Table of Contents

[I. Views & Partials 3](#_Toc461618089)

[II. Layouts 4](#_Toc461618090)

[III. server.js 7](#_Toc461618091)

[a. Modules & Server Setup 7](#_Toc461618092)

[b. Functions 7](#_Toc461618093)

[c. Get Requests 9](#_Toc461618094)

[IV. client.js 10](#_Toc461618095)

[a. Loading Screen 10](#_Toc461618096)

[b. Anchor Transitions 10](#_Toc461618097)

[c. Scroll-to-Top Button 11](#_Toc461618098)

[d. Next/Previous Page Buttons 11](#_Toc461618099)

[e. Acronym Modal 12](#_Toc461618100)

[V. style.css 13](#_Toc461618101)

[a. Global 14](#_Toc461618102)

[b. Header Section 15](#_Toc461618103)

[c. Main Section 16](#_Toc461618104)

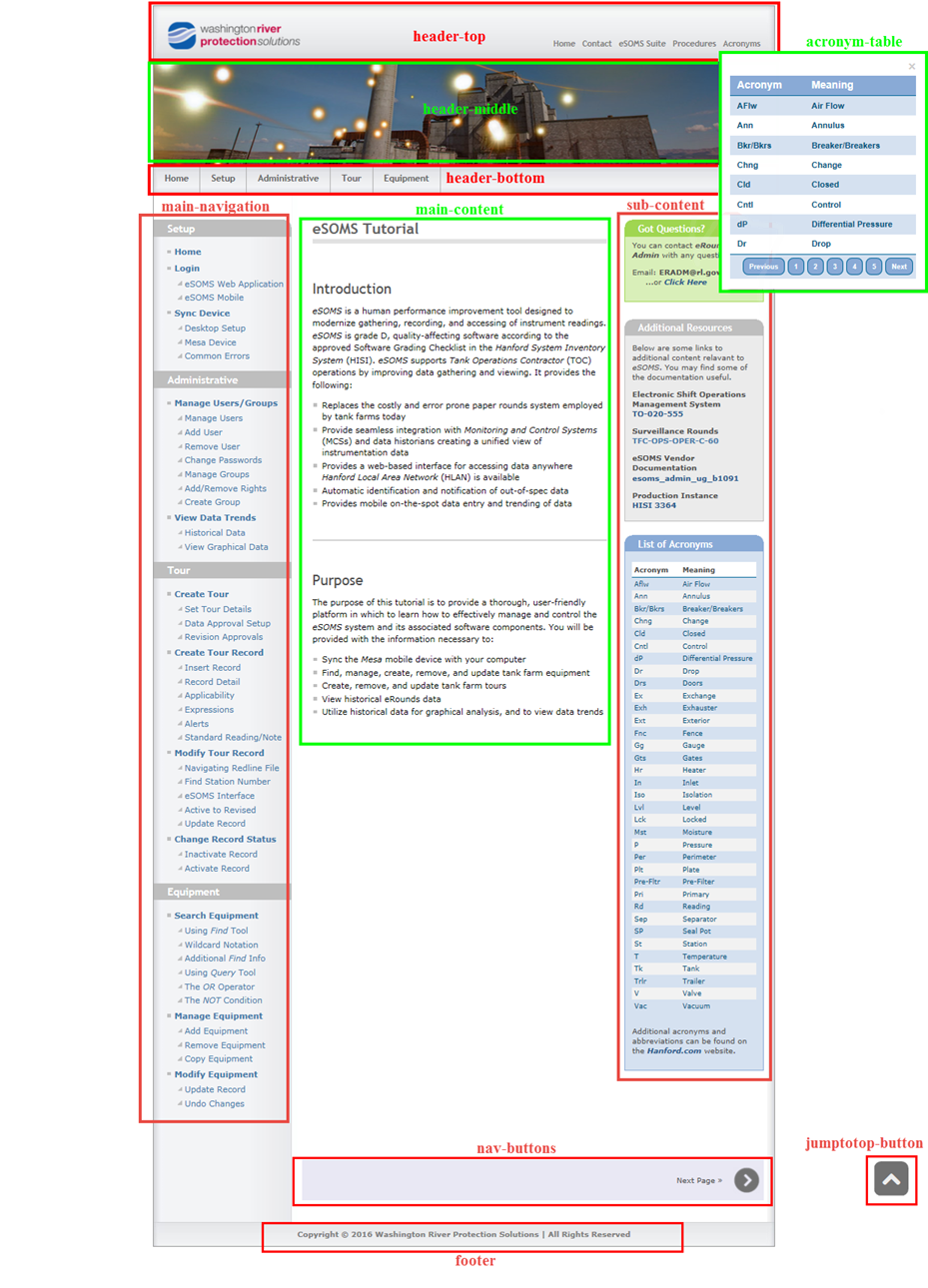
[d. Remaining CSS 19](#_Toc461618105)

[VI. Making Changes to Website 21](#_Toc461618106)

[a. *DCAF* Changes 22](#_Toc461618107)

[b. *SCR* Changes 23](#_Toc461618108)

[c. Formatting Convention 24](#_Toc461618109)



