Lab 1 Assignment 3

Group 30 - Daniel Mohagheghifard, Adam Abdullah

Work distribution (Assignment 3)
Daniel and I teamed up throughout:

- **We kicked off with a whiteboard session where Daniel sketched out all the schema rules and mapped them to individual test ideas.**
- **I took those notes and wrote the first drafts of the pytest fixture and the `test_dao_create.py` file.**
- **Daniel then ran the tests, caught the duplicate-insert mismatch, and we tweaked those three tests together.**
- **I polished the test code formatting, captured the final console output, and updated our repo.**
- **In the end, we sat side-by-side to review every case, make sure nothing was missed, and write up this summary.**

1. Test Levels

1.1 Difference in scope between unit and integration tests

Unit tests check one small piece of code in isolation—typically a single function or class method—to make sure it does exactly what you expect. Integration tests combine several pieces (for example, your DAO talking to a real MongoDB) to verify they work together correctly. Unit tests run fast and focus on logic inside a unit; integration tests are a bit slower and focus on the interactions between units.

1.2 Different purposes of mocking in unit vs. integration tests

- **Unit tests** use mocks to replace every external dependency (databases, web services, file systems) so you only test the code inside the unit. This keeps tests fast and predictable.

- **Integration tests** typically avoid mocking the main external systems under test (e.g., you let your DAO hit a real or in-memory MongoDB) so you can catch problems in the real interactions. You might still mock minor pieces (like a slow third-party API) to keep the test suite stable, but the core integration points stay real.

2.1 List of test cases

Using boundary/value and equivalence-partitioning, we cover for each collection:

2.1.1- Task Collection

1. **Valid minimal insert**
    - **Input:** `{ title: "A", description: "desc" }`
    - **Expected:** success (new document returned with `_id`)
2. **Missing required field**
    - **Input:** `{ description: "desc" }` *(no `title`)*
    - **Expected:** `WriteError`
3. **Wrong type for required field**
    - **Input:** `{ title: 123, description: "desc" }` *(`title` must be string)*
    - **Expected:** `WriteError`
4. **Valid optional field**
    - **Input:** `{ title: "B", description: "desc", startdate: <Date> }`
    - **Expected:** success
5. **Wrong type for optional field**
    - **Input:** `{ title: "C", description: "desc", startdate: "not-a-date" }`
    - **Expected:** `WriteError`

2.1.2- User Collection

1. **Valid minimal insert**
    - **Input:** `{ firstName: "F", lastName: "L", email: "e@x.com" }`
    - **Expected:** success
2. **Missing required field**
    - **Input:** `{ firstName: "F", email: "e@x.com" }` *(no `lastName`)*
    - **Expected:** `WriteError`
3. **Wrong type for required field**
    - **Input:** `{ firstName: "F", lastName: "L", email: 42 }` *(`email` must be string)*
    - **Expected:** `WriteError`
4. **Valid optional field**
    - **Input:** `{ firstName: "F", lastName: "L", email: "e@x.com", tasks: [ ObjectId() ] }`
    - **Expected:** success
5. **Wrong type for optional field**
    - **Input:** `{ firstName: "F", lastName: "L", email: "e@x.com", tasks: ["bad"] }`
    - **Expected:** `WriteError`

2.1.3- Todo Collection

1. **Valid minimal insert**
    - **Input:** `{ description: "Do X" }`
    - **Expected:** success
2. **Missing required field**
    - **Input:** `{}` *(no `description`)*
    - **Expected:** `WriteError`
3. **Wrong type for required field**
    - **Input:** `{ description: false }` *(must be string)*
    - **Expected:** `WriteError`
4. **Valid optional field**
    - **Input:** `{ description: "Do Y", done: true }`
    - **Expected:** success
5. **Wrong type for optional field**
    - **Input:** `{ description: "Do Z", done: "yes" }` *(`done` must be bool)*
    - **Expected:** `WriteError`

2.1.4- Video Collection

1. **Valid minimal insert**
    - **Input:** `{ url: "http://..." }`
    - **Expected:** success
2. **Missing required field**
    - **Input:** `{}` *(no `url`)*
    - **Expected:** `WriteError`
3. **Wrong type for required field**
    - **Input:** `{ url: 123 }` *(must be string)*
    - **Expected:** `WriteError`
4. **Valid optional field**
    - *(none defined—no optional properties)*
    - **Input:** *n/a*
    - **Expected:** *n/a*
5. **Wrong type for optional field**
    - *(none defined—skip)*
    - **Input:** *n/a*
    - **Expected:** *n/a*

> ***Note:*** *each collection also has a "duplicate-insert" case—insert the same valid payload twice and expect success (distinct `_id`s)—but that is outside the core five schema-validation scenarios.*

2.2 Pytest fixture

In `backend/conftest.py`, we point at a throw-away `edutask_test` DB and clean up:

https://github.com/tr3sp4ss3rexe/bsv-edutask/blob/master/backend/conftest.py

2.3 Test implementation link

All 21 cases live in:

`backend/test/test_dao_create.py`
https://github.com/tr3sp4ss3rexe/bsv-edutask/blob/master/backend/test/test_dao_create.py

2.4 Test execution report

```
adam@Adam:~/sys-verification/bsv-edutask/backend$ pytest test/test_dao_create.py -q
.....................                                                    [100%]

---------- coverage: platform linux, python 3.10.12-final-0 ----------
Name                                Stmts   Miss  Cover   Missing
-----------------------------------------------------------------
src/controllers/__init__.py             0      0   100%
src/controllers/controller.py          31     31     0%   1-103
src/controllers/taskcontroller.py      68     68     0%   1-139
src/controllers/todocontroller.py      21     21     0%   1-40
src/controllers/usercontroller.py      24     24     0%   1-46
src/util/dao.py                        67     32    52%   79-83, 101-118, 134-141, 156-162, 172-173
src/util/validators.py                  7      0   100%
-----------------------------------------------------------------
TOTAL                                 218    176    19%

21 passed in 0.32s
```

All integration tests passed, confirming `create()` enforces the JSON-schema validators and talks to the test database correctly.