Fiddling with Electron Fiddle

while making a clone of Card Crawl

In this workshop, we're going to play with Electron Fiddle by making a game in it.

What's Electron Fiddle? It's a framework that allows you to learn electron framework without the setup phase and devops work.

Instead of installation of electron, node, packages and communication setup process, you start up Electron Fiddle and start coding. So it's a more simple way into the framework used to create Slack, Discord and Figma.

The only thing you need to start is to download Electron Fiddle for your platform from here: https://github.com/electron/fiddle/releases/

Please do this and make sure that you installed it and started it. Electron Fiddle will take some time to download Electron, so while it's doing that let's talk about our end game. Our Game.

I think that the most fun way to learn a new language, framework, setup or patter is to make a game using it. However, because "inventing" a well-balanced and exciting game is a huge task. It requires knowledge of game design, game systems, sometimes mathematical simulations, so at the end, it's like writing War and Peace to learn how to spell a couple of Russian words.

Because of this, I create working clones of the games that I play. I made a clone of Card Crawl to learn Electron. A clone of Zombie Dice to learn React. So now we're going to make a clone of Card crawl to learn Electron Fiddle.

What is Card Crawl

Card Crawl is one of the best games ever. It's not just me saying this. It won several "Game Of The Year" rewards. You can check it out on its website: http://www.cardcrawl.com/ The game has a perfect balance of simplicity and challenge, it's effortless to pick up and very hard to put down.

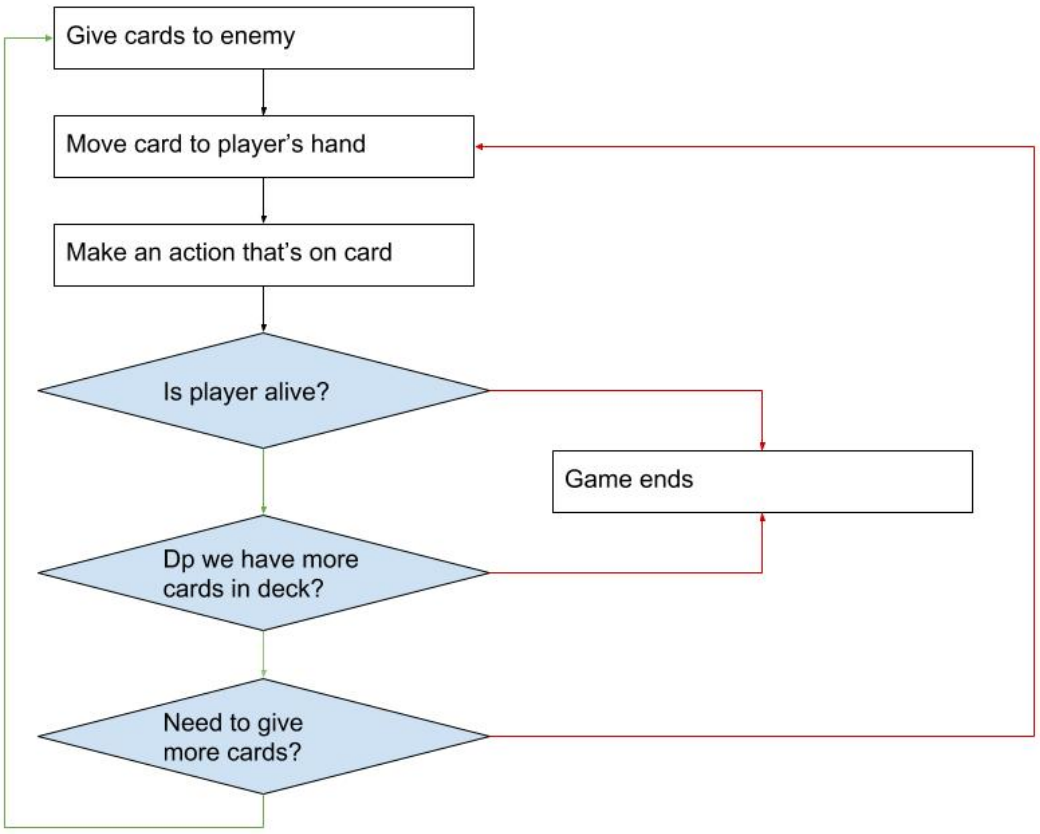
For us to make a clone of it, we need to understand it's mechanics, so if you have time - watch this video about the game to understand the base mechanics. https://www.youtube.com/watch?v=5bYCbSvgcmA

What elements are we going to use?

1. Cards. Our game has a deck of cards that contains an arbitrary number of cards, let's say 15. Cards have 4 types:
    - Enemy - can attack the hero. It has health and attack power. We'll start with 3 for both.
    - Shield - an item that can be put in hand or the pocket, but can be only used if in hand. It has protection power 3. Can be attacked by enemies.
    - Sword - an item that can be put in hand or the pocket, but can be only used if in hand. It has attack power 3. Can attack enemies.
    - Heal - healing potion. An item that can be put in hand or the pocket. In hand, it'a used immediately. It has healing power 3. Heals the player.
2. Table. Our game is represented by a table with 2 rows.
    - 1st row is enemy's row. It's filled with cards from the deck. It is refiled if there are 1 or fewer cards in the first row and there are no actions in 15 seconds.
    - 2nd row is the player's row. It contains 2 places for hands, the hero avatar and a pocket. In each turn, the player has to reduce the enemy's row until there are 1 or fewer cards left and continued to do so until there are no cards left in the deck.

Now that we have our rules, we can form our game loop.

The game loop is the sequence of events that repeats until the winner is determined.
Here's our game loop:



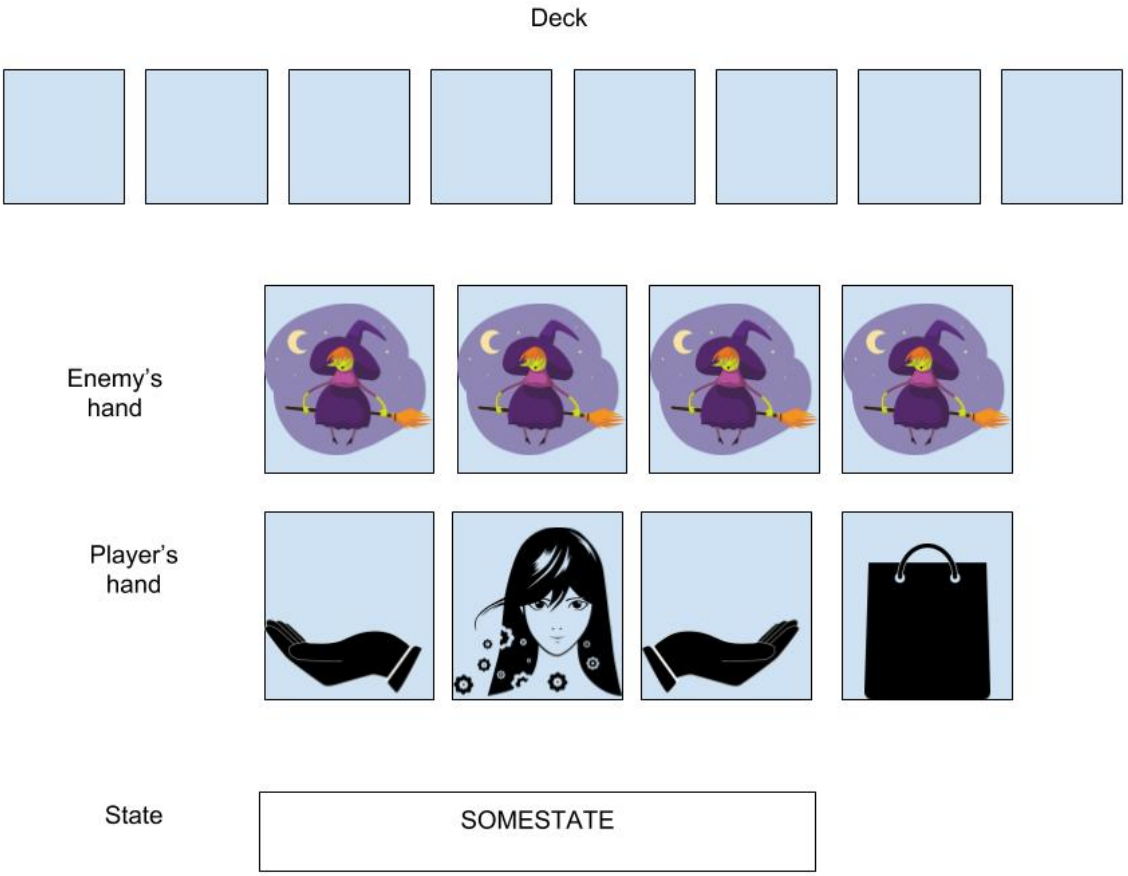
Based on gameLoop we can start thinking about the data that we need to keep.

We need to have a deck with cards. We create cards individually though for now, they have the same data. Why? Because later you may want to experiment and create small, medium and big ghosts with different points of health and power and I want to give you a setup that allows it.

Then we'll need to have a player's hand where the second object is his or her hero. Also, the 4th one is the pocket that has some additional conditions.

Enemy's hand where we are going to put new cards.

We also need a variable to keep the game state.



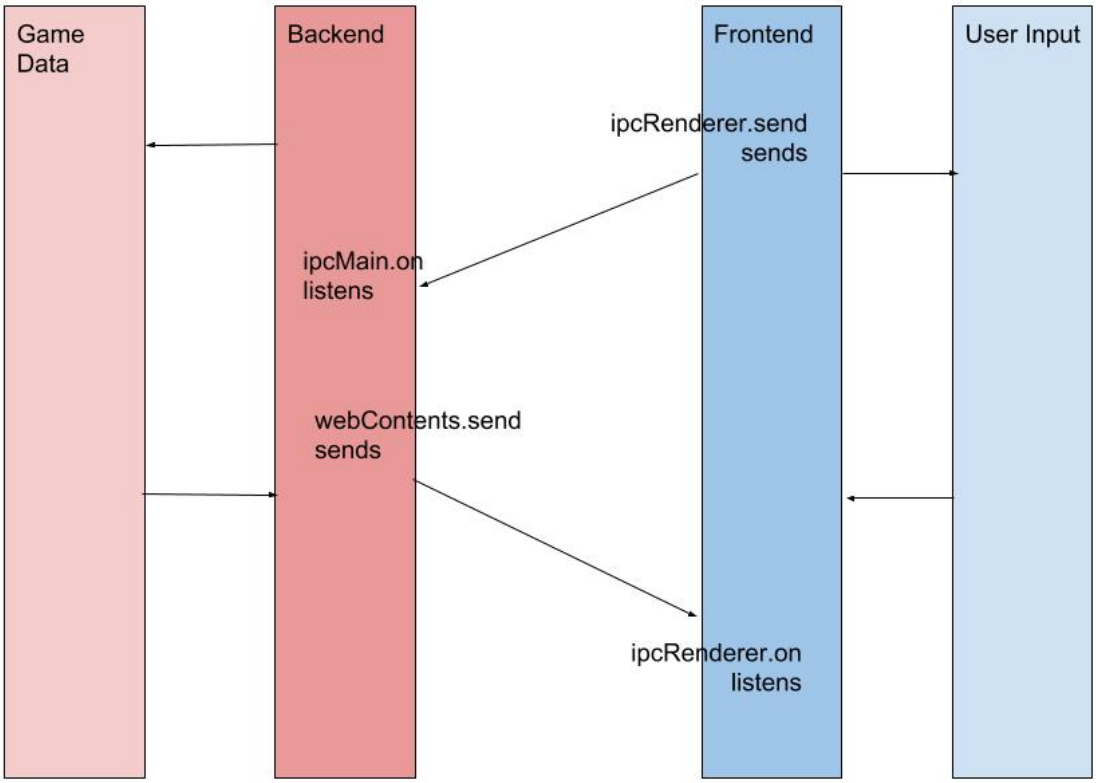
Here's the visual representation of our GSO. We defined out data now we can think about our architecture.

Electron and Electron Fiddle by proxy, give you a perfect structure for your architecture.
Electron gives you 2 processes. Main process that you can think of as your backend and Render process that you can think of as your frontend.

The typical structure would be like this:
Your backend communicates with your data, changes it as required and sends it to the frontend.
Your frontend takes your user's input and sends it to your backend.

This encapsulation allows you to have control over the access and you'll never have to deal with doubled data or conflicting data. Your frontend never changes data. Your backend never has access to the user.

