IT314: Software Engineering
Group - 4 (Real Estate Management System)

Unit Testing

We have used **Jest** for writing test cases, which is a testing framework designed for JavaScript applications.

Testing Framework : `"jest": "^29.7.0"`
Assertion library : `"babel-jest": "^29.7.0"`
Other: `"sinon": "^19.0.2"`

User Controller:

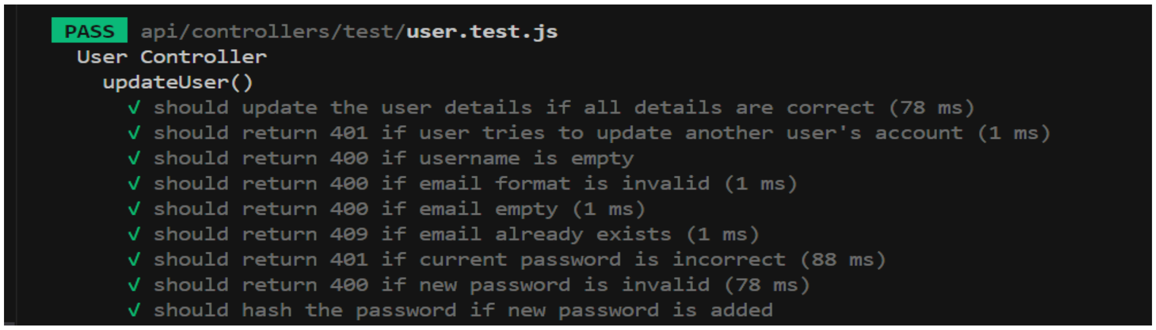
a) updateUser():

```
describe('updateUser()', () => {
    let req, res, next;

    beforeEach(() => {
      req = {
        user: { id: '123' },
        params: { id: '123' },
        body: {},
      };
      res = {
        status: jest.fn(),
        json: jest.fn(),
      };
      next = jest.fn();
    });
    it('should update the user details if all details are
correct', async () => {
      // Arrange
      req.body = {
        username: 'newUsername',
        email: 'mail@example.com',
        currentPassword: 'oldPassword',
        newPassword: 'newPassword123',
      };
      const currentUser = {
        _id: '123',
        username: 'oldUsername',
        email: 'mail@example.com',
        password: bcryptjs.hashSync('oldPassword', 10),
        _doc: { username: 'newUsername', email:
'newemail@example.com' },
      };
      bcryptjs.compareSync = jest.fn().mockReturnValue(true);
      User.findOne.mockResolvedValue(currentUser);
      validateEmail.mockReturnValue(true);
      validatePassword.mockReturnValue(false);
      User.findOne.mockResolvedValue(currentUser);
      bcryptjs.compareSync = jest.fn().mockReturnValue(true);
      User.findByIdAndUpdate.mockResolvedValue(currentUser);

      // Act
      await updateUser(req, res, next);

      // Assert
      expect(res.status).toHaveBeenCalledTimes(1);
    });
    it('should return 401 if user tries to update another
user\'s account', async () => {
      // Arrange
      req.user.id = '456';

      // Act
      await updateUser(req, res, next);

      // Assert
      expect(errorHandler).toHaveBeenCalledWith(401, 'You can
only update your own account!');
      expect(next).toHaveBeenCalledTimes(1);
    });

    it('should return 400 if username is empty', async () => {
      // Arrange
      req.body.username = '';
      // Act
      await updateUser(req, res, next);

      // Assert
      expect(errorHandler).toHaveBeenCalledWith(400, 'Name
cannot be empty!');
      expect(next).toHaveBeenCalledTimes(1);
    });

    it('should return 400 if email format is invalid', async ()
=> {
      // Arrange
      req.body.email = 'invalidEmail';
      validateEmail.mockReturnValue(false);

      // Act
      await updateUser(req, res, next);

      // Assert

expect(validateEmail).toHaveBeenCalledWith('invalidEmail');
```

```
      expect(errorHandler).toHaveBeenCalledWith(400, 'Invalid
Email Format');
      expect(next).toHaveBeenCalledTimes(1);
    });

    it('should return 400 if email empty', async () => {
      // Arrange
      req.body.email = '';
      validateEmail.mockReturnValue(false);

      // Act
      await updateUser(req, res, next);

      // Assert
      expect(validateEmail).toHaveBeenCalledWith('');
      expect(errorHandler).toHaveBeenCalledWith(400, 'Invalid
Email Format');
      expect(next).toHaveBeenCalledTimes(1);
    });

    it('should return 409 if email already exists', async () =>
{
      // Arrange
      req.body.email = 'existingemail@example.com';
      const currentUser = { email: 'oldemail@example.com' };

User.findOne.mockResolvedValueOnce(currentUser).mockResolvedVal
ueOnce({ email: 'existingemail@example.com' });
      validateEmail.mockReturnValue(true);

      // Act
      await updateUser(req, res, next);

      // Assert
      expect(User.findOne).toHaveBeenCalledWith({ email:
'existingemail@example.com' });
      expect(errorHandler).toHaveBeenCalledWith(409, 'Email
already Exists!');
```

```
      expect(next).toHaveBeenCalledTimes(1);
    });

    it('should return 401 if current password is incorrect',
async () => {
      // Arrange
      req.body.currentPassword = 'wrongPassword';
      req.body.newPassword = 'newPassword123';
      const currentUser = { password:
bcryptjs.hashSync('oldPassword', 10) };
      User.findOne.mockResolvedValue(currentUser);
      bcryptjs.compareSync = jest.fn().mockReturnValue(false);

      // Act
      await updateUser(req, res, next);

      // Assert

expect(bcryptjs.compareSync).toHaveBeenCalledWith('wrongPasswor
d', currentUser.password);
      expect(errorHandler).toHaveBeenCalledWith(401, 'Invalid
Credentials!');
      expect(next).toHaveBeenCalledTimes(1);
    });

    it('should return 400 if new password is invalid', async ()
=> {
      // Arrange
      req.body.currentPassword = 'oldPassword';
      req.body.newPassword = 'short';
      const currentUser = { password:
bcryptjs.hashSync('oldPassword', 10) };
      User.findOne.mockResolvedValue(currentUser);
      bcryptjs.compareSync = jest.fn().mockReturnValue(true);
      validatePassword.mockReturnValue('Password is too
short');

      // Act
      await updateUser(req, res, next);

      // Assert
      expect(validatePassword).toHaveBeenCalledWith('short');
      expect(errorHandler).toHaveBeenCalledWith(400, 'Password
is too short');
      expect(next).toHaveBeenCalledTimes(1);
    });

    it('should hash the password if new password is added',
async () => {
      // Arrange
      req.body.newPassword = 'New@pass123';
      bcryptjs.compareSync = jest.fn().mockReturnValue(true);
      validatePassword.mockReturnValue(true);
      bcryptjs.hashSync =
jest.fn().mockReturnValue('hashed-password');
      // Act
      await updateUser(req, res, next);

      // Assert
      expect(next).toHaveBeenCalledTimes(1);
    });
  });
```

