46.5 / 66 70 %.

TaskTable Integration Test Quiz

Multiple Choice

1. 1. What does `screen.getByRole('button', { name: /career/i })` do?
   A. Selects a button with exact text "career"
   B. Selects a button containing the word "Career", case-insensitive
   C. Selects a span with role "button"
   D. Finds a button by its CSS class name

2. Why does the following test pass even though the `keyboard('{Backspace}')` action is called? (From TaskTable.test.jsx)

```js
const mockSetTasks = vi.fn();
const sampleTask = { id: 1, name: 'Task', selected: false };
render(
  <TaskContext.Provider value={{ tasks: [sampleTask], setTasks: mockSetTasks }}>
    <TaskTable />
  </TaskContext.Provider>
);

await user.keyboard('{Backspace}');
expect(mockSetTasks).not.toHaveBeenCalled();
```

A. The mock function fails silently
B. The component unmounts before the call happens
C. The deletion logic only triggers if a task is marked as `selected: true`
D. `keyboard` is not supported by the testing library

Reference the images below for the following questions

```
const calcSum = useMemo(() =>
  tasks.reduce((acc, task) => acc + (Number(task.time_estimation) || 0), 0),
  [tasks]
);
```

```
// 🔻 Handle Backspace/Delete key to remove selected tasks
useEffect(() => {
  const handleKeyDown = (e) => {
    const { key } = e;
    const exists = tasks.some(task => task.selected);
    if (exists && (key === "Delete" || key === "Backspace")) {
      setTasks(prev => prev.filter(task => !task.selected));
      e.preventDefault();
    }
  };

  window.addEventListener("keydown", handleKeyDown);
  return () => window.removeEventListener("keydown", handleKeyDown);
}, [tasks, setTasks]);
```

```
// ✅ 🔻 Handle deselection when clicking outside row selectors
useEffect(() => {
  const handleMouseDown = (e) => {
    // If user clicked inside *any* RowSelector, skip
    const isInsideRowSelector = e.target.closest('[data-role="row-selector"]');
    const isInsideDropdownOrDate = e.target.closest('[data-role="popup"]');

    if (isInsideRowSelector || isInsideDropdownOrDate) return;

    // Defer to next event loop tick to avoid race condition with onChange
    setTimeout(() => {
      const anySelected = tasks.some(task => task.selected);
      if (anySelected) {
        setTasks(prev => prev.map(task => ({ ...task, selected: false })));
      }
    }, 0);
  };

  document.addEventListener('mousedown', handleMouseDown);
  return () => document.removeEventListener('mousedown', handleMouseDown);
}, [tasks, setTasks]);
```

3. What is the primary purpose of the `useMemo` hook in this context?

A. To make `calcSum` recompute every time any key is pressed
B. To improve performance by memoizing the computed sum unless `tasks` changes
C. To avoid re-rendering the table when tasks are deleted
D. To initialize task estimation with a default value

4. What does `(Number(task.time_estimation) || 0)` accomplish inside the reducer?

A. Converts falsy values to null
B. Ensures only string values are added
C. Guards against `undefined`, `null`, or non-numeric values
D. Throws an error if `time_estimation` is missing

5. Why is `e.preventDefault()` used after detecting a Delete or Backspace keypress?

A. To prevent the browser from navigating back or deleting a focused input field
B. To stop the task from being added
C. To block React from re-rendering the component
D. To simulate a database call

6. Which hook adds and cleans up a global keyboard event listener?

A. `useReducer`
B. `useCallback`
C. `useEffect`
D. `useMemo`

7. What does `e.target.closest('[data-role="row-selector"]')` check for in the second `useEffect`?

A. Whether the event came from a button element
B. Whether the clicked element is within a dropdown
C. Whether the clicked target is inside a row selector element
D. Whether the document was clicked at all

8. Why is `setTimeout(..., 0)` used before deselecting tasks?

A. To wait 1 second
B. To debounce the deselection logic
C. To defer execution until after onChange finishes to prevent a race condition
D. To allow React to cancel the action

