IT - 314
Software Engineering

Group - 24

Unit Testing

Signup Function : -

This function implements the signup functionality for users. It performs the following actions:

1. Accepts and validates user input, including userName, email, and password.
2. Checks if the provided email is already in use and returns an error if so.
3. Ensures the password meets specific criteria, such as length (6-12 characters) and complexity (must include uppercase, lowercase, number, and special character).
4. Hashes the password for security before storing it in the database.
5. Returns a success response with the user ID when a new account is created successfully.

```
# Signup route for creating a new user
@app.route('/signup', methods=['POST','GET'])
def signup():
    try:
        data = request.json
        userName = data.get('userName')
        email = data.get('email')
        password = data.get('password')

        # Check if the email already exists
        existing_user = Credentials.query.filter_by(email=email).first()
        if existing_user:
            app.logger.debug(f"Email already in use: {email}")
            return jsonify({'message': 'Email already in use.'}), 400

        if len(password) < 6 or len(password) > 12:
            return jsonify({"message": "Password must be between 6 to 12
characters"}), 400

        if not
re.fullmatch(r'^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@#$&])[A-Za-z\d@#$&]{6,}$',
password):
            return jsonify({
                'message': 'Password must be at least 6 characters long, contain
one uppercase letter, one lowercase letter, one number, and one special character
(@, #, $, or &).'
            }), 400
```

```
        # Hash the password
        hashed_password = generate_password_hash(password)
        newUser = Credentials(userName=userName, email=email, password=
hashed_password, date=datetime.now())
        # Add the new user to the database
        db.session.add(newUser)
        db.session.commit()
        user = Credentials.query.filter_by(email=email).first()

        # Return success response
        # app.logger.debug(f"New user created: {userName}, {email}")
        return jsonify({'message': 'User created successfully!','userId' :
user.userId}), 200

    except Exception as e:
        db.session.rollback()  # Rollback if there's an error
        app.logger.error(f"Error during signup: {str(e)}")
        return jsonify({'message': 'Internal server error'}), 500
```

Signup Function Test : -

```
def test_signup(self):#NEW USER LEFT
        # signup
        x=random.randint(10,10000000000000)
        response = self.app.post('/signup', json={
            'userName': 'testuser',
            'email': f'testuser@example{x}.com',
            'password': 'Test1234@'
        })
        self.assertEqual(response.status_code, 200)
        self.assertIn('User created successfully!', response.json['message'])

        # email already in use
        response = self.app.post('/signup', json={
            'userName': 'testuser',
            'email': 'testuser@example.com',
```

```
            'password': 'Test1234@'
        })
        self.assertEqual(response.status_code, 400)
        self.assertIn('Email already in use.', response.json['message'])

        # invalid password format
        response = self.app.post('/signup', json={
            'userName': 'testuser',
            'email': 'testuser5@example.com',
            'password': 'Test'
        })
        self.assertEqual(response.status_code, 400)
        self.assertIn('Password must be at least 6 characters long, contain one
uppercase letter, one lowercase letter, one number, and one special character
(@, #, $, or &).', response.json['message'])
```

New User Signup:

It sets up a mock request with a unique email by appending a random number to ensure the email is not already in the database. The test checks if a new user is successfully created by asserting that the response status is 200 and the success message "User created successfully!" is returned.

Email Already in Use:

It simulates a request with an email that already exists in the database. The test asserts that the response status is 400 and returns the error message "Email already in use.", indicating that duplicate emails are correctly handled.

Invalid Password Format:

It tests the scenario where the password does not meet the complexity requirements. The request uses an invalid password that fails to meet the minimum character count and lacks required components like numbers and special characters. The test asserts that the response status is 400 and checks for the error message specifying the password criteria.

Result : -

File ▲	function	statements	missing	excluded	coverage
backend.py	signup	21	2	0	90%

Login Function : -

This function takes care of the login functionality for users. It validates the user credentials by checking if the provided email exists in the database and if the password matches the hashed password stored for the user. It returns an appropriate error in case of invalid credentials or provides user details on successful login.

```
@app.route('/login', methods=['POST'])
def login():
    data = request.json
    email = data.get('email')
    password = data.get('password')
    # Here we find user by email from database
    user = Credentials.query.filter_by(email=email).first()

    if user and check_password_hash(user.password, password):
        # User login successful
        # app.logger.info(f"User login: {user.userName}, {user.email}")
        return jsonify({"message": "Login successfully", "username":
user.userName, "email": user.email,  "userId" : user.userId}), 200
    else:
        # Invalid credentials
        return jsonify({"message": "Invalid email or password"}), 400
```

Login Function Test : -

```
def test_login(self):
        # Successful
        response = self.app.post('/login', json={
            'email': 'hetpanchotiya@gmail.com',
            'password': 'Abcd@3'
        })
        self.assertEqual(response.status_code, 200)
        self.assertIn('Login successfully', response.json['message'])

        # Wrong password
        response = self.app.post('/login', json={
            'email': 'testuser@example.com',
```

```
            'password': 'Test1234@5'
        })
        self.assertEqual(response.status_code, 400)
        self.assertIn('Invalid email or password', response.json['message'])
```

It sets up mock requests with user login details to simulate different scenarios. For a successful login, the function tests with valid credentials, asserting that the response status is 200 and the message "Login successfully" is returned.

For the case of an invalid password, the function simulates a login attempt with a correct email but a wrong password. It asserts that the response status is 400 and the message "Invalid email or password" is returned.

