Test Plan:

AE Project Management Tool

Group 2: JSON Bourne

Date: April 7, 2020
Version: 1.0.1

Document Information

Revision History

Version	Date	Status	Prepared By	Comments
1.0.0	03/12/2020	Complete	JSON Bourne	N/A
1.0.1	04/07/2020	Complete	Jacqueline Yin	Updated method of performance testing in performance section

Document Control - Team Participants

Role	Name	Email
Project Manager/Developer	Jacqueline Yin	yin.sangxu@gmail.com
Backend Developer	Jenessa Tan	jenessatan@alumni.ubc.ca
Backend Developer	Bang Chi Duong	bcduong@students.cs.ubc.ca
UX-UI Designer/ Front-end Developer	Katrina Tan	katrinatan91@yahoo.com
Front-end Developer	Lisa O'Brien	lisa.obrien@alumni.ubc.ca
Front-end Developer	Stacy Leson	stacy.leson@gmail.com
Full Stack Developer	Nicole Kitner	nicolekitner@gmail.com

Stakeholders / Signoff list

Role	Name	Signature	Sign-Off Date
Supervisor	Dave Pagurek		
Sponsor	Nash Naidoo		
Sponsor	Steve Robinson		
Sponsor	Shawn Goulet		
Sponsor	Eric Ly		

Table of Contents

Summary

Approach

As our project revolves around a user-friendly GUI by which Resource managers and Project managers may view the information of and manage their resources, we will ensure thorough dynamic System Acceptance Testing using a manual approach. Due to time limitations, we take a risk based approach to System Acceptance Testing, where we will focus on components that have been either 1.) Modified or 2.) Newly implemented. We will also include white-box unit tests for the project’s major components to ensure robustness of the system. We do not anticipate the use of tools in our testing process. Since C# is a compiled language, the compiler performs static code analysis before runtime.

The anticipated acceptance criterion to move forward with the build is that at least 90% of tests ran pass, and that the major functionalities (i.e. create, edit, delete, search) of each component have not broken.

Test Cycles

A brief subset of the System Acceptance Tests will be conducted before every deployment (i.e. once every week). The subset will be chosen via our risk based approach. Likewise, we will run the unit tests relevant to the new/modified components before each deployment, prior to merging the new code into the central repository. The frequency of this will be dependent on how often new code will be merged.

Stages of Testing

During the test execution process, we will undergo different stages of testing:: Unit testing, System Acceptance Testing, and User Acceptance Testing. Unit testing will be completed by the developers (i.e. the JSON Bourne team) to ensure that each code unit is producing the appropriate output for invalid and valid inputs. For System Acceptance Testing (SAT), the JSON Bourne development team will act as QA during the SAT stage of testing, to ensure that all features that had been agreed upon meet the requirements specified in the requirements and design documents. User Acceptance Testing will be conducted by stakeholders.

Test Environment Setup

Test Environment Set-up Requirements

Dev Environment testing (for debugging purposes)

- **OS:** Microsoft Windows 10 Version 1909 (OS Build 18363.719)
- **To run Server side:**
    - C# 8.0
    - .Net Core 3.1
    - IDE: Visual Studio Community v16.0.29519.181
- **To run Client side:**
    - Node.js v10.16.3
    - npm v6.9.0
    - React v16.12.0
    - IDE: Visual Studio Code v1.41.1
    - Material-UI v4.9.1
- **Database:**
    - SQL Server Express v14.0.1000.169
    - IDE: SQL Server Management Studio v18.4
- **API:** Azure Active Directory[1]
- **Testing:** Postman v7.17.0, xUnit v2.4.1, Enzyme v3.11.0
- **Browser:** Chrome v80.0.3987.132 (Official Build) (64-bit), IE11 v11.719.18362.0

Production Environment testing (for Acceptance Testing)