Short Answer

9. Explain what `vi.fn()` is and how it's used in this context (From TaskTable.test.jsx):

```
const mockSetTasks = vi.fn();
renderWithContext(sampleTasks, mockSetTasks);

const textbox = screen.getByRole('textbox');
await user.clear(textbox);
await user.type(textbox, 'Updated Name');

expect(mockSetTasks).toHaveBeenCalled();
```

- mock function
- vi is a virtual mock object used for easily facilitating mock use of functions, APIs, etc for easier testing
- used to ensure that the passed function is called at appropriate times

10. Why is `{ name: /career/i }` written with slashes and the `i` flag in the role query?

- / / indicates a regex pattern match for finding strings that match a pattern, more flexible way of searching
- /i indicates case 'insensitive', will ensure the pattern matches regardless of 'CaREer', 'career', 'CAREER'

-1

11. What does the following test verify? (From TaskTable.test.jsx)

```js
const mockSetTasks = vi.fn();
const selectedTask = { id: 1, name: 'To Delete', selected: true, time_estimation: 10 };

renderWithContext([selectedTask], mockSetTasks);
await user.keyboard('{Backspace}');

const updateFn = mockSetTasks.mock.calls[0][0];
const result = updateFn([selectedTask]);
expect(result).toEqual([]);
```

12. Explain why `[tasks, setTasks]` appears in the dependency array of both `useEffect` hooks.
What would happen if you omitted them?

- they reference the 'selected' property of task objects so they must include tasks in the dependency array or the useEffect would never trigger when a task is selected

13 What side effects do each `useEffect` implement in plain language?
Give one sentence for each `useEffect` block summarizing what they accomplish.

① deletes selected tasks when a 'delete' or 'backspace' key is detected
② deselects all tasks if anything other than a row selector is clicked

14 Describe how deselection is handled when clicking outside a row. Why not just use `onBlur` instead?

-0.5

- the useEffect func doesn't immediately return & proceeds to the rest of code block, then it iterates through all tasks & changes the selected property to false

`onBlur - not sure

15. What behavior would break if you forgot to remove the event listeners in the cleanup function of either `useEffect`?
Explain how this might affect performance or user interaction over time.

- selecting / deselecting items or deleting items would stop working
- Each click or key type event would significantly slow everything down
- Random tasks may be deleted
- App may crash or time programs

16. In the test file, the NameInput component is rendered through the following wrapper:

```
const TestWrapper = ({ defaultTask }) => {
  const [task, setTask] = useState(defaultTask);

  const updateTaskField = (id, field, value) => {
    setTask((prev) => ({
      ...prev,
      [field]: value,
    }));
  };

  return <NameInput task={task} updateTaskField={updateTaskField} />;
};
```

Explain why the TestWrapper component is necessary for testing NameInput. Additionally, explain why the updateTaskField function must accept (id, field, value) as parameters and use dynamic property access like [field]: value. What would break if either of these patterns were changed?

① hooks can't be used in tests so useState can't be used. This means that dynamic changes to inputs won't update state & the functionality of the component when interacted with in a test can't be tested. TestWrapper solves this problem b/c it's defined outside the test suite & can use hooks to maintain state.

② (id, field, value) must be parameters b/c in the NameInput.jsx component file, all 3 of these params are referenced. The id ensures that the currently passed task is updated if need be, the field indicates which attribute of the task to change, & the value indicates what to change the task fields attribute to. This ensures that the NameInput component is stateless, & correctly forwards jsx. [field]: value is necessary for creating a function that can be passed to all components that trigger state changes for all components, dynamic task property updates

③ tasks wouldn't be able to be updated (no id), when an update attempt does occur, since no task object in the tasks state array will match an id of undefined so the state updating function will never trigger a change

Coding Exercises (5pts)

1. `useMemo` Application

Implement a React component called `SumDisplay` that accepts a prop called `tasks` (an array of objects, each with a `time_estimation` field).

