Database testing practical tips

ANTON SMIRNOV

Database testing practical tips.

Learn to create a framework for automation testing fundamentals fast.

This version was published on 2021-05-19

Contact details:

- antony.s.smirnov@gmail.com

Related Websites:

Database testing practical tips: https://leanpub.com/databasetestingpracticaltips

Table of Contents

Introduction.

With rapid technological progress, it is no longer possible to store data on paper, desktops, or bookshelves. For these purposes, you will need too much paper and large areas for its storage. To replace this method of data storage came special data centers with large hard drives and were invented databases (DB), where large amounts of information can be stored for a long time, processed, and logically structured.

A database is a special structure for storing information, organizing, and sharing large amounts of data. They are used in various areas of the IT community: from simple web stores to space flight control centers.

To create, use, and manage a database, a special set of applications and system tools was created, which is called a database management system (DBMS).

As software testers, we often take for granted the fact that our application has a database behind it. We tend to focus on the visible, such as the user interface, or the application logic of the API when we are testing. But it's important to test the database as well.

We'll look at the pros and cons of database testing frameworks. Let's look at different approaches and strategies.

This book is based on more than 5+ years of experience in the field of test automation. During this time, a huge collection of solved questions has accumulated, and the problems and difficulties characteristic of many beginners have become clearly visible. In the course of working in different places, I have repeatedly had to create a framework for testing automation from scratch. It was obvious and reasonable for me to summarize this material in the form of a book that will help novice testers quickly build an automation testing framework on a project and avoid many annoying mistakes.

This book does not aim to fully disclose the entire subject area with all its nuances, so do not take it as a textbook or Handbook — for decades of development testing has accumulated such a volume of data that its formal presentation is not enough, and a dozen books.

Also, reading just this one book is not enough to become a "senior automated testing engineer". Then why do we need this book?

First, this book is worth reading if you are determined to engage in automated testing – it will be useful as a "very beginner" and have some experience in automation.

Secondly, this book can and should be used as reference material.

Thirdly, this book — a kind of "map", which has links to many external sources of information (which can be useful even experienced automation engineer), as well as many examples with explanations.

This book is not intended for people with high experience in test automation. From time to time, I use a learning approach and try to “chew” all the approaches and build the stages step by step.

Some people more experienced in software test automation also having may find it slow, boring, and monotonous.

This book is intended for people who first approach the creation of an automation testing framework, especially if their goal is to add automation to their test approach.

First of all, I wrote this book for a tester with experience in the field of “manual” software testing, the purpose of which is to move to a higher level in the tester career.

Summary:

We can safely say that this book is a kind of guide for beginners in the field of automation software testing.

I have a huge knowledge of the field of test automation. I also have quite a lot of experience building automation on a project from scratch. I have repeatedly had to develop and implement the framework of testing automation on projects.

The learning approach focuses on a huge chunk of theory on building the automation testing framework. The book also discusses the theory of test database automation in detail.

However, the direction of automation to support testing is no longer limited to testing, so this book is suitable for anyone who wants to improve the use of automation: managers, business analysts, users, and, of course, testers.

Testers use different approaches for testing on projects. I remember when I first started doing testing, I was drawing information from traditional books and was unnecessarily confused by some concepts that I rarely had to use. And most of the books, to my great regret, did not address the aspects and approaches to test automation. Most books on testing begin by showing how you can test a software product with basic approaches. But I do not consider the approaches and implementations of test automation at the testing stage.

My main goal is to help you start building an automation testing framework using a strategy and have the basic knowledge you need to do so.

This book focuses on theory rather than a lot of additional libraries, because once you have the basics, building a library and learning how to use it becomes a matter of reading the documentation.

This book is not an "exhaustive" introduction. This is a guide to getting started in building an automation testing framework. I focused on the examples.

I argue that in order to start implementing an automation testing framework, you need a basic set of knowledge in testing and management to start adding value to automation projects.

In fact, when I started creating the automation testing framework first, I used only the initial level of knowledge in the field of testing and development.
I also want the book to be small and accessible so that people actually read it and apply the approaches described in it in practice.

Acknowledgments.

This book was created as a “work in progress” on **leanpub.com**. My thanks go to everyone who bought the book in its early stages, this provided the continued motivation to create something that added value, and then spends the extra time needed to add polish and readability.
I am also grateful to every QA engineer that I have worked with who took the time to explain their approach. You helped me observe what a good QA engineer does and how they work. The fact that you were good, forced me to ‘up my game’ and improve both my coding and testing skills. All mistakes in this book are my fault.

Chapter 1. Theory of database testing.

With rapid technological progress, storing data on paper, desktops, or bookshelves is no longer possible. For these purposes, you will need too much paper and large areas for its storage.

To replace this method of data storage came special data centers with large hard drives and were invented databases (DB), where large amounts of information can be stored for a long time, processed, and logically structured.