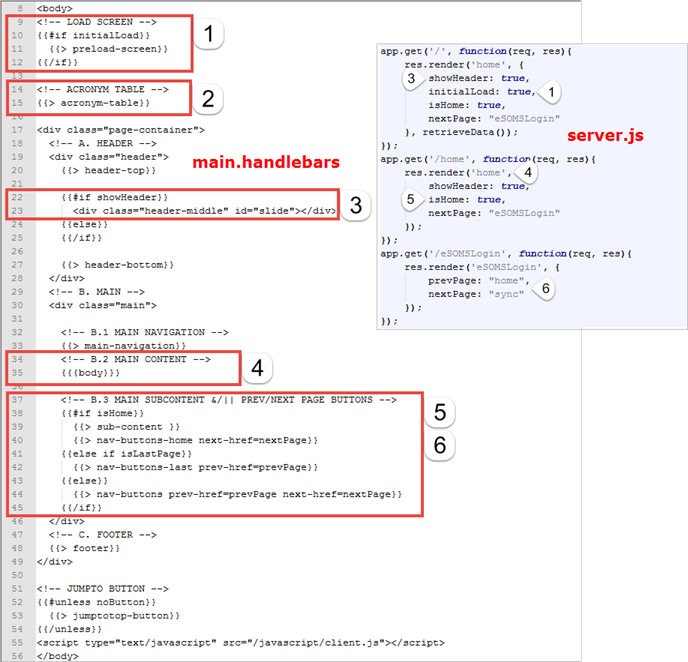
# Views & Partials

The image above shows where the different *views* and *partials* for the webpage can be found. The *views* and *partials* are simply files which contain *HTML* code, and that are organized in the “main.handlebars” layout file. This can make traversing the code in a website easier by breaking the code up into manageable pieces.

The *views* make up the main part of the page; they are the part of the site which changes from page to page. The *views* are visible in the “main-content” section shown on the image, and there is a single view used to represent each individual page for the site.

The *partials* make up sections of the site which never change. There can be logic incorporated into the *layout* file to determine whether a partial is present for a particular page or for what condition it is present in a page, but a partial will never change. The other sections represented in the image above are each represented by a partial. The “sub-content” *partial* and the “header-middle” are only present on the homepage, and the “nav-buttons” *partial* is different for the homepage, the last page, and the remaining pages. This is because the homepage has only a *Next Page* button, the last page has a *Previous Page* and a return to *Home Page* button, and the rest of the pages have both a *Next Page* and a *Previous Page* button. The logic governing when these *partials* are present is found in the “layout” file.

The *partials* not shown in this image are the “preload-screen” and the “head”. The “preload-screen” simply holds the *wrapper* for the loading screen which is only present during the initial website load, during the same time that the “main-content” is being retrieved from the database. This is an unnecessary feature, but makes the site more user friendly by replacing the pause created when data is being queried with a loading screen. The “head” file contains the content which is always present in an *HTML* head. It includes the site *title*, all associated *CSS* files, and all *javascript* files which must be loaded prior to the webpage content being loaded.



# Layouts

Above is an image of the “main” file which is the *layout* file, and a corresponding section of the “server.js” file. It may become necessary to understand some of the logic which is unique to node.js when making future changes some of these will be detailed here. One of the primary purposes of the “server.js” file is to render each page of the website. It is while rendering a page that node.js specific parameters can be passed to the *layout* file.

You may notice that the “acronym-table” partial denoted with the number 2 does not have a conditional block associated with it. This is because it is present in every page so there is no logic necessary. The “preload-screen”, denoted by the number 1, has a simple “if” statement associated with it.

Note: it is important to note that node.js is considered a “logic-less” framework, which essentially means that the associated conditional statements cannot have operators, but can only be true or false. This means that there is no “or” ( || ), no “and” ( && ), and no “greater/less than”, etc. ( >, <, >=, <=, !=, == ). This true or false must be passed as a parameter from the server.js when the page is rendered.

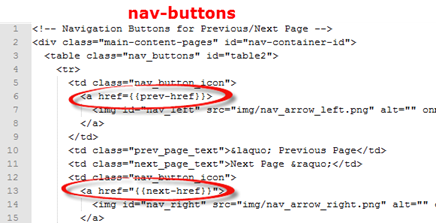
The “if” statement denoted by the number 1 reads, “if initialLoad is true then render the “pre-load” partial.” If you look at the server.js code segment you will notice a corresponding number 1 pointing at a line of code which reads, “initialLoad: true.” Anytime this block of code executes, so will the “preload-screen.”

Note: Something else important to note at this point is in the first line shown in the “server.js” segment. It is the *get* request which tells the server to render a specific page. The first parameter of the *get* function is a forward slash character ( / ). This means when there is no path included in the *URL* to render this block of code. The next line of code hold the *render()* function whose first parameter is the “home” page. You will notice that the next block also loads the “home” page but it is only executed when the path of the *URL* is “/home”. This is what allows the same page to be loaded but with different functionality. One includes the “preload-screen” and the other does not.

The conditional block denoted by the number 3 is similar in purpose. It states that “if showHeader is true, then execute the line of code.” It also contains an “else” condition which essentially reads, “else do nothing.” I don’t believe the “else” is necessary but it helps with readability. The corresponding javascript can be seen in the “server.js” file as well, and is similarly denoted by number 3.

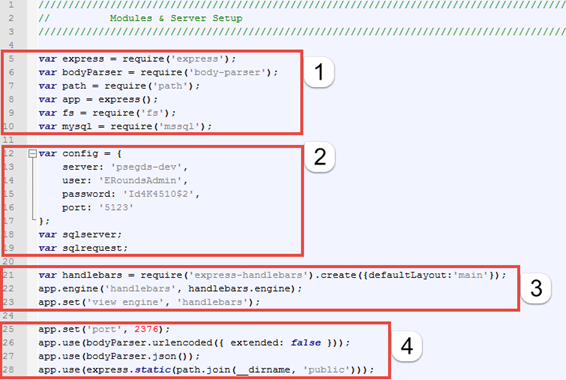
The block of code denoted by the number 4 does not have a conditional, but it is important to point out because of its unique syntax and functionality. The syntax “{{ > somePartial }}” is used strictly as a placeholder for *partials* while the “{{{ body }}}” syntax is used as a placeholder for the *views*. There is further applications for this syntax that is not relevant to the site so only the relevant application will be covered. As mentioned earlier, the first parameter of the *render()* function in the “server.js” file, also denoted by the number 4, determines the *view* being loaded.

