The Benefits of Automation Testing & Tips for Getting Started

SMARTBEAR

Contents

Introduction

Automation testing makes software testing easier, faster and more reliable, and is essential in **today's fast-moving software delivery environment**. Usually seen an alternative to time-consuming and labor-intensive manual testing, automation testing uses software tools to run a large number of tests repeatedly to make sure an application doesn't break whenever new changes are introduced. Implementing a **test automation strategy** within your organization is not for the faint of heart and you need to rely on well-chosen metrics that measure the performance (past, present, and future) of your automated testing process to determine if your company is getting an acceptable return on its test automation investment.

Setting up automation testing is a process, and **any metrics chosen to measure improvement** (e.g. the number of manual versus automated tests) need to take into account unique aspects of your organization, your market, or environment they are being used in. There is no set of "universal metrics" report that will work in every capacity all of the time.

What is Automation Testing?

Automation testing involves using automation tools to run test suites, which are predefined actions, against a software application, creating test reports that compare the actual results to the expected behavior of the application. Running these detailed, repetitive, and data-intensive automatic tests helps teams improve software quality and make better use of development and testing resources.

Automation testing will make your organization's entire software testing process more efficient. The main difference between manual testing and automated testing is that the former requires a person, whereas the latter runs a program. Many companies adopt automation testing to deal with the tedious and time-consuming aspects of manual testing. This saves time and effort for developers, which makes it easier for them to explore innovative ways to optimize the look, feel and performance of an application. Manual testing is important on automation projects because it's a quick way to execute any test cases that have not yet been automated. Since there may be no time to build automation for new features introduced in the current build, manual testing is often the best option for test completion. Manual testers also have an important role in exploratory testing, which is free-form testing that finds defects that not have been addressed through your predefined test cases. Similar to how automation testing saves time and effort for developers, automating tests for core functionality in your applications will free up resources that manual testers can devote toward additional exploratory testing.

QA automation engineers typically have a specific set of skills that manual testers do not, which is a working knowledge of computer programming and software development that allows them to write code and build automation scripts. While these skills presently come in handy for setting up, fine-tuning and maintaining automated tests, the role of automation engineers is likely to become less important as more and more test automation tool vendors embed artificial intelligence and machine learning throughout the software delivery pipeline.

Benefits and Risks of Automation Testing

In addition to assuring higher efficiency of your development and testing teams, test automation can increase the depth and scope of tests to help improve software quality, thereby decreasing the need for bug fixes after release and reducing overall project costs. It also makes it easy to expand coverage of your testing to include things like multiple operating environments and hardware configurations. Increased test coverage leads to testing more features, which results in higher quality, more accurate software that can be released earlier, with fewer problems. Automation testing helps to both reduce the time-to-market of an application and increase the return-on-investment (ROI) of that application.

Among the risks involved in automation testing is the need for version control and maintainability of test scripts and test results. This especially can be a problem when you attempt to automate testing work too early in the development cycle when the test process to be automated isn't fully understood. Relying on poorly designed or maintained test scripts can lead to your entire automation testing process being rejected by developers and testers as being more trouble than it's worth. This risk doesn't outweigh the benefits of test automation, especially if your teams are using the appropriate automated testing tools and/or frameworks.

Test Automation Frameworks

A test automation framework is made of components like function libraries, test data sources and reusable code modules that can be assembled like small building blocks to model different business processes for testing purposes. By enforcing a set of rules or guidelines for creating and designing test cases for each product tested, frameworks greatly simplify the automation effort.

Let's briefly look at a list of the 6 main types of test automation frameworks and the pros and cons of each.

Linear Automation Framework

Also known as the 'Record and Playback' framework, this is the simplest type of framework. Using this framework, testers generate test scripts by recording each and every step--such as navigation or user input--and then playing the script back automatically to conduct the test. Because the framework uses record and playback to create linear scripts, testers don't need to write custom code. This makes it easy and straightforward to set up. The use of hard coded data means the framework can't be run with multiple data sets, which limits its reusability. This type of framework should only be used to test small sized applications.

Module-based Testing Framework

This framework involves testers dividing an application into multiple modules and creating individual test scripts for each of the modules. After breaking down the modules, testers can also combine the individual test scripts into a single, master test script that can be changed or modified for particular test scenarios. Because test scripts for various modules can be reused, creating test cases for an application often takes less effort and time. Modular frameworks are easier to scale and maintain, but data still remains hard-coded into the different test scripts, which means tests still cannot use multiple data sets within a modular framework.

Library Architecture Testing Framework

Built on the module-based testing framework, the Library Architecture Testing Framework provides many of the same benefits, such as easier test maintenance and scalability, which will reduce the time and money spent on testing.This framework creates a common library constituting of common functions that test scripts can call for certain functionality as needed, such as the login steps

to access an application or the steps to validate a credit card.

