Test Strategy

Unit Testing:
Developers manually tested code during the process of developing new code.

Top-down Integration Testing:
With every added query, test to see if the GET/POST/DEL/PATCH requests output the correct information given and work as intended. For example, if we pass a user id with an email, we should get back the same user id and email. Each query should be tested to make sure there are no other conflicts surrounding other requests.

Regression Testing:
Performed manually through postman.

Smoke testing:
Developers test after implementing the final changes, before passing it off to the QA to test, approve, and allow for the front end team to integrate and implement.

Test Plan

The test case design will be scenario-based testing, which focuses on how the user interacts with the system, with test cases which are created to cover all use cases. The user interacts with the frontend to input their user information, such as name, email, and password to create an account. The majority of our test cases revolve around sending and receiving the correct information from the users' inputs, from requests to the database. We will be testing these requests through running the source code using the SAM CLI, alongside Postman and Docker. After performing GET requests, if the information matches the POST and we receive a JSON object that matches the structure we have outlined in DynamoDB, then the test is successful.

Test Cases

User Functions:
GET a user by ID - check if user with respective ID is returned [successful]

```
@RequestMapping(path = "/user/{id}", method = RequestMethod.GET)
  public UserResponse getUser(@PathVariable("id") String id)
```

POST a user using the RequestBody - create a user [successful]

```
@RequestMapping(path = "/user", method = RequestMethod.POST)
  public UserResponse createUser(@RequestBody User user)
```

GET all surveys taken by a user - check if all surveys are returned [successful]

```
@RequestMapping(path = "/user/{id}/survey", method = RequestMethod.GET)
  public List<UserSurveyResponse> findAllUserSurveys(@PathVariable("id")
String id)
```

GET a specific survey taken by a user - check if survey with respective survey ID is returned [successful]

```
@RequestMapping(path = "/user/{uid}/survey/{sid}", method =
RequestMethod.GET)
  public UserSurveyResponse findUserSurvey(

      @PathVariable("uid") String userId, @PathVariable("sid") String
surveyId)
```

POST a survey response taken by a user - create survey under specific user ID [successful]

```
@RequestMapping(path = "/user/{id}/survey", method = RequestMethod.POST)
  public UserSurveyResponse createUserSurvey(
      @PathVariable("id") String id, @RequestBody Survey survey)
```

DELETE a survey response taken by a user - check if survey with respective survey ID is deleted [successful]

```
@RequestMapping(path = "/user/{id}/survey/{sid}", method =
RequestMethod.DELETE)
  public UserSurveyResponse deleteUserSurvey(
      @PathVariable("id") String userId, @PathVariable("sid") String
surveyId)
```

GET info from a specific circle member in a user circle - check if circle member information with respective circle member ID is returned [successful]

```
@RequestMapping(path = "/user/{uid}/circle/{email}", method =
RequestMethod.GET)
  public CircleMemberResponse getCircleMember(
      @PathVariable("uid") String userId, @PathVariable("email") String
email)
```

GET info from all members in a user circle - check if all circle members within a specific user with respective ID is returned [successful]

```
@RequestMapping(path = "/user/{uid}/circle", method = RequestMethod.GET)
  public List<CircleMemberResponse> getUserCircle(@PathVariable("uid")
String userId)
```

POST a new member in the user circle - create a new circle member if circle size is less than 5 and member email does not already exist [successful]

```
@RequestMapping(path = "/user/{uid}/circle", method = RequestMethod.POST)
  public CircleMemberResponse addCircleMember(
      @PathVariable("uid") String userId, @RequestBody Circle circle)
```

DELETE a member in the user circle - check if user with respective ID is returned [successful]

```
@RequestMapping(path = "/user/{uid}/circle/{email}", method =
RequestMethod.DELETE)
  public CircleMemberResponse removeCircleMember(
      @PathVariable("uid") String userId, @PathVariable("email") String
email)
```

