# Create

To implement the create functionality, I first used jQuery to add an event handler to the preexisting “New Character” button, linking it to a ShowCreate() function, which is called when the button is clicked. This is performed in the Init() function, which is run as soon as jQuery determines the webpage is completely loaded.

|  |
| --- |
| const Init = function()  {      /\*          Purpose:    Initialize the page once jQuery determines the document is ready          #button\_create, #popup\_ui, and #char\_list exist in the actual HTML      \*/      $("#popup\_ui").hide();      ListChars();      $("#button\_create").on("click",ShowCreate);  }  $(document).ready(Init); |

In the ShowCreate() function, I create a table containing four text input fields, each with a unique ID, as well as a “submit” button, which I use another event handler to link to a CreateChar() function, which is called when the button is clicked. The table and button are added to the page by using jQuery to overwrite a “popup” <div>.

|  |
| --- |
| const ShowCreate = function()  {      /\*          Purpose:    Show the input fields for creating a new character, and link submit button to CreateChar()      \*/        let ui\_html = `<table><tr><td>Character name (English): </td><td><input id="char\_name"/><br></td></tr>`                  + `<tr><td>Internal name (GHG filename): </td><td><input id="char\_intn\_name"/></td></tr>`                  + `<tr><td>Price: </td><td><input id="char\_price"/></td></tr>`                  + `<tr><td>Icon URL (should be square): </td><td><input id="char\_pic\_url"/></td></tr></table>`                  + `<button id="submit\_char">Submit</button>`;      $("#popup\_ui").html(ui\_html);      $("#popup\_ui").show();      $("#submit\_char").on("click",CreateChar)  } |

In the CreateChar() function, I do the process of actually creating the character. I start by using jQuery to grab what the user typed into the fields created in ShowCreate(), then convert the price to an integer and the internal name to all caps. I then use the functions CheckIfInvalidInput() and CheckIfCharExists() to check if the user entered invalid values (see “Shared functions” section below), and if so, exit the function early with a return value of 1, indicating that the function stopped early (the value isn’t actually used, but rather is just there to help when reading the code).

If the verification checks are successful, I set a global variable to false. This global is used by the popups called if the verification checks are unsuccessful, which use is explained where they’re used in the “shared functions” section below. Next, I create an object with the user-defined data, and “push” it onto the end of the array with all the characters. Finally, I replace the block of the page containing the input dialog with a smaller simple dialog informing the user that the character has been created, then call ListChars() to refresh the character grid (see “Shared functions” below).

|  |
| --- |
| const CreateChar = function()  {      /\*          Purpose:    Create character specified character based upon fields set in ShowCreate(), and add to database if valid      \*/      // Grab inputs      let new\_name = $("#char\_name").val();      let new\_intn\_name = $("#char\_intn\_name").val();      let new\_price = parseInt($("#char\_price").val());      let new\_pic\_url = $("#char\_pic\_url").val();      // like DOS, we default to uppercase. This is consistent with the game proper, which is case insensitive and prioritizes uppercase on Linux systems.      new\_intn\_name = new\_intn\_name.toUpperCase();      if (CheckIfInvalidInput(new\_name,new\_intn\_name,new\_price,new\_pic\_url) == true)      {          // input is invalid; return early          return 1;      }      else if (CheckIfCharExists(new\_intn\_name) == true)      {          // don't create dupes of chars; return early          Popup\_InvalidInput(INVALID\_CHAR\_ALREADY\_EXISTS);          return 1;      }      else      {          // this global var may have been changed to true by Popup\_InvalidInput().          // Reset it to avoid issues, as the next time Popup\_InvalidInput() runs, the popup will be absent.          invalid\_popup\_enabled = false;      }      let new\_char =      {          char\_name : new\_name,          char\_intn\_name : new\_intn\_name,          char\_price: new\_price,          char\_icon: new\_pic\_url,      }      characters.push(new\_char);        let new\_popup = `<p id="new\_popup">Character ${new\_name} (<code>${new\_intn\_name}</code>) created!</p>`;      $("#popup\_ui").html(new\_popup);      ListChars();      return 0;  } |

# Read

To implement the read functionality, each character has a button underneath him/her/it labeled “Info” (see “shared functions” section). This button calls the function ShowRead() with the a parameter containing the character’s internal name (used to tell the function what character to display the info for).

