



IT 314

Software Engineering

SPRINT 1 – USER CONTROLLER UNIT TESTING

Fork & Feast

Group 28

➤ Tool Used:

Mocha @10.8.2

Chai @4.3.4

Other – Sinon @19.0.2

Link to the source code: [UserController](#)

Link to the test file: [UserController test](#)

1. signup_post:

This function takes care of the sign up functionality of the user like giving error in case the user is already signed in or in case of an invalid domain in the email field and so on.

a) should return 409 if user already exists

```
it('should return 409 if user already exists', async () => {
  const req = {
    body: {
      name: 'TestUser',
      email: 'testuser@gmail.com',
      password: '!Password123',
      confirmPassword: '!Password123',
      isOwner: false
    }
  };
  const res = {
    status: sinon.stub().returnsThis(),
    json: sinon.stub()
  };

  const findOneStub = sinon.stub(usermodel, 'findOne');
  findOneStub.withArgs({ email: 'testuser@gmail.com' }).resolves({ email: 'testuser@gmail.com' });
  findOneStub.resolves(null);

  await signup_post(req, res);

  expect(res.status.calledWith(409)).to.be.true;
  expect(res.json.calledWithMatch({ message: 'User already exists' })).to.be.true;
  expect(findOneStub.calledWith({ email: 'testuser@gmail.com' })).to.be.true;
});
```

It sets up a mock request with user details and stubs the usermodel.findOne method to simulate an existing user. The res object is also mocked, and the

function is called. The test asserts that `res.status` was called with 409 and `res.json` returns a message indicating the user already exists.

This test case is successfully covered which is confirmed with the result shown in the terminal as below:

```
signup_post
✓ should return 409 if user already exists
```

b) should return 400 for an invalid email domain

```
it('should return 400 for an invalid email domain', async () => {
  const req = {
    body: {
      name: 'TestUser',
      email: 'invalidemail@test',
      password: '!Password123',
      confirmPassword: '!Password123',
      isOwner: false
    }
  };
  const res = {
    status: sinon.stub().returnsThis(),
    json: sinon.stub()
  };

  sinon.stub(usermodel, 'findOne').resolves(null);
  sinon.stub(dns, 'resolveMx').callsFake((domain, callback) => {
    // Immediately call the callback with empty result to simulate invalid domain
    setImmediate(() => callback(null, []));
  });

  await signup_post(req, res);

  expect(res.status.calledWith(400)).to.be.true;
  expect(res.json.calledWithMatch({ message: 'Invalid email domain' })).to.be.true;
});
```

It sets up a mock request with user details that include an invalid domain and stubs the `usermodel.findOne` method to simulate no existing user. Additionally, it stubs `dns.resolveMx` to simulate an invalid domain by calling back with an empty result. The test asserts that `res.status` is called with 400 and `res.json` returns a message indicating an invalid email domain.

This test case is also covered:

```
✓ should return 400 for an invalid email domain
```

c) should return 401 if passwords do not match

```
it('should return 401 if passwords do not match', async function() {
  this.timeout(5000); // Increase timeout to 5 seconds

  const req = {
    body: {
      name: 'TestUser',
      email: 'testuser@gmail.com',
      password: '!Password123',
      confirmPassword: 'WrongPassword123',
      isOwner: false
    }
  };
  const res = {
    status: sinon.stub().returnsThis(),
    json: sinon.stub()
  };

  sinon.stub(usermodel, 'findOne').resolves(null);

  // Mock DNS resolution
  sinon.stub(require('dns'), 'resolveMx').callsFake((domain, callback) => {
    callback(null, [{exchange: 'test.com', priority: 10}]);
  });

  await signup_post(req, res);

  expect(res.status.calledWith(401)).to.be.true;
  expect(res.json.calledWithMatch({ message: 'Password not matching' })).to.be.true;
});
```

A mock request is set up with differing password and confirmPassword values. The usermodel.findOne method is stubbed to simulate no existing user, and dns.resolveMx is mocked to return a valid result, indicating a legitimate email domain. The test checks that res.status is called with 401 and res.json returns a message stating "Password not matching".

