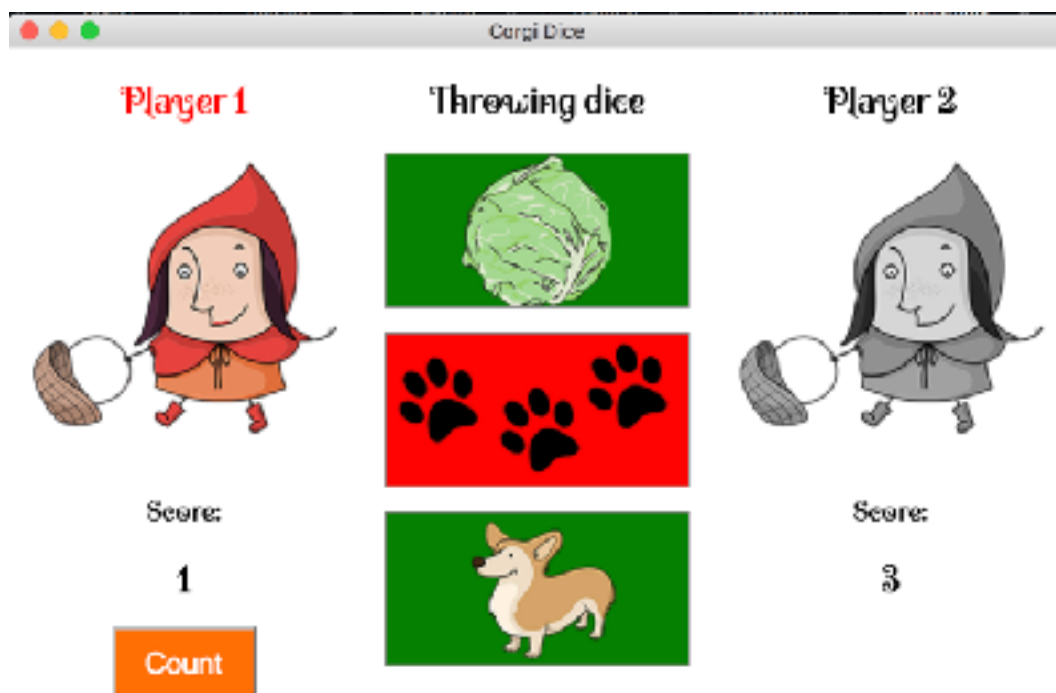


Corgi Dice with Electron

Corgi Dice with Electron	1
Intro	2
Flow	2
GSO and GAME LOOP	2
Setting up	3
React and Electron	6
Running our app	7
Main.js	9
Engine.js and engine.test.js	10
App.js Player.js and Board.js	19
Follow up	29



Intro

Are you familiar with a game “Zombie dice”? If not, here’s the video to watch:

<https://www.youtube.com/watch?v=xodehimqCVs>

Make sure to watch this video first, as it will introduce you to the rules of this game.

I am not a zombie person, I am a corgi person, so we’re going to make Zombie dice about corgis. And here’s the gif to watch too

<https://media.giphy.com/media/WLbtNNR5TKJBS/200.gif>

I don’t know what cabbage did to this corgi, but I like it, so let’s use this for inspiration.

Compared to the original Zombie dice, we’ll make the following changes to game theme:

- * “brain” symbols become corgi - that’s what we need to collect to score
- * “shotgun” becomes cabbage - these are bad
- * “steps” become paws - these can be re-thrown (“undecided”)

A side-note. If you are a beginner in gamedev, it’s always a good exercise to take a simple existing game with good balance (read - zombie dice) and try to recreate it. Because game balance is a discipline on its own, and it may need a separate workshop just for this. But our goal is to understand where to start creating your game and learn Electron, so, Corgi Dice it is.

Flow

In the beginning of everything there’s a planning stage. The first step of it is to close your eyes and imagine your game as clear as possible. Let’s look at our corgi dice. Don’t focus on details like dice or probabilities. Try to imagine and write down the flow of this game. What happens after what. What decisions players make? What logical checks the game needs to do?

1. The first player is selected (randomly)
2. First player grabs 3 dices.
3. First player throws 3 dices.
4. How many Good results, how many Bad results, how many Undecided?
5. Are there 3 Bad results? If yes - turn ends.
6. Does the player want to re-throw?

Now let’s put it into a diagram.

That’s a very important step. It gives you your game loop. The sequence of events that you are going to repeat until you have your winner.

GSO and GAME LOOP

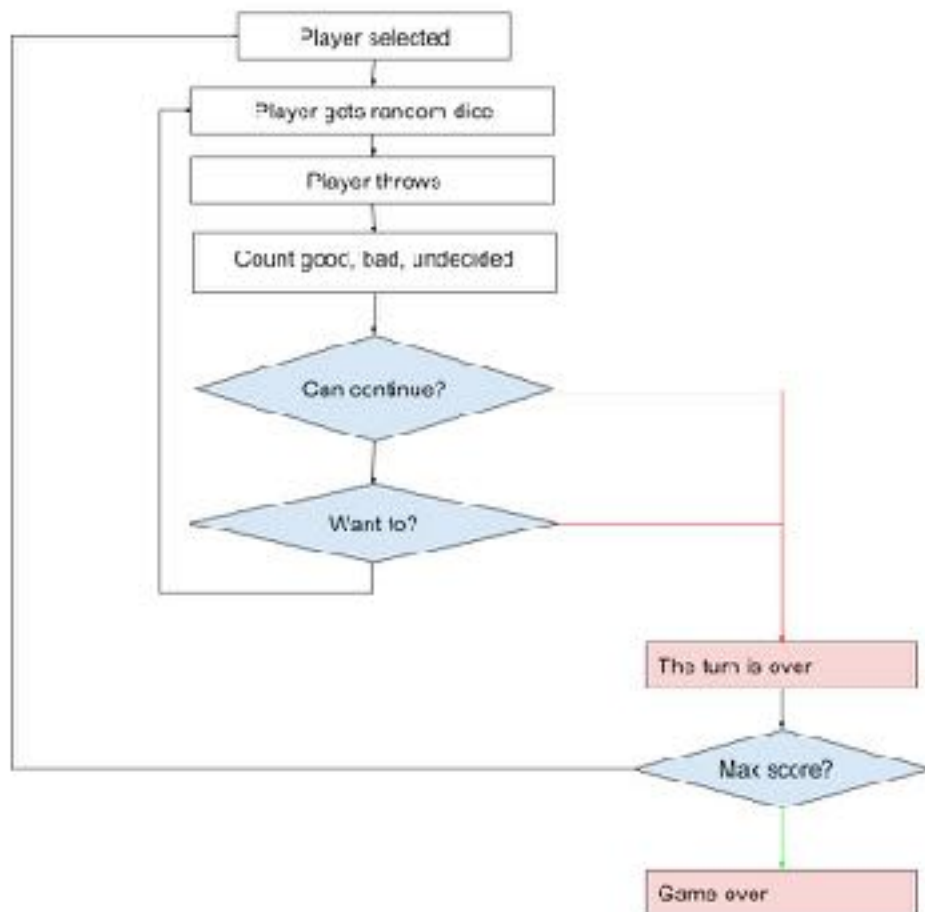
The flow diagram gives you the idea about the logical decisions your game needs to make, the player’s input and events this input is going to generate. With this flow in mind you can start working on your next step - figuring out the stages (or states) your game goes through.

We can model these states as an object that captures everything about the game at any given moment. Think of it as if you want to save and then restore the game, this object is what should bring your game back into its current state from nothing. We’ll be calling it “game state object” (GSO) from now on.

You can look at GSO as a state machine. Your frontend displays your game in this state when it received a GSO. Your Backend changes the state of the game by changing data in the GSO.

Our game loop defines the states that our game can be in. So now let’s list those states.

TURNSTART	Start of player's turn, waiting for player to draw new dice
DEALT	Player has drawn new dice, and has 3 dice to throw
THROWN	Player has thrown 3 dice, and their sides are now known
COUNT	Results of this turn are counted, and player either ends the turn or can go another round
AGAIN	Player goes another round - move "paws" back into dealt dice, start new round
TURNEND	End of turn - clear data, change player



The most complex process in the game is the dice operations. Let's list them separately

1. The Player draws dice. Dice are DEALT
2. Each dice reveals its side. Dice are THROWN
3. Dice that have corgi or cabbage stay in HAND, dices with a paw return to DEALT.

Setting up

To set up our project, I used create-react-app tool. If you want to use it in your own project, you can read documentation here:

Create React App <https://github.com/facebook/create-react-app>

There are additional steps required to set it up with electron, because it needs some extra files. There's a good article on this worth looking at, especially if you want to use this setup in your future projects.