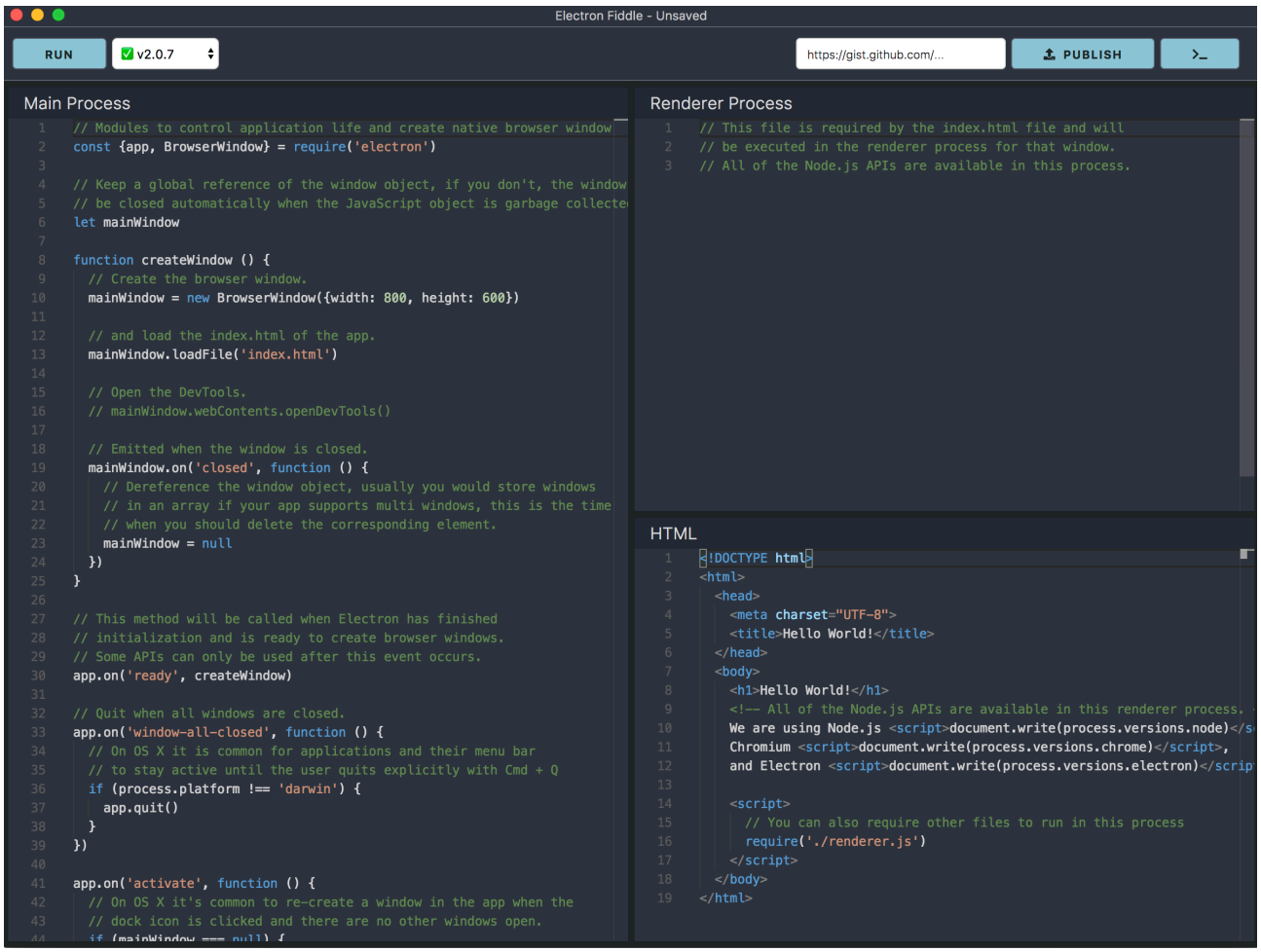
With that structure in mind, we can focus on the communication between frontend (Renderer) and backend (Main).



There are 2 methods for setting up communication.
The backend has a method ipcMain.on - it listens for the messages. Also at the start backend creates a browser window for you and then on this window, it can send objects to the frontend.
So we're going to use ipcMain.on to listen and mainWindow.webContents.send to answer on the backend.
On the frontend, we have ipcRenderer.on that listens for the messages from the backend. Also, ipcRenderer.send that sends the messages to the backend.

Hope your electron fiddle is up and running now. Let's start it and look at what we have.

When you open your Fiddle, you see 3 files. Main process (main.js) Renderer process (render.js) and HTML.
We're going to ignore HTML for now as we're focusing on setting up the communication between our main and renderer.
Run it just to see what happens.
When you Run it, you'll see you backend logs and your window. To see your frontend logs you can either open logs from the top menu (View -> Toggle Developer Tool) or make it active by default. We'll do this in a second.



Fiddling -> Backend communication

Return to the Fiddle.
Your **Main.js** has most of the functionality you need from the start.
It has a variable for **mainWindow** and then it uses this window to make all sorts of magical things.
However, we need **ipcMain** to listen for the messages.
So first let us import it:

```
const {app, ipcMain, BrowserWindow} = require('electron')
```

Read more on requiring node.js modules here: https://www.w3schools.com/nodejs/nodejs_modules.asp

Also, find the line with a comment Open the DevTools and uncomment the next line. This way you'll see your logs on frontend too. After you're done you can comment out this line again.

```
// Open the DevTools.
 mainWindow.webContents.openDevTools()
```

Run your app again - you'll see the dev tools on frontend are also open. I prefer to have them at the bottom of the screen, but it's up to you.

Time to plan our communication!

Let's think of the responsibilities of frontend and backend on our project.
Backend creates the deck manipulates the game data.
Frontend gets the user input and displays the game.

So simplistically we're going to have a dialogue

FRONTEND: “Dear backend, here’s the identification of the card that user selected. Please parse it, change data accordingly and give me back the GSO in its new state”.

- BACKEND: “Tots, catch this GSO in this state”.

So Frontend is going to show the user the current state of the game and available options.
The backend will change the object and return it.

Let’s talk about messages then.
Our backend will get the MSG message that may have or not have data in it.
Our frontend will get GSO that will always give GSO object in it.

Time to code it.

At the end of your **main.js** file on **ipcMain** accept the object with the string identifier MSG, event and arguments.

```
ipcMain.on('MSG', (event, arg) => {
    // Save the message to analyse it
    // Send back the answer with the gso
});
```

Don’t be afraid of the strange syntax. It means that **ipcMain.on** accepts a function and an argument. You can read more about this syntax here: https://www.jstips.co/en/javascript/fat-arrow-functions/
To work with this message we need to save it into the variable and then send the answer back to the frontend.

Let’s switch to the frontend.
render.js is empty for now, so we need to import **ipcRenderer** into it.

```
const { ipcRenderer } = require('electron')
```

And then listen for messages on it

```
ipcRenderer.on("GSO", (event, arg) => {
    // Render the game
});
```

You won’t see any changes to your game if you run it. That’s because it needs more magic. Also, some existing data wouldn’t hurt.