Data is still hard-coded into the script with this framework, however, so any changes to data mean you must change the scripts. Another drawback is that writing and analysis of common functions within scripts requires considerable technical expertise.

Data-Driven Framework

Unlike Linear or Modular-based testing frameworks, a data-driven framework separates the test script logic away from the test data. The test data set is kept in external files or resources such as spreadsheets, databases, XML files, etc., which testers access programmatically. A big advantage of this framework is that it's an easy way to expand test coverage since a wide variety of test scenarios can be executed by just varying the test data in the external data file. If the external source also contains validation values, then the data-driven test can compare the results of the test to the validation value to determine whether the test case passes *(fig 1)*. The downside of the data-driven framework is that it requires testers to have both business knowledge and the ability to write code that can connect the tests to the external data sources.

Keyword-Driven Framework

Keyword-driven testing is similar to data-driven testing in that it separates the test data and script logic but takes this concept even further. In addition to externally stored data, keywords are stored in a separate file along with blocks of code or scripts. This framework promotes re-usability since one keyword can be used across many test scripts and test scripts can be independent of the current application.

The table shows an example of keyword- driven test data containing a simple test case for testing a login web application *(fig 2)*. The test cases consist of keywords Runapp, Username, Password and OK, and the arguments that are inputs and expected outputs for the test cases. As you can see, keywords can contain multiple parameters which makes it easy to add different test cases without implementing new keywords.

A disadvantage of the Keyword-Driven Framework is that the tester needs to have considerable exper-

Data-Driven Framework

Firts Name	Last Name	Desired User Name	Desired Password	Expected Result
Jane	Doe	jdoe	Doepass123	Pass
John	Doe	jdoe_sr	Doepass456	Pass
John	Doe, Jr.	jdoe_jr	Doepass789	Fail

(fig 1)

Keyword-Driven Framework

Step #	Type	Keyword	Operation	Expected Result
1	Function	Runapp		https://mail.google.com/mail/u/0/
2	Item	Username	SetValue	Username=jdoe
3	Item	Password	SetValue	Password=Doepass123
4	Item	OK	Click	

(fig 2)

Hybrid Testing Framework

Step #	Description	Keyword	Locator	Data
1	Navigate to login page	navigate		
2	Enter User Name	input	//*[@id='username']	userA
3	Enter Password	input	//*[@id='password']	password

(fig 3)

tise in the testing tool's scripting language in order to successfully create customized functions and utilities required for a particular application.