How to build an Electron app using create-react-app <https://medium.freecodecamp.org/building-an-electron-application-with-create-react-app-97945861647c>

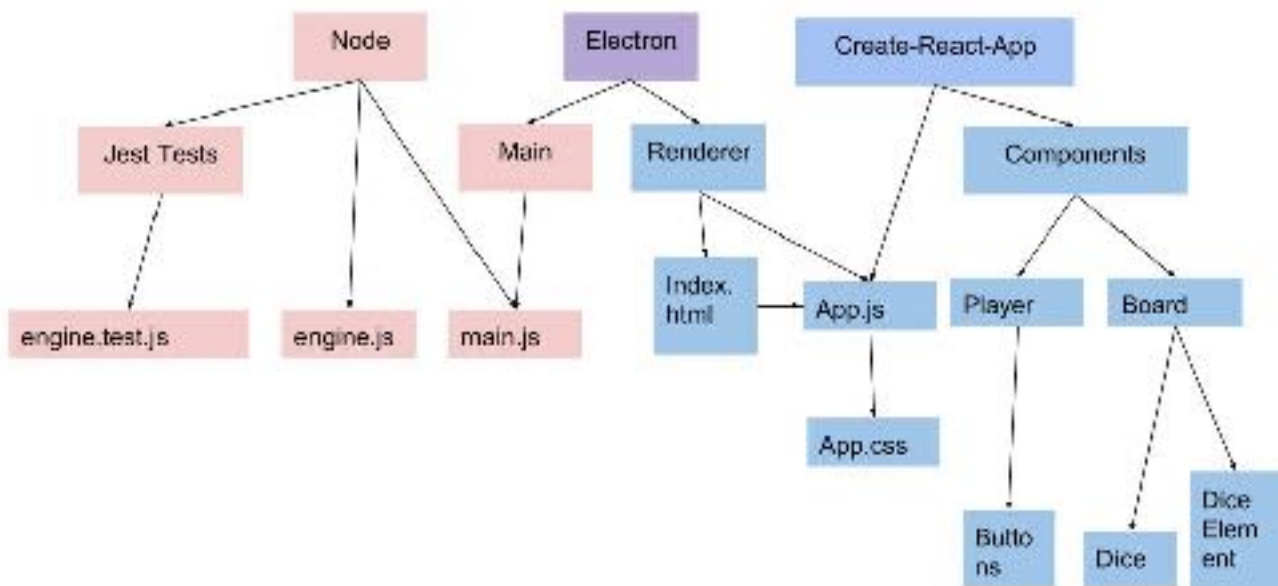
You will need to make sure you have installed node and npm, the guides for the installation on different platforms you can find here:

How to Install Node.js and NPM on a Mac OS X: <https://www.youtube.com/watch?v=rF1ZHmqvm8I>

Installing Node & Updating NPM on Windows <https://www.youtube.com/watch?v=W6S5PMe2DBM>

How to Install Node.js and Npm on Ubuntu Linux <https://www.youtube.com/watch?v=K6QiSKy2zoM>

For this workshop you will need to clone the repository with work files. We will do it in a second, but before that let's take a look at our project structure:



Project on Git: <https://github.com/Rukia3d/CorgiDice>

In your command line go to desktop (cd Desktop) and clone it using command
git clone [git@github.com:Rukia3d/CorgiDice.git](https://github.com/Rukia3d/CorgiDice.git)

When you create a project with Create React App, it gives you a part of the structure. To integrate electron in it, you need to add `main.js` (and `engine.js` in our case) plus a file to write tests (`engine.test.js`)

You can use `App.js` as your `Renderer`, and we will talk about it when we get to the frontend part of the project.

For now, clone the project and let's see the files we need to change.

The files that we need to use from root folder are:

main.js - the main process for backend in the electron and the entry point for the messages from frontend

engine.js - our game logic and the main communication channel for GSO

The `__tests__` folder contains only one file - **engine.test.js**

That's because for the purpose of this project we only test our game logic, described in `engine`.

The **src** folder is where our frontend lives. We will work on **App.js** (main frontend logic), the other files of interest are **App.css** (styles for our elements) **Index.html** (the only change here is a font import and project name)

2 components of our frontend are **Player.js** and **Board.js**

Folder data contains 2 json object files, that will never change directly:

gso.json is our gso in it's initial stage, before backend made any changes to it.

dice.json keeps our 13 dices that we will randomly give to player.

Folder images contains images for our project. We will import them into React components that use them.

Now let's clone the project and start doing magical things!



```
MacBook-Pro:~ inga$  
MacBook-Pro:~ inga$  
MacBook-Pro:~ inga$  
MacBook-Pro:~ inga$ cd Desktop  
MacBook-Pro:Desktop inga$ git clone git@github.com:Rukla3a/ConglDice.git
```

Note that a complete version of those files is included in `Solutions` folder, so if you get stuck at any time - look at a relevant file in `Solutions`.

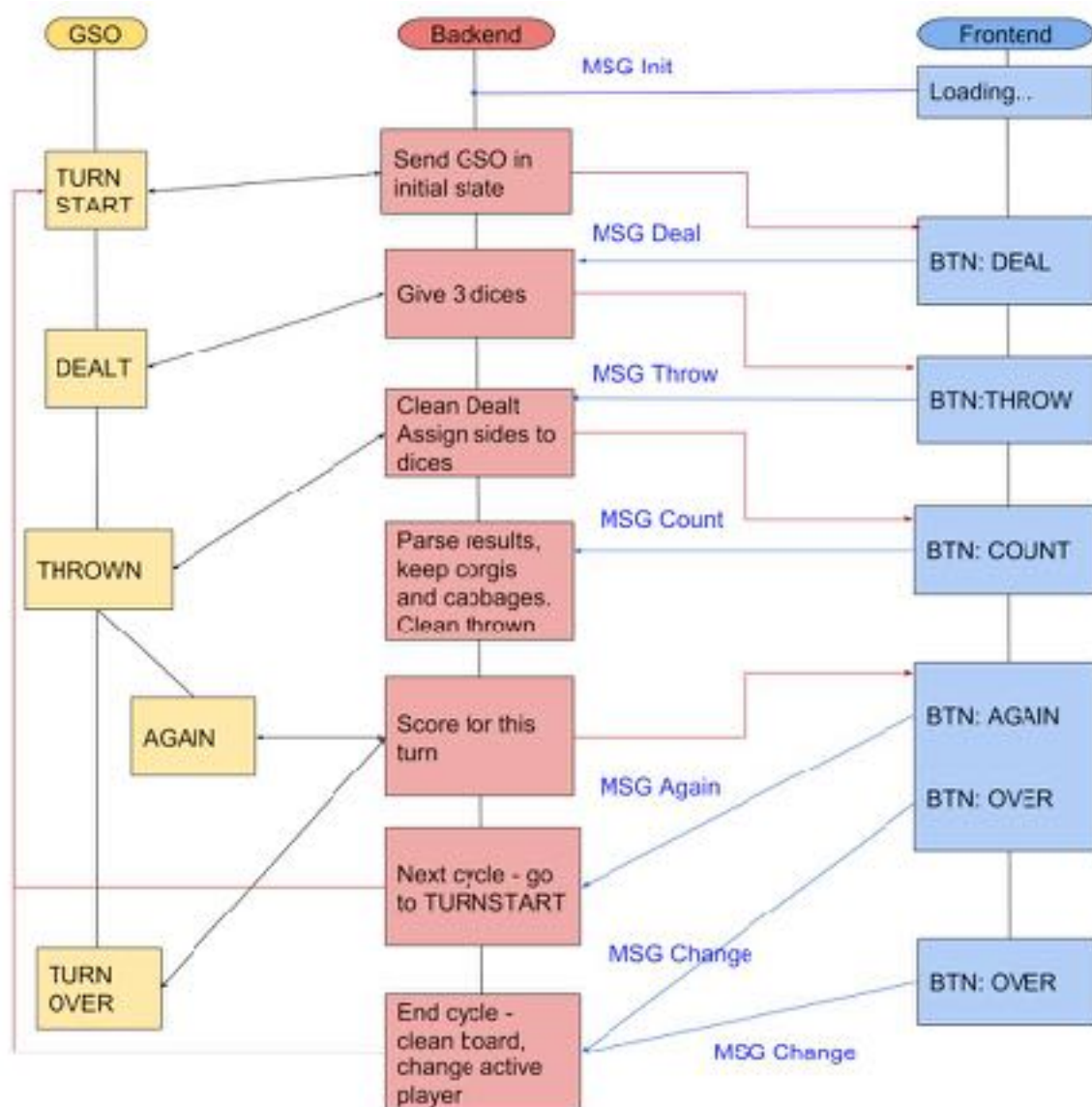
React and Electron

Electron is a framework that allows you to use your web dev tools to create Desktop applications. Including games.

The main thing you need to know about Electron is that it separates your application into 2 processes - Main process and Renderer process. You can think of Renderer as your frontend, as this is the part that is visible to the end user, and of Main process as your backend. Renderer runs in a browser window, and it allows to use HTML, CSS and JS as you normally would in the browser. Main process uses node.js, and controls creation/destruction of the window object that hosts Renderer process.

