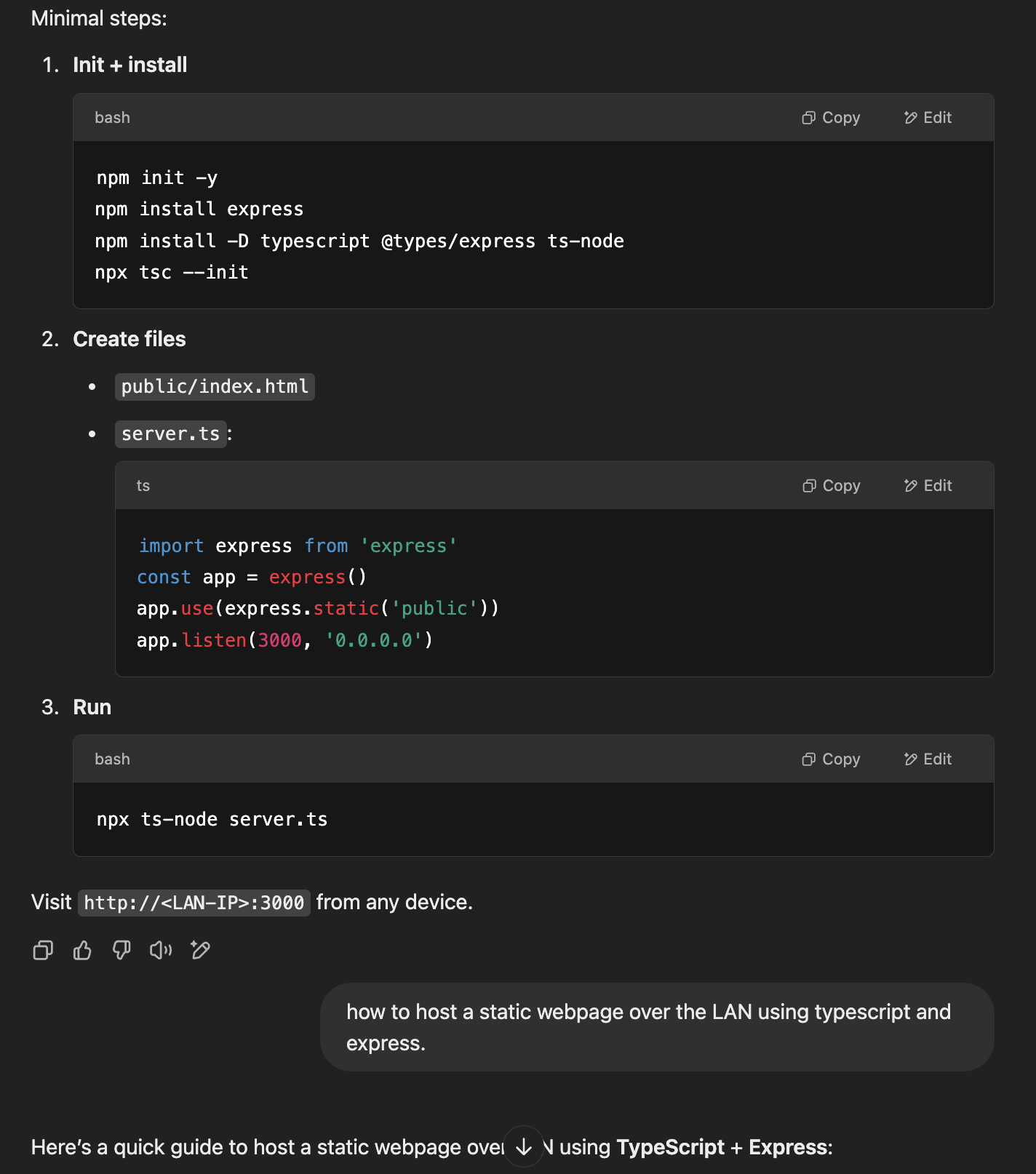
**ChatGPT Prompts:**

**SERVER CREATION:**

**Prompt:  
how to host a static webpage over the LAN using typescript and express.**

**Use less words.**

**Output:**

****

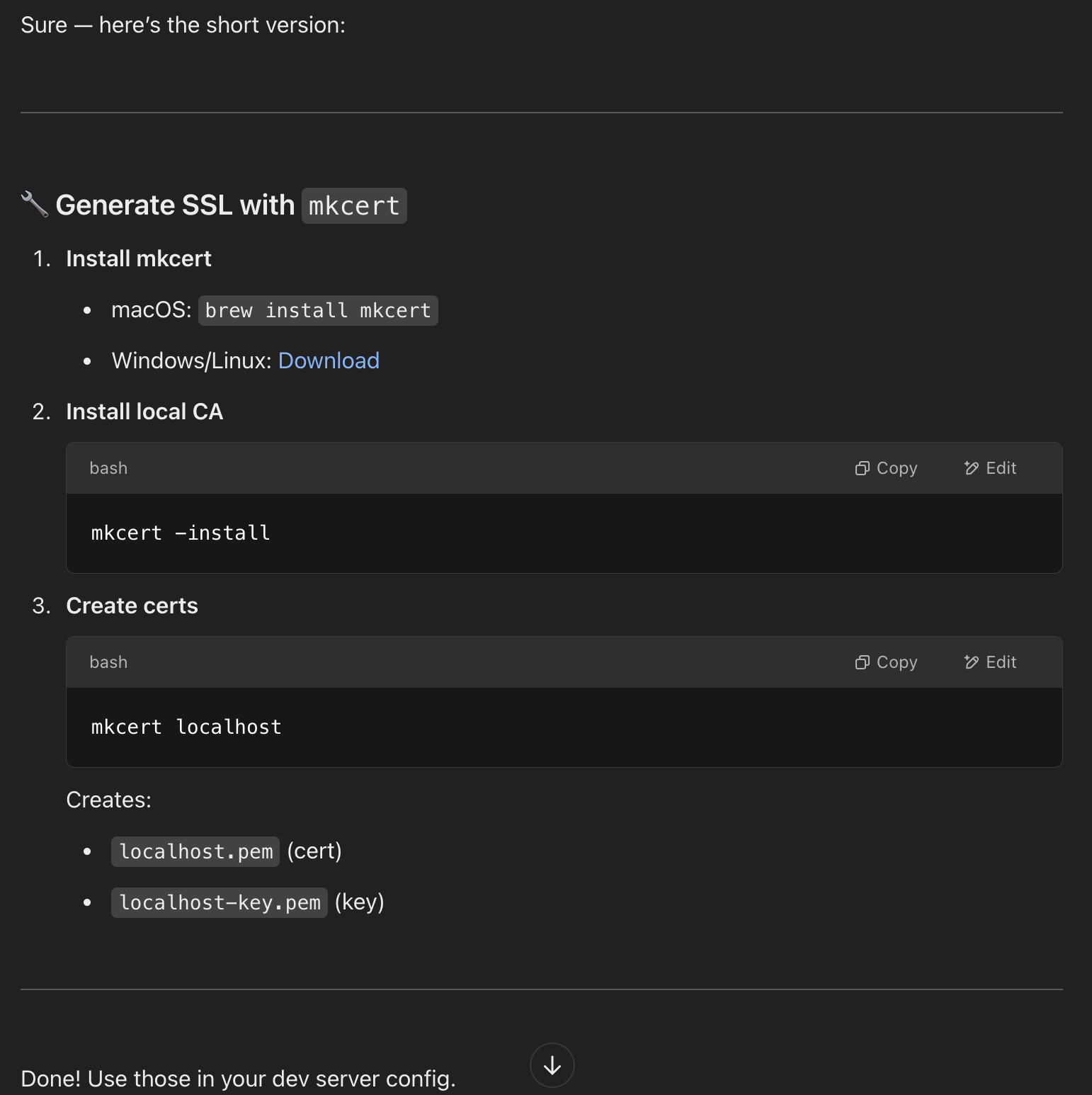
**SSL GENERATION:**

**Prompt:**

**how to generate a self signed SSL using mkcert**

**Use less words**

**Output:**

****

**CONNECTION MANAGEMENT:**

**Prompt:**

| import { app, BrowserWindow } from 'electron';  import path from 'path';  import { isDev } from './util.js';  import { getPreloadPath } from './pathResolver.js';  import { serveControllerApp } from './server.js';  import { ControllerLayout } from './controllers/ControllerLayout.js';  import { KeyboardTarget, Mapping, MouseClickTarget } from '../../types.js';  import {Key} from '@nut-tree-fork/nut-js'  import { ButtonInput } from './controller-inputs/ButtonInput.js';  const mappings: Mapping[] = [  {  id: 'a',  source: 'button',  target: {type: 'keyboard', keybinding: [Key.Num4]}  },  {  id: 'b',  source: 'button',  target: {type: 'keyboard', keybinding: [Key.Num5]}  },  {  id: 'x',  source: 'button',  target: {type: 'keyboard', keybinding: [Key.Num8]}  },  {  id: 'y',  source: 'button',  target: {type: 'keyboard', keybinding: [Key.Num9]}  },  {  id: 'up',  source: 'button',  target: {type: 'keyboard', keybinding: [Key.W]}  },  {  id: 'down',  source: 'button',  target: {type: 'keyboard', keybinding: [Key.S]}  },  {  id: 'left',  source: 'button',  target: {type: 'keyboard', keybinding: [Key.A]}  },  {  id: 'right',  source: 'button',  target: {type: 'keyboard', keybinding: [Key.D]}  }  ]  const maxConnections = 1;  const connectedClients = [];  const initializeController = (controller: ControllerLayout) => {  for (const mapping of mappings) {  // console.log(mapping);  if (mapping.source === 'button') {  if (mapping.target.type === 'keyboard') {  const buttonInput = new ButtonInput(mapping.id, mapping.target as KeyboardTarget);  controller.addInput(buttonInput);  } else if (mapping.target.type === 'mouseClick') {  const buttonInput = new ButtonInput(mapping.id, mapping.target as MouseClickTarget);  controller.addInput(buttonInput);  }  }  }  }  app.on("ready", async () => {  const mainWindow = new BrowserWindow({  webPreferences: {  preload: getPreloadPath(),  }  });  if (isDev()) {  const port = process.env.LOCAL\_PORT;  if (port) {  mainWindow.loadURL(http://localhost:${port})  } else {  mainWindow.loadURL('http://localhost:7777')  }  } else {  mainWindow.loadFile(path.join(app.getAppPath(), "/dist-react/index.html"))  }  const serverUrl = await serveControllerApp();  const [server, url] = serverUrl  const controllerLayout = new ControllerLayout("LayoutA");  mainWindow.webContents.on('did-finish-load', () => {  if (mainWindow) {  mainWindow.webContents.send('setControllerUrl', url);  }  });  //TODO This block has no purpose yet, it is just to log live data from the controller  if (server) {  server.on('connection', async (socket) => {  console.log('A client connected');  // Request client to send device info  socket.emit('request-device-info');  socket.on('device-info', async (data: {deviceName: string}) =>{  console.log("Device Type: ", data.deviceName);  connectedClients.push(data.deviceName);  })  initializeController(controllerLayout)  socket.on('joystick-move', (data) => {  console.log('Joystick moved:', data);  });  socket.on('button', async (data: {button: string, pressed: boolean}) =>{  // console.log(controllerLayout.inputs.get('x'));  console.log(data.pressed);  // console.log(controllerLayout.inputs.get(data.buttonId)?.getMappingTarget())  await controllerLayout.inputs.get(data.button)?.handleInput(data.pressed)  })  socket.on('disconnect', () => {  console.log('A client disconnected');  });  });  server.on('connect\_error', (error) => {  console.log(error)  });  // Handle the 'error' event from the spawn itself (e.g., if 'node' or the script isn't found)  server.once('error', (error) => {  console.error(Failed to start server process: ${error.message});  });  // Handle when the server process exits  server.once('exit', (code, signal) => {  if (code !== 0) {  console.log(Server process exited with code ${code});  } else {  console.log('Server process exited successfully');  }  //e.g. SIGKILL or SIGTERM  if (signal) {  console.log(Server process was terminated by signal: ${signal});  }  });  server.on('close', (code) => {  console.log(Server process exited with code ${code});  });  }  })  edit this code so that if the connectedClients list length is equal to maxConnections, we don't allow the new client to connect. The client that is currently connected is fine |
| --- |

