Final Project: Covid Tracker Application

Team Members:
1. Dhrumil Amish Shah [redacted]

Date:

2021-04-13

—

Subject:

Software Development Concepts

—

Professor:

Matthew Amy

Database Schema

Database **covid_tracker** contains four tables:

1. **mobile_device**
   - To store all mobile device hashes.
2. **test_outcome**
   - To store all COVID-19 tests (i.e., test hash, test date and test result).
3. **mobile_device_test_outcome**
   - To link **mobile_device** and **test_outcome** tables (i.e., mobile device id and test outcome id).
4. **contact**
   - To store all contacts between mobile devices (i.e., contact date and duration, person one and person two ids and contact notified flag).

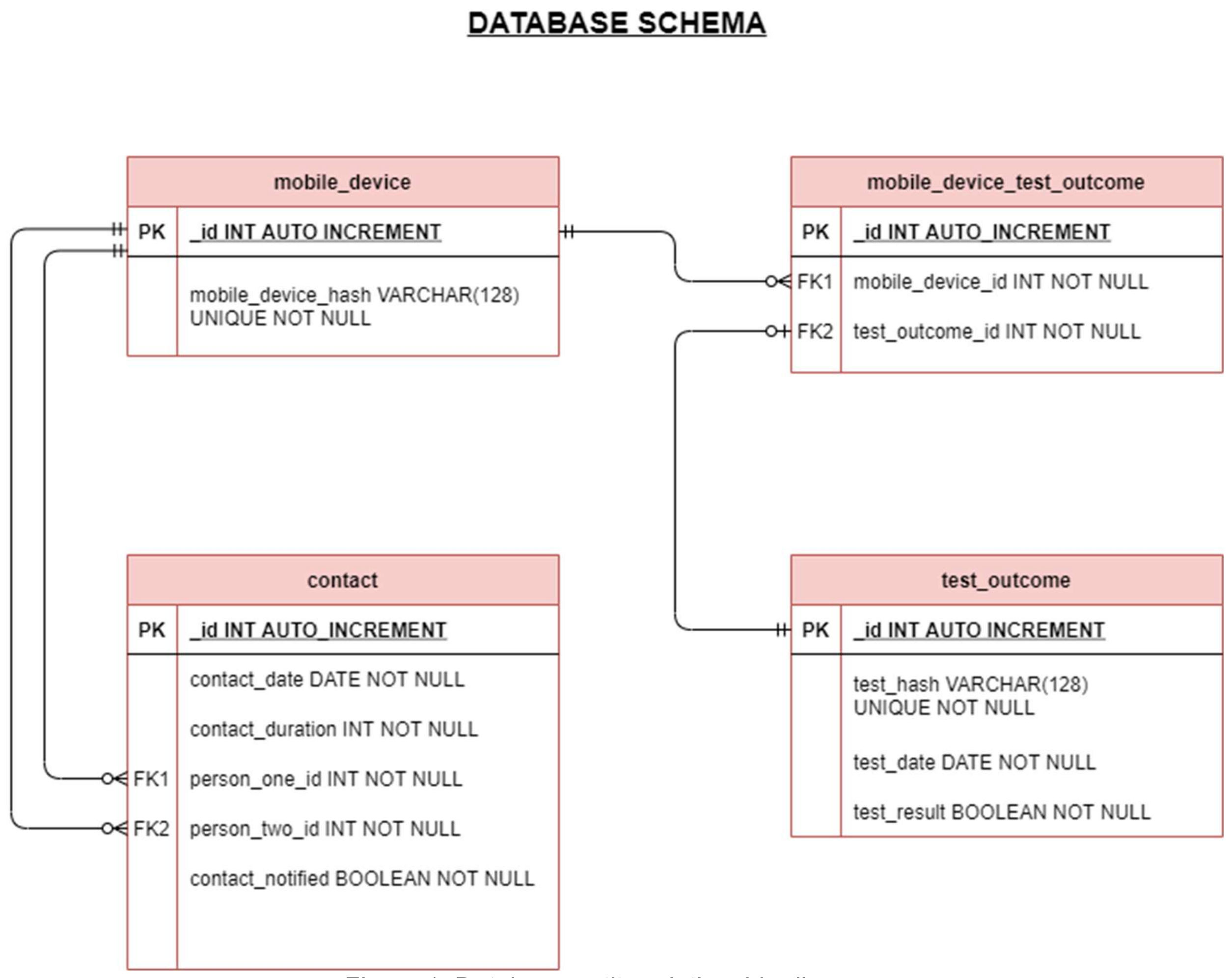

DATABASE SCHEMA
mobile_device
PK
_id INT AUTO INCREMENT
mobile_device_hash VARCHAR(128) UNIQUE NOT NULL
mobile_device_test_outcome
PK
_id INT AUTO_INCREMENT
FK1
mobile_device_id INT NOT NULL
FK2
test_outcome_id INT NOT NULL
contact
PK
_id INT AUTO_INCREMENT
contact_date DATE NOT NULL
contact_duration INT NOT NULL
FK1
person_one_id INT NOT NULL
FK2
person_two_id INT NOT NULL
contact_notified BOOLEAN NOT NULL
test_outcome
PK
_id INT AUTO INCREMENT
test_hash VARCHAR(128) UNIQUE NOT NULL
test_date DATE NOT NULL
test_result BOOLEAN NOT NULL

Figure 1: Database entity-relationship diagram

Assumptions

1. **recordTestResult()** method in Government class is called by COVID-19 testing agencies. They are responsible for uploading test results (i.e., unique test hash, sample collection date and test result) to the database (i.e., in table **test_result**) and give the positive COVID-19 result test hash to the user to identify the same test outcome and link it correctly to **mobile_device** and **mobile_device_test_result** tables.
2. **recordTestResult()** method in **Government** class must be called before **positiveTest()** method of **MobileDevice** class. The database is designed in a way that **mobile_device_test_result** table links **mobile_device** and **test_result** tables. Data in **test_result** table is to be uploaded first and then in **mobile_device** and **mobile_device_test_result** tables. Calling **positiveTest()** method before **recordTestResult()** would not link the positive test to the mobile device since no record exists.
3. To run JUnit test files multiple times without any issues, I truncate tables and create empty local XML files associated with respected **MobileDevice** before script execution and again truncate tables and delete XML files associated with **MobileDevice** after script execution. I am doing this to simplify the testing process. (In an actual environment, there is no need to perform this step).
