Coast Capital Contractor Records Database Management System Test Plan

The Terminal

1st November 2017

Table of Contents

A. SUMMARY

Our team has decided to adopt the Agile philosophy of Test Driven Development for this project and thus we will be testing from the very the day we start writing code. Effectively this means our application will be less error prone and will contain fewer bugs and that the bugs it does contain will be found early on in the development process so we can address them sooner rather than later.
Our approach for static testing is to have feature based branches on github so that each branch in github focuses on implementing one single component. This allows for an easier code review process where prominent bugs as well as stylistic issues can be easily caught. Each branch before being merged into the master branch is required to be approved by at least one other member of the team who is not the creator of the branch.
We have also decided to use Travis CI for continuous integration so that we can detect build issues in branches before merging them in the master branch of our github repository which is the branch we are using to deploy on AWS.
For the frontend, we will be using manual testing and triangulating this type of testing with a browser automation tool, Selenium, to test the UI components of our system. We will be using Mocha and Chai for automated frontend testing, and to fake the server in order to substitute responses that we need for API testing, we will be using Nock.
For the backend we will be using JUnit to automate our Unit Tests for testing our handling of the api calls for the frontend as well as making queries to the database. As we are just only making simple queries to our implemented database, we feel that having a mock database connection for testing purpose is probably overkill but the way we have our testing framework setup, we can easily move to a mock database with an addition of a configuration file should we require one in the future.
System acceptance testing will is anticipated to require 4-5 test cycles to make sure we're meeting the needs of Coast Capital. Firefox and IE11 have their own specifications and sets of challenges which will require their own test cycles individually. Furthermore system integration is an ongoing process that will not have a completed test cycle associated with it until the MVP is close to completion.
The anticipated criteria to move forward is to test the main functionalities first and make sure they are in working order before making the tests more robust, and testing edge cases as well. After that, once the MVP is done, we will move forward with creating test cases for stretch goals.

Responsibilities:

- **Anushka** will be writing test scripts using Selenium to test all the UI components in an automated browser for both IE11 and Firefox.
- **Maia** will be writing tests with mocked APIs to test the functionality of the API calls to the backend. The server will be mocked using Nock. The tests themselves will be written with Chai syntaxing and functions in Mocha testing environment.
- **Shrey** will be writing tests using Mocha to test the basic functions of the front end that deal with the current state of information before passing it to the API.
- **Stephen** will be writing unit tests using JUnit to test the basic functionality of the back-end web server.
- **Vaastav** in addition to JUnit tests to test the back-end server, will also be testing the recoverability as well as the capacity of the Database.
- **Everyone** will also be conducting manual testing for user acceptance testing and code reviews and test driven development means our system will be tested continuously by everyone as well.

Environmental setup:

Framework	Description	Version
Selenium	Selenium is primarily used for automating web applications for testing purposes. - Firefox driver (for Selenium) - IE11 driver (for Selenium)	3.6.0 3.6.0 3.4.0
Mocha	Mocha is a test runner for JS. In order to set it up, we installed it through NPM and wrote an NPM script for it to run the tests in our ‘test’ folder. The script can be called through terminal or IDEs that handle NPM scripts (like WebStorm).	^3.5.3
Chai	Chai provides extra functionalities for Mocha. Just like Mocha, it needed to be installed through NPM and imported into the test files.	^3.5.0
Nock	Nock provides the ability to set up a fake server which responds to API calls with the responses we need. Just like Mocha and Chai, it needed to be installed through NPM and imported to the test files. We then configured API paths it needed to respond to, and the responses it had to return depending on the path and query.	^9.0.27

JUnit	JUnit is a simple framework to write repeatable tests. In order to set it up, we added it as a dependency in Maven. JUnit testing framework is designed for running Unit Tests that test out small components/units and their basic functionality. These tests can be run through either via maven scripts or using Java capable IDEs like IntelliJ or Eclipse	5.0.0