**Output:**

| Below is the updated code. We’ve added a check at the beginning of the connection event that verifies if the number of connected clients has reached the maximum allowed. If so, the new client is informed (via an emitted event) and then disconnected immediately. Optionally, you might also consider removing clients from the list upon disconnect so that new connections can be allowed once a client disconnects.  typescript  Copy  import { app, BrowserWindow } from 'electron';  import path from 'path';  import { isDev } from './util.js';  import { getPreloadPath } from './pathResolver.js';  import { serveControllerApp } from './server.js';  import { ControllerLayout } from './controllers/ControllerLayout.js';  import { KeyboardTarget, Mapping, MouseClickTarget } from '../../types.js';  import { Key } from '@nut-tree-fork/nut-js';  import { ButtonInput } from './controller-inputs/ButtonInput.js';  const mappings: Mapping[] = [  {  id: 'a',  source: 'button',  target: { type: 'keyboard', keybinding: [Key.Num4] }  },  {  id: 'b',  source: 'button',  target: { type: 'keyboard', keybinding: [Key.Num5] }  },  {  id: 'x',  source: 'button',  target: { type: 'keyboard', keybinding: [Key.Num8] }  },  {  id: 'y',  source: 'button',  target: { type: 'keyboard', keybinding: [Key.Num9] }  },  {  id: 'up',  source: 'button',  target: { type: 'keyboard', keybinding: [Key.W] }  },  {  id: 'down',  source: 'button',  target: { type: 'keyboard', keybinding: [Key.S] }  },  {  id: 'left',  source: 'button',  target: { type: 'keyboard', keybinding: [Key.A] }  },  {  id: 'right',  source: 'button',  target: { type: 'keyboard', keybinding: [Key.D] }  }  ];  const maxConnections = 1;  const connectedClients: string[] = [];  const initializeController = (controller: ControllerLayout) => {  for (const mapping of mappings) {  if (mapping.source === 'button') {  if (mapping.target.type === 'keyboard') {  const buttonInput = new ButtonInput(mapping.id, mapping.target as KeyboardTarget);  controller.addInput(buttonInput);  } else if (mapping.target.type === 'mouseClick') {  const buttonInput = new ButtonInput(mapping.id, mapping.target as MouseClickTarget);  controller.addInput(buttonInput);  }  }  }  };  app.on("ready", async () => {  const mainWindow = new BrowserWindow({  webPreferences: {  preload: getPreloadPath(),  }  });  if (isDev()) {  const port = process.env.LOCAL\_PORT;  if (port) {  mainWindow.loadURL(`http://localhost:${port}`);  } else {  mainWindow.loadURL('http://localhost:7777');  }  } else {  mainWindow.loadFile(path.join(app.getAppPath(), "/dist-react/index.html"));  }  const serverUrl = await serveControllerApp();  const [server, url] = serverUrl;  const controllerLayout = new ControllerLayout("LayoutA");  mainWindow.webContents.on('did-finish-load', () => {  if (mainWindow) {  mainWindow.webContents.send('setControllerUrl', url);  }  });  if (server) {  server.on('connection', async (socket) => {  // Check if maximum connections reached  if (connectedClients.length >= maxConnections) {  console.log('Maximum connections reached, rejecting new connection.');  socket.emit('maxConnectionsReached', { message: 'Maximum number of connections reached.' });  socket.disconnect(true);  return;  }    console.log('A client connected');  // Request client to send device info  socket.emit('request-device-info');  // Optional: track the client device name so you can remove it on disconnect.  let clientDeviceName: string | null = null;  socket.on('device-info', async (data: { deviceName: string }) => {  console.log("Device Type: ", data.deviceName);  clientDeviceName = data.deviceName;  connectedClients.push(data.deviceName);  });  initializeController(controllerLayout);  socket.on('joystick-move', (data) => {  console.log('Joystick moved:', data);  });  socket.on('button', async (data: { button: string, pressed: boolean }) => {  console.log(data.pressed);  await controllerLayout.inputs.get(data.button)?.handleInput(data.pressed);  });  socket.on('disconnect', () => {  console.log('A client disconnected');  // Optional: remove client from the connectedClients list on disconnect  if (clientDeviceName) {  const index = connectedClients.indexOf(clientDeviceName);  if (index !== -1) {  connectedClients.splice(index, 1);  }  }  });  });  server.on('connect\_error', (error) => {  console.log(error);  });  server.once('error', (error) => {  console.error(`Failed to start server process: ${error.message}`);  });  server.once('exit', (code, signal) => {  if (code !== 0) {  console.log(`Server process exited with code ${code}`);  } else {  console.log('Server process exited successfully');  }  if (signal) {  console.log(`Server process was terminated by signal: ${signal}`);  }  });  server.on('close', (code) => {  console.log(`Server process exited with code ${code}`);  });  }  });  Explanation  Connection Check:  Right at the start of the server.on('connection', ...) block, we check if connectedClients.length is greater than or equal to maxConnections. If it is, the new client is immediately informed (using socket.emit('maxConnectionsReached', { ... })) and disconnected.  Client Removal (Optional):  On the disconnect event, we remove the client's device name from connectedClients, which allows a new connection once an existing client disconnects.  This update ensures that only one client (as defined by maxConnections) can remain connected at any time, while existing connections remain unaffected. |
| --- |

**CONVERT MAPPINGS:**

**Prompt:**

| [  {  "id": "a",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.Num4"] }  },  {  "id": "b",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.Num5"] }  },  {  "id": "x",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.Num8"] }  },  {  "id": "y",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.Num9"] }  },  {  "id": "up",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.W"] }  },  {  "id": "down",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.S"] }  },  {  "id": "left",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.A"] }  },  {  "id": "right",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.D"] }  },  {  "id": "start",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.Home"] }  },  {  "id": "select",  "source": "button",  "target": { "type": "mouseClick", "mouseClick": "Button.LEFT" }  },  {  "id": "right-analog",  "source": "analog",  "target": { "type": "mouseMotion", "sensitivity": 15 }  },  {  "id": "left-analog",  "source": "analog",  "target": {  "type": "analogKeyboard",  "positiveX": ["Key.D"],  "positiveY": ["Key.W"],  "negativeX": ["Key.A"],  "negativeY": ["Key.S"]  }  },  {  "id": "accelerometer",  "source": "motion",  "target": { "type": "mouseMotion", "sensitivity": 25 }  }  ]  /\*\*  \* The {@link Key} enum represents keys of a standard 105 key US layout keyboard  \*/  export declare enum Key {  Escape = 0,  F1 = 1,  F2 = 2,  F3 = 3,  F4 = 4,  F5 = 5,  F6 = 6,  F7 = 7,  F8 = 8,  F9 = 9,  F10 = 10,  F11 = 11,  F12 = 12,  F13 = 13,  F14 = 14,  F15 = 15,  F16 = 16,  F17 = 17,  F18 = 18,  F19 = 19,  F20 = 20,  F21 = 21,  F22 = 22,  F23 = 23,  F24 = 24,  Print = 25,  ScrollLock = 26,  Pause = 27,  Grave = 28,  Num1 = 29,  Num2 = 30,  Num3 = 31,  Num4 = 32,  Num5 = 33,  Num6 = 34,  Num7 = 35,  Num8 = 36,  Num9 = 37,  Num0 = 38,  Minus = 39,  Equal = 40,  Backspace = 41,  Insert = 42,  Home = 43,  PageUp = 44,  NumLock = 45,  NumPadEqual = 46,  Divide = 47,  Multiply = 48,  Subtract = 49,  Tab = 50,  Q = 51,  W = 52,  E = 53,  R = 54,  T = 55,  Y = 56,  U = 57,  I = 58,  O = 59,  P = 60,  LeftBracket = 61,  RightBracket = 62,  Backslash = 63,  Delete = 64,  End = 65,  PageDown = 66,  NumPad7 = 67,  NumPad8 = 68,  NumPad9 = 69,  Add = 70,  CapsLock = 71,  A = 72,  S = 73,  D = 74,  F = 75,  G = 76,  H = 77,  J = 78,  K = 79,  L = 80,  Semicolon = 81,  Quote = 82,  Return = 83,  NumPad4 = 84,  NumPad5 = 85,  NumPad6 = 86,  LeftShift = 87,  Z = 88,  X = 89,  C = 90,  V = 91,  B = 92,  N = 93,  M = 94,  Comma = 95,  Period = 96,  Slash = 97,  RightShift = 98,  Up = 99,  NumPad1 = 100,  NumPad2 = 101,  NumPad3 = 102,  Enter = 103,  LeftControl = 104,  LeftSuper = 105,  LeftWin = 106,  LeftCmd = 107,  LeftAlt = 108,  LeftMeta = 109,  RightControl = 110,  RightSuper = 111,  RightWin = 112,  RightAlt = 113,  RightCmd = 114,  RightMeta = 115,  Space = 116,  Menu = 117,  Fn = 118,  Left = 119,  Down = 120,  Right = 121,  NumPad0 = 122,  Decimal = 123,  Clear = 124,  AudioMute = 125,  AudioVolDown = 126,  AudioVolUp = 127,  AudioPlay = 128,  AudioStop = 129,  AudioPause = 130,  AudioPrev = 131,  AudioNext = 132,  AudioRewind = 133,  AudioForward = 134,  AudioRepeat = 135,  AudioRandom = 136  }  /\*\*  \* {@link Button} enum represents clickable buttons of a mouse  \*/  export declare enum Button {  LEFT = 0,  MIDDLE = 1,  RIGHT = 2  }  How do I convert the parsed JSON strings into its proper enum values? |
| --- |

**Output:**

| You can convert the JSON string values (like "Key.Num4") to their corresponding enum values by extracting the part after the dot and then using that as a key to index the enum. For example, given your Key enum, the string "Key.Num4" should be converted into Key.Num4, which equals 32.  Here’s one way to do it in TypeScript:  ts  Copy  // Suppose your JSON has been parsed into a config array.  const config = [  {  id: "a",  source: "button",  target: { type: "keyboard", keybinding: ["Key.Num4"] }  },  // ... other mappings  ];  // Function to convert a string like "Key.Num4" into the actual enum value.  function convertKeyString(keyString: string): number {  // Remove the "Key." prefix  const parts = keyString.split('.');  if (parts.length !== 2 || parts[0] !== 'Key') {  throw new Error('Invalid key format');  }  const keyName = parts[1];  // Now lookup the enum value using the key name.  return Key[keyName as keyof typeof Key];  }  // Example usage for a single configuration element:  const keyboardMapping = config.find(mapping => mapping.target.type === "keyboard");  if (keyboardMapping) {  const keyStr = keyboardMapping.target.keybinding[0]; // e.g., "Key.Num4"  const enumValue = convertKeyString(keyStr);  console.log(enumValue); // Logs 32, which is the value for Key.Num4  }  The idea is to:  Extract the Enum Name: Split the string by the dot to separate the enum namespace ("Key") from the member name ("Num4").  Index the Enum: Use the member name to index into the enum. For instance, Key["Num4"] returns the corresponding numeric value (32 in this case).  You can apply a similar approach to convert any other enum strings (such as "Button.LEFT") by adjusting the extraction logic to use the correct enum.  This way, you ensure that the JSON strings are mapped to the proper enum values defined in your code. |
| --- |

**Prompt:**

| how do I run it for every mapping in config? |
| --- |

**Output:**

| You can loop over every mapping in your config and convert the enum strings to their numeric values. For example, you can define a helper function that converts a single mapping, then iterate over the entire array. Here's an example in TypeScript:  ts  Copy  // Assume your enums are defined like this:  enum Key {  Escape = 0,  F1 = 1,  F2 = 2,  // ... other keys ...  Num4 = 32,  Num5 = 33,  Num8 = 36,  Num9 = 37,  W = 52,  A = 72,  S = 73,  D = 74,  Home = 43,  // ... etc ...  }  enum Button {  LEFT = 0,  MIDDLE = 1,  RIGHT = 2  }  // Conversion functions for enum strings.  function convertKeyString(keyString: string): number {  const parts = keyString.split('.');  if (parts.length !== 2 || parts[0] !== 'Key') {  throw new Error('Invalid key format: ' + keyString);  }  const keyName = parts[1];  return Key[keyName as keyof typeof Key];  }  function convertButtonString(buttonString: string): number {  const parts = buttonString.split('.');  if (parts.length !== 2 || parts[0] !== 'Button') {  throw new Error('Invalid button format: ' + buttonString);  }  const buttonName = parts[1];  return Button[buttonName as keyof typeof Button];  }  // Example configuration array.  const config = [  {  id: "a",  source: "button",  target: { type: "keyboard", keybinding: ["Key.Num4"] }  },  {  id: "b",  source: "button",  target: { type: "keyboard", keybinding: ["Key.Num5"] }  },  {  id: "x",  source: "button",  target: { type: "keyboard", keybinding: ["Key.Num8"] }  },  {  id: "y",  source: "button",  target: { type: "keyboard", keybinding: ["Key.Num9"] }  },  {  id: "up",  source: "button",  target: { type: "keyboard", keybinding: ["Key.W"] }  },  {  id: "down",  source: "button",  target: { type: "keyboard", keybinding: ["Key.S"] }  },  {  id: "left",  source: "button",  target: { type: "keyboard", keybinding: ["Key.A"] }  },  {  id: "right",  source: "button",  target: { type: "keyboard", keybinding: ["Key.D"] }  },  {  id: "start",  source: "button",  target: { type: "keyboard", keybinding: ["Key.Home"] }  },  {  id: "select",  source: "button",  target: { type: "mouseClick", mouseClick: "Button.LEFT" }  },  {  id: "right-analog",  source: "analog",  target: { type: "mouseMotion", sensitivity: 15 }  },  {  id: "left-analog",  source: "analog",  target: {  type: "analogKeyboard",  positiveX: ["Key.D"],  positiveY: ["Key.W"],  negativeX: ["Key.A"],  negativeY: ["Key.S"]  }  },  {  id: "accelerometer",  source: "motion",  target: { type: "mouseMotion", sensitivity: 25 }  }  ];  // Function to process each mapping.  function convertMapping(mapping: any): void {  // For "keyboard" type, convert the keybinding array.  if (mapping.target.type === "keyboard" && mapping.target.keybinding) {  mapping.target.keybinding = mapping.target.keybinding.map(convertKeyString);  }    // For "analogKeyboard" type, convert each directional key array.  if (mapping.target.type === "analogKeyboard") {  ["positiveX", "positiveY", "negativeX", "negativeY"].forEach(prop => {  if (mapping.target[prop]) {  mapping.target[prop] = mapping.target[prop].map(convertKeyString);  }  });  }    // For "mouseClick" type, convert the mouseClick property if needed.  if (mapping.target.type === "mouseClick" && mapping.target.mouseClick && mapping.target.mouseClick.startsWith("Button.")) {  mapping.target.mouseClick = convertButtonString(mapping.target.mouseClick);  }  }  // Loop through all mappings in config.  config.forEach(convertMapping);  // Now each mapping in config has its enum strings converted to numeric values.  console.log(config);  Explanation  Conversion Functions:  convertKeyString extracts the key name (e.g., "Num4" from "Key.Num4") and looks up its corresponding value in the Key enum. A similar function, convertButtonString, does the same for mouse buttons.  Iterating Over Config:  The convertMapping function checks the type of each mapping's target and converts:  The keybinding array for "keyboard" mappings.  Each directional key array (positiveX, positiveY, etc.) for "analogKeyboard" mappings.  The mouseClick property for "mouseClick" mappings.  Processing All Mappings:  Using config.forEach(convertMapping), you run the conversion logic on every mapping in your configuration array.  This approach ensures that every mapping gets the proper enum value, so you can safely use them in your application. |
| --- |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**REVERT MAPPINGS:**