The result is:

```
✓ should return 401 if passwords do not match
```

2. login_post:

This function takes care of the login test cases like login the user with valid credentials, return an error in case of wrong password or if a restaurant owner is trying to login as a user and so on.

a) should log in the user with valid credentials

```
it('should log in the user with valid credentials', async () => {
  const req = {
    body: {
      email: 'testuser@gmail.com',
      password: '!Password123',
      isOwner: false,
      emailVerified: true
    }
  };
  const res = {
    status: sinon.stub().returnsThis(),
    json: sinon.stub(),
    setHeader: sinon.stub()
  };

  const user = {
    email: 'testuser@gmail.com',
    password: '$2b$10$1234567890123456789012',
    isOwner: false,
    emailVerified: true,
    _id: '12345'
  };

  const findOneStub = sinon.stub(usermodel, 'findOne').resolves(user);
  sinon.stub(bcrypt, 'compare').resolves(true);
  sinon.stub(jwt, 'sign').returns('mockedToken');

  await login_post(req, res);

  expect(findOneStub.calledWith({ email: req.body.email })).to.be.true;

  expect(res.status.calledWith(200)).to.be.true;
  expect(res.json.calledWithMatch({
    message: 'User logged in successfully',
    userId: '12345',
    token: 'mockedToken'
  })).to.be.true;
  expect(res.setHeader.calledWith('Authorization', 'Bearer mockedToken')).to.be.true;
});
```

It sets up a mock request containing a valid email and password, and mocks the usermodel.findOne method to return a user object. Additionally, bcrypt.compare is stubbed to simulate a successful password match. The test then calls the login_post function and verifies that res.status is called with 200, res.json returns a success message, and res.setHeader sets the Authorization header to indicate a successful login.

This test case is also passed:

```
login_post
✓ should log in the user with valid credentials
```

b) should return 401 for invalid password

```
it('should return 401 for invalid password', async () => {
  const req = {
    body: {
      email: 'testuser@gmail.com',
      password: '!WrongPassword123', // simulated incorrect password
      isOwner: false
    }
  };
  const res = {
    status: sinon.stub().returnsThis(),
    json: sinon.stub()
  };

  const user = {
    email: 'testuser@gmail.com',
    password: '$2b$10$SomeEncryptedPasswordHash', // simulated encrypted password
    isOwner: false,
    _id: '12345'
  };

  // Mock findOne to return the user
  sinon.stub(usermodel, 'findOne').resolves(user);

  // Mock bcrypt.compare to resolve with false (incorrect password)
  sinon.stub(bcrypt, 'compare').resolves(false);

  // Call the login_post function
  await login_post(req, res);

  // Assertions
  expect(res.status.calledWith(401)).to.be.true;
  expect(res.json.calledWithMatch({ message: 'Enter valid password' })).to.be.true;
});
```

It creates a mock request with an email and an incorrect password, while `usermodel.findOne` is stubbed to return a user object with an encrypted password. The `bcrypt.compare` function is also stubbed to simulate a failed password match by resolving to false.

This is satisfied:

```
✓ should return 401 for invalid password
```

c) should return 401 for mismatched user type

```
it('should return 401 for mismatched user type', async () => {
  req = {
    body: {
      email: 'testuser@gmail.com',
      password: '!Password123',
      isOwner: true
    }
  };

  const user = {
    email: 'testuser@gmail.com',
    password: 'hashedPassword',
    isOwner: false,
    _id: '12345'
  };

  sinon.stub(usermodel, 'findOne').resolves(user);

  await login_post(req, res);

  expect(res.status.calledWith(401)).to.be.true;
  expect(res.json.calledWithMatch({ message: 'Invalid User' })).to.be.true;
});
```

It mocks a request with the isOwner field set to true, while the user in the database has isOwner: false. The usermodel.findOne method is stubbed to return this user. The test then calls login_post and verifies that res.status is called with 401, and res.json returns a message indicating "Invalid User."

✓ should return 401 for mismatched user type

d) should return 401 for invalid password

This is same as in signup_post.

