ibm.com/redbooks

Integration Testing for Hybrid Cloud Applications using Galasa

Michael Baylis

Russell Bonner

James Davies

Clive Harris

Caroline McNamara

Catherine Moxey

Richard Somers

Will Yates

Redpaper

IBM Redbooks

Integration Testing for Hybrid Cloud Applications using Galasa

March 2021

REDP-5614-00

Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

First Edition (March 2021)

Contents

Notices

Trademarks

AIX®	IBM®	UrbanCode®
CICS®	IBM Z®	WebSphere®
Db2®	Redbooks®	z/OS®
DB2®	Redbooks (logo) ®	z/VM®

Preface

In this IBM® Redpaper publication, we focus on the importance of thoroughly testing software to ensure its quality. This paper explains how this testing can be achieved only in an effective and efficient way by automating the testing as part of a continuous delivery pipeline. It also introduces a framework that can enable such automation.

We specifically focus on Galasa. Galasa is an open-source deep integration test framework for hybrid cloud applications that allows teams to automate tests to run as part of a DevOps pipeline.

Galasa was built as an integration test framework to test applications spanning multiple platforms as part of a hybrid multi-cloud. It also integrates all the test tools that are needed to test such an application. This feature gives you a single test catalog, single endpoint to run tests and a single UI to review the reports from those tests. These enterprise-level features are key to unlocking the value of your automation and allow you to deliver your DevOps journey.

Authors

This paper was produced by a team of specialists from around the world.

Michael Baylis is a Senior Software Engineer in the UK. He has over 30 years of experience of systems programming, software development, and testing on the mainframe. Since joining IBM 14 years ago, he has modernized the DevOps pipeline and testing architecture for the IBM CICS® TS portfolio, striving to demonstrate that software development on the mainframe should not be any different from distributed or cloud platforms.

Russell Bonner is a Senior Software Engineer and CICS Technical Consultant in the CICS Development team, based at IBM Hursley, UK. In this client-facing role, he works directly with IBM Z® clients, delivering seminars and hands-on workshops to enrich customer experiences, and enhance IBM solutions by working closely with IBM development, sales and technical teams, and other key stakeholders. With experience that is underpinned by roles in mainframe application development and systems programming earlier in his career, he has presented on various aspects of CICS technologies and DevOps for IBM Z at numerous conferences worldwide. A member of the BCS, The Chartered Institute for IT, and a Chartered IT Professional holding a Certificate of Current Competence, Russell is also a Master Certified IT Specialist with The Open Group.

James Davies has been on the Galasa project since its inception as a developer and technical advocate for the project, driven by a keen interest on modernisation and automation. Starting his career in a distributed environment within IBM before joining the CICS team, he provides an insight and respect for the entire application stack for IBM z/OS® and cloud technologies.

Clive Harris is a CICS content developer with nearly 20 years of experience in technical communications. He holds a degree in Engineering Science from Oxford University.

Caroline McNamara is a content developer in the UK. She has 20 years of experience in the fields of business analysis, software testing, and technical writing. She holds an MSc in Information Technology from Birmingham University. Her areas of expertise include test management and information development. She has written extensively about IBM MQ, Internet of Things, and CICS.

Catherine Moxey is an IBM Senior Technical Staff Member in CICS Technical Strategy and Architecture, based at IBM Hursley in the UK. Catherine has over 35 years of experience as a software engineer, more than 30 of those with IBM. Her areas of expertise include CICS Transaction Server, IBM Z, event processing, and DevOps. She is currently focused on delivering the most compelling Developer Experience for CICS, and enabling applications that use CICS to participate in an agile and modern DevOps pipeline. Catherine frequently presents on CICS and related topics at conferences around the world. She has written numerous articles, papers, and Redbook publications about the capabilities of CICS, and has a number of patents and patent applications.

Richard Somers an IBM software engineering intern.