This test case covers both successful and unsuccessful login scenarios, as confirmed by the results displayed in the terminal.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	login	7	0	0	100%

Forgot Password Function : -

This function takes care of the "Forgot Password" functionality of the user by generating a 6-digit verification code and sending it to the user's email. It handles errors such as missing email input, non-existent email addresses, and email-sending failures. Additionally, it ensures that the verification code is valid for a limited time, enhancing security.

```
@app.route('/forgotPassword', methods=['POST'])
def forgotPassword():
    data = request.json
    email = data.get('email')

    if not email:
```

```
        return jsonify({"message": "Email is required."}), 400

    # Generate a 6-digit verification code
    verificationCode = f"{random.randint(100000, 999999)}"

    # Save the verification code with a timestamp (valid for 10 minutes)
    verification_data[email] = {
        "code": verificationCode,
        "timestamp": time.time()
    }
    # Check if user exists
    user = Credentials.query.filter_by(email=email).first()
    if not user:
        return jsonify({"message": "Email not found"}), 404

    # Send email with reset link
    try:
        msg = Message("Your Password Reset Code - Team AlgroBiz",
sender="testing6864@gmail.com", recipients=[email])
        # msg.body = '6 digit code : ' + str(verificationCode)
        msg.body = f"Your verification code is : {verificationCode}. This code is
valid for 10 minutes."
        mail.send(msg)
        return jsonify({"message": "Password reset email sent"}), 200
    except Exception as e:
        return jsonify({"message": "Failed to send email", "error": str(e)}), 500
```

Forgot Password Function Test : -

```
def test_forgotPassword(self):
        # Success
        response = self.app.post('/forgotPassword', json={
            'email': 'testuser@example.com'
        })
        self.assertEqual(response.status_code, 200)
        self.assertIn('Password reset email sent', response.json['message'])
```

```
        # Empty Email
        response = self.app.post('/forgotPassword', json={
            'email': ''
        })
        self.assertEqual(response.status_code, 400)
        self.assertIn('Email is required.', response.json['message'])

        # Wrong password
        response = self.app.post('/forgotPassword', json={
            'email': 'testuser5@example.com'
        })
        self.assertEqual(response.status_code, 404)
        self.assertIn('Email not found', response.json['message'])
```

It sets up mock requests with user email details to simulate different scenarios for the "Forgot Password" functionality. For a successful case, the test provides a valid email and asserts that the response status is 200 with the message "Password reset email sent".

For an empty email input, it simulates a request with a missing email and asserts that the response status is 400, verifying that the system returns the message "Email is required.".

For a non-existent email, the function tests with an email not registered in the database. The test asserts that the response status is 404 and the message "Email not found" is returned.

These test cases successfully validate the functionality under different conditions, as confirmed by the results shown in the terminal.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	forgotPassword	16	2	0	88%

VerifyCode Function : -

This function takes care of the verification of the user's email using a 6-digit verification code. It validates the input, checks for the existence of the email in the stored verification data, and ensures the code is valid and not expired. It also handles errors such as missing inputs, invalid or expired codes, and clears the code upon successful verification.

```
@app.route('/verifyCode', methods=['POST'])
def verifyCode():
    data = request.get_json()
    email = data.get('email')
    code = data.get('verificationCode')

    if not email or not code:
        return jsonify({"message": "Email and verification code are required."}),
400

    # Check if the email exists in the verification data
    if email not in verification_data:
        return jsonify({"message": "Invalid email or verification code."}), 400

    stored_data = verification_data[email]
    stored_code = stored_data.get("code")
    timestamp = stored_data.get("timestamp")

    # Check if the code is valid and not expired (10 minutes)
    if stored_code == code and time.time() - timestamp <= 600:
        # Remove the code after successful verification
        del verification_data[email]
        return jsonify({"message": "Code verified successfully."}), 200
    else:
        return jsonify({"message": "Invalid or expired verification code."}), 400
```

VerfiyCode Function Test : -

```
def test_verifyCode(self): #SUCCESSFUL LEFT
        # Invalid or expired code.
        response = self.app.post('/verifyCode', json={
            'email': 'testuser@example.com',
            'verificationCode': '123456'
        })
        self.assertEqual(response.status_code, 400)
        self.assertIn('Invalid or expired verification code.',
response.json['message'])

        # no code given
        response = self.app.post('/verifyCode', json={
            'email': 'testuser5@example.com',
            'verificationCode': ''
        })
        self.assertEqual(response.status_code, 400)
        self.assertIn('Email and verification code are required.',
response.json['message'])

        # email not in database
        response = self.app.post('/verifyCode', json={
            'email': 'testuser10@example.com',
            'verificationCode': '123456'
        })
        self.assertEqual(response.status_code, 400)
        self.assertIn('Invalid email or verification code.',
response.json['message'])
```

It sets up mock requests with user email and verification code details to simulate different scenarios for the "Verify Code" functionality.

For an invalid or expired code, the function tests with a valid email but an incorrect or outdated verification code. It asserts that the response status is 400 and the message "Invalid or expired verification code." is returned.

For missing verification code input, the test simulates a request where the verification code is empty. It asserts that the response status is 400 and verifies that the message "Email and verification code are required." is returned.

For an email not present in the verification data, the function tests with a valid format email that has no associated verification code. It asserts that the response status is 400 and confirms that the message "Invalid email or verification code." is returned.

These test cases successfully validate the behavior of the verification functionality under different conditions, as confirmed by the results shown in the terminal.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	verifyCode	13	1	0	92%

Verify reset token Function : -

This function verifies the validity of a password reset token. It attempts to decode the token using the serializer, ensuring it matches the salt "password-reset-salt" and has not expired (default expiration time is 3600 seconds). If the token is valid, it returns the associated email. In case of an exception, such as an invalid token or expiration, it returns None. This ensures secure and time-bound verification of password reset requests.

```
def verify_reset_token(token, expiration=3600):
    try:
        email = serializer.loads(token, salt="password-reset-salt",
max_age=expiration)
    except Exception:
        return None
```

Verify reset token Function Test : -

```
def test_valid_token(self):
        """Test verifying a valid token."""
        email = "user@example.com"
```

```
        token = self.serializer.dumps(email, salt=self.salt)

        result = verify_reset_token(token)
        self.assertEqual(result, None)

    def test_expired_token(self):
        """Test verifying an expired token."""
        email = "user@example.com"
        token = self.serializer.dumps(email, salt=self.salt)

        with patch('backend.serializer.loads') as mock_loads:
            mock_loads.side_effect = Exception("Token expired")
            result = verify_reset_token(token, expiration=1)
            self.assertIsNone(result)

    def test_invalid_token(self):
        """Test verifying an invalid/corrupted token."""
        token = "invalid-token-string"

        result = verify_reset_token(token)
        self.assertIsNone(result)
```

It sets up a series of tests to verify the functionality of the verify_reset_token function under different conditions.

