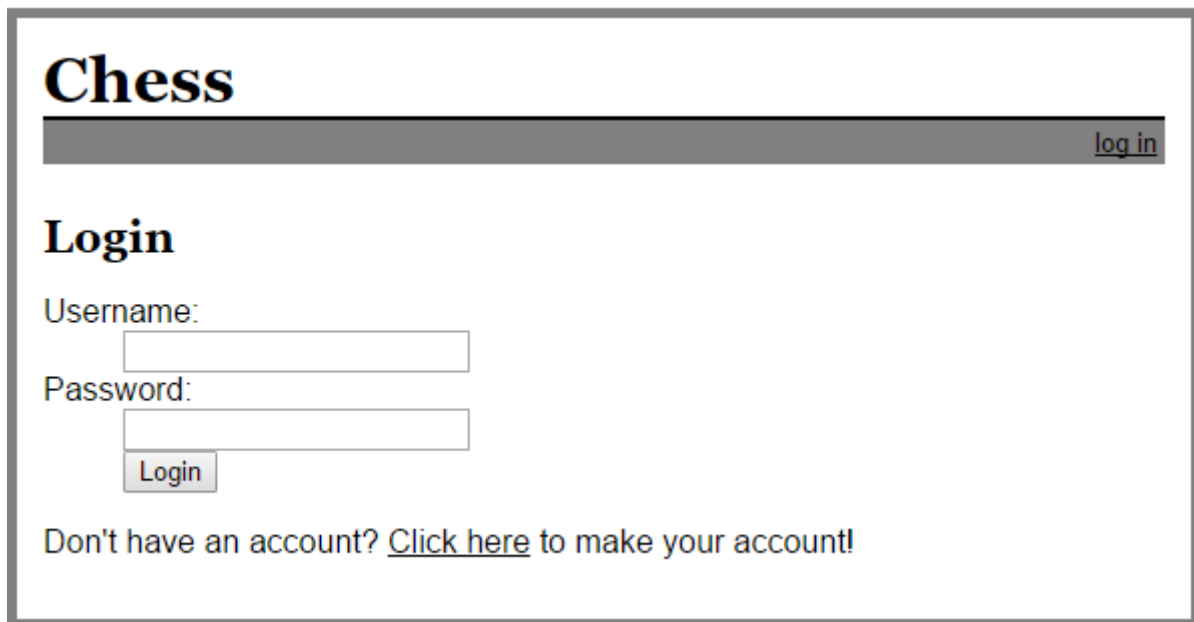


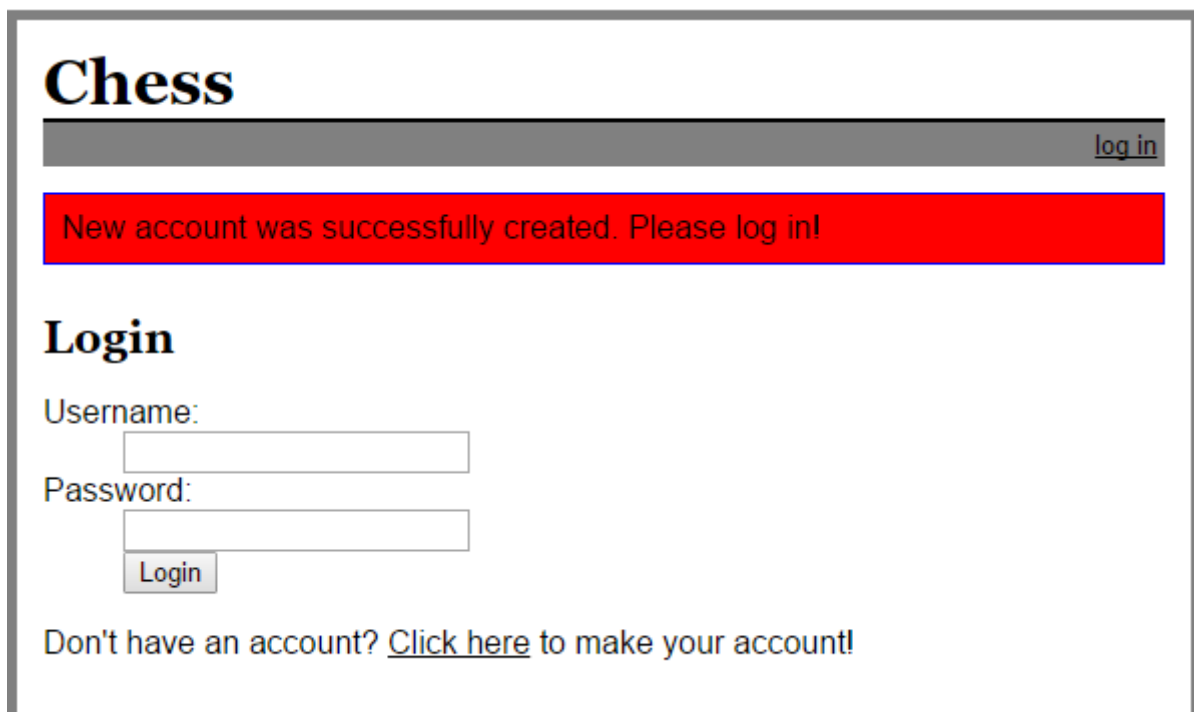
I. User's perspective

After you log in the page, you will see a page with a "Sign in" option for registered users, and a "Create Account" option for new user



The screenshot shows the login page for a website titled "Chess". At the top left is the word "Chess" in a large, bold, serif font. To its right is a dark grey horizontal bar containing the text "log in" in a small, white, sans-serif font. Below the title, the word "Login" is displayed in a bold, serif font. Underneath "Login" are two input fields: "Username:" followed by a text box, and "Password:" followed by a text box. Below the password field is a small, grey button with the word "Login" in a sans-serif font. At the bottom of the form area, there is a line of text: "Don't have an account? [Click here](#) to make your account!".

After you create your account, you will be redirected to the home page, with a message to log in using your credentials your registered.



