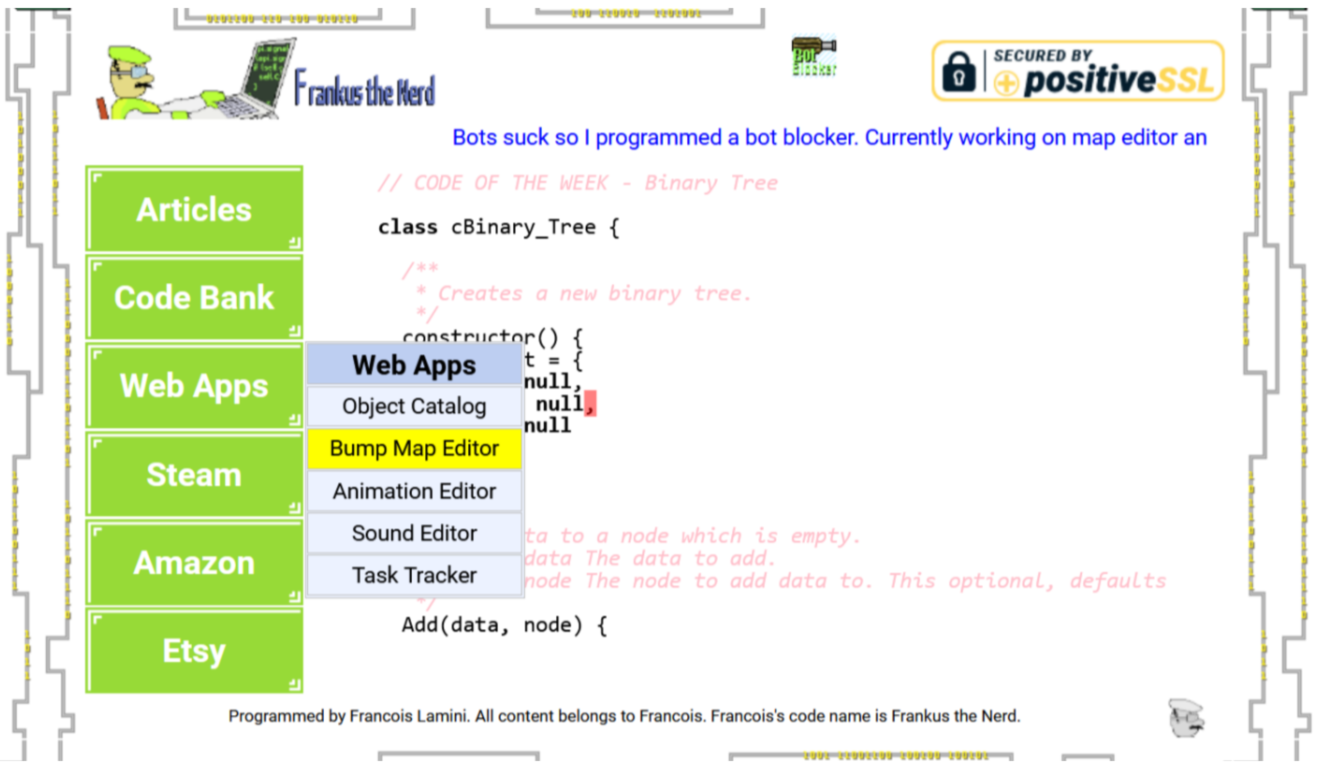


Bots suck so I programmed a bot blocker. Currently working on map editor an

```
// CODE OF THE WEEK - Binary Tree

class cBinary_Tree {

  /**
   * Creates a new binary tree.
   */
  constructor() {
                  t = {
                  null,
                   null,
                  null

                  ta to a node which is empty.
                  data The data to add.
                  node The node to add data to. This optional, defaults
   */
  Add(data, node) {
```

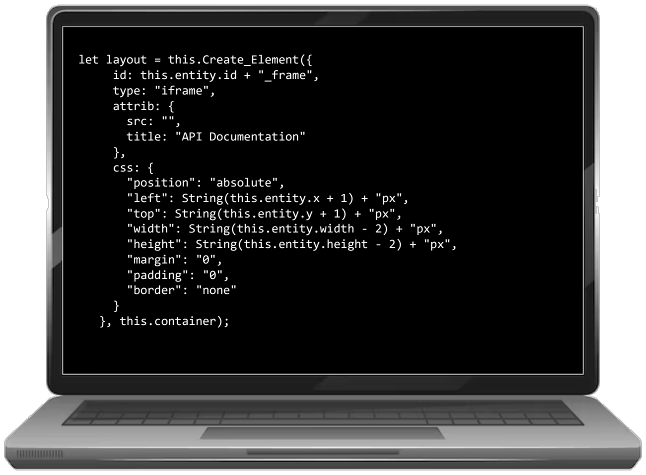
Programmed by Francois Lamini. All content belongs to Francois. Francois's code name is Frankus the Nerd.