- **Web Server (Azure):** Windows Server with IIS 10.0
- **Database:**
    - Cloud SQL Server Express v14.0.1000.169
    - IDE: SQL Server Management Studio v18.4
- **Browser:** Chrome v80.0.3987.132 (Official Build) (64-bit), IE11 v11.719.18362.0

[1] We could not find the version for the Azure Active Directory

Functional Test Plan

Unit and System Acceptance Testing

Server-side (applicable to all components listed below):

- Black-box testing
    - Manual testing
        - Performed on Swagger UI (<domain url>/swagger)
        - Advantage: fast to test individual endpoints, able to speculate data structure of inputs and outputs
        - Risk: slow to test overall, requires immediate attention while manually creating inputs and clicking options
    - Automated testing
        - Performed using xUnit, Moq, Coverlet, Report Generator
        - Advantage: automated, can be implemented into build pipeline, able to debug
        - Risk: unable to pinpoint origin of error due to black box nature
- White-box testing
    - Automate testing
        - Performed using xUnit, Moq, Coverlet, Report Generator
        - Advantage: automated, can be implemented into build pipeline, known error origins
        - Risk: difficult to design

Server Side - Edit User:

- Unit test: automated white box approach
    - Mock repository/data
    - Testing through the controller
- SAT: automated and manual black box approach; white box (due to time constraints may not have this completed)
- Total anticipated scripts from server-side:
    - Happy Paths:
        - The repositories (data manipulation components) return correct result
        - If errors occur while querying, return the errors with proper status codes and messages
    - Negative Paths: check conditions for each of the following
        - UserSummary: userID, locationID
        - Availability: fromDate has to be before toDate
        - Disciplines: disciplineID
    - Roughly 50 automated tests (unit and SAT for server) in an ideal scenario (reality for this project may be much less)

Client Side:

- Black-box testing
    - Manual testing
        - Performed by manually inputting values and testing the output on both IE11 and Chrome
        - Advantage: Ensures that human interaction is taken into account and user inputs work as expected
        - Risks: Not all routes or paths are covered by manual testing. We will attempt to moderate this by adding white-box testing as well.
    - Automated testing
        - We will not be performing any black box automated testing for the client side. In the future, automated testing could be implemented to input random user inputs and interactions to ensure that nothing breaks our implementation.
- White-box testing
    - Automate testing
        - We will be using Enzyme for React to create unit tests for the main components
        - Advantage: Unit tests will ensure that each of the main components will be working as expected
        - Risks: Only having unit tests can be a problem since the integration may not act as expected. We can mitigate this by using black-box testing and manual testing to test the interactions between components.

Client - Edit User:

- Unit Test: Automated White Box
    - Mock the database by adding objects into the initial state in the front end
- Unit Test: Automated Black Box
    - Test through having a user input values and attempting to update the user's profile
- Total anticipated scripts from client-side:
    - Happy Paths:
        - The repositories (data manipulation components) return correct result
        - If errors occur while querying, return the errors with proper status codes and messages
        - Disciplines and Skills renders as expected: When a discipline is selected, only the skills from that discipline are displayed
        - Location Field renders correctly: When a province is selected, only the cities in that province are output
        - Removing a discipline should remove the discipline from the list
        - Clicking Save with the appropriate fields will update the user information
    - Negative Paths: check conditions for each of the following
        - Inputting a null name field will return a warning

- Adding a null skill will result in a warning
- Saving with a null Location will result in a warning

User Acceptance Testing

Suggest Stakeholders to test in accordance with use cases outlined in our Requirements Traceability matrix.