Result:

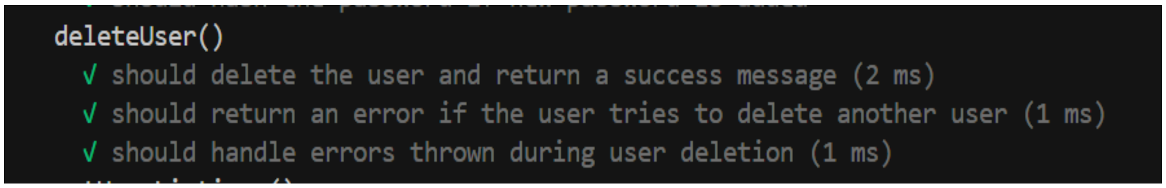


```
 PASS  api/controllers/test/user.test.js
  User Controller
    updateUser()
      √ should update the user details if all details are correct (78 ms)
      √ should return 401 if user tries to update another user's account (1 ms)
      √ should return 400 if username is empty
      √ should return 400 if email format is invalid (1 ms)
      √ should return 400 if email empty (1 ms)
      √ should return 409 if email already exists (1 ms)
      √ should return 401 if current password is incorrect (88 ms)
      √ should return 400 if new password is invalid (78 ms)
      √ should hash the password if new password is added
```