Before we write more code, we need to go deeper into architecture.
What should the backend do when it gets the message?
// Look at the message to know what state GSO it wants should be.
// Manipulate GSO to get it into this state
// Send it back

What should the frontend do?
// Send the initial message so we can start the game
// Listen for the player’s input
// Display the game in its current state
// Send messages to the backend

Let’s work on our **main.js** to finish the communication.

When the message received, we need to parse it in a function that returns us the GSO in the right state and sends it back. To identify the message we need to give it an ID and based in this ID decide what to do when we get this message.

So declare the function **msgReceived(arg)** that gets arguments from **ipcMain.on** and returns GSO. Log your message ID that is a part of the argument received by your function.

If you need more information on functions and arguments, you can read more on it here: https://www.w3schools.com/js/js_function_parameters.asp

```
// Parse the message to call a relevant function
function msgReceived(arg){
  // Console log the id you have inside your arg
  // Return your gso object
}
```

Example:

```
// Parse the message to call a relevant function
function msgReceived(arg){
  console.log("Received message "+arg.id);
  return gso;
}
```

Read more on using **console.log** here: https://www.w3schools.com/js/js_output.asp

Fiddling -> Backend Data

Of course, we don't have GSO yet, so we need to define it as a global variable. We already discussed the structure.
As long as we're on it let's define our deck. It contains objects that we discussed previously.

You can read more on javascript objects here: https://www.w3schools.com/js/js_objects.asp
And on javascript arrays here: https://www.w3schools.com/js/js_arrays.asp

Create GSO with all the data - the place for the deck, the place for enemy's 4 cards, player's 4 cards and state "PLAY".

Example:

```
let gso = {
  deck: [],
  enemyHand: [null, null, null, null],
  playerHand: [null, null, null, null],
  state: "PLAY"
}
```

And here are our cards:

```
let cardsArray = [
  { name: "sword", power: 3, type: "attack" },
  { name: "sword", power: 3, type: "attack" },
```

```
  { name: "sword", power: 3, type: "attack" },
  { name: "shield", power: 3, type: "shield" },
  { name: "shield", power: 3, type: "shield" },
  { name: "shield", power: 3, type: "shield" },
  { name: "heal", power: 3, type: "heal" },
  { name: "heal", power: 3, type: "heal" },
  { name: "heal", power: 3, type: "heal" },
  { name: "troll", health: 3, power: 3, type: "enemy" },
  { name: "troll", health: 3, power: 3, type: "enemy" },
  { name: "troll", health: 3, power: 3, type: "enemy" },
  { name: "ghost", health: 3, power: 3, type: "enemy" },
  { name: "ghost", health: 3, power: 3, type: "enemy" },
  { name: "ghost", health: 3, power: 3, type: "enemy" },
  { name: "witch", health: 3, power: 3, type: "enemy" },
  { name: "witch", health: 3, power: 3, type: "enemy" },
  { name: "witch", health: 3, power: 3, type: "enemy" },
]
```

As we discussed earlier, backend uses the window object to send the messages back. The global window object is already declared in **main.js**. Now call the function **msgReceived**, save the GSO it returns into the new variable and send it back with this global mainWindow through:

```
mainWindow.webContents.send
```

Example:

```
ipcMain.on('MSG', (event, arg) => {
    // Save the message to analyse it
    let received = msgReceived(arg);
    // Send back the answer with the object
    mainWindow.webContents.send("GSO", received);
});
```

If you run your project, you won't see the console.log though. Because we never send the message on the frontend. Let's fix that.

Fiddling -> Frontend communication

Get to **renderer.js** and let's write function **startGame()** that is called when the file loads.

Just declare it and call in right away.

```
function startGame(){
}
startGame();
```

To make sure we start listening for the GSO the moment game loads, move your **ipcRenderer.on** into **startGame()** function.

We will also need a **sendMessage()** function.

Declare it as a function that gets some object (our player input will go into this object) and then uses **ipcRenderer.send** to send the MSG that backend awaits.

Now we can call **sendMessage** function on the game start with id “START” and if all goes well - backend will print this message in console log.

Example:

```
function sendMessage(obj){
    //console.log("Sending a message with", obj);
    ipcRenderer.send('MSG', obj);
}
function startGame(){
    ipcRenderer.on("GSO", (event, arg) => {
        // Render the game
    });
    sendMessage({id: "START"});
}
startGame();
```

Now in your backend log, you should see the line “Received message START”.
To see something on the frontend we need to console.log the received object.

Fiddling -> Rendering the game

Let’s switch to **renderer.js** and write **renderGame()** function that receives arguments from ipcRender and then, for now, will only render GSO it receives.

Declare function **renderGame** that receives newGSO then saves it as a global variable (you’ll need to declare it under your requirement line) and prints it out.
You need
// Declare a global variable gso
// Inside **renderGame** function save the newGso this function receives into this global variable.

What does it do?
Our frontend listens for the GSO from the backend. And when it gets it in **ipcRenderer.on** it calls the function **renderGame()** giving it this GSO that we got from the backend.
renderGame() will save this new GSO as a global variable so all of our functions can read it (they will never change it though)

Example:

```
const { ipcRenderer } = require('electron')
let gso;
function sendMessage(obj){
    //console.log("Sending a message with", obj);
    ipcRenderer.send('MSG', obj);
}
function startGame(){
```

```
    ipcRenderer.on("GSO", (event, arg) => {
        // Render the game
        renderGame(arg);
    });
    sendMessage({id: "START"});
}
function renderGame(newGso){
    gso = newGso;
    console.log(gso);
}
startGame();
```

Run it. You should see your Received message in your backend logs. And an object in your frontend logs. If you open this object, you will see that this is the initial GSO that you've just created on the backend.

Congrats! Your frontend and backend are communicating!

Fiddling -> States

Let's look at our game and try to figure out the possible states for the game.
From backend point of view, the message that it can get from frontend can be one of two.

1. JUST START THE GAME I DON'T HAVE ANYTHING FOR YOU
2. LOOK AT THIS AMAZING USER INPUT I HAVE, DO SOMETHING WITH IT.

As those messages are a bit too long, let's simplify them to START and ACT
If our backend gets the message START it needs to initialise the game.
If our backend gets message ACT it needs to change the game accordingly.

Add a switch statement to the **msgReceived()** function. Based on message.id we'll either
START -> call **createGso()** function that will fill our GSO with the initial information (remember, the gso we created has empty arrays everywhere.
ACT -> call **actParsed()** function that will change our GSO based on the input we got from the user.
Add a default case that would make our backend scream "OMG I DON'T KNOW THIS MESSAGE!"