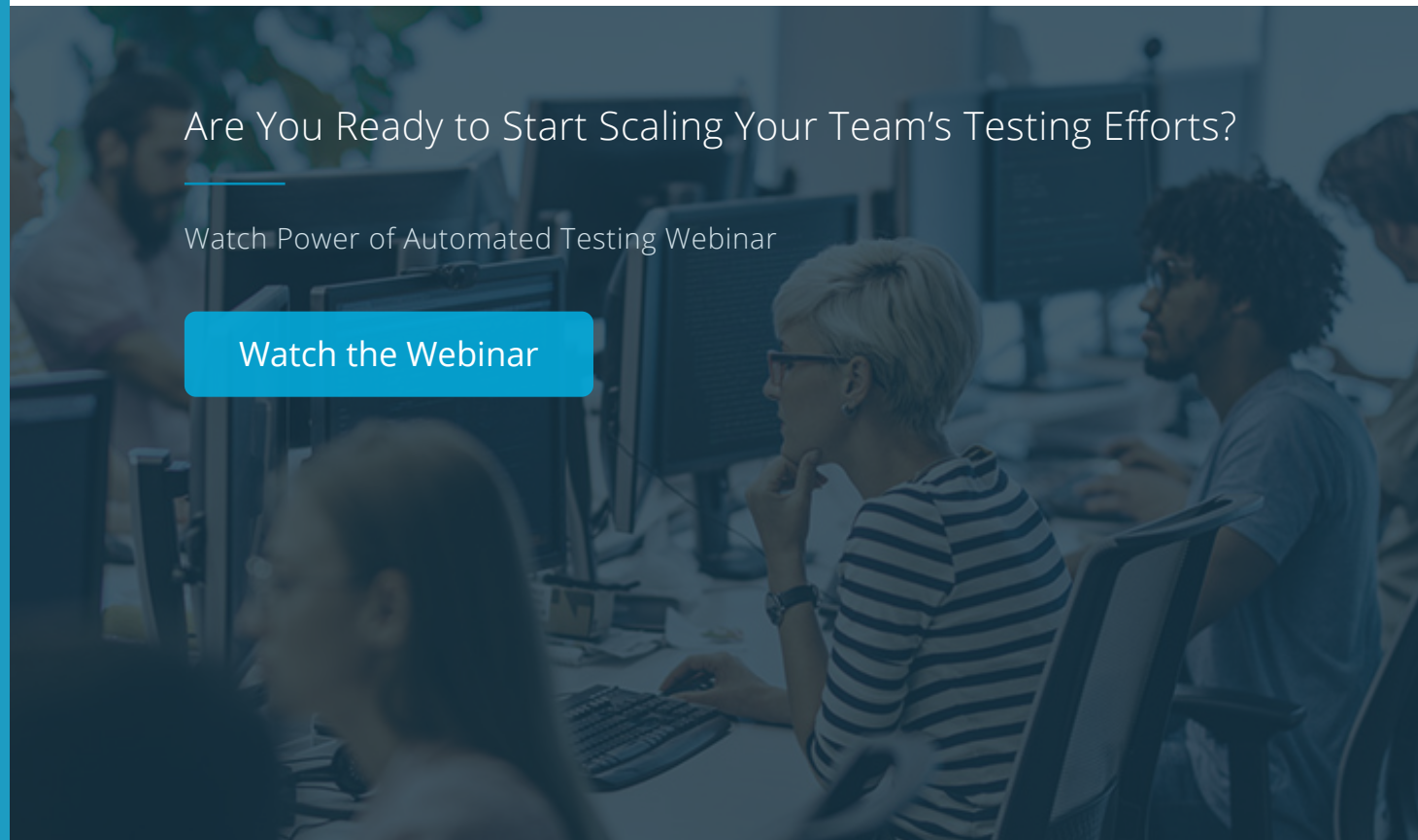
Hybrid Testing Framework

As the name implies, hybrid testing frameworks are combinations of at least two different testing frameworks detailed above. The table below shows an example of this, which is a combination of a Keyword-Driven Framework and a Data-Driven Framework, with a keyword column that contains the necessary keywords used in a particular test case and a separate data column that contains the data required in a test scenario *(fig 3)*.

The advantage of a Hybrid Testing Framework is that it allows testers to combine the benefits of different testing frameworks in order to leverage what best in each framework. It has many of the same disadvantages of the Keyword-Driven Framework in that it requires testers to have programming expertise to effectively use this framework.

How Do You Decide Which Framework is Best?

The key to choosing a framework is find one that works well with your existing development and testing processes. That comparison can involve a lot of trial and error if you're new to test automation frameworks. A good way to speed up your learning process is to try to implement a simple Linear Automation Framework first and then move on to more sophisticated frameworks that offer better support for different key factors like reusability, ease of maintenance, etc. After looking at each of the above frameworks, you're likely to find that a hybrid framework will give you the best test results—but you'll only know that after you've explored the other options.