b) deleteUser();

```
describe('deleteUser()', () => {
    let req, res, next;

    beforeEach(() => {
        req = {
            user: { id: '123' },
            params: { id: '123' }
        };
        res = {
            status: jest.fn().mockReturnThis(),
            json: jest.fn(),
            clearCookie: jest.fn()
        };
        next = jest.fn();
    });
    it('should delete the user and return a success message',
async () => {
        // Arrange
        User.findByIdAndDelete.mockResolvedValue(true);

        // Act
        await deleteUser(req, res, next);

        // Assert

expect(User.findByIdAndDelete).toHaveBeenCalledWith('123');

expect(res.clearCookie).toHaveBeenCalledWith('access_token');
        expect(res.status).toHaveBeenCalledWith(200);
        expect(res.json).toHaveBeenCalledWith('User has been
deleted!');
    });

    it('should return an error if the user tries to delete
another user', async () => {
        // Arrange
        req.user.id = '456';
        // Act
        await deleteUser(req, res, next);

        // Assert
        expect(next).toHaveBeenCalledWith(errorHandler(401,
'You can only delete your own account!'));
    });

    it('should handle errors thrown during user deletion',
async () => {
        // Arrange
        const error = new Error('Database error');
        User.findByIdAndDelete.mockRejectedValue(error);


        await deleteUser(req, res, next);

        // Assert
        expect(next).toHaveBeenCalledWith(error);
    });
});
```

Result:

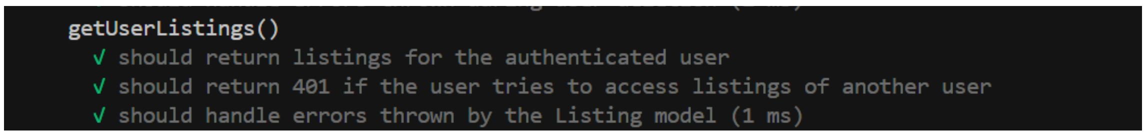


```
  deleteUser()
    √ should delete the user and return a success message (2 ms)
    √ should return an error if the user tries to delete another user (1 ms)
    √ should handle errors thrown during user deletion (1 ms)
```

(c) getuserListings();

```
describe('getUserListings()', () => {
  let req, res, next;

  beforeEach(() => {
      req = {
          user: { id: 'user123' },
          params: { id: 'user123' }
      };
      res = {
          status: jest.fn().mockReturnThis(),
          json: jest.fn()
      };
      next = jest.fn();
  });

  it('should return listings for the authenticated user', async
() => {
      // Arrange
      const mockListings = [{ id: 'listing1' }, { id:
'listing2' }];
      Listing.find.mockResolvedValue(mockListings);

      // Act
      await getUserListings(req, res, next);

      // Assert
      expect(Listing.find).toHaveBeenCalledWith({ userRef:
'user123' });
      expect(res.status).toHaveBeenCalledWith(200);
      expect(res.json).toHaveBeenCalledWith(mockListings);
  });

  it('should return 401 if the user tries to access listings of
another user', async () => {
      // Arrange
      req.params.id = 'anotherUser';

      // Act
      await getUserListings(req, res, next);

      // Assert
      expect(next).toHaveBeenCalledWith(errorHandler(401, 'You can only view your own listings!'));
  });

  it('should handle errors thrown by the Listing model', async () => {
      // Arrange
      const error = new Error('Database error');
      Listing.find.mockRejectedValue(error);

      // Act
      await getUserListings(req, res, next);

      // Assert
      expect(next).toHaveBeenCalledWith(error);
  });
});
```

Result:

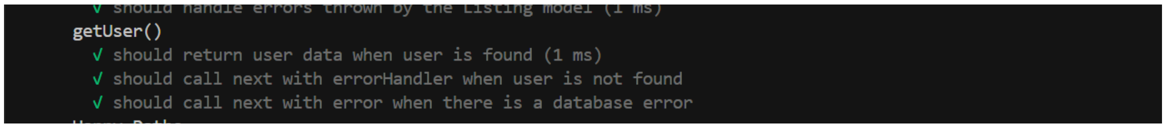


```
  getUserListings()
    √ should return listings for the authenticated user
    √ should return 401 if the user tries to access listings of another user
    √ should handle errors thrown by the Listing model (1 ms)
```