Web Development in Frankus Framework

Overview

So you want to develop a web site using a framework. Well, there's Angular, React, and Vue. I personally used Angular, React, and am learning Vue. They're pretty cool. However, let's talk about something that doesn't use HTML. Whoa? Now things are probably getting weird for you. How can you develop any kind of web site without using HTML. Isn't HTML fundamental to web development? Yeah, sure.

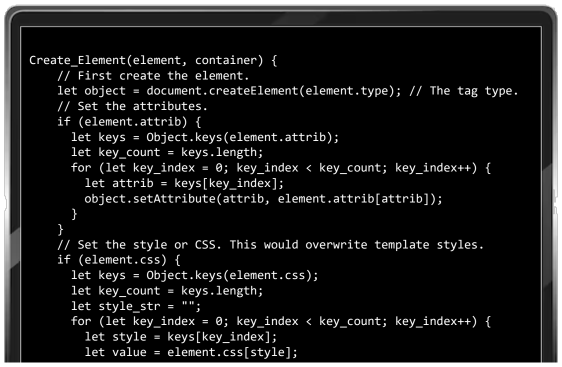
As a programmer you could do this:



```
let layout = this.Create_Element({
     id: this.entity.id + "_frame",
     type: "iframe",
     attrib: {
       src: "",
       title: "API Documentation"
     },
     css: {
       "position": "absolute",
       "left": String(this.entity.x + 1) + "px",
       "top": String(this.entity.y + 1) + "px",
       "width": String(this.entity.width - 2) + "px",
       "height": String(this.entity.height - 2) + "px",
       "margin": "0",
       "padding": "0",
       "border": "none"
     }
   }, this.container);
```

So this looks kind of weird but it's really just a JSON representation of the DOM. As you can see it has a ID, type, an attribute section, and a CSS section as well. Nothing special here. This is all mapped to the DOM by calling createElement() from the DOM API.

Here's the function in case you want to see:

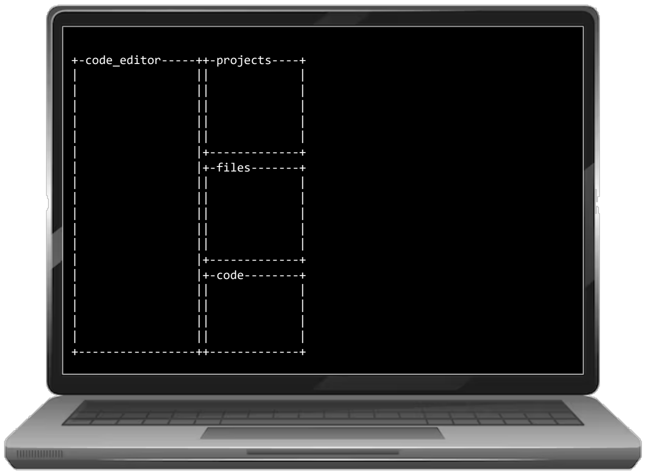


```
Create_Element(element, container) {
    // First create the element.
    let object = document.createElement(element.type); // The tag type.
    // Set the attributes.
    if (element.attrib) {
      let keys = Object.keys(element.attrib);
      let key_count = keys.length;
      for (let key_index = 0; key_index < key_count; key_index++) {
        let attrib = keys[key_index];
        object.setAttribute(attrib, element.attrib[attrib]);
      }
    }
    // Set the style or CSS. This would overwrite template styles.
    if (element.css) {
      let keys = Object.keys(element.css);
      let key_count = keys.length;
      let style_str = "";
      for (let key_index = 0; key_index < key_count; key_index++) {
        let style = keys[key_index];
        let value = element.css[style];
```

```
        if (style == "cursor") {
```