3. forgotPassword:

This function takes in the email for which password needs to be reset and checks if it is valid and that it is not registered as restaurant owner.

a) should return 401 when owner tries to use customer route

```
it('should return 401 when owner tries to use customer route', async () => {
  const req = {
    body: {
      email: 'testuser@gmail.com',
      userType: 'customer' // Trying to use customer route
    }
  };
  const res = {
    status: sinon.stub().returnsThis(),
    json: sinon.stub()
  };

  // Mock user as an owner
  const user = {
    email: 'testuser@gmail.com',
    isOwner: true, // User is an owner
    _id: new mongoose.Types.ObjectId(),
    name: 'TestUser'
  };

  const findOneStub = sinon.stub(usermodel, 'findOne').resolves(user);

  await forgotPassword(req, res);

  expect(res.status.calledWith(401)).to.be.true;
  expect(res.json.calledWithMatch({ message: 'Unauthorized' })).to.be.true;
  expect(findOneStub.calledWith({ email: req.body.email })).to.be.true;
});
```

It mocks a req object with userType: 'customer' and a user object marked as an owner (isOwner: true). The test ensures that the forgotPassword function returns a 401 Unauthorized status with the appropriate error message and confirms that the findOne method of the user model is called with the correct query (email).

```
forgotPassword
  ✓ should return 401 when owner tries to use customer route
```


b) should return 401 for unregistered email

```
it('should return 401 for unregistered email', async () => {
  const req = {
    body: {
      email: 'invaliduser@gmail.com',
      userType: 'customer'
    }
  };
  const res = {
    status: sinon.stub().returnsThis(),
    send: sinon.stub()
  };

  sinon.stub(usermodel, 'findOne').resolves(null);

  await forgotPassword(req, res);

  expect(res.status.calledWith(401)).to.be.true;
  expect(res.send.calledWithMatch({ message: 'Enter valid registered Email Id' })).to.be.true;
});
```

It sets up a mock request with an email (invaliduser@gmail.com) that doesn't exist in the database. The userModel.findOne method is stubbed to return null, indicating the email is unregistered. The test then calls forgotPassword and checks that res.status is called with 401 and res.send returns a message stating "Enter valid registered Email Id," confirming the system handles unregistered emails correctly.

✓ should return 401 for unregistered email

c) should handle error when creating a new token

```
it('should handle error when creating new token', async () => {
  req = {
    body: {
      email: 'testuser@gmail.com',
      userType: 'customer'
    }
  };

  const user = {
    email: 'testuser@gmail.com',
    isOwner: false,
    _id: new mongoose.Types.ObjectId(),
    name: 'TestUser'
  };

  sinon.stub(usermodel, 'findOne').resolves(user);
  sinon.stub(Token, 'findOne').resolves(null);
  sinon.stub(Token.prototype, 'save').rejects(new Error('Token save failed'));

  await forgotPassword(req, res);

  expect(res.status.calledWith(401)).to.be.true;
  expect(res.json.calledWithMatch({ message: 'Token save failed' })).to.be.true;
});
```

It mocks the database queries to find the user and the token, and forces the `Token.prototype.save` method to throw an error. The test expects the function to handle the error by returning a 401 status and a JSON response with the error message `Token save failed`.

✓ should handle error when creating new token

4. resetPassword:

The `resetPassword` function handles password resets by verifying an existing user, it then ensures a password is provided, then updates the user's password. If any step fails, it returns appropriate error messages.

a) should reset the password for a valid token

```
it('should reset the password for a valid token', async () => {
  const tokendata = {
    userid: '12345',
    token: 'validtoken',
    deleteOne: sinon.stub().resolves()
  };
  const user = {
    _id: '12345',
    save: sinon.stub().resolves()
  };

  sinon.stub(Token, 'findOne').resolves(tokendata);
  sinon.stub(usermodel, 'findOne').resolves(user);

  const req = {
    params: { token: 'validtoken' },
    body: { password: '!NewPassword123' }
  };
  const res = {
    status: sinon.stub().returnsThis(),
    json: sinon.stub(),
    send: sinon.stub()
  };

  await resetPassword(req, res);

  expect(user.save.calledOnce).to.be.true;
  expect(res.status.calledWith(200)).to.be.true;
  expect(res.send.calledWithMatch({ message: 'Password changed successfully' })).to.be.true;
});
```

The test checks that the password is updated (via `user.save()`), the response status is 200, and the correct success message ("Password changed successfully") is returned. The test also ensures the `deleteOne` method is called to remove the token after the password reset.