The ShowRead() function takes the input parameter and checks if such a character actually exists, using a find() function to look for the first matching element in the array of characters (which will be the only matching element, as the create and update functions prevent creating duplicates). If it is found, the associated object is extracted, and jQuery is used to overwrite a “popup” <div> with data taken from the extracted object. If it isn’t found, some errors are logged to the console, as it should always exist.

|  |
| --- |
| const ShowRead = function(char2read)  {      /\*          Purpose:    Show extended information about specified character      \*/        let found\_char = characters.find((elem) => elem["char\_intn\_name"] == char2read)      if (found\_char != undefined)      {          let ui\_html = `<table><tr><td>Character name (English): </td><td>${found\_char["char\_name"]}</td></tr>`                      + `<tr><td>Internal name (GHG filename): </td><td><code>${found\_char["char\_intn\_name"]}</code></td></tr>`                      + `<tr><td>Price: </td><td>${found\_char["char\_price"]} studs</td></tr>`;          $("#popup\_ui").html(ui\_html);          $("#popup\_ui").show();          return 0;      }      else      {          // debug; this shouldn't ever run in production          console.log("Character "+ char2read +" not found. Perhaps this function was called incorrectly?  ")          return 1;      }    } |

# Update

To implement the read functionality, each character has a button underneath him/her/it labeled “Edit” (see “shared functions” section). This button calls the function ShowUpdate() with the a parameter containing the character’s internal name (used to tell the function what character to display the info for).

The ShowUpdate() function is very similar to ShowRead(). The difference is in what is drawn to the page if the character object is found in the array; a table very similar to the one in ShowCreate() is drawn, which is pre-filled with the specified character’s existing data from the found object. Unlike ShowCreate(), the UpdateChar() function called by the submit button is directly defined via an onclick in the written HTML, as I couldn’t figure out how to do this using jQuery’s .on() function.

|  |
| --- |
| const ShowUpdate = function(char2update)  {      /\*          Purpose:    Show input fields for editing an existing character, and link submit button to UpdateChar()      \*/      let found\_char = characters.find((elem) => elem["char\_intn\_name"] == char2update)      if (found\_char != undefined)      {          let ui\_html = `<table><tr><td>Character name (English): </td><td><input id="char\_name" value="${found\_char["char\_name"]}"/><br></td></tr>`                      + `<tr><td>Internal name (GHG filename): </td><td><input id="char\_intn\_name" value="${found\_char["char\_intn\_name"]}"/></td></tr>`                      + `<tr><td>Price: </td><td><input id="char\_price" value="${found\_char["char\_price"]}"/></td></tr>`                      + `<tr><td>Icon URL (should be square): </td><td><input id="char\_pic\_url" value="${found\_char["char\_icon"]}"/></td></tr></table>`                      + `<button id="submit\_char" onclick="UpdateChar('${found\_char["char\_intn\_name"]}')">Submit</button></p>`;          $("#popup\_ui").html(ui\_html);          $("#popup\_ui").show();          return 0;      }      else      {          // debug; this shouldn't ever run in production          console.log("Character "+ char2update +" not found. Perhaps this function was called incorrectly?")          return 1;      }  } |

The UpdateChar() function starts off very similarly to CreateChar(), which is why several portions were moved to shared functions between the two. Unlike CreateChar, an additional check is performed to allow the user to leave the character’s internal name as it was before editing, since that would always pass the check for if the character exists. Instead of creating a new object in the array, the code loops through the array of characters and directly replaces the values of the appropriate keyvalues with the values that were grabbed from the user input. This way, when the character grid is subsequently updated, the updated character’s icon and buttons remain in the same location as they were prior to updating.