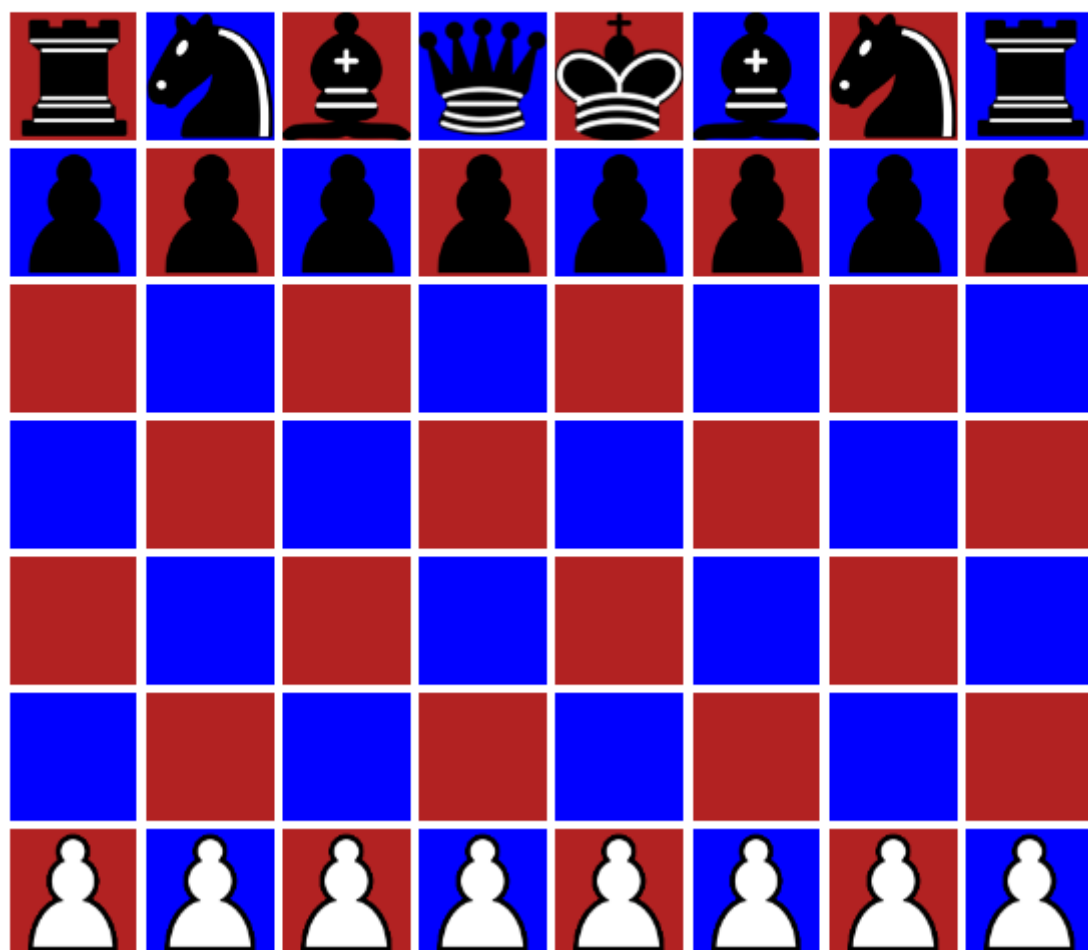
This screenshot shows the same login page as the previous one, but with an additional message. A red rectangular box with a blue border is positioned above the "Login" section. Inside this box, the text "New account was successfully created. Please log in!" is written in a black, sans-serif font. The rest of the page, including the "Chess" title, the "log in" link, the "Login" heading, the username and password fields, the "Login" button, and the "Don't have an account?" link, remains identical to the previous screenshot.

Log in, and you will see a chessboard. The board size is based on the view height of user's device.

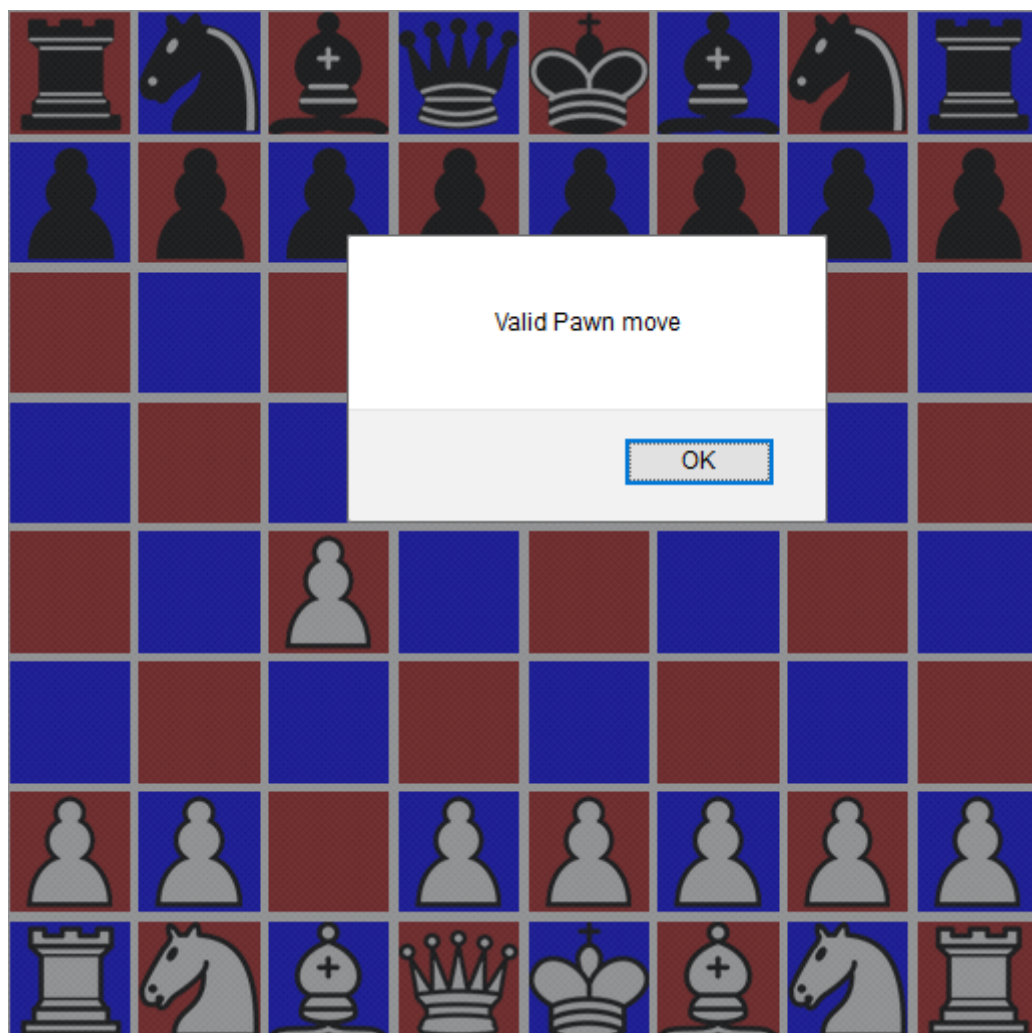
Chess

[log out](#)