This separation affects your project architecture, and you can think of it in a way you think of MVC. GSO is your Model. Backend is your Controller. Main is your View.

Let's look at our architecture a bit closer.



As you can see from it, we separated our functionality and we use one directional data flow. So our data, our GSO is changed only by our backend. Backend sends it to the frontend to be drawn, but frontend never changes the GSO. It draws it and then messages the backend for the next step.

Running our app

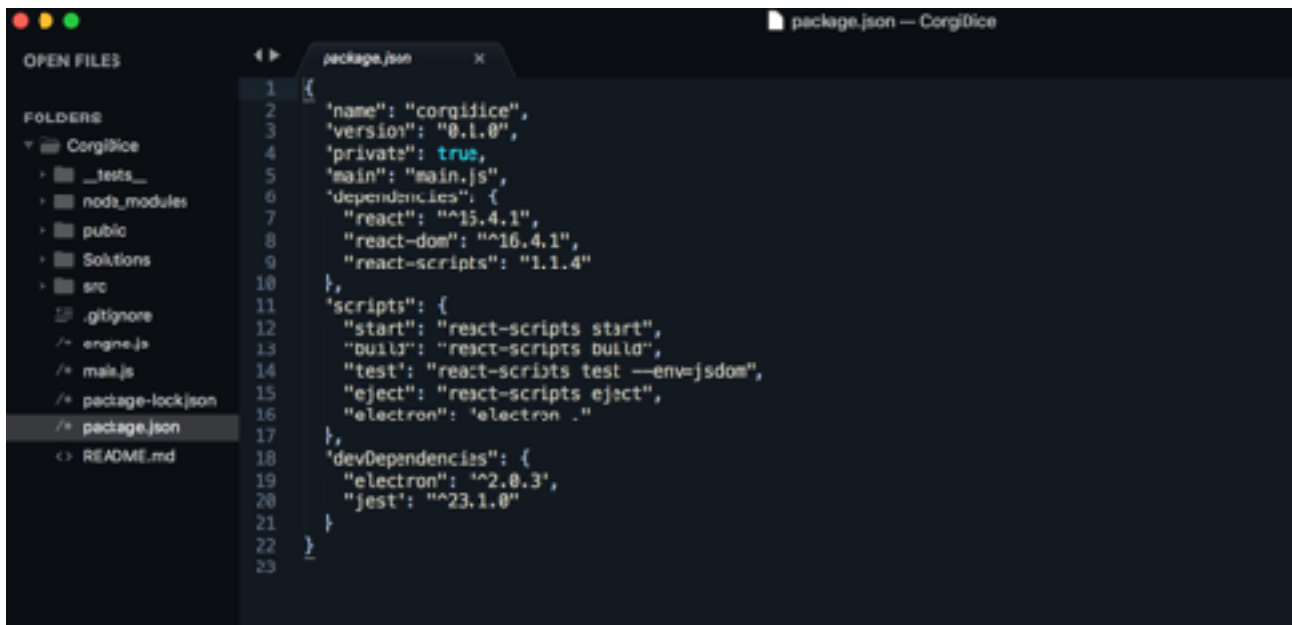
First, before we move to any coding, we need to install all the dependencies of the project, to make sure we have all the relevant packages installed.

To do so, use command line, enter the project folder in it. If you cloned the project to your desktop, then you need to type following:

```
cd Desktop
cd CorgiDice
npm install
```

```
MacBook-Pro:~ inga$
MacBook-Pro:~ inga$
MacBook-Pro:~ inga$
MacBook-Pro:~ inga$ cd Desktop
MacBook-Pro:Desktop inga$ cd CorgiDice
MacBook-Pro:CorgiDice inga$ npm install
```

Let's look at the project structure. Open it in any text editor (I am using Sublime Text) and open package.json. You'll see it has electron and jest in dependencies. We will need them to start working on our backend.



To run our app we will need 3 command line windows.

One to run compiler for React

Second one to run Electron

Third one to run our Tests with Jest

In the first window, get into the project folder and run npm start run.

```
MacBook-Pro:~ inga$
MacBook-Pro:~ inga$
MacBook-Pro:~ inga$
MacBook-Pro:~ inga$ cd Desktop/
MacBook-Pro:Desktop inga$ cd CorgiDice
MacBook-Pro:CorgiDice inga$ npm run start
```

This will run React compiler and open local React application in your browser, showing a number of errors. It's not a problem, because we will fix them pretty soon. Our main goal is to start React compiler, so each time we changed something in our code - the app recompiles its code and displays new frontend to us.



in the second window let's start our electron. In package.json you can see the scripts section and electron has command **"electron ."** defined for it (make sure to type that dot after the command - it tells electron to start application from the current directory). So open the second command line window, get into the project folder and run electron .

```
MacBook-Pro:~ inga$  
MacBook-Pro:~ inga$  
MacBook-Pro:~ inga$ cd Desktop  
MacBook-Pro:Desktop inga$ cd CorgiDice  
MacBook-Pro:CorgiDice inga$ electron .
```

It will open the empty electron window that will have our game after we finish our frontend code. If you want to stop it just press `ctrl+C` and then you can restart it with `"electron ."` at any time.

The third window we will use for tests.

```
MacBook-Pro:~ inga$  
MacBook-Pro:~ inga$  
MacBook-Pro:~ inga$ cd Desktop  
MacBook-Pro:Desktop inga$ cd CorgiDice  
MacBook-Pro:CorgiDice inga$ jest engine.test.js
```

I will walk you through them in the next chapter, but to start tests you need to run command `jest testfilename.js` (or just “`jest`”).

If you run them right now, all the 11 tests should fail miserably, but that’s ok, we are going to write our engine in a way that passes all of those tests.

Now let's code our game!

Main.js

Open our main.js file in your code editor and let's work on it a bit.

You will see that the file you have already contains a lot of code. All of it came from the “Electron in a nutshell” tutorial, and it's almost everything you need to make it work. The file is well commented, so you can read through it and see what each function does.

Electron Development in a Nutshell

<https://electronjs.org/docs/tutorial/first-app#electron-development-in-a-nutshell>

Let's just change some things.

First line in main.js imports app and browser window. We are going to use this file as our Main process, so we need to import ipcMain into it. So change the first import line to:

```
// Modules to control application life and create native browser window
const {app, ipcMain, BrowserWindow} = require('electron')
```

This import allows you to use several electron features in your Main module.

Then find a function createWindow() and instead of loading index.html tell it to load localhost (remember, that's what our React tries to load)

That's what your createWindow function should look like:

```
function createWindow () {
  // Create the browser window.
  win = new BrowserWindow({width: 800, height: 600})

  // and load the index.html of the app.
  win.loadURL('http://localhost:3000/')

  // Open the DevTools.
  win.webContents.openDevTools()

  // Emitted when the window is closed.
  win.on('closed', () => {
    // Dereference the window object, usually you would store windows
    // in an array if your app supports multi windows, this is the time
    // when you should delete the corresponding element.
    win = null
  })
}
```

Now let's add some of our logic.

We already know the messages that our backend will get from frontend. But we don't want to parse those messages here. Why? Because we don't need to mix game logic with the app logic. It's always a good idea to test and keep your logic separate in case you decide to change the app an instead of using Electron use Proton or anything else. Then to change the app you will just need to rewrite the main file, and you can keep your game logic (engine) as it is.

First we need to import our engine file. So right under your declaration of the win (window), import engine as a constant. Insert the following:

```
// Require the setup file to parse messages from frontend.  
const engine = require('./engine.js');
```

Now we can access our engine, and we need to pass all the messages to it. Scroll down to the bottom of the file where you have your own logic and add the message handler.

```
// In this file you can include the rest of your app's specific main process  
// code. You can also put them in separate files and require them here.  
ipcMain.on('MSG', (event, arg) => {  
  // Send the request to game engine to get relevant data.  
  var received = engine.msgReceived(arg);  
  
  // Send back the answer.  
  win.webContents.send("GSO", received);  
});
```

What does it do? When our ipcMain receives a message, it calls our engine and gives the message to it. Our engine answers with GSO. ipcMain takes the GSO it has received and sends it back to the browser window. Exactly as we have it in our diagram.

That's all we need from our main.js.

Engine.js and engine.test.js

Let's open our engine.test.js and engine.js

You will see that engine.test.js requires engine the same way as main.js does. It's needed so test can call the function from the file it's testing.

There you will see 11 different tests that we run. They describe different states we expect our game to be in. Of course, this is just the bare minimum of tests here, you can literally write tests for every sneeze of your engine. Let's look at some of them.