The block of code denoted by the number 5 contains the logic for the previous/next page buttons. The first conditional states, “if isHome is true, then render the sub-content partial and the nav-buttons-home partial.” The additional parameter contained in the “nav-buttons-home” partial is a node.js specific way to pass a variable to a *partial*. The variables “nextPage” and “prevPage” are instantiated in “server.js” according to the appropriate previous and next page *view* names. In the example, denoted by the number 6, “prevPage” is “home” and “nextPage” is “sync”. These values are then passed to “main” where they are set equal to “prev-href” and “next-href” respectively. These aformentioned variable names which are found in “main” are also present in the “nav-buttons” which can be seen in the following image.



You can see how the variable are passed from “server.js” to “main”, and then to “nav-buttons” in order to become the links for the previous/next buttons for the page.

Because the last page only needs to be passed a single variable with the name of the previous page, there is a separate “else if” statement with a separate *partial*, named “nav-buttons-last”, which has a different variable. The same goes with the remaining pages which need a separate *partial*, but both a previous and next page variable.



# server.js

## Modules & Server Setup

The following will detail the parts of “server.js” which are specific to a node.js environment. When using node.js, server side javascript, there are a lot of similarities that can be found in most other programming language. For instance, where you might be familiar with importing a library to utilize various functions associated with it in a language like *C++*, in node.js, the equivalent is to *require* a *module*. In the image above, these modules can be found in the section of code denoted by the number 1.

|  |  |
| --- | --- |
| Module | Description |
| express | Necessary for supporting the template engine, and the routing in node.js |
| body-parser | Provides utilities to parse text, URL-encoded forms, and JSON |
| path | Provides utilities for working with file and directory paths |
| fs | Provides utilities for file synching, i.e. writing to or reading from files |
| mssql | Microsoft SQL Server client for node.js |
| express-handlebars | Allows for the file structure, as well as, utilizing views and partials in node.js |

The block of code denoted by the number 2 contains the variables necessary for the server. I would stronly recommend not messing with these unless you are simply making a change to the server being used, the user, password or port number.

The block of code denoted by the number 3 contains the *require* for “express-handlebars”, as well as, setting the *layout* template to the “main” file. It also sets the “view engine” as “express-handlebars,” and sets the requirement for all associated file to end in the “.handlebars” extension.

The block of code denoted by the number 4 contains various settings for the application. The port is set here. The “urlenceded ({ extended: false }) makes all requests to the server be returned in “key-value” pairs, while the next line sets the default encoding to be done in JSON. The final line in the code block sets the default path, for any further path references, to be set to the “public” directory. Without this, setting up paths would be much more difficult; the paths would begin at the location of the requesting file, and would not have a uniform location for the beginning of the path.

## Functions

There are 4 functions in the server side javascript file. These functions all work together to retrieve the webpage content from the database and write each individual page to its corresponding file. These functions are only ran on the initial rendering of the “home” page. The reason for this is to speed up the transition between pages. If content was retrieved from the database every time a new page was loaded, then latency could occur at ever render. By retrieving the data all at once, there is only ever a single connection to the database made, and this is done while the loading screen is being displayed.

|  |  |
| --- | --- |
| Function | Description |
| retrieveData ( ) | Opens a connection to the database, retrieves data for an individual page by calling *querypage()*, then sends the result as a parameter for *writefile()*. |
| querypage ( ) | Queries database for a single page at a time and returns data to the callback function in *retrieveData()*. |
| writefile ( ) | Sends page data to *constructHTML()*, then writes result to a corresponding file. |
| constructHTML ( ) | Encodes each line of data to html, then returns it to *writefile().* |

The first of the function calls originates in the *render()* function from the initial “home” page rendering.

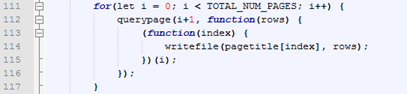


The *retrieveData()* function is used here as a “callback” function. A “callback” function is a function which is set as a parameter to another function, and is not executed until the parent function call has returned. This is done to make the functions run asynchronously, so that the loading screen is already loaded while the *retrieveData()* function is being executed.

When the *retrieveData()* function is called, a database connection is made using the credentials stored in the “config” variable. If the connection is successful, then pages are queried, one at a time, via the *querypage()* function call. The result from the *querypage()* function is then sent to the callback function upon its return. This particular callback function is a special form of a function which is necessary in this particular case to fix a *closure* issue. *Closures* are probably the most difficult concept in javascript to understand, but there is a ton of documentation online so I won’t go into too much detail.



You might imagine that this loop could be written as the above block of code, where there is a simple callback to the *writefile()* function when *querypage()* has returned. This seems logical and this syntax would execute as expected in most programming languages, but not javascript. What is happening is that, for each iteration of the loop, a “writefile ( pagetitle [ i ], rows ))” function call is being stored on a stack. Because there is no closure occurring, the functions are not being executed when called but are being stored. They are not executed until there is a closure in the loop, so with a loop iterating 10 times there would be 10 of these function calls stored on the stack, all with the same variable “i”. In this situation, what would the value of “i” be when the functions are finally being popped off the stack and executed? At the closure of the loop, i = 9, so the first argument for all 10 of the *writefile()* functions would be “pagetitle [ 9 ]”. This is where *closure* become necessary.



