**Magic Memory**

**This is a memory game build using React, players look for the matching pairs of cards by flipping**

**downturned cards to display the card for a brief moment and attempt to remember their position t**

**o matching pairs on a grid**

**This memory game uses React with various dependencies and files:**

**e.g. App.js, index.js, App.css and index.css.**

**We put the use effect hook to good use in a way that doesn’t involve fetching data.**

**This is a simple game, we flip 2 cards (like snap) to see the image underneath.**

**If they match, they stay flipped, if they don’t match they turn back over.**

**At the bottom there’s a turn counter: How many turns to match up the whole thing.**

**At the top is a New Game button that restarts the game after shuffling the cards.**

**The game is set out in a grid: 6 rows, 3 for the front of card 3 for the back of card.**

**4 columns each with the front and back of cards (alternating for each row).**

**As we build out this project, we add a components folder with SingleCard.js, and SingleCard.css:**

**We have our separation of concerns and work on the functionality and styling of an individual card.**

**1# Intro & Setup**

**First we set up the project via a starter project provided by The Net Ninja.**

**We go to GitHub, download the starter project and set it up in VSCode.**

**Updated Node and NPM to the latest versions and installed the modules**

**and dependencies.**

**We have an index.js file with import and function call**

**We use App.js file for building and rendering the game and it’s graphical interface**

**We have an index.css file to style our App.js graphic interface**

**We have an App.css file that targets the App div and applies the styles, button and layout**

**We have a public folder and within that we have various image files: These are for the**

**Graphics for the front of each card (ring, scroll, helmet, sword, potion) and so on**

**Graphic for back of the card (the cover)**

**We use a local dev server to spin up and run the program in the browser**

**The build out begins:**

**2# Shuffling Cards**

1. **Make an array of cards and make each card point to the image src re: public folder**
2. **The array gets created outside the component: const cardImages = []**
3. **We make each card an object that goes inside the array and it has a property**

**Re: the src property that is a path to a specific image**

1. **We create a function re: const shuffleCards = () => {}… inside the component that does 3 things:**
2. **It duplicates each card once as we need 2 of each image in order to get a snap! So, we have an array of 12 cards.**
3. **Next we randomise the order of the cards in the array using the .sort() method, re: .sort(() => Math.random() - 0.5) the result is a shuffled array!**
4. **It provides a random ID to each of the 12 cards - we’ll use that ID as a key for React when we’re outputting them later in some kind of list / grid**

**Re: we use the map method and fire a function for each item inside our new sorted array and for each one we add on an ID property.**

**We want to return an object In each case, re: this object takes all of the card properties. Card is taken in as an argument re: .map((card)…**

**We fire a function for each item in our sorted array. Each item is represented by our card object that we take into our function re: .map((card)…**

**Each time we map it to a new array where we take the current card and we spread the properties of that card (at the moment it’s just the one property re: src**

**And we add a new id property, and that id is Math.random() so this is a random id number on each object!**

**.map((card) => ({ …card, id: Math.random() }))**

**To summarise, we have our constant (const shuffledCards = …) that is an array of shuffled cards, where each card also has an id property as well!**

**We’re not doing anything with const shuffledCards at the moment, so we’ll create some state to store our cards in for a particular game.**

**In our App() function (at the top) so we create it re: const [cards,… and we need a function to update them re: const [cards, setCards]… we’ll set that**

**Equal to useState, this causes React to import it for us at the top, re: import { useState } from ‘react’, the initial value is an empty array:**

**const [cards, setCards] = useState([])**

**After we’ve called the function to shuffle the cards we update the state using setCards from const [cards, setCards] = useState([])**

**We put this in the function body of our shuffleCards() function, (yes, within the braces) and we pass in our shuffledCards variable**

**re: setCards(shuffledCards) so when we call this function it will generate the shuffled cards, update our card state to be those shuffled cards**

**We also want more state for the turns, re: how many turns a player is taking to complete the game, re: it increments by one every turn**

**Re: const [turns, setTurns] = useState(0), we also put setTurns(0) under our setCards(shuffledCards) function call because everytime we**

**start a new game by clicking on the `New Game` button we automatically call the shuffleCards() function, which will in turn**