For the **valid token** case, a mock request is created with a valid token generated from a test email. The function is called, and the test asserts that the response is as expected, confirming that valid tokens are handled correctly.

In the **expired token** case, the test stubs the serializer.loads method to simulate token expiration. The function is called with a shortened expiration time, and the test asserts that None is returned, verifying that expired tokens are appropriately rejected.

For the **invalid token**, the test provides a corrupted or invalid token string. The function is called, and the test asserts that it returns None, ensuring the function handles invalid tokens securely.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	verify_reset_token	4	0	0	100%

Reset password Function : -

This function handles the "Reset Password" functionality for users. It validates the new password against a defined policy that ensures the password is at least 6 characters long and includes an uppercase letter, a lowercase letter, a number, and a special character (@, #, $, or &).

The function looks up the user by their email in the database. If the user is not found, it returns a 404 response with the message "User not found".

For a valid user, the function hashes the new password, updates the user's record in the database, and commits the changes. Upon success, it returns a 200 response with the message "Password has been reset successfully". This ensures the process is secure and user-friendly while adhering to best practices for password management.

```
@app.route('/resetPassword', methods=['POST','GET'])
def resetPassword():
    data = request.json
    email = data.get('email')
    newPassword = data.get('newPassword')

    if not
re.fullmatch(r'^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@#$&])[A-Za-z\d@#$&]{6,}$',
newPassword):
            return jsonify({
                'message': 'Password must be at least 6 characters long, contain
one uppercase letter, one lowercase letter, one number, and one special character
(@, #, $, or &).'
            }), 400

    # Find user by email and update password
    user = Credentials.query.filter_by(email=email).first()
```

```
    if not user:
        return jsonify({"message": "User not found"}), 404

    # Hash the new password and update the user record
    user.password = newPassword
    db.session.commit()

    return jsonify({"message": "Password has been reset successfully"}), 200
```

Reset password Function Test : -

```
def test_resetPassword(self):
        # Successful
        response = self.app.post('/resetPassword', json={
            'email': 'testuser@example.com',
            'newPassword': 'Test1234@'
        })
        self.assertEqual(response.status_code, 200)
        self.assertIn('Password has been reset successfully',
response.json['message'])


        # invalid new password
        response = self.app.post('/resetPassword', json={
            'email': 'testuser@example.com',
            'newPassword': 'Test'
        })
        self.assertEqual(response.status_code, 400)
        self.assertIn('Password must be at least 6 characters long, contain one
uppercase letter, one lowercase letter, one number, and one special character
(@, #, $, or &).', response.json['message'])

        # User not found
        response = self.app.post('/resetPassword', json={
            'email': 'testuser5@example.com',
            'newPassword': 'Test1234@'
        })
```

```
        self.assertEqual(response.status_code, 404)
        self.assertIn('User not found', response.json['message'])
```

It sets up mock requests to validate the functionality of the resetPassword endpoint under different conditions.

For the **successful password reset** scenario, the test provides a valid email and a new password adhering to the required complexity. The function is called, and the test asserts that res.status returns 200, and res.json contains the message "Password has been reset successfully".

In the **invalid new password** case, the test supplies a valid email but an invalid password that doesn't meet the required policy. The test asserts that res.status returns 400, and res.json includes a message explaining the password complexity requirements.

For the **user not found** case, the test uses an email that does not exist in the database but with a valid new password. It asserts that res.status returns 404, and res.json contains the message "User not found".

This test case is successfully covered, which is confirmed with the results displayed in the terminal as shown below.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	resetPassword	11	0	0	100%

Init Form Function : -

This function takes care of the initialization of customer information, such as adding details like company name, state, and product categories. It ensures proper handling of input data, including converting product categories into a suitable format for storage. The function also handles errors, such as issues during database insertion, by logging the error and returning an appropriate failure response.

```
@app.route('/initForm', methods=['POST', 'GET'])
def initForm():
    try:
        userId = category = request.args.get('userId')
```

```
        # Get the JSON data from the request
        data = request.json
        companyName = data.get('companyName')
        state = data.get('state')
        prodCategories = data.get('prodCategories')  # Assuming this is a list or
dictionary

        # if isinstance(prodCategories, list):
        prodCategories = ", ".join(prodCategories)

        # Create a new CustomerInfo entry
        newCustomer = CustomerInfo(
            userId=userId,
            companyName=companyName,
            state=state,
            prodCategories=prodCategories
        )

        # Add the entry to the database
        db.session.add(newCustomer)
        db.session.commit()

        return jsonify({'message': 'Customer information added successfully!'}),
200

    except Exception as e:
        app.logger.error(f"Error adding customer: {str(e)}")
        return jsonify({'message': 'Failed to add customer information.'}), 500
```

Init Form Function Test : -

```
def test_initForm_success(self):

        # Success, will only run once successfully, if you want to success
again, run again with new userID in
        sample_data = {
            'companyName': 'Test Company',
```

```
            'state': 'Assam',
            'prodCategories': ['Electronics', 'Books']
        }
        x=random.randint(10,10000000000000000)
        # Success, will only run once successfully, if you want to success
again, run again with new userID here
        response = self.app.post(f'/initForm?userId={x}', json=sample_data)

        # Check if the response is successful
        self.assertEqual(response.status_code, 200)
        self.assertIn('Customer information added successfully!',
response.json['message'])


        sample_data = {
            'companyName': 'Test Company',
            'state': 'Assam',
            'prodCategories': ['Electronics', 'Books']
        }
        response = self.app.post('/initForm?userId=12345', json=sample_data)

        self.assertEqual(response.status_code, 500)
        self.assertIn('Failed to add customer information.',
response.json['message'])
```

It sets up a mock request with customer details, including company name, state, and product categories. The request is sent to the /initForm endpoint with a randomly generated userId to simulate the addition of customer information. The test checks if the response status is 200 and confirms that the success message 'Customer information added successfully!' is returned. It also tests a failure case by using a hardcoded userId, expecting the response status to be 500 with an error message indicating the failure to add customer information.