So, what is happening in this version, the version which has the *closure*, is that instead of storing *writefile()* functions using the loop iterator in one of the arguments, we pass the current value of the loop iterator to a new function where the current value is then stored as a different variable named “index”. A *closure* is created every time the function which contains the *writefile()* is executed. This forces the *writefile()* function to be executed every iteration opposed to being stored on a stack, which solves the issue.

The *querypage()* function simply takes a page number as its first argument, and the second argument is the callback function. The function creates the *SQL* query based on the page argument, makes the request, and returns the result to the callback function. In this case, the result gets used as the second argument to the *writefile()* function.

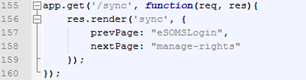
The *writefile()* function takes 2 arguments. The first argument is the desired filename, and the second argument is the, for lack of a better word, “raw” data retrieved from the database. This function first creates a variable to hold the *path* for the file being created. Each line of “raw” data is then sent to the *constructHTML()* function, one at a time, to be converted to *HTML*. After all the lines of “raw” data from the page converted, then they are written to the designated *path* with a newline character between each line to provide readability for anyone viewing the source code.

The *constructHTML()* function takes a *code* that is stored with every line of content in the database, as well as, the content itself. The coding scheme allows the switch statement to determine which *HTML tags* to wrap each line of webpage content with. The lines are simply *HTML* encoded and then returned to the *writefile ()* function to be stored in the “HTMLEncoded [ ]” array.

## Get Requests

There is not much to say about the *render()* functions which hasn’t already been covered in the **Layouts** section of this document.

When a link has been “clicked” which takes a user to another page, the name of the requested page is appended to the *URL* and is processed by “server.js” as a *get* request. When the *get* request with the first argument matching the appendage to the *URL* is found, then the corresponding block of code is executed. For example, if the *URL* is appended with “/sync” the following block will be executed.



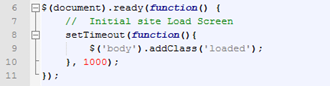
When the *get* request has been closed, then control will again transfer back to the *HTML*, but before that the *render()* function must return; this is because *render()* is a callback function. As mentioned before, the first argument of the *render()* function is the page being loaded, while the remaining arguments are additional parameters that are utilized by the *layout*, or “main.handlebars” in this webpage.

Note: The *views* titled “404” and “500” pertain to webpage errors. The “404” *view* is only ever shown when there is a server error, i.e. server timed out, can’t find a *view*, etc. The “505” *view* is only ever shown when there is a logic, programming, error.

# client.js

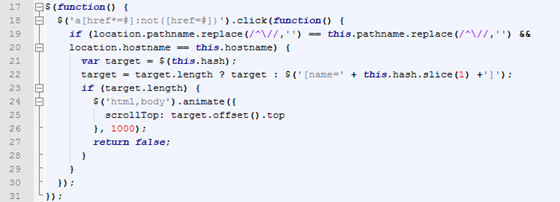
Client side javascript is loaded after the page has rendered, and enables the enhancement and manipulation of the webpages and the client browser. This “client.js” script governs the user interactions and does not directly communicate with the server.

## Loading Screen



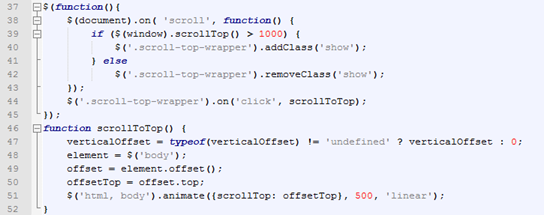
This section of code is executed on the initial webpage rendering. It creates a timeout for 1000 milliseconds (1 second) while “loaded” is added to the class name of *body*. This creates the loading screen at the initial rendering of the webpage and for the duration of 1 second.

## Anchor Transitions



Anchor transitions are animated scrolling transition which occur when a link to somewhere else on the same page is “clicked”. All that is vital to understand here is that the transition takes 1000 milliseconds. For the transition animation to function properly, nothing should be changed aside from the duration of the animation.

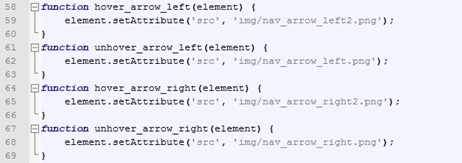
## Scroll-to-Top Button



This block of code is executed when the user’s screen is more than 1000 pixels below the top of the page. The button itself become viewable after 1000 pixels and the “scrollToTop” listener is created for the button. When the button button is “clicked”, then the *scrollToTop()* function in this segment of code is executed. The *scrollToTop()* function sets the “verticalOffset” variable equal to zero and the “offsetTop” variable equal to the top of the webpage’s *body.* The *animate()* function then creates a *scrollTop* action based on the predefined “offset”, which was set to the top of the page, over a 500 millisecond period.

For this animation to function properly, the only variables which should be changed are number of pixels below the top of the page which must be surpassed to display the button, or the period of time for which this animation occurs.