Components

Main Section	**Component**
User	Edit user
	Get a single user
Project	Edit project
	Get a single project
	Delete project
Search	Get all users (search)
	Post all users (search)
	Get all projects (search)
Forecasting	Assign user and update utilization
Admin	Add a discipline
	Delete a discipline
	Add a skill
	Delete a skill
	Add a location
	Delete a location

Non-Functional Test Plan

Performance & Scalability Testing

We will be measuring the response times of our API calls listed below using Chrome's Developer Tools >> Performance >> Recording, and will be ensuring that they are within reasonable values:

API CALL	ACCEPTABLE RESPONSE TIMES (Normal load vs Peak load)	
User Module		
Load list of users (50 users)	< 3s	< 6s
Load single user page	< 2s	< 4s
Save edits on single user	< 3s	< 4s
Project Module		
Load list of projects (50 projects)	< 3s	< 6s
Load single project page	< 2s	< 4s
Create a new project	< 3s	< 4s
Edit a project	< 3s	< 4s
User Search Module		
Return search results with one filter	< 6s	<12s
Return search results with multiple filters	< 10s	< 15s
Return search results with user's text input	< 6s	< 6s
Forecasting Module		
Assigning a new user (after search results)	< 1s	< 3s

Performance tests shall be run multiple times: with the normal expected load of a single user, and with multiple users using the database simultaneously. The tests should be done on a single machine to create an accurate standardized benchmark. All tests must pass and fall within the reasonable response times listed above for each endpoint. We will be manually testing and recording response times for the single user case.

To perform a multiple user test, we recommend using Gatling scripts to simulate multiple concurrent requests. However, due to time constraints, we will not implement these scripts. As an alternative, we plan to manually test simultaneous requests with 7 users, by submitting the same request through the GUI at the same time, and averaging the response times for each API request. (In this case, we will be unable to conduct the manual testing of performance on a single machine).

While the performance tests look at the response times of the API calls with multiple users, the scalability tests use similar test cases but we will be testing the performance of the application with a large test database. The test data will be generated using a script taken from GenerateData to generate large volumes of data in the database. The normal load was specified to be at least 1000 users and around 50 projects. The larger load will be set at roughly 2000 users and 100 projects. To be considered successful, all tests must pass for both passes and the response times must be within the reasonable response times stated above.

Test Scripts

Use cases

- Client side use cases will be tested manually in both the latest version of Google Chrome and IE 11 by following specific acceptance criteria, and using Enzyme, which is a javascript testing utility for React, used to create automated tests for the components' output. **See appendix for details of specific scripts**

Server

- Manual testing using the Swagger UI will be done for the User, Projects, Forecasting, Search and Admin endpoints as programming progresses
- Automated unit tests using xUnit and mock repositories (Moq framework) will be run on each of the Controllers. With Moq, we can mock user's inputs, set up the repository's inputs and outputs to test for backend MVC flow/framework, checking for logic paths inside each of the main methods corresponding to the endpoints. We can also do functional (black-box) testing and implementation-detail (white-box) testing on all relevant repositories' methods.
- Automated integration tests (SAT) for each of the API endpoints will be created using xUnit. A testing database with test seeds will be queried in these integration tests to avoid contamination of data in the existing database. Test scripts will be created for each component following this basic outline to ensure appropriate functional coverage:
    - Valid input request
    - Varying the structure of the input request to trigger error response, including
        - Reference to non-existent item in database
        - Missing field values
        - Null values
        - Conflicting values (e.g. starting date after the end date)
    - Invalid structure for request
- At the end of automated testing, we will generate a coverage report using Coverlet and ReportGenerator.

Security Tests

As this application does not have any sensitive information, we will not be implementing any data masking. As such, no tests will be made to cover this aspect. Penetration testing could be done by AE to ensure the security of the application but this is beyond the scope of the project.

Security testing will focus mostly on access control testing to ensure that users have access to app resources that are appropriate to the role/group they are assigned to in Azure's Active Directory. See the test cases below in the table below. These cases were generated based on the Role Based Access Control matrix submitted with the design document. (Please see RBAC matrix in the appendix)

User Type	**Level of Access**
Human Resource/Regular User	● Edit and save own profile ● View details of projects that they are part of ● View other users' profiles
Project/Resource Manager	● Create new projects, add openings to project ● Edit their own existing project ● View other projects ● Assign a user to a project ● Confirm a regular user's participation in a project ● Edit/Save a regular user's profile ● Update user's availability ● Edit/Save their own profile ● View other user's profiles
Admin	● All the abilities of the Project/Resource Manager ● Add or remove options for disciplines and skills from the list of available disciplines ● Add or remove options for locations ● Add or remove options for years of experience