**Prompt:**

| console.log("Sent Mappings:", data);  try {  // Convert the data to a formatted JSON string.  const jsonString = JSON.stringify(data, null, 2);  // Write the JSON string to the file.  fs.writeFileSync(mappingsLayoutAPath, jsonString, 'utf8');  console.log("Mappings saved successfully.");  } catch (error) {  console.error("Error saving mappings:", error);  }  This code currently saves the JSON file so that it looks like below:  [  {  "id": "a",  "source": "button",  "target": {  "type": "keyboard",  "keybinding": [  33  ]  }  },  {  "id": "b",  "source": "button",  "target": {  "type": "keyboard",  "keybinding": [  32  ]  }  },  {  "id": "x",  "source": "button",  "target": {  "type": "keyboard",  "keybinding": [  36  ]  }  },  {  "id": "y",  "source": "button",  "target": {  "type": "keyboard",  "keybinding": [  37  ]  }  },  {  "id": "up",  "source": "button",  "target": {  "type": "keyboard",  "keybinding": [  52  ]  }  },  {  "id": "down",  "source": "button",  "target": {  "type": "keyboard",  "keybinding": [  73  ]  }  },  {  "id": "left",  "source": "button",  "target": {  "type": "keyboard",  "keybinding": [  72  ]  }  },  {  "id": "right",  "source": "button",  "target": {  "type": "keyboard",  "keybinding": [  74  ]  }  },  {  "id": "start",  "source": "button",  "target": {  "type": "keyboard",  "keybinding": [  43  ]  }  },  {  "id": "select",  "source": "button",  "target": {  "type": "mouseClick",  "mouseClick": 0  }  },  {  "id": "right-analog",  "source": "analog",  "target": {  "type": "mouseMotion",  "sensitivity": 15  }  },  {  "id": "left-analog",  "source": "analog",  "target": {  "type": "analogKeyboard",  "positiveX": [  74  ],  "positiveY": [  52  ],  "negativeX": [  72  ],  "negativeY": [  73  ]  }  },  {  "id": "accelerometer",  "source": "motion",  "target": {  "type": "mouseMotion",  "sensitivity": 25  }  }  ]  I want the saved code to be in the format below though:  [  {  "id": "a",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.Num4"] }  },  {  "id": "b",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.Num5"] }  },  {  "id": "x",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.Num8"] }  },  {  "id": "y",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.Num9"] }  },  {  "id": "up",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.W"] }  },  {  "id": "down",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.S"] }  },  {  "id": "left",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.A"] }  },  {  "id": "right",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.D"] }  },  {  "id": "start",  "source": "button",  "target": { "type": "keyboard", "keybinding": ["Key.Home"] }  },  {  "id": "select",  "source": "button",  "target": { "type": "mouseClick", "mouseClick": "Button.LEFT" }  },  {  "id": "right-analog",  "source": "analog",  "target": { "type": "mouseMotion", "sensitivity": 15 }  },  {  "id": "left-analog",  "source": "analog",  "target": {  "type": "analogKeyboard",  "positiveX": ["Key.D"],  "positiveY": ["Key.W"],  "negativeX": ["Key.A"],  "negativeY": ["Key.S"]  }  },  {  "id": "accelerometer",  "source": "motion",  "target": { "type": "mouseMotion", "sensitivity": 25 }  }  ]  import os from 'os';  import { EventPayloadMapping } from '../types.js';  import { ipcMain } from 'electron';  import { Button, Key } from '@nut-tree-fork/nut-js';  export function isDev(): boolean {  return process.env.NODE\_ENV === "development";  }  // Get private IPv4 address for LAN accessibility  export function findPrivateIp(): string | null {  const interfaces = os.networkInterfaces();  for (const devName in interfaces) {  if (interfaces[devName]) {  for (let i = 0; i < interfaces[devName].length; i++) {  const iface = interfaces[devName][i];  if (iface.family === 'IPv4' && !iface.internal) {  return iface.address;  }  }  }  }  return null;  }  // Wrapper for ipcMain.Handle to make it typesafe.  // by ensuring the handler returns a type that matches one of the EventPayloadMapping keys  export function ipcHandle<Key extends keyof EventPayloadMapping & string>(  key: Key,  handler: () => EventPayloadMapping[Key]  ) {  ipcMain.handle(key, () => handler());  }  // Conversion functions for enum strings.  function convertKeyString(keyString: string): number {  const parts = keyString.split('.');  if (parts.length !== 2 || parts[0] !== 'Key') {  throw new Error('Invalid key format: ' + keyString);  }  const keyName = parts[1];  return Key[keyName as keyof typeof Key];  }    function convertButtonString(buttonString: string): number {  const parts = buttonString.split('.');  if (parts.length !== 2 || parts[0] !== 'Button') {  throw new Error('Invalid button format: ' + buttonString);  }  const buttonName = parts[1];  return Button[buttonName as keyof typeof Button];  }  // Function to process each mapping.  export function convertMapping(mapping: any): void {  // For "keyboard" type, convert the keybinding array.  if (mapping.target.type === "keyboard" && mapping.target.keybinding) {  mapping.target.keybinding = mapping.target.keybinding.map(convertKeyString);  }  // For "analogKeyboard" type, convert each directional key array.  if (mapping.target.type === "analogKeyboard") {  ["positiveX", "positiveY", "negativeX", "negativeY"].forEach(prop => {  if (mapping.target[prop]) {  mapping.target[prop] = mapping.target[prop].map(convertKeyString);  }  });  }  // For "mouseClick" type, convert the mouseClick property if needed.  if (mapping.target.type === "mouseClick" && mapping.target.mouseClick && mapping.target.mouseClick.startsWith("Button.")) {  mapping.target.mouseClick = convertButtonString(mapping.target.mouseClick);  }  }  here are some util functions if needed |
| --- |

**Output:**

| You can achieve this by converting the numeric values back into their string representations before calling JSON.stringify. In other words, you’ll need to “revert” the conversion that was applied when processing the data. One way to do this is to add functions that take a number (for keys or buttons) and look up the corresponding enum name (for example, from the Key or Button enum).  Below is an example of how you might implement this:  js  Copy  import fs from 'fs';  import { Button, Key } from '@nut-tree-fork/nut-js';  // Helper function to convert a key code back into its enum string (e.g., 33 -> "Key.Num4")  function getKeyString(keyCode) {  // Object.keys returns both string keys for enum names (and possibly reverse mappings)  // so we filter to keys that map to a number.  const keyNames = Object.keys(Key).filter(name => typeof Key[name] === 'number');  for (const name of keyNames) {  if (Key[name] === keyCode) {  return `Key.${name}`;  }  }  return keyCode;  }  // Helper function to convert a button code back into its enum string (e.g., 0 -> "Button.LEFT")  function getButtonString(buttonCode) {  const buttonNames = Object.keys(Button).filter(name => typeof Button[name] === 'number');  for (const name of buttonNames) {  if (Button[name] === buttonCode) {  return `Button.${name}`;  }  }  return buttonCode;  }  // Function that “reverts” the mapping conversion for each mapping object.  function revertMapping(mapping) {  // For "keyboard" type mappings, convert the keybinding array numbers to strings.  if (mapping.target.type === "keyboard" && mapping.target.keybinding) {  mapping.target.keybinding = mapping.target.keybinding.map(getKeyString);  }  // For "analogKeyboard" type, convert each directional key array.  if (mapping.target.type === "analogKeyboard") {  ["positiveX", "positiveY", "negativeX", "negativeY"].forEach(prop => {  if (mapping.target[prop]) {  mapping.target[prop] = mapping.target[prop].map(getKeyString);  }  });  }  // For "mouseClick" type, convert the mouseClick property if it's a number.  if (mapping.target.type === "mouseClick" && typeof mapping.target.mouseClick === "number") {  mapping.target.mouseClick = getButtonString(mapping.target.mouseClick);  }  }  // Assume "data" is your mappings array. Convert each mapping back to the desired format.  data.forEach(revertMapping);  try {  // Convert the updated data to a formatted JSON string with 4-space indentation.  const jsonString = JSON.stringify(data, null, 4);  // Write the JSON string to the file.  fs.writeFileSync(mappingsLayoutAPath, jsonString, 'utf8');  console.log("Mappings saved successfully.");  } catch (error) {  console.error("Error saving mappings:", error);  }  Explanation  Conversion Functions  getKeyString: Loops through the keys in the Key enum to find a matching numeric value, then returns the formatted string (e.g., "Key.Num4").  getButtonString: Works similarly for the Button enum, returning strings like "Button.LEFT".  Reverting the Mapping  The revertMapping function checks the type of mapping:  For "keyboard", it maps the numeric key codes back to their string representations.  For "analogKeyboard", it does this for all directional key arrays.  For "mouseClick", it converts the numeric code back to the corresponding button string.  Saving the JSON  After processing the entire mappings array, the code uses JSON.stringify with an indentation of 4 spaces to produce nicely formatted JSON. Then it writes this JSON string to the file using fs.writeFileSync.  This approach ensures that the saved file uses string representations (like "Key.Num4" and "Button.LEFT") instead of the numeric values, matching the desired output format. |
| --- |

**Prompt:**

| this is typescript so you need to define types or else I get the errors below:  src/electron/util.ts:77:23 - error TS7006: Parameter 'keyCode' implicitly has an 'any' type.  77 function getKeyString(keyCode) {  ~~~~~~~  src/electron/util.ts:80:65 - error TS7015: Element implicitly has an 'any' type because index expression is not of type 'number'.  80 const keyNames = Object.keys(Key).filter(name => typeof Key[name] === 'number');  ~~~~  src/electron/util.ts:82:15 - error TS7015: Element implicitly has an 'any' type because index expression is not of type 'number'.  82 if (Key[name] === keyCode) {  ~~~~  src/electron/util.ts:90:28 - error TS7006: Parameter 'buttonCode' implicitly has an 'any' type.  90 function getButtonString(buttonCode) {  ~~~~~~~~~~  src/electron/util.ts:91:74 - error TS7015: Element implicitly has an 'any' type because index expression is not of type 'number'.  91 const buttonNames = Object.keys(Button).filter(name => typeof Button[name] === 'number');  ~~~~  src/electron/util.ts:93:18 - error TS7015: Element implicitly has an 'any' type because index expression is not of type 'number'.  93 if (Button[name] === buttonCode) {  ~~~~  src/electron/util.ts:101:31 - error TS7006: Parameter 'mapping' implicitly has an 'any' type.  101 export function revertMapping(mapping) {  ~~~~~~~  Found 7 errors in the same file, starting at: src/electron/util.ts:77  ERROR: "dev:electron" exited with 2. |
| --- |

**Output:**

| Below is an example of how you can update your code with proper TypeScript type annotations. In this example, we define an interface for your mapping objects and add types to all function parameters. This should eliminate the implicit any errors:  ts  Copy  import fs from 'fs';  import { Button, Key } from '@nut-tree-fork/nut-js';  interface Mapping {  id: string;  source: string;  target: {  type: string;  keybinding?: (number | string)[];  mouseClick?: number | string;  sensitivity?: number;  positiveX?: (number | string)[];  positiveY?: (number | string)[];  negativeX?: (number | string)[];  negativeY?: (number | string)[];  };  }  // Helper function to convert a key code back into its enum string (e.g., 33 -> "Key.Num4")  function getKeyString(keyCode: number): string {  // Cast the Key enum as a record to iterate its keys.  const keyEnum = Key as unknown as Record<string, number>;  const keyNames = Object.keys(keyEnum).filter((name: string) => typeof keyEnum[name] === 'number');  for (const name of keyNames) {  if (keyEnum[name] === keyCode) {  return `Key.${name}`;  }  }  return keyCode.toString();  }  // Helper function to convert a button code back into its enum string (e.g., 0 -> "Button.LEFT")  function getButtonString(buttonCode: number): string {  const buttonEnum = Button as unknown as Record<string, number>;  const buttonNames = Object.keys(buttonEnum).filter((name: string) => typeof buttonEnum[name] === 'number');  for (const name of buttonNames) {  if (buttonEnum[name] === buttonCode) {  return `Button.${name}`;  }  }  return buttonCode.toString();  }  // Function that “reverts” the mapping conversion for each mapping object.  export function revertMapping(mapping: Mapping): void {  // For "keyboard" type mappings, convert the keybinding array numbers to strings.  if (mapping.target.type === "keyboard" && mapping.target.keybinding) {  mapping.target.keybinding = mapping.target.keybinding.map(item => {  return typeof item === 'number' ? getKeyString(item) : item;  });  }  // For "analogKeyboard" type, convert each directional key array.  if (mapping.target.type === "analogKeyboard") {  ["positiveX", "positiveY", "negativeX", "negativeY"].forEach(prop => {  const direction = mapping.target[prop as keyof typeof mapping.target] as (number | string)[] | undefined;  if (direction) {  mapping.target[prop as keyof typeof mapping.target] = direction.map(item =>  typeof item === 'number' ? getKeyString(item) : item  );  }  });  }  // For "mouseClick" type, convert the mouseClick property if it's a number.  if (mapping.target.type === "mouseClick" && typeof mapping.target.mouseClick === "number") {  mapping.target.mouseClick = getButtonString(mapping.target.mouseClick);  }  }  // Example usage:  // Assuming "data" is your array of mappings and mappingsLayoutAPath is your output file path.  function saveMappings(data: Mapping[], mappingsLayoutAPath: string): void {  // Revert conversion for each mapping before saving.  data.forEach(revertMapping);  try {  // Convert the updated data to a formatted JSON string with 4-space indentation.  const jsonString = JSON.stringify(data, null, 4);  fs.writeFileSync(mappingsLayoutAPath, jsonString, 'utf8');  console.log("Mappings saved successfully.");  } catch (error) {  console.error("Error saving mappings:", error);  }  }  Explanation  Interface Definition:  The Mapping interface defines the expected structure of your mapping objects. You can adjust it if your actual data includes additional properties.  Type Annotations:  All functions now include explicit parameter types:  getKeyString and getButtonString both expect a number.  revertMapping expects an argument of type Mapping.  Enum Access:  The enums Key and Button are cast as a Record<string, number> to safely iterate over their keys and ensure proper type checking.  Usage:  Before writing the JSON file, each mapping in the data array is processed by revertMapping to convert numeric values back to their corresponding enum string representations.  This updated TypeScript code should resolve the errors you were encountering. |
| --- |

**Claude Prompts:**

**Prompt**: export const modifiers: KeyNum[] = [

KeyNum.LeftControl,

KeyNum.RightControl,

KeyNum.LeftAlt,

KeyNum.RightAlt,

KeyNum.LeftShift,

KeyNum.RightShift,

]

Basically the selected keys, if they contain any modifier, the modifier must be at the start and those modifiers must have the above order as well and then the other key can go after.

Eg: User clicks keys in the order: A -> LeftShift -> LeftControl

It must be reordered to LeftControl -> LeftShift -> A

The handleKeyPress function was also provided in the prompt

**Output:**

const sortKeysByModifierPriority = (keys: KeyNum[]): KeyNum[] => {

const modifierKeys: KeyNum[] = [];

const nonModifierKeys: KeyNum[] = [];

keys.forEach(key => {

if (modifiers.includes(key)) {

modifierKeys.push(key);

} else {

nonModifierKeys.push(key);

}

});

// Sort modifiers according to their order in the modifiers array

modifierKeys.sort((a, b) => {

const indexA = modifiers.indexOf(a);

const indexB = modifiers.indexOf(b);

return indexA - indexB;

});

// Return modifiers followed by non-modifiers

return [...modifierKeys, ...nonModifierKeys];

};

**Prompt:**

display a preview of the controller inputs and their current mappings.