## Next/Previous Page Buttons



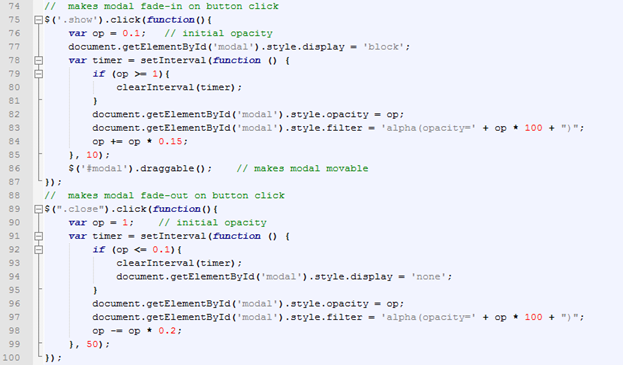
These 4 functions are fairly self-explanatory based on their names. One set of function governs the actions of the “arrow” buttons when the mouse cursor is hovering over the button, and the other governs the actions when the mouse cursor is not hovering over them.

|  |  |
| --- | --- |
| *hover\_arrow\_left()* | *unhover\_arrow\_left()* |
| C:\node.js\public\img\nav_arrow_left2.png | C:\node.js\public\img\nav_arrow_left.png |

When the mouse cursor is hovering over one of the next/previous page buttons, then the image actually changes to the “arrow” with the lighter shade of gray pictured above. When the mouse cursor is not hovering over the same button, then the image returns to the original “arrow” image which has the slightly darker shade of gray.

## Acronym Modal

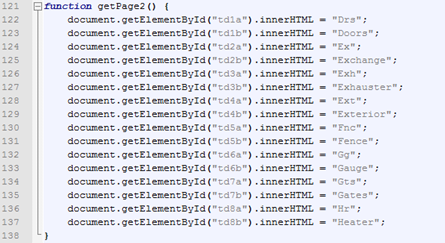
A modal is a pop-up window which takes precedence over its parent window.



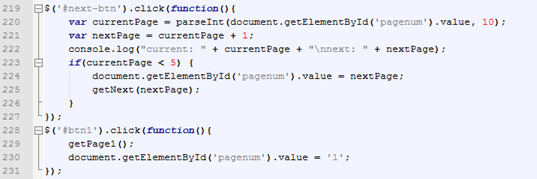
These blocks of code are action listeners. The top listener makes the modal fade-in on button “click” while the second listener makes the modal fade-out on a button “click”.



This line of code from the first listener allow the user to drag the modal to anywhere on the screen.



The five function which resemble the block of code in the image above are used to replace the table cells, which have the matching ids, with the string on the right-hand side of the expression.



These blocks of code are button listeners. The top block is executed when the forward button on the modal is “clicked”. The following line of code is *HTML* which is taken from the “acronym-table” *partial*, where the mark-up for the modal is located.



This is a hidden “input” which acts as an *HTML* variable. The hidden “value” equals 1 which corresponds to the default set of data being represented by the table. This value is changed in the modal’s button listener blocks according to the current page being displayed. This is how the previous/next button listeners can determine which set of data is currently displayed and which set of data needs to be retrieved.

# style.css

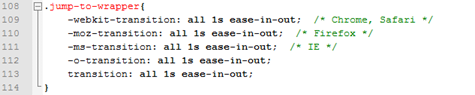
The *CSS* file for the website is the least commented on, and potentially the most difficult file to traverse. To edit the parameters I would strongly suggest knowing the corresponding id number or class name and conducting a search of the document. Very few blocks of code will be discussed here.

## Global

All code in the global section pertains to style attributes which are present on every page. These things include much of the site structure including attributes for the outer-most *div* containing the content from the different *views*, *CSS* mark-up for the site’s animation including *keyframes*, and globally represented tools like the “jump-to top” buttons and the acronym modal.



This block of code fixes the screen resolution issue. This makes the website zoom in or out as appropriate depending on the width of the monitor that the site is being viewed on.



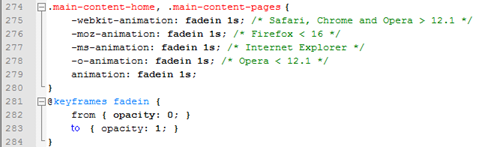
This block of code is responsible for setting the fade-in animation duration for the “jump-to top” button. The animation occurs when the button is first made visible, when the window shown is over 1000px below the top of the page.



All blocks of code beginning with “.scroll-top-wrapper” govern the many attributes of the “jump-to top” button. Some of the code sets the duration of time that button color transitions to a lighter shade of gray when the mouse cursor is hovering over the button, some adjusts the buttons location and rounded corners, and some of the code pertains to its visibility.



Blocks of code beginning with “#loader” pertain to the animated loading screen that is present on the initial load of the “home” page. If you intend on keeping the loading screen operational I would not alter any of the code related to it; anything with %loader% in the id or class.



These blocks of code are responsible for the fade-in animation which occurs when a new page is rendered. The content is set to fade-in, from an opacity of 0 to an opacity of 1, over a period of 1 second.





All blocks of code which have either “.modal” or “.acronym-table” as a class name pertain to the modal, or pop-up window, which contains a table of acronyms.

## Header Section

The header section contains all the *CSS* for the top, middle, and bottom headers.



This block of code is what is responsible for displaying the image located in the header of the “home” page. If the image needs to be changed, this is where the image path must be updated from.

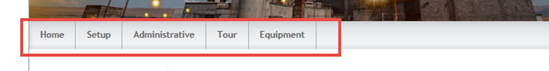


The “nav1” class pertains to the navigation menu in the “header-top” section of the webpage as shown in the following image.

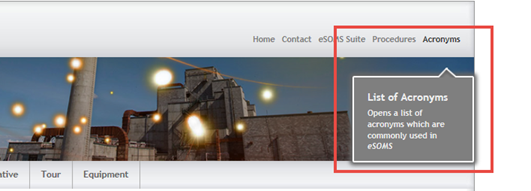




The “nav2” class pertains to the navigation menu in the “header-bottom” section of the webpage as shown in the following image.



The “tooltip” class pertains to the tool tips located in the “header-top” section of the webpage, and which show when the mouse cursor hovers over one of the menu items. The “callout” class has to do with triangle piece of the tooltip which points at the appropriate menu item. This is shown in the following image.