Drag the pieces to play:

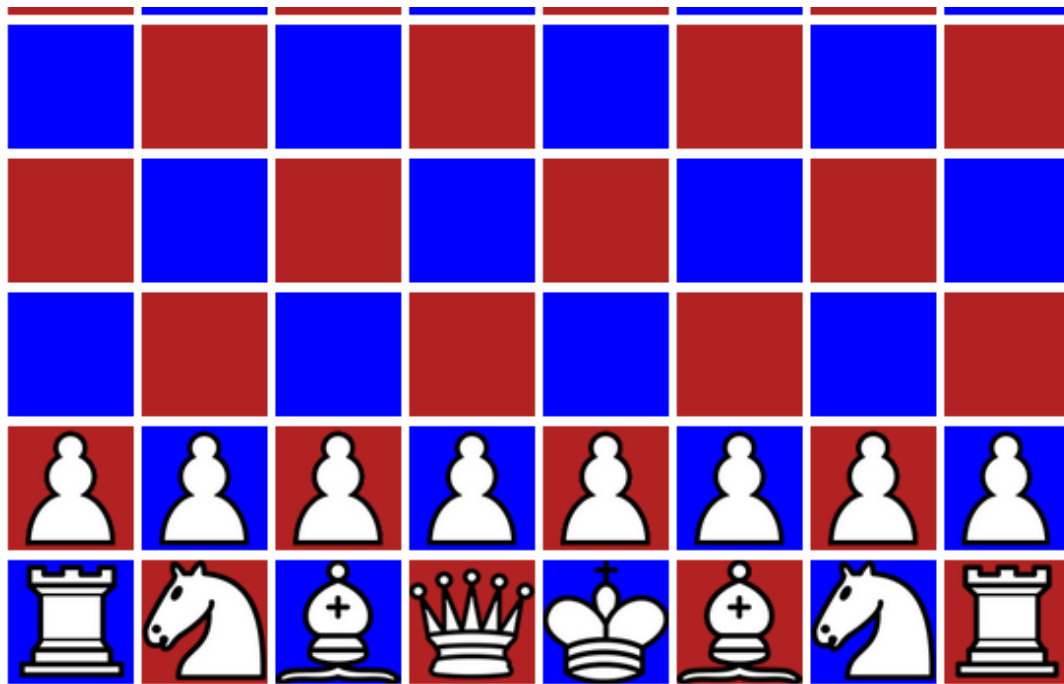


Dragging the pieces, and there will be an alert whether your move is valid correct or not.



The default browser for this document is Firefox

When the piece moves, there will be an alert whether the move is legal or not. Also, there is a record of the last move on the screen:



success

- **blackPawn**
e6

The last move was blackPawn to e6.

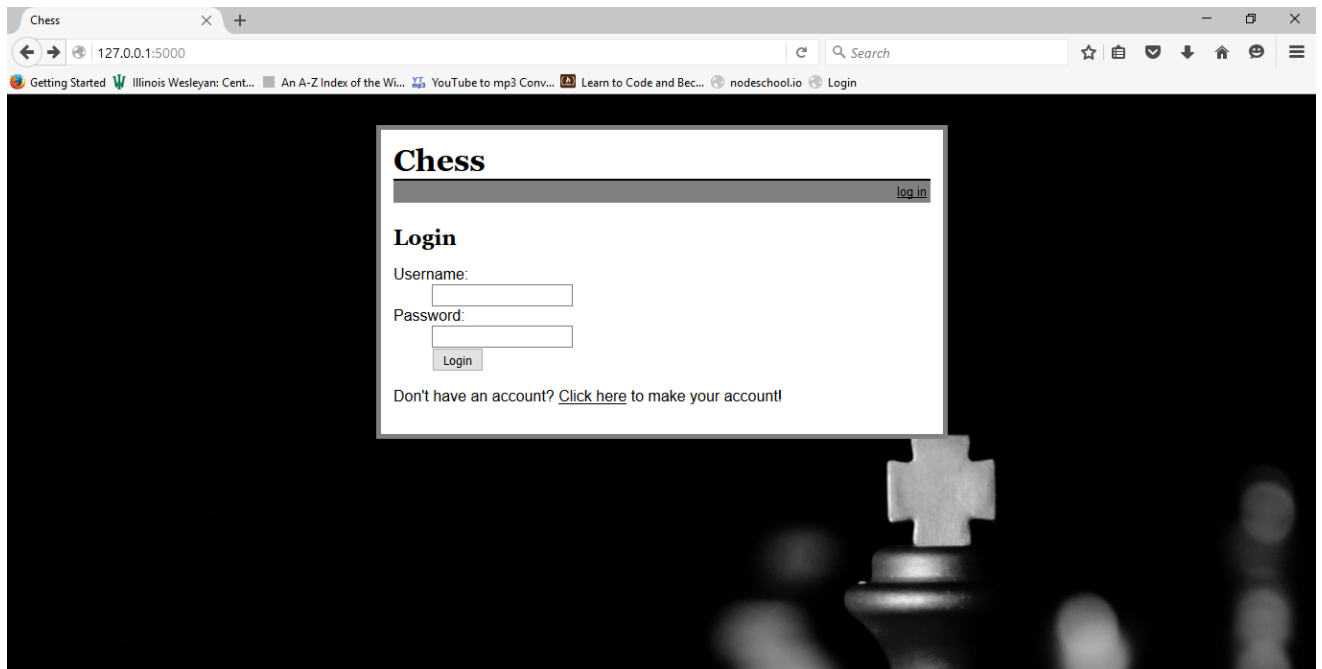
There is a bug: a second board appears on the screen when player make the move. It doesn't affect the game mechanic at all, but we prefer to get rid of it.

II. Documentation

The web form is somewhat based on Flask library that is heavily modified by our team.

