



Explore | Expand | Enrich

# Web Technologies Test Projects

Ethnus  
Mar 2025

<b>Web Technologies Test Projects.....</b>	<b>3</b>
College Level (2.5 hours).....	3
State Level (3.5 hours).....	6
National Level (5 hours - SPA Frontend).....	10



Explore | Expand | Enrich

# Web Technologies Test Projects

## College Level (2.5 hours)

- **Objective:** Build a static, responsive, multi-section "Event Landing Page" using semantic HTML, CSS for layout and styling (including Flexbox or Grid), and basic JavaScript for simple interactivity.
- **Scenario:** "Innovate Startups" is hosting a launch event and needs a simple, informative landing page. You need to create this page based on provided content and a basic wireframe/description.
- **Prerequisites:**
  - Text Editor (e.g., VS Code).
  - Modern Web Browser (e.g., Chrome, Firefox) with Developer Tools.
  - Provided assets:
    - Text content for different sections (e.g., `event_details.txt`).
    - A placeholder logo image (`logo.png`).
    - A basic wireframe image or description (`wireframe.jpg` or `layout_guide.txt`).
- **Detailed Steps/Modules:**
  - **Module 1: HTML Structure (Approx. 45 mins)**
    - 1.1. Create the main HTML file (`index.html`).
    - 1.2. Set up the basic HTML document structure (`<!DOCTYPE html>`, `<html>`, `<head>`, `<body>`). Include appropriate `<meta>` tags for charset and viewport. Link a CSS file (e.g., `style.css`).
    - 1.3. Using the provided text content and wireframe as a guide, structure the `<body>` using semantic HTML5 tags (e.g., `<header>`, `<nav>`, `<main>`, `<section>`, `<article>`, `<footer>`, `<button>`, `<img>`, `<h1>`-`<h6>`, `<p>`, `<ul>`, `<li>`).
    - 1.4. Create distinct sections for:
      - Header (with logo and event title).

- Navigation (simple links like Home, About, Schedule, Venue - non-functional initially).
- Hero/Introduction section.
- About the Event section.
- Event Schedule section (use a list or simple table structure).
- Venue/Location section.
- Footer (with copyright info).
- 1.5. Ensure content is correctly placed within the semantic structure. Use appropriate heading levels. Include the placeholder logo image with an `alt` attribute.
- **Module 2: CSS Styling & Layout (Approx. 75 mins)**
  - 2.1. Create the linked CSS file (`style.css`).
  - 2.2. Apply basic resets or normalization if desired (optional).
  - 2.3. Style the basic elements (body font, colors, background).
  - 2.4. Implement the main page layout using **CSS Flexbox** or **CSS Grid** based on the provided wireframe/guide. Ensure sections are visually distinct.
  - 2.5. Style the header, navigation, sections, and footer according to the visual guide (or apply sensible styling if no strict guide is provided). Focus on padding, margins, borders, colors, and typography.
  - 2.6. Style interactive elements like navigation links (e.g., hover effects).
  - 2.7. **Responsiveness:** Add at least one CSS media query (e.g., for screen widths below 768px) to adjust the layout for smaller screens (e.g., stack columns, adjust font sizes, simplify navigation). Test using browser developer tools.
- **Module 3: Basic JavaScript Interactivity (Approx. 30 mins)**
  - 3.1. Create a JavaScript file (e.g., `script.js`) and link it at the bottom of the `<body>` in `index.html`.
  - 3.2. Add an event listener to a button (e.g., a "Toggle Schedule" button you add in the HTML).

- 3.3. When the button is clicked, toggle the visibility of the "Event Schedule" section (e.g., by adding/removing a CSS class like `.hidden` which has `display: none;`).
- 3.4. (Optional) Add smooth scrolling for the navigation links (if time permits). Find navigation links, add event listeners, prevent default jump, get target section ID, and use `element.scrollIntoView({ behavior: 'smooth' });`.

- **Deliverables:**

- Submit the `index.html`, `style.css`, `script.js` files, and any used assets (`logo.png`).
- Verification involves opening `index.html` in a browser.
- **Criteria:**
  - Correct semantic HTML structure.
  - Layout matches wireframe/guide and uses Flexbox/Grid.
  - Page is reasonably styled and visually organized.
  - Page layout adapts correctly at smaller screen sizes (tested via dev tools).
  - The JavaScript toggle button correctly shows/hides the schedule section.