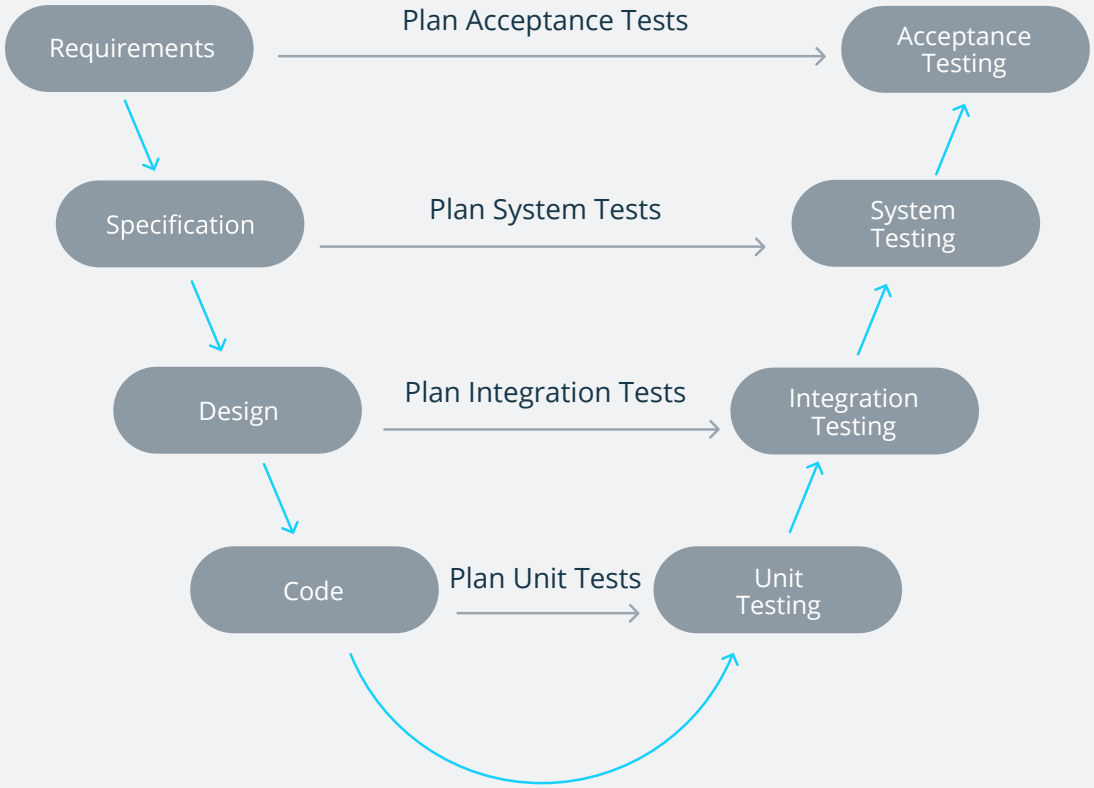


Types of Tests to Automate

The software testing process is generally broken down into four levels in order to test different aspects of a system:

Level	Summary
Unit Testing	This is the level where individual units/components of a software/system are tested, with the goal of validating that each unit of the software performs as designed. Unit testing is typically carried out by the developer.
Integration Testing	At the integration level, individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.
System Testing	At the system level, a complete, integrated system/software is tested. System testing seeks to detect defects both within modules that have passed integration tests and also within the system as a whole.
Acceptance Testing	At the acceptance test level, the system is tested to determine if it meets mutually agreed-upon requirements and is acceptable for delivery.

(fig 4)



The V-model (Verification and Validation model) is useful in showing the relationships between each phase of the software development life cycle and its associated phase of testing *(fig 4)*.

Unit Testing

A definition of unit testing is a software development and testing approach in which the smallest testable parts of an application, called units, are individually and independently tested to see if they are operating properly. Unit testing can be done manually but is usually automated. Unit testing is a part of the test-driven development (TDD) methodology that requires developers to first write failing unit tests. Then they write code in order to change the application until the test passes. Writing the failing test is important because it forces the developer to take into account all possible inputs, errors and outputs.

A payroll algorithm that's responsible for computing weekly wages for hourly workers is one example of where unit tests are useful. The algorithm needs to calculate at a standard rate for pay periods less than 40 hours and at an overtime rate for periods over 40 hours. There also needs to be error handling for wrong inputs like negative hours or negative

wages. Using tools such as NUnit, JUnit, RSpec, etc., developers can quickly write automated unit tests that ensure all paths through the code are properly validated.

Integration Testing

In integration testing, different software modules are combined and tested as a group to make sure that the integrated system is ready for system testing. Integration tests can involve checking behaviors for web services, database calls, or other API interactions. They're usually much slower than unit tests because they need to do things like establish database authentication or deal with network latency. Integration testing checks the data flow from one module to other modules. Continuing with the payroll example from the unit test section above, a good integration test would be: when the payroll service is invoked with valid data, does this give a correct response.

As far as automating integration testing, a best practice among many DevOps teams is to do Shift Left testing, that is, to shift integration testing to the left of its usual position in the delivery pipeline so that it occurs as close as possible to the build process. Since integration testing is where many disruptive, significant defects are often detected, this allows teams to receive feedback on code quality faster, with more accurate results.

System Testing

System Testing includes numerous software testing types that are used to validate software as a whole (software, hardware, and network) against the requirements for which it was built. Different types of tests (Functional Testing, Data Driven Testing, Keyword Testing, Regression Testing, Black Box Testing, Smoke Testing, etc.) are carried out to complete system testing.

Functional Testing

Functional tests verify that each software function satisfies business requirements. A functional test is not concerned with an application's internal details; it tests to see if the application does what it's supposed to do. Does the payroll module work as specified? Does it prevent you from editing an employee's pay data when you're not authorized to do that action?

Functional tests often run via the User Interface and many times are performed manually by testers

who use their knowledge of company processes to see if the application meets business requirements. Functional tests are relatively straightforward to automate using automation tools that have Record and Replay capabilities. These tools use a Linear Automation Framework (see above) to record test steps, which are then executed on the application under test during Replay/Playback.

Data Driven Testing

In **data-driven testing**, test scripts are created using test data and/or output values (i.e. expected results) that are read from external data files instead of using the same hard-coded values each time *(fig 5)*.

See the above section on the Data-Driven Framework for the pros and cons of this approach.

Here are some best practices when initiating a data-driven testing project:

- Start with a standard test case, such as user login test in the table above, since this will give you a good idea of what you are going to test and how to perform the verification. Using a test automation tool with Record and Replay capabilities to record the test inputs will simply this step.
- Identify the data in the test case and the data types needed to store this data (such as a valid user ID and password characters, in the example above).
- Modify the test case to use variables instead of hard data.
- Modify the test case to specify input arguments to be used to pass in the data. Replace the hard-coded data in the test case with variables.
- Call the test case and pass in the data.