(d) getUser();

```
describe('getUser()', () => {
  let req, res, next;

  beforeEach(() => {
    req = { params: { id: '123' } };
    res = {
          status: jest.fn().mockReturnThis(),
          json: jest.fn(),
      };
      next = jest.fn();
  });

  it('should return user data when user is found', async () =>
{
      // Arrange
      const mockUser = { _id: '123', name: 'John Doe', email:
'john@example.com' };
      User.findById.mockResolvedValue(mockUser);

      // Act
      await getUser(req, res, next);

      // Assert
      expect(User.findById).toHaveBeenCalledWith('123');
      expect(res.status).toHaveBeenCalledWith(200);
      expect(res.json).toHaveBeenCalledWith(mockUser);
      expect(next).not.toHaveBeenCalled();
  });

  it('should call next with errorHandler when user is not
found', async () => {
      // Arrange
      User.findById.mockResolvedValue(null);
      const errorMessage = 'User Not Found!';
      errorHandler.mockReturnValue(new Error(errorMessage));

      // Act
      await getUser(req, res, next);

      // Assert
      expect(User.findById).toHaveBeenCalledWith('123');
      expect(next).toHaveBeenCalledWith(expect.any(Error));
      expect(next.mock.calls[0][0].message).toBe(errorMessage);
      expect(res.status).not.toHaveBeenCalled();
      expect(res.json).not.toHaveBeenCalled();
  });

  it('should call next with error when there is a database
error', async () => {
      // Arrange
      const dbError = new Error('Database Error');
      User.findById.mockRejectedValue(dbError);

      // Act
      await getUser(req, res, next);

      // Assert
      expect(User.findById).toHaveBeenCalledWith('123');
      expect(next).toHaveBeenCalledWith(dbError);
      expect(res.status).not.toHaveBeenCalled();
      expect(res.json).not.toHaveBeenCalled();
  });
});

describe('Happy Paths', () => {
  let req, res, next;

  beforeEach(() => {
      req = {
          user: { id: 'user123' },
          params: { id: 'user123' }
      };
      res = {
          status: jest.fn().mockReturnThis(),
          json: jest.fn()
      };
      next = jest.fn();
  });

  it('should return visit slots for the user when user ID
matches', async () => {
      // Arrange
      const mockVisitSlots = [{ id: 'slot1' }, { id: 'slot2'
}];
      VisitSlot.find.mockResolvedValue(mockVisitSlots);

      // Act
      await getUserVisitsSlots(req, res, next);

      // Assert
      expect(VisitSlot.find).toHaveBeenCalledWith({ buyerId:
'user123' });
      expect(res.status).toHaveBeenCalledWith(200);
      expect(res.json).toHaveBeenCalledWith(mockVisitSlots);
  });

  it('should return 401 error if user ID does not match', async
() => {
      // Arrange
      req.params.id = 'differentUserId';

      // Act
      await getUserVisitsSlots(req, res, next);

      // Assert
      expect(VisitSlot.find).toHaveBeenCalledTimes(1);
      expect(next).toHaveBeenCalledWith(errorHandler(401, 'You
can only view your own visit slots!'));
  });

  it('should handle errors thrown by VisitSlot.find', async ()
=> {
      // Arrange
      const error = new Error('Database error');
      VisitSlot.find.mockRejectedValue(error);

      // Act
      await getUserVisitsSlots(req, res, next);

      // Assert
      expect(next).toHaveBeenCalledWith(error);
  });
});
```

Result:



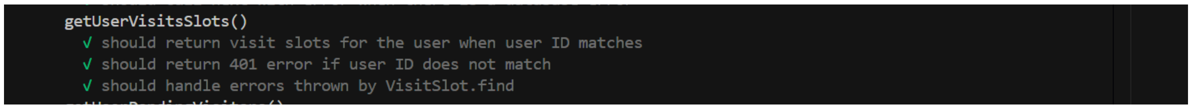
```
        √ should handle errors thrown by the Listing model (1 ms)
      getUser()
        √ should return user data when user is found (1 ms)
        √ should call next with errorHandler when user is not found
        √ should call next with error when there is a database error
```