I want to list the buttons (or other inputs of the controller) in a column, with the current mapping in a column beside it.

For inputs mapped to keyboard, there should be 5 sub columns in the current mapping column. Some of these can be empty if the user has mapped to fewer than 5 keys (but 5 is the max)

Similarly, 2 sub columns for mouse click mappings

[Single column (sensitivity) for mouse motion mapping

This will come later, for now just keypress and mouse click]

For the listen buttons themselves, they should have a style similar to the following

The buttons can be identified from my mappings object:

Please render pills for the current mapping and also include target type above the current mapping

type OutputTarget = KeyboardTarget | MouseClickTarget | MouseMotionTarget

interface KeyboardTarget {

type: 'keyboard';

keybinding: Key[];

}

interface MouseClickTarget {

type: 'mouseClick';

mouseClick: Button

}

interface MouseMotionTarget {

type: 'mouseMotion';

sensitivity: number;

}

interface Mapping {

id: string; // eg: button-A

source: InputType;

target: OutputTarget;

}

Eg:

const mappingsLayoutA: Mapping[] = [

{

id: 'a',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.Num4]}

},

{

id: 'b',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.Num5]}

},

{

id: 'x',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.Num8]}

},

{

id: 'y',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.Num9]}

},

{

id: 'up',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.W]}

},

{

id: 'down',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.S]}

},

{

id: 'left',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.A]}

},

{

id: 'right',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.D]}

}

]

**Also provided starting code for the react component**

**Output:**

import React, { useState } from 'react';

// Enums for Key and Button (placeholder - replace with your actual enums)

const Key = {

Num1: 'Num1', Num2: 'Num2', Num3: 'Num3', Num4: 'Num4', Num5: 'Num5',

Num6: 'Num6', Num7: 'Num7', Num8: 'Num8', Num9: 'Num9', Num0: 'Num0',

A: 'A', B: 'B', C: 'C', D: 'D', E: 'E', F: 'F', G: 'G', H: 'H', I: 'I',

J: 'J', K: 'K', L: 'L', M: 'M', N: 'N', O: 'O', P: 'P', Q: 'Q', R: 'R',

S: 'S', T: 'T', U: 'U', V: 'V', W: 'W', X: 'X', Y: 'Y', Z: 'Z',

Space: 'Space', Enter: 'Enter', Tab: 'Tab', Escape: 'Escape'

};

const Button = {

Left: 'Left',

Right: 'Right',

Middle: 'Middle'

};

// Types

type InputType = 'button' | 'axis' | 'trigger';

interface KeyboardTarget {

type: 'keyboard';

keybinding: typeof Key[keyof typeof Key][];

}

interface MouseClickTarget {

type: 'mouseClick';

mouseClick: typeof Button[keyof typeof Button];

}

interface MouseMotionTarget {

type: 'mouseMotion';

sensitivity: number;

}

type OutputTarget = KeyboardTarget | MouseClickTarget | MouseMotionTarget;

interface Mapping {

id: string;

source: InputType;

target: OutputTarget;

}

const ControllerMappingsUI: React.FC = () => {

// Example mappings (you would pass this as a prop or from a context)

const [mappings, setMappings] = useState<Mapping[]>([

{

id: 'a',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.Num4]}

},

{

id: 'b',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.Num5]}

},

{

id: 'x',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.Num8]}

},

{

id: 'y',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.Num9]}

},

{

id: 'up',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.W]}

},

{

id: 'down',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.S]}

},

{

id: 'left',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.A]}

},

{

id: 'right',

source: 'button',

target: {type: 'keyboard', keybinding: [Key.D]}

},

{

id: 'lb',

source: 'button',

target: {type: 'mouseClick', mouseClick: Button.Left}

},

{

id: 'rb',

source: 'button',

target: {type: 'mouseClick', mouseClick: Button.Right}

},

{

id: 'lt',

source: 'trigger',

target: {type: 'keyboard', keybinding: [Key.Q, Key.W, Key.E]}

}

]);

// Mock handler for the customize button

const handleCustomize = (mappingId: string) => {

console.log(`Customize button clicked for: ${mappingId}`);

// In a real app, you would open a customization modal or panel

};

// Render key binding pills

const renderKeyBindings = (keybinding: typeof Key[keyof typeof Key][]) => {

const pills = [];

// Create 5 pills (some may be empty)

for (let i = 0; i < 5; i++) {

const key = keybinding[i];

pills.push(

<div key={i} className={`w-16 h-10 rounded-md flex items-center justify-center ${key ? 'bg-blue-100 border border-blue-300' : 'bg-gray-100 border border-gray-200'}`}>

{key || ''}

</div>

);

}

return (

<div className="flex space-x-2">

{pills}

</div>

);

};

// Render mouse click binding pills

const renderMouseClickBindings = (mouseClick: typeof Button[keyof typeof Button]) => {

return (

<div className="flex space-x-2">

<div className="w-16 h-10 rounded-md flex items-center justify-center bg-red-100 border border-red-300">

{mouseClick}

</div>

<div className="w-16 h-10 rounded-md flex items-center justify-center bg-gray-100 border border-gray-200">

{}

</div>

</div>

);

};

return (

<div className="p-4">

<h2 className="text-xl font-bold mb-4">Controller Mappings</h2>

<div className="space-y-3">

{mappings.map((mapping) => (

<div key={mapping.id} className="p-4 border rounded-lg shadow-sm hover:shadow-md bg-white transition-shadow">

<div className="flex items-center">

{/\* Input section \*/}

<div className="mr-6 flex items-center">

{/\* Placeholder for an input image if it exists \*/}

<div className="w-10 h-10 rounded-full bg-gray-200 flex items-center justify-center mr-3">

{mapping.id.toUpperCase()}

</div>

<div>

<div className="font-medium">{mapping.source.charAt(0).toUpperCase() + mapping.source.slice(1)}</div>

<div className="text-xs text-gray-500">ID: {mapping.id}</div>

</div>

</div>

{/\* Mapping section \*/}

<div className="flex-grow mx-4">

<div className="mb-2 text-sm font-medium">

<span className={`px-2 py-1 rounded-full ${

mapping.target.type === 'keyboard' ? 'bg-blue-100 text-blue-800' :

mapping.target.type === 'mouseClick' ? 'bg-red-100 text-red-800' :

'bg-purple-100 text-purple-800'

}`}>

{mapping.target.type === 'keyboard' ? 'Keyboard' :

mapping.target.type === 'mouseClick' ? 'Mouse Click' :

'Mouse Motion'}

</span>

</div>

{mapping.target.type === 'keyboard' && renderKeyBindings(mapping.target.keybinding)}

{mapping.target.type === 'mouseClick' && renderMouseClickBindings(mapping.target.mouseClick)}

</div>

{/\* Customize button \*/}

<div>

<button

className="px-4 py-2 bg-indigo-500 text-white rounded-md hover:bg-indigo-600 transition-colors"

onClick={() => handleCustomize(mapping.id)}

>

Customize

</button>

</div>

</div>

</div>

))}

</div>

</div>

);

};

export default ControllerMappingsUI;

**Prompt:**   
Given this code for layout, I need you to help with the handleKeyPress function

Basically, I can select 3 keys at most. Which is fine, but I want to sort the selected keys in a specific manner no matter what order they were selected in

export const modifiers: KeyNum[] = [

KeyNum.LeftControl,

KeyNum.RightControl,

KeyNum.LeftAlt,

KeyNum.RightAlt,

KeyNum.LeftShift,

KeyNum.RightShift,

]

Basically the selected keys, if they contain any modifier, the modifier must be at the start and those modifiers must have the above order as well and then the other key can go after.

Eg: User clicks keys in the order: A -> LeftShift -> LeftControl

It must be reordered to LeftControl -> LeftShift -> A

**Provided started code**

**Output:**

const handleKeyPress = (button: string): void => {

console.log("Button pressed", button);

if (unavailableKeys.includes(keyboardToKeyNumMap[button])) {

return;

}

if (selectedKeys.includes(keyboardToKeyNumMap[button])) {

// Removing key from selection

keyboardRef.current.removeButtonTheme(button, "selected-key")

keyboardArrowRef.current.removeButtonTheme(button, "selected-key")

keyboardControlPadRef.current.removeButtonTheme(button, "selected-key")

keyboardNumPadRef.current.removeButtonTheme(button,"selected-key")

keyboardNumPadEndRef.current.removeButtonTheme(button, "selected-key")

const newSelected = selectedKeys.filter(k => k !== keyboardToKeyNumMap[button]);

// Sort the keys even when removing one

const sortedKeys = sortKeysByModifierPriority(newSelected);

setSelectedKeys(sortedKeys);

if (onMappingChange) {

onMappingChange(sortedKeys);

}

} else {

if (selectedKeys.length >= 3) {

return; // Max 3 keys can be mapped

}

keyboardRef.current.addButtonTheme(button, "selected-key")

keyboardArrowRef.current.addButtonTheme(button, "selected-key")

keyboardControlPadRef.current.addButtonTheme(button, "selected-key")

keyboardNumPadRef.current.addButtonTheme(button,"selected-key")

keyboardNumPadEndRef.current.addButtonTheme(button, "selected-key")

const newSelected = [...selectedKeys, keyboardToKeyNumMap[button]];

// Sort keys according to modifier priority

const sortedKeys = sortKeysByModifierPriority(newSelected);

setSelectedKeys(sortedKeys);

if (onMappingChange) {

onMappingChange(sortedKeys);

}

}

};

// Helper function to sort keys by modifier priority

const sortKeysByModifierPriority = (keys: KeyNum[]): KeyNum[] => {

// Split the keys into modifiers and non-modifiers

const modifierKeys: KeyNum[] = [];

const nonModifierKeys: KeyNum[] = [];

keys.forEach(key => {

if (modifiers.includes(key)) {

modifierKeys.push(key);

} else {

nonModifierKeys.push(key);

}

});

// Sort modifiers according to their order in the modifiers array

modifierKeys.sort((a, b) => {

const indexA = modifiers.indexOf(a);

const indexB = modifiers.indexOf(b);

return indexA - indexB;

});

// Return modifiers followed by non-modifiers

return [...modifierKeys, ...nonModifierKeys];

};

**Prompt:**

Here is a keyboard layout component and my customize modal

I want to create a similar Mouse component with a left and right button and a middle button in a rounded div. Only the top should be rounded. I want to be able to select a button on the mouse and it should highlight red. Similar to how I’m saving the current keybinding by adding the tempKeyBindings, I want you to add the tempMouseClick binding when a button is clicked.

**Provided starting code files**

**Output:**

return (

<div className="flex flex-col items-center">

<div className="mb-6">

<h3 className="text-lg font-medium">

{currentMapping ?

`Binding for: ${currentMapping.id}` :

"Mouse Button Selection"}

</h3>

<p className="text-sm text-neutral-400 mt-1">

Click on a mouse button to select it

</p>

</div>

{/\* Mouse visual representation \*/}

<div className="relative w-64 h-72 border-2 border-neutral-600 rounded-full rounded-b-4xl bg-black flex flex-col items-center">

{/\* Top buttons (left and right clicks) \*/}

<div className="flex w-full mt-2">

<div

className={`w-1/2 h-28 border-r border-neutral-600 cursor-pointer rounded-tl-full flex items-center justify-center

${selectedClick === ButtonNum.LEFT ? 'bg-red-700' : 'hover:bg-neutral-600'}`}

onClick={() => handleButtonClick(ButtonNum.LEFT)}

>

<span className="text-sm font-medium">LEFT</span>

</div>

<div

className={`w-1/2 h-28 cursor-pointer rounded-tr-full flex items-center justify-center

${selectedClick === ButtonNum.RIGHT ? 'bg-red-700' : 'hover:bg-neutral-600'}`}

onClick={() => handleButtonClick(ButtonNum.RIGHT)}

>

<span className="text-sm font-medium">RIGHT</span>

</div>

</div>

{/\* Scroll wheel (middle button) \*/}

<div className="w-full flex justify-center mt-2">

<div

className={`w-12 h-12 rounded-4xl border-2 border-neutral-500 cursor-pointer flex items-center justify-center

${selectedClick === ButtonNum.MIDDLE ? 'bg-red-700 border-red-400' : 'hover:bg-neutral-600'}`}

onClick={() => handleButtonClick(ButtonNum.MIDDLE)}

>

<span className="text-xs font-medium">MID</span>

</div>

</div>

{/\* Mouse body \*/}

<div className="absolute bottom-0 w-full h-24 flex items-end justify-center pb-4">

<div className="w-4 h-10 border-2 border-neutral-500 rounded-t-full rounded-b-3xl"></div>

</div>

</div>

{/\* Currently selected button display \*/}

<div className="mt-8 p-4 bg-neutral-800 rounded-md w-full">

<h4 className="font-medium mb-2">Selected Button:</h4>

{selectedClick !== null ? (

<div className="flex items-center gap-2">

<div className="w-4 h-4 rounded-full bg-red-700"></div>

<span>{getButtonName(selectedClick)}</span>

</div>

) : (

<div className="text-neutral-500 italic">No button selected</div>

)}

</div>

</div>

);

**Prompt:**

Alright, now similar to the mouse component, I want to create a slider for MouseMotion targets. Here is some starter code. Can you add a pretty slider with a red color that I can use to set the sensitivity of an input mapped to mouse motion target?  
**Starter code provided for SensitivitySlider (only required the tsx part of the output)**

**Output:**

import React, { useEffect } from 'react';

import { Mapping } from '../../types';

interface SensitivitySliderProps {

currentMapping: Mapping;

onSensitivityChange: (newSensitivity: number) => void;

}

function SensitivitySlider({ currentMapping, onSensitivityChange }: SensitivitySliderProps) {

const [sensitivity, setSensitivity] = React.useState<number>(1);

const handleSliderChange = (event: React.ChangeEvent<HTMLInputElement>) => {

setSensitivity(parseInt(event.target.value, 10));

onSensitivityChange(parseInt(event.target.value, 10));

};

useEffect(() => {

if (currentMapping && currentMapping.target.type === 'mouseMotion') {

setSensitivity(currentMapping.target.sensitivity)

}

}, [currentMapping])

return (

<div className="flex flex-col items-center">

<h3 className="text-lg font-medium">Mouse Motion Sensitivity</h3>

<p className="text-sm text-neutral-400 mt-1">

Adjust the sensitivity for mouse motion input.

