Given the standalone nature of my project, I am sure that I did not utilize unit testing and code coverage in the same manner as the other groups.

How I applied these tools;

- *JUnit Testing*

In hindsight, I likely should have adopted a different working process of designing unit tests prior to coding- as was likely intended. However, for better or worse, I was only concerned with working out the project's functionality alone, and simply tested my code repeatedly throughout the process of designing it. However, I do not believe this process would have gone much differently even if I had unit tests designed beforehand. After all; it still consists of building and testing the project live, encountering bugs, and going back to repair the code.

Instead, my unit tests came after- which may admittedly defeat much of the purpose. However, in beginning to develop unit tests nearer to the end of my project and its final implementations, it has helped me to resolve a number of bugs I had not encountered in traditional testing. In the ways that it required mulling back over existing code, it helped me to notice oversights such as;

- Database handling classes lacked checks for adding objects with incomplete information
- Database search methods were inefficiently implemented as for loops
- Table handling lacked checks for valid table numbers and capacities
- Reservation date/time verification registered errors when times within an hour of the current time- but on a different, valid day- were selected.
- Reservation deletion failed to toggle table availability

So, despite how it likely would have been more useful to implement sooner, I still found use in integrating JUnit tests. Besides these bug discoveries, however, my JUnit test cases only served to reinforce functionalities that I already knew to work from prior, manual testing. At the least, perhaps they establish grounds to expand tests if the project were continued.

- *JaCoCo Code Coverage*

…Was implemented as rather plainly as it could have been. JaCoCo was used as it is meant to be- as a tool for documenting code coverage for compiled classes- and I utilized it roughly as late as JUnit testing to simply double-check existing, functional code. It helped to clean up some code in final implementations and to rearrange specific code segments *(such as numerous error handling checks performed in each handler's database functions, allowing them to run only if the specified piece of data is even present)*.

Other than this, however, I will admit that I found the tool and documentation largely irrelevant. Unlike JUnit testing, it revealed little to me that I did not know prior about my program, and uncovered code segments are- at a certain point- unhelpful. Some only indicate branches of functions that, while necessary, simply aren't assessed during execution, or redundant code segments that I would nonetheless like to keep. This includes the likes of getRawDB() functions I included in some handlers for use during testing, or the numerous "setter" and "getter" methods of each object class; Accounts, Reservations, and Tables.

Tested Components:
All of my core Java classes had JUnit tests implemented- that is- all classes but my page servlets. This includes every class under the “Core” and “Database” packages;

- Core: *AccountManager, ReservationManager, TableManager*
- Database: *DatabaseHandler, Account, Reservation, Table*

For each manager, which are reliant on local database .csv files, my JUnit test cases create temporary database files for each class to access, and delete them after all tests within a given class are complete via the *@AfterClass* annotation.

Each test case runs off of a slew of *assertEquals()* calls, verifying that object information is added, edited, and deleted correctly. Though some class methods have no exactly tangible output- the core few (adding, editing, and deleting from the database) each return a String error message in the event of invalid information being passed. This error message is, in turn, used by the *assertEquals()* calls to verify that no error occurred- or that the proper error occurred in the case of test cases which test for proper error handling.

The TableHandler error test case, for example;

```
@Test
public void evaluateDBEditErrors() {
    TableManager tblmgr = new TableManager();

    String[] data = {"4", "100", "Gallery C", "false"};
    int tbl_num = 811;

    String error_msg = tblmgr.editTable(tbl_num, data);
    assertEquals(error_msg,  actual: "Given table #" + tbl_num + " does not exist.");

    tbl_num = 8;
    tblmgr.addTable(tbl_num, data);
    data = new String[] {"0", "100", "Gallery C", "false"};

    error_msg = tblmgr.editTable(tbl_num, data);
    assertEquals(error_msg,  actual: "Table capacity must be greater than 0.");
}
```


```
@Test
public void evaluateDBEditErrors() {
    TableManager tblmgr = new TableManager();

    String[] data = {"4", "100", "Gallery C", "false"};
    int tbl_num = 811;

    String error_msg = tblmgr.editTable(tbl_num, data);
    assertEquals(error_msg,  actual: "Given table #" + tbl_num + " does not exist.");

    tbl_num = 8;
    tblmgr.addTable(tbl_num, data);
    data = new String[] {"0", "100", "Gallery C", "false"};

    error_msg = tblmgr.editTable(tbl_num, data);
    assertEquals(error_msg,  actual: "Table capacity must be greater than 0.");
}
```