One of the many benefits of data-driven testing is that it lets you create automated test projects that can be extended indefinitely by simply adding new lines of text to a text file or a spreadsheet. In addition to helping create a simple data-driven test like the one above, a test automation tool will come in handy in designing much more sophisticated data-driven tests that require navigation through a program, reading data files and logging the status of tests.

(fig 5)

Firts Name	Last Name	Desired User Name	Desired Password	Expected Result
Jane	Doe	jdoe	Doepass123	Pass
John	Doe	jdoe_sr	Doepass456	Pass
John	Doe, Jr.	jdoe_jr	Doepass789	Fail

Keyword Testing

In Keyword Driven Testing, you first identify a set of keywords and then associate an action (or function) related to these keywords. For example, the keywords *click, item, openbrowser* can be used to describe testing actions like mouse click, selection of a menu item, or opening or closing of browser, and so on. Each keyword will be linked with at least one command, test script or function, which implements the actions related to that keyword. testing the application piece-by-piece

Keyword-based testing can be done both manually as well as automatically and is compatible with almost any automation tools available in the market. When it's used in automated testing, it's almost always used with a Keyword-Driven Framework like the one described above. This means that when test cases are executed, keywords are interpreted by a test library, which is called by the Keyword-Driven test automation framework.

The fact that keyword driven testing can be done both manually and automatically allows automation to be started earlier in the software development lifecycle, sometimes even before a stable build is delivered for testing. This is a disadvantage if your team is doing agile development where requirements and code are constantly changing. Since keyboard-driven testing allows you to combine manual steps with automated ones, a best practice of keyboard -driven tests is to continually evaluate your automation to see that it is working as expected.

Regression Testing

Regression Testing is a type of software testing used to confirm that a recent program or code change has not adversely impacted your system's functionality. Regression testing can be performed during any level of testing (Unit, Integration, System, or Acceptance) but it is most relevant during System Testing.

During regression testing, new test cases are not created, but either a full or partial selection of previously created test cases are re-executed. To save time and money, many organizations choose to select part of the test suite to be run instead of re-executing the entire test suite. In this case, you should prioritize the test cases chosen depending on frequency of use or business impact. If you're developing a mobile app, for example, it makes little sense to spend time and resources testing your app on platforms that are obsolete or little used by your customers.

It goes without saying that you need to use automation testing software for full-scale regression tests that re-execute an entire test suite.

Black Box Testing

Black Box Testing focuses on the inputs and outputs of the software system without looking at the internal code structure, software paths or implementation details. An example of black box testing is a QA tester (not a programmer) who navigates through an e-commerce portal, performing actions like clicking links or making mock purchases, and checking the results with the expected behavior. Black box testing has the advantage of being unbiased and performed from a user's point-of-view and not the developer's. A big drawback of black box testing is that can be redundant if your developers are using a test-driven

development (TDD) methodology where developers will have already written unit test cases that take into account all possible inputs, errors and outputs.

Automated black box testing can dramatically cut down on testing time since it can do in a few minutes what may take a manual tester several hours. When black box tests are automated, it's possible to run more tests covering many more inputs and conditions using limited testing resources. These kinds of tests can be automated by testing frameworks for graphical user interface testing and/or API driven testing.

Smoke Testing

A smoke test is a quick run through of an application designed to ensure that it can perform basic features. The term smoke testing originates from a similarly basic type of hardware testing in which a device passes the test if it doesn't smoke or catch fire the first time it turns on. Also called as "Build Verification Testing," this type of testing that is carried out by software testers to check if the new build provided by the development team is stable enough to carry out further or detailed testing such as unit and integration tests. An example of smoke testing might be: on a car rental site where a tester ensures the user can sign in, enter a credit card number, look at and reserve a car, and be notified of the car's availability.

While a smoke test has the advantage of helping find software bugs in the early stages of testing, it's easy to miss critical issues that can only be found by more detailed testing.

As with block box testing, automating smoke testing with GUI- or API-driven testing frameworks will make your testing process much more efficient. This is especially important if developers make frequent builds since automated smoke tests will enable QA testers to provide faster feedback on the test builds.

Acceptance Testing

The goal of acceptance testing is to make sure to check if the software confirms to the business requirements provided by the customer. Acceptance testing occurs much earlier and more frequently on agile projects. In agile development, acceptance tests are normally documented at the beginning of a sprint and serve as a means for testers and developers to work towards a common understanding and shared business domain knowledge.

In many ways, acceptance tests are similar to black-box system tests since they focus on inputs and outputs of the system as a whole rather than the internals of the software program. On both traditional waterfall and agile development projects, customers or end-users are responsible for verifying the correctness of the acceptance tests and prioritizing re-work that fails any acceptance tests.

Acceptance testing is difficult to automate since the question of whether not an application meets its acceptance criteria is often subject to legal or contractual review. There are a number of planning and tracking testing tools however, that can be used to speed up and make the review process more efficient, and to ensure customer feedback is prioritized and implemented.

