Group 7

Christoffer Parkkila
Mathias Karlsson
Tommy Ernsund
Erik Pettersson

Minimum requirements

Website design

The responsive design of the website is created with the Bootstrap framework. Bootstrap have two container classes (container and container-fluid) that we mixed and made our own custom-container. When the width of the browser is greater than 1600px, the container class is active, locking the tiles at a fixed position and when it's less the container-fluid takes over. This decision was made because the container-fluid class uses a percentage-width that works very well on small devices and the container class uses a pixel-width, that looks better on bigger devices.

In the custom-container we have placed bootstraps rows which in turn are made up by several different column-classes. In the first row we have the ecraft-logo and the menu button in two separate columns, these columns always take up 50% each of the row.

The second row contains the entire tool-area and its columns have the classes col-xs-12, col-sm-6, and col-md-4. In these columns we have placed Metro-UI's tile-groups that in turn contains separate Metro-UI tiles. On a small screen-device one tile-group takes up the entire row, two tile-groups on a little bit larger and three tile-groups beyond that. We also changed the width of the tile-groups on screens bigger than 1600px and when in landscape-mode on a mobile, in order to fit three of Metro-UI tile-square classes within its tile-group.

The windows for the tools are made by customized Bootstrap modals. When pressing on a tile, some JavaScript are generating the modal, adding events to its buttons and appending it to a container-fluid class. Also, a button is generated that looks like the tile for the tool and placed in the tray of the bottom of the screen. The modal, the button and the tool are placed in an object together, so we know which button belongs to which modal and to be able to extract information from the tool in the modal if needed for later purposes.

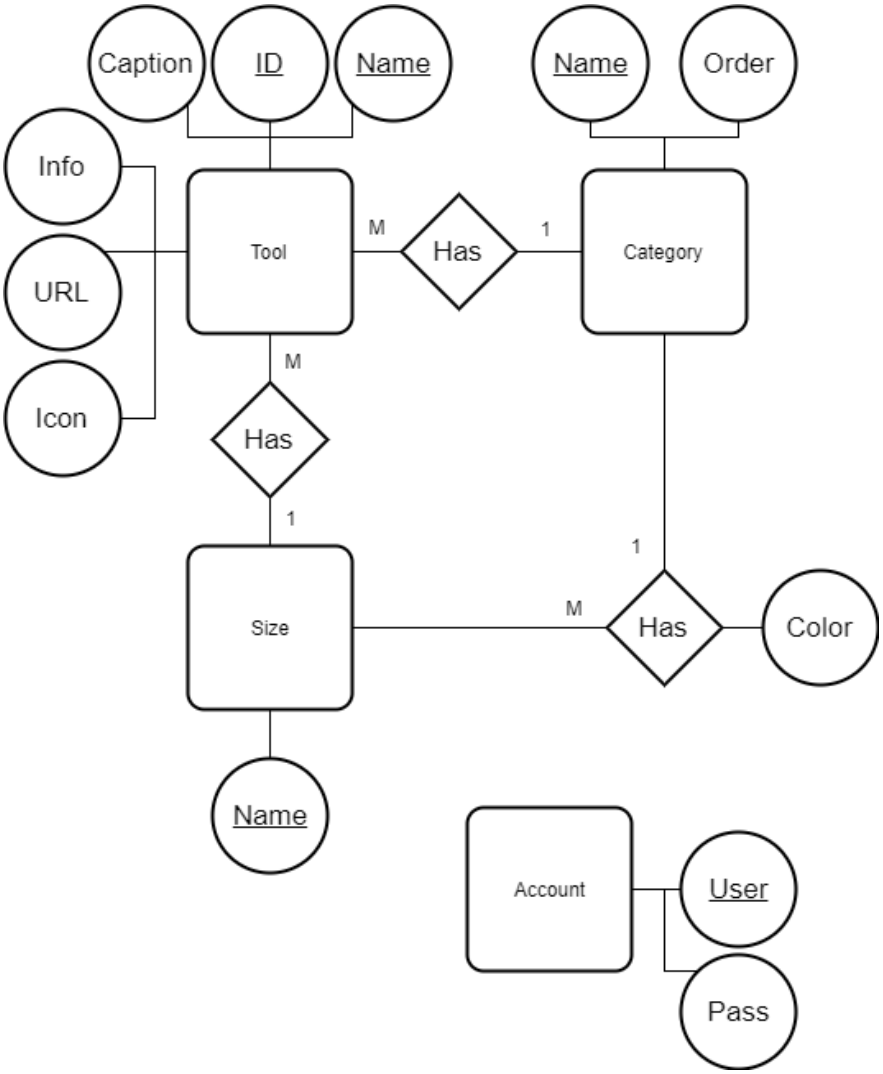
The tray-bar in the bottom of the screen have a fixed position and the highest z-index so it's always accessible. We choose to only allow one window to be open at a time and the idea is to navigate through them with the tray, and if one wish to see the underlying page, one only need to minimize the currently active window and not several that have stacked underneath. We also added a button on the tray-bar so one can hide and show the tray with a simple click.