Will Yates is a Test Architect in the UK. He has 15 years of experience in the fields of software development and testing. He holds a degree in computer science from Portsmouth University. His areas of expertise include CICS, Software Testing and Test Automation. He has written extensively on modernizing CICS applications and automating tests.

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an IBM Redbooks® residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our papers to be as helpful as possible. Send us your comments about this paper or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  `ibm.com/redbooks`

- Send your comments in an email to:

  `redbooks@us.ibm.com`

- Mail your comments to:

  IBM Corporation, IBM Redbooks
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Follow us on Twitter:

  `http://twitter.com/ibmredbooks`

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  `https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm`

- Stay current on recent Redbooks publications with RSS Feeds:

  `http://www.redbooks.ibm.com/rss.html`

1 Automated testing as the key to continuous delivery

In this chapter, we review some of the background to software testing and the different types of testing, the value to an enterprise of using a continuous integration and continuous delivery (CI/CD) pipeline, why automation is vital in CI/CD pipelines, and the risks of not adopting automation.

This chapter includes the following topics:

1.1 Importance of testing

If software is not tested, how can you or your users have confidence that it behaves as intended?

1.1.1 What is testing and why do it?

Testing is the process of validating that software carries out the actions and provides the results that are expected. It can, and should, encompass several different types of testing that are carried out at several different phases during the development lifecycle of the product, and potentially continue after the software is released. This process allows errors and omissions in the software to be uncovered, and ensures that it meets the users' requirements.

Software testing can be described as verification of the system or application under test.

1.1.2 Unit testing

Unit testing is the earliest phase of testing. It validates that an individual unit or component of a software solution performs as designed. Unit testing typically is carried out by the developer who wrote the code within the unit or component under test, along with other early validation activities, such as static analysis and code review. That is not to say that the unit tests that are created by the developer might also be rerun at later stages in the software lifecycle to validate that the individual units still behave as intended.

Unit testing is important because the earlier defects can be found (or avoided), the fewer resources are required to resolve them. A problem that is found early might be corrected by simple editing in minutes, whereas the same problem found later might require much rewriting and retesting. Unit testing also often takes advantage of the developer's knowledge of the internals of the unit or units being tested.

The unit tests that are created as part of unit testing can also be important if they are designed in such a way that they can be run and rerun later in the development cycle. This process helps validate that the behavior of the software units is still correct.

1.1.3 Function, integration, and system testing

Beyond the realm of unit testing, which is carried out by the developer, are a range of types of testing that typically fall into the realm of the tester. These testing types validate that when individual units of code are integrated or introduced into a software system, the overall system continues to work correctly without any regressions and displays the new functions that the new code is intended to enable. Many terms are used to define these different types of testing. In most organizations, a sequence of test phases exit through which code passes, each with their own names.

A useful approach to categorizing types of testing is the Agile Test Quadrants, which were introduced by Brian Marick (for more information, see this web page). These categories consider tests in terms of distinguishing whether they are business facing or technology facing, and whether they are used by or on behalf of programmers to support programming, or are intended to critique the product.

The result is four quadrants of categories (see Figure 1-1):

- Quadrant Q1 looks at unit testing, and component and integration testing, all of which focus on the technology.
- Quadrant Q2 looks at more business facing and system level tests, such as functional tests and story tests.
- Quadrant Q3 represents tests that focus on the business-level capability of the product, and aim to critique or discover any problems with this testing, such as exploratory testing, usability, and user acceptance testing.
- Quadrant Q4 focuses on the system as a whole and how it meets business needs, but from a technology perspective, such as performance, load, stress, and security testing.