Code Coverage:
JaCoCo's coverage report generates under *build/site/jacoco/index.html*.
It reports a rough 97% coverage of code instructions, and ~84% of code branches. The missed code branches are predominantly the result of database editing and data validation methods included in each *'Handler'* class. To the best of my interpretation, these are niche error cases which are only checked for one of two possible values, or code functions included in methods that are simply never designed to be accessed.

- This includes the likes of *ReservationHandler's editReservation()* method, which includes a conditional to edit the account username tied to a reservation. However, the application is not designed to invoke this method at any time.

```
public String editReservation(int res_num, String[] data) {  9 usages
    if (!verifyResNum(res_num)) {
        return ("Given reservation #" + res_num + " does not exist.");
    }
    Reservation cur_res = ReservDB.get(res_num);

    if (!data[0].isEmpty()) { cur_res.setAccUsern(data[0]); }
```

- Or *AccountManager's verifyNonEmptyData()* method, which is simply never invoked to run through every possible empty data value, but which would function identically.

```
public boolean verifyNonEmptyData(String[] data) {  1 usage
    return (!data[0].isEmpty() && !data[1].isEmpty() && !data[2].isEmpty() && !data[3].isEmpty());
}
```

User Stories
Every user story, with the exception of user stories #1.7, #1.8, #2.2, and #2.5 were tested given that the system itself is designed to accommodate them.

User stories #1.7 and #1.8 both stipulate customers receiving email verifications for their reservations, and the extraneous technologies required to implement this functionality were beyond the scope of the time and understanding I had to finish this project. As such, neither was implemented, and neither can be accounted for in JUnit testing.

User stories #2.2 and #2.5 are leftovers from my previous group, which escape the scope of my project altogether. Given your assignment for my individual project was *only* a restaurant reservation system, without any functionality towards menu items, sales, inventory, etc- these user stories were entirely inapplicable to my project, and should not have been included among my user stories in the first place. This was little more than an oversight on my part.

For the remaining stories;

- User Story #1.1 was covered by *ReservationManager's verifyResDateTime()* method, and tested by *ReservationManagerTestCases' evaluateDBEditErrors()* method, which includes an assertion for error messages generated by invalid times or dates.
- User Story #1.4 was implemented, in a form, by the Profiles.jsp page, which calls on *ReservationManager's .getAccReservations()* method to view all reservations under their account. This method is checked by *ReservationManagerTestCases' evaluateDBEdit…()* methods, which utilize the method to find reservations for its editing. If the method did not function, neither would these test cases.
- User Story #1.5 was implemented by *ReservationManager's editReservation()* and *deleteReservation()* methods, which are checked, respectively, by *ReservationManagerTestCases' evaluateDB…()* and *evaluateDBDelete()* methods.
- User Story #2.1 was implemented, frankly, be the entire *AccountManager()* class- from its account database editing methods, login verification, and database accession. As such, it is covered by the *AccountManagerTestCases()* class.
- User Story #2.3 was implemented similarly by the *TableManager* class and tested by its respective *TableManagerTestCases()* class.
- User Story #2.4 was implemented both by *AccountManager's validateLogin()* method and *ReservationManager's* slew of database accession and editing methods; *editReservation…(), deleteReservation(), getReservations(),* etc. As such, this is validated through both the *ReservationManagerTestCases()* and *AccountManagerTestCases()* testing classes.

The remaining user stories #1.2, #1.3 and #1.6 were implemented by the project's JSP pages and nature as a locally hosted Tomcat web application, given they all pertain to the website's front-end qualities and accessibility. Given these are not Java classes, these are not tested by JUnit test cases.

Ant Script Console Outputs

If only to verify that, at least on my end, both JUnit testing and JaCoCo coverage with the build.xml Ant script as I have designed it actually works.