|  |
| --- |
| const UpdateChar = function(char2update)  {      /\*          Purpose:    Update specified character based upon fields set in ShowUpdate(),                      updating the database if valid, and informing user that character has been updated.      \*/      // grab inputs      let new\_name = $("#char\_name").val();      let new\_intn\_name = $("#char\_intn\_name").val();      let new\_price = parseInt($("#char\_price").val());      let new\_pic\_url = $("#char\_pic\_url").val();      // like DOS, we default to uppercase. This is consistent with the game proper, which is case insensitive and prioritizes uppercase on Linux systems.      new\_intn\_name = new\_intn\_name.toUpperCase();          if (CheckIfInvalidInput(new\_name,new\_intn\_name,new\_price,new\_pic\_url) == true)      {          // input is invalid; return early          return 1;      }      else if (new\_intn\_name != char2update)      {          // only check internal name if it has changed          if (CheckIfCharExists(new\_intn\_name) == true)          {              // don't create dupes of chars; return early              Popup\_InvalidInput(INVALID\_CHAR\_ALREADY\_EXISTS);              return 1;          }      }      else      {          // this global var may have been changed to true by Popup\_InvalidInput().          // Reset it to avoid issues, as the next time Popup\_InvalidInput() runs, the popup will be absent.          invalid\_popup\_enabled = false;      }      for (let i = 0; i < characters.length; i++)      {          if (characters[i]["char\_intn\_name"] == char2update)          {              characters[i]["char\_name"] = new\_name;              characters[i]["char\_intn\_name"] = new\_intn\_name;              characters[i]["char\_price"] = new\_price;              characters[i]["char\_icon"] = new\_pic\_url;              // we could technically break out of the for loop now for minor speed savings          }      }      let updated\_popup = `<p id="updated\_popup">Character ${new\_name} (<code>${new\_intn\_name}</code>) updated!</p>`;      $("#popup\_ui").html(updated\_popup);      ListChars();      return 0;  } |

# Delete

To implement the read functionality, each character has a button underneath him/her/it labeled “Delete” (see “shared functions” section). This button calls the function ShowDelete() with the a parameter containing the character’s internal name (used to tell the function what character to display the info for).

ShowDelete() is similar to the ShowUpdate() function, but instead of a table, it uses the grabbed character data to ask if the user really wants to delete the character. The button is labeled “Confirm Delete”, and uses onclick to call the DeleteChar() with a parameter indicating which character to delete.

|  |
| --- |
| const ShowDelete = function(char2del)  {      /\*          Purpose:    Show deletion confirmation, but don't actually delete yet      \*/        // extract character to make our code less convoluted      let found\_char = characters.find((elem) => elem["char\_intn\_name"] == char2del)        if (found\_char != undefined) // undefined == doesn't exist      {          let ui\_html = `<p>You want to delete ${found\_char["char\_name"]} (<code>${found\_char["char\_intn\_name"]}</code>). Are you sure?</p>`                      + `<button id="submit\_char" onclick="DeleteChar('${found\_char["char\_intn\_name"]}')">Confirm Delete</button></p>`;          $("#popup\_ui").html(ui\_html);          $("#popup\_ui").show();          return 0;      }      else      {          // debug; this shouldn't ever run in production          console.log("Character "+ char2del +" not found. Perhaps this function was called incorrectly?")          return 1;      }  } |

DeleteChar() loops through the array of characters, and when it finds a character object containing an internal name that matches the parameter passed into the function, it stuffs the character’s name into a temporary variable and “splices” to remove the object from the array. The temporary variable is then used in a “popup” informing the user what character was deleted, replacing the deletion confirmation. ListChars() is then called to update the character grid (see “shared functions” section).

|  |
| --- |
| const DeleteChar = function(char2del)  {      /\*          Purpose:    Delete character, and inform user that character has been deleted.      \*/      let char2del\_name = "";      for (let i = 0; i < characters.length; i++)      {          if (characters[i]["char\_intn\_name"] == char2del)          {              char2del\_name = characters[i]["char\_name"]; // temporarily save name for popup later              characters.splice(i,1);              console.log(`deleted ${char2del}`)          }      }        let new\_popup = `<p id="new\_popup">Character ${char2del\_name} (<code>${char2del}</code>) deleted!</p>`;      $("#popup\_ui").html(new\_popup);      ListChars();  } |

# Shared functions

A few functions are called at multiple different places in the code.