Since role assignment is done through Azure Active Directory, there are no plans to test the correct assignment of a user to a role. The security test will be done by creating sample users for the app with each user having their own role. Manual testing is required to ensure that each user role has the correct access level as stated in the table above. A second run of the test can be made after changing the roles of each user and ensuring their level of access remains appropriate to their assigned role.

Regression Testing

Regression packs
Updating users

- Checking that they are updated in all places in the system
- Update user, then recheck that they are updated in the Search and Project modules

Adding/Deleting users from projects

- See that these updates are reflected in the user’s profile and Search
- See that the users utilization is changed to reflect the addition/deletion

Adding/deleting projects

- These additions/deletions are reflected in the Search module
- Reflected in the users module

Appendix

Test Script Details

Client-side unit tests

<u>Manual blackbox unit testing</u>

to be done in both the latest version of Google Chrome (this is version 80.0.3987.132 at the time of this writing) and IE 11

Search Page

Instruction	**Expected Result (Acceptance Criteria)**
Navigate to the search page, perform a search using specific filter parameters after clicking the “Apply filters” button.	**Verify** that the results reflect the filters that are being used
Navigate to the search page, perform a search with filling out filter parameters, but **not** clicking “Apply Filters”.	**Verify** that the results **do not** reflect the parameters chosen that have not been saved by the “Apply filters” button
Navigate to the search page, search using only a search word	**Verify** that the results reflect the search word being used
Navigate to the search page, pick some filters and click “Apply filter”.	**Verify** that the filters chosen are added as tabs as the top of the filter section
Navigate to the search page, pick some filters and click “Apply filter”. Delete a few of the tabs at the top, while keeping some.	**Verify** that only the filters that have not been deleted are applied to the search.
Navigate to the search page, click the “+” button next to one of the categories and add a bunch of the same type of filter. Click “-” to remove the filter.	**Verify** that the added filter options are being deleted properly.
Click the search button without any parameters so that all users show as results	**Verify** there are only 50 users per page, and navigating to another page shows different users than the page before

Users Page

Instruction	**Expected Result (Acceptance Criteria)**
Navigate to Users page	**Verify** users show up on the page with edit

	links under their names **Verify** there are only 50 users per page, and navigating to another page shows different users than the page before
Navigate to Users page, click on “edit” under the first user	**Verify** clicking on “edit” links to the edit user page
Add a new discipline for the user and click the “+” button	**Verify** when you click the “+” button it shows the new discipline at the bottom of the page
Click “save” after adding the discipline, and navigate back to the users page. Go find the user you just edited.	**Verify** that the new discipline is saved and added to the users profile when you click “edit”
Click “edit” again and change the name/location of the user and click save	**Verify** that the user’s name/location is changed when the save button is clicked

Project page

Instruction	**Expected Result (Acceptance Criteria)**
Navigate to Projects page	**Verify** project show up on the page with edit links under their names **Verify** there are only 50 project per page, and navigating to another page shows different projects than the page before
Navigate to Projects page, click on “edit” under the first project	**Verify** clicking on “edit” links to the edit project page
Add a new Team Requirement for the project and click the “+” button	**Verify** when you click the “+” button it shows the new team requirement opening at the bottom of the page
Click “save” after adding the team requirement, and navigate back to the projects page. Go find the project you just edited.	**Verify** that the new team requirement is saved and added to the project profile when you click “edit”
Click “edit” again and change the name/location of the project and click save	**Verify** that the project’s name/location is changed when the save button is clicked

