Inlyne - Team Test Plan and Report

Katarina, Taiga, Jivaj, Jason, Alex, Manuel

June 3, 2025

Testing Approach :

The testing approach for our Inlyne VSCode extension and web client project follows a multi-layered strategy informed by both unit test patterns and quality assurance concepts. At the unit level, we isolate and test individual components such as the editor state manager, the document loader, and the communication layer using test doubles (e.g., stubs for API responses and spies for WebSocket events) to control indirect inputs and observe outputs, enabling us to verify logic without external dependencies. These tests follow the F.I.R.S.T. principles (Fast, Independent, Repeatable, Self-validating, Timely) to ensure efficiency and maintainability. However, given the complexity of synchronizing real-time document editing between the VSCode extension and the web client, unit tests alone are insufficient. Therefore, we also emphasize system testing and integration testing to validate functional correctness, performance under load, and reliability across multiple clients. System testing scenarios are derived from user stories and acceptance criteria, focusing on realistic user interactions (e.g., editing, syncing, and linking documents) rather than theoretical code paths. While we initially relied on exploratory manual testing on different environments and static analysis (e.g., linting, type checking), we now recognize the importance of developing automated system tests using frameworks such as Playwright or Puppeteer for the web client and VSCode integration tests using vscode-test or Mocha for the extension. These tests should simulate end-to-end interactions, like creating an account, loading documents, typing, syncing, and embedding rich media, while also verifying expected behaviors under stress conditions such as network latency or server failure. Going forward, we propose incorporating boundary testing, equivalence class partitioning, and random input testing to broaden our test coverage beyond typical scenarios and proactively uncover edge-case bugs.

Our testing approach for the Inlyne server API is fully automated and designed to validate correctness across unit, branch, and path levels. All tests are implemented in a Python-based test suite located in the main/test directory of the release branch. These tests cover both the `/user` and `/docs` endpoints provided by the Java Spring server.

Unit Tests & Integration Tests:
While the server code does not isolate traditional unit-testable components (eg, pure functions), our automated tests serve as high-level integration and system tests by simulating real API usage. For example, the `test_basic_flow()` verifies user registration, document creation, public document sharing, and unauthenticated fetch, representing a full end-to-end path. Each

API operation is exercised through realistic payloads, emulating production usage rather than mocked inputs.

Branch and Path Testing:
Our `test_document_permissions()` method is particularly focused on branch/path coverage. It exercises various authorization flows:

- A user creating documents
- Granting permissions to another user
- Accessing documents with valid/invalid tokens
- Reading public documents anonymously

These tests hit key control branches in both `DocumentController` and `UserController` such as:

- Valid vs. invalid request types
  Authenticated vs. unauthenticated headers
- Ownership and role-based access logic
- Optional parameter handling in `GET` routes

Error and Edge Case Handling:
We include tests for invalid tokens, missing fields, and post-deletion access attempts (`test_cleanup_delete_user()`), helping to confirm robust error responses like `403 Forbidden`, `404 Not Found`, and `400 Bad Request`.

Manual Testing:
Manual testing was minimal and limited to initial smoke testing during development. The test suite provides reproducibility and can be run locally against the server using `localhost:8080`.

Next Steps:
To strengthen unit-level validation, we recommend introducing Java-side unit tests (e.g., JUnit + Mockito) for isolated service logic (`DocumentService`, `UserService`) and enabling CI-based automation to catch regressions earlier.

Test Report Overview :

As a user, I want to be able to create an account.

Web client scenario:

1. Open Inlyne website.

2. The website should open on the landing page, which includes a sign up button. Click on the sign up button
3. Enter these fields:
    a. Email: `testuser@inlyne.io`
    b. Username: `TestUser`
    c. Password: `SecurePass123!`
    d. Confirm Password: `SecurePass123!`
4. Should see a confirmation that you have been logged in

VScode Extension scenario:

1. Start VSCode; open the Inlyne extension sidebar.
2. Click 'Sign In'; enter:
    - Email: `<testuser@inlyne.io>`
    - Username: `<TestUser>`
    - Password: `<SecurePass123!>`
    - Confirm Password: `<SecurePass123!>`
3. Press 'Create Account' button.
4. User should see a VSCode notification stating: *"Account created successfully."*

As a user I want to have a website where I can type documentation for my code.

Web client scenario:

1. Once logged in, the website should automatically lead you to /home.
2. On the top right, press "+New Document"
3. A new document with a unique doc ID should be created and directed to
4. A Tiptap editor should show up, and a user can start editing from there

As a user, I want to type documentation within visual studio code as an extension.