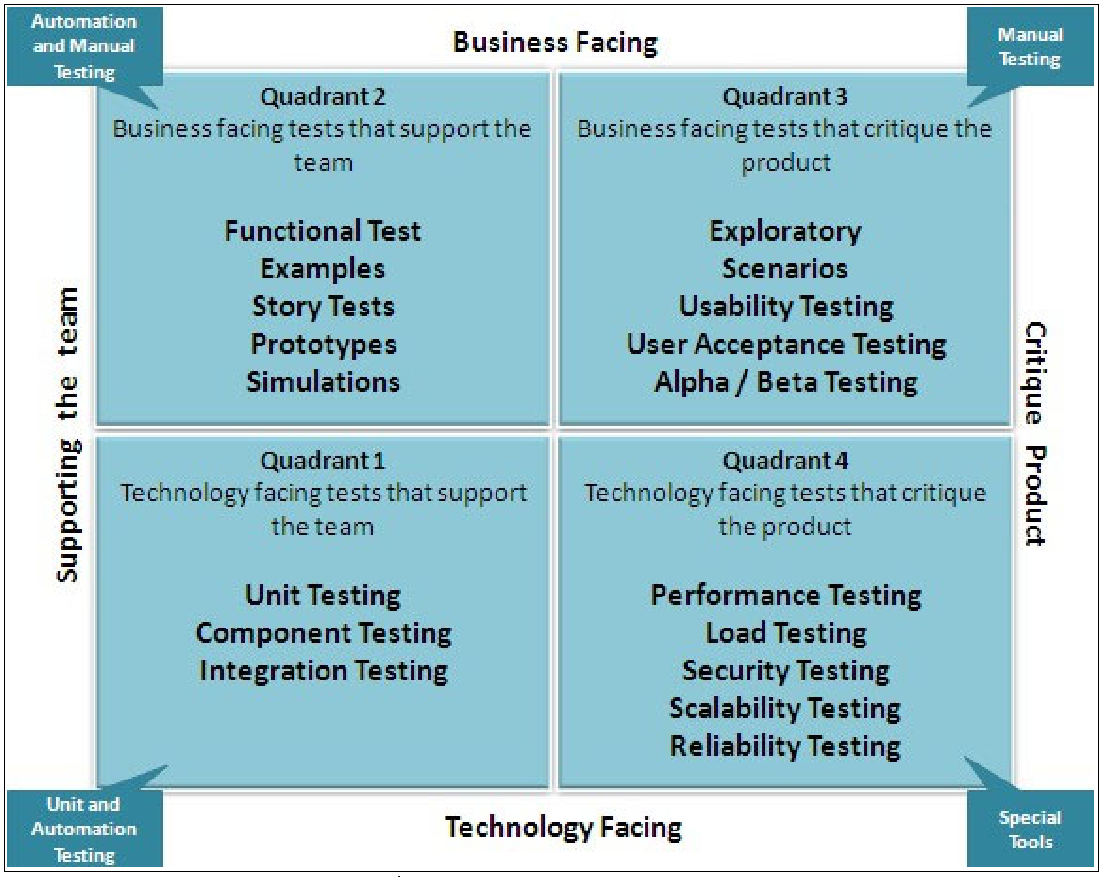


Figure 1-1 Agile Testing Quadrants[1]

Quadrants Q1 and Q2 often lend themselves to automated testing. Q3 requires a more exploratory and manual approach, and Q4 might require tools that are focused on testing for performance or security, for example.

Martin Fowler and Mike Cohn discussed the concept of a Test Pyramid (for more information, see *The Practical Test Pyramid*), which emphasizes the importance of a wide base of many small unit tests, then built upon that a set of equally important but less numerous tests that Mike Cohn called “service tests”.

[1] Source: `http://tryqa.com/what-are-test-pyramid-and-testing-quadrants-in-agile-testing-methodology/`

At the top of the pyramid that is shown in are "end to end tests", which include user interface tests, that test the entire system. Service tests cover a similar scope of testing as integration testing, but is a term that gained little traction. In our usage, integration testing covers all of the pyramid above the unit tests.