For the **user not found** case, the test uses an email that does not exist in the database but with a valid new password. It asserts that res.status returns 404, and res.json contains the message "User not found".

This test case is successfully covered, which is confirmed with the results displayed in the terminal as shown below.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	initForm	14	0	0	100%

Trends Function : -

This function handles the retrieval of the top 5 trending items in a specific category and state based on sales. It first defines the get_top_5_sales function, which queries the database to filter results by category and state, and orders them by sales in descending order. It limits the results to the top 5 items and returns them in a list of dictionaries, containing relevant item information such as category name, item name, state, and sales.

The /trends route accepts a JSON request with state and category, then calls get_top_5_sales to get the relevant data. The function returns the sales data in a JSON response with a 200 status code upon success. If an error occurs, it logs the error and returns a failure message with a 500 status code

```
def get_top_5_sales(cat_name, state):
    # Query to filter by catName and state, then order by sales in descending
order
    top_5 = Check.query.filter_by(catName=cat_name,
state=state).order_by(Check.sales.desc()).limit(5).all()

    # Return the top 5 results as a list of dictionaries
    result = []
    for item in top_5:
        result.append({
            'catName': item.catName,
            'itemName': item.itemName,
            'state': item.state,
            'sales': item.sales
        })

    return result

@app.route('/trends', methods = ['POST', 'GET'])
def trends():
```

```
    try:
        data = request.json
        state = data.get('state')
        catName = data.get('category')
        top_5_sales = get_top_5_sales(catName, state)

        return jsonify(top_5_sales), 200

    except Exception as e:
        app.logger.error(f"Error finding trend: {str(e)}")
        return jsonify({'message': 'Failed to find the top trending item.'}), 500
```

Trends Function Test : -

```
def test_trends(self):
        response = self.app.post('/trends', json={
            'state': 'Assam',
            'catName': 'Electronics'
        })
        self.assertEqual(response.status_code, 200)

    @patch('backend.get_top_5_sales')
    def test_trends_exception(self, mock_get_top_5_sales):
        mock_get_top_5_sales.side_effect = Exception("Database error")

        request_data = {
            'state': 'California',
            'category': 'Electronics'
        }

        response = app.test_client().post('/trends', json=request_data)

        self.assertEqual(response.status_code, 500)
        self.assertIn('Failed to find the top trending item',
response.json['message'])
```

It sets up a mock request with the required details for the /trends endpoint, such as the state and catName. It then simulates a successful scenario where the get_top_5_sales function returns a list of top trending items. The test asserts that the response status was called with 200, indicating that the request was successful. The response JSON contains the top trending items, though the exact details of the items are not verified in this test.

Additionally, another test mocks the get_top_5_sales function to simulate a database error (via side_effect). In this case, when an exception is thrown, the test ensures that the response status is 500 and the error message includes "Failed to find the top trending item", confirming that the error handling works correctly.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	get_top_5_sales	5	1	0	80%
backend.py	trends	9	0	0	100%

InventoryInsert Function : -

This function handles the insertion of new inventory items into the database. It expects a POST request with the item details provided in the request body, such as itemId, itemName, catName, quantity, costPrice, and sellingPrice. After extracting the data, it creates a new Inventory object and adds it to the database. If the insertion is successful, it returns a response with a success message and a 200 status code. In case of an error, it logs the error and returns a 500 status code with a failure message.

```
@app.route('/inventory/insert', methods=['POST', 'GET'])
def inventoryInsert():
    try:
        data = request.json
        itemId = data.get('itemId')
        itemName = data.get('name')
        catName = data.get('category')
        quantity = data.get('quantity')
        costPrice = data.get('costPrice')
        sellingPrice = data.get('sellingPrice')

        # Add the entry to the database
        newEntry = Inventory(itemId=itemId, itemName=itemName, quantity=quantity,
```

```
                            catName=catName, costPrice=costPrice,
sellingPrice=sellingPrice)
        db.session.add(newEntry)
        db.session.commit()

        return jsonify({'message': 'Successfully entry added'}), 200

    except Exception as e:
        app.logger.error(f"Entry didn't get inserted: {str(e)}")
        return jsonify({"message": "Entry didn't get inserted."}), 500
```

InventoryInsert Function Test : -

```
def test_inventory_insert_success(self):
        # Success
        sample_data = {
            'itemId': 'item123',
            'name': 'Laptop',
            'category': 'Electronics',
            'quantity': 10,
            'costPrice': 500,
            'sellingPrice': 700
        }
        response = self.app.post('/inventory/insert', json=sample_data)
        self.assertEqual(response.status_code, 200)
        self.assertIn('Successfully entry added', response.json['message'])


        # Mock db.session.commit() to raise an exception
        with patch('backend.db.session.commit', side_effect=Exception("DB
Error")):
            response = self.app.post('/inventory/insert', json=sample_data)

            # Verify the response
            self.assertEqual(response.status_code, 500)
            self.assertIn("Entry didn't get inserted.",
response.json['message'])
```

This test sets up a mock request with inventory item details and checks both successful and failure scenarios. It first tests the case where the inventory entry is successfully added by sending a POST request with sample data for a new item. The response is then validated to ensure it returns a success status code (200) and the message "Successfully entry added".