You can read more on switch statements here https://www.w3schools.com/js/js_switch.asp

```
// Parse the message to call a relevant function
function msgReceived(arg){
  // Console log the message id from the arg
  // Use switch statement with arg.id to call
  // createGso function if you get "START"
  // actParsed function if you get "ACT"
  // in default case console log that the message was unknown
}
```

Example:

```
// Parse the message to call a relevant function
function msgReceived(arg){
  console.log("Received message "+arg.id);
  switch(arg.id){
    case "START":
      return createGso();
    case "ACT":
      return actParsed();
    default:
      console.log("Unknown message "+ arg);
      return gso;
  }
}
```

Declare both functions and focus on **createGso()** for now. What should it do?
// Create a new deck using the **cardsArray** sorted randomly
// Save it as a deck into **GSO**
// Set our **heroCard** as a second card in the player's hand
// Give cards

To avoid magic numbers, declare a global variable **heroIndex** and another global variable **heroCard** so you can tweak it later if needed, without digging through the code.

```
// Global Game State Object and cards that we're going to use in
the game.
let heroIndex = 1;
let heroCard = { name: "hero", health: 10, type: "hero" };
```

Now using these 2 variables you can write your **createGso()** function. The function that gives cards we'll write in a second.

```
// The initial GSP creation
function createGso(){
  // randomise the cardsArray and save result into the new
variable
  // save this new variable as your deck into gso
  // make sure that in player's hand the card under heroIndex is h
     hero card.
  // call the new giveCards() function
  // return your gso
}
```

You can find randomising functions on google or use my favourite:

```
let newArray = oldArray.sort(()=> Math.random() -0.5);
```

Example:

```
// The initial GSP creation
function createGso(){
  let newDeck = cardsArray.sort(() => Math.random() - 0.5);
  gso.deck = newDeck;
  gso.playerHand[heroIndex] = heroCard;
  giveCards();
  return gso;
}
```

giveCards() is an interesting function. We may start with just using array.pop 4 times to fill the initial 4 spaces in enemy's hand. However, we'll need to rewrite it so we can use it to give cards not just when the game starts, but each time we need to give cards to the player. So let's do it properly the first time.

Read more on array.pop here https://www.w3schools.com/jsref/jsref_pop.asp

Now declare **giveCards()** function.
It has access to GSO because GSO is global. So it can go into **gso.enemyHand**, check how many empty spaces are there and if there are 3 or more empty spaces - fill them with the cards from **gso.deck** using **array.pop()** function.

```
// Function to give cards if the number of empty cards is >= 3
function giveCards(){
  // Check the number of empty cards in enemy hand
  // if there is 1 or less cards, take gso.enemyHand.length and
  // using .pop() function pop cards into enemy hand
}
```

You may want to use filter function, read more about it here: https://www.w3schools.com/jsref/jsref_filter.asp

Here's the one example of how it can be done:

```
function giveCards(){
  // Check the number of empty cards in enemy hand
  const cardsEmpty = gso.enemyHand.filter(function(e){
    return !e;
  }).length;
  if(cardsEmpty < 3) {
    return;
  }
  for(var i=0; i<gso.enemyHand.length; i++){
    if(!gso.enemyHand[i]){
      gso.enemyHand[i] = gso.deck.pop()
    }
```

```
    }
}
```

Run it. Check your frontend log - you should see not the empty GSO that was there before, but GSO with cards in it. I think it's time to work on showing those cards, don't you?

Fiddling -> Render cards

Switch to the frontend. Let's discuss the states of GSO it can get.
We're already dealing with PLAY - that's the initial state of GSO we sent from the backend.
What else can there be? What other states do we need to draw?
PLAY -> here's the GSO please, draw it
WON -> Player won.
LOSE -> Player lost.

Pretty simple.
Add a switch statement to **renderGame()** function. It checks thew **gso.state** and based on it either **renderTable()** or alert us that player won or lost.

Read more on using window.alert here: https://www.w3schools.com/js/js_output.asp

Don't forget **break;** as we are not returning anything.

Example:

```
function renderGame(newGso){
    gso = newGso;
    console.log(gso);
    switch(gso.state){
        case "PLAY":
            renderTable();
            console.log("Playing")
            break;
        case "LOST":
            window.alert("You lost!");
            break;
        case "WON":
            window.alert("You won!");
            break;
    }
}
```

Declare your **renderTable()** function and run the game. There won't be much change. You will see "Playing" in your log because the gso has "PLAY" state initially. There's much work to do!

Fiddling -> HTML and CSS

First, let's talk HTML and CSS.
Because Electron Fiddle is simple, though very effective tool, we're going to write our css in our html file.

Let's switch to **Index.html** and remove the data it has inside body tag. Don't remove <script>, because it connects your frontend javascript to your html.

If you cleaned it up correctly, you will see an empty window when you run your game.

Let's start filling it.
First, let's look into <head></head> part.
Switch title to <title>CARD CRAWL CLONE</title>
Add an empty <style></style> tag for later use.

That's how all of your HTML should look:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
     <title>CARD CRAWL CLONE</title>
     <style></style>
  </head>
  <body>
    <script>
      // You can also require other files to run in this process
      require('./renderer.js')
    </script>
  </body>
</html>
```

Now let's do something not recommended by 90% of frontend developers. Let's use a table!

I mean, I understand that the table is not the best tag to use in layouts, but we have 2 rows and 4 columns in each if it's not the table - what is?

Inside your body, before the script, create **div** element with **id=container**.
Inside of it declare the **table** element.
The first row has **id="enemyHand"** and 4 cells with **data-index** elements from 0 to 4. We will need them to know coordinates of cards that player manipulates. They are super important because by our rules frontend can't change the data. So instead of sending the card to the backend it will send its coordinates - the id of the row that the card is in (they are the same as keys in out GSO, smart, right?) and the index. This way we can easy find the card user selected in GSO on backend.

The second row has **id="playerHand"** and also 4 cells with the same data-index. Here's the result:

```
  <body>
    <h1>Card Crawl Clone!</h1>
    <div id="container">
      <table>
        <tr id="enemyHand">
          <td data-index="0"></td>
```

```
          <td data-index="1"></td>
          <td data-index="2"></td>
          <td data-index="3"></td>
        </tr>
        <tr id="playerHand">
          <td data-index="0"></td>
          <td data-index="1"></td>
          <td data-index="2"></td>
          <td data-index="3"></td>
        </tr>
      </table>
    </div>
    <script>
      // You can also require other files to run in this process
      require('./renderer.js')
    </script>
  </body>
```

CSS TIME!
To start writing css we need to discuss our design for a bit.
What do we need to indicate for the player?
We need to indicate that the cell is:
empty
filled - there's a card in it
target - you can put a card in it
hover - you're about to put a card in it.

So style your elements as you see fit, make sure initially your tds are grey.
Then add 3 css classes:
.filled - background-color changes to blue
.target - background-color changes to red
.hover - background-coloe changes to green

Here's the example of the simplest css ever within <style> tag we created earlier.