B. Functional Test Plan

	Test component	Front-end Testing Technique Description
1.1	Login	**Selenium**:** For the UI, we will be using a black-box automated testing tool to test out the buttons on the login screen. Forgetting to code for a particular UI component will risk a component going untested. The primary use for Selenium is User Acceptance Testing, however it can also be used to check System Acceptance Testing to make sure sure the requirements of the system are being met (e.g. do these components work in Firefox vs. IE11). This will require one script for all test components it's required for.* **Mocha:** We will be using white-box automated testing method to test if the user was able to login successfully and whether they are an admin or not. Mocha is really useful for writing asynchronous unit tests for Javascript. However, a major risk is accidentally writing evergreen tests, which never fail even if the code is broken. There will be one script testing all functionalities in one component. **Manual testing:** Manual testing takes care of the risk garnered from Selenium, however, it has the same risk, with Selenium as backup. It is mainly used for User Acceptance Testing and is a form of black-box testing done to check the validity of an application. This requires no scripts.
2.1	Adding contractors	To test if the contractor was added to the database successfully or not, we will be using the same methods with the same approach and risks as mentioned above.
3.1	Editing contractors	To test if the user was able to edit the contractor's information successfully or not, we will be using the same methods with the same approach and risks as mentioned above.
4.1	Data filtering	To test if the user is able to view the filtered data successfully or not, we will be using the same methods with the same approach and risks as mentioned above.

5.1	Managing filters for Data filtering	To test if the user is able to toggle data filters successfully or not, we will be using the same methods with the same approach and risks as mentioned above.
6.1	Graphical representation of trending reports	To test if the user is able to view and filter trending reports successfully or not, we will be using the same methods with the same approach and risks as mentioned above.
7.1	Managing skills table	To test if the admin is able to view and edit the skills table successfully or not, we will be using the same methods with the same approach and risks as mentioned above.
8.1	Visualization of FX table	To test if the admin is able to view the FX table successfully or not, we will be using the same methods with the same approach and risks as mentioned above.
9.1	Managing HR Pay equivalent table	To test if the admin is able to view and edit the HR Pay equivalent table successfully or not, we will be using the same methods with the same approach and risks as mentioned above.
10.1	Managing HR role table	To test if the admin is able to view and edit the HR Role table successfully or not, we will be using the same methods with the same approach and risks as mentioned above.
11.1	Managing hiring managers	To test if the admin is able to add, view and edit the hiring managers successfully or not, we will be using the same methods with the same approach and risks as mentioned above.
12.1	Managing user permissions	To test if the admin is able to add, view and edit the user permissions successfully or not, we will be using the same methods with the same approach and risks as mentioned above.
13.1	Adding new users	To test if the admin is able to add new users to the system successfully or not, we will be using the same methods with the same approach and risks as mentioned above.

*Multiple tests can be written in one test script for Selenium
**Selenium scripts can only be written once components are completed

	Test component	**Back-end Testing Technique**	
		Test	**Description**
1.2	Login	JUnit	We will be using white-box automated testing method to test whether the user exists in the database and the user permissions they have. The major risk for this kind of testing is that these tests are written by the developers

			themselves and if they don't catch a certain edge case then it can go untested as JUnit can only check for cases that we write tests for. This type of testing is being used to test individual units in the application system as well as to assess system integration, and system acceptance testing. Each component will have 1-3 scripts associated with it.
2.2	Adding Contractors	JUnit	We will be using the same methods with the same risks and approach to test whether the contractor was added to the database or not.
3.2	Editing Contractors	JUnit	We will be using the same methods with the same risks and approach to test whether the contractor's information was updated in the database or not.
7.2	Managing Skills Table	JUnit	We will be using the same methods with the same risks and approach to test whether the skills were updated in the database or not.
9.2	Managing HR Pay Equivalent Table	JUnit	We will be using the same methods with the same risks and approach to test whether the HR Pay Equivalents were updated in the database or not.
10.2	Managing HR Role table	JUnit	We will be using the same methods with the same risks and approach to test whether the HR Roles were updated in the database or not.
12.2	Managing user permissions	JUnit	We will be using the same methods with the same risks and approach to test whether the user permissions were updated in the database or not.
13.2	Adding new users	JUnit	We will be using the same methods with the same risks and approach to test whether new users were added to the database or not.
14.2	Simultaneous Edit Collisions	Manual	We will be using manual white-box testing and this will also be part of User Acceptance Testing.The risk with this type of testing is that it is resource heavy as it requires multiple people working on the same thing at the same time and requires perfect coordination between the people testing it. If there is a lack of communication, then this testing can fail.