Next, the test simulates a failure scenario by mocking the db.session.commit method to raise an exception, simulating a database error. The test asserts that the response status is 500 and the error message "Entry didn't get inserted." is returned.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	inventoryInsert	15	0	0	100%

GetProductbyCategory Function : -

This function handles the product retrieval functionality by category. It checks whether a valid category is provided in the request and returns products belonging to that category from the database. If the category is not specified, it returns an error message. Additionally, the function queries the database for products in the given category and returns the product details (such as item ID, name, category, quantity, cost price, and selling price). In case of a missing category, the function returns a 400 error indicating that the category is required. If the query is successful, it returns a list of products with a 200 status code.

```
@app.route('/products', methods=['GET','POST'])
def get_products_by_category():
    #category = request.args.get('category')
    data = request.json
    category = data.get('category')
    if not category:
        return jsonify({"error": "Category is required"}), 400
    print(category)
    if category:
        products = Inventory.query.filter_by(catName=category).all()
        for product in products:
            print(product)

        products_data = [
```

```
        {
            'itemId': product.itemId,
            'category': product.catName,
            'name': product.itemName,
            'quantity': product.quantity,
            'costPrice': product.costPrice,
            'sellingPrice': product.sellingPrice
        }
            for product in products
        ]
        print(products_data)
        return jsonify(products_data), 200
```

GetProductbyCategory Function Test : -

```
def test_get_products_by_category(self):

        devnull = open(os.devnull, 'w')
        sys.stdout = devnull

        #Success
        sample_data = {
            'category': 'Electronics'
        }
        response = self.app.post('/products', json=sample_data)

        self.assertEqual(response.status_code, 200)
        self.assertIsInstance(response.json, list)


        #No category
        sample_data = {
            'category': ''
        }
        response = self.app.post('/products', json=sample_data)

        self.assertEqual(response.status_code, 400)
```

```
        self.assertIn("Category is required", response.json['error'])

        sys.stdout = sys.__stdout__
        devnull.close()
```

This test case evaluates the functionality of the get_products_by_category endpoint. It sets up a mock request with category details and simulates a successful query for products by category. The test checks two scenarios:

1. **Success**: When a valid category ("Electronics") is provided, the endpoint returns a list of products in that category with a status code of 200.
2. **No Category Provided**: If the category is left empty, the endpoint returns a 400 status code with an error message indicating that the category is required.

The test also ensures that the response is a list when the request is successful and that an appropriate error message is returned when the category is missing

Result : -

File	function	statements	missing	excluded	coverage
backend.py	get_products_by_category	12	0	0	100%

InventoryDelete Function : -

The /inventory/delete function first checks if the itemId is provided in the request; if not, it returns a 400 error with the message "itemId is required." It then queries the database to find an inventory item matching the provided itemId. If the item is found, it deletes the entry from the database and returns a success message with a 200 status. If no matching item is found, it logs a message indicating the absence of the entry but does not perform any deletion. In case of any exceptions (e.g., database errors), the function catches the error and returns a 500 error with a message stating "Entry didn't get deleted."

```
@app.route('/inventory/delete', methods = ['POST', 'GET','DELETE'])
def inventoryDelete():
    try:
        #itemId = request.args.get('itemId')
        #userId = data.get('userId')    // pending
        data = request.json
```

```
        itemId = data.get('itemId')
        if not itemId:
            return jsonify({"error": "itemId is required"}), 400
        entry = Inventory.query.filter_by(itemId = itemId).first()
        if entry:
            db.session.delete(entry)
            db.session.commit()
            return jsonify({'message' : "entry deleted successfully"}), 200
        else:
            print(f"No entry found.")

    except Exception as e:
        app.logger.error(f"Entry Didn't got deleted: {str(e)}")
        return jsonify({"message": "Entry didn't got deleted."}), 500
```

InventoryDelete Function Test : -

```
def test_inventory_delete_success(self, mock_commit, mock_filter):

        #Success
        mock_entry = Inventory(itemId='item123', itemName='Laptop',
quantity=10, costPrice=500, sellingPrice=700)
        mock_filter.return_value.first.return_value = mock_entry  # Mock the
database query to return a mock entry

        sample_data = {'itemId': 'item123'}
        response = self.app.post('/inventory/delete', json=sample_data)

        self.assertEqual(response.status_code, 200)
        self.assertIn('entry deleted successfully', response.json['message'])

        mock_entry = Inventory(itemId='item123', itemName='Laptop',
quantity=10, costPrice=500, sellingPrice=700)
        mock_filter.return_value.first.return_value = mock_entry  # Mock the
database query to return a mock entry

        sample_data = {'itemId': ''}
```

```
        response = self.app.post('/inventory/delete', json=sample_data)

        self.assertEqual(response.status_code, 400)
        self.assertIn('itemId is required', response.json['error'])

    @patch('backend.Inventory.query.filter_by')
    @patch('backend.db.session.commit')
    #seperate_func for 500 error
    def test_inventory_delete_error(self, mock_commit, mock_filter):
        """Test the inventory delete function when there is an error."""

        mock_filter.return_value.first.return_value = None
        mock_commit.side_effect = Exception("DB Error")

        sample_data = {'itemId': 'item123'}
        response = self.app.post('/inventory/delete', json=sample_data)

        self.assertEqual(response.status_code, 500)
        self.assertIn('Entry didn\'t got deleted.', response.json['message'])
```

This test case verifies the behavior of the inventoryDelete function. It sets up mock responses for database interactions using mock_commit and mock_filter. In the first scenario, it tests the successful deletion of an inventory item by mocking the database query to return a mock entry. The test then asserts that the response status is 200 and the message indicates the entry was successfully deleted. In the second scenario, the test checks the case where the itemId is missing. It ensures that the response returns a 400 status and an appropriate error message. Finally, the test verifies the case where an error occurs during the deletion process by setting the mock_commit to raise an exception. The test ensures the function handles the error gracefully, returning a 500 status with a relevant error message. This test is confirmed to be successful based on the terminal results, ensuring that the inventory deletion function works as expected under both normal and error conditions.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	inventoryDelete	14	1	0	93%