**i) shuffle the cards, re: const shuffledCards = […cardImages, …cardImages] .sort(() => Math.random() – 0.5)…**

**ii) set the cards to be that shuffled cards constant re: const [cards, setCards] = useState([])**

**ii) It’s going to reset the turns back to 0 every time we click on that `New Game` button!**

**Next we hook up our function shuffleCards() with our `New Game` button, we use onClick and set it equal to shuffleCards:**

**<button inClick={shuffleCards}>New Game</button> Finally we do console.log(cards, turns)**

**We now test in our browser using our local dev server and see via inspector and the console re: we click on the**

**New Game` button, our shuffleCards() function runs and we see the shuffled cards logged to console and 0 for the no. of turns.**

**The console in our dev tools shows us the cards as an array of 12 items and inside we have a shuffled array of cards, re:**

**0: {src: ‘img/potion-1.png’, id: …}**

**1: {src: ‘img/shield-1.png’, id: …} and so on**

**3# Creating a Card Grid**

**Now that we have our shuffled cards stored in this bit of state re: const [cards, setCards] = useState([]), we are going to cycle through that bit of state**

**and output a bit of template for each card so they are displayed in some kind of grid. So we move down our normal flow to the rendering part of our comp**

**below where we created our button (that once clicked, activates a new game and shuffles the cards) we create a new <div> this will have a class re:**

**<div className=”card-grid”> inside here we’ll map through our card’s state, re: that array of cards re: const [cards, setCards] = useState([])**

**So we use {cards.map(card… re: by passing in the card arg we take each individual card in our array and return some template within parenthesis for that card:**

**{cards.map(card => (… Next we create a <div> re: remember that that parent element has to have a key property, so the key is going to be equal to the card id**

**Re: <div key={card.id}></div> re: remember that we added that property on right here via our map() method re: .map((card) => ({ …card, id: Math.random() }))**

**So that’s the class key and we also need a class name re: so we can style this later if we wish re: <div className=”card” key={card.id}></div> now we need to**

**nest (or if we don’t like the term nest), we need to do inside our 1st <div>**

**a 2nd <div> that will hold 2 images, 1) re: the front image of the card (determined via our `src`) and 2) the back of the card that is the cover re: in the img folder re: img/cover.png.**

1. **The front of the card so we use <img> tag and it has a src that is dynamic re: <img src={card.src… re: that was the property name we have on each card, remember that?**

**Re: const cardImages = [ { “src”: “/img/helmet-1.png” },… that’s the image source. We will also give it a class re: <img className=”front” src={card.src} alt=”card front” />  
re: yes this is the front of the card, the actual picture! So that’s the 1st image. Lets also do the back, re: another img tag and the src won’t be dynamic this time, re:**

**It will be <img className=”back” src=”/img/cover.png” alt=”card back” which corresponds to our img folder in our MAGIC-MEMORY directory and the png file within our img folder**

**and again we give it a class name (so we can style this later on). We preview in our browser and get a bunch of cards in one big column with the front of card on left and the back on right.**

**When we click on New Game it shuffles the cards and the order changes each time as expected!**

**Now we open up App.css and paste in some styles, re: we target the card-grid class re: .card-grid {…. In Apps.css corresponding to the div that surrounds all the different cards in App.js**

**Re: <div className=”card-grid”>… we use the following styles S = .card-grid { P&V = margin-top:40px; display: grid; grid-template-columns: 1fr 1fr 1fr 1fr; grid-gap: 20px**

**So we display the cards in this div as a grid with 4 different columns so we get 4 cards in a row, each the same fraction of the available space with a gap between each col of 20px (re: grid-gap).**

**The game is now set out in a grid:**

**6 rows, or 3 double rows, the top layer being the front of the card and the bottom layer being the back of card.**

**4 columns each with the front and back of cards alternating as we descend through the double rows.**

**Later on we’ll hide the front of the card and only show the back of card and when we click on the back it flips to show the front of the card.**

**#4 Creating a Card Component**

**Project challenge 1:**

**Re: App.js file, extract all of this card template into its own components. This facilitates us applying logic to each card later on!**

**We create a components folder within src folder. In components we create 2 files: SingleCard.css and SingleCard.js**

**We’ll hook up our single card component to SingleCard.css later on!**

**I wish to create a card (SingleCard) component for my Memory Game to make it easier to apply logic to each card later on.**

