[1. General 2](#_Toc43446034)

[2. Terminology 3](#_Toc43446035)

[3. Testing 4](#_Toc43446036)

[3.1 Balanced testing model: 4](#_Toc43446037)

# **General**

# **Terminology**

# Testing

Functional UI – testing the user interface as if a real user was interacting with our app through the browser.

Subcutaneous – just under the graphical user interface – testing HTTP requests to our app.

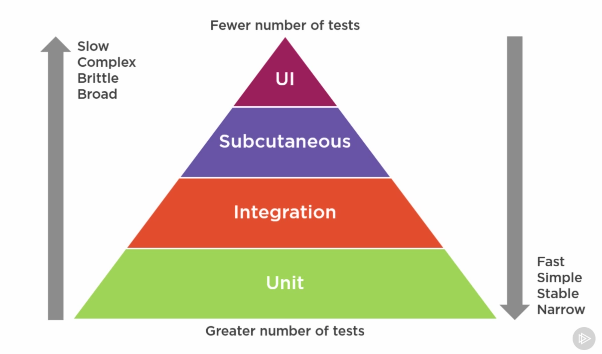
Integration – test different parts of the system together.

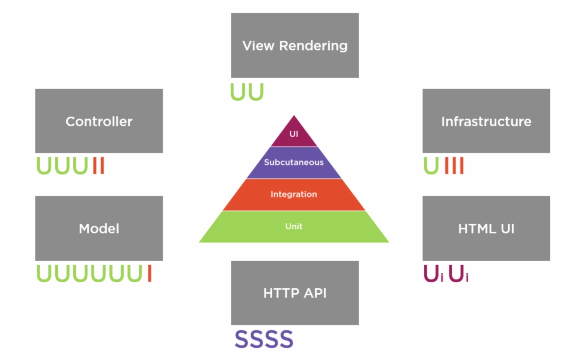
Unit – tests a single class

## Balanced testing model:

The combination of tests that will give us the most cost-to-value benefit.

The Testing Pyramid:





# Debugging

## Men In the Middle

Allow re-directing

## Requestly

Intercept, modify and redirect HTTP Requests

<https://requestly.io/>

## Chrome

<https://github.com/jkup/mastering-chrome-devtools>

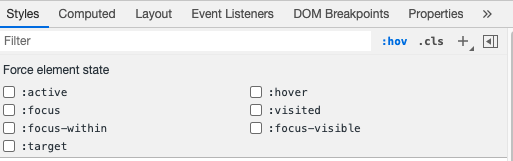
**Note:** Chrome extensions change the way a page is rendered. So when debugging something important, you should disable them by windows->extensions or by opening an incognito tab (which comes with extensions turned off by default). Another good option is to use Chrome for browsing and Chrome Canary for example for debugging without any extensions installed.

* To test how the page is returned to Google search:
  + Go to the page in the browser
  + Open dev-tools, press the … on the right -> more tools -> Network Conditions
  + Uncheck the ‘User agent -> Select automatically’ and choose ‘Googlebot’
  + Refresh the page
  + You should now see the returned data in the page’s assets.
* To search text in any of the source files loaded into the page: Cmd+Opt+f
* When in doubt, double click to see how you can interact with the elements etc.

