***Note: I have made quite a few modifications to the app.***

**Question 1: Describe the application development tools and methodologies you would use to build the todo management web application in ReactJS.**

Below is a summary of the development tools and methodologies I would use to build a Todo Management Web Application in ReactJS:

### 

### 1. Code Editor

Tool:Visual Studio Code (VS Code)

* Intellisense and JSX support for React
* Git integration, extensions like Prettier, ESLint, React snippets

### 

### 2. Project Bootstrapping

Tool: Vite

* Faster build/start times than CRA (Create React App)
* Modern ES module support

### 

### 3. JavaScript Framework

Tool: ReactJS

* Component-based structure
* Reusable UI pieces like TodoForm, TodoItem, etc.
* Hooks like useState, useEffect for managing state and side effects

4. State Management

Methodology: Local state via useState and prop drilling

* todos, filter, and form states are managed in App.jsx
* For larger apps, you might use Context API, Redux, or Zustand

5. Component Design Methodology

Methodology: Atomic or modular design

* Split into clear components: TodoHeader, TodoForm, TodoList, TodoItem, TodoFilter, TodoStats
* Promotes reusability, testing, and clarity

6. Styling

Tool: CSS modules / plain CSS

* We used plain CSS in src/styles/\*.css
* Mobile-first design via media queries
* Separated styling per component (e.g. TodoForm.css, TodoItem.css)

7. Input Security

Methodology: Input sanitization

* Prevented XSS by sanitizing inputs in TodoItem.jsx

8. Testing & Debugging

Tool:

* Console.log for debugging
* Jest + React Testing Library (optional) for unit/component testing

10. API Simulation

Methodology: Mock functions

* We used mock getTodos, addTodo, updateTodo, etc.
* For real apps, replace with Axios or Fetch connected to an API/backend

11. Deployment (for real app)

Tool: Vercel or Netlify

* Automatically builds and deploys from your GitHub repo
* Free HTTPS live URL

**Question 2: Demonstrate with the appropriate endpoints how would you ensure that the application meets the agreed specifications? Focusing on code bundling, security of your API.**

To ensure the application meets agreed specifications:

* Code Bundling: I use Vite to bundle and optimize the application with npm run build, which produces minified files for fast loading and efficient deployment.
* API Security: While the current app uses mock functions and does not yet connect to a real backend, I would secure endpointsfuture like GET /api/todos, POST /api/todos, and DELETE /api/todos/:id by implementing:  
  + Rate limiting (e.g. 100 requests per 15 min per IP using express-rate-limit)
  + CORS restrictions using the cors package to allow only trusted origins
  + Input validation and sanitization to prevent injection attacks and XSS

These practices ensure both performance and protection of application data that is production ready.

**Question 3: Demonstrate how you will add a new feature called “ToDoHeader” and modify TodoList.css to your ToDo ReactJS App. Also show the directory structure that adheres to the application development standard.**

Based on my setup with components like TodoHeader, TodoList, and separate CSS files, below is my directory structure that adheres to the application development standard where there’s component-based separation, dedicated styling folder to keep jsx files clean, central entry point in main.jsx and scalable structure where I can later add contexts/, hooks/, or pages/ if the app grows.

HuiyingTan-todo-app/

├── public/

│ └── index.html

├── src/

│ ├── components/

│ │ ├── TodoHeader.jsx

│ │ ├── TodoForm.jsx

│ │ ├── TodoList.jsx

│ │ ├── TodoItem.jsx

│ │ ├── TodoFilter.jsx

│ │ └── TodoStats.jsx

│ ├── styles/

│ │ ├── TodoHeader.css

│ │ ├── TodoForm.css

│ │ ├── TodoList.css

│ │ ├── TodoItem.css

│ │ ├── TodoFilter.css

│ │ └── TodoStats.css

│ ├── App.jsx

│ ├── App.css

│ ├── main.jsx

│ └── api/ ← optional: for mock or real API logic

│ └── todoService.js

├── index.html

├── package.json

└── vite.config.js

**Question 4: Explain how you would adhere to organizational standards in application development and documentation while building the todo management web application. How would you document the internal design of the application for future maintenance and enhancement?**