1. Client Side

Here is the web page when you first go to the page:



Here is the code behind it

```

1  <!doctype html>
2  <title>Chess</title>
3  <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='StyleSheet1.css') }}">
4  <script src="../../static/JavaScript1.js"></script>
5  <div class="page">
6      <h1>Chess</h1>
7      <div class="metanav">
8          {% if not session.logged_in %}
9              <a href="{{ url_for('login') }}">log in</a>
10             {% else %}
11                 <a href="{{ url_for('logout') }}">log out</a>
12             {% endif %}
13         </div>
14         {% for message in get_flashed_messages() %}
15             <div class="flash">{{ message }}</div>
16         {% endfor %}
17         {% block body %}
18             {% endblock %}
19     </div>
20
21

```

The page is saved as "layout.html" in the Templates folder

From line 5 to 19, if the user is not logged in, "login.html" is redirected to

```

1  {% extends "layout.html" %}
2  {% block body %}
3      <h2>Login</h2>
4      {% if error %}<p class="error"><strong>Error:</strong> {{ error }}{% endif %}
5      <form action="{{ url_for('login') }}" method="post">
6          <dl>
7              <dt>Username:
8              <dd><input type="text" name="username">
9              <dt>Password:
10             <dd><input type="password" name="password">
11             <dd><input type="submit" value="Login">
12         </dl>
13     </form>
14     <p>Don't have an account? <a href="{{ url_for('createAccount') }}">Click here</a> to make your account!</p>
15 {% endblock %}
16

```

login.html

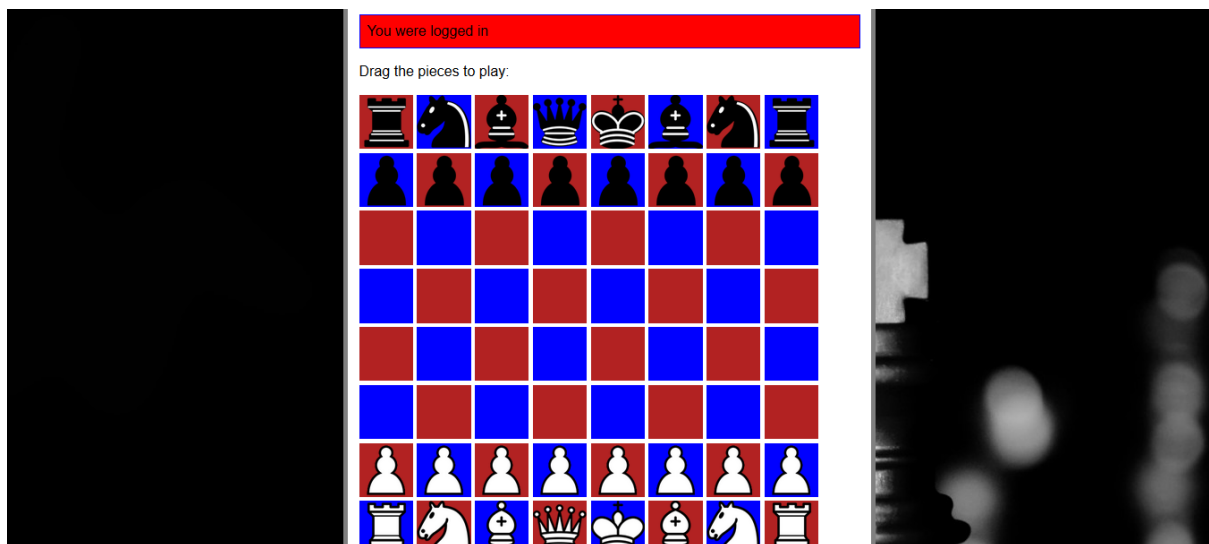
The form will collect the credentials from the user, which is checked with the database whether user can log in or not

To see how the server works, look for page 15 of the documentation

In case the user doesn't have an account, he can make a new one (line 15).

When user makes a new account, the server will check with the database to make sure the current username has not been taken already in the database (to see how this work, turn to page 15).

After the user is logged in, there will be a chessboard and a form that is hidden from the client's eyes.



Here is an excerpt of the HTML code for the chessboard

```
6 <div id="a8" class="div2 blackRook" draggable="true" ondragstart="drag(event)" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
7 <div id="b8" class="div1 blackKnight" draggable="true" ondragstart="drag(event)" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
8 <div id="c8" class="div2 blackBishop" draggable="true" ondragstart="drag(event)" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
9 <div id="d8" class="div1 blackQueen" draggable="true" ondragstart="drag(event)" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
10 <div id="e8" class="div2 blackKing" draggable="true" ondragstart="drag(event)" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
11 <div id="f8" class="div1 blackBishop" draggable="true" ondragstart="drag(event)" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
12 <div id="g8" class="div2 blackKnight" draggable="true" ondragstart="drag(event)" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
13 <div id="h8" class="div1 blackRook" draggable="true" ondragstart="drag(event)" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
14 <br>
```

First line of the chessboard

The chessboard consists of an 8x8 board. The keyword break `
` ensures that each line has only 8 squares.

- And here is the CSS stylesheet that is responsible for the chess pieces, the red (firebrick) and the blue squares (tiles) of the board, as well as some colors and fonts of the page.

```

1  .div1 {
2      display: inline-block;
3      width: 10vh;
4      height: 10vh;
5      background-color:blue;
6  }
7  .div2 {
8      display: inline-block;
9      width: 10vh;
10     height: 10vh;
11     background-color:firebrick;
12 }
13 .blackRook {
14     background-image: url(../static/images/blackRook.png);
15     background-size: 10vh 10vh;
16 }
17 .blackKnight {
18     background-image: url(../static/images/blackKnight.png);
19     background-size: 10vh 10vh;
20 }
21 .blackBishop {
22     background-image: url(../static/images/blackBishop.png);
23     background-size: 10vh 10vh;
24 }
25 .blackQueen {
26     background-image: url(../static/images/blackQueen.png);
27     background-size: 10vh 10vh;
28 }
29 .blackKing {
30     background-image: url(../static/images/blackKing.png);
31     background-size: 10vh 10vh;
32 }
33 .blackPawn {
34     background-image: url(../static/images/blackPawn.png);
35     background-size: 10vh 10vh;
36 }

```

An excerpt from StyleSheet1.css.

The size is measured in `vh`, which stands for view height. This allows the chessboard to eliminate the need to scroll down and up whenever players want to move the pieces. The reason is that each device has different screen size and resolution settings; let the board be relative to the displaying device is the best call.

Now if we go back to the HTML of the chessboard, and look at an example line of code

```

6  <div id="a8" class="div2 blackRook" draggable="true" ondragstart="drag(event)" ondrop="drop(event)" ondragover="allowDrop(event)"></div>

```

Notice each tile has its own id (`id="a8"`), a class `class="div2 blackRook"`, and a number of handler for each events that can happen to the tiles

```

    draggable="true" ondragstart="drag(event)" ondrop="drop(event)" ondragover="allowDrop(event)"

```

Each piece is a background image for the tiles; this allows the piece to capture another piece. However, there are two bugs: the tiles are draggable, and while they do not capture other tiles, they

can capture the pieces. Also, pieces can capture friendly pieces. We hope that we can fix it in the near future.

