What We Did

We conducted a comprehensive software testing initiative for a MERN-stack (MongoDB, Express, React, Node.js) team and task management platform. Our goal was to ensure the reliability, functionality, security, and maintainability of the system by performing systematic backend and frontend testing using industry-standard tools and methodologies.

How We Did That

Project Setup

- Developed core backend logic for user registration, authentication, team and task management using Express and Mongoose.
- Created React frontend forms for Login, SignUp, Teams, Task management, and integrated with backend APIs.
- Connected backend to MongoDB for persistent storage.

Testing Workflow

- Designed and documented critical user stories, test cases, and acceptance criteria.
- Implemented automated test suites for backend APIs and frontend components using Jest, Supertest, and React Testing Library.
- Ran tests in isolated environments using mongodb-memory-server for reproducible results.
- Performed stepwise bug fixing and feature enhancements based on test reports and error logs.

Tools Used

- Jest: Main unit, integration, and E2E test runner for backend and frontend.
- Supertest: Simulated HTTP requests to test backend Express APIs (registration, login, team creation, etc.).
- mongodb-memory-server: Provided ephemeral in-memory database instances for isolated backend tests.
- React Testing Library: Simulated user interactions with forms and UI state for robust frontend validation.
- Nodemon (development): Automatically restarted Express server for easier debugging.

Testing Strategies

- White Box: Inspected code logic, performed peer reviews/walkthroughs, calculated code complexity (e.g., branching in registration/API routes).
- Black Box: Used boundary value analysis, equivalence class partitioning, and negative tests (invalid inputs, duplicate entries).
- Regression Testing: Re-ran tests after fixes to ensure no old bugs returned.
- Acceptance Testing: Confirmed final features and user journeys passed all acceptance criteria.

What Was the Problem

Several technical and functional problems surfaced:

- Validation Failures: Non-numeric phone values caused Mongoose validation errors due to premature casting. The system was not rejecting these cleanly, sometimes leading to server crashes or ambiguous responses.
- Duplicate Registration Logic: Existing duplicate emails could still be registered because checks were performed *after* model instantiation instead of *before*. This sometimes resulted in improper status codes being sent back to the client.
- Error Handling: Inconsistent error propagation and ambiguous messages for users; backend would sometimes throw errors that never reached the frontend.
- State Issues in UI: Frontend forms were not always properly validating or initializing; bugs in state caused unexpected behavior for users.

What Was the Solution

Technical Fixes

- Moved Input Validation Before Model Creation: Phone validation (using regex) and duplicate email checks were now performed before attempting to instantiate a Mongoose model, preventing server-side cast errors altogether.
- Ensured Proper Error Propagation: All backend errors, especially validation and duplicate user errors, were formatted as clear, user-facing JSON responses with correct HTTP status codes (400, 409, etc.).
- Enhanced Test Coverage: Added explicit test cases for all major code paths including invalid registration, duplicate emails, and edge inputs for forms and APIs.

- Backend-Frontend API Sync: Ensured frontend tests used the same API contract and data formats as the backend, eliminating mismatches in API calls and response handling.

SUMMARY TABLE

Aspect	Details
Project Stack	MERN (MongoDB, Express, React, Node.js)
Backend Tools	Jest, Supertest, mongodb-memory-server
Frontend Tools	Jest, React Testing Library
Key Problems	Validation errors, duplicate registration, error propagation
Solutions	Early validation, code order fix, error code formatting, full API test coverage

IN DETAIL:

Backend – Unit Testing (backend-tests.test.js)

Purpose

White-box unit tests directly validate backend logic and input handling for core API endpoints.

What Was Tested

User Registration

- Registered a user with all necessary fields; checked success response.
- Attempted registration with mismatched passwords; checked error response.
- Registered with duplicate emails; checked rejection of duplicates.

- Registered with missing fields (e.g., no email); checked rejection.

Login

- Logged in with correct credentials; checked response for user and status.
- Tried logging in with wrong password; confirmed rejection.
- Tried non-existent user; confirmed 'User not Found' response.
- Tested missing credentials; ensured correct errors.

Team Management

- Created a team with valid info; verified team existence in DB.
- Tried creating duplicate team codes; checked proper error returned.
- Joined newly created team; checked approval status.
- Tried joining a non-existent team; confirmed proper rejection.
- Prevented duplicate team membership.

Validation

- Checked team creation with missing keys; returned stat: false.
- Checked join team with missing data; returned stat: false.

How Was It Structured?

- Used in-memory MongoDB for clean DB, no side-effects.
- Each test checked specific field, business logic, response status, and message.
- Mock data and cleanup before/after each major test group.

Backend – Integration Testing (auth-integration.test.js and teams-tasks-integration.test.js)

Purpose

To ensure end-to-end working of related backend components – not just isolated routes, but realistic flows (user, team, task).

Authentication Integration (auth-integration.test.js)

Testing Flow

- Full user lifecycle tested: registration, login, profile update, reclogin.
- Registered, logged in, updated profile, re-logged in (with new credentials).