Adhering to Organizational Standards in Application Development

To ensure the todo management web application follows organizational development standards, I apply the following best practices:

#### 

#### 1. Consistent Code Structure and Naming

* Use a clear and modular directory structure (src/components, src/styles)
* Follow consistent naming conventions for files, variables, and functions (e.g. TodoItem.jsx, addTodo())

#### 

#### 2. Reusable and Maintainable Code

* Component-based architecture in React (e.g., TodoForm, TodoList)
* Avoid hardcoding by using props and state
* Sanitize user inputs to prevent XSS

#### 

#### 3. Coding Standards

* Use Prettier and ESLint to enforce formatting and avoid common JS/React errors
* Write clear, commented code for complex logic

#### 

#### 4. Testing and Debugging

* Use console.log and React DevTools during development
* Optionally, add Jest + React Testing Library for unit and integration tests

Documenting Internal Design for Future Maintenance

To support future developers or team members:

#### 

#### 1. README.md

* Describe the app purpose, tech stack, setup instructions, and usage
* Include screenshots or links if deployed

#### 

#### 2. Component-Level Comments

* Add inline comments inside JSX and JS logic for tricky or important sections

#### 

#### 3. Architecture Overview

* Create a simple design doc (markdown or PDF) that includes:
  + Component hierarchy (e.g., App → TodoList → TodoItem)
  + State flow (e.g., useState, props)
  + Mock API or backend structure
  + Folder and file structure

#### 

#### 4. Future Enhancements

* Maintain a README.md or GitHub Issues list to track feature ideas (e.g., authentication, backend API integration)

**Question 5: Discuss the tools and techniques you would utilize for coding and programming the todo management web application.**

To code and programme the todo management web application effectively, I used a combination of development tools and coding techniques that align with good software practices.

Visual Studio Code (VS Code) is the main code editor, offering features like syntax highlighting, auto-formatting with Prettier, and linting with ESLint to maintain consistent code quality. I used Vite as the build tool for fast development and bundling, along with NPM to manage React and related dependencies.

In terms of programming techniques, I followed a component-based approach using React, breaking the application into reusable components such as TodoForm, TodoList, and TodoItem. I used React hooks like useState to manage state and useEffect for handling side effects. Props were passed from the parent App component to child components to maintain a clear data flow.

I applied input sanitization in TodoItem to remove potentially dangerous characters and prevent cross-site scripting (XSS) vulnerabilities. Styling was handled with mobile-first CSS and separated into individual .css files per component to ensure clean, modular design.

For debugging, I used console logging and the React Developer Tools browser extension to inspect component props and state during runtime. These tools and techniques ensured that the application was built in a structured, maintainable, and secure way.

**Question 6: Discuss and show code snippets how would you employ basic debugging tools and techniques to identify and resolve application errors or problems?**

To ensure the todo management app runs smoothly, I used basic debugging tools and techniques to identify and fix errors.

### 

### 1. Console Logging

The most straightforward technique is using console.log() to inspect state values, function execution, and API responses.

Example: Logging state updates in TodoForm

const handleSubmit = (e) => {

e.preventDefault();

console.log("Submitting new todo:", title, description, dueDate); // Debug log

if (!title.trim()) return;

addTodo({

title,

description,

dueDate,

completed: false,

});

// Clear form fields

setTitle('');

setDescription('');

setDueDate('');

};

This helps verify if the correct values are being captured and passed before submission.

### 

### 2. Error Boundaries (for runtime errors)

In production apps, I can add a basic error boundary to catch JavaScript errors in React components and show fallback UI instead of crashing the app.