resetPassword
✓ should reset the password for a valid token

b) should return 401 when no password provided

```
it('should return 401 when no password provided', async () => {
  req = {
    params: { token: 'validtoken' },
    body: {}
  };

  sinon.stub(Token, 'findOne').resolves({
    userid: '12345',
    token: 'validtoken'
  });

  await resetPassword(req, res);

  expect(res.status.calledWith(401)).to.be.true;
  expect(res.send.calledWithMatch({ message: 'Password is required' })).to.be.true;
});
```

It mocks the token data to simulate a valid token and user association. Since the password field is missing in the request body, the test checks that the function correctly responds with a 401 status and the message "Password is required."

✓ should return 401 when no password provided

c) should return 400 when user not found

```
it('should return 400 when user not found', async () => {
  req = {
    params: { token: 'validtoken' },
    body: { password: 'newpassword' }
  };

  sinon.stub(Token, 'findOne').resolves({
    userid: '12345',
    token: 'validtoken'
  });
  sinon.stub(usermodel, 'findOne').resolves(null);

  await resetPassword(req, res);

  expect(res.status.calledWith(400)).to.be.true;
  expect(res.send.calledWithMatch({ message: 'Cannot find user' })).to.be.true;
});
```

It mocks a valid token, but simulates a scenario where the user lookup (`usermodel.findOne`) returns null, indicating no matching user. The test checks that the function responds with a 400 status and the message "Cannot find user."

✓ should return 400 when user not found

5. `confirmEmail`:

This function redirects the user to the login site as he clicks on the confirm email button in the mail.

a) should handle non-existent user

```
it('should handle non-existent user', async () => {
  req = {
    params: { token: 'validtoken' }
  };

  const decoded = { _id: 'userid' };

  sinon.stub(jwt, 'verify').returns(decoded);
  sinon.stub(usermodel, 'findByIdAndUpdate').resolves(null);

  await confirmEmail(req, res);

  expect(res.status.calledWith(400)).to.be.true;
  expect(res.json.calledWithMatch({ message: 'Invalid token or user does not exist' })).to.be.true;
});
```

Here the `usermodel.findOne` is stubbed to return null, indicating that no user is found for that ID. The test checks that the function responds with a 400 status and the appropriate error message, "Invalid token or user does not exist".

✓ should handle non-existent user

b) should redirect customer to customer login page

```
it('should redirect customer to customer login page', async () => {
  req = {
    |   params: { token: 'validtoken' }
  };

  const decoded = { _id: 'userid' };
  const user = { _id: 'userid', isOwner: false };

  sinon.stub(jwt, 'verify').returns(decoded);
  sinon.stub(usermodel, 'findByIdAndUpdate').resolves(user);

  await confirmEmail(req, res);

  expect(res.redirect.calledOnce).to.be.true;
  expect(res.redirect.args[0][0]).to.include('/login?type=customer&verified=true');
});
```

The usermodel.findOne is stubbed to return a user with isOwner: false, indicating they are a customer. The test verifies that the res.redirect method is called once and the redirect URL includes the query parameter ?type=customer, ensuring the user is redirected to the correct login page.

✓ should redirect customer to customer login page

6. sendresetpasswordmail:

This is the function which sends the mail to reset password with a link to the reset password page.

a) should handle email sending error with proper error check

```
it('should handle email sending error with proper error check', (done) => {
  const name = 'Test User';
  const email = 'test@example.com';
  const token = 'test-token';

  const testError = new Error('Email sending failed');

  // Track if error callback was actually triggered
  let errorCallbackTriggered = false;

  sendMailStub.callsFake((options, callback) => {
    errorCallbackTriggered = true;
    callback(testError, null);
  });

  const { sendresetpasswordmail } = require('../controllers/userController');

  sendresetpasswordmail(name, email, token)
    .then(() => {
      expect(errorCallbackTriggered).to.be.true;
      expect(sendMailStub.calledOnce).to.be.true;
      expect(consoleLogSpy.firstCall.args[0]).to.equal(testError);
      expect(consoleLogSpy.firstCall.args[0]).to.be instanceof(Error);
      done();
    })
    .catch(done);
});
```

It mocks a failure in sendMailStub by simulating an error (Email sending failed) in the callback. The test ensures that the error callback is triggered, sendMailStub is called once, and the error is logged correctly to the console. It uses assertions to confirm these behaviors, ensuring robust error handling in the email-sending process.