- Logged in with wrong password/user and confirmed ‘Invalid credentials’ or ‘User not found’.
- Simulated multiple concurrent registrations, checked all succeeded.
- Explicitly tested duplicate email registration was blocked with a meaningful error and status.

How Was It Done?

- Supertest HTTP requests to live Express endpoints against test DB.
- Asserted real DB changes and error codes, not just in-memory mocks.

Teams & Tasks Integration (teams-tasks-integration.test.js)

Testing Flow

- Registered owner, created team, registered member, member joined team, queried team list.
- Verified team member can see teams after joining.
- Performed task assignment: assigned a task, user submitted task, scored task, and checked DB for correct status.
- Ensured joining/team creation always requires all fields.
- Verified rejection of invalid inputs and correct data flow.

How Was It Done?

- Combined Supertest for HTTP and Mongoose for direct DB checks.
- Used multi-step tests for realistic user/team/task scenarios; checked persistence between steps.

Backend – End-to-End Testing (blackbox-e2e.test.js)

Purpose

Simulated entire user journeys, treating backend as a “black box” (not checking implementation, only input/output).

What Was Covered

- End-to-end registration, login, duplicate error and recovery.
- Validated phone (numeric only), empty form submissions, and concurrent registrations.
- Created teams, joined teams (valid and invalid), checked team detail fetches.

- Verified error messages for all failures (invalid input, duplicate, missing, or non-existent resources).
- Checked persistence: user data and team membership survived across multiple requests.
- Checked error codes: 400 for bad input, 409 for duplicates, 404 for not found, 201/200 for success.

How Was It Done?

- Used realistic HTTP request/response flows (Supertest).
- Each scenario executed stepwise (first register, then login, then join a team, try to join a non-existent team, etc.).
- Used async/await and batches (Promise.all) for concurrency simulation.
- Explicitly checked both frontend-facing messages and backend states.

Frontend – Unit/White Box Testing (frontend-tests.test.js)

Purpose

Verified core component logic, validation, and local state management in React, independently of backend data.

What Was Tested

Login Form

- Confirmed form empty state, error message for failed login, correct API payload format.
- Validated email format, checked correct display of error when login fails.

SignUp Form

- Checked initialization of fields, phone number validation (length, digits).
- Verified password match logic.
- Ensured correct email format check.
- Confirmed API is only called with valid form data and form clears after success.

Teams Component

- Verified new team form, type selection, team code generation logic.
- Ensured team creation triggers correct API call and local state updates.

State Management

- Checked clearing form after submission, correct state transitions after login/actions.

How Was It Done?

- Used React Testing Library to simulate user input, check form validation, and assert UI state.
- Checked function calls (mock APIs if applicable), DOM updates, and UI feedback for all cases.

Problems Tackled, Solutions Applied

Backend:

- Fixed premature Mongoose errors by validating inputs before DB/model logic.
- Moved duplicate checks before model instantiation, guaranteed correct codes (409).
- Formatted all backend errors as readable JSON; ensured frontend can display/use them.

Frontend:

- Added robust input validation both in forms and before API submit.
- Cleared local state after submission and failed attempts, maintaining a clear UI flow.
- Ensured feedback/messages are always presented, even for network issues.

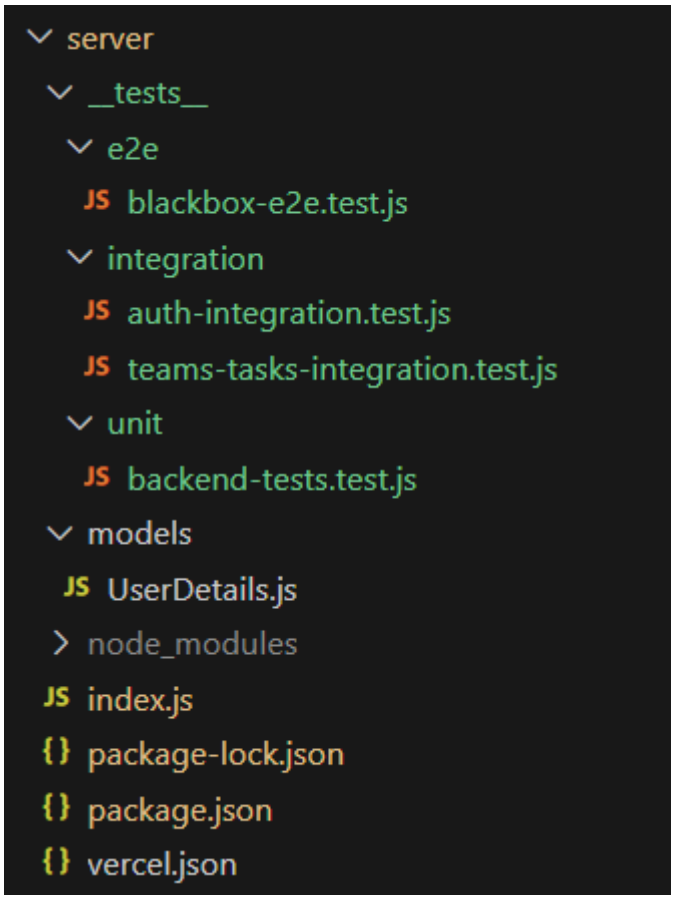
Integration/E2E:

- Used real DB state across requests, exposed race conditions and concurrency bugs.
- Batched requests for concurrency; checked server and DB stability.