On the board, there is also a form that is hidden from the users:

```
80 <form id="form1" action="/action1" method="post">
81   <input type="hidden" name="textInput1" id="textInput1"></input>
82   <input type="hidden" name="textInput2" id="textInput2"></input>
83   <input type="hidden" name="textInput3" id="textInput3"></input>
84 </form>
```

At the early stage of developmental process, we wish to see the information before sending it to the database, so we keep it in a form. The form houses the chess piece (included in the div class), the piece's starting and ending destinations (which are in the divs' ids). This information is vital for us to design an algorithm to ensure the rules are followed.

Note the three text input IDs: `textInput1`, `textInput2`, and `textInput3`: they will be used quite often in the JavaScript. Basically, `textInput1` is for the piece name `pieceName1`, `textInput2` is the dropping position `dropPosition1`, and `textInput3` is the starting position `startPosition1`.

In the JavaScript code:

To recap all events in a single tile:

```
<div id="a8" class="div2 blackRook" draggable="true" ondragstart="drag(event)" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
```

- Drag event:

```
240 function drag(ev) {
241   ev.dataTransfer.setData("text", ev.target.className.split(" ")[1]);
242   ev.target.className = ev.target.className.split(" ")[0];
243   document.getElementById("textInput3").value = document.getElementById(ev.target.id).id;
244 }
```

As the class name is a string with 2 words separate by a space (`div2 blackRook`), line 241 will split the string and take the index 1 (in this case, `blackRook`), and transfer it in the target (dropping point) of the new div. Thus, when the `blackRook` goes to a new div, the old class that used to harbor `blackRook` `div2 blackRook` will now just be `class="div2"`, and the data transfer will be the `blackRook` (for the dropping class to contain).

On line 243, the location (id) of the drag event will be stored to the form whose id is `textInput3`.

- Drop event:


```

247 function drop(ev)
248 {
249     ev.preventDefault();
250     var data = ev.dataTransfer.getData("text");
251     ev.target.className = ev.target.className.split(" ")[0];
252     ev.target.className += " " + data;
253
254     pieceName1 = ev.target.className;
255
256     pieceName1= pieceName1.slice(5);
257     startPosition1 = document.getElementById("textInput3").value; //store Starting position to a variable startPosition1
258
259
260     document.getElementById("textInput1").value = pieceName1;
261     document.getElementById("textInput2").value = document.getElementById(ev.target.id).id;
262     dropPosition1 = document.getElementById("textInput2").value; //store drop position to dropPosition1
263
264     validMove(pieceName1, dropPosition1, startPosition1); //check if the move is valid
265
266
267     fn1(pieceName1, dropPosition1);
268 }

```

From line 250 to 252, `var data` contains the transfer of the dragging event above (in this example, `blackRook`). The dropping class will only take the first part of its class (the div), and replace anything following that (the chess piece that the dropping point is now harboring) with `blackRook`. This is how capturing is implemented

As the class name `div2 blackRook` consists of `div2`, and `blackRook`, we wish to store `blackRook` to a new JavaScript variable `pieceName1`. Thus, the string `div2 blackRook` is processed on line 256 accordingly.

Meanwhile, `startPosition1` contains the starting point of the dragging event from the form input `textInput3`, while form input `textInput2` will take the dropping id (which is also the location), before handing its value to variable `dropPosition1`.

After that, the move is to be verified if it's valid or not:

```

264     validMove(pieceName1, dropPosition1, startPosition1); //check if the move is valid

```

Then, the move is store in the database through a function:

```

267     fn1(pieceName1, dropPosition1);

```

- Valid move

Before we move on, take a look at the div's IDs in the chessboard.html file: the ids also represent the chessboard coordinates. To perform mathematical operation on the coordinate, we need to assign each id to a unique number, in a dictionary:

```

2     var dictionary1 = //map BOARDVALUE2 to boardValue, as we can't assign number to tiles' ids
3     {
4         a8:1, b8:2, c8:3, d8:4, e8:5, f8:6, g8:7, h8:8,
5         a7:9, b7:10, c7:11, d7:12, e7:13, f7:14, g7:15, h7:16,
6         a6:17, b6:18, c6:19, d6:20, e6:21, f6:22, g6:23, h6:24,
7         a5:25, b5:26, c5:27, d5:28, e5:29, f5:30, g5:31, h5:32,
8         a4:33, b4:34, c4:35, d4:36, e4:37, f4:38, g4:39, h4:40,
9         a3:41, b3:42, c3:43, d3:44, e3:45, f3:46, g3:47, h3:48,
10        a2:49, b2:50, c2:51, d2:52, e2:53, f2:54, g2:55, h2:56,
11        a1:57, b1:58, c1:59, d1:60, e1:61, f1:62, g1:63, h1:64
12    };

```

Notice each line is an increment of 8 (8, 16, 32...) As in the form, `startPosition1` and `dropPosition1` (the divs' ids) are the dictionary's keys, and we want to take the numbers out of the keys:

```
var startNumber1 = dictionary1[startPosition1];
var endNumber1 = dictionary1[dropPosition1];
```

- Rook

Rooks move vertically and horizontally.

When you look at the vertical lines (chess term: files) of the dictionary, you will see the numbers have the difference of 8. That is, in every file (vertical line), the numbers will have the same remainder when they are divided by 8

Mathematical equation:

$$int1 \bmod 8 = int2 \ (int2 < 8)$$

$$\Leftrightarrow (int1 + 8x) \bmod 8 = int2, \text{ where } 8x \bmod 8 = 0$$

Notice *int1* and *int2* are starting and dropping positions in the same file.

$$\text{Example: } 10 \bmod 8 = 2 \Leftrightarrow (10 + 16) \bmod 8 = 2$$

In the JavaScript, line 82 to 90:

```
82  if (pieceName1 == "whiteRook" || pieceName1=="blackRook")
83  {
84      var module1 = startNumber1 % 8;
85      var module2 = endNumber1 % 8;
86
87      if (module1 == module2) //if the white rook moves vertically
88      {
89          alert("rook valid file(vertical) move");
90      }
```

If `startNumber1` and `endNumber1` have the same remainder when they are divided by 8, then they are on the same file. Thus, the rook moves vertically, and the move is valid.

If the rook doesn't move vertically, then it has to move horizontally (in row); otherwise it's an invalid move.

```
92  if (module1 != module2) // if white Rook doesn't move on the vertical lines (files).
93  {
94      var dummyVar1 = rookRow(startNumber1, endNumber1);
95      if (dummyVar1 == true)
96      {
97          alert("rook valid row move");
98      }
99      //if (rookRow(startPosition1, dropPosition1)==false)
100     if (dummyVar1 == false)
101     {
102         alert("INVALID ROOK MOVE. NOOB");
103     }
104 }
```

Function `rookRow` will check if the rook moves horizontally or not. We decided to make this a function, as the queen also inherits this trait as well (so we can call the function again).

