Hogwarts School of React Sorcery and Component Craft

Introduction: Welcome, Component Apprentice!

Welcome to Hogwarts School of React Sorcery and Component Craft. You have mastered the ancient art of SpellScript (JavaScript). Now, you will learn the most powerful form of modern magic: **React**—the art of building enchanted, living interfaces that respond to every whisper and gesture.

In the Muggle world, they call it a "JavaScript library for building user interfaces." But we know the truth: React is pure magic. It allows you to craft self-updating scrolls, interactive mirrors, and entire magical worlds that exist in the browser.

Each "Grimoire" contains powerful React spells. Each "Trial" will test your mastery. On the left, you'll study the ancient texts. On the right, your mystical workshop awaits.

Remember: "With great power comes great component architecture." Your journey into React sorcery begins now.

Grimoire I: The Foundations (React Basics)

This grimoire covers the core essence of React: Components, JSX, and the mystical art of rendering.

Trial 1: The First Component (Creating Components)

The Ancient Text (Docs - Left Side):

In the old world of SpellScript, everything was chaos—scattered functions and tangled DOM manipulation. React brings order through **Components**—self-contained magical artifacts that can be summoned and reused.

A Component is like a spell that returns a piece of the Living Parchment (HTML). We write them as **functions** that return **JSX** (JavaScript XML)—a magical syntax that looks like HTML but is actually JavaScript.

There are two ways to declare a Component:

Function Component (Modern Magic):

```
function WelcomeMessage() {
  return <h1>Welcome to Hogwarts!</h1>;
}
```

Arrow Function Component (The Quick-Quill Method):

```
const WelcomeMessage = () => {
```

```
  return <h1>Welcome to Hogwarts!</h1>;
};
```

To use your component, you "summon" it like a self-closing tag:

<WelcomeMessage />

Your Trial (Workshop - Right Side):

1. Create a function component called HouseWelcome
2. Make it return a containing:
    - An <h1> with the text "Welcome to Gryffindor!"
    - A with the text "Where dwell the brave at heart."
3. Export your component: export default HouseWelcome;

Animation Section (On Success): A magical scroll unfurls, revealing the HouseWelcome component code. The component materializes as a glowing card with the Gryffindor crest. When summoned with <HouseWelcome />, it duplicates itself three times, showing the power of reusability.

Trial 2: The Transfiguration Syntax (JSX)

The Ancient Text (Docs - Left Side):

JSX is React's special language. It *looks* like HTML, but it's secretly JavaScript in disguise. Professor McGonagall would call this "transfiguration."

Key Rules of JSX Magic:

1. **Single Parent Rule**: Every component must return ONE parent element
2. // ❌ Wrong - Multiple parents
3. return (
4. <h1>Title</h1>
5. Text
6.);
7.
8. // ✅ Correct - Wrapped in one parent
9. return (

10.
11. <h1>Title</h1>
12. Text
13.
14.);
15. **Self-Closing Tags**: Tags without children must self-close
16. // ✅ Correct
17. <img src="wand.png"> // ❌ Wrong
18. **JavaScript in Curly Braces**: Use {} to embed JavaScript
19. const spell = "Lumos";
20. return The spell is {spell}! ; // "The spell is Lumos!"
21. **className not class**: Use className for CSS classes
22. ...

Your Trial (Workshop - Right Side):

1. Create a component called SpellCard
2. Inside, create a constant: const spellName = "Wingardium Leviosa";
3. Return a with className="spell-card" containing:
    - An <h2> that displays the spellName using {}
    - A with the text "Levitation Charm"
    - An <img> tag (self-closing) with src="feather.png"

Animation Section (On Success): The component renders. The spellName variable glows in curly braces {}, then the actual text "Wingardium Leviosa" materializes inside the <h2>. The className transforms from "class" to "className" with a magical sparkle. The feather image floats upward, demonstrating levitation.

Trial 3: The Potions of Data (Props)

The Ancient Text (Docs - Left Side):

A component is powerful, but it's even better when it can accept ingredients—**Props** (properties). Props allow you to pass data from a parent component to a child component, like handing a student a potion recipe.

