correctly placed overlay when clicked
Modal:
  Modals
    √ Should open and close correctly
    √ Should open and close correctly with children elements in trigger
    √ Should have a dismissible option
    √ Should have callbacks
Sidenav Plugin
  Sidenav
    √ should not break from multiple initializations
    √ should open sidenav from left
    √ should have working callbacks
    √ should destroy correctly
Tabs Plugin
  Tabs
    √ should open to active tab
    √ should switch to clicked tab
    √ shouldn't hide active tab if clicked while active
```

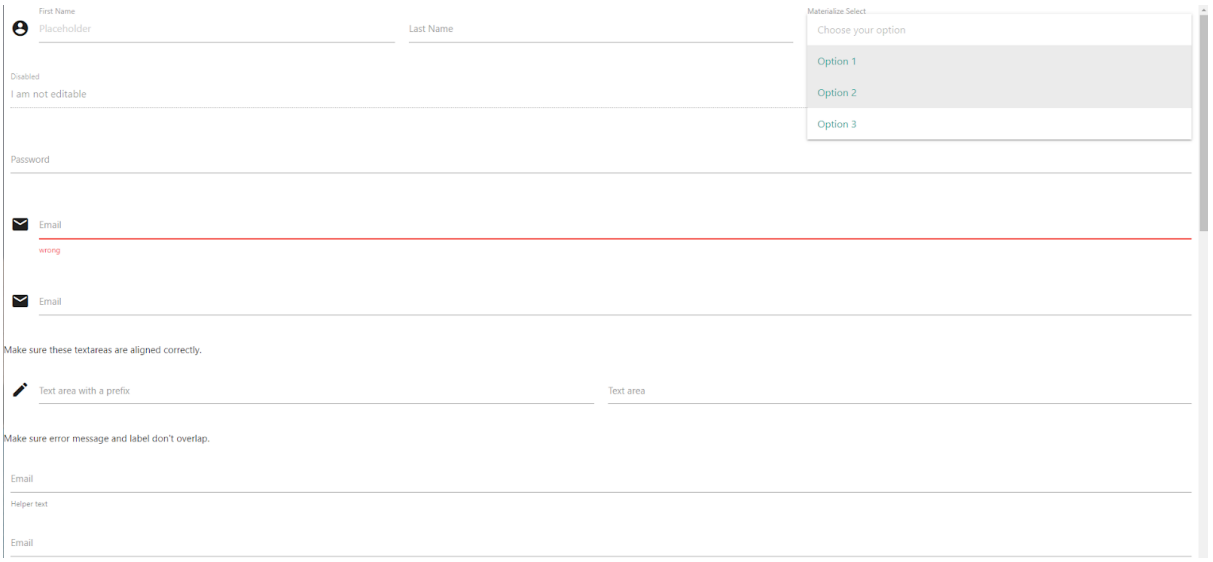
```
    √ should horizontally scroll when too many tabs
    √ should programmatically switch tabs
    √ shouldn't error if tab has no associated content
 Toasts:
   Toast javascript functions
    √ should display and remove a toast
    √ Opens a toast with HTML content
    √ Toasts should call the callback function when dismissed
    √ Apply two custom class to a toast
 Tooltip:
   Tooltip opens and closes properly
    √ Opens a tooltip on mouse enter
    √ Positions tooltips smartly on the bottom within the screen bounds
    √ Removes tooltip dom object
    √ Changes position attribute dynamically and positions tooltips on the right correctly
    √ Accepts delay option from javascript initialization
    √ Works with a fixed position parent