**1) I created a new folder called components in the src dir.**

**2) Within said folder, I added 2 files, SingleCard.js, and SingleCard.css (that we'll hook up to our card component later on**

**3) Extract the card template into its own component.**

**4) I cut a code snippet out of App.js that relates to the card component and pasted it into SingleCard.js**

**5) I added an import statement to App.js to import SingleCard.js**

**6) I added an export statement to SingleCard.js to export component to App.js**

**With ChatGPT I found a solution that worked correctly but not the solution.**

**The solution is as follows:**

**In SingleCard.js we create a functional component \_rfc with tab to create this functional component called SIngleCard()**

**Re: export default function SingleCard() { return ( <div> </div> ) } above this we import our css (this is not part of the challenge but useful for us later on!**

**Re: import ‘./SingleCard.css’ Next we need to grab our template <div> from App.js and place it inside the card component within the <div> re:**

**export default function SingleCard() { return ( <div className=”card” key={card.id}> <div><img className=”front” src={card.src} alt=”card front” />**

**<img className=”back” src=”/img/cover.png” alt=”card back” </div> </div>) }**

**We now have our <div> with a className of card and within it we have another <div. with 2 images; however, we have 2 problems:**

1. **We don’t need the key here because we’re not inside our map() function re: App.js re: {cards.map(card => (… we output the key in our map function in Apps.js**

**To fix this we nest our SingleCard component within our map() function so for each card we’re outputting the single card component re:**

**<div className=”card-grid”> {cards.map(card => ( <SingleCard /> ))} </div> the emmet should automatically import it at the top, if it doesn’t it’s as follows (3rd line from top):**

**import SingleCard from ‘./components/SingleCard’… now we can add on the key property as follows: {cards.map(card => ( <SingleCard key={card.id} /> this is where the key goes!**

**Right inside our component where we’re mapping through the cards. Now that we have it in App.js we can remove it from SingleCard.js.**

1. **The other problem is that we’re attempting to access the card object for the source of the image and we don’t have access to it from SingleCard.js, access is in App.js.**

**We can work around this in SingleCard.js by passing down a prop called card and send it into that component re: {cards.map(card => ( <SingleCard key={card.id} card={card} />**

**Re: the 2nd card (in big letters) is sending in the prop called card re: 1st card (in big letters) We now have to accept the prop above by de-structuring it from the prop:**

**Re: export default function SingleCard({ card }) {… and now we have access to it in the template re: <img className=”front” src={card.src} alt=”card front” />…**

**This completes the challenge and it should all work in the same way yet we are now running a SingleCard component that is connected with App.js and takes care of the**

**functionality related to the interactivity of a single cards in our programme. We also now add the CSS for each card, re: .card { position: relative; }… each card gets a position of relative**

**For each image, re: .card img { width: 100% display: block; border: 2px solid #fff; border-radius: 6px; re: some simple styles to improve the look.**

**Summary: We’ve moved all our card template into its own component that can be reused, this will make it much easier in future to apply logic to single cards inside this component.**

**#5 Making Card Choices**

**A player will be clicking on the cards to make choices about which ones they wish to attempt to match we need a way to store those cards / those player choices.**

**So that we they have chosen 2 cards we can compare those 2 cards, do they match / or not! We create a bit of state for each card choice made:**

**Re: const [choiceOne, setChoiceOne] = useState(null)**

**const [choiceTwo, setChoiceTwo] = useState(null)**

**When a player clicks on a 1st card we will update choiceOne to be that card then when they click on a 2nd card we update choiceTwo to be that card.**

**Next, we need to add a click event to each card, we do this via our SingleCard.js component (remember that one) re: in App.js it is referenced in the rendering part**

**of our App.js function component**

**Re: <SingleCard key={card.id} card={card} />**

**Go to SingleCard.js component, the click event must be on the back of the card re: `the cover` re: later we won’t show the front until a player clicks on the back.**

1. **Add some onClick to `the back` / the cover and set it equal to a function, we’ll call that function {handleClick} re:**

**<img**

**className=”back”**

**src=”/img/cover.png”**

**onClick={handleClick}**

**alt=”card back”**

**/>**

1. **Create our function called handleClick inside this component re: export default function SingleCard({ card }) {…**