PATCH a member in the user circle using the RequestBody - update circle member information in user's circle [successful]

```
@RequestMapping(path = "/user/{uid}/circle", method = RequestMethod.PATCH)
  public CircleMemberResponse updateCircleMember(
      @PathVariable("uid") String userId, @RequestBody Circle circle)
```

GET a user's health score - check if user's health score with respective ID is returned [successful]

```
@RequestMapping(path = "/user/{uid}/healthscore", method =
RequestMethod.GET)
  public UserHealthScoreResponse getUserHealthScore(@PathVariable("uid")
String id)
```

POST a user's health score - create user's health score [successful]

```
@RequestMapping(path = "/user/{uid}/healthscore", method =
RequestMethod.POST)
  public UserHealthScoreResponse updateUserHealthScore(
      @PathVariable("uid") String id, @RequestBody int healthScore)
```

PATCH a user's health score - update user's health score [successful]

```
@RequestMapping(path = "/user/{uid}/healthscore", method =
RequestMethod.PATCH)
  public UserHealthScoreResponse patchUserHealthScore(
      @PathVariable("uid") String id, @RequestBody int healthScore)
```

Survey Functions:

GET a specific survey by its ID - check if survey with respective ID is returned [successful]

```
@RequestMapping(path = "/survey/{sid}", method = RequestMethod.GET)
  public SurveyResponse findSurveyById(@PathVariable("sid") String
surveyId)
```

POST a survey using the RequestBody - create a survey [successful]

```
@RequestMapping(path = "/survey", method = RequestMethod.POST)
  public SurveyResponse addSurvey(@RequestBody Survey survey)
```

Traceability Matrix

Rows = version number

Columns = Test cases

	Req 1.1	Req 1.2	Req 2.1	Req 2.2	Req 2.3	Req 2.4	Req 2.5	Req 2.6	Req 3.1	Req 3.2	Req 3.3	Req 3.4	Req 3.5	Req 4.1	Req 4.2	Req 4.3
1	X	X														
2									X	X	X	X	X			
3			X	X	X	X	X	X						X	X	X

Requirements:

1.1 - create a user

1.2 - get a user by their ID

2.1 - create a survey

2.2 - get specific survey by its ID

2.3 - post survey response taken by a user

2.4 - get a specific survey taken by a user

2.5 - delete a survey response taken by a user

2.6 - get all surveys taken by a user

3.1 - create new member in a user's circle only if the number of members in their circle is less than 5 and a member with that the given email does not already exist

3.2 - delete a member in a user's circle

3.3 - get info from a specific member in a user's circle

3.4 - get info from all members in a user's circle

3.5 - update a member's information in a user's circle

4.1 - create a user's health score

4.2 - get a user's health score

4.3 - update a user's health score

Test Results

Regarding our testing, it was done in conjunction with the developer in which the developer would test the functions locally, push the changes, and then testing would be done again after the changes were deployed to find issues that weren't caught when tested locally. Github was used for version control to communicate between the developers and QA on updated versions of the application and to test for any new bugs or problems with the changes.

During testing, one issue we encountered involved getting a user ID that did not exist. This error would throw an exception and return a 502 Bad Gateway Error. This error was discovered locally by the developer and was fixed by adding this line of code:
"*.orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "No User Found"));*
Instead of getting an exception thrown, this would result in returning a 404 Not Found error which better represents retrieving a user ID that does not exist. This error applied to all of our GET functions, but was first noticed while getting a user ID. Another issue we ran into during testing was receiving errors while using the update function from the DynamoDB API. The error was caused by a syntax error for a circle member's ID. This issue was also discovered locally by a developer, who was able to think of a solution which involved replacing the whole table item instead of updating the fields that we needed to change.

The completed version of the application was finished on the week of our presentation. After giving the completed API endpoint to the frontend team we were working with, all we had to do was wait for them to finish implementing it into their program. The completed program was finished the day of the presentation and we did not run into any errors with the backend and frontend software communication.