This paper does not attempt to provide definitive definitions of the various types of testing, but uses *integration testing* as an umbrella term to cover much of the testing that occurs after development and lends to automation. It is here that individual units of software are tested together and with other components, including external parts of the system. Such tests are usually run in an environment that matches some aspects of the ultimate target environment for the software. Included within integration testing are regression testing, functional testing, system testing, U.I. testing, end-to-end testing, user acceptance testing, and performance testing.

A distinction must be made between types of *testing* and types of *tests*. For example, unit tests can (and should) be run during later test phases, especially as part of regression testing.

1.1.4 Role of exploratory testing

Exploratory testing was defined by Cem Kaner in 1984 (for more information, see *A Tutorial in Exploratory Testing*), as:

> *A style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of their work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project.*

Exploratory testing allows a tester to use their skills and experience to discover, investigate, and learn about the behavior of the software under test. In the spirit of the Agile Manifesto, it emphasizes the "personal freedom and responsibility of the individual tester"[2].

In terms of the Agile Testing Quadrants, exploratory testing lies toward the side that aims to "critique the product"[3]. Exploratory testing is often referred to as being a "thinking" activity. It is also sometimes referred to as ad hoc testing, but in reality, it is a much more directed and organized activity than being purely ad hoc.

This type of testing makes the best use of the skills of the tester, but clearly by its nature does not lend to automation. The value of automation of testing lies in freeing the tester from the need to manually conduct repetitive and unthinking testing.

[2] https://www.guru99.com/exploratory-testing.html
[3] https://www.testingexcellence.com/exploratory-testing-important-agile-projects/

1.2 Continuous integration, continuous delivery overview

CI/CD are important practices within a DevOps approach, which allows software to be evolved and enhanced at a rate that meets the needs of the business and users rather than being artificially delayed by long testing cycles. These cycles, in turn, can result in a grouping of the delivery of software changes.

1.2.1 Introduction to DevOps and CI/CD pipelines

At its heart, *DevOps* refers to a building a greater collaboration between the software development and IT operations teams within an organization. By working together, the systems development lifecycle of building, testing, and releasing software can be shortened. Software changes also can be continuously integrated and delivered to provide value more rapidly and reliably.

Continuous Integration

A key step in adopting a successful DevOps approach is to set up a CI/CD pipeline. *Continuous Integration* (CI) is a technique that was first identified by Grady Booch, that involved frequent checking in of small code changes that are made by a development team, which are merged into a “master” code stream.

CI provides a consistent way of building and packaging changes and validating that they work together, which encourages teams to commit changes more frequently.

Continuous Delivery

Continuous Delivery (CD) progresses the code changes further along the pipeline process by automating the delivery of the changed software to a series of environments for testing, and ultimately production. Some people distinguish between Continuous Delivery (which ensures working and tested releases of software are ready at any time to production but requires a manual decision process before that final deployment) and continuous deployment, which also automates the releasing into production.

All of this requires Continuous Testing to ensure that quality software is being made available at each stage, and ultimately delivered to production. Continuous monitoring of the software in production, continuous feedback from stakeholders and users, and ultimately “Continuous Everything” also are valuable.

Making everything as continuous and automated as possible is what a DevOps pipeline aims to achieve. Such pipelines are often represented pictorially as a funnel, with code units being fed in at one end, passing through various phases of building and testing within a sequence of environments, and delivered into production at the other end.

We prefer to represent the pipeline as a cyclical and iterative process, where developers write, build, and unit test their code. They repeat steps as needed, and then, those units are fed into other cycles of integration and system styles of testing. Following release into production, the software system is continually monitored, and enhancements are planned. This process results in the cycle being repeated. This view of the pipeline is shown in Figure 1-2 on page 6.