## Main Section

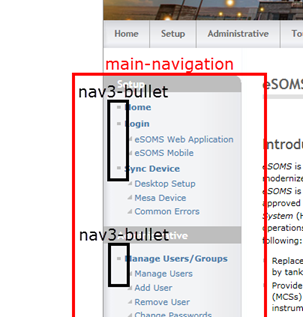
The main section is responsible for the *CSS* that is between the header and the footer; this includes the *HTML* mark-up for the “main-navigation”, “main-content” for each page, and the “sub-content” which is present on the “home” page.



The “main-navigation” class refers to the *div* containing the navigation pane located on the left-most part of the main content. Inside this navigation pane are several unordered lists containing all of the websites’ pages, as well as, the jump-to links for their respective page.



The “nav3-bullet” class refers to the custom bullets for the list sub-item, the jump-to links. The following image shows the “main-navigation”, as well as, the “nav3-bullet”s.

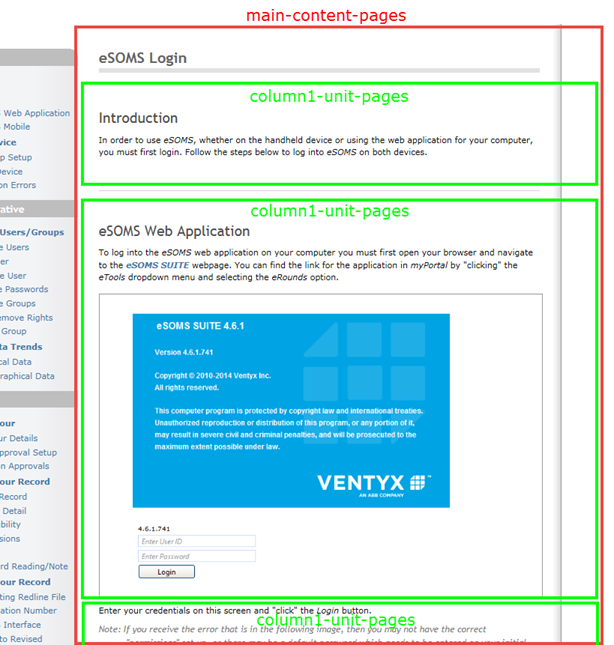




Anything block of code containing “column1-unit” in the class name is referring to the inner-most *divs* which are used to structure the page’s “main-content”. Every *view* contains multiple sub-sections of content which are organized into these *divs*. The area on a page which contains the “main-content” is different for the “home” page than it is for any of the other pages, thus the two different classes. This is because the “home” page contains the “sub-content” section which occupies much of the available space. The “column1-unit-home” governs the attributes of the *div* for the “home” page, and “column1-unit-pages” governs the attributes of the *div* for the remaining pages.



The same is true for the “main-content” classes; there are alternate attributes for the “home” page as opposed to the rest of the pages due to the difference in space to fill. These classes represent the outer most *div* for the main content of each *view*. There is only every one of these *divs* present on a particular page. The following image will illustrate the locations of these *divs* on the website.

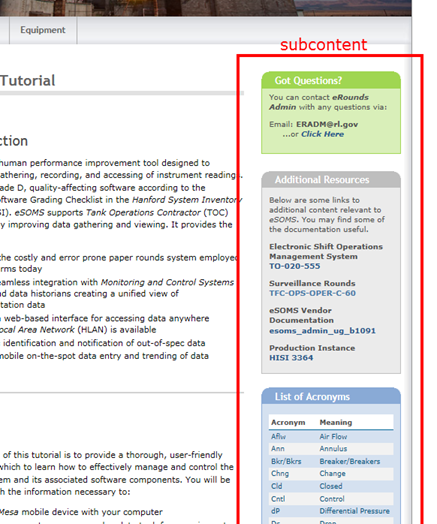




The blocks of code which contain the “hangingindent” class pertain to the “notes” in the “main-content” of the webpage. The following image contains a note from the webpage which utilizes the “hangingindent” class.



All *CSS* blocks in the “Main Subcontent” subsection of the “main section” refer to “sub-content.handlebars” *partial*. This is the sub-content which is present only on the “home” page of the website. The following image shows the sub-content section of the “home” page.





The table with the id of “ac-table” represents the table of acronyms which is located in the lower section of the subcontent.

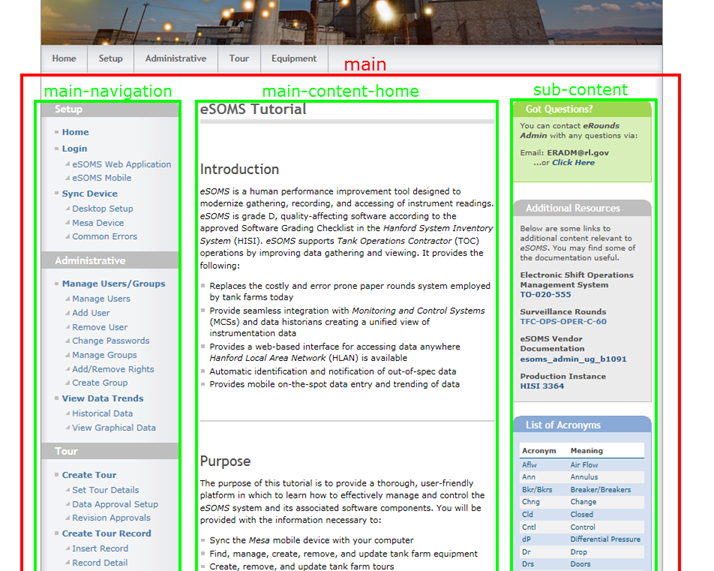
## Remaining CSS



The footer section of the *CSS* is represented by the “footer” class. Both blocks of code set the various text output attributes.