Use `useMemo` to compute and display the total time, ensuring it only recalculates when `tasks` changes.

useMemo(callback, dependency array)

```
const SumDisplay = ({tasks}) => {

const calcSum = useMemo(() =>
    tasks.reduce((acc, task) => acc + task.time_estimation, 0),
    [tasks]);

return <h1>{calcSum}</h1>;

}
```

5/5

2. Task Deletion with Key Press

Create a React component that renders a list of tasks, each with a checkbox to mark it as "selected". Use `useEffect` to add a global keydown listener that deletes all selected tasks when the **Delete** or **Backspace** key is pressed.

(10 pts)

```
const TaskTable = ([]) => {                          needs at least one object in array defined
  -0.5  const [tasks, setTasks] = useState([]);

  -2  [ const handleSelection = (task) => {
          setTasks((prev) => [...prev, {...task, selected: !task.selected}])
        }

  useEffect(() => {
    const handleKeyPress = (e) => {
      const isDelete = e.key === "Delete" || e.key === "Backspace";
      if (isDelete) {
        setTasks(tasks.filter(task => !task.selected))
      }
    -1 doesn't use callback prev => prev.filter
    window.addEventListener('keydown', handleKeyPress);
    return window.removeEventListener('keydown', handleKeyPress);
  }, [tasks])
  -1 forgot dependency array

  return (
    <Table>
      {tasks.map(task => (
        <TableRow>
          <TableCell>
            <input type="checkbox" value={task.selected} onChange={handleSelection} />
          </TableCell>
        </TableRow> ) }
    </Table> );

good [ ]
```

-4.5 → 5.5 / 10

3. Deselect on Outside Click

Using `useEffect`, implement logic to deselect all selected tasks if the user clicks outside of a specific DOM element (e.g., a div with `data-role="row-selector"`).

Use `e.target.closest(...)` to detect click context and `setTimeout(..., 0)` to avoid race conditions.

(10 pts)

```
useEffect( () => {
    const handleSelectClick = (e) => {
-1      const isRowSelector = e.target.closest( ) === data-role="row-selector"
.closest(...)   const isCheckbox = e.target.closest() === (input.type === "checkbox")
good [      if (isRowSelector || isCheckbox) {
            return;
        }
        else {
-3          tasks.map( task =>
no              { ...task, selected: false }
finecut     )
        }

    window.addEventListener('mousedown', handleSelectClick);
-1  return window.removeEventListener('mousedown', handleSelectClick);
3, [tasks] }
```

return () => document.remove...

-5

5/10

4. Test Wrapper Setup

Create a `TestWrapper` component in a test file for a component called `NameInput`.

The `NameInput` component takes two props:

- `task` – an object with a `name` property
- `updateTaskField` – a function that updates task fields

Write the wrapper and a test using `@testing-library/react` that:

- Renders the component
- Simulates typing a new name
- Asserts that the input reflects the updated name

(10 pts)

-5

5/10

-0.5, needs type or prop

```
const TestWrapper = ({initialTask}) => {
  const [tasks, setTask] = useState(initialTask);
  const handleNameInput = (e) => {
    setTask(prev => {...prev, name: e.value})
  }
  return(
    <NameInput task={task}
    updateTaskField={handleNameInput} />
  );
};
```

- func needs -2 to create formatting and in the NameInput function/file

```
if("renders with the correct test", async () => {
  const nameInput = screen.getByRole("textbox");
  const user = userEvent.setup
  await user.type("NewTask name (23", nameInput);
  expect(nameInput).toHaveAttribute('value', 'New task Name');
});
```

-0.5 expects cleared textbox

-0.5 didn't render TestWrapper

-0.5 Passes flipped

-1 to toHaveAttribute

5. Build a Field Updater Function

Write a reusable function `updateTaskField(id, field, value)` that takes in the current task state and returns a new state with the specified field updated.

