Gagan Doddanna

React + TypeScript Todo App – CRUD Functionality with Explanations and Interview Q&A

---

1. Defining the `Todo` Type

What I did:

I created a TypeScript interface called `Todo` that defines the structure of a single todo item. It includes:

- `id`: a unique identifier of type `number`
- `text`: the task string
- `completed`: a boolean to check whether the task is done or not

```
export interface Todo {
  id: number;
  text: string;
  completed: boolean;
}
```

Interview Question:
Q: Why did you define a type for your todo item?
A: To enforce structure and type safety in my app. It ensures each todo has an id, text, and a completed status.

---

2. Setting Up State in App Component

What I did:

I created multiple `useState` hooks:

- `todos`: an array of `Todo` objects
- `text`: holds the value typed into the input field
- `editingId`: to track which todo is being edited
- `editText`: to hold the edited text while updating a todo

```
const [todos, setTodos] = useState<Todo[]>([]);

const [text, setText] = useState("");

const [editingId, setEditingId] = useState<number | null>(null);

const [editText, setEditText] = useState("");
```

Interview Question:
Q: What is the purpose of `editingId` and `editText` in this app?
A: `editingId` helps track which todo is currently being edited. `editText` stores the updated value while editing.

---

3. Adding a New Todo

What I did:

I wrote the `addTodo` function which checks if the input is not empty, creates a new `Todo` object, and adds it to the existing array.

```
const addTodo = () => {
  if (text.trim() === "") return;
  const newTodo: Todo = {
    id: Date.now(),
    text,
```

```
    completed: false,
  };
  setTodos([...todos, newTodo]);
  setText("");
};
```

Interview Question:
Q: How do you prevent empty todos from being added?
A: By using `text.trim()` and returning early if it's empty.

---

4. Toggling a Todo's Completion Status

What I did:

I used the `toggleTodo` function to flip the `completed` boolean for a specific todo by mapping over the `todos` array.

```
const toggleTodo = (id: number) => {
  setTodos(
    todos.map(todo =>
      todo.id === id ? { ...todo, completed: !todo.completed } : todo
    )
  );
};
```

Interview Question:
Q: How did you toggle the completed status in your todos?
A: I used `map()` to iterate through todos, found the matching id, and used object spreading to flip the `completed` value.

---

5. Deleting a Todo

What I did:

I wrote the `deleteTodo` function to remove a todo using `filter()`.

```
const deleteTodo = (id: number) => {
  setTodos(todos.filter(todo => todo.id !== id));
};
```

Interview Question:
Q: How did you delete a todo item?
A: I used `filter()` to return a new array that excludes the todo with the given id.

---

6. Updating a Todo

What I did:

I implemented editing by using `editingId` to track which todo is being edited and `editText` to hold the input. The `updateTodo` function updates the text of the todo.

```
const updateTodo = (id: number) => {
  setTodos(
    todos.map(todo =>
      todo.id === id ? { ...todo, text: editText } : todo
    )
  );
  setEditingId(null);
  setEditText("");
};
```

Interview Question:
Q: How do you update a specific todo?
A: I map through the todos, match by id, and update the text property with `editText`.

---

7. Passing Props to the `TodoItem` Component

What I did:

I created a separate `TodoItem` component and passed necessary props from `App.tsx`:

- `todo` object
- `onToggle`, `onDelete`, `onEdit`, `onUpdate` functions
- `editingId`, `editText`, and `setEditText` to handle edit mode

Interview Question:
Q: Why did you pass so many props to the child component?
A: Because the main logic is in the parent (`App.tsx`), but the UI actions happen in the child. Props connect the UI and logic.

---

8. Conditional Rendering in `TodoItem`

What I did:

I used a conditional to check if a todo is being edited, and showed different UI accordingly.

```
{editingId === todo.id ? (
  <>
    <input
      value={editText}
      onChange={(e) => setEditText(e.target.value)}
    />
    <button onClick={() => onUpdate(todo.id)}>Update</button>
```

```
  </>
) : (
  <>
    <span
      onClick={() => onToggle(todo.id)}
      style={{ textDecoration: todo.completed ? "line-through" : "none" }}
    >
      {todo.text}
    </span>
    <button onClick={onEdit}>Edit</button>
    <button onClick={() => onDelete(todo.id)}>Delete</button>
  </>
)}
```

Interview Question:
Q: How does conditional rendering help here?
A: It lets me show either the normal todo view or the edit view based on the current `editingId`.

---

9. React Fragment

What I did:

I used `<>...</>` to return multiple sibling elements inside JSX without extra divs.

Interview Question:
Q: What is a React Fragment and when do you use it?
A: A fragment (`<> </>`) allows returning multiple elements without wrapping them in an unnecessary parent div.

This document summarizes the structure, logic, and behavior of the todo app, step-by-step. All logic lives in the parent `App.tsx`, and the UI for individual todos is handled in the `TodoItem` child component using props and conditional rendering.