C. Non Functional testing

	Component	Test Description
15.3	Running interface on IE11	**Selenium:** We will be using a black-box testing tool to automate browsers in order to test our application on IE11. It will go through the application and test all the UI components, which will therefore show that the system works on IE11. The primary use for Selenium is User Acceptance Testing, however it can also be used to check System Acceptance Testing to make sure sure the requirements of the system are being met (e.g. do these components work in IE11). This will require one script for all test components it's required for. The risk of using Selenium is that the IE Driver does not specify that it is checking the compatibility for IE11, just that it can run on IE. **Manual testing:** This is a form of black-box testing that can be used for user acceptance testing. This can be used to check if the web application can run on IE11. The risk is human error.
16.3	Running interface on Firefox	The same two methods described for IE11 will be used for Firefox.
17.3	Recoverability of system	Testing the recoverability of the system will require manual black box testing. It will be 1 manual test script. This risk associated with this type of testing is black box as we don't really know the implementation details of the recovery system AWS use for the backup of the databases they host.
18.3	Capacity needs	Testing the capacity needs of the system will also require manual black box testing. It will be 1 manual test script. The risks are the same as highlighted above
19.3	Performance	Testing the performance of the system will be manual white box testing. It will be a few manual test scripts. This will be part of User Acceptance Testing as this will essentially test whether the users can run the system in real-life situations. A risk is that the performance can vary at times due to workload on the server so it can be really hard to test real-life scenarios. So, the test results may vary and may not always be accurately testing the performance of the system.

D. Test scripts

The reference number indicates which test component the test script description is for.
The appendix number indicates where you can find the test script for the component.

Ref. No.	**Test Script Approach Summary**	**Appendix Reference #**
1.1	Test login API calls - Input: user's username and password - Output: the response from the server which includes success/failure message, permissions, and username - Test will have to cover: - Successful and unsuccessful calls - Handling different user permissions - Handling error messages Framework: Mocha, Chai, Nock	A.1.1 A.1.7
2.1 3.1	Test contractor information API calls - View - Input: none - Output: the list of contractors in the database - Add - Input: the information of a new contractor - Output: a success/fail message - Edit - Input: the edited information of an existing contractor - Output: a success/fail message Framework: Mocha, Chai, Nock	A.1.2 A.1.7
12.1 13.1	Test user management API calls - Add - Input: the information of a new user - Output: a success/fail message - Edit - Input: the edited information of an existing user - Output: a success/fail message - Delete - Input: the username of the user that will be deleted - Output: a success/fail message Framework: Mocha, Chai, Nock	A.1.3 A.1.7
9.1	Test HR pay equivalent table API calls - Add	A.1.4 A.1.7

	- Input: the information of a new tier - Output: a success/fail message - Edit - Input: the edited information of an existing tier - Output: a success/fail message - Delete - Input: the ID of the tier that will be deleted - Output: a success/fail message Framework: Mocha, Chai, Nock	
10.1	Test HR role API calls - Add - Input: the information of a new role - Output: a success/fail message - Edit - Input: the edited information of an existing role - Output: a success/fail message - Delete - Input: the ID of the role that will be deleted - Output: a success/fail message Framework: Mocha, Chai, Nock	A.1.5 A.1.7
11.1	Test hiring managers API calls - Add - Input: the information of a new hiring manager - Output: a success/fail message - Edit - Input: the edited information of an existing hiring manager - Output: a success/fail message - Delete - Input: the ID of the hiring manager that will be deleted - Output: a success/fail message Framework: Mocha, Chai, Nock	A.1.6 A.1.7
1.2	Test login back-end functionality - Input: username and password - Output: login response detailing if the login was successful, the user permissions, and the username - Automation Method: JUnit - Test will cover: - Invalid login credentials - Different valid login credentials with different permissions	B.2.1 B.2.2 B.2.3

2.2 3.2	Test contractor information back-end functionality - Automation Method: JUnit - Add - Input: The information of a new contractor - Output: Whether the contractor was added successfully or not - Edit - Input: The information of an existing contractor - Output: Whether the contractor was edited successfully or not - View - Input: None - Output: The information of all contractors in the database	B.2.4 B.2.5 B.2.6
12.2 13.2	Test user management back-end functionality - Automation Method: JUnit - Add - Input: Information of a new user - Output: Whether addition was successful - Edit - Input: The edited information of an existing user - Output: Whether the edit was successful - Delete - Input: Information of an existing user - Output: Whether the deletion was successful	B.2.7 B.2.8 B.2.9
9.2	Test HR pay equivalent back-end functionality - Automation Method: JUnit - View - Input: None - Output: All the HR pay equivalences - Add - Input: A new HR Pay equivalence - Output: Whether the addition was successful - Edit - Input: The edited information of an existing HR pay equivalence tier - Output: Whether the edit was successful - Delete - Input: An existing HR Pay equivalence tier to delete - Output: Whether the delete was successful	B.2.10 B.2.11