50 specs in 24.1s.
>> 0 failures
```

The manual tests do not have clear cut “FAIL/OK” results, per se, as these are mostly used to view the different components in action. We did, however, find some issues with a few of the components, which can be found in the chapter “*Our testing*”.

Manual testing

In addition to the unit tests the library also contains multiple HTML pages where the components are already used. The purpose of these pages are to let the user visually see the elements and integrate with them. These HTML pages are not the same as the one's loaded by the automatic tests.

A screenshot of one of the pre-built HTML pages can be found below. It has multiple input components that can be clicked / written into to manually test their behaviours. The placeholders make the desired behaviour fairly obvious, for the most part. Where the intended behaviour is not as obvious, small help snippets have been included to guide the user on the right track.



One of the provided pre-built HTML pages used for exploratory testing.

Our testing

In this chapter we will explain how we tested Materialize. This includes some exploratory testing, as well as writing our own automated unit tests.

.

Automatic Testing

In order to better understand the automated unit tests, we ran the pre-existing tests ourselves (with the results found in the previous chapter). After that we studied the actual tests and their implementation, which then led us to create our own automated test.

Custom automated tests

We implemented the following custom automated tests to learn more about the framework. We followed the same structure as the pre-existing tests, where they used one folder per test that contained the HTML file, and one with the JavaScript file.

The way the tests work is that the JavaScript file finds triggers for the component, activates them, and then checks for the results. The results could for instance be that a certain class has been applied to an element in the HTML fixture element.

AutomaticTest for testing that a "modal" can be opened inside another modal

To run this specific custom made test (clone the latest repo version first [https://github.com/lhuber92/materialize]), then run the command "*grunt travis --filter=customModal*".

customModalSpec.js

tests > spec > customModal > customModalSpec.js > describe('Modal:') callback > describe('Custom Modal Tests') callback > it('

```
describe( 'Modal:', function() {
  beforeEach(function() {
    loadFixtures('customModal/customModalFixture.html');
    // Custom tests:
    outerModalTrigger = $('.btn[href="#outer-modal"]');
    innerModalTrigger = $('.btn[href="#inner-modal"]');
    outerModal = $('#outer-modal');
    innerModal = $('#inner-modal');
  });

  describe('Custom Modal Tests', function() {
    it('Nested modals should be opened', function(done) {
      outerModal.modal();
      click(outerModalTrigger[0]);

      setTimeout(function() {
        expect(outerModal).toBeVisible('Outer modal should be shown');
        expect(outerModal.hasClass('open')).toEqual(true, 'Outer modal should have class open');

        innerModal.modal();
        click(innerModalTrigger[0]);

        setTimeout(function() {
          expect(innerModal).toBeVisible('Inner modal should be shown');
          expect(innerModal.hasClass('open')).toEqual(true, 'Inner modal should have class open');

          done();
        }, 500);
      }, 500);
    });
  });
});

```

The Javascript file that performs the modal test

customModalFixture.html

tests > spec > customModal > customModalFixture.html > div#inner-modal.modal.bottom-sheet

```
<!-- Modal Trigger -->
<a class="waves-effect waves-light btn modal-trigger" href="#outer-modal">Modal</a>

<!-- Modal Structure -->
<div id="outer-modal" class="modal bottom-sheet">
  <div class="modal-content">
    <h4>Modal Header</h4>
    <!-- Nested Modal Trigger -->
    <a class="waves-effect waves-light btn modal-trigger" href="#inner-modal">Modal</a>
  </div>
  <div class="modal-footer">
    <a href="#!" class="modal-close waves-effect waves-green btn-flat">Agree</a>
  </div>
</div>
<!-- Modal Structure -->
<div id="inner-modal" class="modal bottom-sheet">
  <div class="modal-content">
    <h4>Modal Header</h4>
    <p>A bunch of text</p>
  </div>
  <div class="modal-footer">
    <a href="#!" class="modal-close waves-effect waves-green btn-flat">Agree</a>
  </div>
</div>
```

The HTML file that is used by the paired Javascript file

```
Modal:
  Custom Modal Tests
    - Nested modals should be opened......√

1 spec in 1.087s.
>> 0 failures

Done.
```

The result of running the automated unit test

The manual test above is fully automatic. To test the same behaviour we could click around using the pre-built manual testing html files like the gif below:

Click the link below to view the gif:
https://github.com/lhuber92/1dv609-a2/blob/main/modal.gif
Using one of the pre-built exploratory testing HTML pages

Automated unit test for nested “waves” component inside “collapsible”

This test was intended to check that a “waves” component could be nested inside a “collapsible” component. We had some issues getting the entire test to work, mainly due to inexperience with Jasmine, but in the end we learned a lot from the exercise. The JavaScript file looks like this:

```
describe('Custom Collapsible/Waves Test', function() {
  var collapsible, image

  beforeEach(function() {
    loadFixtures('customCollapsible/customCollapsibleFixture.html')
    collapsible = $('.collapsible')
    image = $('.waves-effect')

    collapsible.collapsible()
  })

  describe('Test nested waves in collapsible', function() {
    it('should be hidden before the collapsible has expanded', function() {
      expect(image[0]).toBeHidden('because collapsible bodies should be hidden initially.')
    })

    it('should be visible when expanded', function(done) {
      var clickable = collapsible.find('.collapsible-header')
      clickable.click()

      expect(image[0]).toBeVisible('because collapsible bodies should be visible after they are opened')

      done()
    })

    it('should emit waves when clicked', function(done) {
      var header = collapsible.find('.collapsible-header')

      expect(image[0]).toBeHidden('because collapsible should be closed')
      header.click()

      setTimeout(function() {
        expect(image[0]).toBeVisible('because collapsible bodies should be visible after they are opened')

        var emitter = collapsible.find('.activator')
        emitter.click()

        setTimeout(function() {
          var waves = $('.waves-ripple')
          expect(waves[0]).toBeVisible('waves should be visible when clicked')
          done()
        }, 25)
      }, 100)
    })
  })
})
```

The JavaScript file for the nested waves test.

And the HTML file was the following:

```
<!-- Collapsible -->
<ul id="test-1" class="collapsible" data-collapsible="accordion">
  <li>
    <div class="collapsible-header"><i class="mdi-maps-place"></i>This is the header.</div>
    <div class="collapsible-body waves-effect waves-block waves-light">
      <img class="activator" src="https://images.unsplash.com/photo-1439694458393-78ecf14da7f9?q=80&fm=jpg&s=643e44f5177366de6cc914c8bcfbe064" width="100%">
    </div>
  </li>
</ul>

<!-- Scripts -->
<script src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
<script src="../../../bin/materialize.js"></script>

<script type="text/javascript">
  $( document ).ready(function() {
    $(".sidenav").sidenav({
      edge: 'right', menuWidth: '90%'
    });

    $('select').select();

    $('.collapsible').collapsible();

    $('.collapsible[data-collapsible="expandable"]').collapsible({
      accordion: false
    });

    $('#callbacks').collapsible({
      accordion: false, // A setting that changes the collapsible behavior to expandable instead of the default accordion style
      onOpen: function(el) { alert('Open'); }, // Callback for Collapsible open
      onClose: function(el) { alert('Closed'); } // Callback for Collapsible close
    });
  });

  $( document ).ready(function() {
  });

  </script>
```

The first test makes sure that the image the “waves” have been attached to is not visible from the start, as the collapsible has not been expanded. The image is not visible, and this test passes.

The second test clicks the collapsible to expand it, and then checks whether the image is visible in the collapsible body. The image is visible, and this test passes.

The third test is where the issues arose. Here, we click the image to emit waves from the clicked position, then we set a timeout and check whether or not an element with the class “waves-ripple” exists. Here we get the error that we cannot expect undefined to be visible, which is most likely an error stemming from the fact that the element emitting the waves is dynamically added to the DOM, which might not be detectable in the way we tested.

The results from the tests are hence the following:

```
Custom Collapsible/Waves Test
  Test nested waves in collapsible
    √ should be hidden before the collapsible has expanded
    √ should be visible when expanded
    × should emit waves when clicked
      Expected undefined to be visible 'waves should be visible when clicked'. (1)
```

The results from the tests above.

Manual Testing

We manually tested each component found in Materialize, using the previously mentioned HTML test files.

This process was repetitive and not very fun, but we found that the manual exploratory testing we did was a lot more helpful in understanding the framework than the automated unit tests ever could have been, which is not surprising as Materialize is a package used for developing user interfaces.

Something that alleviated this process immensely, however, was the actual inclusion of these files, as creating them ourselves to test each component would have been a lot of work (work that would have likely been error-prone, given our limited understanding of the framework).

There were manual test HTML pages made for 18 components in total, as opposed to the 14 components that were included in the automated test cases. The manual tests included the following:

- Badges,
- Buttons,
- Cards,
- Chips,
- Collapsible,
- Carousel,
- Dropdown,
- Fixed Navbar,
- Forms,
- Materialbox,
- Multiple Modals,
- Multiple Sidenav,
- Overlay Z-Index,
- Pushpin,
- Scrollfire,
- Scrollspy,
- Tabs,
- Waves, and
- Class Nav-wrapper.

All tests were run using the “Live Server” extension in Visual Studio Code.

The majority of the manual tests show the intended behavior of the highlighted components, at least according to our understanding of what the components’ behaviors are supposed to be. The tests we managed to find issues with were:

- Chips,
- Dropdown,
- Materialbox,
- Scrollfire,
- Scrollspy, and
- Class Nav-wrapper.

The issues we found in ***Chips*** (chips.html) were some strange behavior, where the "Telephone" input fields are interactive and the rest are unresponsive. Using the browser's built-in inspector we got the following error message:

> *"Uncaught TypeError:$(...).material_chip is not a function"*

We tried doing some troubleshooting to try to rectify this issues, including using the autocomplete functionality of Chips instead, like this:

```
$(document).ready(function() {
        $('.chips-autocomplete').material_chip({
          autocompleteData: {
                'Apple': null,
                'Microsoft': null,
                'Google': null
          }
        });
});
```

We also tried running the code in the browser using the provided packages with the *"grunt travis"* command, as well as downgrading our version of jQuery, but we still got the same error message.

After some research we found that the way the component is initialized in the documentation [10] differs from the way it was in the pre-built HTML-page. We assume this was updated in a previous version of Materialize, and that the developers simply forgot to update the corresponding manual test file.

To make the Chip component work we replaced this:

```
<body>
  <div class="chips chips-autocomplete input-field" id="test"></div>

  <script src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
  <script src="../../../bin/materialize.js"></script>
  <script type="text/javascript">
    $('.chips').chips();
    $(document).ready(function() {
      $('#chips-autocomplete').material_chip({
        autocompleteData: {
          'Apple': null,
          'Microsoft': null,
          'Google': null
        }
      });
    });
  </script>