The most integral of these is ListChars(), which is called to update the character grid every time the character array is modified. This function uses forEach() to loop through the character array. Each character’s icon and name is displayed, and buttons labeled “Info”, “Edit”, and “Delete” are placed underneath those. Each button is directly linked to the appropriate function using onclick, passing a parameter to indicate which character to process. As mentioned in the “Create” section above, I could not figure out how to use jQuery to do that. Once the grid is built, it is written to the page using jQuery.

|  |
| --- |
| const ListChars = function()  {      /\*          Purpose:    Create a "table" listing all characters in the database, with buttons under each of the chars      \*/        // this was originally an actual table instead of a div. Tables can't be resized dynamically based upon viewport width,      // so it's been changed to a div flexbox instead. Now each character is a separate table, to ensure the spacing is even.      let output\_text = `<div id="char\_table">`;      // share vars between loops so the memory address doesn't shuffle around      // (is it necessary? no. does it make me feel better? yes)      let found\_char\_ghg = "";      let found\_char\_icon = "";      let found\_char\_name = "";      characters.forEach((character) =>      {          found\_char\_name = character["char\_name"];          found\_char\_ghg = character["char\_intn\_name"];          found\_char\_icon = character["char\_icon"];          output\_text += `<table>`                      + `<tr>`                      + `<td colspan="3" style="text-align:center"><img src="${found\_char\_icon}" alt="${found\_char\_name}" width="64px" height="64px"><br>${found\_char\_name}</td>`                      + `</tr>`                      // if we write onclick directly, we can define an arg. For some inexplicable reason,                      // functions sometimes have to be called without parentheses in js, which is the case for                      // jQuery's event handler binder. There may be a way to work around it, but if there is,                      // I didn't learn how in class, and couldn't find how to online.                      + `<tr>`                      + `<td><button onclick="ShowRead('${found\_char\_ghg}')">Info</button><br>` // "Show Info" would sometimes wrap if the character list wrapped.                      + `<td><button onclick="ShowUpdate('${found\_char\_ghg}')">Edit</button><br>`                      + `<td><button onclick="ShowDelete('${found\_char\_ghg}')">Delete</button><br>`                      + `</tr></table>`;      });      output\_text += `</div>` // end "table"      $("#char\_list").html(output\_text)      $("#char\_list").show();  } |

CheckIfCharExists() is a function that checks if there is a given character with the internal name specified in this function’s parameter. It uses .find() on the characters array to search for a match. Since the result of .find() is an object, this function then checks if the result is “undefined”, which is the result if it isn’t found. If undefined, the function returns true (character exists), and otherwise the function returns false (character doesn’t exist).

|  |
| --- |
| const CheckIfCharExists = function(char2find)  {      /\*          Return value:   bool          Purpose:        Checks if a character with a given internal name exists in the database, and returns true or false appropriately      \*/      let found\_char = characters.find((elem) => elem["char\_intn\_name"] == char2find);      if (found\_char != undefined) // undefined == not found      {          // character exists          return true;      }      else      {          // character doesn't exist          return false;      }  } |

CheckIfInvalidInput() takes several input parameters, and runs them through some sanitation checks. If a check fails, Popup\_InvalidInput() is called with a parameter containing a reference to a global constant, and this function returns true. If all checks pass, this function returns false.

The first check verifies that none of the input values contain a null string, meaning the user didn’t fill in a value. This uses JavaScript’s unique “exact equality” evaluator using a triple equals, since using a standard double equals causes a price of zero to be treated as the same as a null string. Zero is a valid price (being free), so this is undesirable.

The second check verifies that the input price is a valid number, and isn’t “NaN”. Since the builtin parseInt() function is used to convert the user input to a number, the user can technically input a hexadecimal or scientific notation value, and those will be parsed as valid as long as the correct syntax is used.

The third check verifies that the price isn’t set to a negative value. Negative prices would likely cause weird bug if used in-game, so this prevents them.