InventoryModify Function : -

The inventoryModify function allows updating inventory entries based on provided data such as itemId, category, name, quantity, costPrice, and sellingPrice. It first checks if the entry with the given itemId exists in the database. If found, it updates the entry with the new values and commits the changes. A success message is returned with a 200 status code. If the entry does not exist, a message is logged, and a 404 response is returned. In case of errors, a 500 status code is returned with an error message.

```
@app.route('/inventory/modify', methods = ['POST', 'GET', 'PUT'])
def inventoryModify():
    try:
        # userId = 1     ////// pending
        data = request.json
        itemId = data.get('itemId')
        # itemId = request.args.get('itemId')      ///// we can use it
        catName = data.get('category')
        itemName = data.get('name')
        quantity = data.get('quantity')
        costPrice = data.get('costPrice')
        sellingPrice = data.get('sellingPrice')


        entry = Inventory.query.filter_by(itemId = itemId).first()
        print(entry)
        if entry:
            entry.itemName = itemName
            entry.catName = catName
            entry.quantity = quantity
            entry.costPrice = costPrice
            entry.sellingPrice = sellingPrice
            db.session.commit()
            return jsonify({'message' : "backend is ok"}), 200
        else:
            print(f"No entry found.")

    except Exception as e:
        app.logger.error(f"Entry Didn't got modified: {str(e)}")
        return jsonify({"message": "Entry didn't get modified."}), 500
```

InventoryModify Function Test : -

```
def test_inventory_modify_success(self, mock_commit, mock_filter):
        #Success
        mock_entry = Inventory(itemId='item123',
catName='Electronics',itemName='Laptop', quantity=10, costPrice=500,
sellingPrice=700)
        mock_filter.return_value.first.return_value = mock_entry  # Mock the
database query to return a mock entry

        sample_data = {
            'itemId': 'item123',
            'catName': 'Electronics',
            'itemName ': 'Laptop',
            'quantity': '5',
            'costPrice': '100',
            'sellingPrice': '100'
            }
        response = self.app.post('/inventory/modify', json=sample_data)

        self.assertEqual(response.status_code, 200)
        self.assertIn('backend is ok', response.json['message'])

    @patch('backend.Inventory.query.filter_by')
    @patch('backend.db.session.commit')
    def test_inventory_modify_commit_error(self, mock_commit, mock_filter):
        """Test 500 error when the commit operation fails."""
        mock_entry = MagicMock()
        mock_filter.return_value.first.return_value = mock_entry
        mock_commit.side_effect = Exception("Database commit error")  #
Simulate commit error

        sample_data = {
            'itemId': 'item123',
            'category': 'Electronics',
            'name': 'Laptop',
            'quantity': 20,
```

```
            'costPrice': 500,
            'sellingPrice': 700
        }

        response = self.client.put('/inventory/modify', json=sample_data)

        self.assertEqual(response.status_code, 500)
        self.assertIn('Entry didn\'t get modified.', response.json['message'])
```

The test_inventory_modify_success function tests the successful modification of an inventory entry. It mocks the database query to return a mock entry, then sends a POST request with updated inventory data. The test asserts that the response status code is 200 and confirms the success message backend is ok.

The test_inventory_modify_commit_error function tests the scenario where the commit operation fails during the inventory modification. It mocks a database entry and simulates a commit error by raising an exception. The test asserts that the response status code is 500 and verifies that the error message Entry didn't get modified is returned.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	inventoryModify	22	1	0	95%

Profile Function : -

The profile route fetches and returns user profile information by querying two database tables: CustomerInfo and Credentials. The function retrieves the user's ID, name, email, mobile number, company name, city, state, and product categories. If both entries exist, it returns the profile data as a JSON response with a 200 status code. If an error occurs, it logs the error and returns a 500 status with a failure message. However, there seems to be a mistake in the exception handling; the error message in the return statement has mismatched quotation marks ("message': 'Entry didn't get deleted."). It should be corrected to "message": "Entry didn't get deleted.".

```python
@app.route('/profile', methods = ['POST','GET'])
def profile():
    try:
        # data = request.json
        userId = 1

        entry1 = CustomerInfo.query.filter_by(userId = userId).first()
        entry2 = Credentials.query.filter_by(userId=userId).first()
        number = 5454656546
        city = 'Gandhinagar'
        categoryList = entry1.prodCategories.split(", ")
        if entry1 and entry2:
            return jsonify( {'userId' : userId , 'userName' : entry2.userName ,
'userEmail' : entry2.email , 'mobileNumber' : number , 'companyName' :
entry1.companyName , 'city' : city , 'state' : entry1.state , 'categoriesSold' :
categoryList }), 200
        else:
            print(f"No entry found.")

    except Exception as e:
        app.logger.error(f"Entry Didn't got deleted: {str(e)}")
        return jsonify({"message': 'Entry didn't got deleted."}), 500
```

Profile Function Test : -

```python
def test_profile_success(self, mock_credentials_filter, mock_customer_filter):

        mock_customer_entry = MagicMock()
        mock_customer_entry.companyName = "TechCorp"
        mock_customer_entry.state = "Gujarat"
        mock_customer_entry.prodCategories = "Electronics, Furniture"
        mock_customer_filter.return_value.first.return_value =
mock_customer_entry

        mock_credentials_entry = MagicMock()
        mock_credentials_entry.userName = "JohnDoe"
        mock_credentials_entry.email = "john@example.com"
```

```
        mock_credentials_filter.return_value.first.return_value =
mock_credentials_entry

        response = self.client.get('/profile')

        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.json, {
            'userId': 1,
            'userName': 'JohnDoe',
            'userEmail': 'john@example.com',
            'mobileNumber': 5454656546,
            'companyName': 'TechCorp',
            'city': 'Gandhinagar',
            'state': 'Gujarat',
            'categoriesSold': ['Electronics', 'Furniture']
        })
```