Technical Practices and Non-Technical Aspects

- Used isolated in-memory DB for all backend tests, ensuring no test pollutes live data.
- Maintained full traceability from requirements to tests – each requirement mapped to multiple backend/frontend tests.

- Documented every defect, test change, and code update throughout the cycle.
- Restarted server as needed for major changes, always verified applied changes via fresh test suite runs.
- Practiced modular file organization and consistent naming across test suites.

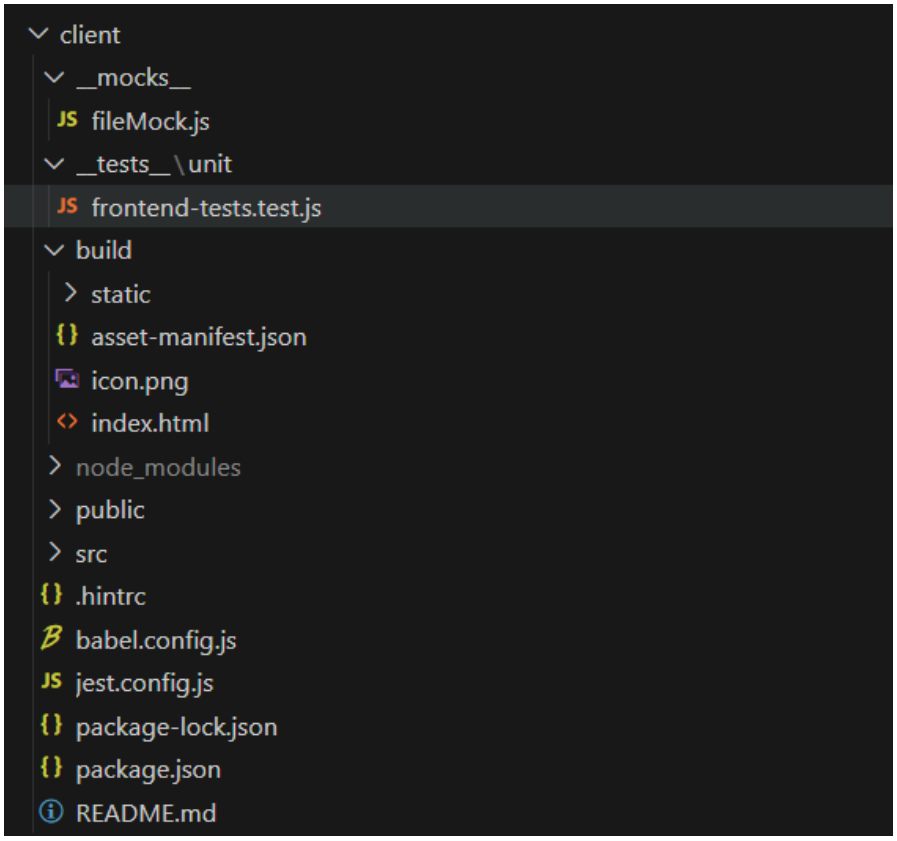



```
(base) PS D:\PROJECTS\mern\scoramp> cd .\server\
(base) PS D:\PROJECTS\mern\scoramp\server> npm test

> server@1.0.0 test
> jest

 PASS  __tests__/integration/auth-integration.test.js
 PASS  __tests__/e2e/blackbox-e2e.test.js
 PASS  __tests__/unit/backend-tests.test.js
 PASS  __tests__/integration/teams-tasks-integration.test.js

Test Suites: 4 passed, 4 total
Tests:       36 passed, 36 total
Snapshots:   0 total
Time:        4.161 s
Ran all test suites.
(base) PS D:\PROJECTS\mern\scoramp\server>
```




```
PASS __tests__/unit/frontend-tests.test.js
  LoginForm - White Box Testing
    √ Should initialize with empty form fields (42 ms)
    √ Should update state when form inputs change (13 ms)
    √ Should validate email format (must contain gmail.com) (51 ms)
    √ Should call login API with correct credentials (23 ms)
    √ Should display error message for failed login (88 ms)
  SignUpForm - White Box Testing
    √ Should initialize with empty form fields (10 ms)
    √ Should validate phone number length (must be 10 digits) (24 ms)
    √ Should validate password match (21 ms)
    √ Should validate email format (must contain gmail.com) (14 ms)
    √ Should call register API with valid data (21 ms)
  Teams Component - White Box Testing
    √ Should initialize with empty new team form (60 ms)
    √ Should generate team code when type is selected (39 ms)
    √ Should validate team creation with all required fields (34 ms)
  State Management - White Box Testing
    √ Should handle state updates correctly in LoginForm (6 ms)
    √ Should clear form after successful submission (6 ms)

Test Suites: 1 passed, 1 total
Tests:       15 passed, 15 total
Snapshots:   0 total
Time:        3.109 s
Ran all test suites.
(base) PS D:\PROJECTS\mern\scoramp\client>
```