Automation Testing Tools

Choose the Right Testing Solution the First Time

Download Test Management Buyer's Guide

Download the Guide

Choosing the right automation testing tool is vitally important for your organization since the right software testing tool in the right hands can foster team collaboration, drive down costs, shorten release cycles, and provide real-time visibility into the status and quality of your software projects.

Automation testing works by running a large number of tests repeatedly to make sure an application doesn't break whenever new changes are introduced—both at the GUI- and API-level. For most agile development teams, these automated tests are usually executed as part of a **Continuous Integration (CI)** build process, where developers check code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect errors and conflicts as soon as possible. CI tools such as Jenkins, Bamboo, and Selenium are also used to build, test and deploy applications automatically when requirements change in order to speed up the release process.

Jenkins

Jenkins is a **CI/CD server** that runs tests automatically every time a developer pushes new code into the source repository. Because CI detects bugs early on in development, bugs are typically smaller, less complex and easier to resolve. Originally created to be a build automation tool for Java applications, Jenkins has since evolved into a multi-faceted platform with over a thousand plug-ins for other software tools. Because of the rich ecosystem of plug-ins, Jenkins can be used to build, deploy and automate almost any software project--regardless of the computer language, database or version control system used.

Bamboo

Bamboo is a CI/CD server from Atlassian. Like Jenkins and other CI/CD servers, Bamboo allows developers to automatically build, integrate, test and deploy source code. Bamboo is closely connected with other Atlassian tools such as Jira for project management and Hipchat for team communication. Unlike Jenkins, which is a free and open source agile automation tool, Bamboo is commercial software that is integrated (and supported) out of the box with other Atlassian products such as Bitbucket, Jira, and Confluence.

Selenium

Selenium is a suite of different open-source software tools used for automated software testing of web applications across various browsers/ platforms. Most often used to create robust, browser-based regression automation suites and tests, Selenium--like Jenkins--has a rich repository of open source tools that are useful for different kinds of automation problems. With support for programming languages like C#, Java, JavaScript, Python, Ruby, .Net, Perl, PHP, etc., Selenium can be used to write automation scripts that run against most modern web browsers.

TestComplete

TestComplete has a powerful and comprehensive set of features for web, mobile, and desktop application testing. In addition to having an easy-to-use record and playback feature, TestComplete allows testers to use JavaScript, VBScript, Python, or C++Script to write test scripts. The tool also an object recognition engine that can accurately detect dynamic user interface elements, which makes it especially useful in applications that have dynamic and frequently changing user interfaces. Since it's a SmartBear product, TestComplete can be integrated easily with other products offered by SmartBear.

SoapUI

SoapUI is a test automation tool for functional testing, web services testing, security testing and load testing. Specifically designed for API testing, SoapUI supports both REST and SOAP services. SoapUI provides drag and drops options for creating test suites, test steps and test requests to build complex test scenarios without having to write scripts.

API automation testers can use either the open-source or pro version. The pro edition has several advanced features such as support for data-driven testing, native CI/CD integrations, and multi environment switching, which makes it easy to switch between different sets of SOAP and REST services, properties and database connections.

In addition to these CI/CD and test automation tools, most agile teams also rely on test automation frameworks such as the ones described above. As you've seen, these test automation frameworks are made of function libraries, test data sources, and other reusable modules that can be assembled like building blocks so teams can create automation tests specific to different business needs. So, for example, a team might use a specific test automation tool or framework to automate GUI tests if their software end users expect a fast, rich and easy user interface experience. If the team is developing an app for an **Internet of Things (IoT)** device that primarily talks to other IoT devices, they would likely use a different test tool or automation framework.

Bring Test Automation and Test Management into Jira

Synchronize Results from Automated Tests into Zephyr for Jira

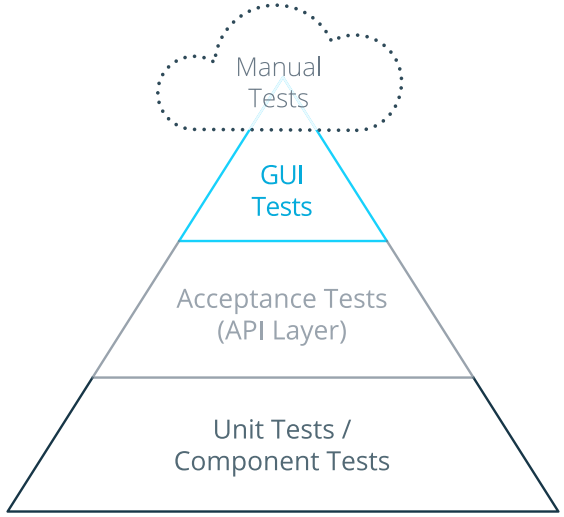
Watch Integrating Open Source Test Automation with Zephyr for Jira

Watch the Webinar

Choosing What Tests to Automate

The testing pyramid is a popular strategy guide that agile teams often use when in planning their **test automation strategy**. As shown in the illustration above, the base or largest section of the pyramid is made up of Unit Tests—which will be the case if developers in your organization are integrating code into a shared repository several times a day. This involves running unit tests, component tests (unit tests that touch the filesystem or database), and a variety of acceptance and integration tests on every check-in.