4. The date difference between the contact date and the positive test date is absolute. So, it can be either plus 14 days or minus 14 days.
5. I have considered January 1, 2021 as integer 0. So consecutive dates will start from 1. Dates before January 1, 2021 are not considered.

XML Conversation Format

MobileDevice and **Government** have a standard XML format for conversation.
It is required for both of them to follow the same format so that communication happens without any issues.

XML DTD Schema

```
<!ELEMENT MobileDevice(ContactsList, TestHashesList)>
<!ELEMENT ContactsList(Contact*)>
<!ELEMENT TestHashesList(TestHash*)>
<!ELEMENT Contact(Individual, Date, Duration)>
<!ELEMENT Individual (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT Duration (#PCDATA)>
<!ELEMENT TestHash (#PCDATA)>
```

1. **MobileDevice** contains one element named **ContactsList**, followed by one element named **TestHashesList**.
2. **ContactsList** contains any number of **Contact** and **TestHashesList** contains any number of **TeshHash**. The * indicates that there can 0 or more elements.
3. **Contact** contains one element named **Individual**, followed by one element name **Date** and **Duration**.
4. **Individual** is a valid element name, and it contains parsed character data.
5. **Date** is a valid element name, and it contains parsed character data.

6. **Duration** is a valid element name, and it contains parsed character data
7. **TestHash** is a valid element name, and it contains parsed character data.

Sample XML

```
<MobileDevice>
        <ContactsList>
                <Contact>
                        <Individual>individual_hash</Individual>
                        <Date>date</Date>
                        <Duration>duration</Duration>
                <Contact>
                <Contact>
                        <Individual>individual_hash</Individual>
                        <Date>date</Date>
                        <Duration>duration</Duration>
                <Contact>

        </ContactsList>
        <TestHashesList>
                <TestHash>positive_hash_1</TestHash>
                <TestHash>positive_hash_2</TestHash>
                <TestHash>positive_hash_3</TestHash>
        </TestHashesList>
</MobileDevice>
```

Design Detailing of Solution

The solution contains two java files:

1. MobileDevice.java with public class **MobileDevice**.
2. Government.java with public class **Government**.

Public constructors and methods in **MobileDevice** class are as below:

1. **public MobileDevice(String configFile, Government contactTracer)**
   throws – IllegalArgumentException, RuntimeException.