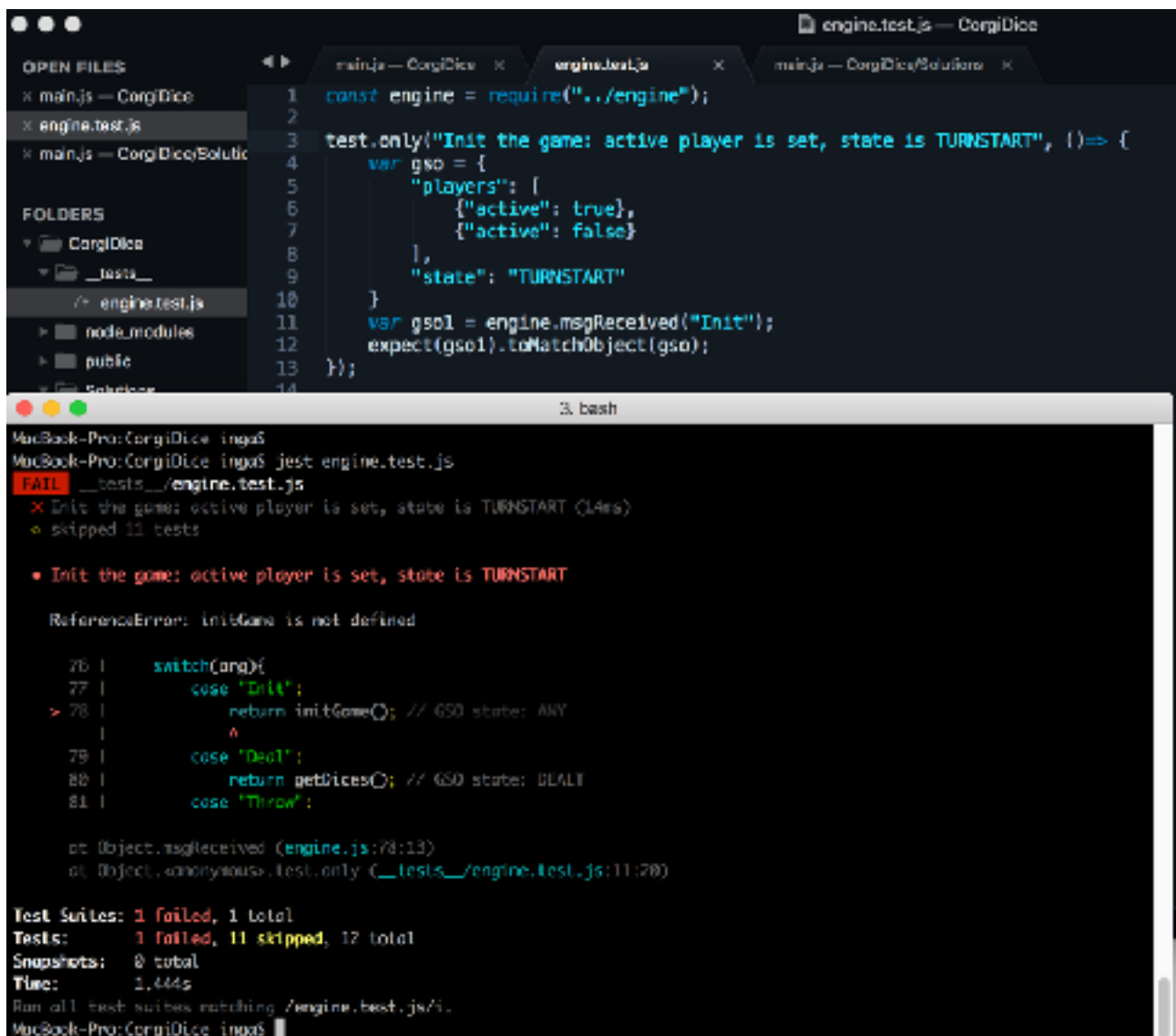
```
test("Init the game: active player is set, state is TURNSTART", () => {  
  var gso = {  
    "players": [  
      {"active": true},  
      {"active": false}  
    ],  
    "state": "TURNSTART"  
  }  
  var gso1 = engine.msgReceived("Init");  
  expect(gso1).toMatchObject(gso);  
});
```

What is happening here?

First we create a minimal gso that has 1 active player and the needed state - this is the state we expect our game to be in after Init message is processed.
Then we call an engine function msgReceived() giving it a message that we want our engine to receive as a parameter ("Init"). Then we check that our resulting gso matches the expected gso we just created.

Here you can see a typical way tests with jest are written. You define a test case, write the expected result and then expect that the result of a call to this function will match this expected result.

To run only this test you need to add the word .only after the test and then run it from command line with jest



The screenshot shows a VS Code editor with three tabs: 'main.js - CorgiDice', 'engine.test.js', and 'main.js - CorgiDice/Solutions'. The 'engine.test.js' tab is active, showing the following code:

```
1 const engine = require("../engine");
2
3 test.only("Init the game: active player is set, state is TURNSTART", () => {
4     var gso = {
5         "players": [
6             {"active": true},
7             {"active": false}
8         ],
9         "state": "TURNSTART"
10    }
11    var gso1 = engine.msgReceived("Init");
12    expect(gso1).toMatchObject(gso);
13 });
14
```

Below the editor, a terminal window shows the output of running 'jest engine.test.js'. The test fails with the message 'ReferenceError: initGame is not defined'. The terminal output is as follows:

```
MacBook-Pro:~ ingas$
MacBook-Pro:~ ingas$ jest engine.test.js
FAIL __tests__/engine.test.js
  ✕ Init the game: active player is set, state is TURNSTART (14ms)
  • skipped 11 tests

  ● Init the game: active player is set, state is TURNSTART

    ReferenceError: initGame is not defined

      76 |     switch(arg){
      77 |       case "Init":
    >    78 |         return initGame(); // GSO state: ANY
          |         ^
      79 |       case "Deal":
      80 |         return getDices(); // GSO state: DEALT
      81 |       case "Throw":

      at Object.msgReceived (engine.js:78:13)
      at Object.<anonymous> test.only (__tests__/engine.test.js:11:20)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 11 skipped, 12 total
Snapshots:   0 total
Time:        1.444s
Run all test suites matching /engine.test.js/i.
MacBook-Pro:~ ingas$
```

This test will fail and it will tell you the reason - your initGame is not defined. So let's define it and test again. To do it open our engine.js

Our engine.js already have several functions defined. In the first 2 lines it imports our dice as a constant (we won't need to change them) and our gso as var (because for some tests we will need to change it). It also has all the helpers (functions that our game process needs), we will talk about them when we need to use them. For now scroll to the end of the file.

You will see 2 exports at the bottom - they export 2 functions that we need to access from outside.

setGSO() function is needed for our tests. Sometime for your tests you need to set your GSO into specific state and then test. That's what this function does.

msgReceived() function is the function we already saw in our main and our test. It takes the message as an argument, and it returns the game state object in the new changed state (remember our diagram).

Let's look at this function closer:

```

/***** MESSAGES *****/
// Parsing the message we get front frontend to engine through Main
function msgReceived(arg){
  switch(arg){
    case "Init":
      return initGame(); // GSO state: ANY
    case "Deal":
      return getDice(); // GSO state: DEALT
    case "Throw":
      return throwDices(); // GSO state: THROWN
    case "Count":
      return countScore(); // GSO state: TURNEND or AGAIN
    case "Again":
      return moreDice(); // GSO state: TURNSTART
    case "Change":
      return changePlayer(); // GSO state: TURNSTART
  }
};

```

*If you need any help with switch statements, you can read up on them:
JavaScript Switch Statement: https://www.w3schools.com/js/js_switch.asp*

As you can see msgReceived() has all the messages that our frontend sends. Then based on them it should return the gso as a result of the function it's called. So the error we had in our test was that initGame() wasn't defined. Let's work on it.

function InitGame()

This function should return the GSO in its current state. You can check the gso.json to see that this is the state our GSO is in by default. So our InitGame function is very simple. Let's write it in engine.js as our first Game State

```

/***** GAME STATES *****/
// The initial state of the game
function initGame(){
  return gso;
}

```

Our gso is imported at the beginning of the file, so we just return it in its initial state. Time to run tests!

It should pass now. Remove the word .only from it, as we are done with it for now.

```

MacBook-Pro:CongiDice inga$
MacBook-Pro:CongiDice inga$ jest engine.test.js
PASS __tests__/engine.test.js
  ✓ Init the game: active player is set, state is TURNSTART (5ms)
  ○ skipped 11 tests

Test Suites: 1 passed, 1 total
Tests:       11 skipped, 1 passed, 12 total
Snapshots:   0 total
Time:        0.915s, estimated 1s
Ran all test suites matching /engine.test.js/i.
MacBook-Pro:CongiDice inga$

```

function getDice()

Move to our second test, add .only to it. Let's see what it does.

```

test.only("Give dice - player didn't throw yet, state is DEALT", () => {
  engine.setGSO({
    "players": [
      { "active": true },
      { "active": false }
    ],
    "board": {
      "dealt": [],
      "thrown": [],
      "hand": []
    }
  });
  var gso = engine.msgReceived("Deal");
  expect(gso.state).toEqual("DEALT");
  expect(gso.board["dealt"].length).toEqual(3);
  expect(gso.board["thrown"].length).toEqual(0);
  expect(gso.board["dealt"][0]).not.toEqual(gso.board["dealt"][1]);
  expect(gso.board["dealt"][0]).not.toEqual(gso.board["dealt"][2]);
  expect(gso.board["dealt"][1]).not.toEqual(gso.board["dealt"][2]);
});

```

If you are having trouble reading this test, refresh your JSON knowledge and arrays knowledge.