✓ should handle email sending error

b) should include correct mail options

```
it('should include correct mail options', (done) => {
  const name = 'Test User';
  const email = 'test@example.com';
  const token = 'test-token';

  sendMailStub.callsFake((options, callback) => {
    expect(options.to).to.equal(email);
    expect(options.subject).to.equal('For reset password');
    expect(options.html).to.include(name);
    expect(options.html).to.include(token);
    callback(null, { response: 'Success' });
  });

  const { sendresetpasswordmail } = require('../controllers/userController');

  sendresetpasswordmail(name, email, token)
    .then(() => {
      expect(sendMailStub.calledOnce).to.be.true;
      const mailOptions = sendMailStub.getCall(0).args[0];
      expect(mailOptions.html).to.include(name);
      expect(mailOptions.html).to.not.include('{Username}');
      expect(mailOptions.html).to.not.include('{url}');
      done();
    })
    .catch(done);
});
```

It checks that the to field matches the recipient's email, the subject is set to "For reset password," and the HTML content includes the user's name and the token. Additionally, it ensures that placeholders like {Username} or {url} are not left in the email content. The test confirms that sendMailStub is called once and that the mail options passed to it are correct.

✓ should include correct mail options

c) should properly log success message with response info

```
it('should properly log success message with response info', (done) => {
  const name = 'Test User';
  const email = 'test@example.com';
  const token = 'test-token';

  const successResponse = { response: 'Success Response' };

  sendMailStub.callsFake((options, callback) => {
    callback(null, successResponse);
  });

  const { sendresetpasswordmail } = require('../controllers/userController');

  sendresetpasswordmail(name, email, token)
    .then(() => {
      expect(sendMailStub.calledOnce).to.be.true;
      expect(consoleLogSpy.firstCall.args[0]).to.include('email send successfully');
      expect(consoleLogSpy.firstCall.args[0]).to.include(successResponse.response);
      expect(consoleLogSpy.firstCall.args[0].length).to.be.greaterThan(0);
      done();
    })
    .catch(done);
});
```

It mocks the sendMailStub to simulate a successful email-sending response (Success Response). The test verifies that sendMailStub is called once and checks that the logged message includes "email send successfully" and the success response details. It also asserts that the logged message is non-empty, confirming proper handling of successful email operations.

✓ should properly log success message with response info

7. sendConfirmationEmail

This is the function responsible for sending the mail to confirm email at the time of sign up with a link to the login page.

a) should send email with correct options

```
it('should send email with correct options', async () => {
  const name = 'Test User';
  const email = 'test@example.com';
  const token = 'test-token';

  sendMailStub.resolves();

  const { sendConfirmationEmail } = require('../controllers/userController');

  await sendConfirmationEmail(name, email, token);

  // Verify mail options
  const mailOptions = sendMailStub.getCall(0).args[0];
  expect(mailOptions).to.have.property('to', email);
  expect(mailOptions).to.have.property('subject', 'Account Confirmation');
  expect(mailOptions.html).to.include(name);
  expect(mailOptions.html).to.include(token);

  expect(mailOptions.html).to.include(name);
  expect(mailOptions.html).to.not.include('{Username}');
  expect(mailOptions.html).to.not.include('{url}');
});
```

Similar to the resetpasswordmail.

✓ should send email with correct options

At last it showed us that all our test cases were passed.

31 passing (110ms)

➤ Coverage Report

For this we used the “nyc --reporter=text” in package.json and “npm run coverage” command which gave us the following output:

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
userController.js	100	100	100	100	

Showing that 100% statements, branches and functions were covered in the unit testing.

For in depth report we used “nyc --reporter=lcov” in package.json which gave us an html file highlighting the number of times a particular line ran during the entire testing.

All files

100% Statements 120/120 100% Branches 43/43 100% Functions 12/12 100% Lines 120/120

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File		Statements		Branches		Functions		Lines	
userController.js	<div></div>	100%	120/120	100%	43/43	100%	12/12	100%	120/120

Link to the HTML file: [Coverage Report](#)