Forecasting module

Instruction	**Expected Result (Acceptance Criteria)**

Navigate to a project with available openings and click the “assign” button next to an opening	Verify this brings you to the forecasting page
Assign any user to this opening, making note of the users initial utilization	**Verify** you are navigated back to the project page and that the opening is no longer visible **Verify** the user is now a part of the team and their utilization is properly updated

Admin page

Instruction	**Expected Result (Acceptance Criteria)**
Navigate to the Admin page	**Verify** you see a list of all disciplines, skills, provinces, and cities
Display updates	**Verify** clicking a new discipline updates the skills shown **Verify** clicking a new province updates the cities shown
Add fields	**Verify** adding a new discipline results in the discipline displaying **Verify** you can add a new skill to an existing discipline **Verify** adding a new province results in the new province being displayed **Verify** adding a new city for a province results in that city showing
Delete fields	**Verify** you can delete a discipline and the skills are deleted too **Verify** you can delete a skill **Verify** you can delete a province and the cities are deleted too **Verify** you can delete a city

Client side Enzyme automated testing

User module

Component	**Input**	**Output**	**Description**

UserList	const renderUserList = args => {
	  const user = [
	    {
	      id: 0,
	      firstName: '',
	      lastName: '',
	      username: '',
	      locationId: 0,
	    },
	{
	      id: 1,
	      firstName: '',
	      lastName: '',
	      username: '',
	      locationId: 0,
	    },
	
	  ];
	  return shallow(<UserList users={user} />);
	};
	
	renderUserList()
		expect(userList length= 2)	UserList component returns correct number of users given input

UserCard	const renderUserCard = args => {
	  const user = [
	    {
	      id: 0,
	      firstName: 'John',
	      lastName: 'Smith',
	      username: jsmith'',
	      locationId: 2,
	    },
	  ];
	  return
		expect(UserCard.id = 0)
		
		expect(UserCard.firstName = ‘John’)
		
		expect(UserCard.lastName = ‘Smith’)
		
		expect(UserCard.username = ‘jsmith’)
		
		expect(UserCard.l
			UserCard component returns correct user details given input

	shallow(<UserList
	users={user} />);
	};
	
	renderUserCard()
		ocationId = 2)	

UserDetails	const
	renderUserCard =
	args => {
	  const userProfile = [
	    {
	      UserSummary
	    }]
	renderUserDetaills()
	
	(**UserSummary** is
	extremely long)
		expect(userProfile
		.id, firstname,
		lastname,
		location,
		utilization,
		resourceDiscipline
		, currentProjects,
		availability,
		disciplines,
		positions)
			UserDetails
			component returns
			correct
			userSummary data

EditUser	const renderEditUser
	= args => {
	  const editUser = [
	    {
	      userProfile: {},
	        pending: true,
	        masterlist: {}
	    },
	]
	renderEditUser()
		expect(editUser.a
		ddDisciplines(ope
		nings) =
		userProfile with
		added discipline)
		
		expect(editUser.a
		ddUserDetails(use
		rProfile) =
		userProfile with
		new information)
			EditUser
			component
			modifies the
			EditUser
			component when
			the functions
			addDisciplines() or
			addUserDetails()
			are called.

Search module

Component	**Input**	**Output**	**Description**

FilterTab	const
	renderFilterTab =
	args => [{
	utilization: {
	    min: 0,
	    max: 100
	},
	locations: [],
		expect(onSubmit()
		= list of users that
		fit the filters)
		
		expect(saveFilter()
		= filter specs are
		saved to the state
		but not submitted
			FilterTab returns a
			list of the users with
			the specific filters
			outlined in the
			const FilterTab

	disciplines: {}, yearsOfExps: [], startDate: null, endDate: null, }, orderKey: "utilization", order: "desc", page: 1, }, },] renderFilterTab()	yet)	

Projects module

Component	Input	Output	Description

ProjectList	const renderProjectsList = ([ProjectObjectOne, ProjectObjectTwo]);
	
	return shallow(<ProjectList projects={project} />);
	
	renderProjectsList()
		expect(ProjectList length =2)	ProjectList component returns correct number of projects given input