Explore | Expand | Enrich

## State Level (3.5 hours)

- **Objective:** Build an interactive single-page application (SPA) feel for a "Movie Finder". Fetch movie data asynchronously from a provided mock API endpoint using JavaScript, display it dynamically, implement client-side filtering/search, and use Local Storage for a simple "favorites" feature.
- **Scenario:** "Innovate Startups" wants a frontend prototype for their movie database service. You need to build the interface that allows users to browse and filter movies fetched from their API.
- **Prerequisites:**
  - Text Editor (VS Code recommended).
  - Modern Web Browser with Developer Tools.
  - **Provided Mock API Endpoint URL:** A URL (e.g., `https://api.example.com/movies` or a locally hosted JSON file URL if network access is restricted) that returns a JSON array of movie objects when a GET request is made. Each movie object might look like: `{ "id": 1, "title": "...", "genre": "...", "description": "...", "posterUrl": "..." }`.
  - Basic wireframe or layout description.
- **Detailed Steps/Modules:**
  - **Module 1: HTML Structure & CSS Layout (Approx. 45 mins)**
    - 1.1. Create `index.html` and link `style.css` and `script.js`.
    - 1.2. Set up the basic HTML structure. Include:
      - A header area.
      - A filter/search area containing:
        - A text input field for searching by title (`<input type="text" id="searchInput">`).
        - A dropdown (`<select id="genreFilter">`) for filtering by genre (populate with placeholder or dynamically later).
      - A main content area (`<main id="movieListContainer">`) where movie cards will be displayed.

- 1.3. Create basic CSS rules in `style.css` for layout (e.g., using Flexbox/Grid for the main areas and the movie list), padding, margins, and basic styling for the input/select elements. Style a basic "movie card" structure (it will be empty initially).
- 1.4. Ensure the layout is responsive using media queries.
- **Module 2: Fetching & Displaying API Data (Approx. 60 mins)**
  - 2.1. In `script.js`, write an asynchronous function (e.g., `WorkspaceMovies`) using the `Workspace` API to make a GET request to the **provided mock API endpoint URL**.
  - 2.2. Handle the response: parse the JSON data. Include basic error handling (e.g., log errors to the console if the fetch fails).
  - 2.3. Write a function (e.g., `displayMovies(moviesArray)`) that takes an array of movie objects.
  - 2.4. Inside `displayMovies`, clear the existing content of the `movieListContainer`.
  - 2.5. Loop through the `moviesArray`. For each movie object, dynamically create HTML elements for a "movie card" (e.g., a `<div>` containing an `<img>` for the poster, `<h3>` for the title, `<p>` for genre/description, and a "Favorite" button).
  - 2.6. Populate the created elements with data from the movie object (title, posterUrl, etc.). Remember to set `src` and `alt` for images.
  - 2.7. Append each created movie card element to the `movieListContainer`.
  - 2.8. Call `WorkspaceMovies()` when the page loads to initially populate the list.
  - 2.9. (Optional) Dynamically populate the genre filter dropdown (`<select>`) with unique genres found in the fetched movie data.
- **Module 3: Client-Side Search & Filtering (Approx. 60 mins)**
  - 3.1. Store the initially fetched movie data in a global variable (e.g., `allMovies`).

- 3.2. Add an event listener to the search input (`#searchInput`) that triggers on `input` or `keyup`.
- 3.3. Inside the event listener, get the current search term (lowercase). Filter the `allMovies` array based on whether the movie title (lowercase) includes the search term.
- 3.4. Add an event listener to the genre filter dropdown (`#genreFilter`) that triggers on `change`.
- 3.5. Inside the event listener, get the selected genre. Filter the `allMovies` array based on whether the movie genre matches the selected genre (handle "All Genres" option if added).
- 3.6. **Combine Filters:** Modify the event listeners so that both search term and selected genre are considered when filtering the `allMovies` array. Create a combined filtering function if needed.
- 3.7. After filtering, call the `displayMovies()` function, passing the *filtered* array of movies to update the UI. Ensure filtering happens purely on the client side without re-fetching from the API.
- **Module 4: Local Storage for Favorites (Approx. 30 mins)**
  - 4.1. When creating movie cards in `displayMovies`, add an event listener to each "Favorite" button. Store the movie's unique `id` in a data attribute (e.g., `data-movie-id`) on the button.
  - 4.2. In the button's event listener:
    - Get the `movieId` from the data attribute.
    - Retrieve the current list of favorite IDs from `localStorage` (e.g., parse a JSON array stored under a key like `favoriteMovies`). If nothing exists, start with an empty array.
    - Check if the `movieId` is already in the favorites list.
    - If not present, add it to the array. If present, optionally remove it (toggle functionality).
    - Save the updated favorites array back to `localStorage` (stringify the array first).



