Table of Contents

Test Cases

Test ID	Scenario	Precondition	Input Steps	Expected Result
TC_AUTH_01	Admin Login	System Online	1. Navigate to Login 2. Enter 'admin', 'password'	Redirect to Dashboard
TC_AUTH_02	Invalid Login	System Online	1. Enter 'wrong', 'user'	Show error "Invalid credentials"
TC_CHK_01	Successful Check-in	Exam Created, Student Roster loaded, Models loaded	1. Select Exam X 2. Search 'Student A' 3. Capture Photo (Match) 4. Click Verify	Status 'Success', Roster updated to Present
TC_CHK_02	Face Mismatch (Security)	Same as above	1. Select 'Student A' 2. Capture Photo (Person B) 3. Click Verify	Status 'Fail', Mismatch Alert
TC_CHK_03	Missing Model	Models not downloaded	1. Open Check-in Page	Show "Loading Models..." indefinitely or Error message
TC_SEAT_01	Correct Seat	Student assigned A1	1. Check Seating Display	Show confirmation "Seat A1"
TC_DB_01	Data Persistence	Check-in completed	1. Restart Server 2. Query CheckInLogs	Log entry still exists

TC_CHK_04	Duplicate Check-in	Student A already checked in	1. Attempt to check in Student A again	System warns "Already Checked In" or updates log with new timestamp
TC_CHK_05	Multiple Faces Detected	Two people in camera view	1. Position two faces in frame 2. Click Verify	Error: "Multiple faces detected"
TC_CHK_06	Missing Reference Photo	Student has no ref photo in DB	1. Search 'Student B' (no photo) 2. Attempt Verify	Error: "No reference photo available"
TC_VAL_01	Empty Credentials	Login Page	1. Leave fields empty 2. Click Login	Error: "Username and password required"
TC_SEC_01	**Secure ML Verification**	Student Selected	1. Capture Photo 2. Click Verify	Backend processes image Returns valid Match/NoMatch Frontend shows "Server-Side ML Active"
TC_SEC_02	**Seat Code Verification**	Student at Wrong Seat	1. Capture Photo (Match) 2. Seat Code != Assigned	Status 'Present' but Warning: "Wrong Seat!" Log includes IsSeatCorrect=0
TC_SEC_03	**Secure File Upload**	Manage Roster	1. Create Student 2. Upload via Webcam/File	File saved to src/server/uploads/{uuid} DB stores relative path
TC_UNIT_01	**ML Service Resilience**	Backend	1. Run npm test	All tests pass (Mock Mode active if binaries missing)

Unit Test Cases

```
const { validateSeat, validateExamTime, validateCheckInStatus,
validateMLData } = require('../src/utils/validation');
```

```
const faceService = require('../services/faceService');
```

1. Seating Compliance Validation (2 Tests)

Description:
These tests validate the seating compliance logic to ensure that students are seated according to their assigned seat codes.

Test Cases:

- **Correct Seat Validation:** Verifies that the system returns true when the student's actual seat matches the assigned seat.
- **Incorrect Seat Detection:** Verifies that the system returns false when the student is seated in a different seat than assigned.

Code:

```
describe('Seating Compliance Validation', () => {
    test('Student in correct seat returns true', () => {
        expect(validateSeat('A1', 'A1')).toBe(true);
    });

    test('Student in wrong seat returns false', () => {
        expect(validateSeat('A2', 'A1')).toBe(false);
    });
});
```

2. Exam Time Validation (2 Tests)

Description:
These tests ensure that student check-ins are allowed only within the valid exam time window.

Test Cases:

- **Valid Check-in Time:** Verifies that check-in is accepted when performed at or after the exam start time.
- **Early Check-in Rejection:** Verifies that check-ins attempted before the exam start time are rejected.

Code:

```
describe('Exam Time Validation', () => {
    test('Check-in within window returns true', () => {
        const examStart = new Date();
        const now = new Date();
        expect(validateExamTime(examStart, now)).toBe(true);
    });

    test('Check-in too early returns false', () => {
        const examStart = new Date(Date.now() + 3600000); // 1 hour later
        const now = new Date();
        expect(validateExamTime(examStart, now)).toBe(false);
    });
});
```

3. Business Rules Validation – Duplicate Check-in (2 Tests)

Description:
These tests validate business rules related to preventing duplicate student check-ins.