</p>

<div className="w-full flex items-center mt-4">

<input

type="range"

min="1"

max="100"

value={sensitivity}

onChange={handleSliderChange}

className="w-full accent-red-600 cursor-pointer"

/>

</div>

<div className="mt-2">

<span className="text-sm">Current Sensitivity: {sensitivity}</span>

</div>

</div>

);

}

export default SensitivitySlider;

**Prompt:**

Given the following code I got from documentation, can you slightly modify your code to use the correct types of keyboard ref. I get type errors currently

import React, { FunctionComponent, useState, useRef, ChangeEvent } from "react";

import KeyboardWrapper from "./KeyboardWrapper";

const MyComponent: FunctionComponent = () => {

const [input, setInput] = useState("");

const keyboard = useRef(null);

const onChangeInput = (event: ChangeEvent<HTMLInputElement>): void => {

const input = event.target.value;

setInput(input);

keyboard.current.setInput(input);

};

return (

<div>

<input

value={input}

placeholder={"Tap on the virtual keyboard to start"}

onChange={e => onChangeInput(e)}

/>

<KeyboardWrapper keyboardRef={keyboard} onChange={setInput} />

</div>

);

};

export default MyComponent;

import React, { FunctionComponent, useState, MutableRefObject } from "react";

import Keyboard from "react-simple-keyboard";

import "react-simple-keyboard/build/css/index.css";

interface IProps {

onChange: (input: string) => void;

keyboardRef: MutableRefObject<Keyboard>;

}

const KeyboardWrapper: FunctionComponent<IProps> = ({

onChange,

keyboardRef

}) => {

const [layoutName, setLayoutName] = useState("default");

const onKeyPress = (button: string) => {

if (button === "{shift}" || button === "{lock}") {

setLayoutName(layoutName === "default" ? "shift" : "default");

}

};

return (

<Keyboard

keyboardRef={r => (keyboardRef.current = r)}

layoutName={layoutName}

onChange={onChange}

onKeyPress={onKeyPress}

onRender={() => console.log("Rendered")}

/>

);

};

export default KeyboardWrapper;

**Output:**

import React, { useState, useRef, ChangeEvent, MutableRefObject } from "react";

import { render } from "react-dom";

import Keyboard from "react-simple-keyboard";

import "react-simple-keyboard/build/css/index.css";

import "./index.css";

// Define the keyboard options interface

interface KeyboardOptions {

onChange: (input: string) => void;

onKeyPress: (button: string) => void;

theme: string;

physicalKeyboardHighlight: boolean;

syncInstanceInputs: boolean;

mergeDisplay: boolean;

debug: boolean;

layout?: {

[key: string]: string[];

};

display?: {

[key: string]: string;

};

}

const App: React.FC = () => {

// State management with hooks

const [layoutName, setLayoutName] = useState<string>("default");

const [input, setInput] = useState<string>("");

// Use proper typing for the keyboard ref

const keyboardRef = useRef<Keyboard>(null);

// Define common keyboard options

const getCommonKeyboardOptions = (): KeyboardOptions => ({

onChange: handleChange,

onKeyPress: handleKeyPress,

theme: "simple-keyboard hg-theme-default hg-layout-default",

physicalKeyboardHighlight: true,

syncInstanceInputs: true,

mergeDisplay: true,

debug: true

});

// Define all keyboard options

const getKeyboardOptions = (): KeyboardOptions => ({

...getCommonKeyboardOptions(),

/\*\*

\* Layout by:

\* Sterling Butters (https://github.com/SterlingButters)

\*/

layout: {

default: [

"{escape} {f1} {f2} {f3} {f4} {f5} {f6} {f7} {f8} {f9} {f10} {f11} {f12}",

"` 1 2 3 4 5 6 7 8 9 0 - = {backspace}",

"{tab} q w e r t y u i o p [ ] \\",

"{capslock} a s d f g h j k l ; ' {enter}",

"{shiftleft} z x c v b n m , . / {shiftright}",

"{controlleft} {altleft} {metaleft} {space} {metaright} {altright}"

],

shift: [

"{escape} {f1} {f2} {f3} {f4} {f5} {f6} {f7} {f8} {f9} {f10} {f11} {f12}",

"~ ! @ # $ % ^ & \* ( ) \_ + {backspace}",

"{tab} Q W E R T Y U I O P { } |",

'{capslock} A S D F G H J K L : " {enter}',

"{shiftleft} Z X C V B N M < > ? {shiftright}",

"{controlleft} {altleft} {metaleft} {space} {metaright} {altright}"

]

},

display: {

"{escape}": "esc ⎋",

"{tab}": "tab ⇥",

"{backspace}": "backspace ⌫",

"{enter}": "enter ↵",

"{capslock}": "caps lock ⇪",

"{shiftleft}": "shift ⇧",

"{shiftright}": "shift ⇧",

"{controlleft}": "ctrl ⌃",

"{controlright}": "ctrl ⌃",

"{altleft}": "alt ⌥",

"{altright}": "alt ⌥",

"{metaleft}": "cmd ⌘",

"{metaright}": "cmd ⌘"

}

});

const getKeyboardControlPadOptions = (): KeyboardOptions => ({

...getCommonKeyboardOptions(),

layout: {

default: [

"{prtscr} {scrolllock} {pause}",

"{insert} {home} {pageup}",

"{delete} {end} {pagedown}"

]

}

});

const getKeyboardArrowsOptions = (): KeyboardOptions => ({

...getCommonKeyboardOptions(),

layout: {

default: ["{arrowup}", "{arrowleft} {arrowdown} {arrowright}"]

}

});

const getKeyboardNumPadOptions = (): KeyboardOptions => ({

...getCommonKeyboardOptions(),

layout: {

default: [

"{numlock} {numpaddivide} {numpadmultiply}",

"{numpad7} {numpad8} {numpad9}",

"{numpad4} {numpad5} {numpad6}",

"{numpad1} {numpad2} {numpad3}",

"{numpad0} {numpaddecimal}"

]

}

});

const getKeyboardNumPadEndOptions = (): KeyboardOptions => ({

...getCommonKeyboardOptions(),

layout: {

default: ["{numpadsubtract}", "{numpadadd}", "{numpadenter}"]

}

});

const handleChange = (newInput: string): void => {

setInput(newInput);

console.log("Input changed", newInput);

};

const handleKeyPress = (button: string): void => {

console.log("Button pressed", button);

/\*\*

\* If you want to handle the shift and caps lock buttons

\*/

if (

button === "{shift}" ||

button === "{shiftleft}" ||

button === "{shiftright}" ||

button === "{capslock}"

) {

handleShift();

}

};

const handleShift = (): void => {

setLayoutName(prevLayout =>

prevLayout === "default" ? "shift" : "default"

);

};

const handleInputChange = (event: ChangeEvent<HTMLInputElement>): void => {

const newInput = event.target.value;

setInput(newInput);

// Update keyboard input with proper null check

if (keyboardRef.current) {

keyboardRef.current.setInput(newInput);

}

};

// Define a component for the sub-keyboard components with proper ref typing

interface KeyboardComponentProps {

baseClass: string;

options: KeyboardOptions;

keyboardRef?: MutableRefObject<Keyboard | null>;

layoutName?: string;

}

const KeyboardComponent: React.FC<KeyboardComponentProps> = ({

baseClass, options, keyboardRef, layoutName

}) => {

return (

<Keyboard

baseClass={baseClass}

keyboardRef={keyboardRef ? (r => keyboardRef.current = r) : undefined}

layoutName={layoutName}

{...options}

/>

);

};

return (

<div>

<input

value={input}

placeholder={"Tap on the virtual keyboard to start"}

onChange={handleInputChange}

/>

<div className={"keyboardContainer"}>

<Keyboard

baseClass={"simple-keyboard-main"}

keyboardRef={r => (keyboardRef.current = r)}

layoutName={layoutName}

{...getKeyboardOptions()}

/>

<div className="controlArrows">

<KeyboardComponent

baseClass={"simple-keyboard-control"}

options={getKeyboardControlPadOptions()}

/>

<KeyboardComponent

baseClass={"simple-keyboard-arrows"}

options={getKeyboardArrowsOptions()}

/>

</div>

<div className="numPad">

<KeyboardComponent

baseClass={"simple-keyboard-numpad"}

options={getKeyboardNumPadOptions()}

/>

<KeyboardComponent

baseClass={"simple-keyboard-numpadEnd"}

options={getKeyboardNumPadEndOptions()}

/>

</div>

</div>

</div>

);

};

render(<App />, document.getElementById("root"));

**Prompt:**

Given the following LayoutOne and associated components.

I want you to create another Layout, LayoutTwo. This should include still have the buttons, but slightly displaced toward the top so that there's space for 2 joystick components in the center bottom.

I've included App.tsx that includes an example of the joystick component. Don't worry about the routing or which component is displayed for now. I will set up routing later to switch between LayoutOne and LayoutTwo.

For now just create LayoiutTwo and move the handleJoystickMove socket stuff to this Layout as well.

can you scale the button sizes down just slightly and move the analog stick a bit up so that they all fit nicely on a phone sized screen?

**Output:**

"use client";

import { useEffect, useState } from "react";

import GameButton from "../components/GameButton";

import { DPad } from "../components/DPad";

import { Joystick } from "react-joystick-component";

interface LayoutTwoProps {

socket: SocketIOClient.Socket;

}

function getDeviceType(socket: SocketIOClient.Socket) {

const ua = navigator.userAgent;

let deviceType = "";

if (/android/i.test(ua)) {

deviceType = "Android Device";

} else if (/iPhone|iPad|iPod/i.test(ua)) {

deviceType = "iOS Device";

} else {

deviceType = "Unknown Device Type";

}

return `${deviceType} | Socket ID - ${socket.id}`;

}

export default function LayoutTwo({ socket }: LayoutTwoProps) {

const [connected, setConnected] = useState(false);

const [isLandscape, setIsLandscape] = useState(false);

const [manuallyDisconnected, setManuallyDisconnected] = useState(false);

const [maxConnections, setMaxConnections] = useState(false);

useEffect(() => {

const handleConnect = () => setConnected(true);

const handleDisconnect = () => setConnected(false);

const handleManualDisconnect = () => setManuallyDisconnected(true);

const handleMaxConnections = () => setMaxConnections(true);

const sendDeviceInfo = () => {

const deviceName = getDeviceType(socket);

socket.emit('device-info', { deviceName: deviceName });

};

socket.on("connect", handleConnect);

socket.on("disconnect", handleDisconnect);

socket.on("request-device-info", sendDeviceInfo);

socket.on("max-connections-reached", handleMaxConnections)

socket.on("manually-disconnect", handleManualDisconnect)

// Check and update orientation

const checkOrientation = () => {

setIsLandscape(window.innerWidth > window.innerHeight);

};

// Initial check

checkOrientation();

// Listen for orientation changes

window.addEventListener('resize', checkOrientation);

window.addEventListener('orientationchange', checkOrientation);

return () => {

socket.off("connect", handleConnect);

socket.off("disconnect", handleDisconnect);

socket.off("request-device-info", sendDeviceInfo);

socket.off("max-connections-reached", handleMaxConnections);

socket.off("manually-disconnect", handleManualDisconnect);

window.removeEventListener('resize', checkOrientation);

window.removeEventListener('orientationchange', checkOrientation);

};

}, [socket]);

const handleButtonEvent = (buttonId: string, isPressed: boolean) => {

if (connected) {

socket.emit("button", { button: buttonId, pressed: isPressed });

console.log(`Button ${buttonId} ${isPressed ? "pressed" : "released"}`);

}

};

const handleJoystickMove = (joystickId: string, data: any) => {

if (connected) {

socket.emit("joystick-move", { joystickId, ...data });

console.log(`Joystick ${joystickId} moved:`, data);

}

};

const handleJoystickStop = (joystickId: string) => {

if (connected) {

socket.emit("joystick-stop", { joystickId });

console.log(`Joystick ${joystickId} stopped`);

}

};

if (!isLandscape) {

return (

<div className="fixed inset-0 flex items-center justify-center bg-black bg-opacity-90 text-white p-4 z-50">

<div className="text-center">

<svg xmlns="http://www.w3.org/2000/svg" className="h-12 w-12 mx-auto mb-2 animate-pulse" fill="none" viewBox="0 0 24 24" stroke="currentColor">

<path strokeLinecap="round" strokeLinejoin="round" strokeWidth={2} d="M4 4v7h7V4H4zm9 0v7h7V4h-7zm-9 9v7h7v-7H4zm9 0v7h7v-7h-7z" />

</svg>

<h2 className="text-xl font-bold mb-2">Please Rotate Your Device</h2>

<p>This controller is designed for landscape orientation.</p>

</div>

</div>

);

}

return (

<div className="fixed inset-0 bg-gray-900 w-full h-full overflow-hidden">

<div className={`absolute top-2 right-2 flex items-center gap-2 ${connected ? "text-green-500" : "text-red-500"}`}>

<div className={`h-3 w-3 rounded-full ${connected ? "bg-green-500" : "bg-red-500"}`}></div>

<span className="text-sm">

{maxConnections

? "Max connections currently connected. You have been disconnected. You may now close this window."

: connected

? `${getDeviceType(socket)}: Connected`

: manuallyDisconnected

? "Manually Disconnected. You may now close this window."