- (Optional) Visually update the button/card (e.g., change button text to "Unfavorite", add a CSS class) to indicate favorite status. Check favorite status when initially displaying cards.

- **Deliverables:**

- Submit the `index.html`, `style.css`, `script.js` files.
- Verification involves opening `index.html` in a browser.
- **Criteria:**
  - Movie data is fetched from the provided API and displayed in cards.
  - Search input correctly filters the displayed movies by title in real-time.
  - Genre dropdown correctly filters the displayed movies by genre.
  - Search and genre filters work together correctly.
  - Clicking the "Favorite" button stores the movie ID in `localStorage` (verifiable via browser dev tools -> Application -> Local Storage).
  - Page layout is responsive.

Explore | Expand | Enrich

## National Level (5 hours - SPA Frontend)

- **Objective:** Build a more complex Single Page Application (SPA) frontend for a "Project Dashboard". Consume data from multiple provided mock API endpoints, implement client-side routing (hash-based), manage application state effectively (loading, errors), implement advanced form validation, and optionally visualize data using a simple chart library. Emphasize clean code structure and accessibility.
- **Scenario:** "Innovate Startups" needs a dashboard interface for their project management tool. You will build the frontend that interacts with their backend APIs (mocked) to display projects, tasks, and allow adding new tasks.
- **Prerequisites:**
  - Text Editor (VS Code recommended).
  - Modern Web Browser with Developer Tools.
  - **Provided Mock API Endpoint URLs:**
    - `GET /api/projects`: Returns array of projects { `id`, `name`, `description` }.
    - `GET /api/projects/{projectId}/tasks`: Returns array of tasks for a specific project { `id`, `title`, `status` ('Pending', 'In Progress', 'Completed'), `projectId` }.
    - `POST /api/tasks`: Accepts a new task object { `title`, `status`: 'Pending', `projectId` }, returns the created task object with a new `id`. (Mock will just echo back with a dummy ID).
  - (Optional) CDN link for a simple charting library like Chart.js.
  - Basic UI mockups or description of required views/components.
- **Detailed Steps/Modules:**
  - **Module 1: Project Setup & Client-Side Routing (Approx. 60 mins)**
    - 1.1. Create `index.html` (SPA shell), `style.css`, `script.js`.
    - 1.2. Design the HTML shell (`index.html`) with a main content area (`<div id="app">`) where different "views" will be rendered dynamically by JavaScript. Include placeholders for header/navigation if needed.

- 1.3. In `script.js`, implement a simple hash-based router:
  - Listen for `hashchange` event on the `window`.
  - Listen for the initial `DOMContentLoaded` event.
  - Create a function (e.g., `handleRouteChange`) that reads the `location.hash` (e.g., `#`, `#projects`, `#project/123/tasks`, `#addTask/123`).
  - Based on the hash, determine which "view" function to call (e.g., `renderProjectList()`, `renderTaskList(projectId)`, `renderAddTaskForm(projectId)`).
  - Implement basic functions (stubs for now) for rendering each view, which will clear the `#app` div and add the appropriate content.
  - Ensure navigation links (if any) use hash paths (e.g., `<a href="#projects">Projects</a>`).
- **Module 2: Project List View & State Management (Approx. 75 mins)**
  - 2.1. Implement the `renderProjectList()` function.
  - 2.2. Inside it, display a loading indicator in the `#app` div.
  - 2.3. Use `Workspace` to call the provided `GET /api/projects` endpoint.
  - 2.4. Handle application state:
    - On success: Store the fetched projects array. Remove loading indicator. Dynamically render the list of projects (e.g., as clickable links/cards showing name and description). Each project link should point to its task view hash (e.g., `#project/{id}/tasks`).
    - On failure: Remove loading indicator. Display a user-friendly error message in the `#app` div.
  - 2.5. Structure your JavaScript code logically (e.g., separate functions for API calls, rendering, state management variables).
  - 2.6. Style the project list and loading/error states using CSS.