A database is a special structure for storing information, organizing, and sharing large amounts of data. They are used in various areas of the IT community: from simple web stores to space flight control centers.

To create, use, and manage the database, a special set of applications and system tools was created called the database management system (Database Management System).

DMS is a special set of programs that are used to develop, display, fill in and modify information.

The most popular DBMS are Oracle, PostgreSQL, MongoDB, MySQL, and Microsoft SQL Server.

For any system that is embedded in the software, it is necessary to conduct a number of checks to minimize subsequent errors when using it as much as possible.

In various software, information is transferred from the user to the internal database and vice versa.

Checking the database health, first of all, consists of testing the performance that is associated with the database, as well as testing the reliability and structural integrity of information.

As a rule, both in testing and in development, we pay quite a lot of attention to the GUI part, because it is the user interface that is regularly in the public eye. However, it is very important to check the information that comes in the part that can rightfully be considered the heart of the application-the database.

Database testing is the testing of schemas, tables, triggers, operations for getting/adding information, and other parts and functions of the database under test. This can include creating complex queries to load/stress test the database and test its response rate. The uniformity of the data placed in the database is also considered part of such testing.

The database health check can be divided into three types, depending on the parameters and structure of the database:

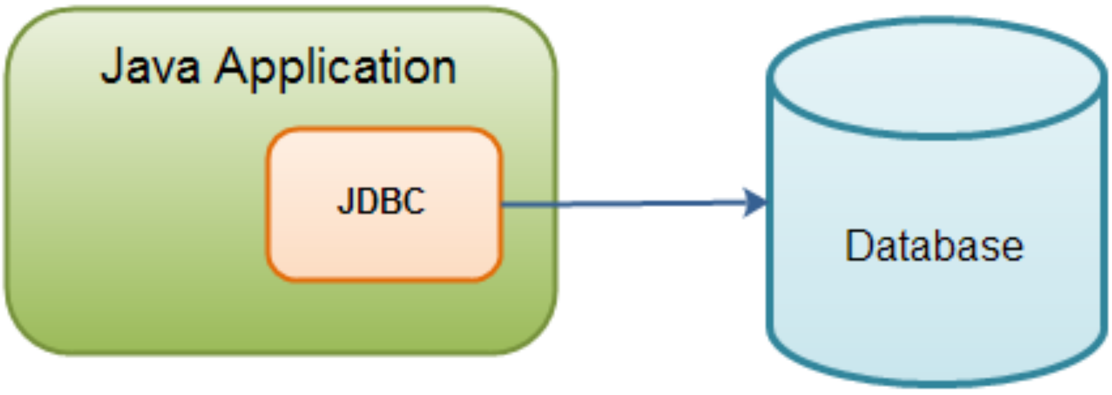
- Structural testing includes checking the elements that are used primarily for storing information and access to change which cannot be granted to ordinary users.
- Functional testing it involves verifying that the operations and transactions performed by end users meet the functional requirements described in the specification.
- Non-Functional it can be divided into such types as load testing, stress testing, usability testing, etc. Sometimes they also include security testing and compatibility testing.

Chapter 2. How to database testing using frameworks.

Computer applications are more complex these days with technologies like Android and also with lots of Smartphone apps. The more complex the front ends, the more intricate the back ends become.

So it is all the more important to learn about DB testing and be able to validate Databases effectively to ensure security and quality databases.

Sometimes a QA automation engineer needs to work with a database. You need to run queries, call stored procedures, and write some data. The standard for working with databases in java is **JDBC**.



Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access a database. It is a Java-based data access technology used for Java database connectivity. It is part of the Java Standard Edition platform, from Oracle Corporation. It provides methods to query and update data in a database and is oriented toward relational databases. A JDBC-to-ODBC bridge enables connections to any ODBC-accessible data source in the Java virtual machine (JVM) host environment.

Program code written using JDBC looks like this:

```
public class JDBCExample {
    private static final Logger LOGGER = Logger.getLogger("java.util.logger");
         public static void main(String[] args) {
        Connection con = null;
        Statement stmt = null;
        ResultSet resultSet = null;
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
            con = DriverManager.getConnection("jdbc:mysql://localhost:3306/words",
                    "words", "words");
            stmt = con.createStatement();
            resultSet = stmt.executeQuery("select * from word");
            while (resultSet.next()) {
                LOGGER.info("word id: " + resultSet.getLong(1) +
                        " spelling: " + resultSet.getString(2) +
                        " part of speech: " + resultSet.getString(3));
            }
        } catch (SQLException exception) {
            LOGGER.info("The method SQL is down" + exception);
        } catch (ClassNotFoundException ex) {
            LOGGER.info("The method SQL is down" + ex);
        } finally {
            try {
                resultSet.close();
            } catch (Exception e) {
            }
            try {
                stmt.close();
            } catch (Exception e) {
            }
            try {
                con.close();
            } catch (Exception e) {
            }
        }
    }
}
```

Writing such code is not correct, especially in a test framework that should be lightweight. What can be done about it? You need to use libraries that will make your life much easier.