**Re: const handleClick = () => { Can’t set the state from here!}… We switch to App.js at this point!**

**We now need to update some state in our App.js component, this will be either choiceOne or choiceTwo. Because we cannot set the state directly from**

**Our handleClick() function within our SingleCard.js we have to do it within our App.js component re: const [choiceTwo, setChoiceTwo] = useState(null)**

**So we create another function within this component that enables us to handle a choice, we put this new function below the shuffleCards()**

**function and above the rendering section of this comp. Re: const handleChoice = (card) => {}**

**As we can see this function takes as an arg the card that a player has chosen. Within the function block we log to console: console.log(card)**

**We pass in our handleChoice() function re: const handleChoice = (card) => {… as a prop into the rendering part of our component where we have our SingleCard**

**<SingleCard**

**Key={card.id}**

**Card={card}**

**handleChoice={handleChoice} // this is our handleChoice prop that we’ve set equal to the handleChoice function!**

**/>**

**So we’re passing this function in, as a prop, to the SingleCard comp, remember that it takes a `card` arg (that we then logged to the console)**

**Now we switch to SingleCard.js re: within our SingleCard comp we can extract that in the props object re: export default function SingleCard({ card, handleChoice }) {…**

**Then we can call that function re: export default function SingleCard({ card, handleChoice }) {… re: via what we add to the body of our handleClick() function**

**NB: const handleClick = () => {} is nested within : export default function SingleCard({ card, handleChoice }) {…**

**Re: here’s the modified handleClick() function, re: the const handleClick = () => {handleChoice} and then pass in the card that we’ve chosen**

**re: const handleClick = () => { handleChoice(card)…**

**Remember that we have access to the card via export default function SingleCard({ card… re: because for this SingleCard we pass in the card as a prop**

**Re: App.js re: <SingleCard**

**Key={card.id}**

**Card={card} // Here!**

**handleChoice={handleChoice} // so we’re passing this card re: const handleClick = () => { handleChoice(card)…**

**into the handleChoice function that is defined in the App.js comp re: Re: const handleChoice = (card) => {}**

**and we’re logging it to the console!**

**We now check to see if it worked in the browser re: inspector/console. Remember that we attached our click event to the back / cover of the card so we click on the cover:**

**Re: We click on the back of say the helmet and the corresponding {src: ‘/img/helmet-1.png’,… is logged to console, so it works!**

**Next we return to the App.js and we need to update our state re: choiceOne / choiceTwo with whatever card the player chose:**

**Re: const [choiceOne, setChoiceOne] = useState(null)… YET how do we know whether its choice 1 / 2? We can check what choiceOne is inside our handleChoice() function**

**E.g. Has a value means we have a choiceOne / has no value means we have no choiceOne. If no value = update choiceOne, if value = update choiceTwo**

**We can do this (programmatically) using a ternary operator within our handleChoice() function body: First delete the console.log statement that is currently there, we don’t need it:**

**choiceOne ? setChoiceTwo(card) : setChoiceOne(card)…**

1. **Ternary operator: choiceOne ?... gets evaluated: re: if it’s null then it will evaluate as false if it’s not null (in other words, if it has a value) it evaluates as true.**
2. **We are going to return 2 values here, they’re both going to be a call to either set choiceOne / set choiceTwo, re: choiceOne ? setChoiceTwo(card) : setChoiceOne(card)…**
3. **On the left we have setChoiceTwo (that’s set to be the card), on the right we have setChoiceOne. If setChoiceOne is null it means we don’t have a selection for choiceOne**
4. **Remember: choiceOne ? setChoiceTwo(card) : setChoiceOne(card)… if choiceOne is null it means that we don’t have a selection for choiceOne**
5. **If choiceOne evaluates as false - if it’s null - it means the code on the right side of the colon is run, re: choiceOne ? setChoiceTwo(card) : setChoiceOne(card)…’ re: choiceOne is updated.**
6. **If choiceOne is not null then it has a value and this means we have a selection for choiceOne so it evaluates as true. So what happens if it evaluates as true:**
7. **The code on the left of the colon is run and we update choiceTwo because we have our value for choiceOne**

**Summary: We’re handling our user selections and we’re updating either choiceTwo or choiceOne.**

**#6 Comparing Card Choices**

**Challenge 2: 0:00 – 2:21**