Declaring Props:

```
function PotionCard(props) {
  return (
    <div>
      <h2>{props.name}</h2>
      <p>Color: {props.color}</p>
    </div>
  );
}
```

Using Props (Summoning the Component):

```
<PotionCard name="Polyjuice" color="Muddy Brown" />
<PotionCard name="Felix Felicis" color="Molten Gold" />
```

Modern Destructuring (Advanced Students):

```
function PotionCard({ name, color }) {
  return (
    <div>
      <h2>{name}</h2>
      <p>Color: {color}</p>
    </div>
  );
}
```

Your Trial (Workshop - Right Side):

1. Create a component called StudentCard that accepts props
2. Use destructuring to extract name and house from props
3. Return a with className="student-card" containing:
    - An <h3> displaying the student's name

- A <p> displaying "House: {house}"

4. In your main component, summon <StudentCard /> three times with different names and houses:
   - <StudentCard name="Harry Potter" house="Gryffindor" />
   - <StudentCard name="Luna Lovegood" house="Ravenclaw" />
   - <StudentCard name="Cedric Diggory" house="Hufflepuff" />

Animation Section (On Success): Three empty card frames appear. The first <StudentCard /> summons with props flying in like owls delivering letters. "Harry Potter" and "Gryffindor" materialize in their respective slots. This repeats for Luna and Cedric, each card appearing with their house colors glowing.

Trial 4: The Remembrall (Rendering Lists)

The Ancient Text (Docs - Left Side):

What if you have an array of data—like 100 students—and you need to create a component for each one? You use the .map() spell to transform your array into an array of components.

The Map Spell:

```
const students = ["Harry", "Hermione", "Ron"];


function StudentList() {
  return (
    <ul>
      {students.map((student) => (
        <li key={student}>{student}</li>
      ))}
    </ul>
  );
}
```

⚠️ **The Key Curse:** React needs a unique key prop for each item in a list. Without it, your magic will work but with a warning curse. Use a unique identifier (like an ID) or the item itself if unique.

Your Trial (Workshop - Right Side):

1. Create an array of spell objects:
2. const spells = [
3. { id: 1, name: "Expelliarmus", type: "Disarming" },
4. { id: 2, name: "Protego", type: "Shield" },
5. { id: 3, name: "Stupefy", type: "Stunning" }
6.];
7. Create a component called SpellList
8. Use .map() to render each spell as a with:
    - key={spell.id}
    - className="spell-item"
    - Display the spell name and type

Animation Section (On Success): The array of spells appears as floating data objects. The .map() function activates, and a magical assembly line begins. Each spell object is transformed into a component with a glowing key attached like a magical ID tag. The three spell cards assemble themselves in a row.

Grimoire II: The State of Mind (React State)

You've learned to create static components. Now, you'll learn the most powerful magic of all: **State**—the ability to make your components remember and change.

Trial 1: The Pensieve (useState Hook)

The Ancient Text (Docs - Left Side):

In the old world, components were lifeless—they couldn't remember anything. React introduced **Hooks**—magical tools that give components memory and power. The first and most important is useState.

useState is like a Pensieve—it stores a memory (state) and gives you a spell to change it.

The Incantation:

```
import { useState } from 'react';


function Counter() {
```

```
  const [count, setCount] = useState(0);


  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}
```

Breaking Down the Spell:

- useState(0): Creates a state variable starting at 0
- [count, setCount]: Destructuring! count is the current value, setCount is the function to change it
- onClick={() => setCount(count + 1)}: When clicked, increase the count

Your Trial (Workshop - Right Side):

1. Import useState from 'react'
2. Create a component called HousePoints
3. Create a state variable called points starting at 0
4. Return a containing:
    - An <h2> displaying "Gryffindor Points: {points}"
    - A <button> that adds 10 points when clicked
    - A <button> that subtracts 5 points when clicked

Animation Section (On Success): A Gryffindor hourglass appears. The initial state useState(0) fills the hourglass to 0. When the "Add 10" button is clicked, the setPoints(points + 10) spell activates, and sand flows upward, updating the display to 10. When "Subtract 5" is clicked, sand flows down to 5. The component re-renders with each state change, shown as a magical shimmer.