10.2	Test HR Role back-end functionality - Automation Method: JUnit - View - Input: None - Output: All the HR Roles - Add - Input: A newly defined HR Role - Output: Whether the addition was successful - Edit - Input: An edited existing HR Role - Output: Whether the edit was successful - Delete - Input: An existing HR Role to delete - Output: Whether the delete was successful	B.2.12
14.2	Test Simultaneous Edit Collisions - This will test the fact that only 1 user at a time is allowed to edit the same row of data - Input : Multiple users will try to edit the same row - Output : Success or Failure in editing the row	B.2.13
18.3	Test Capacity - Volume of Data Stored in Database - Input : Large amount of data to be added to the database - Output : Querying the database to verify if all the data was successfully added	C.3.1
17.3	Test Recoverability - Delete & Restore - Input : Remove some subset of the data. - Output : Query the database to verify that the data was removed - Input : Restore the data from the back-up - Output : Query the database to verify that the removed data was correctly restored	C.3.2
19.3	Test Performance - Getting all the data - Input : Query to back-end to get all the contractor records of all the contractors - Output : All the relevant data in an acceptable period of time	C.3.3

E. Security testing and Test Data approach

Security Testing:

Test	Test Description
Protecting sensitive information	We will be using https for a secure connection and encrypt the data with SSL to ensure that all sensitive information is not compromised. To test this we would be using Wireshark to analyze the packets going from the back-end to the frontend and checking that the encrypted data looks nothing like the unencrypted data. This will again be a manual form of testing.
Preventing SQL injections	To test the prevention of SQL injections, a test injection script can be written in a user field. For example, an injection script such as “test username; DROP TABLE Users” written into any free form field will test if inputs are sanitized properly to prevent the end user from executing SQL commands. This will be completed manually via writing malicious commands into user fields. Furthermore, this will also be tested via unit testing of our back-end and unit testing of our front-end.
User permissions	Please refer to Section B, component 1.1 and 12.1

Test Data Approach

- Sample data has already been provided to us from Coast Capital which we would be using for testing.
- Apart from the sample data provided, for testing the capacity we would be writing a function that generates random data for us to use

F. Regression testing needs

We will be including the following tests in our regression pack:

- Preventing SQL injections as we feel that is one of major important security tests that we have in our test suite.
- Add/Edit Contractor tests (API and backend) are also a major important component of our application so the tests for those will be included in the regression test pack (A.1.2 & B.2.5 & B.2.6).

- Filtering tests will also be included in the regression pack as filtering is another major aspect of our application.
- Login tests will also be included in our regression pack as our login component is the gateway to the system and it will be critical failure if our login component stops functioning and users are unable to login to the system.

Appendix A (Front-end API Testing)

D. Test Script Snippets	
1.1	`it('Should return correct response on successful call - normal user permissions', () => {` `  nock('https://localhost:8443')` `  .get('/login')` `  .query({` `  username: 'user',` `  password: 'user'` `  })` `  .delay(0)` `  .reply(200, {` `  'error': false,` `  'errorMessage': null,` `  'loginSuccessful': true,` `  'username': 'user',` `  'permissions': 'write'` `  });` `  const expectedActions = [` `  {` `  type: TYPES.TOGGLE_LOADING,` `  isLoading: true` `  },` `  {` `  type: TYPES.TOGGLE_LOADING,` `  isLoading: false` `  },` `  {` `  type: TYPES.LOGIN,` `  username: 'user',` `  isAdmin: false` `  }` `  ];` `  return store.dispatch(loginUser('user', 'user')).then(() => {` `  expect(store.getActions()).to.deep.equal(expectedActions);` `  }).catch((err) => {` `  expect.fail(err);` `  });` `});`
1.2	`it('Should get the contractors list on successful return', () => {` `  const data = [` `  {` `  id: 1,` `  name: 'example 1'` `  },` `  {` `  id: 2,` `  name: 'example 2'` `  }` `  ];` `  nock('https://localhost:8443')` `  .get('contractors/view')` `  .delay(0)` `  .reply(200, {` `  'error': false,` `  'errorMessage': null,` `  'data': data` `  });`

```
  const expectedActions = [
    {
      type: TYPES.TOGGLE_LOADING,
      isLoading: true
    },
    {
      type: TYPES.TOGGLE_LOADING,
      isLoading: false
    },
    {
      type: TYPES.VIEW_CONTRACTORS,
      data
    }
  ];
  return store.dispatch(viewContractors()).then(() => {
    expect(store.getActions()).to.deep.equal(expectedActions);
  }).catch((err) => {
    expect.fail(err);
  });
});
```