JSON Syntax https://www.w3schools.com/js/js_json_syntax.asp

JavaScript Arrays https://www.w3schools.com/js/js_arrays.asp

This tests sets our expected gso and makes sure that if we message “Deal” to our backend, the returned gso has its state changed to “DEALT”, the board has 3 dice, all of them are unique and none of them are thrown at that point.

If you run this test it will also fail because we didn’t define our getDice() function in engine.js. What does this function need to do?

First it needs to know how many dice we need to take from dice.json. We know that player always needs 3 dice, but if any of the dice in the previous throw got him “paws” - these dice need to be thrown again.

Dice that were already dealt to the player are kept in `gso.board.dealt`. So to get the number of the new dice we need, we have to deduct the already dealt dice from 3 that we need.

Then add the number of dice we calculated to the `gso.board.dealt` and set GSO into "DEALT" state.

We have a helper function **`getRandomDice(gso.board)`** that returns us the dice needed. It has some interesting logic in it, so we will look at it first:

```
// Helper to make sure we get random dices
function getRandomDice(board){
  var newDice;
  do {
    var diceIndex = getRandomUpTo(12);
    newDice = dice["dice"+diceIndex]["name"];
  } while(board.dealt.indexOf(newDice) != -1 ||
    board.hand.map(x => x.name).indexOf(newDice.name) != -1);

  return newDice;
}
```

If you have trouble reading this code, here's some helpful info:

JavaScript do/while Statement https://www.w3schools.com/jsref/jsref_dowhile.asp

JavaScript Array indexOf() Method https://www.w3schools.com/jsref/jsref_indexof_array.asp

Array.prototype.map() https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

What does **`getRandomDice(gso.board)`** do?

It takes the board.

Our board in `gso` contains info about the dice that is dealt, thrown or kept in hand. So `getRandomDice` creates a placeholder for a new dice.

Then it generates random number from 0 to 12 (we have 13 dice in game at all times and indices in Javascript start from 0).

Then it grabs a dice with this random index from our `dice.json` that we imported earlier.

But then it checks if this random dice that we got is already in dealt or in hand (so it's already on the table and can't be used). If this is the case (`indexOf` returns something that is not -1) - the function will do this random search again.

So, this function will find us a random new dice that has not been used. Exactly what we need! We just need to save its result into "dealt" as many times as we need.

```
// Give dices to player based on his previous actions
function getDice(){
  var diceNeeded = 3-gso.board["dealt"].length;
  for(var i=0; i<diceNeeded; i++) {
    gso.board["dealt"].push(getRandomDice(gso.board));
  }

  gso.state = "DEALT";
  return gso;
}
```

If you are not sure how For loop works, here you can find some info: JavaScript For Loop https://www.w3schools.com/js/js_loop_for.asp

To explain what is happening in getDice()

You count how many dice you need (the dice that will be re-thrown are in “dealt” on the board, so you always need 3, but if you have 2 in “dealt”, you just need to add 1 new dice to them.

Run your test again. it should pass! Even more so, tests 3 and 4 test the same function, but with a different set of data. Let’s see what they change.

Run .only Test 3 and let’s look at it. This test does pretty much the same checks as Test 2, but this test sets GSO as if it has something in “dealt” on the board. So if we wrote our functions correctly, dice8 and dice4 will stay in dealt and only 1 new dice will be added. It will be a new dice - not dice8 and not dice4. Run it. It should pass.

```
test.only("Give dice - player has 2 paws", () =>{
  engine.setGSO(
    {
      "players": [
        { "active": true }
      ],
      "board": {
        "dealt":["dice8", "dice4"],
        "thrown": [],
        "hand" : [],
      }
    }
  )
  var gso = engine.msgReceived("Deal");
  expect(gso.state).toEqual("DEALT");
  expect(gso.board["dealt"].length).toEqual(3);
  expect(gso.board["dealt"][0]).toEqual("dice8");
  expect(gso.board["dealt"][1]).toEqual("dice4");
  expect(gso.board["dealt"][2]).not.toEqual("dice8");
  expect(gso.board["dealt"][2]).not.toEqual("dice4");
});
```

Test 4 runs the same test for another player. We will deal with it a bit later, though it should pass anyway.

function throwDices()

Test only test 5. It will fail for the same reason. We don’t have throwDices function yet. Let’s write it.

What should it do?

Pretty straightforward.

It should take the dice that were dealt and assign each one of them a side.

You have a helper function throwRandomSides(board) let’s look at it first:

```
// Helper to throw 1 of 6 sides of this dice
function throwRandomSides(board){
  board.dealt.forEach(function(diceName){
    var playersDice = dice[diceName];
    var result = {name: playersDice["name"], side : playersDice.sides[getRandomUpTo(5)]};
    board.thrown.push(result);
  });
  board.dealt = [];
  return board;
}
```

For each of the dice we have in dealt, it finds dice with this name in our dice.json and then uses this original dice to create the result object.

*If you need help understanding what we save into the result and why - check JavaScript Objects https://www.w3schools.com/js/js_objects.asp
Help with forEach function
Array.prototype.forEach() https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach*

The result object contains the name of the dice and the random side. Random side is assigned the same way as in the previous helper functions - it takes one of the “side” elements from the array based on random index from 0 to 5. Then it pushes this result into thrown.
In the last step it clears “dealt” array, because we always throw all the dice we were dealt.

Now let’s write our **throwDices()** function. It should call our **throwRandomSide()** function and change the state of GSO

```
// Assign sides to dices
function throwDices(){
  throwRandomSides(gso.board);
  gso.state = "THROWN";
  return gso;
}
```

Run the test again. It should work now.

function countScore()

As usual, start with failing test. Our test 6, 7, 8, 9 test countScore function. You can set them all to .only and run - all 4 of them will fail.

To define our countScore() functions we need 2 of our helpers. Let’s first look at what countScore function needs to do.

It needs to analyse throw results to see if our player can proceed (3 cabbages stop the game for this player).

If we have 3 cabbages - we set the state to TURNEND.

If we don’t have 3 cabbages - player can decide if he wants to throw AGAIN or TURNEND.

2 helper functions are parseThrowResults() - to analyse the results and scoreTable() to set the score.


```
// Check and count the results
function parseThrowResults(board){
  board.thrown.forEach(function(playersDice){
    if(playersDice.side=="paws"){
      board.dealt.push(playersDice.name);
    } else {
      board.hand.push(playersDice);
    }
  });

  board.thrown = [];
  return board;
}
```

This function takes all thrown dice, and if the side symbol is paw - pushes it back to dealt so player can throw it again. If the side is not paw - dice stays in hand.
Then this function clears thrown, because in the next throw it needs to be refilled.

We get the board as a parameter, and return the board that we changed. There's no need to do this, but it allows us to keep the pattern where our message functions don't get any parameters and return gso, but helper functions get the object they need to change.

Now the second function:

```
// Check the number of points for cabbages and corgis
function scoreTable(board){
  var score = {
    "corgi" : 0,
    "cabbage" : 0
  }

  board.hand.forEach(function(playerDice){
    score[playerDice.side] += 1;
  });

  return score;
}
```

This function counts corgis and cabbages and returns the score. Now we can use both of this functions in countScore()

```
// Count the score and change the state based on it
function countScore(){
  var playerWithResults = parseThrowResults(gso.board);
  if(scoreTable(playerWithResults).cabbage >=3){
    gso.state = "TURNEND";
  } else {
    gso.state = "AGAIN";
  }

  return gso;
}
```

If the score is 3 cabbages - it will set the state to TURNED. In any other case it will set the state to AGAIN and the user can decide what to do next.

Test it with jest. Should work now.

function moreDice()

We have 3 tests left. Test 9 tests the Again condition. If you run only it - it should fail, because we need a function moreDice() for it to work.

What this function should do? The only thing it needs to do is to set the gso state to TURNSTART so our player can request the dice again. Write it and test again.

```
// Change the game state to Turnstart if the uplayer wants to continue
function moreDice(){
  gso.state = "TURNSTART";
  return gso;
}
```

function changePlayer()

This function needs to update the player's score according to the turn score (we have scoreTable function to get it). Then it needs to change the active player and clear out the board data so we can fill it for the next player.

To get the score in this turn we need to call our scoreTable (it returns the number of corgis and cabbages for this set of dice) and the score will be corgis minus cabbages.

Then we need to add this number to active player score if it's > 0 (we don't save the negative score).

Then we clear the board by setting dealt, thrown and hand to empty arrays.