- **Module 3: Task List View (Approx. 75 mins)**
  - 3.1. Implement the `renderTaskList(projectId)` function. It receives the `projectId` from the router.
  - 3.2. Display loading state.
  - 3.3. Fetch tasks using the provided `GET /api/projects/{projectId}/tasks` endpoint (substitute the actual `projectId`).
  - 3.4. Handle loading, success, and error states similarly to the project list view.
  - 3.5. On success, display the list of tasks for the specific project. Show task `title` and `status`. Maybe use different styling based on status.
  - 3.6. Include a button/link within this view that navigates to the "Add Task" view for the current project (e.g., linking to `#addTask/{projectId}`).
  - 3.7. Add a "Back to Projects" link (`<a href="#projects">`).
  - 3.8. Style the task list appropriately.
- **Module 4: Add Task Form & Validation (Approx. 75 mins)**
  - 4.1. Implement the `renderAddTaskForm(projectId)` function.
  - 4.2. Render an HTML form within the `#app` div. The form should have:
    - An input field for the task title (`<input type="text" id="taskTitle" required>`).
    - A hidden input or way to associate the `projectId`.
    - A submit button.
    - A "Cancel" link/button navigating back to the task list (`#project/{projectId}/tasks`).
  - 4.3. Add an event listener to the form's `submit` event. Prevent the default form submission.
  - 4.4. **Client-Side Validation:** Before submitting, validate the task title input (e.g., ensure it's not empty, maybe minimum length). Display clear validation error messages near the input field if invalid, and prevent submission.

- 4.5. If validation passes:
  - Construct the new task object: `{ title: taskTitleValue, status: 'Pending', projectId: projectId }`.
  - Disable the submit button and show a submitting/loading indicator.
  - Use `Workspace` to make a `POST` request to the provided `/api/tasks` endpoint, sending the new task object in the request body (as JSON). Set appropriate headers (`Content-Type: application/json`).
  - Handle the response:
    - On success (e.g., 201 Created): Navigate the user back to the task list view for that project (`location.hash = '#project/{projectId}/tasks'`). The task list should re-fetch to show the new task.
    - On failure: Display an error message to the user. Re-enable the submit button.

○ **Module 5: (Optional) Data Visualization & Accessibility (Approx. 60 mins)**

- 5.1. **Charting:** If Chart.js CDN link is provided/allowed:
  - In the Task List view (`renderTaskList`), after fetching tasks, calculate the count of tasks per status ('Pending', 'In Progress', 'Completed').
  - Add a `<canvas>` element to the HTML structure for this view.
  - Use Chart.js to render a simple pie or bar chart showing the task status distribution for the current project.
- 5.2. **Accessibility Review:** Review your HTML structure. Ensure:
  - Semantic elements are used correctly.
  - Images have meaningful `alt` attributes.
  - Form inputs have associated `<label>` elements.
  - Interactive elements (buttons, links) are keyboard accessible.

- (Advanced) Consider adding basic ARIA attributes if needed for dynamically updated regions (e.g., `aria-live` for error messages).

- **Deliverables:**

- Submit the `index.html`, `style.css`, `script.js` files.
- Verification involves opening `index.html` in a browser and testing the SPA functionality.
- **Criteria:**
  - Client-side routing correctly displays different views based on URL hash.
  - Project list is fetched and displayed correctly, with links working.
  - Task list for a selected project is fetched and displayed correctly.
  - Loading and error states are handled gracefully for API calls.
  - Add Task form includes client-side validation.
  - Successfully adding a task makes a POST request and redirects back to the task list (new task should appear on refresh/refetch).
  - Code is well-structured and readable.
  - (If attempted) Chart displays task status distribution correctly.
  - Basic accessibility principles are followed in the HTML structure.
  - Layout is responsive.

– End of Document –

Explore | Expand | Enrich