If we look at one horizontal line (row) of the dictionary:

```
a4:33, b4:34, c4:35, d4:36, e4:37, f4:38, g4:39, h4:40
```

Notice any number in the line is between 33 and 40 inclusive. Thus, `startNumber1` and `endNumber1` are between 33 and 40 on this particular row, if the rook is to move horizontally.

The same idea applies to other row as well: for each row, the `startNumber1` and `endNumber1` have to be between the opening and ending values.

Here is an excerpt from function `rookRow`:

```
27 function rookRow(startNumber1, endNumber1) //check if white rook moves in the horizontal rows or not
28 {
29     if (startNumber1 >= 1 && startNumber1 <= 8 && endNumber1 >= 1 && endNumber1 <= 8 )
30     {
31         return true;
32     }
33
34     if (startNumber1 >= 9 && startNumber1 <= 16 && endNumber1 >= 9 && endNumber1 <= 16)
35     {
36         return true;
37     }
38 }
```

From line 29 to 32, if `startNumber1` and `endNumber1` are between 1 and 8 (if they are on the same row), function returns true. Otherwise, function returns false.

If the rook fails the conditions for moving horizontally or vertically, the move is invalid.

```
92 if (module1 != module2) // if white Rook doesn't move on the vertical lines (files).
93 {
94     var dummyVar1 = rookRow(startNumber1, endNumber1);
95     if (dummyVar1 == true)
96     {
97         alert("rook valid row move");
98     }
99     //if (rookRow(startPosition1, dropPosition1)==false)
100     if (dummyVar1 == false)
101     {
102         alert("INVALID ROOK MOVE. NOOB");
103     }
104 }
```

So far, there has only been alert if the move is invalid. We expect in the near future to put the piece back to its starting position in such events (and replace it with the alert event).

- Bishop

Recap the dictionary:

```

a8:1, b8:2, c8:3, d8:4, e8:5, f8:6, g8:7, h8:8,
a7:9, b7:10, c7:11, d7:12, e7:13, f7:14, g7:15, h7:16,
a6:17, b6:18, c6:19, d6:20, e6:21, f6:22, g6:23, h6:24,
a5:25, b5:26, c5:27, d5:28, e5:29, f5:30, g5:31, h5:32,
a4:33, b4:34, c4:35, d4:36, e4:37, f4:38, g4:39, h4:40,
a3:41, b3:42, c3:43, d3:44, e3:45, f3:46, g3:47, h3:48,
a2:49, b2:50, c2:51, d2:52, e2:53, f2:54, g2:55, h2:56,
a1:57, b1:58, c1:59, d1:60, e1:61, f1:62, g1:63, h1:64

```

If the bishop starts at **e4:37**, then it can only move in 2 directions: diagonal b1 – h7 (58 – 16), and diagonal h1 – a8 (64 – 1).

In the diagonal b1 – h7, notice each tile has a difference of 7 (58 – 51 – 44 – 37 – 30 – 23 – 16).

In the diagonal h1 – 18, notice each tile has a difference of 9 (64 – 55 – 46 – 37 – 28 – 19 – 10 – 1).

Thus, the difference between **startNumber1** and **endNumber1** either is divisible by 7 or 9.

```

107   if (pieceName1 == "whiteBishop" || pieceName1=="blackBishop")
108   {
109       if ((startNumber1 - endNumber1)%7==0 || (startNumber1 - endNumber1)%9 ==0)
110       {
111           alert("Valid move");
112       }
113       else
114       {
115           alert("Invalid bishop move");
116       }
117   }

```

- Queen

Queen moves like a Rook and a Bishop. If the queen passes Rook and Bishop's conditions, then it moves validly; otherwise it doesn't.

- Knight

```

a8:1, b8:2, c8:3, d8:4, e8:5, f8:6, g8:7, h8:8,
a7:9, b7:10, c7:11, d7:12, e7:13, f7:14, g7:15, h7:16,
a6:17, b6:18, c6:19, d6:20, e6:21, f6:22, g6:23, h6:24,
a5:25, b5:26, c5:27, d5:28, e5:29, f5:30, g5:31, h5:32,
a4:33, b4:34, c4:35, d4:36, e4:37, f4:38, g4:39, h4:40,
a3:41, b3:42, c3:43, d3:44, e3:45, f3:46, g3:47, h3:48,
a2:49, b2:50, c2:51, d2:52, e2:53, f2:54, g2:55, h2:56,
a1:57, b1:58, c1:59, d1:60, e1:61, f1:62, g1:63, h1:64

```

In the dictionary, if the knight starts at **e4:37**, then it can move to these tiles

- **d6:20**, **f6:22**. Notice $37 - 16 = 21$. $21 - 1 = 20$, and $21 + 1 = 22$
- **d2:52**, **f2:54**. Notice $37 + 16 = 53$. $53 - 1 = 52$, and $53 + 1 = 54$
- **g5:31**, **g3:47**. Notice $37 + 2 = 39$. Notice $39 - 8 = 31$, and $39 + 8 = 47$
- **c5:27**, **c3:43**. Notice $37 - 2 = 35$. Notice $35 - 8 = 27$, and $35 + 8 = 43$

Thus,

```

161 if (pieceName1 == "whiteKnight" || pieceName1 == "blackKnight")
162 {
163     if ((endNumber1==startNumber1-16-1) || (endNumber1 == startNumber1 - 16 + 1) || (endNumber1 == startNumber1 + 16 + 1) || (endNumber1==startNumber1+16-1))
164     {
165         alert("Valid Knight move");
166     }
167     else if ((endNumber1 == startNumber1-2+8) || (endNumber1 == startNumber1 - 2 - 8) || (endNumber1 == startNumber1 + 2 + 8) || (endNumber1==startNumber1+2-8))
168     {
169         alert("Valid Knight Move");
170     }
171     else
172     {
173         alert("Invalid Knight move");
174     }
175 }

```

- Kings

King can only move to adjacent spaces. Thus, if King is in **e4:37**, it can only move around

```

d5:28, e5:29, f5:30
d4:36, e4:37, f4:38
d3:44, e3:45, f3:46

```

The distance between **e4:37** and its adjacent spaces can be 1, 7, 8 or 9

```

if (pieceName1 == "whiteKing" || pieceName1 == "blackKing") //NOTE:
castle rule has not been established.
{
    if ((endNumber1 == startNumber1 - 1) || (endNumber1 == startNumber1
+ 1) || (endNumber1 == startNumber1 - 8) || (endNumber1 == startNumber1+ 8)
|| (endNumber1 == startNumber1 -8) || (endNumber1 == startNumber1 + 9) ||
(endNumber1 == startNumber1 - 9) || (endNumber1 == startNumber1 + 7) ||
(endNumber1 == startNumber1 -7))

    {
        alert("Valid King move");
    }
    else
    {
        alert("Invalid King Move");
    }
}

```

Note: the castling and checking rule have not been established.