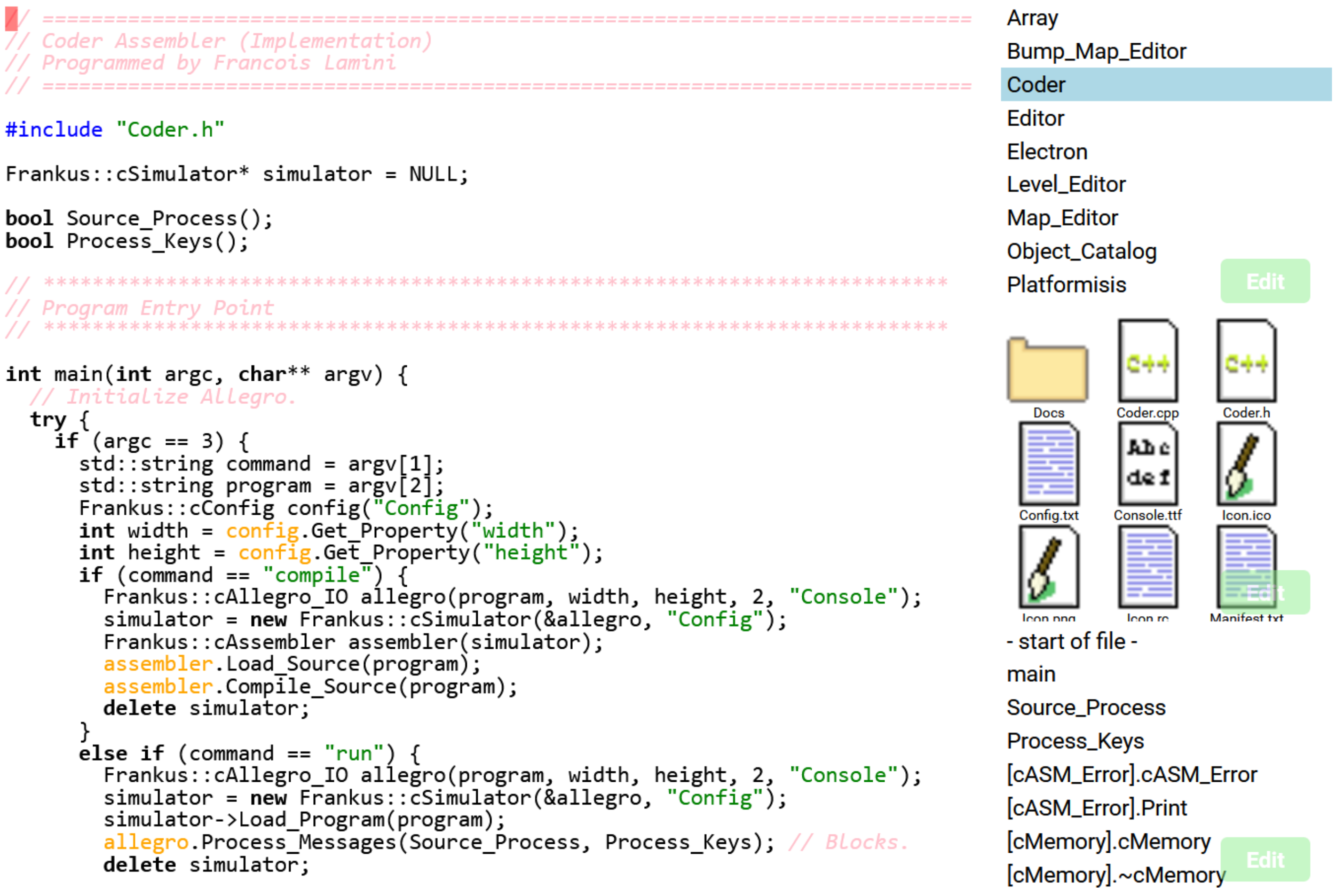
See, nothing special. Just some good ol' DOM API calls. This function is actually part of the component object tree from which all Frankus components are derived from. It is in the base component class.

So now, let's switch gears and look at something even weirder...



```
+-code_editor-----++-projects----+
|                 ||             |
|                 ||             |
|                 ||             |
|                 ||             |
|                 ||             |
|                 |+-------------+
|                 |+-files-------+
|                 ||             |
|                 ||             |
|                 ||             |
|                 ||             |
|                 ||             |
|                 |+-------------+
|                 |+-code--------+
|                 ||             |
|                 ||             |
|                 ||             |
|                 ||             |
+-----------------++-------------+
```

Whoa? What in the name of programming is this? It looks like some kind of ASCII art which it is. As you might have guessed, the components are mapped to the text boxes and the properties are set on the bottom. Here are the results:

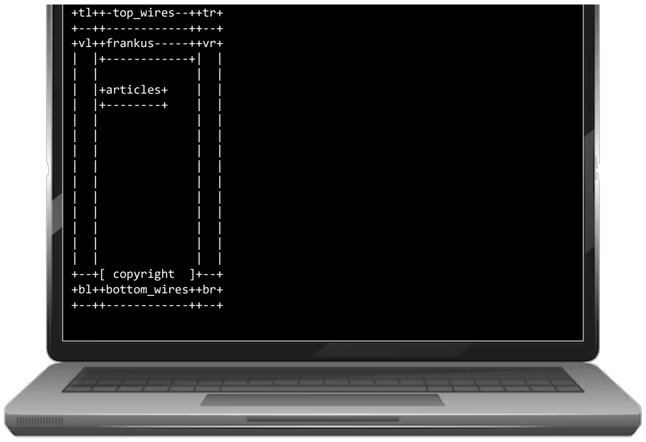


```
// ==============================================================================
// Coder Assembler (Implementation)
// Programmed by Francois Lamini
// ==============================================================================

#include "Coder.h"

Frankus::cSimulator* simulator = NULL;

bool Source_Process();
bool Process_Keys();

// ****************************************************************************
// Program Entry Point
// ****************************************************************************

int main(int argc, char** argv) {
  // Initialize Allegro.
  try {
    if (argc == 3) {
      std::string command = argv[1];
      std::string program = argv[2];
      Frankus::cConfig config("Config");
      int width = config.Get_Property("width");
      int height = config.Get_Property("height");
      if (command == "compile") {
        Frankus::cAllegro_IO allegro(program, width, height, 2, "Console");
        simulator = new Frankus::cSimulator(&allegro, "Config");
        Frankus::cAssembler assembler(simulator);
        assembler.Load_Source(program);
        assembler.Compile_Source(program);
        delete simulator;
      }
      else if (command == "run") {
        Frankus::cAllegro_IO allegro(program, width, height, 2, "Console");
        simulator = new Frankus::cSimulator(&allegro, "Config");
        simulator->Load_Program(program);
        allegro.Process_Messages(Source_Process, Process_Keys); // Blocks.
        delete simulator;
```

If you look carefully you can see the code editor, project list, file list for the selected project, and code browser with mapped functions.

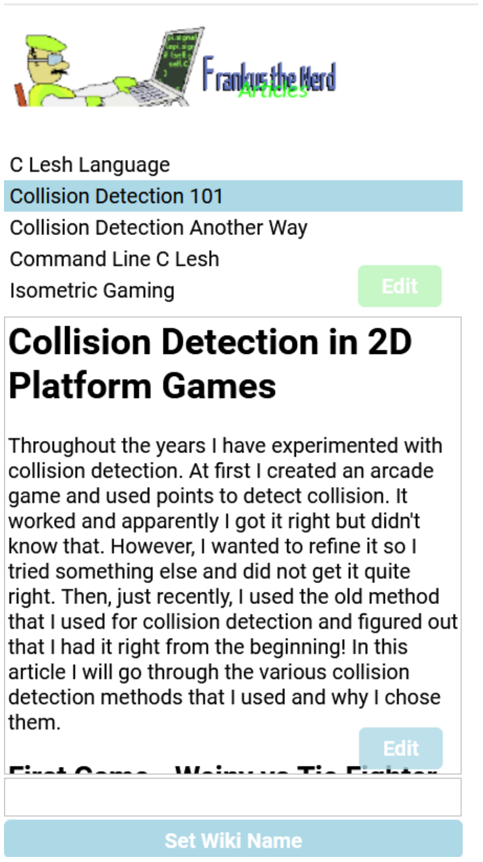
So now the other question you might have is that if this is responsive. The answer is that the entire page is rendered on a fixed size container (960x640) and that container is stretched to the size of the window. Why do that, the answer is that text is stretched too eliminating the need for media queries to size text. Also people who have hard time seeing small text can see if more easily. (Hopefully!)

So what about mobile devices like phones. Yeah, they're small so for those you would create mobile version of the page. Like this:



```
+tl++-top_wires--++tr+
+--++------------++--+
+vl++frankus-----++vr+
|   |+------------+|   |
|   |              |   |
|   |+articles+    |   |
|   |+--------+    |   |
|   |              |   |
|   |              |   |
|   |              |   |
|   |              |   |
|   |              |   |
|   |              |   |
|   |              |   |
|   |              |   |
|   |              |   |
+--+[ copyright  ]+--+
+bl++bottom_wires++br+
+--++------------++--+
```

So this is the mobile version of the home page of the Frankus site. And the rendering...



