**Back End Testing:**

Software has a lot of other elements too that aren’t directly visible or available to the user for direct interaction. It does not make these elements any less important and they must too undergo thorough testing.

The combination of all these well-functioning elements makes a fully formed software application. We can combine everything we do not directly see as ‘Back-end’.

Most [**system testing efforts**](https://www.softwaretestinghelp.com/system-testing/) go through GUI.

**System testing** means testing the system as a whole. All the **modules/components** are **integrated** in order to verify if the system works as expected or not.

**System testing is done after integration testing**. This plays an important role in delivering a high-quality product.

**System testing** can be considered as a **black-box test technique**.

While performing **System testing, functional** & **non-functional**, **Security**, **Performance** and **many other testing types are covered**, and they are tested using black box technique wherein the input is provided to the system and the output is verified. System internal knowledge is not required.

**Elements of Back end Testing**:

* Database
* APIs
* Servers

**1. Database Testing:**

The database is an important element of any application. When the GUI and the DB interact with one another seamlessly your application works well.

There are several database tools available in the market e.g. MS-Access, MS SQL Server, SQL Server, Oracle, Oracle Financial, MySQL,

**Databases are usually validated for**:

* ACID properties
* CRUD operations
* Schema
* Migration
* Business rule conformance
* Security
* Performance

**Advanced ETL** (**Extract Transform and Load**) and data warehouse maintaining systems will need tests run against them too.

**Few categories of Backend Database testing tools**:

1) **Interfaces** that let you **connect** and **run** your **queries** against the **databases**. Some of them have a GUI and some don’t.

* **TOAD**: I am sure everyone has heard of this. It supports many DBs and platforms. It comes both as free and commercial versions. More information, resources, and the free version is found at [toadworld](http://www.toadworld.com/).
* **pHpMyAdmin**: This is an excellent open source tool that lets you run queries and interact with your DB via a User interface. I have personally used this and my team loves how intuitive the tool is. We needed zero training to get comfortable. I highly recommend this tool if you are looking for a connection medium to your MySQL and MariaDB databases [phpmyadmin](https://www.phpmyadmin.net/" \t "_blank)
* **HeidiSQL**: Very similar to pHpMyAdmin. It connects to MySQL , Microsoft SQL databases, and PostgreSQL. Open sourced. Find more information at [hheidisql](http://www.heidisql.com/" \t "_blank)

The list of tools is endless, but the above are some of the most popular choices.

2) **DB load and performance benchmarking tools**:

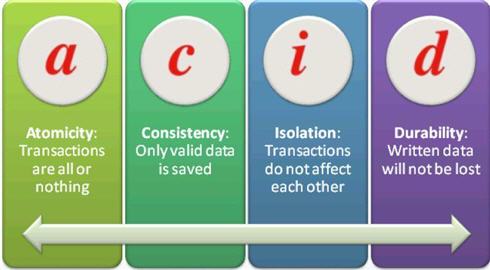
* **HammerDB**: It is an open source tool that many DB experts vouch for. I have personally not used this but it supports many databases. From the screenshots and the looks of it, it looks like a tool worth checking out. More details at [hammerdb](http://www.hammerdb.com/" \t "_blank)
* **SLOB**: The Silly Little Oracle Benchmark tool helps you time and assess I/O style of DB transactions. It can help you understand CPU, Memory and processing times for bulk transactions on your system. More details at [kevinclosson](https://kevinclosson.net/slob/" \t "_blank)
* **Swingbench**: This is a very similar tool to HammerDB. This works on Oracle DBs and is very effective. To understand the tool and its features try this guide: [dominicgiles](http://www.dominicgiles.com/Swingbench.pdf" \t "_blank)

**1) Data Mapping**:

* In software systems, data often travels back and forth from the UI (user interface) to the backend DB and vice versa. So these are some aspects to watch for:
* Check whether the fields in the UI/frontend forms are mapped consistently with the corresponding fields in the DB table.  Typically this mapping information is defined in the requirements documents.
* Whenever a certain action is performed in the front end of an application, a corresponding CRUD (Create, Retrieve, Update and Delete) action gets invoked at the back end. A tester will have to check if the right action is invoked and whether the invoked action in itself is successful or not.

**2) ACID properties validation**:

Atomicity, Consistency, Isolation, and Durability. Every transaction a DB performs has to adhere to these four properties.0

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2013/08/DB-Testing.jpg)

* Atomicity: means that a transaction either fails or passes. This means that even if a single part of the transaction fails- it means that the entire transaction has failed. Usually, this is called the “all-or-nothing” rule.
* Consistency: A transaction will always result in a valid state of the DB
* Isolation: If there are multiple transactions and they are executed all at once, the result/state of the DB should be the same as if they were executed one after the other.
* Durability: Once a transaction is done and committed, no external factors like power loss or crash should be able to change it

**3) Data integrity**:

For any of the [CRUD operations](https://www.softwaretestinghelp.com/crud-testing/), the updated and most recent values/status of shared data should appear on all the forms and screens. The value should not be updated on one screen and display an older value on another one.

When the application is under execution, the end user mainly utilizes the ‘CRUD' operations facilitated by the DB Tool.

* C: Create – When user ‘Save' any new transaction, ‘Create' operation is performed.
* R: Retrieve – When user ‘Search’ or ‘View’ any saved transaction, ‘Retrieve' operation is performed.
* U: Update – When user ‘Edit’ or ‘Modify’ an existing record, the ‘Update' operation of DB is performed.
* D: Delete – When a user ‘Remove’ any record from the system, ‘Delete’ operation of DB is performed.