- *'ant test'* Output:

```
test:
[jacoco:coverage] Enhancing junit with coverage
    [junit] Running com.example.demo.Tests.AccountManagerTestCases
    [junit] Testsuite: com.example.demo.Tests.AccountManagerTestCases
    [junit] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.146 sec
    [junit] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.146 sec
    [junit]
    [junit] ------------- Standard Output ---------------
    [junit] C:\Users\trons\Desktop\School Work\Software Engineering - CSCI 472\Software Project\dbAccounts.csv
    [junit] Testing file cleaned!
    [junit] ------------- ---------------- ---------------
    [junit] Running com.example.demo.Tests.DBHandlerTestCases
    [junit] Testsuite: com.example.demo.Tests.DBHandlerTestCases
    [junit] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.095 sec
    [junit] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.095 sec
    [junit]
    [junit] ------------- Standard Output ---------------
    [junit] C:\Users\trons\Desktop\School Work\Software Engineering - CSCI 472\Software Project\TestDB.csv
    [junit] Testing file cleaned!
    [junit] ------------- ---------------- ---------------
    [junit] Running com.example.demo.Tests.DBObjectTestCases
    [junit] Testsuite: com.example.demo.Tests.DBObjectTestCases
    [junit] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.06 sec
    [junit] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.06 sec
    [junit]
    [junit] Running com.example.demo.Tests.ReservationManagerTestCases
    [junit] Testsuite: com.example.demo.Tests.ReservationManagerTestCases
    [junit] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.315 sec
    [junit] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.315 sec
    [junit]
    [junit] ------------- Standard Output ---------------
    [junit] C:\Users\trons\Desktop\School Work\Software Engineering - CSCI 472\Software Project\dbReservations.csv
    [junit] C:\Users\trons\Desktop\School Work\Software Engineering - CSCI 472\Software Project\dbTables.csv
    [junit] Testing file cleaned!
    [junit] Testing file cleaned!
```

```
    [junit] Testing file cleaned!
    [junit] ------------- ---------------- ---------------
    [junit] Running com.example.demo.Tests.TableManagerTestCases
    [junit] Testsuite: com.example.demo.Tests.TableManagerTestCases
    [junit] Tests run: 13, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.177 sec
    [junit] Tests run: 13, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.177 sec
    [junit]
    [junit] ------------- Standard Output ---------------
    [junit] C:\Users\trons\Desktop\School Work\Software Engineering - CSCI 472\Software Project\dbTables.csv
    [junit] Testing file cleaned!
    [junit] ------------- ---------------- ---------------
```

- *'ant report'* Output:

  Output above, as *ant report* depends on *ant test,* followed by;

```
report:
[jacoco:report] Loading execution data file C:\Users\trons\Desktop\School Work\Software Engineering - CSCI 472\Software Project\build\jacoco.exec
[jacoco:report] Writing bundle 'JaCoCo Ant' with 12 classes
[jacoco:report] To enable source code annotation class files for bundle 'JaCoCo Ant' have to be compiled with debug information.
```

- *ant report*'s generated JaCoCo report can be accessed by the command *'start build/site/jacoco/index.html'*, with the following content;

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Methods	Missed	Classes
com.example.demo.Tests		98%		84%	9	92	0	63	0	5
com.example.demo.Core		96%		84%	22	112	1	43	0	3
com.example.demo.Database		92%		100%	4	45	4	40	0	4
Total	143 of 6,111	97%	31 of 206	84%	35	249	5	146	0	12

com.example.demo.Core

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Methods	Missed	Classes
ReservationManager		95%		87%	7	41	1	14	0	1
AccountManager		95%		78%	8	33	0	14	0	1
TableManager		97%		84%	7	38	0	15	0	1
Total	43 of 1,124	96%	22 of 138	84%	22	112	1	43	0	3

com.example.demo.Database

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Methods	Missed	Classes
DatabaseHandler		89%		100%	0	9	0	4	0	1
Reservation		91%		n/a	2	14	2	14	0	1
Table		95%		n/a	1	12	1	12	0	1
Account		94%		n/a	1	10	1	10	0	1
Total	33 of 416	92%	0 of 10	100%	4	45	4	40	0	4

…and more as individually viewed. I do not feel the need to exhaustively provide screenshots of every inner document.