Database design

Before deciding on any specific database structure, we discussed the general purpose of a database for this project. Given what the client wanted, we expected the following:

- A secure login, which requires some database backend with user data.
- Definition of available tools with various attributes.
- Categories for these tools.
- A color-scheme that is bound to tools somehow.

Next we create the actual design of the database, starting with an ER-diagram.



The starting-point of the design stems from the fact that every tool belongs to one category, and each tool has a size, this defines the tile size to display on the actual interface. Normally this ‘size’ could just had been hard-coded, but having a separate table helps against possible typos and for adding future sizes. But the main reason for its existence is the way we decided to handle the color-scheme. Each category has a color for each size, essentially creating a color-scheme per category.

While this design is sound, it raised a question-mark for us regarding the intention behind the project. Should a user not be able to define their own space? Under normal circumstances user data would be intertwined with the rest of the database in some manner, while not just being a small stand-alone table.

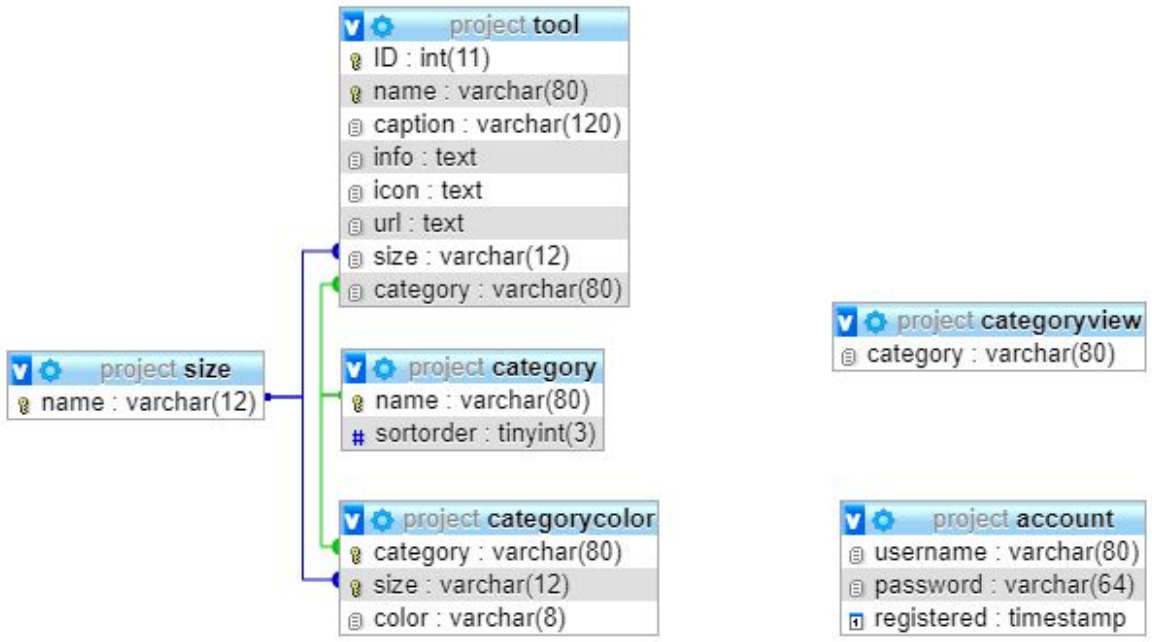
The implementation is pretty much a 1 to 1 translation of the ER-diagram, except for added procedures and views and some later addition, and removal of features to ensure a secure login. I will describe the views, procedures and their functionality below.