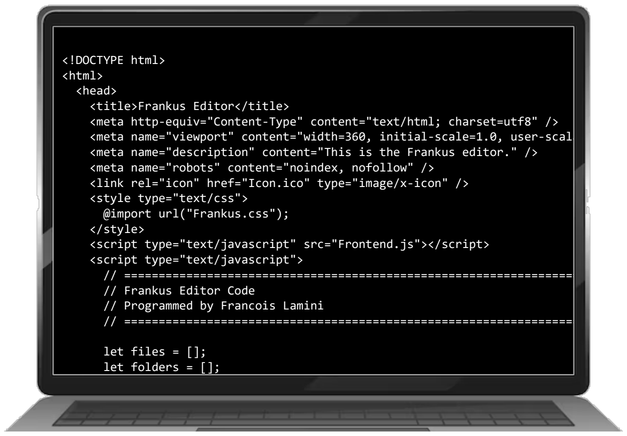
C Lesh Language
Collision Detection 101
Collision Detection Another Way
Command Line C Lesh
Isometric Gaming

Collision Detection in 2D Platform Games

Throughout the years I have experimented with collision detection. At first I created an arcade game and used points to detect collision. It worked and apparently I got it right but didn't know that. However, I wanted to refine it so I tried something else and did not get it quite right. Then, just recently, I used the old method that I used for collision detection and figured out that I had it right from the beginning! In this article I will go through the various collision detection methods that I used and why I chose them.

So you can see that it looks simple but yet you can read an article on it. Unlike HTML, if there is a lot of text to be read you put it in a Wiki component. In HTML you just write the text there or use a markdown file. The whole page scrolls. Not in Frankus, here you neatly place the text in a component and edit it there.

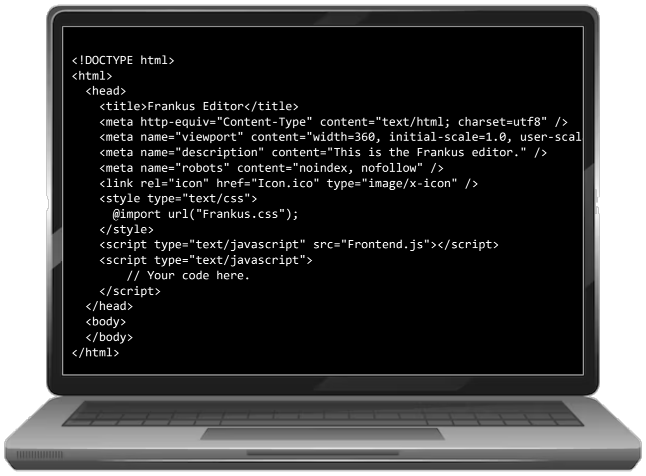
So another question you might have is if you have to write any JavaScript code. Well, you do and it is mostly responding to event handlers. Here's an example of code written for the Code Editor page.



```
<!DOCTYPE html>
<html>
  <head>
    <title>Frankus Editor</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf8" />
    <meta name="viewport" content="width=360, initial-scale=1.0, user-scal
    <meta name="description" content="This is the Frankus editor." />
    <meta name="robots" content="noindex, nofollow" />
    <link rel="icon" href="Icon.ico" type="image/x-icon" />
    <style type="text/css">
      @import url("Frankus.css");
    </style>
    <script type="text/javascript" src="Frontend.js"></script>
    <script type="text/javascript">
      // =================================================================
      // Frankus Editor Code
      // Programmed by Francois Lamini
      // =================================================================

      let files = [];
      let folders = [];
```

Sorry, there's a lot in here. So, in Frankus every site is placed in it's own HTML file. Wait a second, HTML? But I said there wasn't HTML. Well, the HTML page is the starting point for the framework just like any other framework. Let's break up this monstrosity!

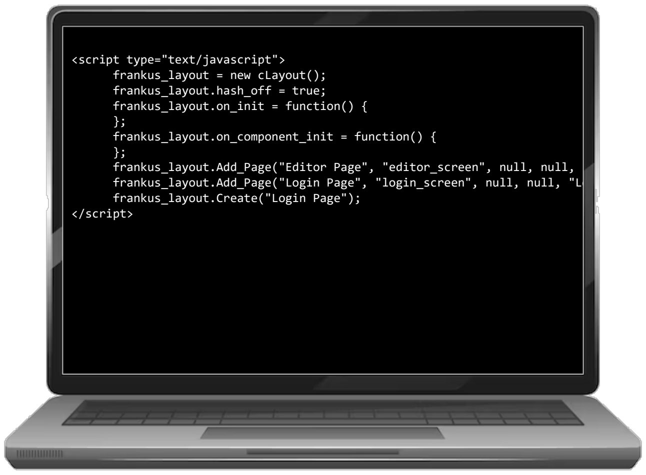
Basic HTML



```
<!DOCTYPE html>
<html>
  <head>
    <title>Frankus Editor</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf8" />
    <meta name="viewport" content="width=360, initial-scale=1.0, user-scal
    <meta name="description" content="This is the Frankus editor." />
    <meta name="robots" content="noindex, nofollow" />
    <link rel="icon" href="Icon.ico" type="image/x-icon" />
    <style type="text/css">
      @import url("Frankus.css");
    </style>
    <script type="text/javascript" src="Frontend.js"></script>
    <script type="text/javascript">
        // Your code here.
    </script>
  </head>
  <body>
  </body>
</html>
```

So we're building a basic HTML document. Frankus is a multi-page single document file. You need to include "Frontend.js". Frankus also has a "Backend.js" which contains the server and other modules to process things like authentication and a basic API.