: "Disconnected"

}

</span>

</div>

<div className="h-full w-full flex flex-col">

{/\* Top section - Controls shifted up \*/}

<div className="flex-1 flex flex-row items-start justify-between p-2 pt-6">

{/\* Left side - D-Pad \*/}

<div className="flex-1 flex justify-center items-center">

{/\* Scale down the DPad by wrapping it in a smaller container \*/}

<div className="transform scale-75 origin-top-left">

<DPad onDirectionPress={handleButtonEvent} />

</div>

</div>

{/\* Center - Start/Options buttons \*/}

<div className="flex flex-row items-center justify-center gap-8">

<GameButton

name="start"

label="START"

color="bg-[#4E4848]"

textColor="text-white"

onPress={(isPressed) => handleButtonEvent("start", isPressed)}

className="h-[20px] w-[80px] rounded-full px-2 py-1 text-xs"

/>

<GameButton

name="options"

label="OPTIONS"

color="bg-[#4E4848]"

textColor="text-white"

onPress={(isPressed) => handleButtonEvent("options", isPressed)}

className="h-[20px] w-[80px] rounded-full px-2 py-1 text-xs"

/>

</div>

{/\* Right side - Action buttons \*/}

<div className="flex-1 flex justify-center items-center">

<div className="relative h-[180px] w-[180px]">

<div className="absolute left-1/2 top-2 -translate-x-1/2">

<GameButton

name="y"

label="Y"

color="bg-[#807300]"

onPress={(isPressed) => handleButtonEvent("y", isPressed)}

className="h-[60px] w-[60px] rounded-full"

/>

</div>

<div className="absolute right-2 top-1/2 -translate-y-1/2">

<GameButton

name="b"

label="B"

color="bg-[#A90202]"

onPress={(isPressed) => handleButtonEvent("b", isPressed)}

className="h-[60px] w-[60px] rounded-full"

/>

</div>

<div className="absolute bottom-2 left-1/2 -translate-x-1/2">

<GameButton

name="a"

label="A"

color="bg-[#00802F]"

onPress={(isPressed) => handleButtonEvent("a", isPressed)}

className="h-[60px] w-[60px] rounded-full"

/>

</div>

<div className="absolute left-2 top-1/2 -translate-y-1/2">

<GameButton

name="x"

label="X"

color="bg-[#001880]"

onPress={(isPressed) => handleButtonEvent("x", isPressed)}

className="h-[60px] w-[60px] rounded-full"

/>

</div>

</div>

</div>

</div>

{/\* Bottom section - Joysticks \*/}

<div className="h-1/4 flex flex-row items-center justify-center gap-16 mb-2">

{/\* Left Joystick \*/}

<div className="flex flex-col items-center">

<div className="mb-1 text-white text-xs">Left</div>

<div className="h-24 w-24 flex items-center justify-center">

<Joystick

size={80}

baseColor="#333333"

stickColor="#666666"

move={(data) => handleJoystickMove("left", data)}

stop={() => handleJoystickStop("left")}

/>

</div>

</div>

{/\* Right Joystick \*/}

<div className="flex flex-col items-center">

<div className="mb-1 text-white text-xs">Right</div>

<div className="h-24 w-24 flex items-center justify-center">

<Joystick

size={80}

baseColor="#333333"

stickColor="#666666"

move={(data) => handleJoystickMove("right", data)}

stop={() => handleJoystickStop("right")}

/>

</div>

</div>

</div>

</div>

</div>

);

}

**Prompt:**

I have the following map which is fine, but sometimes I need to get the other way around

export const keyboardToKeyNumMap: Record<string, KeyNum> = {

// Function keys and escape

"{escape}": KeyNum.Escape,

"{f1}": KeyNum.F1,

"{f2}": KeyNum.F2,

"{f3}": KeyNum.F3,

"{f4}": KeyNum.F4,

"{f5}": KeyNum.F5,

"{f6}": KeyNum.F6,

"{f7}": KeyNum.F7,

"{f8}": KeyNum.F8,

"{f9}": KeyNum.F9,

"{f10}": KeyNum.F10,

"{f11}": KeyNum.F11,

"{f12}": KeyNum.F12,

// Main keyboard - top row

"": KeyNum.Grave,

"1": KeyNum.Num1,

"2": KeyNum.Num2,

"3": KeyNum.Num3,

"4": KeyNum.Num4,

"5": KeyNum.Num5,

"6": KeyNum.Num6,

"7": KeyNum.Num7,

"8": KeyNum.Num8,

"9": KeyNum.Num9,

"0": KeyNum.Num0,

"-": KeyNum.Minus,

"=": KeyNum.Equal,

"{backspace}": KeyNum.Backspace,

// Main keyboard - second row

"{tab}": KeyNum.Tab,

"q": KeyNum.Q,

"w": KeyNum.W,

"e": KeyNum.E,

"r": KeyNum.R,

"t": KeyNum.T,

"y": KeyNum.Y,

"u": KeyNum.U,

"i": KeyNum.I,

"o": KeyNum.O,

"p": KeyNum.P,

"[": KeyNum.LeftBracket,

"]": KeyNum.RightBracket,

"\\": KeyNum.Backslash,

// Main keyboard - third row

"{capslock}": KeyNum.CapsLock,

"a": KeyNum.A,

"s": KeyNum.S,

"d": KeyNum.D,

"f": KeyNum.F,

"g": KeyNum.G,

"h": KeyNum.H,

"j": KeyNum.J,

"k": KeyNum.K,

"l": KeyNum.L,

";": KeyNum.Semicolon,

"'": KeyNum.Quote,

"{enter}": KeyNum.Enter,

// Main keyboard - fourth row

"{shiftleft}": KeyNum.LeftShift,

"z": KeyNum.Z,

"x": KeyNum.X,

"c": KeyNum.C,

"v": KeyNum.V,

"b": KeyNum.B,

"n": KeyNum.N,

"m": KeyNum.M,

",": KeyNum.Comma,

".": KeyNum.Period,

"/": KeyNum.Slash,

"{shiftright}": KeyNum.RightShift,

// Main keyboard - bottom row

"{controlleft}": KeyNum.LeftControl,

"{altleft}": KeyNum.LeftAlt,

"{metaleft}": KeyNum.LeftCmd, // or LeftSuper

"{space}": KeyNum.Space,

"{metaright}": KeyNum.RightCmd, // or RightSuper

"{altright}": KeyNum.RightAlt,

// Control pad

"{prtscr}": KeyNum.Print,

"{scrolllock}": KeyNum.ScrollLock,

"{pause}": KeyNum.Pause,

"{insert}": KeyNum.Insert,

"{home}": KeyNum.Home,

"{pageup}": KeyNum.PageUp,

"{delete}": KeyNum.Delete,

"{end}": KeyNum.End,

"{pagedown}": KeyNum.PageDown,

// Arrow keys

"{arrowup}": KeyNum.Up,

"{arrowleft}": KeyNum.Left,

"{arrowdown}": KeyNum.Down,

"{arrowright}": KeyNum.Right,

// Numpad

"{numlock}": KeyNum.NumLock,

"{numpaddivide}": KeyNum.Divide,

"{numpadmultiply}": KeyNum.Multiply,

"{numpad7}": KeyNum.NumPad7,

"{numpad8}": KeyNum.NumPad8,

"{numpad9}": KeyNum.NumPad9,

"{numpad4}": KeyNum.NumPad4,

"{numpad5}": KeyNum.NumPad5,

"{numpad6}": KeyNum.NumPad6,

"{numpad1}": KeyNum.NumPad1,

"{numpad2}": KeyNum.NumPad2,

"{numpad3}": KeyNum.NumPad3,

"{numpad0}": KeyNum.NumPad0,

"{numpaddecimal}": KeyNum.Decimal,

"{numpadsubtract}": KeyNum.Subtract,

"{numpadadd}": KeyNum.Add,

"{numpadenter}": KeyNum.Return

};

What do you suggest I do?

**Output:**

### **1. Precompute a Reverse Map**

You can generate a reverse lookup object once and reuse it:

ts

Copy

export const keyNumToKeyboardMap: Record<KeyNum, string> = Object.fromEntries(

Object.entries(keyboardToKeyNumMap).map(([key, value]) => [value, key])

);

This creates a mapping where KeyNum values map back to their corresponding key strings.

### **2. Create a Lookup Function on Demand**

If you don’t want to store an extra object, you can create a function to find the key dynamically:

ts

Copy

export function getKeyFromKeyNum(keyNum: KeyNum): string | undefined {

return Object.keys(keyboardToKeyNumMap).find(key => keyboardToKeyNumMap[key] === keyNum);

}

### **Which One to Choose?**

* If performance is a concern and you’ll be looking up values frequently, go with **Option 1 (Precompute a Reverse Map)**.
* If memory usage is a concern or lookups are rare, **Option 2 (Lookup Function)** is fine.

Let me know if you want something more optimized! 🚀

**V0 prompt:**I am building a web interface supposed to be run on a mobile in a web browser which interfaces a gamepad or controller. It has buttons and joysticks. pressing/releasing buttons or moving joystick sends data to a server via sockets

I want to first start with buttons. I want 10 buttons to begin with 4 representing a dpad, 4 being the main buttons (like, x,y,a,b on xbox), 1 button being the 'start' button and the last being 'options' button

I want there to be a nice animation when I press the button and release the button. If possible I want the design to closely represent a retro style gamepad with the buttons having some elevation as well and it changing when I press down on it.

So basically I want a page with these 10 buttons laid out like a gamepad and having a nice retro design. I want each button to have pressed/released state to handle

**Provided Figma Mockups in prompt**

**Output**

Game-button.tsx

"use client"

import { useState, useEffect } from "react"

interface GameButtonProps {

name: string

label?: string

color: string

textColor?: string

className?: string

onPress: (isPressed: boolean) => void

}

export default function GameButton({

name,

label,

color,

textColor = "text-white",

className = "",

onPress,

}: GameButtonProps) {

const [isPressed, setIsPressed] = useState(false)

// Handle pointer events (works for both touch and mouse)

const handlePointerDown = () => {

setIsPressed(true)

onPress(true)

}

const handlePointerUp = () => {

setIsPressed(false)

onPress(false)

}

const handlePointerLeave = () => {

if (isPressed) {

setIsPressed(false)

onPress(false)

}

}

// Clean up pressed state when component unmounts

useEffect(() => {

return () => {

if (isPressed) {

onPress(false)

}

}

}, [isPressed, onPress])

return (

<button

className={`

${color} ${textColor} font-bold

${isPressed ? "translate-y-2 brightness-75 shadow-inner" : "shadow-lg"}

transition-all duration-100 ease-in-out

select-none outline-none

${className}

`}

onPointerDown={handlePointerDown}

onPointerUp={handlePointerUp}

onPointerLeave={handlePointerLeave}

aria-pressed={isPressed}

data-button={name}

>

{label || name.toUpperCase()}

</button>

)

}

Dpad.tsx  
"use client"

import { useState } from "react"

interface DPadProps {

onDirectionPress: (buttonId: string, isPressed: boolean) => void

}

export function DPad({ onDirectionPress }: DPadProps) {

const [pressedButtons, setPressedButtons] = useState<Record<string, boolean>>({

up: false,

right: false,

down: false,

left: false,

})

// Unified pointer event handlers

const handlePointerDown = (buttonId: string) => {

setPressedButtons((prev) => ({ ...prev, [buttonId]: true }))

onDirectionPress(buttonId, true)

}

const handlePointerUp = (buttonId: string) => {

setPressedButtons((prev) => ({ ...prev, [buttonId]: false }))

onDirectionPress(buttonId, false)

}

const handlePointerLeave = (buttonId: string) => {

if (pressedButtons[buttonId]) {

setPressedButtons((prev) => ({ ...prev, [buttonId]: false }))

onDirectionPress(buttonId, false)

}

}

return (

<div className="relative h-[240px] w-[340px]">

{/\* Up button - steeper triangle and longer pad \*/}

<div

className={`

absolute left-1/2 top-0 h-[110px] w-[120px] -translate-x-1/2

bg-[#4E4848]

${pressedButtons.up ? "translate-y-1 brightness-75" : "shadow-md"}

transition-all duration-100 ease-in-out

cursor-pointer

border border-black

`}

style={{

clipPath: "polygon(0% 0%, 100% 0%, 100% 60%, 50% 100%, 0% 60%)",

}}

onPointerDown={() => handlePointerDown("up")}

onPointerUp={() => handlePointerUp("up")}

onPointerLeave={() => handlePointerLeave("up")}

aria-pressed={pressedButtons.up}

data-button="up"

/>

{/\* Right button - steeper triangle and longer pad \*/}

<div

className={`

absolute right-0 top-1/2 h-[90px] w-[160px] -translate-y-1/2

bg-[#4E4848]

${pressedButtons.right ? "translate-x-1 brightness-75" : "shadow-md"}

transition-all duration-100 ease-in-out

cursor-pointer

border border-black

`}

style={{

clipPath: "polygon(40% 0%, 100% 0%, 100% 100%, 40% 100%, 0% 50%)",

}}

onPointerDown={() => handlePointerDown("right")}

onPointerUp={() => handlePointerUp("right")}

onPointerLeave={() => handlePointerLeave("right")}

aria-pressed={pressedButtons.right}

data-button="right"

/>

{/\* Down button - steeper triangle and longer pad \*/}

<div

className={`

absolute bottom-0 left-1/2 h-[110px] w-[120px] -translate-x-1/2

bg-[#4E4848]

${pressedButtons.down ? "translate-y-[-1px] brightness-75" : "shadow-md"}

transition-all duration-100 ease-in-out

cursor-pointer

border border-black

`}

style={{

clipPath: "polygon(0% 40%, 50% 0%, 100% 40%, 100% 100%, 0% 100%)",

}}

onPointerDown={() => handlePointerDown("down")}

onPointerUp={() => handlePointerUp("down")}

onPointerLeave={() => handlePointerLeave("down")}

aria-pressed={pressedButtons.down}

data-button="down"

/>

{/\* Left button - steeper triangle and longer pad \*/}

<div

className={`

absolute left-0 top-1/2 h-[90px] w-[160px] -translate-y-1/2

bg-[#4E4848]

${pressedButtons.left ? "translate-x-[-1px] brightness-75" : "shadow-md"}

transition-all duration-100 ease-in-out

cursor-pointer

border border-black

`}

style={{

clipPath: "polygon(0% 0%, 60% 0%, 100% 50%, 60% 100%, 0% 100%)",

}}

onPointerDown={() => handlePointerDown("left")}

onPointerUp={() => handlePointerUp("left")}

onPointerLeave={() => handlePointerLeave("left")}

aria-pressed={pressedButtons.left}

data-button="left"

/>

{/\* Center piece - circular \*/}

<div className="absolute left-1/2 top-1/2 h-[50px] w-[50px] -translate-x-1/2 -translate-y-1/2 rounded-full bg-[#4E4848] border border-black"></div>

</div>

)

}

Virtual-gamepad.tsx

"use client"

import { useEffect, useState } from "react"

import { io, type Socket } from "socket.io-client"

import GameButton from "./game-button"

import { DPad } from "./d-pad"

export default function VirtualGamepad() {

const [socket, setSocket] = useState<Socket | null>(null)

const [connected, setConnected] = useState(false)

useEffect(() => {

// Initialize socket connection

// Replace with your actual server URL

const socketInstance = io("http://localhost:3001")

socketInstance.on("connect", () => {

console.log("Connected to server")

setConnected(true)

})

socketInstance.on("disconnect", () => {

console.log("Disconnected from server")

setConnected(false)

})

setSocket(socketInstance)

return () => {

socketInstance.disconnect()

}

}, [])

const handleButtonEvent = (buttonId: string, isPressed: boolean) => {

if (socket && connected) {

socket.emit("gamepad", { button: buttonId, pressed: isPressed })

console.log(`Button ${buttonId} ${isPressed ? "pressed" : "released"}`)