Trial 2: The Marauder's Toggle (Boolean State)

The Ancient Text (Docs - Left Side):

State isn't just for numbers. The most common use is **boolean state** (true/false)—perfect for toggling visibility, like the Marauder's Map revealing its secrets.

The Toggle Pattern:

```
const [isVisible, setIsVisible] = useState(false);


function toggleVisibility() {
  setIsVisible(!isVisible); // ! flips true to false or false to true
}
```

Conditional Rendering:

```
{isVisible && <p>Secret message!</p>}
// This only renders the <p> if isVisible is true
```

Your Trial (Workshop - Right Side):

1. Create a component called MaraudersMap
2. Create a state variable isRevealed starting at false
3. Create a function revealMap that toggles isRevealed
4. Return a containing:
    - A <button> that calls revealMap with text: "I solemnly swear I am up to no good"
    - Conditionally render: If isRevealed is true, show a <p> with "The map reveals all footsteps!"
    - Conditionally render: If isRevealed is false, show a <p> with "The map is blank..."

Animation Section (On Success): A blank parchment (map) is shown. The state starts at false, and the map displays "The map is blank...". When the button is clicked, setIsRevealed(!isRevealed) activates. The boolean flips to true, and footprints magically appear on the map with the text "The map reveals all footsteps!". Clicking again makes them disappear.

Trial 3: The Potion Brewer (Complex State Objects)

The Ancient Text (Docs - Left Side):

Sometimes your state isn't a single value—it's an entire object, like a potion recipe with multiple ingredients. We can store objects in state!

Object State:

```
const [potion, setPotion] = useState({
  name: "Unknown",
  color: "Clear",
  temperature: 20
});
```

⚠️ **Important Rule**: Never mutate state directly! Always create a new object using the spread operator:

```
// ❌ Wrong
potion.temperature = 50;


// ✅ Correct
setPotion({ ...potion, temperature: 50 });
```

Your Trial (Workshop - Right Side):

1. Create a component called PotionBrewer
2. Create state for a potion object:
3. const [potion, setPotion] = useState({ name: "Felix Felicis", color: "Gold", temperature: 100});
4. Create a function heatPotion that increases temperature by 10
5. Create a function changeColor that sets color to "Molten Silver"
6. Return a displaying:
    - The potion name, color, and temperature
    - A button to heat the potion
    - A button to change the color

Animation Section (On Success): A cauldron with the potion object data displayed above it. When "Heat Potion" is clicked, the spread operator { ...potion, temperature:

potion.temperature + 10 } is visualized—the old potion is copied, then the temperature property is overwritten. The cauldron bubbles more vigorously. When "Change Color" is clicked, the potion liquid animates from gold to silver.

Trial 4: The Goblet of Fire (Array State)

The Ancient Text (Docs - Left Side):

What if your state is an array, like a list of champions? Array state is extremely common in React—for todo lists, shopping carts, user lists, and more.

Array State Patterns:

Adding an item:

```
const [champions, setChampions] = useState(["Harry"]);


function addChampion(name) {
  setChampions([...champions, name]); // Spread old array, add new item
}
```

Removing an item:

```
function removeChampion(name) {
  setChampions(champions.filter(champ => champ !== name));
}
```

Your Trial (Workshop - Right Side):

1. Create a component called ChampionsList
2. Create state for an array starting with ["Cedric", "Fleur", "Viktor"]
3. Create an input field (controlled component style—we'll learn this next!)
4. Create a function addChampion that adds a new name to the array
5. Create a function removeChampion that removes a champion by name
6. Display the list using .map()
7. Add buttons to test adding "Harry" and removing "Viktor"

Animation Section (On Success): The Goblet of Fire burns with three names floating above it. When "Add Harry" is clicked, the spread operator [...champions, "Harry"] is visualized—

the old array is copied, and "Harry" is appended to the end. His name flies out of the flames. When "Remove Viktor" is clicked, the .filter() spell activates, and Viktor's name dissolves away, leaving the filtered array.