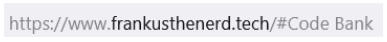
Next, Frankus needs to be initialized. Without that no site!



```
<script type="text/javascript">
      frankus_layout = new cLayout();
      frankus_layout.hash_off = true;
      frankus_layout.on_init = function() {
      };
      frankus_layout.on_component_init = function() {
      };
      frankus_layout.Add_Page("Editor Page", "editor_screen", null, null,
      frankus_layout.Add_Page("Login Page", "login_screen", null, null, "L
      frankus_layout.Create("Login Page");
</script>
```

So this might seem a bit overwhelming but it's not too complex. In the first like you initialize Frankus in a global variable called "frankus_layout". This is a predefined global variable which is used internally in the Frankus framework.

In the next line we turn off the hash. What is a hash? Well, this:

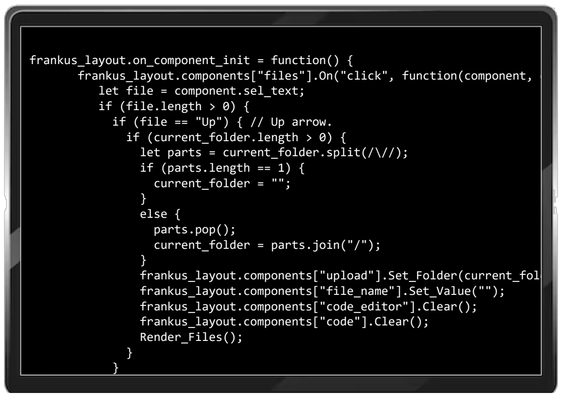


https://www.frankusthenerd.tech/#Code Bank

After the # sign there is the name of the page. You can go to a page with the name placed after the # sign. It's part of the routing.

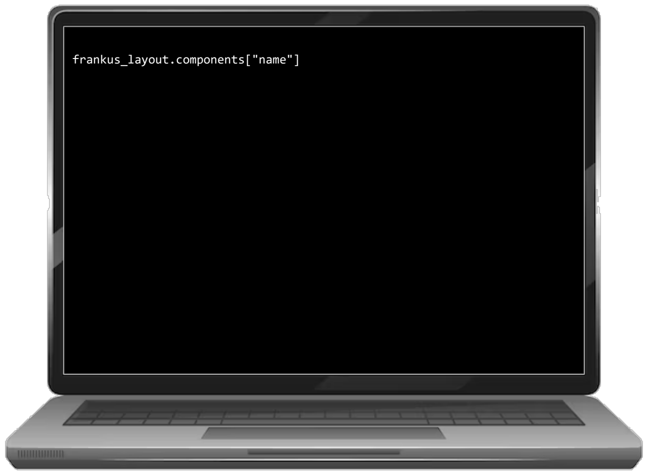
So, let's continue. The next thing we need to look at is the callback function for on_init(). What's this? Well, you'd place code here that you want to execute prior to creation of components. For example if you want to query project names or something before feeding them to a list component.

Now in the line after on_init() we see on_component_init(). This is where we write our component callback code. This is code for component events. Here's some of that code:



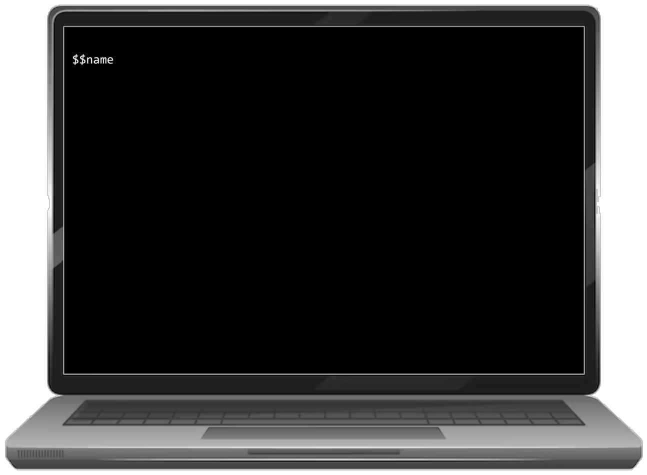
```
frankus_layout.on_component_init = function() {
       frankus_layout.components["files"].On("click", function(component,
          let file = component.sel_text;
          if (file.length > 0) {
            if (file == "Up") { // Up arrow.
              if (current_folder.length > 0) {
                let parts = current_folder.split(/\//);
                if (parts.length == 1) {
                  current_folder = "";
                }
                else {
                  parts.pop();
                  current_folder = parts.join("/");
                }
                frankus_layout.components["upload"].Set_Folder(current_fol
                frankus_layout.components["file_name"].Set_Value("");
                frankus_layout.components["code_editor"].Clear();
                frankus_layout.components["code"].Clear();
                Render_Files();
              }
            }
```