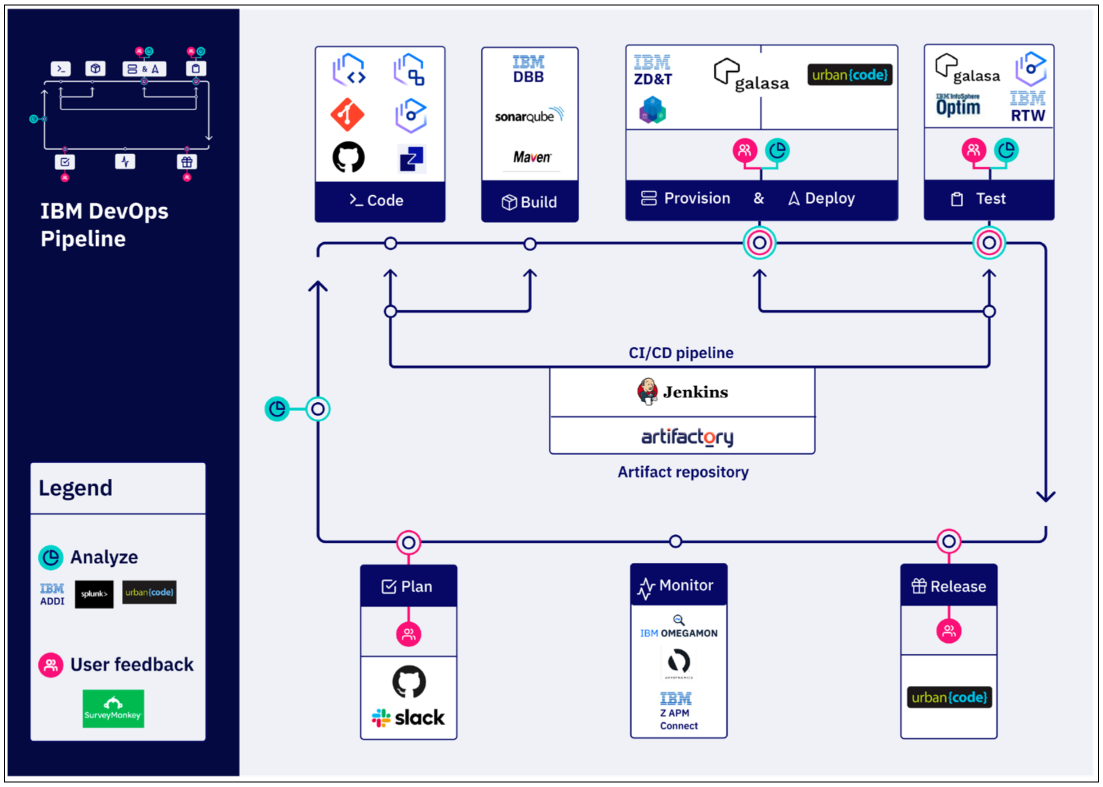


Figure 1-2 Representation of a DevOps pipeline

As shown in Figure 1-2 after planning for the next release based on user input and potentially analyzing the code to understand where to introduce the changes, the coding phase begins. The developer writes, builds, and unit tests the code, gradually adding function and ensuring that it works as an individual unit as intended by using their preferred integrated development environment (IDE) and tools for source code management, dependency resolution, and so forth.

When the code is ready, it is delivered into the pipeline, which uses an artifact repository to manage the process. The software now enters the testing phases, where the test environments are provisioned (or might exist), the code changes are deployed into the environment to be tested, and tests are run.

The tests might drive the provisioning and deployment, or this process might be done separately. This phase of the pipeline is an iterative process that moves through different levels of testing, often in different environments.

When failures occur, they must be efficiently diagnosed, and if needed, the code is amended, rebuilt, and unit tested again. Feedback from users is sought during this phase to ensure that what is being delivered meets their needs.

When the code changes successfully pass all of the required phases of testing, it is released to production. The software changes and the production environment continue to be monitored, and planning for the next release begins. Figure 1-2 shows some of the tools and products that might be used to implement the various stages of the pipeline.