Grimoire III: The Two-Way Mirror (Forms & Events)

Components that respond to user input are the heart of React. You'll learn to capture events, control forms, and create interactive experiences.

Trial 1: The Howler (Event Handling)

The Ancient Text (Docs - Left Side):

In React, we attach event handlers directly to elements using camelCase attributes like onClick, onChange, onSubmit, etc.

The Pattern:

```
function HowlerButton() {
  function scream() {
    alert("RONALD WEASLEY! HOW DARE YOU!");
  }

  return <button onClick={scream}>Open Howler</button>;
}
```

Or inline:

```
<button onClick={() => alert("HOWLER!")}>Open Howler</button>
```

Your Trial (Workshop - Right Side):

1. Create a component called SpellCaster
2. Create a state variable lastSpell starting at "None"
3. Create functions for three spells:
    - castLumos(): Sets lastSpell to "Lumos - Light!"
    - castNox(): Sets lastSpell to "Nox - Darkness!"
    - castExpelliarmus(): Sets lastSpell to "Expelliarmus - Disarm!"
4. Display the lastSpell in an <h2>

5. Create three buttons, each casting a different spell when clicked

Animation Section (On Success): Three wands appear as buttons. When "Lumos" is clicked, the onClick event fires, the castLumos function runs, setLastSpell updates the state, and a bright light emanates from the wand. The component re-renders, showing "Lumos - Light!" in the display. Each button produces its own visual effect.

Trial 2: The Prophecy Orb (Controlled Inputs)

The Ancient Text (Docs - Left Side):

In React, form inputs should be **controlled components**—their value is controlled by state. This creates a "single source of truth."

The Controlled Input Pattern:

```
const [prophecy, setProphecy] = useState("");


return (
  <input
    type="text"
    value={prophecy}
    onChange={(e) => setProphecy(e.target.value)}
  />
);
```

Breaking it down:

- value={prophecy}: The input's value is controlled by state
- onChange={(e) => setProphecy(e.target.value)}: When typing, update state
- e.target.value: The current value of the input

Your Trial (Workshop - Right Side):

1. Create a component called ProphecyWriter
2. Create state variables:
    - prophecyText (string, starts empty)
    - savedProphecy (string, starts empty)

3. Create a controlled input that updates prophecyText as you type
4. Create a button "Save Prophecy" that copies prophecyText to savedProphecy
5. Display both the current input text and the saved prophecy
6. Add a "Clear" button that resets both to empty strings

Animation Section (On Success): A glowing crystal orb (input field) appears. As the user types, each keystroke triggers onChange, which updates prophecyText state, which flows back to the input's value. When "Save Prophecy" is clicked, the text is copied to savedProphecy and appears in a sealed scroll. The two-way data flow is visualized with glowing arrows.

Trial 3: The Sorting Ceremony (Form Submission)

The Ancient Text (Docs - Left Side):

Forms in React use the onSubmit event. Always remember to prevent the default behavior (page refresh) with e.preventDefault().

The Form Pattern:

```
function SortingForm() {
  const [name, setName] = useState("");
  const [house, setHouse] = useState("");

  function handleSubmit(e) {
    e.preventDefault(); // CRITICAL!
    setHouse(determineHouse(name));
  }

  return (
    <form onSubmit={handleSubmit}>
      <input value={name} onChange={(e) => setName(e.target.value)} />
      <button type="submit">Sort Me!</button>
    </form>
```

```
  );
}
```

Your Trial (Workshop - Right Side):

1. Create a component called SortingHat
2. Create state for studentName, trait (brave/wise/loyal), and sortedHouse
3. Create a form with:
    - An input for student name (controlled)
    - A select dropdown for trait (controlled):
        - "brave" → Gryffindor
        - "wise" → Ravenclaw
        - "loyal" → Hufflepuff
        - "ambitious" → Slytherin
    - A submit button
4. On submit:
    - Prevent default
    - Set sortedHouse based on the selected trait
5. Display the result: "{studentName} is sorted into {sortedHouse}!"