Views

- *CategoryView*
  Provides rows with a single column ‘name’ that returns available categories in a order defined within the database.

Procedures

- *GetToolsByCategory(tool)*
  Provides rows with tools for a specific category, and joins in color depending on its size and category. Columns: ‘name’, ‘caption’, ‘info’, ‘icon’, ‘url’, ‘size’, ‘color’.



Generating the tool tiles with php

Each category is fetched through a database query. Then, a nested for-loop containing code for generating the page content is executed. The content is generated with the echo function that contains HTML code. The first for-loop runs through all categories and the for-loop inside of the first one then generates all tools for that particular category, again using the echo function.

All categories and tools are fetched from the database, and by doing so it is easy to add more categories and tools to the page if needed; only the database needs to be taken into account.

Secure login

When a new user is registered, the password is hashed with the password_hash() function in PHP. This function also adds the algorithm used, cost and salt to the hashed password. Higher cost increases the number of operations it takes to compute the hash, making it harder for someone to hash the most common passwords with each user's salt with the goal to find login credentials. Having all this information stored in the same string makes it a lot easier to verify passwords later on. Note that in our case you can't register a new user through the website yet, so user login credentials was added manually to the database.

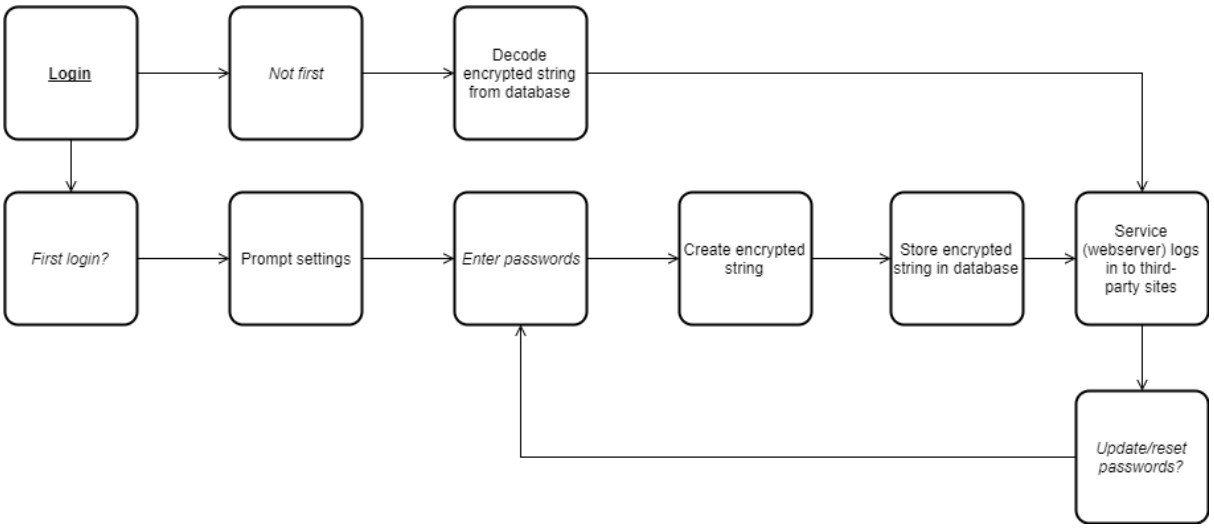
After a user has entered his/her username and password, we use mysqli_real_escape_string() to escape special characters. The hashed password belonging to the entered username is then fetched from the database. Now we only need use password_verify() to verify that the entered password matches the hash, and if it does log in the user.

Advanced

Universal login

A universal login system, at this current time seems unfeasible, without a common standard which all (read: majority) of webpages adopt, the closest thing to a universal login system would be that of an Google ID, or Facebook Connect. While these technologies are a step in the right direction (subjective), they are far from universal. And this presents a problem for our project in particular. Far from every maker tool supports these, so, should we limit the range of the project to tools whom support these types of logins? We think this is unreasonable.

We have an alternative. It is possible to store individual passwords for other tools in our database, originally, we thought this might pose a security issue but realized that if we encrypt the tool passwords, with the login password of the user (Not hashed! Important!), it should be solid, security-wise. Note though, that storing passwords with encryption is highly irregular, and should usually be done with hashes. But since we need to login to other services, we will need the raw passwords server-side later. For illustrative purposes, we imagine the following structure for implementation.