Some of these building blocks are likely in place in many enterprises, but it is likely that most of the activities are carried out in a manually. It is the lack of automation of those steps that slows delivery. A DevOps pipeline can be efficient, or indeed practical, only if it can be automated.

1.2.2 Focus on efficiency and automation through the pipeline

The driver for building a CI/CD pipeline is to increase efficiency and speed of delivery. Therefore, all of the steps in the pipeline must be quick and reliable, and the only realistic way of achieving this goal is through automation.

1.2.3 Tests as the quality gatekeeper through phases of the pipeline

As a software change is delivered through a CI/CD pipeline, it is important to ensure its quality and readiness to move on to the next phase. Testing is the gatekeeper that can give confidence that this is the case and can also flag up where it is not.

This testing also must be carried out continuously as each change is delivered. *Continuous testing* is described as being a process of *"testing early, testing often, testing everywhere, and automate"*[4].

A key tenet of the move to DevOps and a CI/CD pipeline is that as much of the testing as possible must be automated. The process that is used to checking whether the tests passed also must be automated as much as possible.

However, not all testing can or should be automated. Exploratory testing might well follow different paths as a result of what is discovered during the process. Penetration testing often relies on innovation and trying something new, which is difficult to automate. User acceptance testing often involves users interacting with the system in a flexible way.

Testing versus checking

Testing is an interactive activity that involves evaluating whether software meets its purpose by exploring and experimenting with its behavior. This process requires human creativity and cannot be automated.

What is possible to automate is the checking that the software meets the expectations that were discovered by testing. When we refer to *testing* in this publication from this point on, we are referring to what is more strictly called checking rather than testing.

The key point is that anything that lends to automation should be automated to allow time for these other activities where automation is not suitable. The fact that something is difficult to automate cannot be used as an excuse for failing to automate it.

[4] https://www.guru99.com/continuous-testing.html

1.2.4 Managing the pipeline when test cases fail

If the tests are automated but checking or reacting to their results is not, then that is only half of the story. When building a CI/CD pipeline, thought must be given to what occurs when test cases fail.

The diagnostic tests that reveal the cause of the failure must be collected in a known location and made easily available to the engineer who reviews the problem. The software change that caused the failure must be easily identifiable, and its progress through the pipeline must be halted, or reversed.

Other software changes that did not cause the failure must continue unimpeded, unless they are viable only with the failing change, which must also be something the pipeline can detect and act upon. Finally, it is possible that the failure can be caused a problem with the test or test environment, and those possibilities must also be easy to detect and investigate.

Considering how the pipeline checks for pass or failure is more important than you might think. We learned of an example where tests ran automatically and published a report to an external source. The status from the tests indicated pass or fail based on whether the test ran and successfully wrote a report. Over time, the implications of a “pass” were forgotten, and tests were failing without anyone ever knowing or checking.

1.3 Testing on z/OS

Many organizations chose IBM Z for the most critical aspects of their businesses, especially in industries including banking, insurance, and retail, where disruptions cannot be tolerated. Therefore, in addition to relying on the inherent reliability, security, and resilience of the platform, such companies conduct extensive testing before introducing any change.

1.3.1 Importance of testing on z/OS

Because of the mission critical nature of the software and applications that are running on z/OS, testing is vital to ensure that services can be provided uninterrupted, and that any change works as intended and does not affect anything else. This notion makes testing arguably more vital on z/OS than on any other platform.

The terminology that is used to describe the various phases of testing of z/OS applications indicate the importance that organizations attach to this testing, with terms, such as “quality assurance” testing, “pre-production” testing, and “user acceptance” testing, and the extensive phases of regression testing and performance testing that are required when any change into the system is introduced.

The estimates for any project that makes changes to an application on z/OS (or for a new application) must include a large portion of effort that is allocated to testing.

1.3.2 Challenges for test automation on z/OS

With such extensive testing required, it might be thought that test automation is widespread on z/OS. However, test automation on z/OS historically was proven to be difficult.