// ErrorBoundary.jsx

import React from 'react';

export class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

static getDerivedStateFromError() {

return { hasError: true };

}

render() {

if (this.state.hasError) {

return <h3>Something went wrong.</h3>;

}

return this.props.children;

}

}

Wrap App in main.jsx:

import { ErrorBoundary } from './components/ErrorBoundary';

createRoot(document.getElementById("root")).render(

<StrictMode>

<ErrorBoundary>

<App />

</ErrorBoundary>

</StrictMode>

);

### 

### 3. Using DevTools to Trace DOM and Styling Issues

I use the browser’s Elements and Styles tab to:

* Check if elements are rendering correctly
* Debug layout problems like misalignment or overflow
* Verify applied CSS classes and styles

### 

### 4. Reading and Resolving Console Errors

Errors like this, which actually happened, help pinpoint issues:

Uncaught SyntaxError: The requested module '/src/components/TodoItem.jsx' does not provide an export named 'default'

Fix: If used:

export function TodoItem() { ... }

then must import it using:

import { TodoItem } from './components/TodoItem.jsx';

Instead of:

import TodoItem from './components/TodoItem.jsx';

**Question 7: Explain and demonstrate concisely the software testing process you would follow to ensure the reliability and bug-free nature of the todo management web application. Show unit testing codes for defects/errors/potential security vulnerabilities checking through software tests.**

### 

### Software Testing Process for the Todo Management Web Application

To ensure the todo app is reliable and free from bugs, I would use Jest with React Testing Library to perform unit testing and component testing. These tools simulate user interactions and verify expected behavior without depending on the browser environment. Refer to a simple testing done in the app under src/tests.

### 

### Testing Process Overview

1. Set up testing environment using jest and @testing-library/react
2. Write unit tests for:
   * Component rendering (e.g. TodoItem, TodoForm)
   * User input and form submission
   * Filtering, editing, and deleting todos
3. Test edge cases and invalid input handling
4. Run tests using npm test
5. Check coverage to identify untested logic

Simple Test Examples

#### 

#### Test if TodoForm adds a new todo when submitted:

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

import { TodoForm } from '../components/TodoForm';

test('submits form and calls addTodo', () => {

const mockAddTodo = jest.fn();

render(<TodoForm addTodo={mockAddTodo} />);

fireEvent.change(screen.getByPlaceholderText(/title/i), {

target: { value: 'Buy milk' },

});

fireEvent.click(screen.getByText(/add/i));

expect(mockAddTodo).toHaveBeenCalledWith(

expect.objectContaining({ title: 'Buy milk' })

);

});

Test if input is sanitized (security check):

import { sanitize } from '../components/TodoItem'; // if exported

test('sanitizes dangerous characters to prevent XSS', () => {

const input = '<script>alert("xss")</script>';

const result = sanitize(input);

expect(result).not.toMatch(/<|>|script|alert/);

});

**Question 8: Describe commonly encountered errors that may occur during the development of the todo management web application. How would you address and make simple revisions or modifications to the application to resolve these errors?**

During the development of the todo management web application, several common errors may arise.

1. One frequent issue is the TypeError, such as "undefined is not a function", which usually occurs when a function is expected but hasn't been properly defined or passed as a prop. This can be resolved by checking that all necessary props are correctly passed down to components.
2. SyntaxErrors, like "Unexpected token '<'", often result from incorrect JSX formatting—typically when JSX is returned improperly or without parentheses. These can be fixed by ensuring that JSX syntax is valid and properly structured.
3. Another common issue is missing or incorrect exports, where an error such as "does not provide an export named 'default'" appears. This is addressed by matching the import statement to the correct export type—using named imports for functions exported without default.
4. Rendering errors, like components failing silently or displaying a blank screen, can be debugged using console logs and React Developer Tools to trace rendering logic and state changes.
5. Security vulnerabilities such as XSS (Cross-Site Scripting) may arise if user input is rendered without sanitization. This is resolved by applying a sanitization function that escapes dangerous characters before storing or rendering user input.

### 

### Making Revisions to Resolve Errors

* Type/syntax errors: Update import/export syntax, ensure variables are defined, and JSX syntax is valid.
* XSS issues: Apply input sanitization as already implemented in TodoItem.jsx.
* Styling/layout issues: Use browser dev tools to trace CSS class application.
* Component bugs: Break components into smaller parts and write tests to isolate issues.

By combining automated unit tests, security input checks, and component testing with React Testing Library and Jest, I can ensure the todo application is robust, secure, and bug-free. This process also improves maintainability and future-proofing of the application.