Write a test that confirms:

- Updating `'name'` works
- Updating `'time_estimation'` works
- Other fields remain unchanged

(10 pts)

-4.5

5.5/10

```
const updateTaskField(id, field, value) => {
  setTasks(tasks.map(prev => {
    if (id === prev.id) {
      return {...prev, [field]: value};
    else {
      return prev
    }
  }))
}
```

State not defined -3

```
it("correctly updates the field of the correct task (not the other)",
  () => {
    const testTask = {id:1, name: "test 123", "time-estimation":90,
                      category:"career"};
    const [tasks, setTasks] = act(() => useState([testTask]));
    updateTaskField(1,"name", "new name");
    expect(tasks).toEqual({id:1, name:"new name", te:90, cat:"career"})
    updateTaskField(1, "te", 180);
    expect(tasks).toEqual({id:1, name:"new", te:180, cat:"career"})
```

Const act works in tests -1

console.assert -0.5

6. Regex Matching with `getByRole`

Write a unit test using React Testing Library to verify that a button labeled `"Career"` is rendered.

Use `getByRole('button', { name: /career/i })`.

Add a variant where the label is `"CAREER"` and show that the same test still passes.

(5 pts)

4.5/5

```
it ("finds & renders a button", ()=> {
    render (
      <label for Html="career-button"> Career </label>
      <button id="career-button"> Career </button>
    );

    Const button = Screen.getByRole ('button', {name: /career/i})
    expect (button). to Be In The Document();

    Const label = screen. getByText ("Career Button");
    label. textContent = "CAREER"

    expect (button). to Be In The Document();
)
```

- -0.5 button label just refers to it's inner text

Section 2: Answer Key

1. B – Selects a button containing the word “Career”, case-insensitive
screen.getByRole('button', { name: /career/i }) uses a case-insensitive regex to find a button labeled “Career”.

2. C – The deletion logic only triggers if a task is marked as selected: true
The test verifies that nothing is deleted unless a task is selected.

3. B – To improve performance by memoizing the computed sum unless tasks changes
useMemo is used to avoid recalculating the sum on every render unless tasks has changed.

4. C – Guards against undefined, null, or non-numeric values
Number(...) || 0 ensures a valid number is added to the total even if a task has an invalid or missing time_estimation.

5. A – To prevent the browser from navigating back or deleting a focused input field
e.preventDefault() blocks native behavior that could interfere with custom task deletion.

6. C – useEffect
This hook is responsible for managing side effects, including adding/removing event listeners.

7. C – Whether the clicked target is inside a row selector element
This ensures we don’t deselect tasks if the user clicked inside a multi-select checkbox area.

8. C – To defer execution until after onChange finishes to prevent a race condition
Using setTimeout(..., 0) lets UI events complete before deselection logic runs.

9.
vi.fn() is a mock function from Vitest.
It’s used to spy on setTasks to confirm it’s called when the user interacts with the input (like typing a new task name).

10.
{ name: /career/i } uses a regular expression with the i flag for case-insensitive matching.
This allows the test to match "Career", "career", or "CAREER" consistently.

11.
The test verifies that selected tasks are removed when Delete or Backspace is pressed.
It inspects whether setTasks filters out all task.selected === true entries.

12.
Including [tasks, setTasks] in the dependency array ensures the useEffect hooks respond to changes.
Omitting them could cause the event listeners to reference stale state or behave inconsistently.

13.

- First useEffect: Adds a global keyboard listener for Delete/Backspace to remove selected tasks.
- Second useEffect: Adds a mousedown listener that deselects all tasks when clicking outside a selector or popup.

14.
Deselection is triggered by detecting clicks outside any row selector or popup using .closest().
onBlur wouldn't work reliably across dropdowns or calendar popups, as focus changes too easily between elements.

15.
Forgetting to clean up event listeners would cause memory leaks or repeated handler calls. Over time, tasks might be deleted multiple times or undesired state might persist due to outdated event logic.

16.

The TestWrapper component is necessary because NameInput relies on receiving a task object and an updateTaskField function via props. Since NameInput is a controlled input component, it doesn't manage its own internal state—instead, it expects its parent to pass updated props when the input changes. TestWrapper fulfills this role by:

1. **Providing local state (task)** using useState, which allows the test to simulate and track changes.
2. **Implementing the updateTaskField function**, which updates the specific field (name) in the task when the input is edited.