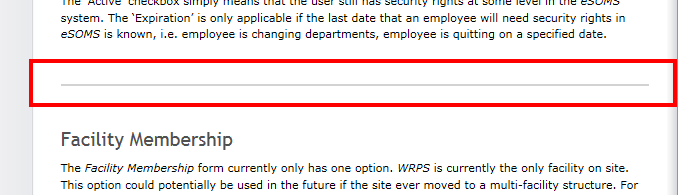


The “main” class refers to the *div* which encompasses everything between the header and the footer. This *div* contains the “main-navigation”, the “main-content” and the “sub-content”, and there is only ever one “main”. The *CSS* containing the class “main” is inherited by all child *divs* and elements of the webpage. The following image shows the “main” *div* in relation to the rest of the webpage.



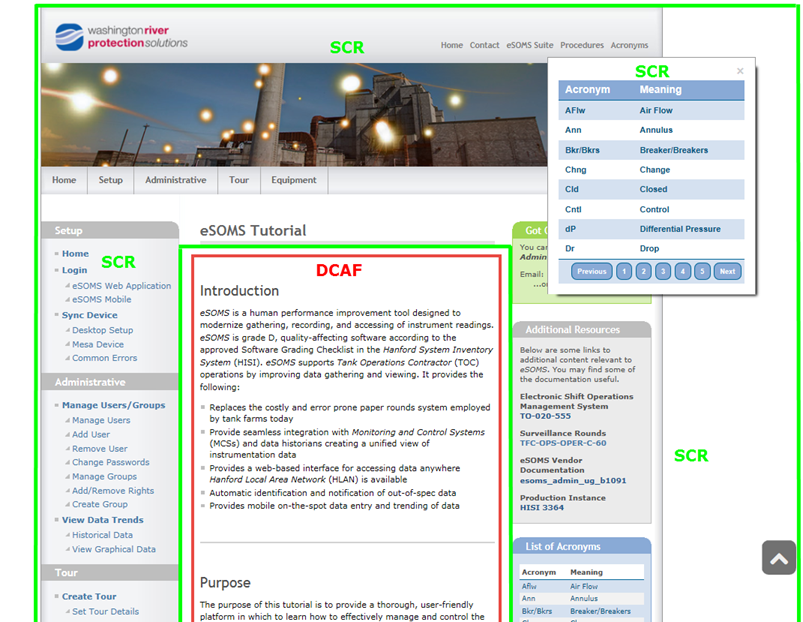


The “clear-contentunit” class refers to a *div* which appears as a line on the website. These lines appear prior to headings and are used to separate sections of the “main-content”. The area on a page which can contain a “clear-contentunit” is different for the “home” page than it is for any of the other pages, thus the two different classes are necessary. The reason for this is that the “home” page contains the “sub-content” section which occupies much of the available space. The following image will show what this line appears like in the website.



# Making Changes to Website

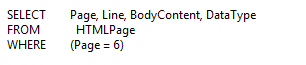
Some of the data on the website is stored as “raw” data on the database, and can be fairly easily changed with a *DCAF* and by using basic *SQL* queries. Other parts of the website, the *CSS, Javascript*, and some of the *HTML*, must be modified using an *SCR*. The way the site is set up is so that changes to the main content will require a simple *DCAF* while changes to the site’s framework will require an *SCR*. The following image will clarify which portions of a page can be modified with which request.



## *DCAF* Changes

If a change needs to be made to the “main-content” of the webpage which doesn’t require a new line of *HTML* it is fairly simple. What is meant by this is that the change only modifies the content of an existing row stored in the database, i.e. fixing a typo, rewriting or adding a sentence, changing an image. More complicated changes would be adding an entire new *HTML* tag to the middle of a page which would require a new row in the database and the sequence of all the rows for the page to be incremented. An example of this would be adding a new heading ( <h1> ), a new paragraph ( <p> ), or a new image ( <img> ).

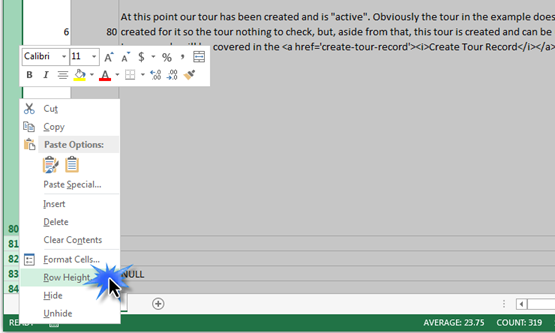
To simply modify an existing row you can search the database for the ‘Page’ which contains the row.



Once you determine the row, or ‘Line’ number, then you can edit it manually in *Microsoft SQL Server Management Studio* or by using an *SQL* update query.

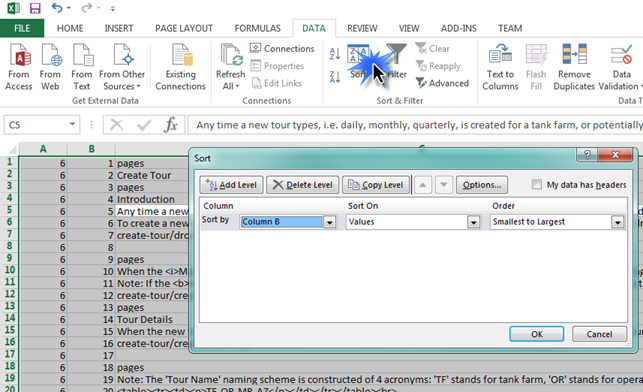
To add a new element to the *HTML*, i.e. add a new row to the database, I find it is easiest to transfer the entire page to an *Excel* spreadsheet. This will allow you later to increment the “Line” numbers automatically so that the elements in the webpage are displayed in the appropriate order.