ProjectCard	const renderProjectCard = ({
	3, project, true, true, userRole});
	
	Return shallow(<ProjectList projects={project} />);
	
	Const project = {
		expect(ProjectCard. number == 3)
		
		expect(ProjectCard. project == project)
		
		expect(ProjectCard. canEditProject == true)
		
		expect(ProjectCard. onUserCard ==
			ProjectCard returns correct details given input

	title: "Martensville Athletic Pavillion",
	location: {
	locationID: 1,
	province: "Seskatchewan",
	city: "Saskatoon"
	},
	
	projectStartDate: "2020-10-31T00:00:00.0000000",
	
	projectEndDate: "2021-12-31T00:00:00.0000000",
	
	projectNumber: "2009-VD9D-15"
	}
	
	renderProjectCard()
		true)	

EditProject (CreateEditProjectDetails)	const renderCreateEditProjectDetails = ({project},
	[city_x, city_y],
	[province_x, province_y],
	pending: true
	}
	
	Const project = {
	title: "Martensville Athletic Pavillion",
	location: {
	locationID: 1,
		expect(CreateEditProjectDetails.projectSummary == project)
		
		expect(CreateEditProjectDetails.projectSummary == projectEdit)
		
		#Make sure new project isn't just created on edit
		
		expect(ProjectList length = 1)
			Able to edit a project

	province: "Seskatchewan",
	city: "Saskatoon"
	},
	
	projectStartDate: "2020-10-31T00:00:00.0000000",
	
	projectEndDate: "2021-12-31T00:00:00.0000000",
	
	projectNumber: "2009-VD9D-15"
	}
	
	Const projectEdit = {
	title: "Martensville Athletic Pavillion NAME EDIT",
	location: {
	locationID: 1,
	province: "Seskatchewan",
	city: "Saskatoon"
	},
	
	projectStartDate: "2020-10-31T00:00:00.0000000",
	
	projectEndDate: "2021-12-31T00:00:00.0000000",
	
	projectNumber: "2009-VD9D-15"
	}		

	renderCreateEditProjectDetails(project) renderCreateEditProjectDetails(project)		

Forecasting module

Component	Input	Output	Description
AssignUser	const renderForecasting = ({ User, OpeningID }) renderForecasting()	expect(Forecasting.user.utilization == newUtilization)	Forecasting values are updated

Admin module

Component	Input	Output	Description

Admin	const renderAdmin = ({
	{discipline,
	skill,
	location,
	selectedprovince,
	masterlist});
	
	renderAdmin()
		expect(Admin.discipline == discipline)
		
		expect(Admin.discipline == skill)
		
		expect(Admin.discipline == location)
		
		expect(Admin.discipline == selectedprovince)
			Make sure Admin values are correct

Server-side Testing:

- This is the set of tests written for requests associated to users. Since the other requests in our system will be very similar aside from the data structure used (projects component), this strategy will also be applied to those. For simplicity, we will not be explicitly writing out those test scripts.
- These automated tests will be written with the use of the *xUnit (v2.4.1)* ; the strategy also follows for our manual tests in our *Swagger UI*

Server-side Unit Tests

- For this Unit Suite Test, we also used *Moq (v4.13.1)* to mock the repository components and also the mapper

