Advantages of Build Automata:

- *A necessary precondition for continuous integration and continuous testing*
- *Improve product quality*
- *Accelerate the compile and link processing*
- *Eliminate redundant tasks*
- *Minimize "bad builds"*
- *Eliminate dependencies on key personnel*
- *Have history of builds and releases in order to investigate issues*

1.Compile your application's Java code and make sure that all class files and resources such as images are in a separate directory.
In the Dynamic Tree Demo application, the compiled classes are placed in the `build/classes/webstartComponentArch` *directory.*

Create a text file that contains any JAR file manifest attributes that your applet needs.
For the DynamicTree Demo applet, create a file named `mymanifest.txt` *in the* `build/classes` *directory, and add the* `Permissions`*,* `Codebase`*, and* `Application-Name` *attributes. The applet does not require access to the user's system resources, so use* `sandbox` *for the permissions. Use the domain from which you will load the sample for the code base, for example,* `myserver.com`*. Add the following attributes to the* `mymanifest.txt` *file.*

```
Permissions: sandbox

Codebase: myserver.com

Application-Name: Dynamic Tree Demo
```

Other manifest attributes are available to restrict an applet to using only trusted code, and to provide security for applets that need to make calls between privileged Java code and sandbox Java code, or have JavaScript code that calls the applet. See the Enhancing Security with Manifest Attributes lesson to learn more about the manifest attributes that are available.

Create a JAR file containing your application's class files and resources. Include the manifest attributes in the `mymanifest.txt` *file that you created in the previous step.*
For example, the following command creates a JAR file with the class files in the `build/classes/webstartComponentArch` *directory and the manifest file in the* `build/classes` *directory.*

```
% cd build/classes

% jar cvfm  DynamicTreeDemo.jar  mymanifest.txt webstartComponentArch
```

See the Packaging Programs in JAR Files lesson to learn more about creating and using JAR files.

1. *Sign the JAR file for your applet and time stamp the signature. Use a valid, current code signing certificate issued by a trusted certificate authority to provide your users with assurance that it is safe to run the applet.*
   See the Signing JAR Files lesson for more information.
   If you want to use a signed JNLP file for security, create the JNLP file as described in the next step

and include it in the JAR file before the JAR file is signed. See Signed JNLP Files in the Java Platform, Standard Edition Deployment Guide for information.

Create a JNLP file that describes how your application should be launched.
Here is the JNLP file that is used to launch the Dynamic Tree Demo application. Permissions are not requested for this application so it runs in the security sandbox. The source for dynamictree_webstart.jnlp follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<jnlp spec="1.0+" codebase=

"https://docs.oracle.com/javase/tutorialJWS/samples/deployment/webstart_Compone
ntArch_DynamicTreeDemo"

    href="dynamictree_webstart.jnlp">

    <information>

        <title>Dynamic Tree Demo</title>

        <vendor>Dynamic Team</vendor>

    </information>

    <resources>

        <!-- Application Resources -->

        <j2se version="1.7+"

              href="http://java.sun.com/products/autodl/j2se"/>

  <jar href="DynamicTreeDemo.jar"

            main="true" />

</resources>

    <application-desc

         name="Dynamic Tree Demo Application"

         main-class=

           "webstartComponentArch.DynamicTreeApplication"

         width="300"

         height="300">

     </application-desc>

     <update check="background"/>
```

```
</jnlp>
```

Create the HTML page from which your application will be launched. Invoke Deployment Toolkit functions to deploy the Java Web Start application.

In the example, the Dynamic Tree Demo application is deployed in `JavaWebStartAppPage.html`.