(e)GetuserVisitSlots();

```
describe('getUserVisitsSlots()', () => {
  let req, res, next;

  beforeEach(() => {
      req = {
          user: { id: 'user123' },
          params: { id: 'user123' }
      };
      res = {
          status: jest.fn().mockReturnThis(),
          json: jest.fn()
      };
      next = jest.fn();
  });

  it('should return visit slots for the user when user ID
matches', async () => {
      // Arrange
      const mockVisitSlots = [{ id: 'slot1' }, { id: 'slot2'
}];
      VisitSlot.find.mockResolvedValue(mockVisitSlots);

      // Act
      await getUserVisitsSlots(req, res, next);

      // Assert
      expect(VisitSlot.find).toHaveBeenCalledWith({ buyerId:
'user123' });
```

```
      expect(res.status).toHaveBeenCalledWith(200);
      expect(res.json).toHaveBeenCalledWith(mockVisitSlots);
  });

  it('should return 401 error if user ID does not match', async () => {
      // Arrange
      req.params.id = 'differentUserId';

      // Act
      await getUserVisitsSlots(req, res, next);

      // Assert
      expect(VisitSlot.find).toHaveBeenCalledTimes(1);
      expect(next).toHaveBeenCalledWith(errorHandler(401, 'You can only view your own visit slots!'));
  });

  it('should handle errors thrown by VisitSlot.find', async () => {
      // Arrange
      const error = new Error('Database error');
      VisitSlot.find.mockRejectedValue(error);

      // Act
      await getUserVisitsSlots(req, res, next);

      // Assert
      expect(next).toHaveBeenCalledWith(error);
  });
});
```

Result:

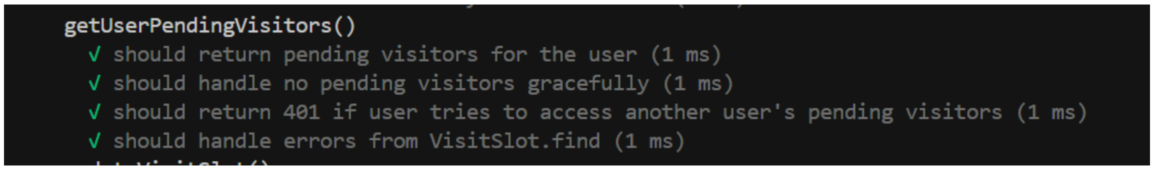


```
getUserVisitsSlots()
  √ should return visit slots for the user when user ID matches
  √ should return 401 error if user ID does not match
  √ should handle errors thrown by VisitSlot.find
```

(f) GetUserPendingVisitSlots();

```
describe('getUserPendingVisitors()', () => {
  let req, res, next;

  beforeEach(() => {
      req = {
          user: { id: 'user123' },
          params: { id: 'user123' }
      };
      res = {
          status: jest.fn().mockReturnThis(),
          json: jest.fn()
      };
      next = jest.fn();
  });

  it('should return pending visitors for the user', async () =>
{
      // Arrange
      const mockPendingVisitors = [{ id: 'visit1' }, { id:
'visit2' }];
      VisitSlot.find.mockResolvedValue(mockPendingVisitors);

      // Act
      await getUserPendingVisitors(req, res, next);

      // Assert
      expect(VisitSlot.find).toHaveBeenCalledWith({ sellerId:
'user123' });
      expect(res.status).toHaveBeenCalledWith(200);

expect(res.json).toHaveBeenCalledWith(mockPendingVisitors);
  });
  it('should handle no pending visitors gracefully', async ()
=> {
      // Arrange
      VisitSlot.find.mockResolvedValue([]);

      // Act
      await getUserPendingVisitors(req, res, next);

      // Assert
      expect(VisitSlot.find).toHaveBeenCalledWith({ sellerId:
'user123' });
      expect(res.status).toHaveBeenCalledWith(200);
      expect(res.json).toHaveBeenCalledWith([]);
  });

  it('should return 401 if user tries to access another user\'s
pending visitors', async () => {
      // Arrange
      req.params.id = 'anotherUser';

      // Act
      await getUserPendingVisitors(req, res, next);

      // Assert
      expect(next).toHaveBeenCalledWith(errorHandler(401, 'You
can only view your own pending visitors!'));
  });

  it('should handle errors from VisitSlot.find', async () => {
      // Arrange
      const error = new Error('Database error');
      VisitSlot.find.mockRejectedValue(error);

      // Act
      await getUserPendingVisitors(req, res, next);

      // Assert
      expect(next).toHaveBeenCalledWith(error);
  });
});
```