Any database operation performed by the end user is always one of the above four.

So, devise your DB test cases in a way to include checking the data in all the places it appears to see if it is consistently the same.

**4) Business rule conformity**:

More complexity in databases means more complicated components like relational constraints, triggers, stored procedures, etc. So testers will have to come up with appropriate SQL queries in order to validate these complex objects.

**What to Test (Database Testing Checklist)**

**1) Transactions**

When testing transactions, it is important to make sure that they satisfy the ACID properties.

These are the statements commonly used:

* BEGIN TRANSACTION TRANSACTION#
* END TRANSACTION TRANSACTION#

The Rollback statement ensures that the database remains in a consistent state.

* ROLLBACK TRANSACTION#

After these statements are executed, use a Select to make sure the changes have been reflected.

* SELECT \* FROM TABLENAME <tables which involve the transactions>

**2) Database Schemas**

A database schema is nothing more than a formal definition of how the data is going to be organized inside a DB. To test it:

* Identify the requirements based on which the database operates. Sample requirements:
  + Primary keys to be created before any other fields are created.
  + Foreign keys should be completely indexed for easy retrieval and search.
  + Field names starting or ending with certain characters.
  + Fields with a constraint that certain values can or cannot be inserted.
* Use one of the following methods according to the relevance:
  + SQL Query DESC<table name> to validate the schema.
  + Regular expressions for validating the names of the individual fields and their values
  + Tools like SchemaCrawler

**3) Triggers:**

When a certain event takes places on a certain table, a piece of code (a trigger) can be auto-instructed to be executed.

For example, a new student joined a school. The student is taking 2 classes: math and science. The student is added to the “student table”.  A trigger could add the student to the corresponding subject tables once he is added to the student table.

The common method to test is to execute the SQL query embedded in the trigger independently first and record the result. Follow this up with executing the trigger as a whole. Compare the results.

These are tested in both the black-box and white-box testing phases.

* **White box testing**:  Stubs and drivers are used to insert or update or delete data that would result in the trigger being invoked. The basic idea is to just test the DB alone even before the integration with the front end (UI) is made.
* **Black box testing**:

a) Since the UI and DB, integration is now available; we can insert/delete/update data from the front end in a way that the trigger gets invoked. Following that, Select statements can be used to retrieve the DB data to see if the trigger was successful in performing the intended operation.

b) The second way to test this is to directly load the data that would invoke the trigger and see if it works as intended.

**4) Stored Procedures**:

Stored procedures are more or less similar to user-defined functions. These can be invoked by Call Procedure/Execute Procedure statements and the output is usually in the form of result sets.

These are stored in the RDBMS and are available for applications.

These are also tested during:

* White box testing: Stubs are used to invoke the stored procedures and then the results are validated against the expected values.
* Black box testing: Perform an operation from the front end (UI) of the application and check for the execution of the stored procedure and its results.

**5) Field constraints:**

The default value, unique value, and foreign key:

* Perform a front-end operation which exercises the database object condition
* Validate the results with a SQL Query.

Checking the default value for a certain field is quite simple. It is part of business rule validation. You can do it manually or you can use tools like QTP. Manually, you can perform an action that will add value other than the default value of the field from the front end and see if it results in an error.

Checking the unique value can be done exactly the way we did for the default values. Try entering values from the UI that will violate this rule and see if an error is displayed.

**API testing:**

API is strictly speaking not the back-end but since we are loosely grouping everything that is not visible to the end-user as the back-end, let’s talk about this briefly too.

API stands for Application Program Interface and this is basically where all the programming logic resides. It does not have a UI which is one of the biggest challenges when it comes to testing it. On the other hand, since APIs are generally created before the application’s UI comes into existence, testing the API usually means early testing.

Messaging and send/receive calls are used instead of direct send and receiving of input and output data.

The most popular tool used for API testing is SOAPUI.

**Servers:**

All databases and Applications themselves are installed on servers that keep these systems up and running.

There are a few tests that are run here:

**1) Installation**: Once the installation is complete, you can go to the respective folders and make sure that the files/elements have made it to their target folders in the way they were supposed to. Now, if you are wondering ‘how will I know where everything needs to go?’ ask your development or deployment teams and they can confirm this for you.

This step might not be mandatory, but some companies use manual deployments. In that case, it could become an important [smoke/sanity test](https://www.softwaretestinghelp.com/smoke-testing-and-sanity-testing-difference/) step.

**2) Logs**: There are logs maintained for every transaction’s status in the servers. This will give us insights into whether the end-to-end process has been a success.

Sometimes the front end is sending valid data and the database might get updated right. What if this operation is throwing an exception, causing a memory leak, or causing some kind of a malfunction?  It is the server side logs that will reveal this information to you.

It is not a rule, but generally, most servers are UNIX based systems. So to be able to work through them easily, you are going to need a way to connect to your server.

PuTTy, hands down is the most popular choice to connect to your servers. [Putty](http://www.putty.org/) is an open source product and needs no installation. All you have to do is download and use it.

UNIX systems don’t have a graphic user interface and that is what makes them perfect to be App and DB servers. They are secure, abstract, faster and cheaper. There are many flavors of UNIX and due to the absence of GUI, we will have to use commands to communicate with the server. We all have our go-to resources for [UNIX commands](https://www.softwaretestinghelp.com/unix-tutorials/) and this one is mine: [freeengineer](http://freeengineer.org/learnUNIXin10minutes.html" \t "_blank)

**3) Server’s performance and security**: Just like any other part of the software, the server has to be secure and responsive.