Test Case ID: UC-01 – Create Student Record
Description:
This test case validates that when a new student is created through the API, the record is correctly inserted into MongoDB with all fields preserved, and that the chosen `studentId` remains unique in the collection.
Preconditions:

- MongoDB instance is running and reachable.
- The value of `studentId` in the input object S_{in} does not already exist in the `Students` collection.

Steps:

1. Submit `POST /student/add` with JSON payload S_{in} containing the new student's data.
2. Query MongoDB for any document with `studentId` equal to S_{in}.studentId.

Expected Result (Mathematical Explanation):
If the insertion behaves correctly, there must exist a document S_{db} in the collection whose `studentId` matches the input and whose fields match the input payload:

$$\exists S_{db} : S_{db}.\text{studentId} = S_{in}.\text{studentId}$$

For every application-level field f_i (e.g., name, email, department), the stored document preserves the values:

$$S_{db}[f_i] = S_{in}[f_i], \quad \forall f_i$$

Uniqueness of `studentId` means there is no other document with the same identifier:

$$\nexists S' \neq S_{db} : S'.\text{studentId} = S_{in}.\text{studentId}$$

Result (Mathematical Explanation):
After executing the steps, the observed state satisfies the same constraints: exactly one document S_{db} is associated with the new `studentId`, all fields from S_{in} are present with unchanged values, and no other document shares that `studentId`. Formally:

$$S_{db}.\text{studentId} = S_{in}.\text{studentId}, \quad S_{db}[f_i] = S_{in}[f_i], \quad \nexists S' : S'.\text{studentId} = S_{in}.\text{studentId}$$

Test Case ID: UC-02 – Enrollment Update and Hash Change
Description:
This test ensures that modifying an enrollment record (such as changing a grade) results in a new blockchain hash and that the old hash is preserved as the `previousHash`, maintaining a clear hash history.
Preconditions:

- An existing enrollment record E_{old} is stored in MongoDB.
- A cryptographic hash function H (SHA-256) is used for computing hashes.

Steps:

1. Update E_{old} by changing at least one field (e.g., grade) to form E_{new}.
2. The system computes a new hash $H(E_{new})$.
3. MongoDB updates the enrollment document with a new `currentHash` and shifts the prior hash into `previousHash`.

Expected Result (Mathematical Explanation):
Because E_{new} differs from E_{old} at least in one field, the hash should also differ due to the collision resistance of H:

$$E_{new} \neq E_{old} \Rightarrow H(E_{new}) \neq H(E_{old})$$

The hash-chaining logic should capture the evolution:

$$previousHash_{new} = currentHash_{old}$$

$$currentHash_{new} = H(E_{new})$$

Result (Mathematical Explanation):
The updated document reflects that the old `currentHash` is now stored in `previousHash`, and the new `currentHash` equals $H(E_{new})$, with $H(E_{new}) \neq H(E_{old})$ due to the changed contents. This preserves both integrity and historical traceability of the enrollment.

Test Case ID: UC-03 – Hash Determinism for Identical Data
Description:
This test verifies that the hash of a record is deterministic with respect to its logical content, even if the JSON field ordering differs. This is important to ensure that two semantically equivalent records always map to the same blockchain hash.
Preconditions:

- Two JSON objects J_1 and J_2 represent the same logical record with identical key-value pairs.
- A canonicalization function $C(\cdot)$ sorts and encodes fields in a consistent way.

Steps:

1. Canonicalize both objects to obtain $C(J_1)$ and $C(J_2)$.
2. Compute $h_1 = H(C(J_1))$ and $h_2 = H(C(J_2))$.

Expected Result (Mathematical Explanation):
Because the key-value sets of J_1 and J_2 are identical, the canonicalization removes ordering differences:

$$C(J_1) = C(J_2)$$

Applying the hash function to equal inputs must yield equal outputs:

$$H(C(J_1)) = H(C(J_2))$$

Result (Mathematical Explanation):
The computed hashes h_1 and h_2 coincide, confirming that the system's hashing process depends only on the logical content of the record and not on the syntactic ordering of fields.

Test Case ID: UC-04 – Tamper Detection
Description:
This test examines whether any modification to a stored transcript or enrollment record is detectable by comparing a recomputed hash from MongoDB with the original hash anchored on the blockchain.
Preconditions:

- The blockchain holds $h_{bc} = H(R_{orig})$ for some original record R_{orig}.
- MongoDB currently stores a (possibly modified) record R_{db}.