And if the active player was player 1, we make him inactive (or vice versa for player 2)

```
// Change the played at the end of the turn, clean up the board
function changePlayer(){
  // count score
  var score = scoreTable(gso.board);
  var realScore = score.corgi - score.cabbage ;

  const activePlayer = gso.players[0].active ? gso.players[0] : gso.players[1];
  if(realScore>0){
    activePlayer.score += realScore;
  }

  // reset board
  gso.board.dealt = [];
  gso.board.thrown = [];
  gso.board.hand = [];
  // change player
  gso.players[0].active = !gso.players[0].active;
  gso.players[1].active = !gso.players[1].active;
  gso.state = "TURNSTART";
  return gso;
}
```

Run all of your tests (remove the word `.only` from all the tests) they should pass now.
And you are done with backend!

```

$ npm run all test suites matching /engine.test.js/i.
MacBook-Pro:CongiDice inga$ jest engine.test.js
PASS  __tests__/engine.test.js
  ✓ Init the game: active player is set, state is TURNSTART (6ms)
  ✓ Give dice - player didn't throw yet, state is DEALT (2ms)
  ✓ Give dice - player has 2 paws in hand (3ms)
  ✓ Give dice to second player - player didn't throw yet, state is DEALT (1ms)
  ✓ Throw the dices player got, set state to THROWN (2ms)
  ✓ Player threw 3 cabbages - his turn is over, set state to TURNEND (1ms)
  ✓ Player threw 3 paws, set state to AGAIN (2ms)
  ✓ Player has 1 corgi, 1 paw and 1 cabbage, it's second cabbage, state is AGAIN (1ms)
  ✓ Player has 2 corgis and 1 cabbage, it's third cabbage, state is TURNEND (1ms)
  ✓ Player wants to play again, state is TURNSTART (1ms)
  ✓ The turn is over, count points, player 1 was active (2ms)
  ✓ The turn is over, count points, player 2 was active (1ms)

Test Suites: 1 passed, 1 total
Tests:       12 passed, 12 total
Snapshots:   0 total
Time:        0.918s, estimated 1s
$ npm run all test suites matching /engine.test.js/i.
MacBook-Pro:CongiDice inga$

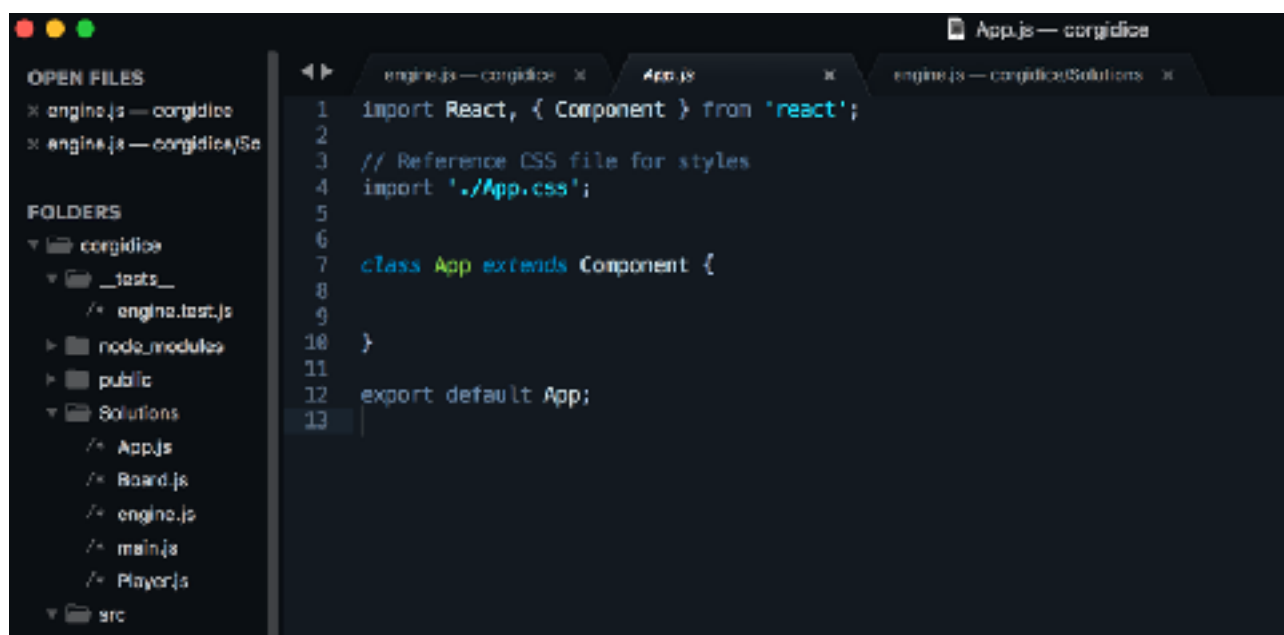
```

App.js Player.js and Board.js

Let's move to frontend.

Side note - to speed up the process, you have a bare minimum of styles in your App.css class. This workshop is not about styling with css, but we will talk about assigning and using css classes. If you are comfortable with css - feel free to explore it and change in any way you like.

If you remember our project structure, App.js should play the role of Render process in electron and message our backend depending on player's actions. So it will need some electron files to function. Let's look at where we start.



You can see that we import react components in the first line. Then we import our css. And then we have our App class that has a type of Component. This class is exported to be used outside of this code (we will export all of our classes on frontend).

So, let's write some code!

First we import electron classes that we need to use. Electron itself and ipcRenderer to message backend.

```
// Import electron and establish connection to use app.js as Renderer
const electron = window.require('electron');
const ipcRenderer = electron.ipcRenderer;
```

The smart react components (components that have logic and state in them) have several functions in them and we are going to use those functions. We will need a constructor function. It has a standard syntax where it gets the props argument (you can think of it as additional data - we will use it later), calls the super class and then saves the state.

```
// Set the initial state, before contacting backend.
// Bind the function to send messages
constructor(props){
  super(props);
  this.state = {gso: {state: "loading"}}
  this.sendMessage = this.sendMessage.bind(this);
}
```

Only our App will have constructor, we won't need it for our smaller components. So let's add a constructor inside our App class.

What does it do? First it calls the super class and gives it props. It is not relevant in our case, just think of it as a requirement for all the constructors in React. Then it sets the initial state of the application. In rare cases that our connection between Renderer and Main is slow and the player started the application but didn't get the GSO to draw - it will start in this initial state and then changes it when it gets to GSO. Then we bind the function sendMessage to the component.

You can read on binding React components and functions here. Binding functions in React <https://codeburst.io/binding-functions-in-react-b168d2d006cb>

Now we need to write this sendMessage() function. It's only function is to send the messages to backend using ipcRenderer that we exported a bit earlier.

```
// Using electron Renderer send message to Main process.
sendMessage(msg){
  ipcRenderer.send('MSG', msg);
}
```

And that's it for our communication. We will need to use this `sendMessage()` function on our buttons and each time the user presses a button on frontend - the backend will react. Cool, right?

```
// When the page loads, get GSO from backend, save it to state.
componentDidMount(){
  ipcRenderer.on("GSO", (event, arg) => {
    this.setState({ gso: arg });
  });
  this.sendMessage("Init");
}
```

Now we can focus on drawing our GSO that backend will give us.

Another important React function that we need to write inside `App.js` is `componentDidMount()`

This special React function tells your application to do something when the component mounts.

What do we want to do in this case? We want to get our GSO from Main, save it to state and send our first message. In our backend we expect "Init" at the beginning of the game, so that's what we need to send.

If you step back for a bit, you will see the pattern of how Electron works. On our backend we use `ipcMain.on("MSG")` and `win.webContents.send("GSO")` to receive and send messages. In frontend we use `ipcRenderer.send('MSG')` and `ipcRenderer.on("GSO")` to continue communication.

But let's get back to drawing our game on frontend.

The most important function in the component is the `render()` method that literally renders your component. Note the syntax - the render should always return only one root component, so make sure your component data is wrapped up in one `<div>` or something like this.

What should our render do? It needs to render our table. To clean up a bit we will move the table render to another function and just call it here.

```
// Render the application
render() {
  return (
    <div className="App">
      <div>
        {this.state.gso.state==="loading" ? <i>Loading...</i> : this.renderTable()}
      </div>
    </div>
  );
}
```

Ok, this syntax may look confusing, but we will walk through it.

First our return statement returns one big `<div>` element. The `className` inside it just attaches a CSS class to make it look good and pretty (you can look at the style for ".App" in `App.css` if you want).

Inside this App div we have a smaller div that contains javascript condition. It says "if this state is "loading" (and we save loading as a state in our constructor) then draw me the word Loading on the table. If the state is something else - render me the table.

We use `{ }` to write javascript inside React components and to speed things up most of the our if statements use short ternary operator `? : -` it is just the fastest way to write if statements.

You can read on ternary IF statements here:

Conditional (ternary) Operator https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

Another thing to note is that we use this.renderTable with parentheses because we want to call function. If we want to pass the function as an argument without calling it, then we need to bind it exactly as we did with our sendMessage function.

Now let's write our renderTable function. It will also return the rendered object and should have only one <div> element in it.