```
<style>
       body {
         text-align: center;
       }
       table {
         margin: 0 auto;
       }
       td {
         width: 100px;
         height: 150px;
         border: 1, solid, black;
```

```
        background-color: grey;
      }
      .filled {
        background-color: blue;
      }
      .target {
        background-color: red;
      }
      .hover {
        background-color: green;
      }
    </style>
```

Run your game. You'll see 8 card places and the name. Good enough to start, now let's write some interactions.

Fiddling -> Cards

Return to our **renderTable()** function.
It should call a function to **renderRow** and give it **the hand** to render and the **id** of the element to render it into.

```
function renderTable(){
    // Call renderRow function, give it enemyHand from gso and
     "enemyHand" id.
    // Call renderRow function, give it playerHand from gso and
     "playerHand" id.
}
```

Example:

```
function renderTable(){
    renderRow(gso.enemyHand, "enemyHand");
    renderRow(gso.playerHand, "playerHand");
}
```

Let's write **renderRow(hand, selector)** function. What should it do?
// Get the children of this HTML selector
// Go through the hand and render cells for this selector

You can get the children of the selector via **document.getElementById(selector).children** and then iterate through the hand, rendering cells.
We need a function to render a single cell, and we need to call it for the cards we have in hand.

To get the child of the element, you can use **getElementById** and read about it here https://www.w3schools.com/jsref/met_document_getelementbyid.asp
Then you can get the **children of this element**, read about it here: https://www.w3schools.com/jsref/prop_element_children.asp

```
function renderRow(hand, selector){
    // get the children of the selector and save into the variable
    // Iterate through the hand and cell renderCell giving it the
     element in hand, the cell and selector
}
```

Here's the implementation example:

```
function renderRow(hand, selector){
    const cells = document.getElementById(selector).children;
    for(let i=0; i<hand.length; i++){
        renderCell(hand[i], cells[i], selector);
    }
}
```

The next step is **renderCell(card, cell, row)** function. It gets the hand item (card to render), the cell (as we need to know coordinates) and the initial selector that we can later use to see if it was player's card or enemy's card.

What should it do?

```
function renderCell(card, cell, row){
    // Clear the cell class and inner HTML - don't need the
old info
    // If there's a card, fill it's info.
    // Put card name into description
    // If it's a hero card - display name and health
    // If it's enemy card - display name, power and health
    // If it's shield or sword or heal - display name and power
    // Add a "filled" class so we know the cell is taken
}
```

You can clear the class using **.className** property, read about it here: https://www.w3schools.com/jsref/prop_html_classname.asp
To add class use **classList.add(className)**. https://www.w3schools.com/jsref/prop_element_classlist.asp
Access to the html inside the element is granted via **innerHTML** attribute. https://www.w3schools.com/jsref/prop_html_innerhtml.asp

Here's the example:

```
function renderCell(card, cell, row){
    cell.className = "";
    cell.innerHTML = "";
    if(card){
        let description = "Card:"+card.name+"</br>";
        if(card.type=="hero"){
            description = description+"Health: "+card.health;
```

```
        } else if(card.type=="enemy"){
            description = description+"Attack: “+card.power+
          "</br>Health: "+card.health;
        } else {
            description = description+"Power: "+card.power;
        }
        cell.classList.add("filled");
        cell.innerHTML = description;
    }

}
```

Run your game! You’ll see the cards and their data. However, they are not interactive yet. Before we can make them interactive, we need to have a conversation about event listeners.

Fiddling -> Events

The central interactive element of our game is dragging cards here and there. It’s all possible via the draggable attribute.
Our game is pretty small. So we can listen for the **drag events** globally - in out **startGame()** function on our **#container** element.

if you need to, read about event listeners here: https://www.w3schools.com/jsref/met_document_addeventlistener.asp

So let’s start with setting our event listeners.
What do we need to listen to?
“dragstart" to call **onDragStart(event)** function that saves the card that we drag around into a global variable so we can send it to backend later.

“dragend" to call **onDragEnd(event)** function that clears the selected card and redraws the table.

“dragover" to call **onDragOver(event)** function that takes our .target card and adds .hover to it so the player knows she can put the card down in this spot.

“dragleave" to call **onDragLeave(event)** function that removes the .hover class because the user decided not to put the card down here.

“drop" to call **onDrop(event)** function that sends the message to the backend - here’s the active card, here’s the target card, do some math, backend.

So in your **startGame** find the container element in your document and add all of these **eventListeners** to it.

Here’s the example of how it can look like now:

```
function startGame(){
    ipcRenderer.on("GSO", (event, arg) => {
        renderGame(arg);
    });
    let container = document.getElementById("container");
    container.addEventListener("dragstart", onDragStart);
```

```
    container.addEventListener("dragend", onDragEnd);
    container.addEventListener("dragover", onDragOver);
    container.addEventListener("dragleave", onDragLeave);
    container.addEventListener("drop", onDrop);
    sendMessage({id: "START"});
}
```

Declare all these 5 functions that get (event) parameter and, to start, console.log the function name and event. Here's the example of onDragStart:

```
function onDragStart(event){
    console.log("Drag started ", event);
}
```

If you run your game now, nothing will change. Draggable objects require their draggable attribute to be set to true. And that's where our game becomes quite complicated.

Fiddling -> Render Cells

Let's start simple.
First, when you **renderCell** and clean up the **className** and **innerHTML** of the cell, also set it's **draggable** to false.

```
    cell.className = "";
    cell.innerHTML = "";
    cell.draggable = false;
```

Then add a condition that calls the new **isDropTarget(card, row)** function for this cell and row and based on the result adds a "target" class to the cell. This way we are going to make sure that the user can drop their card only on the allowed targets.

Inside your **if(card)** condition also set the draggable to **card.type != "hero"** which means that if our card type is the hero we can't drag it around.

Here's how the changed function should look like:

```
function renderCell(card, cell, row){
    cell.className = "";
    cell.innerHTML = "";
    cell.draggable = false;
    if(isDropTarget(card, row)){
        cell.classList.add("target");
    }
    if(card){
        let description = "Card:"+card.name+"</br>";
        if(card.type=="hero"){
            description = description+"Health: "+card.health;
```

```
        } else if(card.type=="enemy"){
            description = description+"Attack: "+card.power+"</
br>Health: "+card.health;
        } else {
            description = description+"Power: "+card.power;
        }
        cell.classList.add("filled");
        cell.innerHTML = description;
        cell.draggable = card.type != "hero";
    }

}
```

Declare **isDropTarget(card, row)** function and make it return true all the time. We will fix it later.