}

}

return (

<div className="relative w-full max-w-[900px] mx-auto">

{/\* Connection status indicator \*/}

<div

className={`absolute -top-10 right-0 flex items-center gap-2 ${connected ? "text-green-500" : "text-red-500"}`}

>

<div className={`h-3 w-3 rounded-full ${connected ? "bg-green-500" : "bg-red-500"}`}></div>

<span className="text-sm">{connected ? "Connected" : "Disconnected"}</span>

</div>

{/\* Gamepad container \*/}

<div className="relative rounded-xl border-[10px] border-[#686A74] bg-white p-4 md:p-8 shadow-2xl">

<div className="flex flex-col md:flex-row justify-between items-center gap-8 md:gap-4">

{/\* D-Pad section \*/}

<div className="md:ml-4">

<DPad onDirectionPress={handleButtonEvent} />

</div>

{/\* Action buttons section \*/}

<div className="md:mr-4 flex h-[240px] w-[240px] items-center justify-center">

<div className="relative h-full w-full">

{/\* Yellow button (top) \*/}

<div className="absolute left-1/2 top-0 -translate-x-1/2">

<GameButton

name="y"

label="Y"

color="bg-[#807300]"

onPress={(isPressed) => handleButtonEvent("y", isPressed)}

className="h-[85px] w-[85px] rounded-full"

/>

</div>

{/\* Red button (right) \*/}

<div className="absolute right-0 top-1/2 -translate-y-1/2">

<GameButton

name="b"

label="B"

color="bg-[#A90202]"

onPress={(isPressed) => handleButtonEvent("b", isPressed)}

className="h-[85px] w-[85px] rounded-full"

/>

</div>

{/\* Green button (bottom) \*/}

<div className="absolute bottom-0 left-1/2 -translate-x-1/2">

<GameButton

name="a"

label="A"

color="bg-[#00802F]"

onPress={(isPressed) => handleButtonEvent("a", isPressed)}

className="h-[85px] w-[85px] rounded-full"

/>

</div>

{/\* Blue button (left) \*/}

<div className="absolute left-0 top-1/2 -translate-y-1/2">

<GameButton

name="x"

label="X"

color="bg-[#001880]"

onPress={(isPressed) => handleButtonEvent("x", isPressed)}

className="h-[85px] w-[85px] rounded-full"

/>

</div>

</div>

</div>

</div>

{/\* Start and Options buttons \*/}

<div className="mt-4 flex justify-center gap-16">

<GameButton

name="start"

label="START"

color="bg-[#4E4848]"

textColor="text-white"

onPress={(isPressed) => handleButtonEvent("start", isPressed)}

className="h-[15px] w-[70px] rounded-full px-2 py-1 text-xs"

/>

<GameButton

name="options"

label="OPTIONS"

color="bg-[#4E4848]"

textColor="text-white"

onPress={(isPressed) => handleButtonEvent("options", isPressed)}

className="h-[15px] w-[70px] rounded-full px-2 py-1 text-xs"

/>

</div>

</div>

</div>

)

}

**Co-Pilot Prompts:**

**Prompt: #main.ts test this file aiming for maximum branch coverage. Write the tests in main.test.ts using vitest**

**Output:**

**import { test, expect, vi } from 'vitest';**

**import { app, BrowserWindow, ipcMain } from 'electron';**

**import path from 'path';**

**import fs from 'fs';**

**import { serveControllerApp } from './electron/server.ts';**

**import { ControllerLayout } from './electron/controllers/ControllerLayout.js';**

**import { initializeController } from './electron/main.ts';**

**vi.mock('electron', () => ({**

**app: {**

**on: vi.fn(),**

**getAppPath: vi.fn(() => '/mocked/path'),**

**},**

**BrowserWindow: vi.fn(() => ({**

**loadURL: vi.fn(),**

**loadFile: vi.fn(),**

**webContents: {**

**send: vi.fn(),**

**on: vi.fn(),**

**},**

**setTitle: vi.fn(),**

**setIcon: vi.fn(),**

**})),**

**ipcMain: {**

**on: vi.fn(),**

**handle: vi.fn(),**

**},**

**}));**

**vi.mock('fs', () => ({**

**readFileSync: vi.fn(),**

**}));**

**vi.mock('./electron/server.js', () => ({**

**serveControllerApp: vi.fn(),**

**}));**

**vi.mock('./electron/controllers/ControllerLayout.js', () => ({**

**ControllerLayout: vi.fn(),**

**}));**

**// Pass**

**test('should initialize the app and load the correct URL in development mode', async () => {**

**const mockedLoadURL = vi.fn();**

**BrowserWindow.mockImplementation(() => ({**

**loadURL: mockedLoadURL,**

**webContents: { send: vi.fn(), on: vi.fn() },**

**setTitle: vi.fn(),**

**setIcon: vi.fn(),**

**}));**

**process.env.NODE\_ENV = 'development';**

**process.env.LOCAL\_PORT = '3000';**

**const main = await import('./electron/main.js');**

**app.on.mock.calls[0][1](); // Simulate app "ready" event**

**expect(mockedLoadURL).toHaveBeenCalledWith('http://localhost:3000');**

**});**

**// Pass**

**test('should initialize the app and load the correct file in production mode', async () => {**

**const mockedLoadFile = vi.fn();**

**BrowserWindow.mockImplementation(() => ({**

**loadFile: mockedLoadFile,**

**webContents: { send: vi.fn(), on: vi.fn() },**

**setTitle: vi.fn(),**

**setIcon: vi.fn(),**

**}));**

**process.env.NODE\_ENV = 'production';**

**const main = await import('./electron/main.js');**

**app.on.mock.calls[0][1](); // Simulate app "ready" event**

**expect(mockedLoadFile).toHaveBeenCalledWith(**

**path.join('/mocked/path', '/dist-react/index.html')**

**);**

**});**

**test('should handle getControllerMappings IPC call', async () => {**

**const mockMappings = [{ id: 'a', source: 'button', target: { type: 'keyboard', keybinding: ['KeyA'] } }];**

**const main = await import('./electron/main.js');**

**main.currentLayout = mockMappings;**

**const handler = ipcMain.handle.mock.calls.find(([channel]) => channel === 'getControllerMappings')[1];**

**const result = handler();**

**expect(result).toEqual(mockMappings);**

**});**

**test('should handle set-layout IPC call and update current layout', async () => {**

**const mockMappingsLayoutOne = [{ id: 'a', source: 'button', target: { type: 'keyboard', keybinding: ['KeyA'] } }];**

**const mockMappingsLayoutTwo = [{ id: 'b', source: 'button', target: { type: 'keyboard', keybinding: ['KeyB'] } }];**

**const main = await import('./electron/main.js');**

**main.mappingsLayoutOne = mockMappingsLayoutOne;**

**main.mappingsLayoutTwo = mockMappingsLayoutTwo;**

**const handler = ipcMain.on.mock.calls.find(([channel]) => channel === 'set-layout')[1];**

**handler(null, 'layout-two');**

**expect(main.currentLayout).toEqual(mockMappingsLayoutTwo);**

**});**

**test('should initialize controller with correct mappings', async () => {**

**const mockController = { clearInputs: vi.fn() };**

**const mockMappings = [**

**{ id: 'a', source: 'button', target: { type: 'keyboard', keybinding: ['KeyA'] } },**

**{ id: 'b', source: 'button', target: { type: 'mouseClick', mouseClick: 'left' } },**

**];**

**await initializeController(mockController, mockMappings);**

**expect(mockController.clearInputs).toHaveBeenCalled();**

**// Add more assertions for specific input initialization logic**

**});**

**Prompt: #util.ts test this file aiming for maximum branch coverage. Write the tests in util.test.ts using vitest**

**Output:**

**import { describe, it, expect } from 'vitest';**

**import { isDev, findPrivateIp, ipcHandle, convertMapping, revertMapping } from './electron/util';**

**import { ipcMain } from 'electron';**

**import { Key, Button } from '@nut-tree-fork/nut-js';**

**describe('isDev', () => {**

**it('should return true if NODE\_ENV is "development"', () => {**

**process.env.NODE\_ENV = 'development';**

**expect(isDev()).toBe(true);**

**});**

**it('should return false if NODE\_ENV is not "development"', () => {**

**process.env.NODE\_ENV = 'production';**

**expect(isDev()).toBe(false);**

**});**

**});**

**describe('findPrivateIp', () => {**

**it('should return a private IPv4 address if available', () => {**

**const mockInterfaces = {**

**eth0: [{ family: 'IPv4', internal: false, address: '192.168.1.1' }],**

**};**

**vi.spyOn(require('os'), 'networkInterfaces').mockReturnValue(mockInterfaces);**

**expect(findPrivateIp()).toBe('192.168.1.1');**

**});**

**it('should return null if no private IPv4 address is found', () => {**

**const mockInterfaces = {**

**eth0: [{ family: 'IPv6', internal: false, address: '::1' }],**

**};**

**vi.spyOn(require('os'), 'networkInterfaces').mockReturnValue(mockInterfaces);**

**expect(findPrivateIp()).toBeNull();**

**});**

**});**

**describe('ipcHandle', () => {**

**it('should register an ipcMain handler with the correct key and handler', () => {**

**const mockHandler = vi.fn();**

**const mockIpcMainHandle = vi.spyOn(ipcMain, 'handle');**

**ipcHandle('testKey', mockHandler);**

**expect(mockIpcMainHandle).toHaveBeenCalledWith('testKey', expect.any(Function));**

**});**

**});**

**describe('convertMapping', () => {**

**it('should convert keybinding for "keyboard" type', () => {**

**const mapping = {**

**target: { type: 'keyboard', keybinding: ['Key.A', 'Key.B'] },**

**};**

**convertMapping(mapping);**

**expect(mapping.target.keybinding).toEqual([Key.A, Key.B]);**

**});**

**it('should convert directional keys for "analogKeyboard" type', () => {**

**const mapping = {**

**target: {**

**type: 'analogKeyboard',**

**positiveX: ['Key.A'],**

**positiveY: ['Key.B'],**

**},**

**};**

**convertMapping(mapping);**

**expect(mapping.target.positiveX).toEqual([Key.A]);**

**expect(mapping.target.positiveY).toEqual([Key.B]);**

**});**

**it('should convert mouseClick for "mouseClick" type', () => {**

**const mapping = {**

**target: { type: 'mouseClick', mouseClick: 'Button.LEFT' },**

**};**

**convertMapping(mapping);**

**expect(mapping.target.mouseClick).toEqual(Button.LEFT);**

**});**

**it('should not modify mapping if type is unknown', () => {**

**const mapping = { target: { type: 'unknown' } };**

**const originalMapping = JSON.stringify(mapping);**

**convertMapping(mapping);**

**expect(JSON.stringify(mapping)).toBe(originalMapping);**

**});**

**});**

**describe('revertMapping', () => {**

**it('should revert keybinding for "keyboard" type', () => {**

**const mapping = {**

**target: { type: 'keyboard', keybinding: [Key.A, Key.B] },**

**};**

**revertMapping(mapping);**

**expect(mapping.target.keybinding).toEqual(['Key.A', 'Key.B']);**

**});**

**it('should revert directional keys for "analogKeyboard" type', () => {**

**const mapping = {**

**target: {**

**type: 'analogKeyboard',**

**positiveX: [Key.A],**

**positiveY: [Key.B],**

**},**

**};**

**revertMapping(mapping);**

**expect(mapping.target.positiveX).toEqual(['Key.A']);**

**expect(mapping.target.positiveY).toEqual(['Key.B']);**

**});**

**it('should revert mouseClick for "mouseClick" type', () => {**

**const mapping = {**

**target: { type: 'mouseClick', mouseClick: Button.LEFT },**

**};**

**revertMapping(mapping);**

**expect(mapping.target.mouseClick).toEqual('Button.LEFT');**

**});**

**it('should not modify mapping if type is unknown', () => {**

**const mapping = { target: { type: 'unknown' } };**

**const originalMapping = JSON.stringify(mapping);**

**revertMapping(mapping);**

**expect(JSON.stringify(mapping)).toBe(originalMapping);**

**});**

**});**

**Prompt:**

**#pathResolver.ts test these files using vitest aiming for maximum branch coverage. Write all of the tests to inputs.test.ts file**

**Output:**

import { describe, it, expect, vi } from 'vitest';

import { getPreloadPath, getUIPath, getAssetPath, getControllerPath, getCertPath, getLayoutPath } from './electron/pathResolver';

import { app } from 'electron';

import path from 'path';

import { isDev } from './electron/util';

vi.mock('electron', () => ({

app: {

getAppPath: vi.fn(() => '/mocked/app/path'),

},

}));

vi.mock('./electron/util', () => ({

isDev: vi.fn(),

}));

describe('pathResolver', () => {

describe('getPreloadPath', () => {

it('should return the correct preload path in development mode', () => {

vi.mocked(isDev).mockReturnValue(true);

const result = getPreloadPath();

expect(result).toBe(path.join('/mocked/app/path', '.', '/dist-electron/preload.cjs'));

});

it('should return the correct preload path in production mode', () => {

vi.mocked(isDev).mockReturnValue(false);

const result = getPreloadPath();

expect(result).toBe(path.join('/mocked/app/path', '..', '/dist-electron/preload.cjs'));

});

});

describe('getUIPath', () => {

it('should return the correct UI path', () => {

const result = getUIPath();

expect(result).toBe(path.join('/mocked/app/path', '/dist-react/index.html'));

});

});

describe('getAssetPath', () => {

it('should return the correct asset path in development mode', () => {

vi.mocked(isDev).mockReturnValue(true);

const result = getAssetPath();

expect(result).toBe(path.join('/mocked/app/path', '.', '/src/assets'));

});

it('should return the correct asset path in production mode', () => {

vi.mocked(isDev).mockReturnValue(false);

const result = getAssetPath();

expect(result).toBe(path.join('/mocked/app/path', '..', '/src/assets'));

});

});

describe('getControllerPath', () => {

it('should return the correct controller path', () => {

const result = getControllerPath();

expect(result).toBe(path.join('/mocked/app/path', '../controller-app/dist-controller'));

});

});

describe('getCertPath', () => {

it('should return the correct certificate path', () => {

const result = getCertPath();

expect(result).toBe(path.join('/mocked/app/path', '/certs'));

});

});

describe('getLayoutPath', () => {

it('should return the correct layout path in development mode', () => {

vi.mocked(isDev).mockReturnValue(true);

const result = getLayoutPath();

expect(result).toBe(path.join('/mocked/app/path', '.', '/src/electron/layouts'));