Animation Section (On Success): A form appears with the Sorting Hat above it. As the user fills out the form, the controlled inputs update state. When submitted, e.preventDefault() stops the page refresh (shown as a blocked browser reload icon). The sorting logic runs, and the hat "thinks" before announcing the house. A house banner unfurls with the result.

Grimoire IV: The Life Cycle (useEffect Hook)

Components aren't just static—they have a life cycle. They mount (appear), update (change), and unmount (disappear). The useEffect hook lets you tap into these moments.

Trial 1: The Room of Requirement (useEffect Basics)

The Ancient Text (Docs - Left Side):

useEffect is a hook that runs **side effects**—code that interacts with the outside world, like fetching data, setting timers, or manipulating the DOM.

The Basic Incantation:

```
import { useEffect } from 'react';


useEffect(() => {
  console.log("Component mounted!");
}, []);
```

The Dependency Array ([]):

- []: Run only once when component mounts
- No array: Run after every render
- [count]: Run when count changes

Your Trial (Workshop - Right Side):

1. Create a component called MagicClock
2. Create a state variable time starting at 0
3. Use useEffect with an empty dependency array [] to:
    - Log "Clock started!" to the console
    - Set up an interval that increases time by 1 every second
    - Return a cleanup function that clears the interval
4. Display the current time in seconds

Animation Section (On Success): The component mounts (appears). The useEffect with [] runs once—a clock materializes and starts ticking. Every second, the state updates, and the component re-renders with the new time. When the component unmounts (disappears), the cleanup function runs, and the clock stops and fades away.

Trial 2: The Marauder's Map (useEffect Dependencies)

The Ancient Text (Docs - Left Side):

The real power of useEffect is watching for changes. If you include variables in the dependency array, the effect runs whenever those variables change.

Watching State:

```
const [location, setLocation] = useState("Great Hall");
```

```
useEffect(() => {
  console.log(`You moved to ${location}`);
}, [location]); // Runs when location changes
```

Your Trial (Workshop - Right Side):

1. Create a component called FootprintTracker
2. Create state for currentLocation starting at "Great Hall"
3. Create state for visitedLocations (array) starting empty
4. Use useEffect that watches currentLocation:
    - When location changes, add it to the visitedLocations array
    - Log "New footprint detected at: {location}"
5. Create buttons to change location to different places (Library, Dungeons, etc.)
6. Display the list of visited locations

Animation Section (On Success): The Marauder's Map is shown. When location changes (button clicked), the useEffect dependency triggers—footprints appear on the map at the new location, and the location is added to the visited array. The dependency array [currentLocation] is shown glowing, indicating it's being watched for changes.

Trial 3: The Prophecy Orb (Fetching Data)

The Ancient Text (Docs - Left Side):

The most common use of useEffect is fetching data from an API when a component mounts. This is asynchronous magic!

The Fetch Pattern:

```
const [prophecy, setProphecy] = useState(null);
const [loading, setLoading] = useState(true);

useEffect(() => {
  async function fetchProphecy() {
    const response = await fetch("https://api.hogwarts.dev/prophecy");
```

```
    const data = await response.json();
    setProphecy(data.text);
    setLoading(false);
  }

  fetchProphecy();
}, []);
```

Your Trial (Workshop - Right Side):

1. Create a component called DailyProphet
2. Create state for articles (array, starts empty) and loading (boolean, starts true)
3. Use useEffect to fetch from a mock API (use a setTimeout to simulate):
4. useEffect(() => { setTimeout(() => { setArticles([{ id: 1, headline: "Harry Potter Defeats Dark Lord!" }, { id: 2, headline: "Hogwarts Reopens After Battle" }]); setLoading(false); }, 2000);}, []);
5. Show "Loading..." while loading is true
6. When loaded, map through articles and display them

Animation Section (On Success): An empty newspaper appears. The component mounts, useEffect runs with [], and a loading spinner appears. After 2 seconds (simulating API), the articles arrive, setArticles updates state, setLoading(false) removes the spinner, and the newspaper fills with headlines. The async flow is visualized as an owl flying to get the news.