   - This constructor gets the configFile and contactTracer as arguments. If arguments are not valid or content is not proper, it throws an exception (IllegalArgumentException for invalid argument or RuntimeException for any other error) otherwise, it instantiates the MobileDevice.
   - A local XML file is associated with the instantiated MobileDevice to store contact information and positive test hashes. The XML file preserves the data to survive MobileDevice instance deletion so that data can be synchronized afterwards. The XML file is created if not exists already.
   - The XML file name is device configuration hash (device_config_hash.xml) so that filename is unique for all devices.

2. **public String getMobileDeviceHash()**
   throws – RuntimeException.
   returns – MobileDevice configuration hash string.

   - This method gets the hash string of the MobileDevice configuration. It uses SHA-256 cryptographic algorithm to hash the MobileDevice configuration. If any error occurs, it throws RuntimeException.

3. **public boolean recordContact(String individual, int date, int duration)**
   throws – IllegalArgumentException, RuntimeException.
   returns – true if contact is added successfully in XML file associated with the MobileDevice otherwise false.

   - This method gets the individual MobileDevice configuration hash, contact date and contact duration as arguments. If arguments are not valid or content is not proper, it throws an exception (IllegalArgumentException for invalid argument or RuntimeException for any other error) otherwise, it records the contact information in the XML file associated with the MobileDevice. Each contact is recorded in the XML file as below:
     ```
     <Contact>
             <Individual>individual_hash</Individual>
             <Date>date</Date>
             <Duration>duration</Duration>
     <Contact>
     ```
   - Self contact cannot be recorded and false is returned by this method.

4. **public boolean positiveTest(String testHash)**
   throws – IllegalArgumentException, RuntimeException
   returns – true if test is recorded successfully otherwise false.

   - This method gets the testHash as the argument. If the argument is not valid or the content is not proper, it throws an exception (IllegalArgumentException for invalid argument or RuntimeException for any other error) otherwise, it records the test information in the XML file associated with the MobileDevice. Each test is recorded in the XML file as below:
     ```
     <TestHash>test_hash</TestHash>
     ```
   - Same test cannot be recorded multiple times and false is returned by this method.

5. **public boolean synchronizeData()**
   throws – RuntimeException.
   returns – true if MobileDevice has been near anyone diagnosed with COVID-19 in the 14 days otherwise false.

   - This method synchronizes the MobileDevice data with the Government periodically. Information is packaged in an XML string as shown above in the topic: XML Conversation Format and sent to the Government. If any error occurs, it throws RuntimeException.

Public constructors and methods in **Government** class are as below:

1. **public static Government getInstance(String configFile)**
   throws – IllegalArgumentException, RuntimeException.

   - This method gets the configFile as the argument. If the argument is not valid or content is not proper, it throws an exception (IllegalArgumentException for invalid argument or RuntimeException for any other error) otherwise, it instantiates the Government. Only one copy of Government is created throughout the application because it is designed using Singleton Design Principle.

2. **public boolean mobileContact(String initiator, String contactInfo)**
   throws – IllegalArgumentException, RuntimeException.
   returns – true if the initiator has been near anyone diagnosed with COVID-19 in the 14 days otherwise false.

   - This method gets the initiator string and contactInfo XML string as arguments. If arguments are not valid or any error occurs while storing information in the database or querying information from the database, it throws an exception (IllegalArgumentException for invalid argument or RuntimeException for any other error).
   - This method stores the contact information in the database. The caller of this method is identified by the initiator string with is the hash value of the caller's MobileDevice configuration hash string.
   - Queries are executed in a batch and this method ensures that atomicity is maintained. Commit is only made if all queries are executed successfully.

3. **public boolean recordTestResult(String testHash, int date, boolean result)**
   throws – IllegalArgumentException, RuntimeException.
   returns – true if the report is recorded successfully.

   - This method gets the testHash string, sample collection date and report result as arguments. If arguments are not valid or any error occurs while storing information in the database, it throws an exception (IllegalArgumentException for invalid argument or RuntimeException for any other error).
   - This method stores the record in the database. Returns true if recorded successfully.