The best way to start this process is through adopting a combination of **Test-Driven Development (TDD)** and **Behavior Driven Development (BDD)**. TDD defines the principal of writing unit tests to verify each piece of code prior to development. BDD derives from TDD to provide further focus on writing requirements for your development based on user behavior. BDD also provides a common business language so that teams of all technical levels can have a clear shared understanding. As a result, agile teams will have the ability to develop automated tests and change their codebase quickly based on feedback from users.



The Test Automation Pyramid

Unit and component tests are the least expensive to write and maintain, and arguably provide the most value because they allow agile teams to detect errors and conflicts as soon as possible.

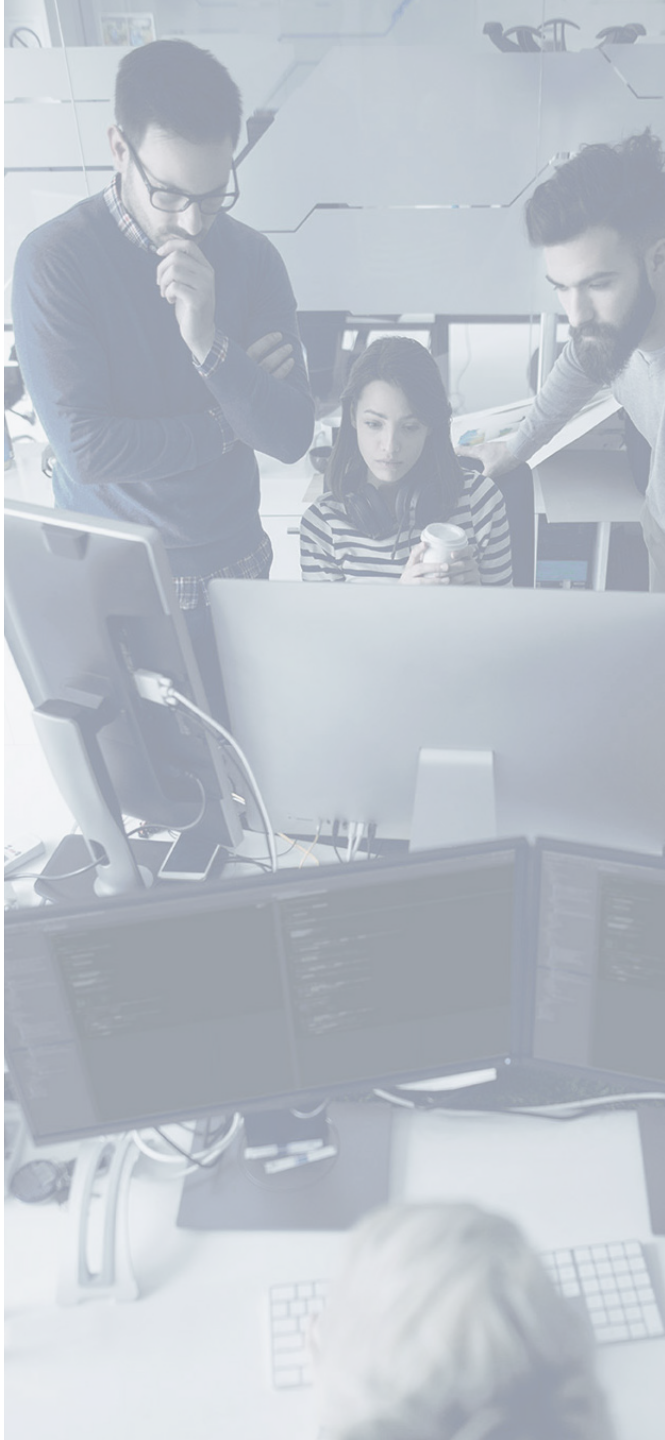
The middle of the pyramid consists of Acceptance and GUI integration tests, which represent smaller pieces of the total number of automation test types that should be created.

At the top or eye of the pyramid are the last tests that should be considered for automation, the manual exploratory tests, which are tests where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests. Exploratory testing is done in a more freestyle fashion than scripted automated testing, where test cases are designed in advance. With **modern test management software**, however, it's possible to semi-automate these kinds of tests, which entails recording and playing back the test path taken by an exploratory tester during a testing session. This helps other agile team members recreate the defect and fix the bug.