So, again, a lot of code but as you can see you can access a component using:



```
frankus_layout.components["name"]
```

But that's too much to type, what gives? Actually, in a previous version of Frankus called Codeloader you used to be able to write:



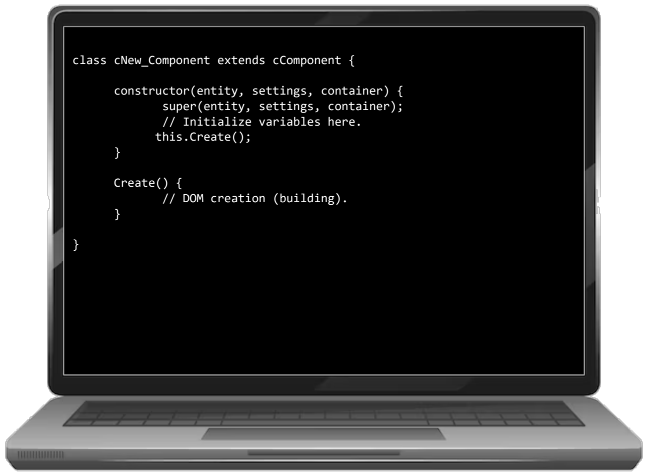
```
$$name
```

So the component name used to be placed into the global namespace and accessed with two $ signs. Well, actually, it was mapped onto there. So why did I go the other way. I wanted to not pollute the global namespace.

So, now let's talk about something else - implementing a component.

Basic Component

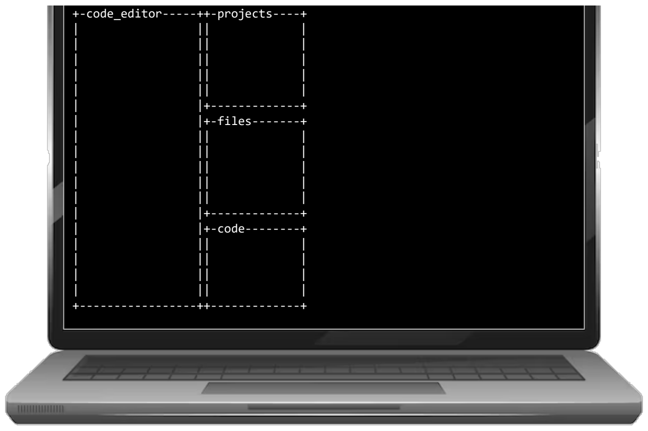
To create a component we subclass "cComponent". What does the "c" mean? It means class.



```
class cNew_Component extends cComponent {

      constructor(entity, settings, container) {
             super(entity, settings, container);
             // Initialize variables here.
            this.Create();
      }

      Create() {
             // DOM creation (building).
      }

}
```

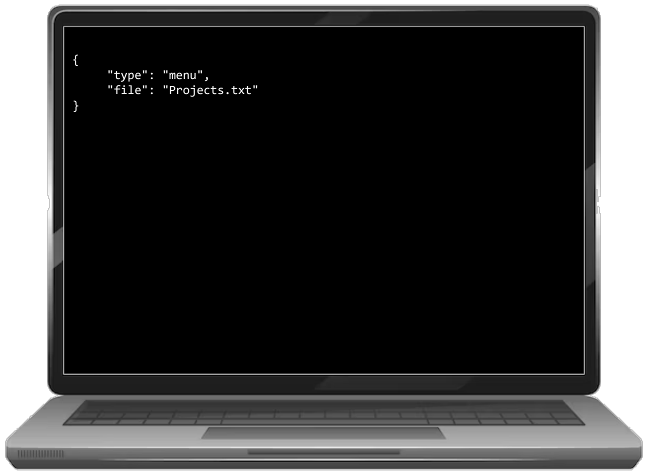
So, first, you create component constructor. Call the super class constructor with the entity, settings, and container. Wait... what are these?