});

it('should return the correct layout path in production mode', () => {

vi.mocked(isDev).mockReturnValue(false);

const result = getLayoutPath();

expect(result).toBe(path.join('/mocked/app/path', '..', '/src/electron/layouts'));

});

});

});

**Prompt:**

#AnaogInput.ts test these files using vitest aiming for maximum branch coverage. Write all of the tests to tests.test.ts file

**Output:**

**import { describe, it, expect, vi } from "vitest";**

**import { AnalogInput } from "./electron/controller-inputs/AnaogInput";**

**import { MouseMotionTarget, AnalogKeyboardTarget, Coordinates } from "./types";**

**import { keyboard, mouse, Point } from "@nut-tree-fork/nut-js";**

**vi.mock("@nut-tree-fork/nut-js", () => ({**

**keyboard: {**

**pressKey: vi.fn(),**

**releaseKey: vi.fn(),**

**config: { autoDelayMs: 0 },**

**},**

**mouse: {**

**setPosition: vi.fn(),**

**getPosition: vi.fn(() => Promise.resolve(new Point(0, 0))),**

**config: { autoDelayMs: 0 },**

**},**

**Point: class {**

**constructor(public x: number, public y: number) {}**

**},**

**}));**

**describe("AnalogInput", () => {**

**const mockMouseMotionTarget: MouseMotionTarget = {**

**type: "mouseMotion",**

**};**

**const mockAnalogKeyboardTarget: AnalogKeyboardTarget = {**

**type: "analogKeyboard",**

**positiveX: ["d"],**

**positiveY: ["w"],**

**negativeX: ["a"],**

**negativeY: ["s"],**

**};**

**it("should initialize with correct properties", () => {**

**const analogInput = new AnalogInput("test-id", mockMouseMotionTarget);**

**expect(analogInput.getId()).toBe("test-id");**

**expect(analogInput.getMappingTarget()).toBe(mockMouseMotionTarget);**

**});**

**it("should handle mouse motion input correctly", async () => {**

**const analogInput = new AnalogInput("test-id", mockMouseMotionTarget);**

**await analogInput.setScreenDimensions();**

**const position: Coordinates = { x: 0.5, y: -0.5 };**

**await analogInput.handleInput(position);**

**expect(mouse.setPosition).toHaveBeenCalled();**

**});**

**it("should handle analog keyboard input correctly", async () => {**

**const analogInput = new AnalogInput("test-id", mockAnalogKeyboardTarget);**

**const position: Coordinates = { x: 1, y: -1 };**

**await analogInput.handleInput(position);**

**expect(keyboard.pressKey).toHaveBeenCalledWith("d");**

**expect(keyboard.pressKey).toHaveBeenCalledWith("s");**

**});**

**it("should stop motion when input is zero", async () => {**

**const analogInput = new AnalogInput("test-id", mockMouseMotionTarget);**

**await analogInput.setScreenDimensions();**

**const position: Coordinates = { x: 0, y: 0 };**

**await analogInput.handleInput(position);**

**expect(mouse.setPosition).not.toHaveBeenCalled();**

**});**

**it("should apply deadzone correctly", async () => {**

**const analogInput = new AnalogInput("test-id", mockMouseMotionTarget);**

**await analogInput.setScreenDimensions();**

**const position: Coordinates = { x: 0.1, y: 0.1 }; // Within deadzone**

**await analogInput.handleInput(position);**

**expect(mouse.setPosition).not.toHaveBeenCalled();**

**});**

**it("should start and stop motion correctly", async () => {**

**const analogInput = new AnalogInput("test-id", mockMouseMotionTarget);**

**await analogInput.setScreenDimensions();**

**const startPosition: Coordinates = { x: 0.5, y: 0.5 };**

**await analogInput.handleInput(startPosition);**

**expect(mouse.setPosition).toHaveBeenCalled();**

**const stopPosition: Coordinates = { x: 0, y: 0 };**

**await analogInput.handleInput(stopPosition);**

**expect(mouse.setPosition).toHaveBeenCalledTimes(1); // No additional calls after stopping**

**});**

**it("should handle keyboard input correctly when moving in positive X direction", async () => {**

**const analogInput = new AnalogInput("test-id", mockAnalogKeyboardTarget);**

**const position: Coordinates = { x: 1, y: 0 };**

**await analogInput.handleInput(position);**

**expect(keyboard.pressKey).toHaveBeenCalledWith("d");**

**expect(keyboard.releaseKey).toHaveBeenCalledWith("a");**

**});**

**it("should handle keyboard input correctly when moving in negative Y direction", async () => {**

**const analogInput = new AnalogInput("test-id", mockAnalogKeyboardTarget);**

**const position: Coordinates = { x: 0, y: -1 };**

**await analogInput.handleInput(position);**

**expect(keyboard.pressKey).toHaveBeenCalledWith("s");**

**expect(keyboard.releaseKey).toHaveBeenCalledWith("w");**

**});**

**it("should clear intervals when stopping motion", async () => {**

**const analogInput = new AnalogInput("test-id", mockMouseMotionTarget);**

**await analogInput.setScreenDimensions();**

**const startPosition: Coordinates = { x: 0.5, y: 0.5 };**

**await analogInput.handleInput(startPosition);**

**analogInput.stopMotion();**

**expect(mouse.setPosition).toHaveBeenCalled();**

**});**

**});**

**Prompt:**

#ButtonInput.ts test these files using vitest aiming for maximum branch coverage. Write all of the tests to tests.test.ts file

**Output:**

**import { describe, it, expect, vi } from "vitest";**

**import { ButtonInput } from "./electron/controller-inputs/ButtonInput";**

**import { KeyboardTarget, MouseClickTarget } from "./types";**

**import { keyboard, mouse } from "@nut-tree-fork/nut-js";**

**vi.mock("@nut-tree-fork/nut-js", () => ({**

**keyboard: {**

**pressKey: vi.fn(),**

**releaseKey: vi.fn(),**

**config: { autoDelayMs: 0 },**

**},**

**mouse: {**

**pressButton: vi.fn(),**

**releaseButton: vi.fn(),**

**config: { autoDelayMs: 0 },**

**},**

**}));**

**describe("ButtonInput", () => {**

**const keyboardTarget: KeyboardTarget = {**

**type: "keyboard",**

**keybinding: ["Shift", "A"],**

**};**

**const mouseClickTarget: MouseClickTarget = {**

**type: "mouseClick",**

**mouseClick: "left",**

**};**

**it("should handle keyboard input when pressed", async () => {**

**const buttonInput = new ButtonInput("test-id", keyboardTarget);**

**await buttonInput.handleInput(true);**

**expect(keyboard.pressKey).toHaveBeenCalledWith("Shift", "A");**

**expect(keyboard.releaseKey).not.toHaveBeenCalled();**

**});**

**it("should handle mouse click input when pressed", async () => {**

**const buttonInput = new ButtonInput("test-id", mouseClickTarget);**

**await buttonInput.handleInput(true);**

**expect(mouse.pressButton).toHaveBeenCalledWith("left");**

**expect(mouse.releaseButton).not.toHaveBeenCalled();**

**});**

**});**

**Prompt:**

#MotionControllerInput.ts test these files using vitest aiming for maximum branch coverage. Write all of the tests to tests.test.ts file

**Output:**

**import { describe, it, expect, vi } from "vitest";**

**import { MotionInput } from "./electron/controller-inputs/MotionControllerInput";**

**import { MouseMotionTarget, AnalogKeyboardTarget, Accelerometer } from "./types";**

**import { keyboard, mouse, Point } from "@nut-tree-fork/nut-js";**

**vi.mock("@nut-tree-fork/nut-js", () => ({**

**keyboard: {**

**config: { autoDelayMs: 0 },**

**pressKey: vi.fn(),**

**releaseKey: vi.fn(),**

**},**

**mouse: {**

**config: { autoDelayMs: 0 },**

**setPosition: vi.fn(),**

**},**

**Point: vi.fn((x, y) => ({ x, y })),**

**}));**

**describe("MotionInput", () => {**

**const mockMouseTarget: MouseMotionTarget = { type: "mouseMotion" };**

**const mockKeyboardTarget: AnalogKeyboardTarget = {**

**type: "analogKeyboard",**

**positiveX: ["d"],**

**positiveY: ["w"],**

**negativeX: ["a"],**

**negativeY: ["s"],**

**};**

**it("should initialize with correct properties", () => {**

**const motionInput = new MotionInput("test-id", mockMouseTarget);**

**expect(motionInput.getId()).toBe("test-id");**

**expect(motionInput.getMappingTarget()).toBe(mockMouseTarget);**

**});**

**it("should handle analog keyboard input with negative Y direction", async () => {**

**const motionInput = new MotionInput("test-id", mockKeyboardTarget);**

**const accelerometer: Accelerometer = { x: 0, y: -9.81, z: 0 };**

**await motionInput.handleInput(accelerometer);**

**expect(keyboard.pressKey).toHaveBeenCalledWith("s");**

**expect(keyboard.releaseKey).toHaveBeenCalledWith("w");**

**});**

**it("should clip values within range", () => {**

**const motionInput = new MotionInput("test-id", mockMouseTarget);**

**const result = (motionInput as any).clipInRange(15, 0, 10);**

**expect(result).toBe(10);**

**});**

**});**

**Prompt:**

#VoiceCommandInput.ts test these files using vitest aiming for maximum branch coverage. Write all of the tests to tests.test.ts file

**Output:**

**import { describe, it, expect, vi } from "vitest";**

**import { VoiceCommandInput } from "./electron/controller-inputs/VoiceCommandInput";**

**import { KeyboardTarget, MouseClickTarget } from "./types";**

**import { keyboard, mouse } from "@nut-tree-fork/nut-js";**

**vi.mock("@nut-tree-fork/nut-js", () => ({**

**keyboard: {**

**pressKey: vi.fn(),**

**releaseKey: vi.fn(),**

**},**

**mouse: {**

**pressButton: vi.fn(),**

**releaseButton: vi.fn(),**

**},**

**}));**

**describe("VoiceCommandInput", () => {**

**describe("handleInput", () => {**

**it("should handle keyboard input correctly", async () => {**

**const keybinding = ["KeyA"];**

**const mappingTarget: KeyboardTarget = { type: "keyboard", keybinding };**

**const voiceCommandInput = new VoiceCommandInput("test-id", mappingTarget);**

**await voiceCommandInput.handleInput();**

**expect(keyboard.pressKey).toHaveBeenCalledWith(...keybinding);**

**expect(keyboard.releaseKey).toHaveBeenCalledWith(...keybinding);**

**});**

**it("should handle mouse click input correctly", async () => {**

**const mouseClick = "left";**

**const mappingTarget: MouseClickTarget = { type: "mouseClick", mouseClick };**

**const voiceCommandInput = new VoiceCommandInput("test-id", mappingTarget);**

**await voiceCommandInput.handleInput();**

**expect(mouse.pressButton).toHaveBeenCalledWith(mouseClick);**

**expect(mouse.releaseButton).toHaveBeenCalledWith(mouseClick);**

**});**

**it("should throw an error for invalid mapping target type", async () => {**

**const invalidMappingTarget = { type: "invalid" } as any;**

**const voiceCommandInput = new VoiceCommandInput("test-id", invalidMappingTarget);**

**await expect(voiceCommandInput.handleInput()).rejects.toThrowError(**

**"Invalid mapping target type."**

**);**

**});**

**});**

**describe("handleKeyboardInput", () => {**

**it("should press and release the correct keys", async () => {**

**const keybinding = ["KeyB"];**

**const mappingTarget: KeyboardTarget = { type: "keyboard", keybinding };**

**const voiceCommandInput = new VoiceCommandInput("test-id", mappingTarget);**

**await (voiceCommandInput as any).handleKeyboardInput();**

**expect(keyboard.pressKey).toHaveBeenCalledWith(...keybinding);**

**expect(keyboard.releaseKey).toHaveBeenCalledWith(...keybinding);**

**});**

**});**

**describe("handleMouseInput", () => {**

**it("should press and release the correct mouse button", async () => {**

**const mouseClick = "right";**

**const mappingTarget: MouseClickTarget = { type: "mouseClick", mouseClick };**

**const voiceCommandInput = new VoiceCommandInput("test-id", mappingTarget);**

**await (voiceCommandInput as any).handleMouseInput();**

**expect(mouse.pressButton).toHaveBeenCalledWith(mouseClick);**

**expect(mouse.releaseButton).toHaveBeenCalledWith(mouseClick);**

**});**

**});**

**});**

**Prompt:**

#ControllerLayout.ts test these files using vitest aiming for maximum branch coverage. Write all of the tests to tests.test.ts file

**Output:**

**import { describe, it, expect, vi, beforeEach } from "vitest";**

**import { ControllerLayout } from "./electron/controllers/ControllerLayout";**

**import { ControllerInput } from "./electron/controller-inputs/ControllerInput";**

**describe("ControllerLayout", () => {**

**let controllerLayout: ControllerLayout;**

**let mockInput: ControllerInput;**

**beforeEach(() => {**

**controllerLayout = new ControllerLayout("test-layout");**

**mockInput = {**

**getId: vi.fn(() => "mock-input"),**

**} as unknown as ControllerInput;**

**});**

**it("should initialize with an empty inputs map", () => {**

**expect(controllerLayout.inputs.size).toBe(0);**

**});**

**it("should add an input to the inputs map", () => {**

**controllerLayout.addInput(mockInput);**

**expect(controllerLayout.inputs.size).toBe(1);**

**expect(controllerLayout.inputs.get("mock-input")).toBe(mockInput);**

**});**

**it("should clear all inputs from the inputs map", () => {**

**controllerLayout.addInput(mockInput);**

**expect(controllerLayout.inputs.size).toBe(1);**

**controllerLayout.clearInputs();**

**expect(controllerLayout.inputs.size).toBe(0);**

**});**

**it("should not throw an error when clearing an already empty inputs map", () => {**

**expect(() => controllerLayout.clearInputs()).not.toThrow();**

**});**

**it("should overwrite an existing input with the same ID", () => {**

**const anotherMockInput = {**

**getId: vi.fn(() => "mock-input"),**

**} as unknown as ControllerInput;**

**controllerLayout.addInput(mockInput);**

**controllerLayout.addInput(anotherMockInput);**

**expect(controllerLayout.inputs.size).toBe(1);**

**expect(controllerLayout.inputs.get("mock-input")).toBe(anotherMockInput);**

**});**

**});**