The fourth and final check verifies that the price isn’t greater than 4294967295, the maximum size of an unsigned 32-bit integer. JavaScript’s limit (Number.MAX\_SAFE\_INTEGER, which is currently 9007199254740991) is much greater than that, but the game itself stores the player’s money in an unsigned 32-bit integer, and as such, exceeding that would cause integer overflow in-game.

|  |
| --- |
| const CheckIfInvalidInput = function(new\_name,new\_intn\_name,new\_price,new\_pic\_url)  {      /\*          Return value:   bool          Purpose:        Checks if input is invalid, and calls popup if true      \*/        // ideally we'd check if the image exists. I have no clue how to do that, though        if (new\_name === "" || new\_intn\_name === ""  || new\_pic\_url === "" || new\_price === "") // apparently "" =/= null? why?      {          // if we don't do triple equals, 0 gets parsed as "", breaking freebies          Popup\_InvalidInput(INVALID\_EMPTY\_FIELD);          return true;      }      else if (isNaN(new\_price)) // for some reason, you can't check equality with NaN, and have to use isNaN() instead      {          Popup\_InvalidInput(INVALID\_PRICE);          return true;      }      else if (new\_price < 0)      {          Popup\_InvalidInput(INVALID\_PRICE\_NEGATIVE);          return true;      }      else if (new\_price > 4294967295)      {          // fine with JavaScript, but would cause an integer overflow in-game          Popup\_InvalidInput(INVALID\_PRICE\_OVERFLOW);          return true;      }      else      {          // All valid!          return false;      }  } |

Popup\_InvalidInput() takes a single parameter, being a reference to a global constant indicating the reason the input is invalid. First, the function checks if a global variable is set to false, which indicates that the HTML element where the sub-popup would appear doesn’t exist. If it is false, jQuery’s .append() is used to add it to the main popup, the variable is set to false, and jQuery’s .css() is used to change the text color of the sub-popup to red.

After that, a variable to store the text that will be added to the sub-popup is initialized. A switch/case statement is used to check what the reason set in the function’s parameters was, and set the text that will be output accordingly. These use the same constants that are used to pass into this function from other functions. Finally, jQuery’s .text() function is used to change the sub-popup’s contents to those of the variable containing the text set in the switch/case.

|  |
| --- |
| const Popup\_InvalidInput = function(reason)  {      /\*          Purpose:    Show popup informing user of invalid input            This is separate from CheckInvalidInput, because checking if char already exists differs when editing vs creating          (don't want to complain when editing a character without changing the internal name)          Unusual name with underscore is because I originally had a separate function for if the character didn't exist,          but I realized it'd be better to use a switch/case to set the printed reason (which had the side effect of allowing          more informative reasons for why the input was invalid). The other (removed) function was called Popup\_CharAlreadyExists().      \*/      if (!invalid\_popup\_enabled)      {          // If the popup doesn't exist, create it.          $("#popup\_ui").append(`<p id="popup\_invalid"></p>`)          invalid\_popup\_enabled = true;          $("#popup\_invalid").css("color","#E3322E"); // Red value taken from Lego logo      }      let reason\_text = ""; // idk how much js really cares about variable type, but this way it's clear it's supposed to be a string      switch(reason)      {          case INVALID\_EMPTY\_FIELD:              reason\_text = "All fields must be filled!"              break;          case INVALID\_PRICE:              reason\_text = "Invalid price"              break;          case INVALID\_PRICE\_OVERFLOW:              reason\_text = "Price uses an unsigned 32-bit integer in-game, and cannot exceed 4,294,967,295."              break;          case INVALID\_PRICE\_NEGATIVE:              reason\_text = "Price must be positive or zero!"              break;          case INVALID\_CHAR\_ALREADY\_EXISTS:              reason\_text = "Character already exists!";              break;          case INVALID\_GENERIC:              // we used to have a generic option, which would just say "invalid input"              // it was deprecated in favor of more user-friendly prompts              // this is the same as default, ergo we don't break          default:              reason\_text = "Invalid input";      }      $("#popup\_invalid").text(reason\_text);  } |