Grimoire V: The Forbidden Forest (Advanced Patterns)

You've mastered the fundamentals. Now venture into advanced territory—custom hooks, context, and complex state management.

Trial 1: The Elder Wand (Custom Hooks)

The Ancient Text (Docs - Left Side):

Custom hooks are your own reusable spells. They're just functions that use other hooks. They must start with "use" to follow React's rules.

Creating a Custom Hook:

```
function useCounter(initialValue = 0) {
```

```
  const [count, setCount] = useState(initialValue);

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);
  const reset = () => setCount(initialValue);

  return { count, increment, decrement, reset };
}

// Using it:
function MyComponent() {
  const counter = useCounter(10);

  return <button onClick={counter.increment}>{counter.count}</button>;
}
```

Your Trial (Workshop - Right Side):

1. Create a custom hook called useHousePoints
2. It should accept an initial house name
3. Inside the hook, create state for:
    - points (starting at 0)
    - house (the initial house name)
4. Create functions: addPoints(amount), subtractPoints(amount), setHouse(newHouse)
5. Return an object with all state and functions
6. Use your hook in a component called HousePointsTracker
7. Display the house name and points, with buttons to add/subtract

Animation Section (On Success): A wand (custom hook) is crafted. Inside it, multiple hooks (useState) are combined. The wand is then "gifted" to multiple components—each component uses the same hook but has its own independent state. Multiple house point trackers appear, all powered by the same custom hook logic.

Trial 2: The Room of Requirement (React Context)

The Ancient Text (Docs - Left Side):

Passing props down through many levels is tedious ("prop drilling"). **Context** solves this—it's like a magical room that appears wherever you need it.

Creating Context:

```
import { createContext, useContext, useState } from 'react';


const ThemeContext = createContext();


function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");


  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}


// Using it:
function ThemedButton() {
  const { theme, setTheme } = useContext(ThemeContext);
  return <button>{theme}</button>;
}
```

Your Trial (Workshop - Right Side):

1. Create a HouseContext using createContext()
2. Create a HouseProvider component that:

    - Has state for currentHouse
    - Provides currentHouse and setCurrentHouse via Context
3. Create a HouseSwitcher component that:
    - Uses useContext to access and change the house
    - Has buttons for all four houses
4. Create a HouseBanner component that:
    - Uses useContext to display the current house
    - Shows the house crest
5. Wrap your app in the HouseProvider

Animation Section (On Success): The app is wrapped in a glowing Context sphere. Inside, multiple deeply nested components exist. When one component changes the context (switches houses), the change ripples through the entire sphere, instantly updating all components that use the context—without passing props through the tree. The "prop drilling" solution is shown side-by-side, with props being manually passed down 5 levels, then crossed out.

Trial 3: The Horcrux (useReducer)

The Ancient Text (Docs - Left Side):

When state gets complex with multiple related values and operations, useReducer is more powerful than useState. It's like having a spellbook of actions.

The Pattern:

```
const initialState = { count: 0, name: "" };


function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { ...state, count: state.count + 1 };
    case 'setName':
      return { ...state, name: action.payload };
    default:
```

```
    return state;
  }
}

function MyComponent() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <button onClick={() => dispatch({ type: 'increment' })}>
      {state.count}
    </button>
  );
}
```

Your Trial (Workshop - Right Side):

1. Create a component called InventoryManager
2. Define initial state:
3. { gold: 100, items: [], spellCount: 0}
4. Create a reducer function handling:
    - 'ADD_GOLD': Adds 50 gold
    - 'SPEND_GOLD': Subtracts 20 gold (if available)
    - 'ADD_ITEM': Adds an item to the array
    - 'LEARN_SPELL': Increments spellCount
5. Use useReducer and create buttons to dispatch each action
6. Display the current state

Animation Section (On Success): A wizard's inventory screen appears. When "Add Gold" is clicked, an action { type: 'ADD_GOLD' } is dispatched to the reducer. The reducer function activates like a spellbook, finds the matching case, and returns the new state. The gold counter animates upward. Multiple actions can be dispatched in sequence, each one handled by the reducer's switch statement.