We know that we have 2 players and one board, so we will need 3 columns. 2 of them will render Players and one will render Board.

We can make both Board and Player external components and call them here, giving them all the data they need. But for this to work we need to import them first. So at the top of App.js file where all of your imports are, let's import Board and Player.

```
// Import components
import Player from './Player';
import Board from './Board';
```

Now we can write our renderTable.

It will have 3 columns, 2 of them will draw players, one will draw Board.

Players will need to get players info from the GSO and be able to send messages when the button is pressed.

To know what message to send on the button, they will also need the state of the GSO and we can give it to them because we save it as a state in App.js

```
// Separate function to render our game table
renderTable(){
  return (
    <div className="Table">
      <div className="Column">
        <h2 className = {this.state.gso.players[0].active ? "Active" : "Inactive"}>Player 1</h2>
        <Player data={this.state.gso.players[0]} state={this.state.gso.state} send={this.sendMessage}/>
      </div>
      <div className="Column">
        <Board data={this.state.gso.board} state={this.state.gso.state}/>
      </div>
      <div className="Column">
        <h2 className = {this.state.gso.players[1].active ? "Active" : "Inactive"}>Player 2</h2>
        <Player data={this.state.gso.players[1]} state={this.state.gso.state} send={this.sendMessage}/>
      </div>
    </div>
  )
}
```

You can see a lot of ternary IFs here, but they are really super comfortable to use in React.

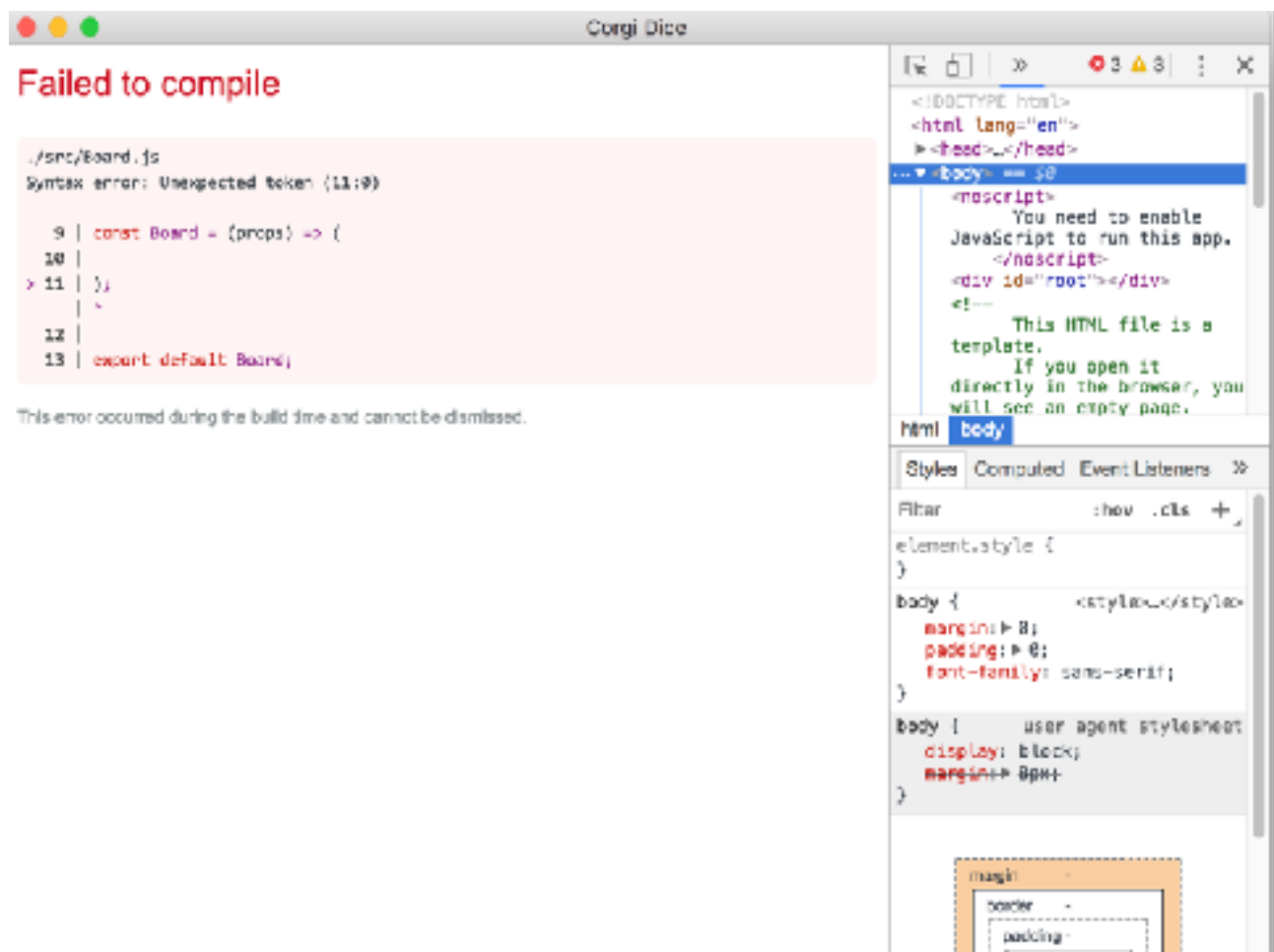
Let's look through this function to make sure we didn't forget anything.
It returns one big `<div>` component.
It has a `className` to attach CSS styles to it.
Inside it has 3 columns.

Columns 1 and 3 are player columns. In the header we use ternary if to see if this player is active. If it is - we add a class `Active` to the header. If not - we add `Inactive` class. Spoiler alert, this CSS class just makes the text for the Active player Red. No magic there.
Then we have a `Player` component that gets some additional data. We give it the player active state data, the state of the game and the function to call on buttons.

The middle column is the Board. You wrote backend, so you know what info board contains. So we give it the board from our `gso` and also the state.

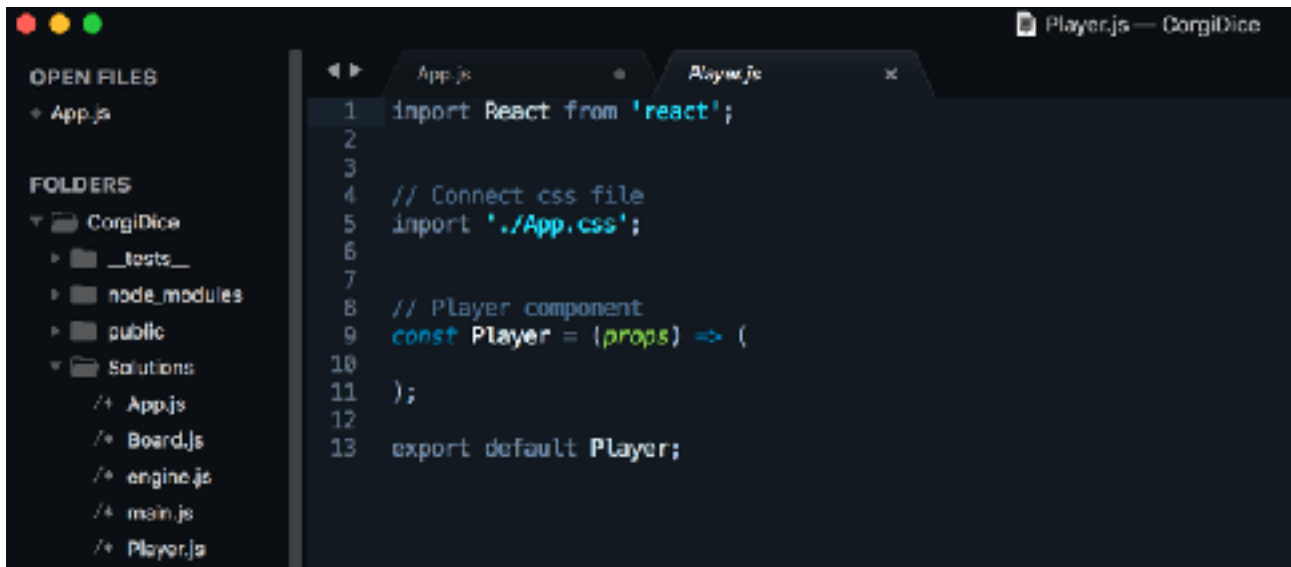
If at any point you had questions on why we keep dice and player separate - that's the answer. It's just more comfortable to draw them separately.

So, it's time to run our application. Use command `"electron ."` in your terminal and make sure the react is running in the second window. Your application won't compile, because you are missing important functions in your `Board` and `Player` components. But it should open an application window instead of browser window now.



Time to jump to our `Player.js`

Your `Player` also imports `React` and `App.css` to style elements.
But its declaration looks a bit different
Why?



We talked about smart Components earlier. Player and Board are not smart components. They don't have their own state, all they do is they react to the data that App.js gives them. To emphasise this we don't define them as a class, we define them as constant

So what our player needs to draw?

It needs to draw an avatar (active if the player is active, inactive if not).