Performing the actual encryption is very simple with the use of the built-in function *mcrypt_encrypt()* and like-wise *mcrypt_decrypt().*

But, even if we have all the passwords, we are required to login to the site in question. Upon further research on the subject, there are some options, such as

- Sockets
- cURL

Both alternatives require the address of the login page for the tool website. Assuming it should be possible to add tools over time to this project, this presents one more problem. All webpages are not named in the same way.

We could save the address to the login-page manually upon adding a new tool, but what if that tools web page is changed? This would require a person continuously updating our database, that's an issue, and not a good solution.

What about a web crawler (spider-bot used to discover webpage files) to discover the webpage? While possible (maybe), it would have to be intelligent enough to distinguish a script as responsible for login. Or if the crawler acquires all file addresses and we parse these through an HTML-parser, looking for <form> tags, hoping to get lucky? But even if we do get results from this, how would the script distinguish one tag from another, which file contains the login form?

But let's for a moment contemplate the possibility that we get past this hurdle, we then run into one more problem. If the login is successful through cURL or sockets in PHP, the cookie containing the session variable is sent to the server making the request. Saving this cookie and reusing it is possible, but this means that future requests to the webpage would be required to go through a PHP proxy, since the server has logged in for us.

At the end of the day, we have more questions than answers, trying to implement this would require far more than the allotted time for our project, if it is even possible in the first place with the current structure of the internet.

During our research, we found out about a project that has been trying to tackle this issue, called *WebID.* Their solution takes advantage of already built-in features in one of the internet's' web protocols, creating a unique identifier in the form of a security protocol, but have yet to overcome the fact that individual websites must enable support for their system.

Running executables

As web browsers run web pages in a sandbox, you can't simply execute any application you want outside of that environment. One way around this is to register the application you want to launch to a URI scheme. How you make handlers for custom URI schemes varies depending on which operating system you use (and other things like your desktop environment if you use a Linux distribution), so in this project we focused on Windows.

We wanted to be able to launch Ardublocks via the browser, which is an extension made in Java for Arduino IDE. To do this we added an the URI scheme "arduino:" to the registry as follows:

```
HKEY_CLASSES_ROOT
  arduino
    (Default) = ""
    URL Protocol = ""
    shell
      open
        command
          (Default) = "C:\Program Files (x86)\Arduino\arduino.exe\"
```

Now we were able to launch Arduino.exe through HTML with "arduino:" as the URL. If you want to launch it with any flags (the link would then look like this: arduino:*flags*) you can add "%1" after the path in the registry. The reason for adding flags in our case was to see if we could make Arduino IDE also launch the extension Ardublocks at startup. After looking through the documentation for Arduino to see which actions were available via CLI, we concluded that the one we were looking for were not. Maybe it would be possible via a URI scheme to launch the Ardublocks .jar file itself, but then you would also need to add Java to the environment variable PATH.

So while we were able to launch Arduino IDE via the browser, we were not able to launch Ardublocks. If a user also wanted to do the same, we could supply them with a .reg file to make the same changes as us in the registry. We would not recommend this though, as the path to the application may vary among users.

In conclusion; to launch an application via a web browser, the developers should add support for it so an URI scheme is created upon installation (to set the right path in the registry). And as in our case, the Arduino developers would also need to make it possible to launch extensions via CLI.

X-Frame-Options

The x-frame-options can be set to prevent other sites to frame their webpages. This option can be set directly to a server's configuration and therefore apply to all pages residing on that server. The x-frame-options can take the values:

- DENY
  (Denying all attempts to frame a page).
- SAMEORIGIN
  (Allow to frame pages if the page that trying to frame resides on the same server).
- ALLOW-FROM
  (Takes additional options, that specifies which pages should be allowed to frame the page).