Edit User

```
"userProfile": {
    "userSummary": {
        "userID": 3,
        "firstName": "Nat", // cannnot be null
        "lastName": "Romanov", // cannnot be null
        "location": {
            "locationID": 1,
            "province": "Alberta", // cannnot be null
            "city": "Calgary" // cannnot be null
        },
        "utilization": 117,
        "resourceDiscipline": {
            "discipline": null,
            "yearsOfExp": null
        },
        "isConfirmed": false
    },
    "currentProjects": [
        {
            "projectNumber": "2005-KJS4-46",
            "title": "Budapest",
            "locationId": 19,
            "projectStartDate": "2020-04-19T00:00:00",
            "projectEndDate": "2020-07-01T00:00:00"
        }
    ],
    "availability": [
        {
            "fromDate": "2020-04-07T00:00:00", // check if fromDate is
before toDate
            "toDate": "2020-04-19T00:00:00",
            "reason": "Maternal Leave"
        },
        {
            "fromDate": "2020-10-31T00:00:00", // check if fromDate is
before toDate
            "toDate": "2020-11-11T00:00:00",
            "reason": "Maternal Leave"
        }
    ],
    "disciplines": [
        {
            "disciplineID": 1,
            "discipline": "Language", // cannot be null
            "yearsOfExp": "10+", // cannot be null
            "skills": [
                "Russian"
            ]
        },
        {
            "disciplineID": 2,
            "discipline": "Weapons", // cannot be null
            "yearsOfExp": "10+", // cannot be null
            "skills": [
                "Glock", "Sniper Rifle"
            ]
        }
    ],
    "positions": [
        {
            "projectTitle": "Budapest",
            "disciplineName": "Intel",
            "projectedMonthlyHours": 170
        }
    ]
}
```

Input	Output	Description
The userID on the request URL does not match with the userId inside userProfile > userSummary > userID	400 - Bad Request	Changes to database should not be made with invalid input
Have a test case for each of the following scenarios (create them): -userProfile is null 1. userSummary is null a. firstName is null b. lastName is null c. location is null i. province is null ii. city is null 2. availability is null a. fromDate is not before toDate 3. disciplines is null a. discipline is null b. yearsOfExp is null	400 - Bad Request	Changes to database should not be made with invalid input
Create and input a userId that does not exist in database	404 - Not Found	Should not update DB of a user that doesn't exist
A request without authentication bearer inside the request header	401 - Not authorized	Should not be able to access endpoint without proper authorization
The repository components we use in this endpoint are: 1. locationsRepository 2. usersRepository 3. disciplinesRepository 4. skillsRepository 5. outOfOfficeRepository These will be mocked to be setup and output a desirable result from querying	200 - Success	If all positive paths get called leading to the final response call to return to client then should return 200 to the client

the database, and see if the final return from the endpoint is what's expected.
For example:
Mock locationsRepository:
1. Setup:
    a. getALocation("Toronto")
2. Setup Output:
    a. {
        "id": 1,
        "province": "Ontario",
        "city": "Toronto"
    }
3. Check that the next call for whichever helper actually gets called
4. Do this to test sequentially that the call path for all positive (happy) paths does not break unexpectedly		

Throws SQLException while setting up the mock repositories	500 - Internal Server Error	Should catch SQLException and return a 500 to client
Intentionally throws a System.Exception (most generic error) for unexpected error	400 - Bad Request	Should return a 400 to client if unexpected error is thrown (design choice)

Server-side SAT

Edit User:

```
"userProfile": {
    "userSummary": {
        "userID": 3,
        "firstName": "Nat", // cannnot be null
        "lastName": "Romanov", // cannnot be null
        "location": {
            "locationID": 1,
            "province": "Alberta", // cannnot be null
            "city": "Calgary" // cannnot be null
        },
        "utilization": 117,
        "resourceDiscipline": {
            "discipline": null,
            "yearsOfExp": null
        },
        "isConfirmed": false
    },
    "currentProjects": [
        {
            "projectNumber": "2005-KJS4-46",
            "title": "Budapest",
            "locationId": 19,
            "projectStartDate": "2020-04-19T00:00:00",
            "projectEndDate": "2020-07-01T00:00:00"
        }
    ],
    "availability": [
        {
            "fromDate": "2020-04-07T00:00:00", // check if fromDate is
before toDate
            "toDate": "2020-04-19T00:00:00",
            "reason": "Maternal Leave"
        },
        {
            "fromDate": "2020-10-31T00:00:00", // check if fromDate is
before toDate
            "toDate": "2020-11-11T00:00:00",
            "reason": "Maternal Leave"
        }
    ],
    "disciplines": [
        {
            "disciplineID": 1,
            "discipline": "Language", // cannot be null
            "yearsOfExp": "10+", // cannot be null
            "skills": [
                "Russian"
            ]
        },
        {
            "disciplineID": 2,
            "discipline": "Weapons", // cannot be null
            "yearsOfExp": "10+", // cannot be null
            "skills": [
                "Glock", "Sniper Rifle"
            ]
        }
    ],
    "positions": [
        {
            "projectTitle": "Budapest",
            "disciplineName": "Intel",
            "projectedMonthlyHours": 170
        }
    ]
}
```