It needs to draw the score.

It needs to draw the button.

Of all of these actions the Button drawing is the most complex one, so we can make Button another component inside Player.js. Why don't we create an external file for it? Because there's no need, no one else in the project needs access to these Buttons.

So first lets import our images. They are in images folder, so we just need to reference them:

```
// Import images for the player
import playerActive from './images/ava_active.png';
import playerInactive from './images/ava_inactive.png';
```

Now let's write our not so smart Player component.

```
// Player component
const Player = (props) => (
  <div className="Player">
    {props.data.active ? <img src={playerActive}/> : <img src={playerInactive}/> }
    <div className="Score">
      <h3>Score:</h3>
      <div>{props.data.score}</div>
    </div>
    {props.data.active ? <Buttons state={props.state} send={props.send} /> : null}
  </div>
);
```


It gets props from the parent component (props here are all the data we gave it in App.js)

It creates one div with className Player.

Then we check props.data.active. Remember, in App.js we create this element and give it `data={this.state.gso.players[0]}` So all we need to do is check if props.data.active is true and draw either playerActive avatar or playerInactive avatar.

Then we create header Score with class Score to make it Pretty and render props.data.score. We know that's exactly where GSO keeps the score for this player.

Then if the player is active we draw a Button with a state (because the text on the button depends on what is happening in GSO right now) and with the send function to send Message to GSO.

Note the `": null"` at the end of the last If. It means if the player is inactive - don't render anything.

Now we need to add Button component to the Player.js file. Again it's not a smart component, so we declare it the same way as Player, just a bit earlier. It's a good practice to declare your components before they are used.

What does our Button need to do?

It gets state from Player (who gets it from App). And it gets the function it needs to call.

So in Button we just need to check the state and send the relevant message using switch statement.

```
// Button component
const Buttons = (props) => {
  // The rendered button depends on the state
  switch(props.state){
    case "TURNSTART":
      return <button onClick={() => props.send("Deal")}>Deal</button>
    case "DEALT":
      return <button onClick={() => props.send("Throw")}>Throw</button>
    case "THROWN":
      return <button onClick={() => props.send("Count")}>Count</button>
    case "AGAIN":
      return <div>
        <button onClick={() => props.send("Again")}>Again</button>
        <button onClick={() => props.send("Change")}>Over</button>
      </div>
    case "TURNEND":
      return <button onClick={() => props.send("Change")}>Over</button>
  }
}
```

Note the syntax. OnClick result is in {} because it's javascript. And we use an anonymous function to call props.send (that contains our sendMessage() function). The benefit of it is that you don't need to bind it to the context. So it is called where it defined with no issues.

Arrow functions https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

If you run your app again, it won't compile still. Because it needs the Board!

Let's write the Board then.
Open your Board.js

First let's import all the images we need. There is a lot!

```
// Import images for dices and score
import dice from './data/dice.json'
import corgi from './images/corgi.png';
import cabbage from './images/cabbage.png';
import paws from './images/paws.png';
import corgiIcon from './images/corgiIcon.png';
import cabbageIcon from './images/cabbageIcon.png';
```

Because at this point you understand your not so smart components, this file contains empty components for you. Note you only export Board, because other components won't be referenced from outside.

```
1  import React from 'react';
2
3  // Import images for dices and score
4  import dice from './data/dice.json'
5  import corgi from './images/corgi.png';
6  import cabbage from './images/cabbage.png';
7  import paws from './images/paws.png';
8  import corgiIcon from './images/corgiIcon.png';
9  import cabbageIcon from './images/cabbageIcon.png';
10
11 // Connect css file
12 import './App.css';
13
14
15 // Component for Dice
16 const Dice = ({props}) => {
17
18 }
19
20 // Component that analyses the state and calls relevant function to display the elements.
21 const DiceElement = ({props}) => {
22
23 }
24
25 // Main component of the board.
26 const Board = ({props}) => {
27
28 };
29
30 export default Board;
```

Why do we need so many components? To keep the code clean. Before the game starts, our board will display the rules. After the game starts - the DiceElement. DiceElement will display either dice that is not thrown (just the color), dice that are thrown (color and icon) and results (just icon) depending on state.

Let's start with the Board component.

```
// Main component of the board.
const Board = (props) => (
  <div className="Board">
    {props.state !== "TURNSTART" ?
      <DiceElement state={props.state} data={props.data}/> :
      <p>Get corgis, avoid cabbages!</p>}
    </div>
);
```

If our state that we got from App is not TURNSTART, then we are going to display DiceElement that will get the state and data (board). Else we will display the rules.

```
// Component that analyses the state and calls relevant function to display the elements.
const DiceElement = (props) => {
  switch(props.state){
    case "DEALT":
      return renderEmptyDice(props.data.dealt);
    case "THROWN":
      return renderFilledDice(props.data.thrown);
    case "AGAIN":
      return renderCurrentResult(props.data.hand);
    case "TURNEND":
      return renderCurrentResult(props.data.hand);
  }
}
```

Or DiceElement will grab the state and the data(board) and will call the relevant function giving it the relevant data. We will write this functions after we are done with Dice element.

What about our Dice? It's a box with a background and in some cases - an image. So we will give it style and side through props each time we draw it. Then if the side is corgi we will draw a corgi image, if it's a cabbage - cabbage image and so on.

```
// Component for Dice
const Dice = (props) => (
  <div className="Dice" style={{backgroundColor: props.color}}>
    {props.side=="corgi" ? <img src={corgi}/> : null}
    {props.side=="cabbage" ? <img src={cabbage}/> : null}
    {props.side=="paws" ? <img src={paws}/> : null}
  </div>
)
```

Ok, now we need to write 3 different functions for 3 different states of our Dice, let's talk about differences between them.

renderEmptyDice will need to get the color from dice and set it as a background. It doesn't need to display the side because we don't have the sides at the DEALT state.

renderFilledDice needs to display the background and the side, because we have them at THROWN.

renderCurrentResult needs to grab all the sides in hand and display them all so the Player can see the result.

All the functions need to return one element (don't forget about it)

So, EmptyDice will draw 3 Dice elements. It will give them the color (remember our dice.json - all the dice have color and we planned it in a way that it is a simple color that can be used as a CSS background) and null where the side should be, because we don't have sides yet.

// Separate function to render our dice before they have sides. Uses Dice component.

```
function renderEmptyDice(dealt){
  return (
    <div>
      <h2>Dealing dice</h2>
      <Dice color={dice[dealt[0]].color} side={null}/>
      <Dice color={dice[dealt[1]].color} side={null}/>
      <Dice color={dice[dealt[2]].color} side={null}/>
    </div>
  )
}
```

// Separate function to render our dice with sides. Uses Dice component.

```
function renderFilledDice(thrown){
  return (
    <div>
      <h2>Throwing dice</h2>
      <Dice color={dice[thrown[0].name].color} side={thrown[0].side}/>
      <Dice color={dice[thrown[1].name].color} side={thrown[1].side}/>
      <Dice color={dice[thrown[2].name].color} side={thrown[2].side}/>
    </div>
  )
}
```

This function does almost the same thing. We just change the header and give dice their sides.

// Separate function to render the results for the player within a turn.

```
function renderCurrentResult(hand){
  return (
    <div>
      <h2>Turn Results</h2>
      {hand.map(x => <div key={x.name} className="Dicelcon">
        <img src={x.side=="corgi" ? corgilcon : cabbagelcon}/> </div>)}
    </div>
  )
}
```

The last function is a bit different. It doesn't draw dice, because we only need sides and we can't know for sure how many dice user has at this point. So we map the data using the function that draws a div element and either corgi or cabbage icon. Note that if you map the data this way - building elements inside a loop - React requires each element to have a key. And that's what we do - we assign the dice name as a key to each dice (we can guarantee all of our dice have original names - we made sure of it on backend).

Run your application now. And Play!

Follow up

As you can see, our game is pretty simple and there are several conditions that we don't take into consideration.

1. When does the game end? We don't have the max score, so players can play as long as they like. If you want to do some exploration on your own - think on how you can cap the number of turns on backend.
2. What happens if all 13 dice are used? That's a very interesting edge case. It can happen that the player is super lucky and he adds more and more dice to their hand. The rules of the game says that if there are no more dice to add in dealt - dice will return to the available pile, but the score will continue. How can you implement it on backend?
3. And our game doesn't really look good, so can you make it look better by editing CSS styles or creating more cool components?

Also to accomodate the workshop format I presented functions in their complete way, especially on frontend. That's not a normal way to write functions and test them. Instead of trying to write all the components and then run the app, you would write elements and function one by one, debugging as you go and expanding where needed. Without me telling you to write a separate function for drawing a Table, you would probably try to write it in general App.js render and then move to the separate function and so on.

The goal of this workshop is not to teach you how to debug your projects. It is to give you an overflow of plan - test - development cycle.

The main idea here is that you need to create the architecture that is flexible enough to change, but has some fundamental principles that you're going to build on.

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

Thank you!