```
function isDropTarget(targetCard, targetRow){
    return true;
}
```

Run the game. All of your cards are red now because they are all viable drop targets. But you should also be able to drag them around!

Let's make it a bit crazy and make sure our isDropTarget returns true only if **targetRow=="playerHand"**

```
function isDropTarget(targetCard, targetRow){
    return targetRow=="playerHand";
}
```

After this, all of the enemy's cards should be draggable, and all of the player's cards should be viable targets. Leave it at this for now. We need to work on our drag functions before we can implement the complicated game logic.

Fiddling -> Drag Events

First, let's talk about what the hell we're doing.

The goal of all of this is to send the message to the backend. The message should contain 2 cards. The card that acts. And the target of the action.

It doesn't need to send card objects. Remember, frontend doesn't do data. It works with the player's input.

So we don't send the cards to the backend. Instead, we send coordinates that have row and index.
The communication will look as follows;

- FRONTEND: Here's the message with id ACT. The card that acted is in the row playerHand and has index 2. The card that was a target of the action is in the row enemyHand and has index 1.

- BACKEND: I am looking into my GSO. Found this player's card. Found this enemy's card. Now I'll do my math, save the surviving card and give you back the GSO that changed after all this crazy action.

So for the system to function correctly, we need to translate the selected card and the target card into the address that frontend will attach to the message.

So let's write a function **findSelectedCard(target)** that gets the **event.tagret** and transforms it into the address.

This may sound complicated, so let's simplify it more.

You have your console.log in onDragStart function. So run your game, start the drag and find the console log.

It has a DragEvent and if you scroll through the data you'll find:
target - it is your td
dataset - it is your card index

For every card that is the target of the event, you can find the **parent id** via **parentElement.id** and the **index** via **dataset.index**. Save them into the object and return it.

```
function findSelectedCard(target){
    return { row: target.parentElement.id, index: target.dataset.index };
}
```

Now declare a global variable **selectedCard** that you will save the card player had selected so it can be accessed by all of our event functions. Set it to **null.**

The only thing that's left is to plan what our events will do with it.

When **drag starts**, we need to save the card that player drags around and render the table.
When **drag ends**, we need to clear the selected card by setting it to null and render the table.
When **drag is over** the card, we need to check if the card has the class .target and add class .hover to it. Also this function requires a call to event.preventDefault();
When **drag leaves**, we need to remove the class hover from the target.
And, **on drop**, we need to call event.preventDefault(), then send the message to the backend. In this message the ID is ACT, the first card is the selected card and the second card is the card we are on top of (target)
Then we need to clear the selected Card again.

Try to write all these functions on your own. They all are 4 lines tops!

Then check with my result:

```
// When we start dragging - save the card that we use and redraw
the table
function onDragStart(event){
    //console.log("Drag started ", event);
    selectedCard = findSelectedCard(event.target);
    renderTable();
}

// When we finish dragging the card around, remove the selected
card and redraw the table
function onDragEnd(event){
    //console.log("Drag ended ", event);
    selectedCard = null;
    renderTable();
}

// When the drag is over - remove styled from the available
targets
function onDragOver(event){
    //console.log("Drag Over ", event);
    if(event.target.classList.contains("target")){
        event.target.classList.add("hover");
        event.preventDefault();
    }
}

// When we leave the card, remove the style from the selected
target
function onDragLeave(event){
    //console.log("Drag Leave ", event);
    event.target.classList.remove("hover");
```

```
}

// When we drop the card, send message to the backend
function onDrop(event){
    //console.log("Drop Event ", event);
    event.preventDefault();
    sendMessage({id: "ACT", active: selectedCard, target:
findSelectedCard(event.target)});
    selectedCard = null;
}
```

Now we have a clear idea of what is sent to our **backend**, so let's work on it for a bit.

First, let's make sure we give it data to work with. For that in a function call give your **actParsed()** 2 parameters that are 2 cards from the message - we just wrote them in our **onDrop** function. First one is active the second one is the target.

Example:

```
function msgReceived(arg){
  console.log("Received message "+arg.id);
  switch(arg.id){
    case "START":
      return createGso();
    case "ACT":
      return actParsed(arg.active, arg.target);
    default:
      console.log("Unknown message "+ arg);
      return gso;
  }
}
```

Now find the declaration of **actParsed()** function and make sure it also gets 2 parameters. Keep the names for consistency. What should this function do?

```
function actParsed(active, target){
  console.log(active, target);
  // find the first card in gso using coordinates
  // find the first card in gso using coordinates
  // Console log them to make sure we have right cards
  // if we don't have card two - player wants to put the card down
     at the empty space. Save this card to gso with target
     coordinates
  //else
```

```
  // if card 2 is hero, it means enemy card attacks the hero, then
     we need to lessen hero's health for card power and remove the
     second card from the gso using coordinates
  // if card 2 is shield we need to remove the second card from
     the gso using coordinates
  //if card 2 is attack card it means our hero attacked the enemy.
     Because our attack is always 3 and hero's health is always 3,
     we can simply remove the second card from the gso using
     coordinates.
  // for all this cases we need to remove active card too
  // A very special case is if our card 1 is healing card and it's
     not in the pocket (it's index is not 3) then we need to add
      health to hero and remove this card from gso
  // Make sure the game can continue
  // Return gso
}
```

First, let's note that we have a repeating action - removal of the card in GSO. Let's declare and write a function r**emoveCard()** that gets a card and position and will remove this card from this position. But only if it's not a hero card (the hero card always stays in game).