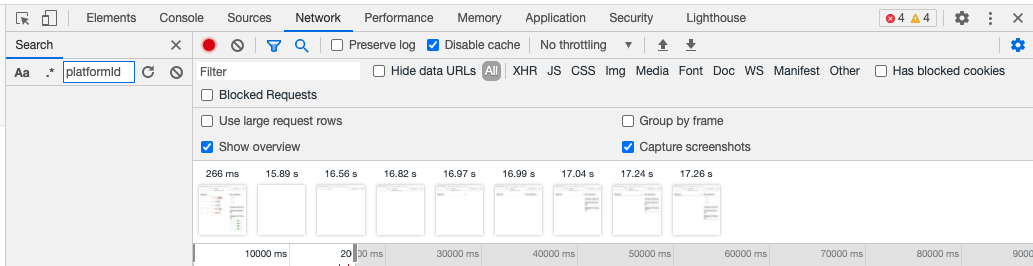
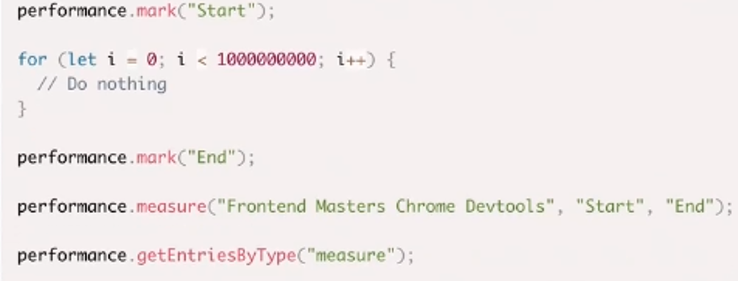
### General

* Use the device types to test on important devices
* You can plug in your android device and debug on it using the Chrome dev tools. See additional info here:

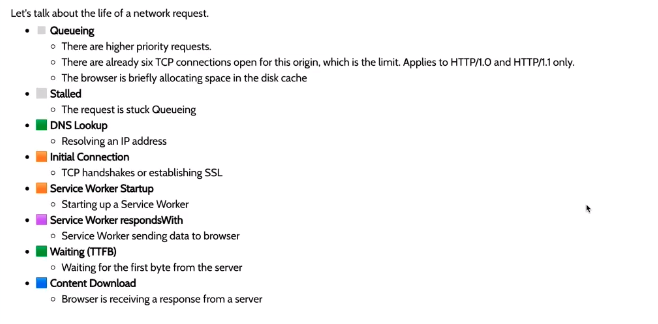
### Elements Panel

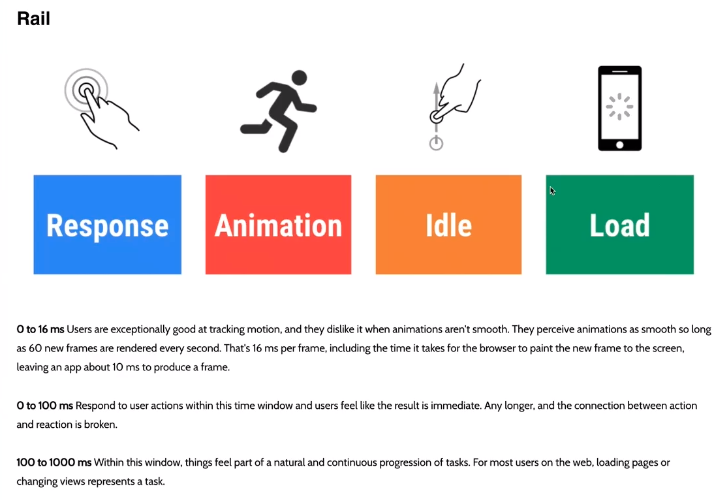
* Almost all the elements, tags, ids etc are clickable and draggable.
* In Styles, you can add new styles
* Use the element hover () to see additional information of each element as you hover over the page and go into it in the elements panel when you click on it.
* On Element -> right click -> scroll into view – will mark the element in the page.
* Click on an element :
  + h – will hide the element
  + del or right-click->delete will remove it from the dom
* to see hover states: click on the element.
  + In the right panel: Styles-> :hov you can choose the different states to look at.  
    
* Compute styles:
  + Choose an element. And in style->Computed is the final computed css for each element and you can drill down into each property to see how it was computed.  
    It also show you the computed boarders.
* Breakpoints on an element: choose element -> right click -> break on and choose on what to break. When it breaks, you can use ‘Call Stack’ to see which line caused this breakpoint.
* Find event listeners: choose the element, in the right hand side, ‘Event Listeners’ (next to the Style tab)
* Choosing colours – press on the colours box and you can also choose with a picker, using pre-made pelletes and even repopulate with the page’s main colours.
* Accessibility tab – in the right hand side (next to style) – to see your accessibility definitions.