Steps:

1. Recompute $h_{db} = H(R_{db})$ from the current MongoDB record.
2. Compare h_{db} with the blockchain hash h_{bc}.

Expected Result (Mathematical Explanation):
If any field in R_{db} has been altered relative to R_{orig}, the hash must change:

$$R_{db} \neq R_{orig} \Rightarrow H(R_{db}) \neq h_{bc}$$

Result (Mathematical Explanation):
The comparison $H(R_{db}) \neq h_{bc}$ mathematically expresses that the current record does not match the original state recorded on the blockchain, and thus any tampering is captured by the mismatch of hashes.

Test Case ID: UC-05 – API Routing Correctness
Description:
This test checks that each Flask API endpoint invokes the intended backend service or handler, ensuring that logical responsibilities (e.g., student operations, fee operations) are not mixed up in the routing layer.
Preconditions:

- Flask application is running with all routes registered.
- Each endpoint e_i has an intended handler $f_i^{intended}$.

Steps:

1. Invoke each endpoint e_i once (e.g., `/student`, `/enrollment`, `/fee`, `/verifyRecord`).
2. Observe which backend function or service f_i is actually executed.

Expected Result (Mathematical Explanation):
The routing function R should map each endpoint to its intended handler:

$$R(e_i) = f_i^{intended} \quad \forall i$$

Result (Mathematical Explanation):
For every tested endpoint, the observed handler f_i coincides with the intended mapping $f_i^{intended}$, confirming that $R(e_i)$ equals $f_i^{intended}$ for all invoked routes.

Test Case ID: UC-06 – Blockchain Hash Stored in MongoDB
Description:
This test ensures that the hash produced by the blockchain for a fee transaction is accurately reflected in the corresponding MongoDB document, allowing later verification against the chain.
Preconditions:

- A fee transaction payload T is submitted.
- The blockchain computes $h_{bc} = H(T)$ for this transaction.

Steps:

1. Submit the fee transaction through the API.
2. Inspect the resulting fee document in MongoDB, focusing on the field `txnHash`.

Expected Result (Mathematical Explanation):
The stored hash in MongoDB must equal the hash produced from the transaction payload:

$$txnHash_{db} = H(T)$$

Result (Mathematical Explanation):
The value found in `txnHash` matches $H(T)$, indicating that the database representation of the transaction is cryptographically linked to the blockchain entry.

Test Case ID: UC-07 – Ganache Offline Handling
Description:
This test validates how the system behaves when the blockchain node (Ganache) is offline. The intention is to ensure that transactions are not falsely recorded as finalized and are instead marked as pending.
Preconditions:

$$\neg reachable(BC)$$

(i.e., the blockchain node cannot be contacted.)
Steps:

1. Submit a fee or enrollment transaction that would normally generate a blockchain hash.
2. Check the stored document in MongoDB for fields such as `txnHash` and `status`.

Expected Result (Mathematical Explanation):
Because the blockchain is not reachable, no hash can be created or finalized:

$$txnHash = null$$

and the record is not considered confirmed:

$$status = \text{"pending"}$$

Result (Mathematical Explanation):
The database state contains a `null` or empty `txnHash` and a `status` set to "pending", reflecting that the transaction awaits blockchain confirmation.

Test Case ID: IT-01 – End-to-End Flow
Description:
This integration test follows the full path of a transaction from the GUI to the API, into MongoDB, then to the blockchain, and finally through verification. It demonstrates that data at each layer stays consistent and verifiable.
Preconditions:

- GUI, API, MongoDB, and Blockchain components are all operational.

Steps:

1. Create a new fee or record through the GUI.
2. API writes the core record into MongoDB.
3. Blockchain stores the hash h_{bc} derived from the record.
4. MongoDB updates the record with the same hash $txnHash_{db}$.
5. Verification endpoint recomputes the hash from MongoDB and compares it with h_{bc}.

Expected Result (Mathematical Explanation):
The stored database hash equals the blockchain hash:

$$txnHash_{db} = h_{bc}$$

and the recomputed hash from the record also matches:

$$H(R_{db}) = h_{bc}$$

Result (Mathematical Explanation):
The observed equality $txnHash_{db} = h_{bc}$ and $H(R_{db}) = h_{bc}$ shows that the system maintains integrity across layers and that an end-to-end verification confirms consistency.