Result:



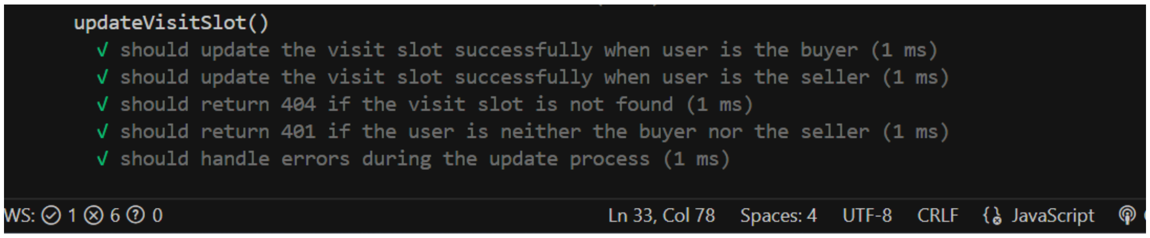
```
getUserPendingVisitors()
  √ should return pending visitors for the user (1 ms)
  √ should handle no pending visitors gracefully (1 ms)
  √ should return 401 if user tries to access another user's pending visitors (1 ms)
  √ should handle errors from VisitSlot.find (1 ms)
```

(g)updateVisitSlot();

```
describe('updateVisitSlot()', () => {
  let req, res, next;

  beforeEach(() => {
      req = {
          params: { id: 'visitSlotId' },
          user: { id: 'userId' },
          body: { date: '2023-10-10', time: '10:00 AM' }
      };
      res = {
          status: jest.fn().mockReturnThis(),
          json: jest.fn()
      };
      next = jest.fn();
  });

  it('should update the visit slot successfully when user is the buyer', async () => {
      // Arrange
      const visitSlot = { buyerId: 'userId', sellerId: 'anotherUserId' };
      VisitSlot.findById.mockResolvedValue(visitSlot);
      VisitSlot.findByIdAndUpdate.mockResolvedValue({ ...visitSlot, ...req.body });

      // Act
      await updateVisitSlot(req, res, next);
      // Assert

expect(VisitSlot.findById).toHaveBeenCalledWith('visitSlotId');

expect(VisitSlot.findByIdAndUpdate).toHaveBeenCalledWith('visit
SlotId', req.body, { new: true });
      expect(res.status).toHaveBeenCalledWith(200);
      expect(res.json).toHaveBeenCalledWith(visitSlot);
  });

  it('should update the visit slot successfully when user is
the seller', async () => {
      // Arrange
      const visitSlot = { buyerId: 'anotherUserId', sellerId:
'userId' };
      VisitSlot.findById.mockResolvedValue(visitSlot);
      VisitSlot.findByIdAndUpdate.mockResolvedValue({
...visitSlot, ...req.body });

      // Act
      await updateVisitSlot(req, res, next);

      // Assert

expect(VisitSlot.findById).toHaveBeenCalledWith('visitSlotId');

expect(VisitSlot.findByIdAndUpdate).toHaveBeenCalledWith('visit
SlotId', req.body, { new: true });
      expect(res.status).toHaveBeenCalledWith(200);
      expect(res.json).toHaveBeenCalledWith(visitSlot);
  });

  it('should return 404 if the visit slot is not found', async
() => {
      // Arrange
      VisitSlot.findById.mockResolvedValue(null);

      // Act
      await updateVisitSlot(req, res, next);

      // Assert

expect(VisitSlot.findById).toHaveBeenCalledWith('visitSlotId');
      expect(next).toHaveBeenCalledWith(errorHandler(404,
'Visit Slot Not Found!'));
  });

  it('should return 401 if the user is neither the buyer nor
the seller', async () => {
      // Arrange
      const visitSlot = { buyerId: 'anotherUserId', sellerId:
'yetAnotherUserId' };
      VisitSlot.findById.mockResolvedValue(visitSlot);

      // Act
      await updateVisitSlot(req, res, next);

      // Assert

expect(VisitSlot.findById).toHaveBeenCalledWith('visitSlotId');
      expect(next).toHaveBeenCalledWith(errorHandler(401, 'You
can only update your own visit slots!'));
  });

  it('should handle errors during the update process', async ()
=> {
      // Arrange
      const visitSlot = { buyerId: 'userId', sellerId:
'anotherUserId' };
      VisitSlot.findById.mockResolvedValue(visitSlot);
      const error = new Error('Database error');
      VisitSlot.findByIdAndUpdate.mockRejectedValue(error);

      // Act
      await updateVisitSlot(req, res, next);
```