4. **public int findGatherings(int date, int minSize, int minTime, float density)**
   throws – IllegalArgumentException, RuntimeException.
   returns – number of gatherings found.

   This method gets the date, minSize, minTime and density as arguments. If arguments are not valid or any error occurs while storing information in the database, it throws an exception (IllegalArgumentException for invalid argument or RuntimeException for any other error).

   I have compared the density with greater or equals to condition instead of greater than. This is because for later case if found gatherings have a density of 1 and argument density is 1, then condition (1 > 1) would be false but for greater or equals to (1 >= 1) would be true.

I have implemented this in a way that a single user can can be present in multiple gatherings. For example, I queried pairs for a particular day where all meetings are greater than provided time, and after removing duplicates, I have something like this.

```
1  2
1  3
1  4
2  3
2  4
3  4
4  5
4  6
5  6
```

So total individuals: 1, 2, 3, 4, 5, 6

In my case, the answer is
Gathering 1: (1, 2), (1, 3), (2, 3), (1, 4), (2, 4), (3, 4)
Individuals: 1, 2, 3, 4

Gathering 2: (4, 5), (4, 6), (5, 6)
Individuals: 4, 5, 6

Assuming for both the gatherings, c/m is greater or equal to density, I will report 2 gatherings.

I allow storing of contacts of two same individuals on the same date because two individuals may meet multiple times on the same date. In this method, I SUM the duration for the same individual pairs so that duplicate pairs are removed.

Things I followed while developing this application are as below:

1. I am using **Singleton Design Pattern** and **lazy initialization** to instantiate the **Government** class. It ensures that only one instance of the **Government** class is created throughout the application.
2. I am storing contacts and positive test hashes of a **MobileDevice** in a **local XML** file associated with the same **MobileDevice**. It ensures that even when the **MobileDevice** instance is deleted, the data is persisted locally so it can be synchronized afterwards. **XML** file preserves the data to survive **MobileDevice** instance deletion.
3. All constants used by the **MobileDevice** class are stored in **MobileDeviceConstant** class and, constants used by the **Government** class are stored in **GovernmentConstant** class. Both **MobileDeviceConstant** and **GovernmentConstant** are marked as private and static and are added as inner classes in their respective parent classes.
4. All database related constants and queries used by the **Government** class are stored in private static inner class **GoverenmentDatabase**.
5. Queries are executed in a way that **atomicity** is maintained. It means, either all queries are executed or none of them.
6. To reduce network calls, I am executing queries in batch. Also, queries are optimized to reduce multiple transactions and fail cases.

7. The database is designed by following **ACID** (Atomicity, Consistency, Isolation and Durability) properties and **normalization** techniques for consistency, scalability, and robustness.
8. If any error occurs inside a constructor or a method, it wraps the **main** cause(exception) inside **RuntimeException** and throws it to the user interface or command prompt. It is the responsibility of the user interface to handle such exceptions and perform required actions.
9. **RuntimeException** in java is an **unchecked** exception so, throws can be ignored from the method definition. It simplifies the testing process by avoiding the use of unnecessary try-catch.

JavaDoc Comments

The code is documented using **JavaDoc comments** using standard styles and predefined java documentation format. Explanation of constructors, methods, variables, and tests can be found commented within the code.

Test Plan and Strategy

I have implemented three **JUnit** test files to perform automated testing of the application.

1. **CovidTrackerValidationTest.java** validates the input parameters and boundary cases. There is a total of seven tests in this file.
2. **CovidTrackerFlowTest.java** validates the control flow and data flow cases. There is a total of six tests in this file.
3. **CovidTrackerBigTest.java** validates the control flow and data flow cases on large data. There is a total of nine tests in this file.