```
function removeCard(card, position){
  // Simply return if we have a hero
  // In any other case remove the card with position row and index
     from gso
}
```

Example:

```
// Remove used card only if it's not hero card
function removeCard(card, position){
  if(card.type=="hero"){
    return;
  }
  gso[position.row][position.index] = null;
}
```

Another function we just discussed is needed to check the game state. Declare it:

```
// Based on the game logic check if the player won or lost or game
continues
function checkGameState(){
  //if player's health is less then 1, set state in gso to LOST
  //if all card played (deck is empty) set state to WON
```

```
  //in all other cases five cards and set state to PLAY
}
```

Try to write it. If you need any help here's my example:

```
function checkGameState(){
  if(gso.playerHand[heroIndex].health <=0 ){
    gso.state = "LOST";
  } else if(gso.deck.length == 0){
    gso.state = "WON";
  } else {
    giveCards();
    gso.state = "PLAY";
  }
}
```

Time to return to our big **actParsed(active, target)** function and now, using these 2 helper functions, write it.
Compare your result with this example:

```
// Determine the act result
function actParsed(active, target){
  console.log(active, target);
  let card1 = gso[active.row][active.index];
  let card2 = gso[target.row][target.index];
  console.log(card1, card2);
  if(card2==null){
    console.log("Got an empty space");
    gso[target.row][target.index] = card1;
  } else {
    if(card2.type=="hero"){
      console.log("got an attack against hero");
      gso.playerHand[heroIndex].health =
          gso.playerHand[heroIndex].health - card1.power;
      removeCard(card2, target);
    }
    if(card2.type=="shield"){
      console.log("got an attack against shield");
      removeCard(card2, target);
    }
    if(card1.type=="attack"){
      console.log("hero attacks enemy");
```

```
      removeCard(card2, target);
    }
  }
  removeCard(card1, active);

  if(card1.type=="heal" && target.index!=3){
    gso.playerHand[heroIndex].health =
              gso.playerHand[heroIndex].health + card1.power;
    gso.playerHand[target.index] = null;
  }
  checkGameState();
  return gso;
}
```

You know what? That's all we need on the backend! Return to the frontend.

Fiddling -> Can we drop

The last big thing that's left here is the isDropTarget() function. We need to make sure the card is the right target for the action the player wants to make.

To write it we need to put all the game rules into game logic. We discussed them in the second chapter, let's return to them again.
If the player hasn't selected the card - action is impossible.
If the player selected the card but there's no target - we're dragging a card to the empty slot.

Empty slot conditions:
Then action depends on the type of the card we selected.
We can only put a card in a spot in player'sHand.
Enemy cards can't take spots in player's hand, and all other types can.
However, there's an exception - we can't move selected cards from player's hand to pocket.

If the slot that we're dragging a card in is taken, there are other rules.
Taken slot conditions:
if we're dragging an enemy card, it can be put on player's card (attack the player) or shield card (attack the shield).
sword card can only attack if it 's attacking a card in enemy's hand if the sword is in player's hand but not in a pocket.

It's easy to get lost in such a complicated set of rules. There are several things we can do to simplify it.

As you see, we only discuss allowed actions with a couple of exceptions. So, as a rule, our **isDropTarget** needs to return false. With that, we can start forming conditions.

Our conditions are based on 2 things - the type of the selected card and type of the target.

First, we'll need to make sure that if our selected card doesn't have a type (so no selected card), we also return false.

Then if there's no target (we're dragging to the empty space) we return true. Unless the selected card is in the pocket.

If we have a target and it's in player's hand while the selected card is enemy, we return true if the target is hero or shield.

Also, if we have a target and we selected sword, it can only attack if the target is in enemy's hand if it's type is enemy, a sword is in player's hand and not in the pocket.

The template with comments:

```
// Check if this action satisfies all the game logic conditions
function isDropTarget(targetCard, targetRow){
    console.log("Selected card ", selectedCard);
    // Save the selected card type into a vatiable, if we don't
have the selected card - set it to null
     // If selectedType is null - return false, we don't need to
     check for anything else
     //If targetCard is null, run a set of checks:
          //If targetRow is player hand and selectedType is not
          enemy check for the only exception: selectedCrds is in
          player's hand and the selectedCard is not in pocket
          (index !=3). In all other cases return true
     // If targetCard is not null it means the slot is taken.
     Start set of checks for taken slot.
          //If targetRow is playerHand and selectedType is enemy -
          The action only allowed if targetCard type is hero or
          shield
          //If targetRow is enemyHand, targetCard type is enemy,
          selectedType is attacking card, selectedCard is in
          player's hand and it's not in the pocket (index !=3) -
          allow the action
     // Return false in all other cases
}
```

Try it. If you get stuck at any point - check the complete function:

```
// Check if this action satisfies all the game logic conditions
function isDropTarget(targetCard, targetRow){
    console.log("Selected card ", selectedCard);
    const selectedType = selectedCard ? gso[selectedCard.row]
[selectedCard.index].type : null;
    //console.log("Selected type ", selectedType);
    if(!selectedType){
        return false;
    }
```

```
    if(targetCard==null){
        // we're dragging a card to an empty slot
        if(targetRow=="playerHand" && selectedType != 'enemy') {
            // we can only put into empty spot in a hand a shield,
potion or sword
            if(selectedCard.row=="playerHand" &&
selectedCard.index!=3){
                // we can move card from pocket to hand, but not
back to the pocket
                return false;
            }
            return true;
        }
    } else {
        console.log(targetRow, selectedType);
        // the slot is taken
        if(targetRow=="playerHand" && selectedType == 'enemy'){
             // enemy card can only attack hero and shields
            if(targetCard.type=="hero" ||
targetCard.type=="shield"){
                return true;
            }
        }
        // sword can only attack if it's in player's hand and the
enemy is in the enemy's hand
        if(targetRow=="enemyHand" && selectedType=="attack" &&
selectedCard.row=="playerHand" && selectedCard.index!=3 &&
targetCard.type=="enemy"){
            console.log("Here we are")
            return true;
        }
    }
    return false;
}
```

Run your game. PLAY YOUR GAME!

Congratulations, you've just made your first game in Electron. If you need to see my code for all 3 files, here's the gist link: https://gist.github.com/Rukia3d/92dd31ed8e7dc18cc2b3b2add25afa6a Of course, it's very simple and there are many things you can do to improve it.

- Make separate screens that you'll show when you lost or won.

- Create a button to restart a game.
- Make sure enemy can't attack the shield if the shield is in the pocket, not in hand.
- There's an edge case when the user has a full hand, an enemy has a full hand, but there are not enough possible actions. Create an additional place on the screen where the player can drag a card from her hand and discard this card.
- Add enemies with different health and attack. Work on cases when the player's attack didn't kill the enemy.
- Add shields with different power. Work on cases when the enemy's attack didn't kill the player.
- Move all of it to "Real" Electron (the connection system will stay the same!) and make it a lot more pretty with React and Images.

If you have any questions or want to share your feedback, please don't hesitate to contact me on inga.pflaumer@gmail.com or Twitter @IngaPflaumer
Thank you!