- White pawns

Pawns don't move backward – they can only push forward. Thus, the rule for white pawn is different from that of black pawn, as they move in the opposite direction.

If white pawns are at their starting positions

```

a2:49, b2:50, c2:51, d2:52, e2:53, f2:54, g2:55, h2:56,

```

where their starting positions are between 49 and 56, then pawns can either jump start 2 spaces forward (2 rows, that is $2 \times 8 = 16$ units away from starting position) or move 1 space forward ($1 \times 8 = 8$ units away from starting position).

```

180     if (pieceName1 == "whitePawn") //rules for white pawn is different from black pawn's
181     //NOTE: No capture and promotion rule have been set up for white pawn
182     {
183         if (startNumber1 >= 49 && startNumber1 <= 56)
184         {
185             if ( (endNumber1 == startNumber1 - 8) || (endNumber1 == startNumber1 - 16) )
186             {
187                 alert("Valid Pawn move");
188             }
189             else
190             {
191                 alert("Invalid Pawn Jump");
192             }
193         }

```

If white pawns are not at their starting positions, then they can only move forward one space at a time.

```

195     else
196     {
197         if (endNumber1 == startNumber1 - 8 )
198         {
199             alert("valid pawn move");
200         }
201         else //COME BACK HERE and edit the code when capture rule has been setup for white pawn
202         {
203             alert("invalid pawn jump");
204         }
205     }
206 }

```

Note: promotion and capturing rule has not been set up.

- Black pawn

Black pawns move in the same fashion; they only move in the opposite direction.

```

208     if (pieceName1 == "blackPawn") //rules for white pawn is different from black pawn's
209     //NOTE: No capture and promotion rule have been set up for black pawn
210     {
211         if (startNumber1 >= 9 && startNumber1 <= 16)
212         {
213             if ( (endNumber1 == startNumber1 + 8) || (endNumber1 == startNumber1 + 16) )
214             {
215                 alert("Valid Pawn move");
216             }
217             else
218             {
219                 alert("Invalid Pawn Jump");
220             }
221         }
222         else
223         {
224             if (endNumber1 == startNumber1 + 8 )
225             {
226                 alert("valid pawn move");
227             }
228             else //COME BACK HERE and edit the code when capture rule has been setup for white pawn
229             {
230                 alert("invalid pawn jump");
231             }
232         }
233     }
234 }

```

Note: promotion and capturing rules have not been set up for pawns.

Now that the move is validated, we get back to the drop function

```
247 function drop(ev)
248 {
249     ev.preventDefault();
250     var data = ev.dataTransfer.getData("text");
251     ev.target.className = ev.target.className.split(" ")[0];
252     ev.target.className += " " + data;
253
254     pieceName1 = ev.target.className;
255
256     pieceName1= pieceName1.slice(5);
257     startPosition1 = document.getElementById("textInput3").value; //store Starting position to a variable startPosition1
258
259
260     document.getElementById("textInput1").value = pieceName1;
261     document.getElementById("textInput2").value = document.getElementById(ev.target.id).id;
262     dropPosition1 = document.getElementById("textInput2").value; //store drop position to dropPosition1
263
264     validMove(pieceName1, dropPosition1, startPosition1); //check if the move is valid
265
266
267     fn1(pieceName1, dropPosition1);
268 }
```

After the move is validated on line 264, on line 267, function

```
fn1(pieceName1, dropPosition1);
```

will send the data (piece Name and drop position) to the database.

```
270 function fn1(var1, var2)
271 {
272     $.ajax
273     ({
274         type:"post",
275         url:"/action1",
276         data: {textInput1: var1, textInput2: var2},
277         cache: true,
278
279         success: function fn2(var1, var2)
280         {
281             document.getElementById("p1").innerHTML = var1;
282             document.getElementById("p2").innerHTML = var2;
283         }
284     });
285     return false;
286 }
```

The data comes from the form with input form's name as `name="textInput1"` and `name="textInput2"` in chessboard.html

The data is sent to `url:"/action1"` for the server to handle. If success, paragraphs `p1` and `p2` will display the sent data.

This feature is far from perfect: we wish to see the browser constantly pulls data from the server in multiplayer games, whereas in this application, the server only response when there is a drop event in javascript

2. Server side

The server borrows heavily from the open source Flask documentation, although it has been heavily modified to fit our purpose.

- Log in

When the user logs in, he will send a GET and a POST request

```
90 @app.route('/', methods=['GET', 'POST'])
91 def login():
92     db = get_db()
93     if request.method == 'POST':
94         cur = db.execute('SELECT password FROM accounts WHERE username = ?', [request.form['username']])
95         cur_password = cur.fetchall()
96
97         #Checks to see if the username is used.
98         if not cur_password:
99             flash('Invalid username!')
100             return render_template('login.html')
101
102         #Makes sure that you typed in information.
103         elif not request.form['username'] or not request.form['password']:
104             flash('Enter the missing information')
105
106         #Checks password.
107         else:
108             #Selects the information from the SQL statement. Fetches the first item in the tuple.
109             #Turns the first item into a string.
110             cur1 = db.execute('SELECT password FROM accounts WHERE username = ?', [request.form['username']])
111             cur_password_str = str(cur1.fetchone()[0])
112
113             #Uses the check_password_hash from werkzeug.security which checks the hash of the password.
114             check = check_password_hash(cur_password_str, request.form['password'])
115
116             #Logs in if passwords match
117             if check:
118                 session['logged_in'] = True
119                 flash('You were logged in')
120                 return redirect(url_for('show_entries'))
121
122             #Otherwise, returns an invalid password statement and makes you attempt to log in again.
123             else:
124                 flash('Invalid password!')
125         return render_template('login.html')
```

The server will check the credentials in the form. If the form's username doesn't match any in the database or if it's blank, user will be requested to log in again.

