



IT 314

Software Engineering

Reservation CONTROLLER
UNIT TESTING

Fork & Feast

Group 28

Tool used:

Mocha: @10.8.2

Chat: @4.3.4

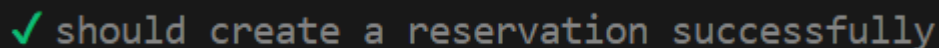
Other – Sinon @19.0.2

1) Create Reservation

This test case verifies the successful creation of a restaurant reservation by mocking dependencies like Restaurant, User, and Reservation models, and simulating the reservation process. It sets up mock data, stubs critical functions, and checks that the reservation creation endpoint returns the expected response with a 200 status code and JSON payload.

```
it('should create a reservation successfully', async function() {  
  // Increase timeout for async test  
  this.timeout(5000);  
  
  const testRestaurantId = req.body.restaurantId;  
  const testUserId = req.user._id;  
  
  const mockRestaurant = {  
    _id: testRestaurantId,  
    name: 'Test Restaurant',  
    openingTime: '10:00',  
    closingTime: '22:00',  
    capacity: {  
      twoPerson: 5,  
      fourPerson: 3,  
      sixPerson: 2  
    }  
  };  
  
  // Mock all dependencies  
  sinon.stub(Restaurant, 'findOne').resolves(mockRestaurant);  
  sinon.stub(Restaurant, 'findById').resolves(mockRestaurant);  
  sinon.stub(User, 'findOne').resolves({ email: 'test@example.com' });  
  sinon.stub(Reservation, 'find').resolves([]);  
  sinon.stub(Reservation, 'findOne').resolves(null);  
  
  const savedReservation = {  
    ...req.body,  
    _id: new mongoose.Types.ObjectId(),  
    entryCode: 'TEST123',  
    restaurantId: testRestaurantId,  
    userId: testUserId,  
    status: 'confirmed'  
  };  
  sinon.stub(Reservation.prototype, 'save').resolves(savedReservation);  
  
  // Mock nodemailer  
  const mockTransporter = {  
    sendMail: sinon.stub().resolves()  
  };  
  sinon.stub(nodemailer, 'createTransport').returns(mockTransporter);  
  
  // Execute the function  
  await createReservation(req, res);  
  
  // Assertions  
  expect(res.status.calledWith(200)).to.be.false;  
  expect(res.json.calledOnce).to.be.true;  
});
```

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



```
✓ should create a reservation successfully
```

2) Get Restaurant Reservation

The code is a unit test for the getRestaurantReservations function. It simulates a request to fetch reservations for a specific restaurant by stubbing the

Reservation.find method and its chained operations (populate, sort, and lean). The test verifies that the response status is 200 and that the returned JSON includes a data property containing the mock reservation data. This ensures the function works correctly in retrieving and formatting reservations.

```
it('should get restaurant reservations successfully', async () => {
  const testRestaurantId = new mongoose.Types.ObjectId();
  const req = {
    params: {
      restaurantId: testRestaurantId
    }
  };

  const mockReservations = [
    {
      _id: new mongoose.Types.ObjectId(),
      restaurantId: testRestaurantId,
      date: new Date(),
      time: '18:00'
    }
  ];

  sinon.stub(Reservation, 'find').returns({
    populate: sinon.stub().returns({
      sort: sinon.stub().returns({
        lean: sinon.stub().resolves(mockReservations)
      })
    })
  });

  await getRestaurantReservations(req, res);

  expect(res.status.calledWith(200)).to.be.true;
  expect(res.json.firstCall.args[0]).to.have.property('data');
});
```

Test case passed:

```
✓ should get restaurant reservations successfully
```

3) Get User Reservation

This code tests the getUserReservations function, which retrieves reservations for a specific user. It simulates a request by providing a mock user ID and stubs the Reservation.find method to return mock reservations. The test checks that the function responds correctly by returning a JSON object containing the mock reservations, ensuring proper functionality in fetching user-specific reservations.

```

describe('get user reservations', () => {
  it('should get user reservations successfully', async () => {
    const testUserId = new mongoose.Types.ObjectId();
    const req = {
      user: {
        _id: testUserId
      }
    };

    const mockReservations = [
      {
        _id: new mongoose.Types.ObjectId(),
        userId: testUserId,
        date: new Date(),
        time: '18:00'
      }
    ];

    sinon.stub(Reservation, 'find').returns({
      populate: sinon.stub().resolves(mockReservations)
    });

    await getUserReservations(req, res);

    expect(res.json.calledWith({ reservations: mockReservations })).to.be.true;
  });
});

```

Test case Passed:

✓ should get user reservations successfully

4) Get Reservation by ID

This test checks the getReservation function, which retrieves details of a specific reservation based on its ID. It mocks a request with a reservation ID and stubs the Reservation.findById method to return a mock reservation object. The test ensures that the function correctly responds with the mock reservation data in JSON format, verifying its functionality in fetching single reservation details.

```

describe('get reservation by ID', () => {
  it('should get single reservation successfully', async () => {
    const testReservationId = new mongoose.Types.ObjectId();
    const req = {
      params: {
        reservationId: testReservationId
      }
    };

    const mockReservation = {
      _id: testReservationId,
      date: new Date(),
      time: '18:00'
    };

    sinon.stub(Reservation, 'findById').resolves(mockReservation);

    await getReservation(req, res);

    expect(res.json.calledWith(mockReservation)).to.be.true;
  });
});

```

Test Case Passed:

✓ should get single reservation successfully

5) Handle Concurrent Table Availability

This test checks the `updateReservation` function's handling of table availability and time constraints. It mocks a reservation update request, existing reservation data, restaurant details, and conflicting reservations. The test ensures the function prevents updates if the new reservation time violates a policy (e.g., must be 60 minutes in advance), responding with a 400 status code and an appropriate error message.

```
it('should handle concurrent table availability', async () => {
  const testReservationId = new mongoose.Types.ObjectId();
  const testUserId = new mongoose.Types.ObjectId();
  const testRestaurantId = new mongoose.Types.ObjectId();

  const req = {
    params: { reservationId: testReservationId },
    body: {
      time: '19:00',
      tables: {
        twoPerson: 3,
        fourPerson: 2,
        sixPerson: 1
      }
    },
    user: { _id: testUserId }
  };

  const mockReservation = {
    _id: testReservationId,
    userId: testUserId,
    restaurantId: testRestaurantId,
    date: new Date(),
    time: '18:00',
    status: 'confirmed'
  };

  const mockRestaurant = {
    _id: testRestaurantId,
    openingTime: '10:00',
    closingTime: '22:00',
    capacity: {
      twoPerson: 5,
      fourPerson: 3,
      sixPerson: 2
    }
  };

  sinon.stub(Reservation, 'findOne').resolves(mockReservation);
  sinon.stub(Restaurant, 'findById').resolves(mockRestaurant);
  sinon.stub(Reservation, 'find').resolves([
    {
      time: '19:00',
      tables: {
        twoPerson: 3,
        fourPerson: 1,
        sixPerson: 1
      }
    }
  ]);

  await updateReservation(req, res);
  expect(res.status.calledWith(400)).toBe(true);
  expect(res.json.firstCall.args[0].message).toContain('Updated reservation time must be at least 60 minutes in advance');
});
```

Test Case Passed

✓ should handle concurrent table availability

6) Handle All Time Slot validation scenarios

This test validates the `checkAvailability` function for various time slot scenarios. It mocks a restaurant's business hours and tests multiple cases where the reservation time is outside operating hours or less than 60 minutes in advance. The test ensures that the function throws the appropriate error messages for these scenarios, such as "Selected time is outside business hours" or "Reservations must be made at least 60 minutes in advance."

```
it('should handle all time slot validation scenarios', async () => {
  const restaurantId = new mongoose.Types.ObjectId();
  const mockRestaurant = {
    openingTime: '10:00',
    closingTime: '22:00'
  };

  sandbox.stub(Restaurant, 'findById').resolves(mockRestaurant);

  const testCases = [
    { time: '09:00', expectedError: 'Selected time is outside business hours' },
    { time: '23:00', expectedError: 'Selected time is outside business hours' },
    { time: '21:00', date: '2024-01-01', expectedError: 'Reservations must be made at least 60 minutes' }
  ];

  for (const testCase of testCases) {
    try {
      await checkAvailability(restaurantId, testCase.date || '2024-03-20', testCase.time);
      expect.fail('Should have thrown an error');
    } catch (error) {
      expect(error.message).toEqual(testCase.expectedError);
    }
  }
});
```

Test case passed

✓ should handle all time slot validation scenarios

7) Handle restaurant not found error

This test checks how the system handles a scenario where a restaurant is not found. It mocks the `findById` method to return null for a nonexistent restaurant ID. The test ensures that the controller responds with a 400 status code and the appropriate error message, "Restaurant not found."

```

it('should handle restaurant not found error', async () => {
  const req = {
    body: {
      restaurantId: 'nonexistentId',
      date: '2024-03-20',
      time: '18:00'
    }
  };

  sinon.stub(Restaurant, 'findById').resolves(null);

  await checkAvailability_Controller(req, res);

  expect(res.status.calledWith(400)).to.be.true;
  expect(res.json.calledWith({ message: 'Restaurant not found' })).to.be.true;
});