Test Case ID: IT-02 – Blockchain Down to Pending State
Description:
This test examines the integration behavior when the blockchain is unavailable during write operations. The system should record a pending state rather than a finalized blockchain hash.
Preconditions:

$$\neg reachable(BC)$$

Steps:

1. Submit a transaction that normally requires blockchain confirmation.
2. Inspect the resulting document in MongoDB.

Expected Result (Mathematical Explanation):
Without blockchain connectivity, the transaction cannot be assigned a valid hash:

$$txnHash = null$$

and the record is left in a non-final state:

$$status = \text{"pending"}$$

Result (Mathematical Explanation):
The stored fields `txnHash` and `status` reflect the absence of a blockchain hash and a pending confirmation status, aligning with the intended failure-handling logic.

Test Case ID: IT-03 – Sharded Query Routing
Description:
This test looks at how queries are routed in a sharded MongoDB cluster. It checks that queries based on a shard key are directed to the correct shard, which is crucial for performance and correctness.
Preconditions:

- Sharding is configured with a partition function $p(k)$ that maps a shard key k to shard $S_{p(k)}$.

Steps:

1. Issue a query Q that filters by shard key k.
2. Observe which shard S_i processes the query.

Expected Result (Mathematical Explanation):
The query is routed to the shard associated with its shard key:

$$route(Q) = S_{p(k)}$$

Result (Mathematical Explanation):
The identified shard handling the query coincides with $S_{p(k)}$, indicating that the routing behavior adheres to the configured partitioning function.

Test Case ID: IT-04 – Replica Failover
Description:
This test evaluates the failover behavior in a MongoDB replica set. When the primary node fails, another node should become the new primary within a bounded time, without losing acknowledged writes.
Preconditions:

- A MongoDB replica set is running with a designated primary node P.

Steps:

1. Terminate the primary node P at time t_0.
2. Observe the election process until a new primary is selected at time t_{new}.

Expected Result (Mathematical Explanation):
The recovery time objective (RTO) restricts how long failover can take:

$$t_{new} - t_0 \leq T_{RTO}$$

The recovery point objective (RPO) indicates no loss of committed data:

$$RPO = 0$$

Result (Mathematical Explanation):
The measured interval $t_{new} - t_0$ is within the allowed T_{RTO}, and all writes acknowledged before t_0 remain present after failover, meaning the system satisfies the intended RTO and RPO constraints.

Test Case ID: IT-05 – Cross-Node Tamper Detection
Description:
This test checks whether tampering with data on one node of the distributed system can be detected from another node by comparing against the blockchain hash.
Preconditions:

- Multiple database nodes share replicated copies of the records.
- Blockchain stores $h_{bc} = H(R_{orig})$ for some original record.

Steps:

1. Modify the record on node N_1, yielding $R_{db}^{(1)}$.
2. On a different node N_j, run verification to obtain $H(R_{db}^{(j)})$.

Expected Result (Mathematical Explanation):
Once replication propagates the tampered state, the recomputed hash no longer matches the blockchain:

$$H(R_{db}^{(j)}) \neq h_{bc}$$

Result (Mathematical Explanation):
The hash calculated on node N_j differs from h_{bc}, showing that the system can detect tampering from any node that holds the modified copy.

Test Case ID: IT-06 – Large Dataset Performance
Description:
This test investigates whether the system maintains acceptable throughput and latency when scaling to tens or hundreds of thousands of records, as defined in the evaluation metrics.
Preconditions:

$$N \in [10^4, 2 \times 10^5]$$

(i.e., the database contains between ten thousand and two hundred thousand records.)
Steps:

1. Execute a workload that performs CRUD operations and hash computations on the dataset of size N.
2. Measure throughput $\text{TPS}(N)$ and the 95th percentile write and verification latencies, $L_{write}^{p95}(N)$ and $L_{verify}^{p95}(N)$.

Expected Result (Mathematical Explanation):
The system should meet the configured performance bounds:

$$\text{TPS}(N) \geq TPS_{target}$$

$$L_{write}^{p95}(N) \leq L_{write}^{max}$$
$$L_{verify}^{p95}(N) \leq L_{verify}^{max}$$

Result (Mathematical Explanation):
The measured values for $\text{TPS}(N)$, $L_{write}^{p95}(N)$, and $L_{verify}^{p95}(N)$ satisfy these inequalities, indicating that performance goals are maintained even under larger dataset sizes.