</body>
```

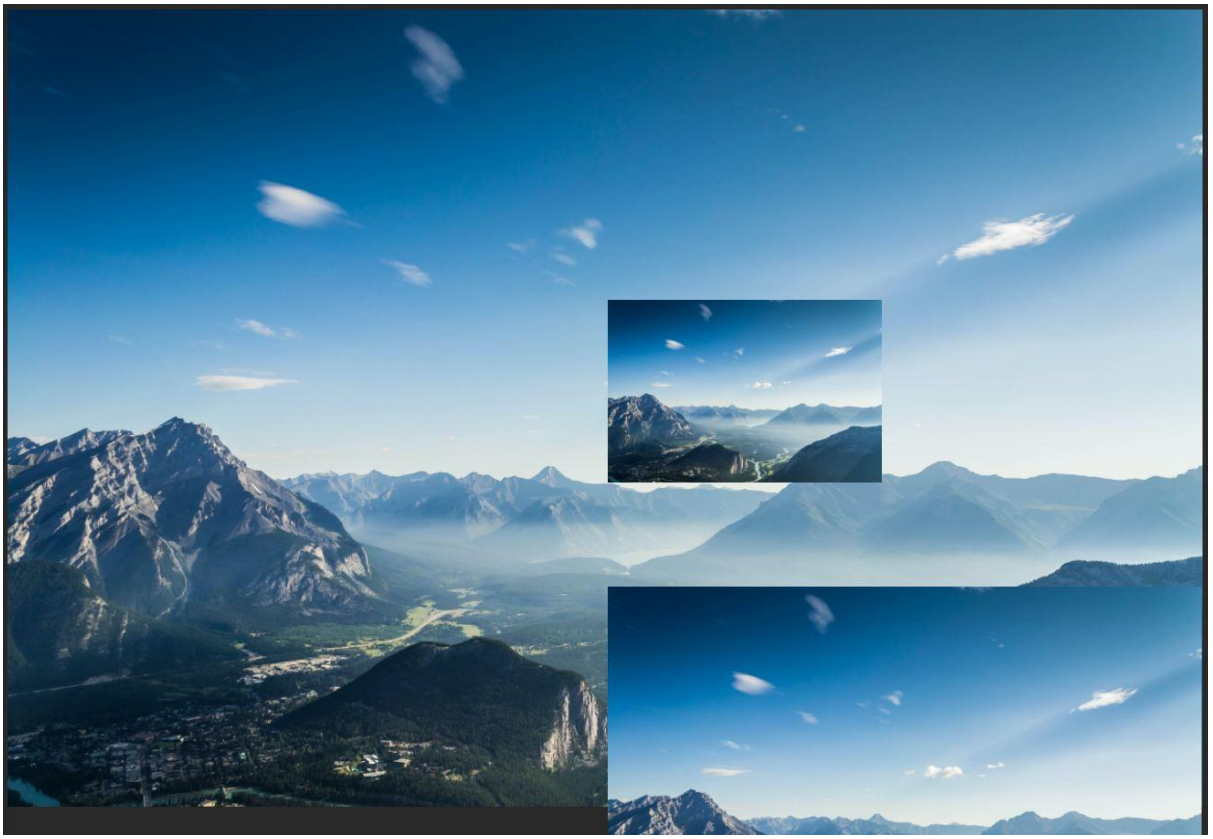
The file before modification.

With this:

```
<body>
  <div class="chips chips-autocomplete input-field" id="test"></div>

  <script src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
  <script src="../../../bin/materialize.js"></script>
  <script type="text/javascript">
    $('.chips').chips();
    $('.chips-autocomplete').chips({
      autocompleteOptions: {
        data: {
          'Apple': null,
          'Microsoft': null,
          'Google': null
        },
        limit: Infinity,
        minLength: 1
      }
    });
  </script>
</body>
```

The file after modification.

The ***Materialbox*** test had some minor issues to it, being that there was some incorrect Z-ordering on the page, where some images are still on top of the image in focus. Aside from the issues displayed in the image below, the component seems to function as intended:



Z-ordering issue in Materialbox.

With ***Overlay Z-index*** and ***Pushpin*** the desired behavior was not made entirely clear, which meant that we were unable to properly observe whether or not the components worked as intended. In the case of the Pushpin component, there were some buttons on the test page that modified certain CSS rules of the components, but in the end we were still unable to determine the intended behavior with confidence.

Scrollfire was not responding during our testing. Upon further troubleshooting we once again found an error displayed in the browser's console, with the message:

Uncaught ReferenceError: Materialize is not defined

Seeing as this was similar to the error we found while testing Chips, we went to the Materialize documentation to find the proper way to initialize the component. To our surprise it was not listed as a component in the documentation. It appears that the Scrollfire component is deprecated and that the developers forgot to remove the HTML test page for it. This is further supported by the fact that the component appears on older mirrors of the documentation [9].

The ***Scrollspy*** component also had some issues with it. This component "fires" whenever a certain scroll level has been met, but it fails to fire as it reaches the last section on the site:

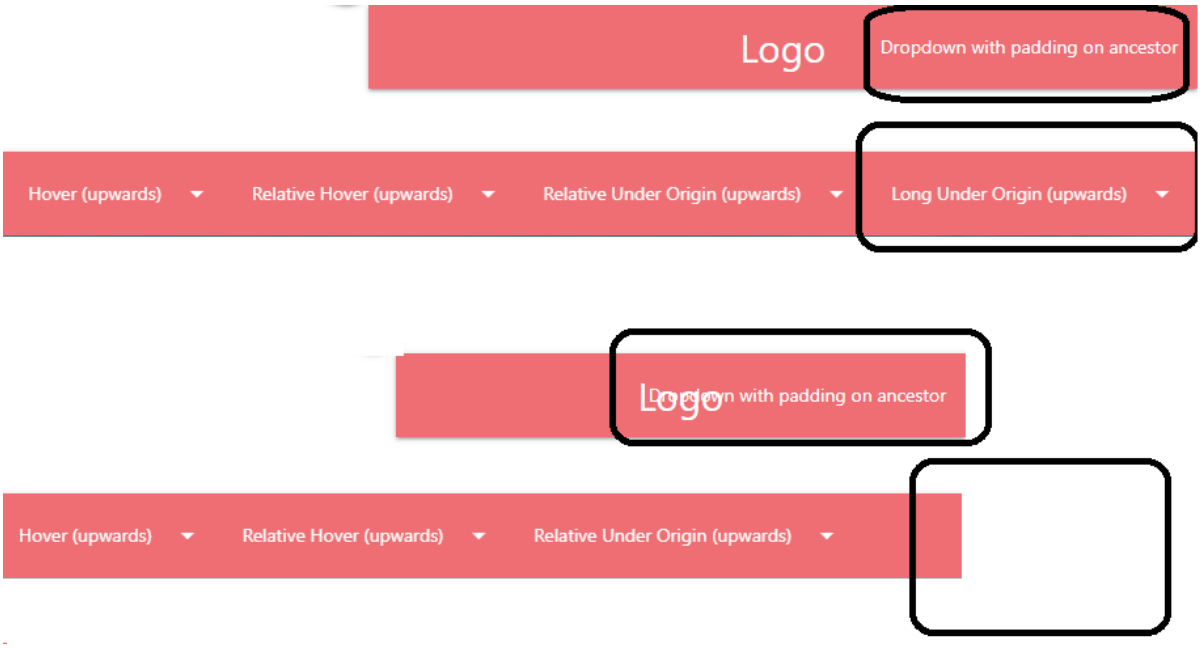
An error encountered in the Scrollspy component (the highlighted color should be "Red").

Furthermore, when the "HALF HEIGHTS OF SECTIONS" option has been selected, the component fails to fire for the third section as well. We suspected that this might have something to do with a rounding error, or that the component only fires when the first pixel has been passed, meaning that it would fail to recognize the scrolling unless it goes one pixel above the intended threshold. However, when we tried adding content beneath the last section, we could not observe any noticeable difference.

When testing the previously mentioned components, we found a few issues with the ***"nav-wrapper"*** class (which were most pronounced in the ***Dropdown*** component):

- If the content to the right of the centered logo is too wide/long, it will overlap with the logo. This is something that could be fixed by aligning the logo to the left if the content is too long, or by simply escaping the content to minimize its size.
- If elements contained within the nav-wrapper class are simply hidden if they cannot be wrapped within the parent element, meaning that some elements have a risk of disappearing. We did not find the root of this, but a potential fix would be to add another row beneath the other elements, instead of hiding the overflowing ones.

A demonstration of these issues can be found in the following figure:



Issues with text overflow and dropdown element overflow.

Sources

1. *About - Materialize*, https://materializecss.com/about.html
2. *Material Design*, https://material.io/
3. *Introduction*, https://material.io/design/introduction#principles
4. *Ionic - The Cross-Platform App Development Leader*, https://ionic.io/
5. *Dogfalo/materialize*, https://github.com/Dogfalo/materialize
6. *Make Material Design Websites with the Materialize CSS Framework*, https://www.digitalocean.com/community/tutorials/make-material-design-websites-with-the-materialize-css-framework
7. *Material Design 3*, https://m3.material.io/
8. *velesin/jasmine-jquery*, https://github.com/velesin/jasmine-jquery
9. *ScrollFire - Materialize*, https://www.um.es/docencia/barzana/materializecss/scrollfire.html
10. *Chips - Materialize*, https://materializecss.com/chips.html
11. *Showcase - Materialize*, https://materializecss.com/showcase.html
12. Alexander Tarlinger, “Chapter 7”, in *Developer Testing: Building Quality into Software*, Pearson, 2017