```
<body>
    <!-- ... -->
    <script src=
      "https://www.java.com/js/deployJava.js"></script>
    <script>
        // using JavaScript to get location of JNLP
        // file relative to HTML page
        var dir = location.href.substring(0,
            location.href.lastIndexOf('/')+1);
        var url = dir + "dynamictree_webstart.jnlp";
        deployJava.createWebStartLaunchButton(url, '1.7.0');
    </script>
    <!-- ... -->
</body>
```

Place the application's JAR file, JNLP file, and HTML page in the appropriate folders.
For this example, place `DynamicTreeDemo.jar`, `dynamictree_webstart.jnlp`, and `JavaWebStartAppPage.html` in the same directory on the local machine or a web server. A web server is preferred. To run from the local machine, you must add your application to the exception site list, which is managed from the Security tab of the Java Control Panel.

Open the application's HTML page in a browser to view the application. Agree to run the application when prompted. Check the Java Console log for error and debugging mess

Automation Based on the Type of Testing

Automation of Functional Tests:

Functional tests are written to test the business logic behind an application. Automating these mean writing scripts to validate the business logic and the functionality expected from the application.

Automation of Non-Functional Tests:

Non-functional tests define the non-business requirements of the application. These are the requirements related to performance, security, databases, etc. These requirements can remain constant or can be scaled as per the size of the software.

Automation Based on the Phase of Testing

Automation of Unit Tests:

These tests are run during the development phase itself, ideally by the dev after the completion of development and before handing over the system to the testers for testing.

Automation of API Tests:

API tests are run during the integration phase. These may be run by the development or testing team and can be run before or after the UI layer is built for the application. These tests target the testing based on the request and response on which the application is built.

Automation of UI based tests:

UI Based tests are run during the test execution phase. These are specifically run by the testers and are run only once before the UI of the application is handed over to them. These test the functionality and business logic of the application from the front end of the application.

Automation Based on the Type of Tests

Unit Tests:

Unit Tests are the tests that are built to test the code of an application and are usually built into the code itself. They target the coding standards like how the methods and functions are written.

These tests are more often written by the developers themselves, however, in today's world, automation testers may also be asked to write them.

Executing these tests and getting no bugs from them will mean that your code will compile and run without any code issues. These tests usually do not target the functional aspects of the application and as they target code, it is more appropriate to automate them so that they can be run as and when required by the developer.

Smoke Tests:

The smoke test is a famous test performed in the test life cycle. These are post-build tests, they are executed immediately after any build is given out of the application to ensure that the application is still functioning after the build is done.

This is a small test suite and is something that will be executed multiple times and thereby it makes sense to automate it. These tests will usually be of a functional nature and depending on the type of application a tool can be picked for them.

API tests:

API testing has become very famous in the past few years. Applications built on the API architecture can perform this testing.

In API testing, the testers validate the business layer of the application by checking the request-response combinations for the various API's on which the application is built. API Tests can also be done as a part of the integration tests below.

Integration Tests:

Integration test as the name itself suggests means testing the application by integrating all the modules and checking the functionality of the application.

Integration testing can be done through API testing or can be done through the UI layer of the application.

UI tests:

UI tests are done from the UI layer or the frontend of the application. These may target testing the functionality or simply test the UI elements of an application.

Automating the UI to test the functionality is a common practice. However, automating the GUI features is one of the more complicated automation.

Regression tests:

One of the most commonly automated test suites is the regression test suite. Regression, as you may already know, is the test that is done at the end of testing a new module to ensure that none of the existing modules have been affected by it.

It is repeated after each new iteration of testing and the main test cases stay fixed with usually a few new additions after a new iteration. As it is frequently run almost all the test teams try to automate this pack.

Automation as Continuous integration:

Continuous Integration may again be running on the automated regression tests itself, however, in achieving CI, we enable the regression or identified test suite to be run every time when a new deployment is done.

Security Tests:

Security testing can be both functional as well as a non-functional type of testing which involves testing the application for vulnerabilities. Functional tests will compose of tests related to authorization etc., whereas non-functional requirements maybe test for SQL injection, cross-site scripting, etc.

Performance Tests and Quality control:

Performance tests are non-functional tests which target the requirements like testing of load, stress, scalability of the application.

Acceptance tests:

Acceptance tests again fall under functional tests which are usually done to ensure if the acceptance criteria given by the client has been fulfilled.

So far, we have described the type of tests that can be automated and various classifications of the same, all classifications eventually will lead to the same end results of a test suite being automated. As we said earlier a little understanding is required on how these are different from frameworks.

Once you have identified the tests that you want to automate from the above classification, then you will need to design your logic in a manner to execute these tests smoothly, without much manual intervention. This design of a manual test suite into an automated test suite is where frameworks come in.

How To Create A Frontend Website Testing Plan?

Creating Frontend testing plan is a simple 4 step process.

Step 1) Find out tools for Managing Your Test Plan

Step 2) Decide the budget for Front End Testing

Step 3) Set the timeline for the entire process

Step 4) Decide the entire scope of the project. The scope includes the following items

- *OS and browsers used by users ISP plans of your audience*
- *Popular devices used by audience*
- *Proficiency of your audience*
- *Internet correction speed of the audience*