After the user enters his username, the server will check the hash of the password (not the password itself) with method `check_password_hash` from library

```
15 from werkzeug.security import generate_password_hash, \
16     check_password_hash
```

- Create account

For new users, they will need to create new accounts before they can log in.


```

135 #This code takes the information you give to the site and stores it into the database.
136 @app.route('/addAccount', methods=['POST'])
137 def addAccount():
138     if request.method == 'POST':
139         db = get_db()
140         #Checks to see if the database has any information saved to the given username.
141         cur = db.execute('SELECT username FROM accounts WHERE username = ?', [request.form['username']])
142         check_existing_username = cur.fetchall()
143
144         #Creates a password using a hash
145         new_account_password = User(request.form['username'], request.form['password'])
146         hash = new_account_password.pw_hash
147
148         #If you don't type something into a box, then you get returned to the create account screen.
149         if not request.form['username'] or not request.form['password'] :
150             flash('Enter the missing information!')
151             return redirect(url_for('createAccount'))
152
153         #If the username doesn't exist yet, it saves the username and a hash of the password input
154         #by the user.
155         elif not check_existing_username:
156             error = None
157             db.execute('INSERT INTO accounts (username, password) VALUES (?, ?)', [request.form['username'], hash])
158             db.commit()
159             flash('New account was successfully created. Please log in!')
160             return redirect(url_for('login'))
161             return render_template('login.html', error=error)
162
163         #Prevents someone from making an account with a similar username.
164         else:
165             flash('Username already exists! Please log in.')
166             return redirect(url_for('createAccount'))

```

First, the server will check if the registering username has been taken in the database or not. If username has been reserved, user will be notified, and they have to choose a new username.

After user creates a new account successfully (in which the server will hash the password, and store the credentials in the database), he will be redirected to the login site, and he can log in himself.

- Send and receive data from moves in the game

For each move, user will send a POST request to the server

```

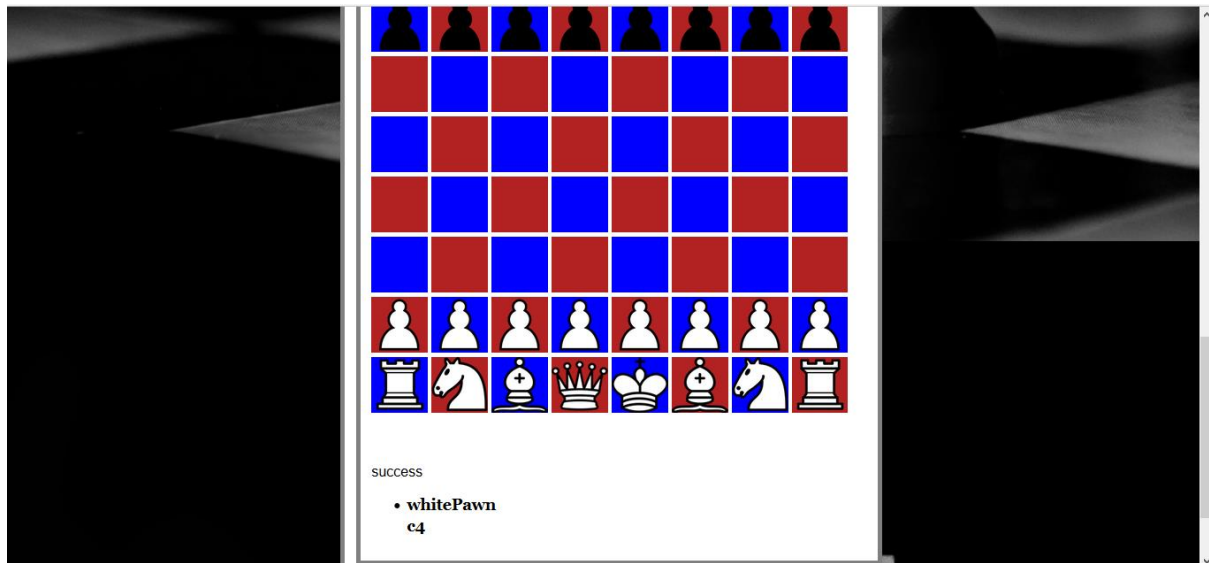
176 @app.route('/action1', methods=['POST'])
177 def action1():
178     db1 = get_db()
179     db1.execute('Insert into action1 (input1, input2) values (?, ?)', [request.form['textInput1'], request.form['textInput2']])
180
181
182     cur = db1.execute('SELECT input1, input2 FROM action1 order by id desc')
183     action1 = cur.fetchall()
184     #db`
185     flash('Javascript sends successfully') #Happens when loads new page.
186     flash(request.form['textInput1'])
187     flash(request.form['textInput2'])
188     return render_template('chessBoard.html', action1=action1)

```

Recap: `request.form['textInput1']` and `request.form['textInput2']` are the chess piece name and dropping position.

`input1, input2` are the columns of the SQL table in the database.

There is still a bug for this feature: a second chessboard appears after the first data is sent to the database through the drop event.



Notice the scrollbar: this is the second chessboard, which does not affect the game overall, but we would prefer to get rid of it.

- Database structure

So far there are two major tables: one for storing users' credentials, and one for storing moves in the game.

```

1  drop table if exists accounts;
2  create table accounts (
3      id integer primary key autoincrement,
4      username text not null UNIQUE,
5      password text not null
6  );
7
8  drop table if exists action1;
9  create table action1
10 (
11     id integer primary key autoincrement,
12     input1 text,
13     input2 text
14 );

```

III. Problems/Bugs

The application has a huge foundation so far, but we are facing some challenges:

- While the code knows when the pieces make illegal move, we need to put them back into the starting position. With the chess piece and the starting position in hand, we should know how to set up the chessboard from JavaScript (and replace the illegal move alert with that).

Furthermore, pieces can be captured by friendly pieces. Also, king castling, pawn capturing and pawn promotion haven't been implemented yet.

- The empty tiles are draggable, and they can capture the pieces. We should temporarily disable the dragging feature of tiles when they are empty.

- Kings are not checked. We should set up how hostile pieces' movement range can affect the Kings.
- The game's client side should constantly pull request and data from the server to keep the game up-to-date automatically.

Further developmental plan:

- We wish to see the game supports online multiplayer feature. There should be many more features: players can chat, resign, claim win on opponent's disconnected Internet, report trolls, and so on.
- We should work more on graphical side; so far, the interface seems a bit too simple compared to those of other online games.

Developers should dig deeper about how to implement the game's rule using JavaScript (and replace them with the current alerts). For server side, developers can look for Python instruction in Flask documentation. They can have trouble making the game become multiplayer, as instruction for setting up a live server for multiple computers can be a bit hard to find.