1.3

```
it('Should respond properly when adding a new user successfully', () => {
  const data = { username: 'newUsername', password: 'newPassword' };
  nock('https://localhost:8443')
    .post('users/add')
    .send(data)
    .delay(0)
    .reply(200, {
      'error': false,
      'errorMessage': null
    });
  const expectedActions = [
    {
      type: TYPES.TOGGLE_LOADING,
      isLoading: true
    },
    {
      type: TYPES.TOGGLE_LOADING,
      isLoading: false
    },
    {
      type: TYPES.ADD_USER
    }
  ];
  return store.dispatch(addUser(data)).then(() => {
    expect(store.getActions()).to.deep.equal(expectedActions);
  }).catch((err) => {
    expect.fail(err);
  });
});
```

1.4

```
it('Should respond properly when adding a new tier', () => {
  const data = { name: 'name test', info: 'info test' };
  nock('https://localhost:8443')
    .post('pay/add')
    .send(data)
    .delay(0)
    .reply(200, {
      'error': false,
      'errorMessage': null
    });
  const expectedActions = [
    {
      type: TYPES.TOGGLE_LOADING,
```

	`` isLoading: true }, { type: TYPES.TOGGLE_LOADING, isLoading: false }, { type: TYPES.ADD_TIER }]; return store.dispatch(addPayTier(data)).then(() => { expect(store.getActions()).to.deep.equal(expectedActions); }).catch((err) => { expect.fail(err); }); }); ```
1.5	``` it('Should respond properly when adding a new role', () => { const data = { name: 'name test', info: 'info test' }; nock('https://localhost:8443') .post('role/add') .send(data) .delay(0) .reply(200, { 'error': false, 'errorMessage': null }); const expectedActions = [{ type: TYPES.TOGGLE_LOADING, isLoading: true }, { type: TYPES.TOGGLE_LOADING, isLoading: false }, { type: TYPES.ADD_ROLE }]; return store.dispatch(addRole(data)).then(() => { expect(store.getActions()).to.deep.equal(expectedActions); }).catch((err) => { expect.fail(err); }); }); ```
1.6	``` it('Should respond properly when adding a new hiring manager', () => { const data = { name: 'name test', info: 'info test' }; nock('https://localhost:8443') .post('manager/add') .send(data) .delay(0) .reply(200, { 'error': false, 'errorMessage': null }); const expectedActions = [{ type: TYPES.TOGGLE_LOADING, isLoading: true }, { type: TYPES.TOGGLE_LOADING, isLoading: false ```

```
    },
    {
      type: TYPES.ADD_HIRING-MANAGER
    }
  ];
  return store.dispatch(addManager(data)).then(() => {
    expect(store.getActions()).to.deep.equal(expectedActions);
  }).catch((err) => {
    expect.fail(err);
  });
});
```

1.7 An example of a failed API call test

```
it('Should return correct error on failure', () => {
  const errorMessage = 'test error message';
  nock('https://localhost:8443')
    .post(endpoint)
    .send(data)
    .delay(0)
    .reply(200, {
      'error': true,
      'errorMessage': errorMessage
    });
  const expectedActions = [
    {
      type: TYPES.TOGGLE_LOADING,
      isLoading: true
    },
    {
      type: TYPES.TOGGLE_LOADING,
      isLoading: false
    },
    {
      type: TYPES.CALL_FAILED,
      error: errorMessage
    }
  ];
  return store.dispatch(callApi(data)).then(() => {
    expect.fail('Call should have failed');
  }).catch((err) => {
    expect(err).to.equal(errorMessage);
    expect(store.getActions()).to.deep.equal(expectedActions);
  });
});
```

Appendix B (Back-end Testing)