Agile projects still need manual testers to engage in exploratory test sessions while the automation test suite runs. In addition to revising and fine-tuning the automated tests, exploratory testers are important on agile projects since developers and other team members often get used to following a defined process and may stop thinking outside the box. Because of the desire for fast consensus among self-organizing agile teams (including globally distributed ones), collaboration can devolve into groupthink. Exploratory testing combats this tendency by allowing a team member to play the devil's advocate role and ask tough, "what if"-type testing questions. Because of the adaptable nature of exploratory testing, it can also be run in parallel with automated testing and doesn't have to slow deployment down on agile projects.

Successfully Transition from Manual to Automated Testing with Zephyr's Vortex and Capture for Jira

Agile teams can execute one-touch control of test automation from within the Zephyr Platform with **Vortex**, Zephyr's advanced add-on, which allows you to integrate with a growing suite of automated



testing frameworks (including EggPlant, Cucumber, Selenium, UFT, Tricentis, and more) with minimal configuration. In addition to being able to create and reuse manual tests on agile projects, Vortex makes it easy to bring in and work with automation information from across your development stack, including from systems external to your organization. Vortex allows users--wherever they are in your organization—to integrate, execute, and report on test automation activities. By providing an intuitive screen that lets users access both manual and automated test cases at the same time, Vortex helps agile teams better monitor their overall automation effort (that is, the number of manual versus automated tests) from one release to another.

Capture for Jira helps software testers on agile projects create and record **exploratory and collaborative testing sessions**, which are useful for planning, executing and tracking manual or exploratory testing. Session-based test management is a type of structured exploratory testing that requires testers to identify test objectives and focus their testing efforts on fulfilling them. This type of **exploratory testing** is an extremely powerful way of optimizing test coverage without incurring the costs associated with writing and maintaining test cases. Like **Zephyr for Jira**, Capture for Jira has a deep integration with the Jira platform, allowing users to capture screenshots within browsers, create annotations, and validate application functionality within Jira.

Both of these tools from Zephyr will help you measure the performance (past, present and future) of your automation testing process to make sure your company is getting an acceptable return on its test automation investment.

Integrate Behavior Driven Development with Jira

Expand your Behavior Driven Development in Jira with Zephyr and Cucumber

Learn how bringing BDD into Jira can improve your testing

Watch the Demo

Agile teams can execute one-touch control of test automation from within the **Zephyr Standalone** solution with **Vortex**, which allows you to integrate with a growing suite of automated testing frameworks (including EggPlant, Cucumber, Selenium, UFT, Tricentis, and more) with minimal configuration. In addition to being able to create and reuse manual tests on agile projects, Vortex makes it easy to bring in and work with automation information from across your development stack, including from systems external to your organization. Vortex allows users--wherever they are in your organization--to integrate, execute, and report on test automation activities. By providing an intuitive screen that lets users access both manual and automated test cases at the same time, Vortex helps agile teams better monitor their overall automation effort (that is, the number of manual versus automated tests) from one release to another.

Cucumber for Jira brings the power of Behavior Driven Development into Jira by synchronizing a developer's feature files from GitHub, BitBucket, or GitLab into our living documentation. Teams accustomed to managing their project/product in Jira can now understand the work being done by developers without having to leave Jira. This work can be linked to existing or new Jira issues and provides a Gherkin syntax editor in any Jira Issue view. Cucumber for Jira integrates with Zephyr for Jira so that testers can also leverage BDD during their test creation and execution.

Zephyr for Jira enables development and testing teams of all sizes to utilize full-featured test management capabilities by integrating testing into the product life cycle within Jira. As automation plays a critical role in achieving continuous testing on the path towards Agile and DevOps adoption, Zephyr for Jira enables agile teams to unify test management and test automation with a native Jira experience. This starts with seamlessly integrating with the leading test automation tools from SmartBear: **SoapUI Pro**, **TestComplete**, **CrossBrowserTesting**, and **LoadNinja**. These native integrations complement the user's already existing workflows in Jira by

enabling integration of functional and performance test automation efforts across UI, API, and data layers, as well as covers a range of mobile devices and browsers. To incorporate more Open Source frameworks, **A.T.O.M (Automated Test results Organizer and Manager)** integrates test results from Selenium, Cucumber, and frameworks using Junit or TestNG reports to automatically update test status in Zephyr for Jira. A.T.O.M supports either the single tester manually uploading results files or testing teams using our ZBot remote agents to sync results from multiple machines simultaneously.

These solutions from Zephyr will help you measure the performance (past, present and future) of your automation testing process to make sure your company is getting an acceptable return on its test automation investment.

Software Testing **As Agile As *You* Need It to Be**

Spend less time testing and more time building. From agile development to predictive analytics, you'll achieve full Continuous Testing Agility.

Discover Zephyr


SMARTBEAR