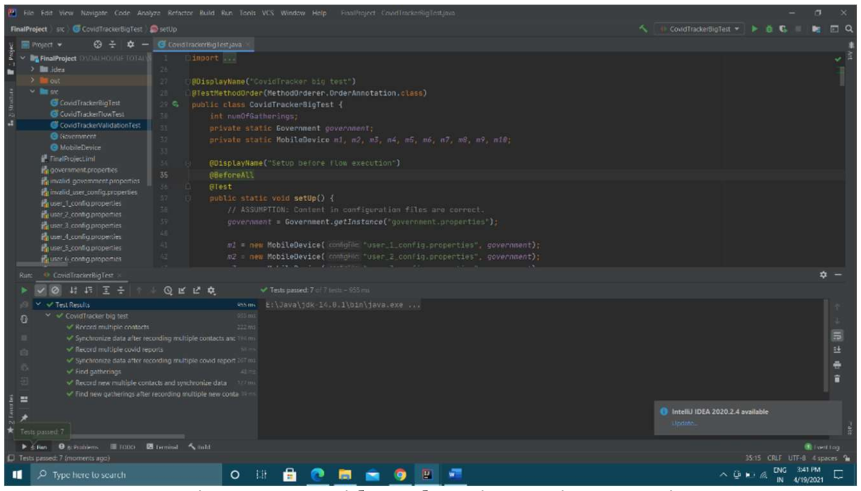
All testing scenarios are covered in these files. The automated testing strategy is convenient compared to testing by hand. Manual tasks like clearing the database and deleting files after running the test scripts are also automated so that tests can be run multiple times without breaking the code or the database.

All public methods and constructors are tested multiple times with different inputs to ensure the correct behaviour of the application. I decided on test cases in a way that all scenarios are covered. Testing focuses on both MobileDevice and Government class as well as on the database operations.

Instead of listing the test cases in PDF file and perform manual testing, automated testing makes things easier since we can run tests any time after code changes to ensure that new changes do not break the code. Further, we can achieve broader code coverage using automated testing.

Automated testing guarantees that crucial steps such as maintaining atomicity, parsing XML, creating a file, storing entries locally in a file, avoiding duplicate entries in a file and inside the database can be tested more thoroughly and makes the code more robust.

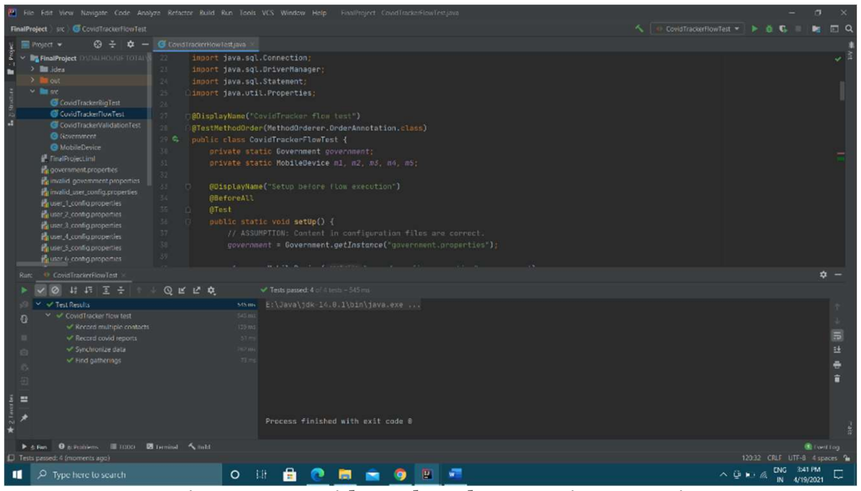
Below are the screenshots of successful JUnit files execution: (Next Page)



```
import ...

@DisplayName("CovidTracker big test")
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class CovidTrackerBigTest {
    int numOfGatherings;
    private static Government government;
    private static MobileDevice m1, m2, m3, m4, m5, m6, m7, m8, m9, m10;

    @DisplayName("Setup before flow execution")
    @BeforeAll
    @Test
    public static void setUp() {
        // ASSUMPTION: Content in configuration files are correct.
        government = Government.getInstance("government.properties");

        m1 = new MobileDevice( configFile: "user_1_config.properties", government);
        m2 = new MobileDevice( configFile: "user_2_config.properties", government);
```

```
Tests passed: 7 of 7 tests – 955 ms

Test Results
  CovidTracker big test
    Record multiple contacts
    Synchronize data after recording multiple contacts anc
    Record multiple covid reports
    Synchronize data after recording multiple covid report
    Find gatherings
    Record new multiple contacts and synchronize data
    Find new gatherings after recording multiple new conta

E:\Java\jdk-14.0.1\bin\java.exe ...
```