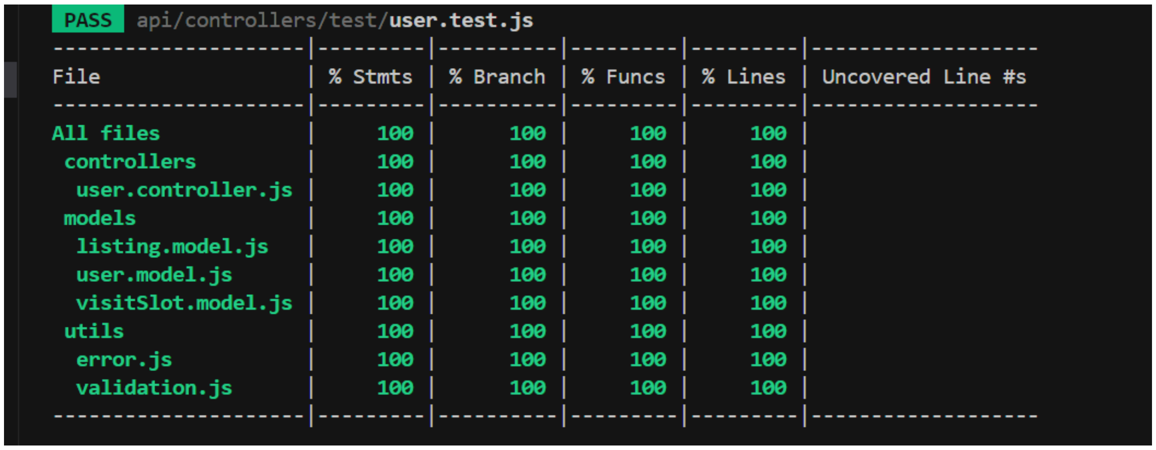
```
        // Assert

expect(VisitSlot.findById).toHaveBeenCalledWith('visitSlotId');
        expect(next).toHaveBeenCalledWith(error);
   });
});
```

Result:



```
    updateVisitSlot()
      √ should update the visit slot successfully when user is the buyer (1 ms)
      √ should update the visit slot successfully when user is the seller (1 ms)
      √ should return 404 if the visit slot is not found (1 ms)
      √ should return 401 if the user is neither the buyer nor the seller (1 ms)
      √ should handle errors during the update process (1 ms)

WS: ⊘ 1 ⊗ 6 ⓘ 0                                   Ln 33, Col 78   Spaces: 4   UTF-8   CRLF   {} JavaScript
```

Code Coverage:



```
 PASS  api/controllers/test/user.test.js
----------------------|---------|----------|---------|---------|-------------------
File                  | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------------|---------|----------|---------|---------|-------------------
All files             |     100 |      100 |     100 |     100 |
 controllers          |     100 |      100 |     100 |     100 |
  user.controller.js  |     100 |      100 |     100 |     100 |
 models               |     100 |      100 |     100 |     100 |
  listing.model.js    |     100 |      100 |     100 |     100 |
  user.model.js       |     100 |      100 |     100 |     100 |
  visitSlot.model.js  |     100 |      100 |     100 |     100 |
 utils                |     100 |      100 |     100 |     100 |
  error.js            |     100 |      100 |     100 |     100 |
  validation.js       |     100 |      100 |     100 |     100 |
----------------------|---------|----------|---------|---------|-------------------
```