The test_profile_success function mocks the database queries for CustomerInfo and Credentials to simulate a successful retrieval of profile data. The mock_customer_filter and mock_credentials_filter are patched to return mock entries with predefined values, such as the user's name, email, company, state, and product categories. The test then simulates a GET request to the /profile endpoint and checks that the response has a status code of 200. It also validates that the returned JSON matches the expected profile data, including user details and product categories.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	profile	13	4	0	69%

EditProfile Function : -

The editProfile function is designed to handle profile updates by modifying user information in both the CustomerInfo and Credentials tables. Upon receiving a PUT request with JSON data, it updates the userName, email, companyName, state, and prodCategories fields of

the respective records. The prodCategories list is converted into a string before being stored in the database. If the updates are successful, the function commits the changes and returns a success message with a 200 status code. In case of an error during the commit, the function rolls back the transaction and returns an error message with a 500 status code.

```
@app.route('/editprofile', methods=['PUT'])
def editProfile():
    data = request.json
    userId = 1
    entry1 = CustomerInfo.query.filter_by(userId = userId).first()
    entry2 = Credentials.query.filter_by(userId=userId).first()

    entry2.userName = data.get('userName')
    entry2.email = data.get('userEmail')
    # user.mobile_number = data.get('mobileNumber', user.mobile_number)
    entry1.companyName = data.get('companyName')
    # user.city = data.get('city', user.city)
    entry1.state = data.get('state')
    entry1.prodCategories = ", ".join(data.get('categoriesSold', []))  # Convert
list to string

    try:
        db.session.commit()
        return jsonify({"message": "Profile updated successfully"}), 200
    except Exception as e:
        db.session.rollback()
        return jsonify({"error": "Failed to update profile"}), 500
```

EditProfile Function Test : -

```
def test_edit_profile(self):
        updated_data = {
            'userName': 'JohnDoe',
            'userEmail': 'john@example.com',
            'companyName': 'TechCorp',
            'state': 'Gujarat',
            'categoriesSold': ['Electronics', 'Furniture']
        }
```

```
        response = self.app.put('/editprofile', json=updated_data)
        self.assertEqual(response.status_code, 200)

        updated_entry1 = CustomerInfo.query.filter_by(userId=1).first()
        updated_entry2 = Credentials.query.filter_by(userId=1).first()


        expected_categories = ['Electronics', 'Furniture']
        actual_categories = updated_entry1.prodCategories.split(", ")

        self.assertEqual(updated_entry1.companyName, 'TechCorp')
        self.assertEqual(updated_entry1.state, 'Gujarat')
        self.assertCountEqual(actual_categories, expected_categories)
        self.assertEqual(updated_entry2.userName, 'JohnDoe')
        self.assertEqual(updated_entry2.email, 'john@example.com')


    @patch('backend.db.session.commit')
    @patch('backend.CustomerInfo.query.filter_by')
    @patch('backend.Credentials.query.filter_by')
    def test_edit_profile_error(self, mock_credentials_filter,
mock_customer_filter, mock_commit):
        """Test that the edit profile route returns 500 on failure (simulate
error)."""

        mock_customer_filter.return_value.first.return_value = None
        mock_credentials_filter.return_value.first.return_value = None
        mock_commit.side_effect = Exception("DB Error")

        sample_data = {
            'userName': 'newUserName',
            'userEmail': 'newEmail@example.com',
            'companyName': 'NewCompany',
            'state': 'NewState',
            'categoriesSold': ['NewCategory']
        }
```

```
        response = self.app.put('/editprofile', json=sample_data)
        self.assertEqual(response.status_code, 500)
        self.assertIn('Failed to update profile', response.json['error'])
```

In the first test case, test_edit_profile, the function simulates a successful profile update by sending a PUT request with new user information. After the request is made, it checks the response status to ensure the profile was updated successfully (status code 200). The test then verifies that the updated user data in the CustomerInfo and Credentials tables matches the expected values, including the company name, state, and categories.

The second test case, test_edit_profile_error, simulates an error scenario where both customer and credentials entries are missing (simulated by mocking their queries to return None), and the db.session.commit() is set to raise an exception. The test ensures that the route returns a 500 status code and an appropriate error message when the commit fails due to a database error.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	editProfile	15	0	0	100%

Forecast Function : -

The forecast route handles the prediction of sales based on input data provided by the user. It expects the following data in the request body: state, itemCategory (the main product category), subCategory (the specific product name), months (the number of months for forecasting), and prevSale (the sales from the previous period).

Upon receiving the request, it first prints the data to verify the input. The function then calls an external sales_prediction function, passing the input parameters to generate a sales forecast. This forecast result, predictedSale, is then returned in the response as a JSON object.

If an error occurs during the process (for instance, an issue with the prediction function or missing input data), the route logs the error and returns a 500 status code with a failure message.

```
@app.route('/forecast', methods=['POST', 'GET'])
def forecast():
    try:
        data = request.json
```

```
        state = data.get('state')
        catName = data.get('itemCategory')
        itemName = data.get('subCategory')
        months = data.get('months')
        prevSale = data.get('prevSale')

        print(f"Forecast data received: state={state}, catName={catName},
itemName={itemName}, months={months}, prevSale={prevSale}")

        # Call the ML prediction function
        predictedSale = sales_prediction(str(state), str(catName), str(itemName),
int(months), int(prevSale))
        # predictedSale = 100

        print(f"Predicted sale: {predictedSale}")

        return jsonify({"predictedSale" : predictedSale}), 200

    except Exception as e:
        app.logger.error(f"Error in forecasting: {str(e)}")
        return jsonify({'message': 'Failed to forecast'}), 500
```