If a page isn't specified in the ALLOW-FROM nor resides in the SAMEORIGIN or the server have the option set to DENY, the page cannot be framed. This refusal to frame a page is done by the browser if the x-frame-options are present in the HTTP(S) response header. This means that the row that containing the x-frame-options is the reason some pages cannot be framed, and if removed it wouldn't be a problem.

```
array (size=14)
  0 => string 'HTTP/1.0 200 OK' (length=15)
  1 => string 'Date: Sat, 28 Oct 2017 16:12:40 GMT' (length=35)
  2 => string 'Expires: -1' (length=11)
  3 => string 'Cache-Control: private, max-age=0' (length=33)
  4 => string 'Content-Type: text/html; charset=ISO-8859-1' (length=43)
  5 => string 'P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."' (length=67)
  6 => string 'Server: gws' (length=11)
  7 => string 'X-XSS-Protection: 1; mode=block' (length=31)
  8 => string 'X-Frame-Options: SAMEORIGIN' (length=27)
  9 => string 'Set-Cookie: 1P_JAR=2017-10-28-16; expires=Sat, 04-Nov-2017 16:12:40 GMT; path=/; domain=.google.se' (length=98)
  10 => string 'Set-Cookie: NID=116=V8ek3kO3RF_7Kz85bz8yQVVGzE4CpKF4Wqnb-Gm4Av8Ytz9JC5t13BHjtzAOiIjzOWuYGvTW-iOdzIzAhznJSOimieer
  11 => string 'Alt-Svc: quic=":443"; ma=2592000; v="41,39,38,37,35"' (length=52)
  12 => string 'Accept-Ranges: none' (length=19)
  13 => string 'Vary: Accept-Encoding' (length=21)
```

PHP var_dump() of get_headers("http://www.google.se");

The removal of the x-frame-options can be done with a chrome-extension. The extension is made up by a manifest.json and a JavaScript file. The manifest.json is needed for all chrome extensions and is basically a file that specifies some meta-data, which API's to use and linking in the JavaScript. For more information of the manifest.json file, see comments in the source code. Chrome provides several API's for developers to make extensions, and one of these are the chrome.webRequest API that are used for analyzing the traffic and blocking/modifying requests. This API have several events that are useful and for modifying the HTTP(S) response headers, the onHeaderReceived event can be used.

The JavaScript removing the x-frame-options looks like:

```
chrome.webRequest.onHeadersReceived.addListener(
  /* First parameter to addListener() */
  function(details) {
    for (var i = 0; i < details.responseHeaders.length; i++) {
      if (details.responseHeaders[i].name.toLowerCase() == "x-frame-options") {
        details.responseHeaders.splice(i, 1);
        return { responseHeaders: details.responseHeaders };
      }
    }
  },
  /* Second parameter to addListener() */
  {urls: ["<all_urls>"]},
  /* Third parameter to addListener() */
  ["blocking", "responseHeaders"]);
```

The onHeaderReceived event is raised every time an HTTP(S) response header is received. The second parameter of the addListener decides which events that should execute the first parameter (the callback function), in this case we want all responses to be executed. To include the response headers into the details object, the string “responseHeaders” needs to be specified in the third parameters array. We also have included the string “blocking”, which means that the callback function is going to be executed synchronously so the response header can be modified (default is asynchronous).

The callback function is looping through the responseHeaders dictionary and removing the x-frame-options. To make the browser act like it is the server that have sent the modified response header an object containing a responseHeaders property needs to be returned, that contains the modified response header residing in the details object.

To test the extension one can navigate to the extensions tab, check the developer-mode checkbox and press the load unpacked extensions and open the x-frame-remover folder.

For more detailed information see the following documentation:
https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/webRequest/onHeadersReceived
https://developer.chrome.com/extensions/webRequest

PHP proxy

A forward proxy works by taking a request from the client/clients and then, depending on its functionality, sending the requested page back to the client. In order to modify this requested page before sending it back to the user it is necessary to retrieve the html source code. The problem is to find a generic way to modify various web pages that probably do not follow any specific structural standard. An easier solution would require that the proxy knew about all the website that are to be visited so that certain code would be executed for certain web pages. However, more issues would arise from that solution: what happens when the web pages get updated? And what to do when another web page needs to be added?

The easiest way to retrieve the source code of a page with php is by using the **file_get_contents** function. This will return the page as a string which then needs to be parsed in some way. allow_url_fopen needs to be set to ‘true’ in order for this to work.

A simple HTML DOM parser written in PHP could be used such as: http://simplehtmldom.sourceforge.net/. To create a DOM from an URL you use a function called **file_get_html**. In its parameters you pass the requested web page and then the page is returned as a DOM object. If a variable $html, contains the returned DOM, then $html->find() would lookup any needed element from that page. For example; $html->find(‘p’, 1)->innertext =’näver gonna give you up’, would change the content of the first occurrence of an paragraph element (hence the parameter ‘1’).