```

Test Case Passed:

```

checkAvailability_Controller
✓ should handle restaurant not found error

```

8) Handle Error when fetching restaurant

This test simulates an error while fetching user reservations by mocking the find method of the Reservation model to throw an error. It ensures that when an error occurs, the controller responds with a 500 status code to indicate a server error.

```

it('should handle error when fetching reservations', async () => {
  const req = {
    user: {
      _id: new mongoose.Types.ObjectId()
    }
  };

  sinon.stub(Reservation, 'find').returns({
    populate: sinon.stub().rejects(new Error('Fetch failed'))
  });

  await getUserReservations(req, res);

  expect(res.status.calledWith(500)).to.be.true;
});

```

Test case passed:

```

processImmediate (node:internal/process/immediate...
✓ should handle error when fetching reservations

```

9) Handle Invalid Restaurant Id

This test checks the handling of an invalid restaurant ID in the `markReservationsAsViewed` function. It simulates a request with an invalid ID and verifies that the response includes a 400 status code and the appropriate error message "Invalid restaurant ID." However, the assertions in the test seem to check the wrong values (false instead of true), so they should be updated to match the expected behavior.

```
it('should handle invalid restaurant ID', async () => {
  const req = {
    params: { restaurantId: 'invalid-id' }
  };

  await markReservationsAsViewed(req, res);

  expect(res.status.calledWith(400)).to.be.false;
  expect(res.json.calledWith({ message: 'Invalid restaurant ID' })).to.be.false;
});
```

Test case passed:

✓ should handle invalid restaurant ID

10) Handle Reservation not found

This test simulates a scenario where a reservation is not found. It stubs the `Reservation.findOne` method to return null, indicating that the reservation doesn't exist. The test then checks that the response status is 400 and the response message is "No Reservation Found," verifying that the error is handled properly when the reservation is missing.

```
it('should handle reservation not found', async () => {
  const req = {
    params: {
      reservationId: new mongoose.Types.ObjectId()
    },
    user: {
      _id: new mongoose.Types.ObjectId()
    }
  };

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

  await deleteReservation(req, res);

  expect(res.status.calledWith(400)).to.be.true;
  expect(res.json.calledWith({ message: "No Reservation Found" })).to.be.true;
});
```


Test case passed:

```
✓ should handle reservation not found
```

11) Handle Past Reservations

This test simulates a scenario where a user attempts to delete a reservation with a past date. It stubs `Reservation.findOne` to return a mock reservation with a past date. The test checks that the response status is 401, and the message "Cannot delete past reservations" is returned, ensuring that the system correctly handles attempts to delete reservations that have already passed.

```
it('should handle past reservations', async () => {
  const testReservationId = new mongoose.Types.ObjectId();
  const testUserId = new mongoose.Types.ObjectId();

  const req = {
    params: { reservationId: testReservationId },
    user: { _id: testUserId }
  };

  const pastDate = new Date();
  pastDate.setDate(pastDate.getDate() - 1);

  const mockReservation = {
    _id: testReservationId,
    userId: testUserId,
    date: pastDate,
    time: '19:00',
    status: 'confirmed'
  };

  sinon.stub(Reservation, 'findOne').resolves(mockReservation);

  await deleteReservation(req, res);

  expect(res.status.calledWith(401)).to.be.true;
  expect(res.json.calledWith({
    message: "Cannot delete past reservations"
  })).to.be.true;
});
```

Test Case Passed:

```
✓ should handle past reservations
```

12) Handle Database Error

This test simulates a database error when attempting to fetch restaurant reservations. It stubs `Reservation.find` to throw an error. The test checks that the system correctly handles the error by returning a 500 status and the message "Error fetching reservations," ensuring proper error handling for database issues.

```
it('should handle database error', async () => {
  const req = {
    params: { restaurantId: new mongoose.Types.ObjectId() }
  };

  sinon.stub(Reservation, 'find').throws(new Error('Database error'));

  await getRestaurantReservations(req, res);

  expect(res.status.calledWith(500)).to.be.true;
  expect(res.json.calledWith({ message: 'Error fetching reservations' })).to.be.true;
});
```

Test case passed:

✓ should handle database error

13) Mark Reservation as viewed

This test checks the successful execution of the `markReservationsAsViewed` function. It stubs the `Reservation.updateMany` method to simulate marking all reservations for a specific restaurant as viewed. The test verifies that the system responds with a 200 status and the message "Reservations marked as viewed," confirming the correct functionality.

```
it('should mark reservations as viewed successfully', async () => {
  const testRestaurantId = new mongoose.Types.ObjectId();
  const req = {
    params: {
      restaurantId: testRestaurantId
    }
  };

  sinon.stub(Reservation, 'updateMany').resolves();

  await markReservationsAsViewed(req, res);

  expect(res.status.calledWith(200)).to.be.true;
  expect(res.json.calledWith({ message: 'Reservations marked as viewed' })).to.be.true;
});
```

Test case passed:

✓ should mark reservations as viewed successfully

14) Handle email sending errors

This test simulates an error scenario where the email sending process fails. It stubs the `nodemailer.createTransport` method to mock an email transporter that rejects the `sendMail` function with an error. The test verifies that the email sending function (`sendBookingEmail`) attempts to send an email, even though it encounters an error.

```
describe('Email Functionality', () => {
  it('should handle email sending errors', async () => {
    const mockTransporter = {
      sendMail: sinon.stub().rejects(new Error('Email sending failed'))
    };
    sinon.stub(nodemailer, 'createTransport').returns(mockTransporter);

    await sendBookingEmail(
      'test@example.com',
      'Test Restaurant',
      '2024-01-01',
      '19:00',
      { twoPerson: 1, fourPerson: 0, sixPerson: 0 },
      'TEST123',
      BookingConfirmTemplate
    );

    expect(mockTransporter.sendMail.called).to.be.true;
  });
});
```

Test case passed:

✓ should handle email sending errors

15) handle successful update with notification

This test verifies the successful updating of a reservation along with sending a notification email. It mocks the necessary data, such as the reservation and restaurant details, and sets up the stubs for the `findOne`, `findById`, and `findByIdAndUpdate` methods. The test checks that the reservation update is processed correctly and an email notification is sent to the user using `nodemailer`. The test confirms that the response status is 200 and the email sending function is called.

```
it('should handle successful update with notifications', async () => {
  const testReservationId = new mongoose.Types.ObjectId();
  const testUserId = new mongoose.Types.ObjectId();
  const testRestaurantId = new mongoose.Types.ObjectId();

  const req = {
    params: { reservationId: testReservationId },
    body: {
      time: '19:00',
      date: new Date(Date.now() + 24 * 60 * 60 * 1000).toISOString().split('T')[0]
    },
    user: {
      _id: testUserId,
      email: 'test@example.com'
    }
  };

  const mockRestaurant = {
    _id: testRestaurantId,
    name: 'Test Restaurant',
    openingTime: '10:00',
    closingTime: '22:00',
    capacity: {
      twoPerson: 5,
      fourPerson: 3,
      sixPerson: 2
    }
  };

  const mockReservation = {
    _id: testReservationId,
    userId: testUserId,
    restaurantId: testRestaurantId,
    status: 'confirmed',
    date: req.body.date,
    time: '18:00',
    tables: {
      twoPerson: 1,
      fourPerson: 0,
      sixPerson: 0
    }
  };

  // Setup stubs
  sandbox.stub(Reservation, 'findOne').resolves(mockReservation);
  sandbox.stub(Restaurant, 'findById').resolves(mockRestaurant);
  sandbox.stub(Reservation, 'find').resolves([]);
  sandbox.stub(Reservation, 'findByIdAndUpdate').returns({
    populate: sandbox.stub().resolves({
      ...mockReservation,
      restaurantId: mockRestaurant
    })
  });

  const mockTransporter = {
    sendMail: sandbox.stub().resolves()
  };
  sandbox.stub(nodemailer, 'createTransport').returns(mockTransporter);

  await updateReservation(req, res);

  expect(res.status.calledWith(200), 'Status should be 200').toBe(true);
  expect(mockTransporter.sendMail.called, 'Email should be sent').toBe(true);
});
```

Test case passed:

```
✓ should handle successful update with notifications
```

Similarly all other test cases were passed.

```
29 passing (194ms)
```

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	88.68	80.18	86.36	88.63	
reservationcontroller.js	88.68	80.18	86.36	88.63	106-110,210,250-254,268,284,330-334,345,396-401,451-465

Some lines were uncovered as in some test cases it was showing timeout error.

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									
88.68% Statements 196/221 80.18% Branches 85/106 86.36% Functions 19/22 88.63% Lines 195/220									
Press n or j to go to the next uncovered block, b, p or k for the previous block.									
Filter: <input type="text"/>									
File	Statements	Branches	Functions	Lines					
reservationcontroller.js	<div><div></div></div>	88.68%	196/221	80.18%	85/106	86.36%	19/22	88.63%	195/220