**Report on EX Banking Services**

Sample of Request and Response Structures for Exbanking Services

|  |  |  |
| --- | --- | --- |
| API | Request | Respond |
| Create\_user | {  "name": "Kiki Williams",  "email": "kiki.w@example.com",  "password": "secretpassword"  } | {  "user\_id": "1234567890",  "message": "User created successfully"  } |
| Deposit | {  "user\_id": "1234567890",  "amount": 100.00  } | {  "message": "Deposit successful",  "new\_balance": 500.00  } |
| Withdraw | {  "user\_id": "1234567890",  "amount": 50.00  } | {  "message": "Withdrawal successful",  "new\_balance": 450.00  } |
| Get\_balance | {  "user\_id": "1234567890"  } | {  "user\_id": "1234567890",  "balance": 450.00  } |
| Send | {  "sender\_id": "1234567890",  "recipient\_id": "0987654321",  "amount": 50.00  } | {  "message": "Funds sent successfully",  "new\_sender\_balance": 400.00  } |

Task

1. Test Case Summary

This test case report presents the functional and non-functional test cases for Ex Banking. The test cases cover various functionalities, including user creation, deposit, withdrawal, balance retrieval, and sending transactions. The non-functional test cases focus on the system's performance, usability, compactibility and scalability.

The provided test cases cover both functional and non-functional aspects of Ex Banking. These test cases aim to ensure the system's reliability, performance, compatibility, usability, and scalability.

1. Automation for non-functional testing with the use of the cypress framework.

Test Case 1

Ensure the account balance request returns a successful result in less than 500 milliseconds:

* + The test intercepts a GET request to '/API/account\_balance' and assigns it an alias 'getBalance.'
  + It then visits the URL '/balance' to trigger the account balance request.
  + The test waits for the 'getBalance' interception to complete and asserts that the duration of the intercepted response is less than 500 milliseconds.

Test Case 2

These test cases validate the expected behavior of the API when invalid amount values are provided.

1. **Negative Amount Test**: Sends a request with an amount of -10. The test expects the API to respond with a 400 Bad Request status code and an error message of 'Invalid amount.'
2. **Zero Amount Test**: Sends a request with an amount of 0. The test expects the API to respond with a 400 Bad Request status code and an error message of 'Invalid amount.'
3. **Non-Numeric Amount Test**: Sends a request with an amount of 'invalid' (a non-numeric value). The test expects the API to respond with a 400 Bad Request status code and an error message of 'Invalid amount.'

Each test uses the cy. request() function provided by Cypress to make the HTTP requests and the .then() function to handle the response and perform assertions using expect(). The failOnStatusCode: false option ensures that Cypress doesn't fail the test when the response has a non-2xx status code.

1. A

I used Postman because Postman's mock service virtualization capabilities efficiently simulate API responses, decouple frontend and backend development, and enable rapid iteration and testing. Its user-friendly interface, customization options, collaboration features, and integration with other Postman functionalities make it suitable for implementing mock services in the API development process.

B

The functional tests for an ExBanking service cover various functionalities of the ExBanking service, including user creation, depositing and withdrawing funds, and retrieving account balances. The tests are written using the Cypress testing framework. Here's a summary of what each test does:

1. Should create a new user with valid input data: This test sends a POST request to the /create\_user endpoint with valid user data and verifies that the response has a status code of 200 and contains a user\_id.
2. Should throw an error if the email provided already exists: This test sends a POST request to the /create\_user endpoint with an email that already exists in the system. It expects a response with a status code of 400 and an error message stating that the email already exists.
3. Should deposit a specified amount into the user's account successfully: This test sends a POST request to the /deposit endpoint with a user ID and an amount to deposit. It expects a response with a status code 200, a success message, and the new account balance.
4. Should only accept numeric and special character (.) in the amount deposited field: This test sends two POST requests to the /deposit endpoint, one with an invalid amount (non-numeric) and one with a valid amount (numeric with a decimal point). It expects the first request to return a status code of 400 and an error message and the second request to return a status code of 200 and the new account balance.
5. Should make a withdrawal from the user's account successfully: This test first deposits funds into the user's account by sending a POST request to the /deposit endpoint. Then, it sends a POST request to the /withdraw endpoint with the user ID and an amount to withdraw. It expects a response with a status code 200, a success message, and the new account balance.
6. Should retrieve the current account balance correctly for the given user: This test first deposits funds into the user's account by sending a POST request to the /deposit endpoint. Then, it sends a GET request to the /get\_balance endpoint with the user ID. It expects a response with a status code 200, the user ID, and the current account balance.