D. Test Script Snippets	
2.1	`void loginFailure() {` `  String LOGIN_INCORRECT_RESPONSE = "Incorrect username or password";` `  String exampleUsenname = "exUsername";` `  String examplePassword = "exPassword";` `  LoginResponse response = controller.login(exampleUsenname, examplePassword);` `  *assertEquals*(LOGIN_INCORRECT_RESPONSE,response.getErrorMessage());` `}`
2.2	`void loginSuccessAdmin() {` `  final String EXPECTED_PERMISSIONS = "admin";` `  String adminUsername = "admin";` `  String adminPassword = "admin";` `  LoginResponse response = controller.login(adminUsername, adminPassword);` `  *assertEquals*(EXPECTED_PERMISSIONS, response.getPermissions());` `}`
2.3	`void loginSuccessNonAdmin() {` `  final String EXPECTED_PERMISSIONS = "write";` `  String username = "user";` `  String userPassword = "user";` `  LoginResponse response = controller.login(username, userPassword);` `  *assertEquals*(EXPECTED_PERMISSIONS, response.getPermissions());` `}`
2.4	`void contractorsTest() {` `  ContractorsResponse response = contractorController.contractors();` `  *assertFalse*(response.isError());` `  *assertFalse*(response.getContractors().isEmpty());` `}`
2.5	`void addContractorsTest() {` `  Contractor contractor = new Contractor(UUID.*randomUUID*().toString(), "Test first name", "test last name", "ex agency source", "active", true);` `  Response response = contractorController.addContractor(contractor);` `  *assertFalse*(response.isError());` `}`
2.6	`void editContractorsTest() {` `  final int FIRST = 0;` `  Contractor contractor = contractorController.contractors().getContractors().get(FIRST);` `  contractor.setAgencySource("Test New Agency Source");` `  Response response = contractorController.editContractor(contractor);`

	`assertFalse(response.isError());` `}`
2.7	`void addUserTest() {` `  User user = new User("testUsername", "testPassword", "none");` `  Response response = controller.addUser(user);` `  assertFalse(response.isError());` `  assertTrue(controller.users().getUsers().contains(user));` `}`
2.8	`void editUserTest() {` `  final int FIRST = 0;` `  List<User> allUsers = controller.users().getUsers();` `  User user = allUsers.get(FIRST);` `  user.setPermissions("write");` `  Response response = controller.editUser(user);` `  assertFalse(response.isError());` `  assertEquals(controller.users().getUsers().get(FIRST), user);` `}`
2.9	`void deleteUserTest() {` `  final int FIRST = 0;` `  List<User> allUsers = controller.users().getUsers();` `  User user = allUsers.get(FIRST);` `  Response response = controller.deleteUser(user);` `  assertFalse(response.isError());` `  assertFalse(controller.users().getUsers().contains(user));` `}`
2.10	`void viewPayGradesTest(){` `  HRPayGradeResponse response = hrPayGradeController.paygrades();` `  assertFalse(response.getPayGrades().isEmpty());` `}`
2.11	`void addPayGradeTest() {` `  HRPayGrade payGrade = new HRPayGrade("newPayGrade", 0, 100);` `  Response response = hrPayGradeController.addPayGrade(payGrade);` `  assertFalse(response.isError());` `}`
2.12	`void viewHRPositionsTest(){` `  HRPositionRoleResponse response =  hrPositionRoleController.hrroles();` `  assertFalse(response.isError());` `  assertFalse(response.getHrPositionRoles().isEmpty());` `}`
2.13	`Here is the sequence of events for running this test:` `1. Multiple users to log on to the system from different instances of the front-end` `2. Select a row in Contractor Records table that they want to edit` `3. Both person should try to edit the record and save it` `4. The test will pass if only 1 user is able to successfully save it while the other gets an error message`

Appendix C (Non-functional Testing)

D. Test Script Snippets	
3.1	Here is the sequence of events for testing the capacity of the database 1. Generate random valid data 2. Add it to the database using some mass import function (Needs to be implemented as of now) 3. Query the database to view the data 4. The test passes if the data we get when querying the database is the full set we added during the mass import
3.2	Here is the sequence of events for testing the recoverability of the database 1. Create a backup of the existing database 2. Delete some data from the database 3. Query the tables to verify that the data has been deleted 4. Restore the original data back to the table 5. Query the tables to verify that the data has been restored.
3.3	Testing the performance is basically a part of User Acceptance Testing. The sequence of events are expected to be as follows: 1. Log in as a user in the frontend 2. Perform some query or action that requires the front-end to do some time consuming work like either send a request for a big amount of data to the backend or performs filtering on a large amount of data. 3. The action is completed. 4. The test passes if the user deems the amount of time taken to complete the action is acceptable.