So remember the properties? We'll take an example from the code editor.



```
+-code_editor-----++-projects----+
|                 ||             |
|                 ||             |
|                 ||             |
|                 ||             |
|                 ||             |
|                 |+-------------+
|                 |+-files-------+
|                 ||             |
|                 ||             |
|                 ||             |
|                 ||             |
|                 ||             |
|                 |+-------------+
|                 |+-code--------+
|                 ||             |
|                 ||             |
|                 ||             |
|                 ||             |
+-----------------++-------------+
```

Let's focus on the projects component. We see the "change-type" and "file" properties. These name-value pairs are mapped onto the settings parameter like this:



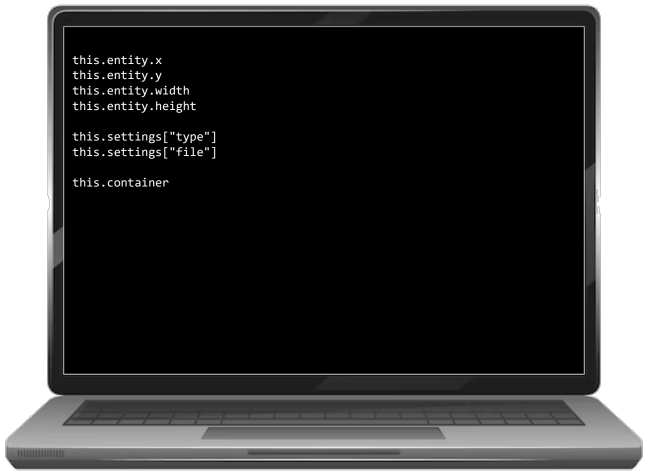
```
{
     "type": "menu",
     "file": "Projects.txt"
}
```

But I thought there was the "change-type" property. Yeah, it's changed by the framework to the property name "type" because "change-type" actually, tells the framework to change the type from a box to something else. The default type for any ASCII box is "box".

The entity parameter is an object containing the coordinates plus the width and height of the component.

The container is the reference of the DOM element where the component is attached. Each page has its own DOM element.

To summarize we can access each of the parameters internally using the "this" pointer.

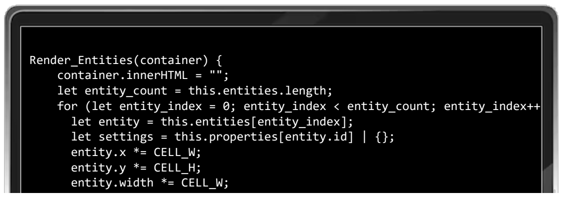


```
this.entity.x
this.entity.y
this.entity.width
this.entity.height

this.settings["type"]
this.settings["file"]

this.container
```

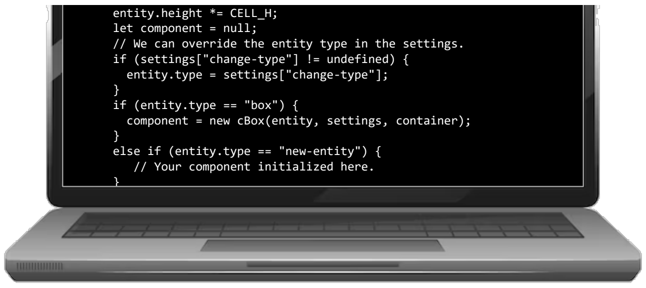
We'll use these more when we implement our own component later on.

So now how do you make your component visible so the framework can process it. Well, with some hacking. Yeah, hacking... it's fun.

To hack, go into "Frontend.js" and find this function:



```
Render_Entities(container) {
    container.innerHTML = "";
    let entity_count = this.entities.length;
    for (let entity_index = 0; entity_index < entity_count; entity_index++
      let entity = this.entities[entity_index];
      let settings = this.properties[entity.id] | {};
      entity.x *= CELL_W;
      entity.y *= CELL_H;
      entity.width *= CELL_W;
```


```
        entity.height *= CELL_H;
        let component = null;
        // We can override the entity type in the settings.
        if (settings["change-type"] != undefined) {
          entity.type = settings["change-type"];
        }
        if (entity.type == "box") {
          component = new cBox(entity, settings, container);
        }
        else if (entity.type == "new-entity") {
           // Your component initialized here.
        }
```

That's "Render_Entities". So just initialize your entity here. That's it!

Closing (For now.)

Ok, I hope that's a good primer but it's late and can't think of more. However, there will a large tutorial on how to program your own Frankus site.