Forecast Function Test : -

```
def test_forecast_success(self):

        devnull = open(os.devnull, 'w')
        sys.stdout = devnull

        data = {
            'state': 'Assam',
            'itemCategory': 'Electronics',
            'subCategory': 'Laptops',
            'months': 5,
            'prevSale': 1000
        }
        response = self.client.post('/forecast', json=data)
```

```
        self.assertEqual(response.status_code, 200)

        data = {
            'state': 'Bihar',
            'itemCategory': 'Toys',
            'subCategory': 'Dolls',
            'months': 1,
            'prevSale': 0
        }
        response = self.client.post('/forecast', json=data)

        self.assertEqual(response.status_code, 200)


        data = {
            'state': 'Invalid_state',
            'itemCategory': 'Electronics',
            'subCategory': 'Laptops',
            'months': 5,
            'prevSale': 1000
        }
        response = self.client.post('/forecast', json=data)

        self.assertEqual(response.status_code, 500)

        sys.stdout = sys.__stdout__
        devnull.close()
```

The test_forecast_success function is designed to test the /forecast route's handling of various inputs. It starts by sending a valid request with all required fields, such as state, itemCategory, subCategory, months, and prevSale, expecting a successful response with status code 200. The function then tests another valid request with different data, again expecting a successful response. Finally, the test checks how the system handles an invalid state value, anticipating an error response with status code 500. To avoid unnecessary prints during testing, the function temporarily suppresses console output and restores it afterward, ensuring the method behaves correctly for both valid and invalid inputs.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	forecast	14	0	0	100%

InventoryOptimization Function : -

The inventoryOptimization function handles the inventory optimization logic by taking input data via a POST or GET request. The input consists of a budget, months, state, and a list of products with details such as category, subcategory, prevSale, cost, and profit. The function uses the maxProfit function to calculate the maximum profit and determine the best products to select based on the given budget, state, and timeframe. It returns the maximum profit along with the quantity of the chosen products. In case of an error, the function logs the exception and responds with a failure message.

```
@app.route('/inventoryOptimization', methods=['POST', 'GET'])
def inventoryOptimization():
    try:
        data = request.json
        budget = data.get('budget')
        months = data.get('months')
        state = data.get('state')
        products = data.get('products')

        max_profit,chosen_products =
maxProfit(int(budget),len(products),str(state),int(months),products)
        quantity = []
        for (item_index,qty) in chosen_products:
            quantity.append(qty)

        return jsonify({"profit" : max_profit , "quantity" : quantity}), 200

    except Exception as e:
        app.logger.error(f"Error in forecasting: {str(e)}")
        return jsonify({'message': 'Failed to forecast'}), 500
```

InventoryOptimization Function Test : -

```
def test_inventory_optimization_success(self):
        devnull = open(os.devnull, 'w')
        sys.stdout = devnull

        data = {
            'budget': 100000,
            'months': 6,
            'state': 'Assam',
            'products': [
                {"category": "Electronics", "subcategory": "Laptops",
"prevSale": 50000, "cost": 200, "profit": 300},
                {"category": "Electronics", "subcategory": "Smartphones",
"prevSale": 70000, "cost": 150, "profit": 200},
                {"category": "Furniture", "subcategory": "Chairs", "prevSale":
20000, "cost": 50, "profit": 100}
            ]
        }

        response = self.app.post('/inventoryOptimization', json=data)
        self.assertEqual(response.status_code, 200)
        self.assertIn('profit', response.json)
        self.assertIn('quantity', response.json)

        data = {
            'budget': 500,
            'months': 1,
            'state': 'Assam',
            'products': [
                {"category": "Electronics", "subcategory": "Laptops",
"prevSale": 500, "cost": 200, "profit": 300},
                {"category": "Electronics", "subcategory": "Smartphones",
"prevSale": 700, "cost": 150, "profit": 200},
                {"category": "Furniture", "subcategory": "Chairs", "prevSale":
200, "cost": 50, "profit": 100}
            ]
        }

        response = self.app.post('/inventoryOptimization', json=data)
```

```
        self.assertEqual(response.status_code, 200)
        self.assertIn('profit', response.json)
        self.assertIn('quantity', response.json)

        data = {
            'budget': 100000,
            'months': 6,
            'state': 'Invalid_state',
            'products': [
                {"category": "Electronics", "subcategory": "Laptops",
"prevSale": 500, "cost": 200, "profit": 300},
                {"category": "Electronics", "subcategory": "Smartphones",
"prevSale": 700, "cost": 150, "profit": 200},
                {"category": "Furniture", "subcategory": "Chairs", "prevSale":
200, "cost": 50, "profit": 100}
            ]
        }

        response = self.app.post('/inventoryOptimization', json=data)
        self.assertEqual(response.status_code, 500)

        data = {
            'budget': 100000,
            'months': 10,
            'state': 'Assam',
            'products': [
                {"category": "Electronics", "subcategory": "Laptops",
"prevSale": 500, "cost": 200, "profit": 300},
                {"category": "Electronics", "subcategory": "Smartphones",
"prevSale": 700, "cost": 150, "profit": 200},
                {"category": "Furniture", "subcategory": "Chairs", "prevSale":
200, "cost": 50, "profit": 100}
            ]
        }

        response = self.app.post('/inventoryOptimization', json=data)
        self.assertEqual(response.status_code, 500)
```

```
        sys.stdout = sys.__stdout__
        devnull.close()
```

The test_inventory_optimization_success function tests the inventoryOptimization API endpoint by simulating multiple requests with varying inputs. In the first test, it provides a valid budget, months, state, and product data to ensure the API responds with a 200 status code, and the response includes both profit and quantity keys. It repeats the test with different valid inputs to verify that the optimization works under different conditions. It also tests an invalid state and a high number of months to check the handling of errors, expecting a 500 status code in these cases. Finally, it restores the standard output to avoid unnecessary logging during the test execution.

Result : -

File	function	statements	missing	excluded	coverage
backend.py	inventoryOptimization	14	0	0	100%