Grimoire VI: The Final Challenge (Capstone Project)

Introduction: "You have journeyed far, young sorcerer. You've mastered components, state, effects, and advanced patterns. Now comes your final trial—a test that combines everything you've learned. Build the Hogwarts House Cup Tracker, a complete React application."

The Ultimate Trial: House Cup Championship Tracker

The Grand Scroll (Docs - Left Side):

The Headmaster needs a new system to track the House Cup competition throughout the year. Your task is to build a complete React application with these features:

Requirements:

1. **Display all four houses** with their current points
2. **Add/subtract points** from any house with a reason
3. **Show a history log** of all point changes
4. **Crown the winner** - highlight the house with the most points
5. **Persist data** - use localStorage so data survives page refresh
6. **Responsive design** - works on all devices

Technical Requirements:

- Use **multiple components** (HouseCard, PointsHistory, AddPointsForm, etc.)
- Use **useState** for managing points and history
- Use **useEffect** to load/save from localStorage
- Use **props** to pass data between components
- Use **event handlers** for all interactions
- Use **conditional rendering** to show the winner
- Use **.map()** to render lists
- Use **forms** with controlled inputs

Your Trial (Workshop - Right Side):

Build the complete application! Here's a suggested structure:

// Suggested Components:

// 1. App - Main component

```
// 2. HouseCard - Displays a single house's points
// 3. AddPointsForm - Form to add/subtract points
// 4. HistoryLog - Shows all point changes
// 5. WinnerBanner - Displays the current leader

// Suggested State Structure:
{
  houses: [
    { id: 1, name: "Gryffindor", points: 0, color: "#740001" },
    { id: 2, name: "Slytherin", points: 0, color: "#1a472a" },
    { id: 3, name: "Ravenclaw", points: 0, color: "#0e1a40" },
    { id: 4, name: "Hufflepuff", points: 0, color: "#ecb939" }
  ],
  history: [
    { id: 1, house: "Gryffindor", points: 10, reason: "Bravery", timestamp: Date }
  ]
}
```

Bonus Challenges:

- Add animations when points change
- Add sound effects for adding/subtracting points
- Add a "Reset All" button with confirmation
- Add filtering to the history (by house, by date)
- Add a dark mode toggle
- Add achievements (e.g., "First house to reach 100 points")

Animation Section (On Success): The grand finale! A complete application builds itself piece by piece. The component tree is visualized - App at the top, branching to HouseCard components, AddPointsForm, and HistoryLog. State flows down as props (shown as glowing rivers). Events flow up (shown as spell bolts). When points are added, the change ripples through the state, triggers a re-render, updates localStorage, and animates the UI. The house

with the most points is crowned with a glowing winner badge. A "N.E.W.T.s in React - OUTSTANDING!" certificate appears.

Appendix: Quick Reference Grimoire

Hook Incantations

useState:

```
const [value, setValue] = useState(initialValue);
```

useEffect:

```
useEffect(() => {
  // Effect code
  return () => {/* cleanup */};
}, [dependencies]);
```

useContext:

```
const value = useContext(MyContext);
```

useReducer:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

Common Patterns

Controlled Input:

```
<input
  value={value}
  onChange={(e) => setValue(e.target.value)}
/>
```

Conditional Rendering:

```
{condition && <Component />}
{condition ? <ComponentA /> : <ComponentB />}
```

List Rendering:

```
{array.map(item => (
  <Component key={item.id} data={item} />
```

))}

Event Handling:

<button onClick={() => handleClick(param)}>Click</button>

Graduation

Congratulations, Master of React Sorcery! You have completed your training and are now ready to build powerful, interactive magical applications. Go forth and create wonders!

Remember the words of the great Albus Component-dore:

"Happiness can be found, even in the darkest of bugs, if one only remembers to console.log()."

Your next steps:

1. Build real projects to practice
2. Explore React Router for multi-page apps
3. Learn state management libraries (Redux, Zustand)
4. Master TypeScript for type-safe magic
5. Explore Next.js for server-side sorcery

May your components always render, and your state never be undefined!

🪄 **Mischief Managed** 🪄