The reason updateTaskField must have the signature (id, field, value) and use dynamic property access ([field]: value) is because:

- It is designed to be a **generic field updater** for any task field (e.g., name, date, time_estimation), not just the name field.
- The NameInput component calls updateTaskField(task.id, "name", e.target.value). If the function didn't accept these three parameters or didn't use [field]: value, the test would break or fail silently because:
    - The state wouldn't update correctly,
    - The component wouldn't re-render with the new input value,
    - And the input box would appear non-editable or stale in tests.

In summary, the wrapper simulates parent behavior, and the field-based update logic mirrors the app's actual dynamic state update patterns, making the test meaningful and realistic.

```
const TestWrapper = ({ defaultTask }) => {
  const [task, setTask] = useState(defaultTask);

  const updateTaskField = (id, field, value) => {
    setTask((prev) => ({
      ...prev,
      [field]: value,
    }));
  };

  return <NameInput task={task} updateTaskField={updateTaskField} />;
};
```

Coding Exercises

1.

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

const SumDisplay = ({ tasks }) => {
  const totalTime = useMemo(() => {
    return tasks.reduce((sum, task) => sum + (Number(task.time_estimation) || 0), 0);
  }, [tasks]);

  return <div>Total Time: {totalTime}</div>;
};
```

2

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

const TaskList = () => {
  const [tasks, setTasks] = useState([
    { id: 1, name: 'A', selected: false },
    { id: 2, name: 'B', selected: false }
  ]);

  useEffect(() => {
    const handleKeyDown = (e) => {
      if (e.key === 'Delete' || e.key === 'Backspace') {
        const anySelected = tasks.some(task => task.selected);
        if (anySelected) {
          setTasks(prev => prev.filter(task => !task.selected));
          e.preventDefault();
        }
      }
    };

    window.addEventListener('keydown', handleKeyDown);
    return () => window.removeEventListener('keydown', handleKeyDown);
  }, [tasks]);
```

```
  return (
    <ul>
      {tasks.map(task => (
        <li key={task.id}>
          <input
            type="checkbox"
            checked={task.selected}
            onChange={() =>
              setTasks(prev =>
                prev.map(t => t.id === task.id ? { ...t, selected: !t.selected } : t)
              )
            }
          />
          {task.name}
        </li>
      ))}
    </ul>
  );
};
```

3

jsx

Copy Edit

```jsx
useEffect(() => {
  const handleMouseDown = (e) => {
    const insideRow = e.target.closest('[data-role="row-selector"]');
    if (insideRow) return;

    setTimeout(() => {
      const anySelected = tasks.some(task => task.selected);
      if (anySelected) {
        setTasks(prev => prev.map(task => ({ ...task, selected: false })));
      }
    }, 0);
  };

  document.addEventListener('mousedown', handleMouseDown);
  return () => document.removeEventListener('mousedown', handleMouseDown);
}, [tasks, setTasks]);
```

4

```jsx
// TestWrapper.jsx
const TestWrapper = ({ defaultTask }) => {
  const [task, setTask] = useState(defaultTask);

  const updateTaskField = (id, field, value) => {
    setTask(prev => ({
      ...prev,
      [field]: value
    }));
  };

  return <NameInput task={task} updateTaskField={updateTaskField} />;
};

// NameInput.test.jsx
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';

it('updates the task name on input change', async () => {
  const user = userEvent.setup();
  render(<TestWrapper defaultTask={{ id: 1, name: 'Initial' }} />);

  const textbox = screen.getByRole('textbox');
  await user.clear(textbox);
  await user.type(textbox, 'Updated');

  expect(textbox).toHaveValue('Updated');
});
```

5.

```js
const updateTaskField = (task, field, value) => {
  return {
    ...task,
    [field]: value
  };
};

// Test
const original = { id: 1, name: 'X', time_estimation: 20 };
const updated = updateTaskField(original, 'name', 'Y');
console.assert(updated.name === 'Y');
console.assert(updated.time_estimation === 20);
```

6.

```jsx
// Component
const ButtonComponent = () => (
  <button>CAREER</button>
);

// Test
import { render, screen } from '@testing-library/react';

test('finds career button case-insensitively', () => {
  render(<ButtonComponent />);
  const btn = screen.getByRole('button', { name: /career/i });
  expect(btn).toBeInTheDocument();
});
```