When you have an *Excel* spreadsheet populated with the appropriate page from the database there are a couple tips to make traversing the spreadsheet easier. The first thing you can do is change the row height so the document is more manageable. This can be done by selecting all cells with “Ctrl – A”, then “right-clicking” on a row and selecting “Row Height”. I generally change the value to 14 (14 pixels).



Once the row heights are fixed, then you will need to ensure the ‘Line’ numbers are in the correct order. Remember, it is not necessary for the rows to be in correct order in a database so they are not necessarily in order when they are transferred to an *Excel* spreadsheet.

To sort the rows according to their ‘Line’ number, first “click” on the *DATA* tab and select the *Sort* option from the toolbar. This will bring up the *Sort* tool which is shown in the following image.



Next just find the location in spreadsheet where you want to add a row, insert the row, fill in the fields, and make sure to drag the top two line numbers to the bottom of the table so the rows will be renumbered.

That is it, run a delete query in the sql to delete the entire page, then copy and paste the new data back in.

Note: How to format the ‘DataType’ and ‘BodyContent’ columns will be discussed the the *Formatting Convention* section of this document.

To add an image, not only must you insert the appropriate row into the database, but you must also store the image in the *img* directory which is located with the source files.

## *SCR* Changes

Much of the webpage is supposed to be semi-permanent, and it’s not as easy to modify. This includes the structural elements of the website, the style (*CSS*), and the *Javascript*. These changes will require an *SCR*, as well as, directly modifying the source code. The first five sections of this document should assist with making any of these changes which are required.

There are a couple things which can be changed that are not in the “main-content”. These are the banner in the “home” page header and the company logo in the upper-left corner of the website. These can be changed without accessing the source code if their replacements have the same filename.

The header is named “header-img.png” and has the dimensions “900 x 169”.

The company logo is named “wrpslogo.png” and has the dimensions “195 x 46”

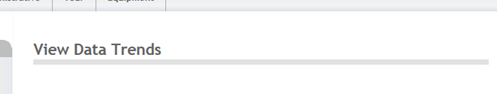
Both of these images are located in the “~/public/img” directory.

## Formatting Convention

When adding content to the website, there is a special format that must be adhered to in order for the database rows to be correctly parsed and *HTML* encoded. The ‘DataType’ column of the database table holds a key, or a code per say, which tells the server side *Javascript* how to interpret the ‘BodyContent’.

**DataType Codes**

pagetitle – This key is used once per page and creates the title at the top of each page,as well as the thick line beneath it. When this tag is used it is necessary to input either the word “home” or the word “pages” into the ‘BodyContent’ column of the database depending on if the title will be located in the “home” page or if it will be located on another page.

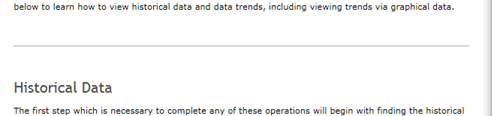


divmain – This key creates the main *div* which contains all the “main-content” elements for the page. There should only be one ‘divmain’ per page. When this tag is used it is necessary to input either the word “home” or the word “pages” into the ‘BodyContent’ column depending on if the *div* will be located in the “home” page or if it will be located on another page.

divopen – This key creates the opening tag for a “column1-unit” *div*. There are several of these *divs* on each page; a new tag is used after each image. When this tag is used it is necessary to input either the word “home” or the word “pages” into the ‘BodyContent’ column depending on if the *div* will be located in the “home” page if it will be located on another page. These *divs* are used primarily to maintain a specific structure for the site and spacing between elements.

divclose – This key creates the closing tag or a *div*. It is equivalent to a “</div>” tag. These are used after every image and directly before a new “divopen”, unless there is a heading directly after the image.

break – This key creates a break and is used prior to a heading tag. These are used to create a uniform spacing between elements and to create a line to separate sections of the page.

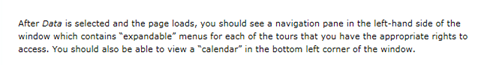


h1 – This key creates a section heading. It is equivalent to a “<h1>” tag. It is important to also note that a heading “id” can also be included here which is only necessary if you intend on making the heading a jump-to location for a link. An example of this is in the following image.





p – This key creates a section of text on the page. It is equivalent to a “<p>” tag.



note – This key creates a section of text which has been specially formatted with font color, italics, and with a hanging indent. It is used to signify important information.



img – This key denotes an image. The input for ‘BodyContent’ needs to be the path to the image starting from the “img” directory. The following will show an example of what an “img” row should resemble.

C:\Users\H8835791\AppData\Local\Temp\SNAGHTML1dd2f6f8.PNG

imgmesa – This key denotes a screenshot image from the *Mesa* device. The formatting is different because of the maximum width is less for the *Mesa* screenshots which is why there is a separate key. The ‘BodyContent’ should follow the same criteria as with the “img” key.

li – This key creates a bulleted list item with square bullets. The opening and closing “<ul>” tag should be created using the “AsIs” key. An example of the “<ul>” tags is in the second image.





AsIs – This key is reserved for any *HTML* not specified by one of the previous keys. I commonly use this for opening and closing tags for lists, and for tables. What goes in the ‘BodyContent’ of these needs to be perfectly encoded *HTML* because the content will go straight into the webpage. An example of a table is in the following image.