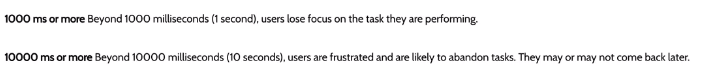
### Network

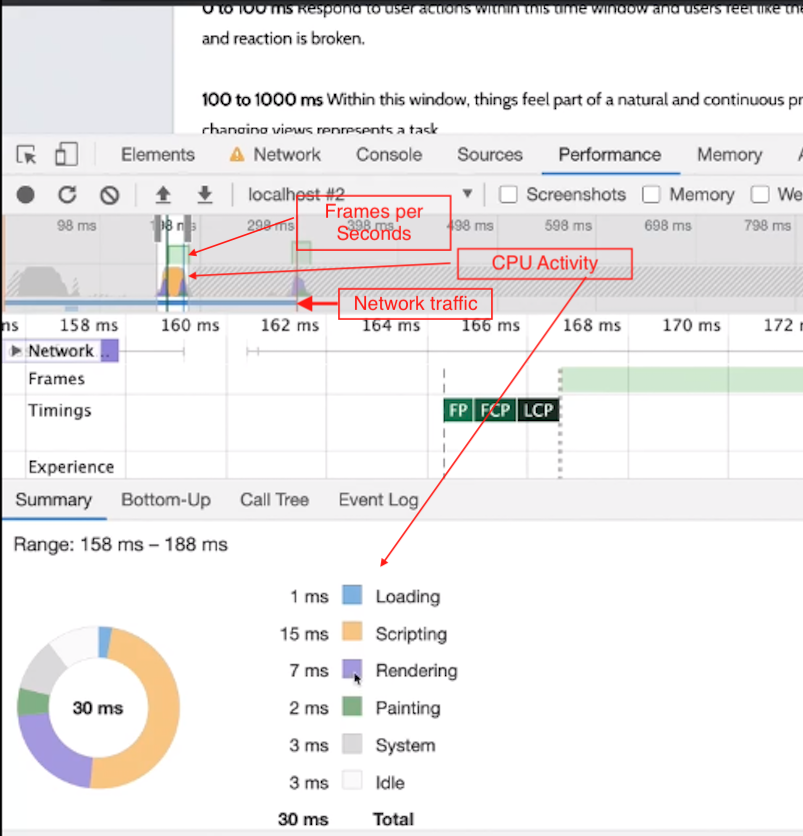
* The top will always be an html file with ‘document’ type.
* The browser will go over it and will all additional files and scripts when it discover them.
* Important events are also listed at the bottom – DOMContentLoaded and Load time
* Capture all the stages of the page loading with dev-tools setting->Capture screenshots:  
  
* Initiator – who called this?
* Measuring real user performance:
  + Even if it’s fast for me,
  + 
  + Performance API is very strong and useful tooling you can use to measure and improve your app performance

### Performace

* Go into the tab, click record, interact with the site and press stop.
  + Try to limit your recording to exactly what you need to limit the noise.
  + Hold Shift and hover over the items – everything this call called, are marked red and the request that called your request is marked green
  + You can also add additional columns
* Note: there is a limit to the number of concurrent requests (~6). So if you have 20 requests to get images for example, the browser will call them in groups.
  + You can also add the priority order which will show you the priorities that the browser order your requests.
  + You can override the priority order. You can pre-load or defer and you can move things.
  + 
* Time To First Byte (TTFB) – how long it take for the 1st byte of your data to get to the browser. Slow here will usually mean server issues (the server takes too long before it answers)
* Content Download – how long it takes from TTFB until all your content is downloaded. Slow here will usually mean that you’re sending too much data.
* Note: when you compress your data to make the download quicker, you should also take into account the time to un-compress. So usually, we’ll want to think about sending less JS off the wire and not only compress it.
* Google Rails’ performance Model:





* Page Jank – any stuttering, juddering or just plain halting that users see when a site or app isn’t keeping up with the refresh rate.
* Understanding 60 Frames Per Seconds (FPS) – Most devices refresh their screens 60 times per second. Therefore, for completely smooth animation, you should be able to serve the browser a new frame every 10 ms. Otherwise, the browser will start dropping frames. Note: the browser also does other things, therefore, you don’t have the full 16.6 ms for your app and you’ll probably only get around 10ms per frame.
  + 
  + If the browser is dropping frames, you will also see read lines above the ‘Frames Per Second’ every time a frame is dropped:  
    
* Pragmatic Approaches to reduce Page Junk:
  + Use requestAnimationFrame – instead of reading and writing the DOM (and by this invalidating the cache which will make subsequent reads take longer) all over the place – you can use requestAnimationFrame to do all these things just before the browser is next set to refresh the page (and therefore, to invalidate the cache anyway).   
    A bigger takeaway here is – try to not use vanilla JS. Instead use some front-end framework that will do the right thing (inc. requestAnimationFrame) for you.
  + Use web Worker – to kick off big tasks in the front end. This will create a new thread that won’t stop the page from refreshing.
  + Use framework
  + Reduce complexity.

### Sources

#### Workspaces

* To persist code/html/css – if what you’re seeing on the web-page is what you send from the server, you should be able to connect your workspace to the browser so that everything you save in the browser is actually being saved to your hard disk.
* Instead of saving to disk, we can set it up so it will survive a refresh, even if we can’t save to disk.
* For linux windows sub-system or when working remove, sometimes, although the files are being saved to the file system.

#### Overrides

* If you don’t want to persist your changes to disk but you do want to make changes and have them persist past refresh:
* In the left tab (next to filesystem) choose ‘Overrides’.
* Give Chrome a folder where it can store all your temporary changes.
* Now it will automatically persist your override and you’ll be able to refresh without losing your changes.

#### Step Through Debugging

In the Debug Tab:

* XHR/fetch breakpoints – when your app is doing a lot of API calls – you can break only when it calls a specific URL
* Scope – show us all the scopes we have access to
* Call Stack
  + If you use framework libraries – you can right-click->blackbox it to stop seeing it in the call stack if you know that the issue is not there.

### Console

* Go to the element tab and click on an element.
  + In the console $0 is a reference to the thing you last clicked on, $1 the previous element etc.
  + You can now manipulate it etc.

### Security

Show you the security state of your site. You can also look at the certificate if you need to.

### Application

* Storage – all your local data, chache, cookies etc
* Storage->Clear Site Data – to clear all local data if it got corrupted.

### Memory

* Allocation sampling – can be used to find memory leaks source
* Memory Leaks:  
  3 great tools to detect memory leaks in Chrome:
  + Chrome Task Manger: ;;;->More Tools->Task Manager  
    right-click on the ‘Memory Footpring’->JavaScript Memory
  + Performance Tab:  
    You can click on Memory to get
  + Memory Tab:
    - Heap Snapshot – will show us everything that is taking our memory:
      * Shallow Size – how big the object’s memory is
      * Retailed Size – how memory could be free if the object is deleted (include all dependencies on this object that will be grabaged collected once it’s deleted)
    - Allocation Instrumentation on Timeline – memory allocated over time
    - Allocation sampling: shows which method created the memory leak.
    - Note: you can click the garbage bin to see if the memory leak will consist across garbage collection of if it’s transitional only.

### Lighthouse

* Open source – audit for website best practices
* Always

### Running Node.js on dev tools:

Instead of running npm start which is set to run node server/server.js from the commanline, run:  
node –inspect server/server.js

It will load the app as normal but will have an additional Node.JS dev tool suite that allow us to debug our backend code in the browsers as well.