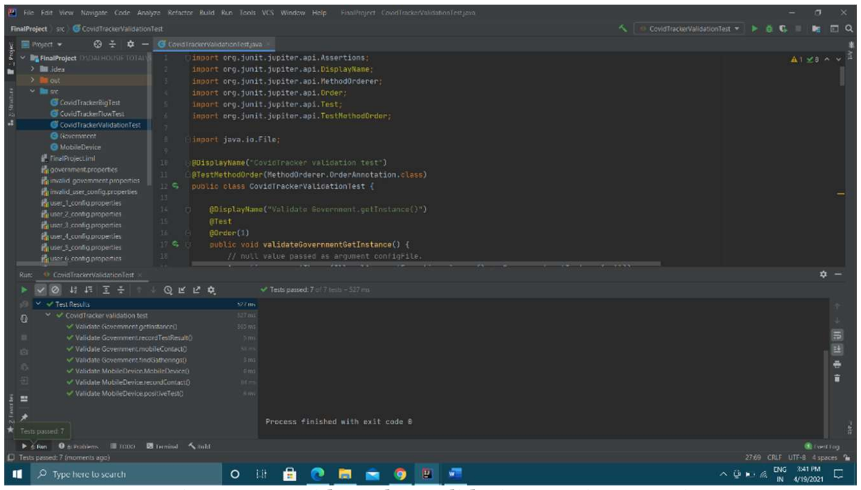
Figure 2: CovidTrackerBigTest.java JUnit



```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.util.Properties;

@DisplayName("CovidTracker flow test")
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class CovidTrackerFlowTest {
    private static Government government;
    private static MobileDevice m1, m2, m3, m4, m5;

    @DisplayName("Setup before flow execution")
    @BeforeAll
    @Test
    public static void setUp() {
        // ASSUMPTION: Content in configuration files are correct.
        government = Government.getInstance("government.properties");
```

```
Tests passed: 4 of 4 tests – 545 ms

Test Results
  CovidTracker flow test
    Record multiple contacts
    Record covid reports
    Synchronize data
    Find gatherings

E:\Java\jdk-14.0.1\bin\java.exe ...

Process finished with exit code 0
```

Figure 3: CovidTrackerFlowTest.java JUnit



```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.MethodOrderer;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;

import java.io.File;

@DisplayName("CovidTracker validation test")
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class CovidTrackerValidationTest {

    @DisplayName("Validate Government.getInstance()")
    @Test
    @Order(1)
    public void validateGovernmentGetInstance() {
        // null value passed as argument configFile.
```

```
Tests passed: 7 of 7 tests – 527 ms

Test Results
  CovidTracker validation test
    Validate Government.getInstance()
    Validate Government.recordTestResult()
    Validate Government.mobileContact()
    Validate Government.findGatherings()
    Validate MobileDevice.MobileDevice()
    Validate MobileDevice.recordContact()
    Validate MobileDevice.positiveTest()

Process finished with exit code 0
```

Figure 4: CovidTrackerValidationTest.java JUnit

Files submitted

One SQL script file:

- covid_tracker.sql (Creates database and tables)

Two Java solution files:

- Government.java (Java solution file)
- MobileDevice.java (Java solution file)

Three JUnit test files:

- CovidTrackerBigTest.java (JUnit test file 1)
- CovidTrackerFlowTest.java (JUnit test file 2)
- CovidTrackerValidationTest.java (JUnit test file 3)

One Government configuration file:

- government.properties (Government configuration file)

Twelve MobileDevice configuration files:

- user_1_config.properties (Mobile 1 configuration file)
- user_2_config.properties (Mobile 2 configuration file)
- user_3_config.properties (Mobile 3 configuration file)
- user_4_config.properties (Mobile 4 configuration file)
- user_5_config.properties (Mobile 5 configuration file)
- user_6_config.properties (Mobile 6 configuration file)
- user_7_config.properties (Mobile 7 configuration file)
- user_8_config.properties (Mobile 8 configuration file)
- user_9_config.properties (Mobile 9 configuration file)
- user_10_config.properties (Mobile 10 configuration file)
- user_11_config.properties (Mobile 11 configuration file)
- user_12_config.properties (Mobile 12 configuration file)

Invalid configuration files for testing:

- invalid_government.properties (Invalid Government configuration file)
- invalid_user_config.properties (Invalid Mobile configuration file)