VSCode Extension scenario:

1. Start VSCode; load the Inlyne extension.
2. Click the "Open Editor" button in the sidebar.
3. A popout editor window should appear in the VSCode editor area, displaying a rich text editor.
4. User should be able to type text into the editor.

As a user, I want to be able to have a link to my documentation that I can embed into my code.

Web client scenario:

1. In the home page, go to the document that the user wants to link to the code.
2. On the top left of the screen, click on the "Share" button

3. Click on the “Copy URL” button, and the link to the document should be copied to the users clipboard
4. Go to your code base and insert the URL in the desired place.

VSCode Extension scenario:

1. In VSCode, open a file containing a link of the form `dockey{abc123}`.
2. Ctrl/Cmd+Click on the link.
3. A popout editor window should open in the VSCode editor area, displaying the corresponding document’s content.
4. User should be able to type in the editor.

As a user, I want what appears on my editor in VSCode to appear in the website and sync the edits.

Web client scenario:

1. In the Inlyne website, create a new document or open an already created document
2. In the editor, enter “Hello”
3. Copy the dockey which can be copied next to the title on the top in the editor view, and paste it to the `<dockey>` field in VSCode and click “Load”
4. The user should be able to see the text “Hello” that was written in the website.
5. Type “ World” in the VSCode editor
6. The website should say both “Hello World”, showing that the syncing is working between the two clients

VSCode Extension scenario:

1. In VSCode, enter `abc123` into the `<dockey>` field in the sidebar; click "Load".
2. In web client, open document with key `abc123`.
3. Type `"Hello"` in the VSCode editor.
4. Web client editor should display `"Hello"` without manual refresh.
5. Type `" World"` in the web client editor.
6. VSCode editor should display `"Hello World"` without manual refresh.
7. Same procedure should hold for webclient scenario

As a user, I want to be able to manage and access multiple documentation links.

Web client scenario:

1. Log into the Inlyne website
2. Once logged in, the user should be directed to /home in which all of their created docs should be displayed
3. Users can also see the most recently opened documents on the left sidebar to quickly access the docs as well

VSCode extension scenario:

4. In VSCode sidebar, enter document key `<docKey123>` in the `<dockey>` field; click "Load".
5. The document editor should display the content of `<docKey123>`.
6. Repeat for `<docKey456>` and verify the correct content is loaded.
7. Enter an invalid document key `<invalidKey>` and click "Load".
8. User should see an error message: *"Access denied or document not found."*

As a user, I want to have automatic saving and syncing across my writing sessions.

Web client scenario:

1. Open a document once logged into Inlyne.
2. Once a document is opened, a user can freely start typing and editing the contents of the document
3. When the user reloads the page, the contents should still be saved as before.

VSCode extension scenario:

1. In VSCode, load document key `<abc123>`; make edits such as typing `"Autosave test"`.
2. Close VSCode without explicitly saving.
3. Reopen VSCode, load document key `<abc123>` again.
4. The editor should display `"Autosave test"`.

As a user, I want to have my documents support rich media.

Web client scenario:

1. Open a document once you logged in to Inlyne
2. In the editor, a user can choose from one of the tools in the Menu Bar above the Tiptap editor, allowing them to change the font color, header type, images, etc.
3. Users can also drag and drop images directly to the tiptap editor

VSCode Extension Scenario:

1. In VSCode, load the document key `<mediaTest123>`; click the "Insert Image" button.
2. Enter image URL: `https://example.com/image.png`; press Enter.
3. Image should appear embedded in the document editor.
4. Open the same document in the web client; verify image is displayed.

As a user, I want to be able to share my documents and set permissions with other accounts.

Web client scenario:

1. Open a document once user logs into Inlyne that the user wants to share
2. Once opened, click on the “Share” button on the top right

3. Users will be able to add an email and set them to either Reader, Writer, or Admin permissions. Reader would only be able to see the document contents, Writer can edit the contents, and Admin will have full access to the documents
4. Once the email and the permissions are set, click “Add Permissions” and the user that was shared to will be able to see the document

As a user I want to collaborate on documents with others in real time so that I can co-edit efficiently.

1. Login into 2 different accounts on 2 different browsers
2. Open a document that both accounts have access to.
3. Both users will be able to edit the document
4. Verify that both users edits appear for both users
5. Caveat: If not logged in, other users edits will not appear unless browser is refreshed