Test Cases:

- **Duplicate Check-in Detection:** Verifies that the system detects an existing check-in and flags it as a duplicate.
- **First Check-in Allowance:** Verifies that a student with no previous check-ins is allowed to proceed.

Code:

```
describe('Business Rules Validation', () => {
    test('Detects Duplicate Check-in', () => {
        const existingLogs = [{ LogID: 1, Timestamp: new Date() }];
        expect(validateCheckInStatus(existingLogs)).toBe(true); // Is
Duplicate
    });

    test('Allows First Check-in', () => {
        const existingLogs = [];
        expect(validateCheckInStatus(existingLogs)).toBe(false); // Not
Duplicate
    });
});
```

4. ML Service Wrapper Mock – Validation Logic (3 Tests)

Description:
These tests validate the ML verification wrapper logic using mocked confidence scores.
The ML model itself is **not tested**; only the system's interpretation of ML output is validated.

Test Cases:

- **High Confidence Acceptance:** Verifies that a high ML confidence score is accepted as a valid identity match.
- **Low Confidence Rejection:** Verifies that low confidence scores are rejected.
- **Invalid Output Handling:** Verifies that invalid or unexpected data types are safely rejected.

Code:

```
describe('ML Service Wrapper Mock', () => {
    // Mocking the inputs expected from the ML service
    test('Valid confidence score passes', () => {
        const mockScore = 0.85; // High confidence
        expect(validateMLData(mockScore)).toBe(true);
    });

    test('Low confidence score fails', () => {
        const mockScore = 0.4; // Low confidence
        expect(validateMLData(mockScore)).toBe(false);
    });

    test('Invalid type fails', () => {
        expect(validateMLData("high")).toBe(false);
    });
});
```

5. Face Service ML Wrapper – Integration Resilience (3 Tests)

Description:
These tests validate the robustness of the ML face verification service wrapper, ensuring system stability even when ML services are unavailable or input data is invalid.

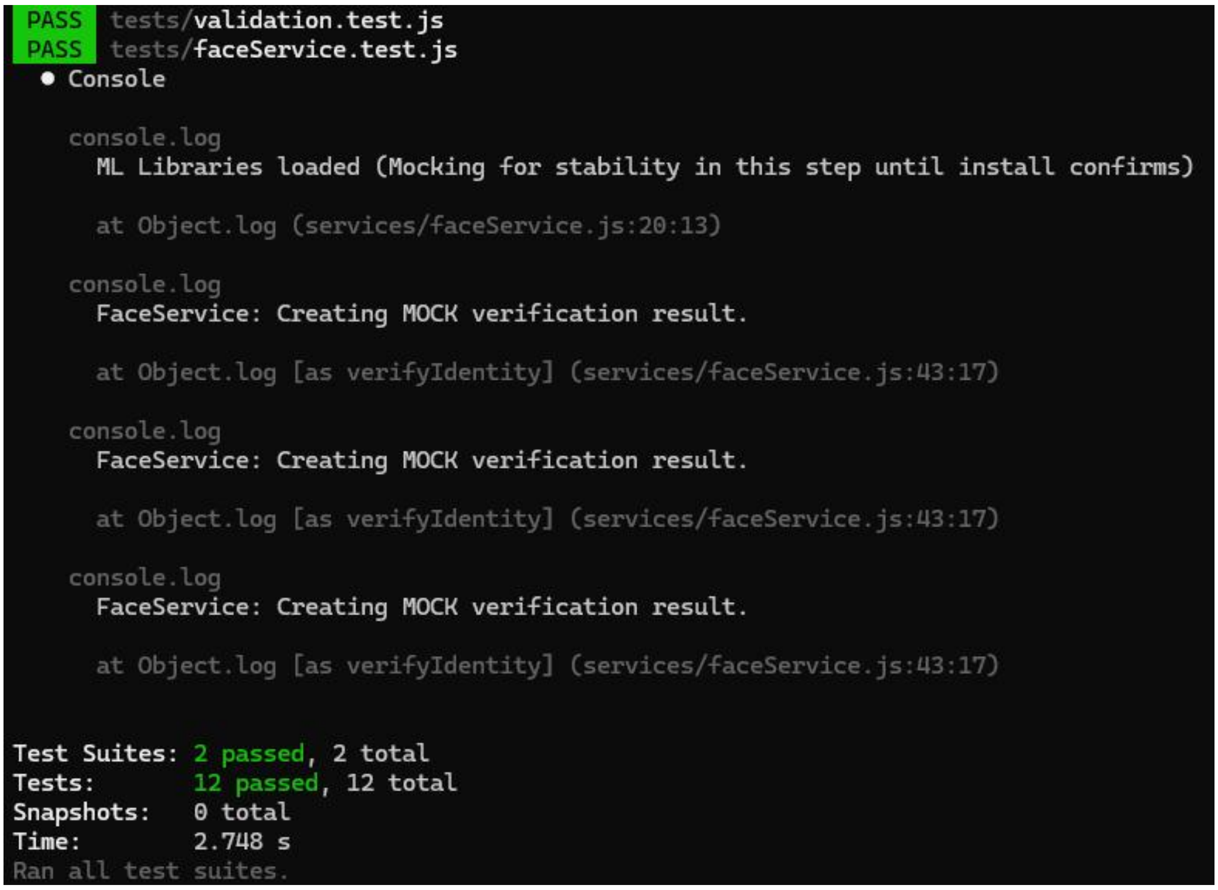
Test Cases:

- **Mock Mode Execution:** Verifies that the service returns a valid mock response when the ML engine is unavailable.
- **Invalid Input Handling:** Verifies that null or empty inputs are handled gracefully without crashing the system.
- **Failure Resilience:** Verifies that corrupted inputs do not crash the service and return a controlled response.

Code:

```
describe('Face Service ML Wrapper', () => {

    // Test 1: Mock Mode Functionality
    test('verifyIdentity should return mock result when ML is unavailable
(or forced to mock)', async () => {
        // Create a dummy buffer
        const dummyBuffer = Buffer.from('fake-image-data');
        const dummyRefs = ['/path/to/ref1.jpg'];

        const result = await faceService.verifyIdentity(dummyBuffer,
dummyRefs);

        // Expect structure matches expected output
        expect(result).toHaveProperty('isMatch');
        expect(typeof result.isMatch).toBe('boolean');
        expect(result).toHaveProperty('score');

        // Since we know our current env likely doesn't have the heavy
binaries loaded in the test runner context
        // (unless installed), it might default to mock.
        // If it actually runs real ML, isMock might be false.
        // But for this requirement, we check that it *runs*.

        // If the service exports a way to check mode, we could assert that.
        // Based on implementation, if it catches load errors, it goes to
mock.
        if (result.isMock) {
            expect(result.isMock).toBe(true);
            expect(result.isMatch).toBe(true); // Mock logic usually returns
true
        }
    });

    // Test 2: Error Handling with Invalid Inputs
    test('verifyIdentity should handle empty/null inputs gracefully', async
() => {
        // Passing null buffer
        try {
            const result = await faceService.verifyIdentity(null, []);
            // Should probably return an error object or succeed with false
            expect(result).toBeDefined();
        } catch (e) {
            // If it throws, that's one behavior, but robust services should
catch internal errors
            // Our service catches errors and returns { isMatch: false,
error: ... }
        }
    });

    // Test 3: Resilience verification
    test('verifyIdentity returns error object on catastrophic failure
(simulated)', async () => {
        // Use a path that definitely doesn't exist if strictly file based,
        // or a buffer that is corrupt if using canvas.
        const corruptBuffer = Buffer.from([0, 0, 0]);

        const result = await faceService.verifyIdentity(corruptBuffer,
['bad/path']);
        // Should not crash process
        expect(result).toHaveProperty('isMatch');
    });

});
```

Unit Test Results



```
 PASS  tests/validation.test.js
 PASS  tests/faceService.test.js
  ● Console

    console.log
      ML Libraries loaded (Mocking for stability in this step until install confirms)

      at Object.log (services/faceService.js:20:13)

    console.log
      FaceService: Creating MOCK verification result.

      at Object.log [as verifyIdentity] (services/faceService.js:43:17)

    console.log
      FaceService: Creating MOCK verification result.

      at Object.log [as verifyIdentity] (services/faceService.js:43:17)

    console.log
      FaceService: Creating MOCK verification result.

      at Object.log [as verifyIdentity] (services/faceService.js:43:17)


Test Suites: 2 passed, 2 total
Tests:       12 passed, 12 total
Snapshots:   0 total
Time:        2.748 s
Ran all test suites.
```