Input	**Output**	**Description**
The userID on the request URL does not match with the userId inside userProfile > userSummary > userID	400 - Bad Request	Changes to database should not be made with invalid input

Have a test case for each of the following scenarios:
 	
-userProfile is null
4. userSummary is null
    a. firstName is null
    b. lastName is null
    c. location is null
        i. province is null
        ii. city is null
5. availability is null
    a. fromDate is not before toDate
6. disciplines is null
    a. discipline is null
    b. yearsOfExp is null
	400 - Bad Request	Changes to database should not be made with invalid input

A userId that does not exist in database	404 - Not Found	Should not update DB of a user that doesn't exist
A request without authentication bearer inside the request header	401 - Not authorized	Should not be able to access endpoint without proper authorization
UserProfile with valid entries in all of the fields in the structure (changing fields in separate tests)	200 - User ID returned	Valid input should successfully update DB entry
Get All Users (Search)		
Request to get all users (includes valid & invalid inputs for searchWord, orderKey, order and pageNumber)	200 - returns list of all users from database that match search parameters	Request should return all users matching parameters
A request without authentication bearer inside the request header	401 - Not authorized	Should not be able to access endpoint without proper authorization
Get A User		
Request a user with a valid userId in URL	200 - returns UserProfile of that user	Valid request should return requested user
Request user with invalid userId in URL	404 - Not found	Invalid request should not return a user that doesn't exist
A request without authentication bearer inside the request header	401 - Not authorized	Should not be able to access endpoint without proper authorization
Post User (Search)		
Request to get all users (includes valid & invalid inputs for searchWord, orderKey, order and pageNumber)	200 - returns list of all users from database that match search parameters	Request should return all users matching parameters
A request without authentication bearer inside the request header	401 - Not authorized	Should not be able to access endpoint without proper authorization

- These test cases would are also extended to the remaining admin components

Add Skill (Admin)		
Request with a valid skill and valid disciplineID	200 - returns skillID	Request should insert new valid skill in DB
Request with invalid skill, invalid disciplineID or URL disciplineID does not match skill's disciplineID	400 - Bad Request	Should not be able to add invalid skill
Delete Skill (Admin)		
Request with valid skill name and valid disciplineID	200 - returns skillID that was deleted	Request should successfully delete valid skill from DB
Request with invalid skill name or invalid disciplineID	400 - Bad Request	Should not be able to delete an invalid skill from DB
Request with non-existent skill name	404 - Not found	Should not be able to delete a non-existent skill from DB

- These are specific test cases for forecasting that are not covered above

Assign Resource (Forecasting)		
Request with valid openingID and valid userID	200 - returns openingID, userID, user's utilization	
Request with either invalid openingID or invalid userID	404 - Not found	
Confirm Resource (Forecasting)		
Request with valid openingID and valid userID	200 - returns openingID, userID, user's updated utilization	
Request with either invalid openingID or invalid userID	404 - Not found	

Role-Based Access Control Matrix

	Application Resources

Roles	Users		Projects		Settings	
	Own Profile	Others' Profile	Own Projects	Others' Projects	Locations	Discipline & Skills
Resource (Regular User)	RU	R	R	R	-	-
Project Manager	RU	RU	CRUD	R	-	-
Administrator	RU	RU	CRUD	CRUD	CRUD	CRUD