1. **JDBI** is built on top of JDBC. If your database has a JDBC driver, you can use JDBI with it. JDBI improves JDBC's rough interface, providing a more natural Java database interface that is easy to bind to your domain data types. Unlike an ORM, we do not aim to provide a complete object-relational mapping framework — instead of that hidden complexity, we provide building blocks that allow you to construct the mapping between relations and objects as appropriate for your application.

The implementation of this library is similar to Groovy SQL. Using the JBDI library, you can write the following code:

```java
public class JDBIExample {

    public static void main(String[] args) {
        DataSource dataSource = JdbcConnectionPool.create("jdbc:h2:mem:test",
                "username",
                "password");
        DBI dbi = new DBI(dataSource);
        Handle handle = dbi.open();
        handle.execute("create table something (id int primary key, name varchar(100))");
        handle.execute("insert into something (id, name) values (?, ?)", 1, "Brian");
        String name = handle.createQuery("select name from something where id = :id")
                .bind("id", 1)
                .map(StringMapper.FIRST)
                .first();

        assertThat(name, equalTo("Brian"));
        handle.close();
    }
}
```

The code becomes more readable. Moreover, you can map queries to objects, which will make life even easier.

2. **jOOQ** a fluent API for typesafe SQL query construction and execution.

```java
public class JOOQExample {

    public static void main(String[] args) {
        select(KeyColumnUsage.CONSTRAINT_NAME, KeyColumnUsage.TABLE_NAME, KeyColumnUsage.COLUMN_NAME)
                .from(KEY_COLUMN_USAGE)
                .join(TABLE_CONSTRAINTS)
                .on(KeyColumnUsage.TABLE_SCHEMA.equal(TableConstraints.TABLE_SCHEMA))
                .and(KeyColumnUsage.TABLE_NAME.equal(TableConstraints.TABLE_NAME))
                .and(KeyColumnUsage.CONSTRAINT_NAME.equal(TableConstraints.CONSTRAINT_NAME))
                .where(TableConstraints.CONSTRAINT_TYPE.equal(constraintType))
                .and(KeyColumnUsage.TABLE_SCHEMA.equal(getSchemaName()))
                .orderBy(KeyColumnUsage.TABLE_NAME.ascending(), KeyColumnUsage.ORDINAL_POSITION.ascending())
                .fetch();
    }
}
```

The library is quite powerful and can do a lot of things. But the code turns out to be not quite readable and poorly debugged.

3. **QueryDSL** is a framework that enables the construction of type-safe SQL-like queries for multiple backends including JPA, MongoDB, and SQL in Java.

Very powerful library. Can do a lot of useful things. The code that can be written looks like this:

```
public class QueryDslExample {

    public static void main(String[] args) {
        List persons = queryFactory.selectFrom(person)
                .where(
                        person.firstName.eq("John"),
                        person.lastName.eq("Doe"))
                .fetch();
    }
}
```

The library is too difficult to learn. The documentation is quite confusing.

4. **Sql2o** is a small java library, with the purpose of making database interaction easy. When fetching data from the database, the ResultSet will automatically be filled into your POJO objects. Kind of like an ORM, but without the SQL generation capabilities.

A fairly good library that allows you to do things quickly and easily. You can write code like this:

```
public class SqlTwoExample {

    public class Task {
        private int id;
        private String category;
        private Date dueDate;

// getters and setters

}

    Sql2o sql2o = new Sql2o(DB_URL, USER, PASS);

    String sql =
            "SELECT id, category, duedate " +
                    "FROM tasks " +
                    "WHERE category = :category";

    Connection con = sql2o.open();

    {
        List tasks = con.createQuery(sql)
                .addParameter("category", "foo")
                .executeAndFetch(Task.class);
    }
}
```

Everything is quite simple and convenient, with good documentation with clear and understandable examples.

5. **Groovy's SQL** features are implemented in the elegant GroovySql API. Using closures and iterators, Groovy Sql neatly transfers JDBC resource management from you, the developer, to the Groovy environment. This way, JDBC programming is less cumbersome, and you can focus on queries and their results.

```
public class GroovySqlExample {
    public static void main(String[] args) {
        sql = Sql.newInstance("jdbc:mysql://localhost:3306/words", "words",
                "words", "org.gjt.mm.mysql.Driver")
        sql.eachRow("select * from word") {
            row |
                    println row.word_id + " " + row.spelling + " " + row.part_of_speech
}
    }
}
```

Using just a few lines, I created the function code without performing Connection closures, ResultSet closures, or any of the usual heavy-handed JDBC programming routines.