Surveys and user research that was conducted by the IBM CICS Transaction Server for z/OS organization showed that 92 - 95% of testing on the platform is entirely or mostly manual, which is in line with industry estimates that place the percentage of manual testing at approximately 80%. Manual testing can vary in nature from a test suite that must be set up and run manually, to entering in a sequence of steps that are described in a hardcopy book of test cases.

Test automation is a challenge on z/OS for the following reasons:

- Large systems were built up over the years, in terms of the number and size of components that making up each application, and the environment in which the applications run. Finding a way to drive these large systems as part of an automated test proved challenging.
- A lack of test automation tools exists that understand of the z/OS operating system, its subsystems and file stores, to make it practical to adopt these tools.
- The data is tightly integrated with the applications that use it, and is often used by multiple applications. Providing suitable test data that can be isolated for use by each test run and reset to known values is especially challenging.

Many applications rely on components that were developed many years ago, which means that testing must ensure that these components still run without issues or regressions. The difficulty in achieving this goal resulted in falling back to manual processes and checks, and as a result continuing to use waterfall processes.

If the testing cycle takes a long time, developers are tempted to group many changes, which is counter to the idea of continuous integration.

1.4 Why testing matters

Because organizations that use z/OS for their mission-critical systems enjoyed growth and success for many years, it matters if most of the testing is done manually. Today, it is important to respond rapidly to new market opportunities and threats, and be confident in the quality and robustness of the services that you provide.

1.4.1 Value of automated testing in continuous delivery

If testing cannot be automated, it is difficult (if not impossible) to gain the benefits of continuous delivery, which reduces the agility of an organization and its ability to react and innovate quickly. Even responding to a new competitive threat or compliance regulation might not be possible without a CI/CD pipeline and automated testing.

1.4.2 Dangers of not automating testing

If an organization continues to test manually, they run the risk that the test cycle is too long to enable the required agility and speed to market, or the amount of testing must be reduced, which leads to uncertainty over the quality of the software being delivered, or perhaps both.

1.4.3 Risks of not adopting continuous delivery

If an organization does not adopt a continuous delivery approach in which small changes can be constantly delivered and tested, tendency occurs to save all of the changes until enough were made to justify the large effort that is involved in testing a software release. Not only does this issue mean that users endure a long wait for new function to become available, but if problems occur, it is difficult to identify the cause and isolate the failing component.

Even more damaging can be the effect of not delivering functions rapidly into the marketplace and missing opportunities as a result.

1.4.4 Use of automated testing for system reliability

With automated testing in place, it is easy to run the checks that ensure the software is working as expected. Therefore, whenever a change is introduced into the system, such as applying maintenance, a hardware upgrade, or updates to another component, automation is available to verify that everything works as expected. Such changes can be made more easily and with more confidence.

1.5 Achieving test automation on z/OS

In this chapter, we painted a somewhat gloomy picture of the state of the art on z/OS, but this picture does not have to be the case. Many organizations built, or are now building, CI/CD pipelines for their z/OS applications with considerable success.

These teams are looking to test automation to help them achieve an efficient pipeline. This IBM Redpaper describes an approach to test automation on z/OS by using a framework for automating tests that offers deep integration with z/OS capabilities.

2

Test automation in a hybrid cloud world

Modern day applications seldom operate in isolated in environments. Instead, they commonly interact with many other services by running in various environments. An application can be found to span across mainframes, cloud services, and other locally running x86 technologies. Every technology that is involved can all have its nuances, difficulties, and complexities, especially in terms of testing.

Hybrid cloud applications can be difficult to test fluidly. The more components that are used in an application, the more technologies and tools that are required to test reliably.

In Chapter 1, “Automated testing as the key to continuous delivery” on page 1, we saw how test automation is integral to continuous delivery. In this chapter, we expand on this idea and how this is commonly achieved in different environments and what can we learn from this process.

This chapter includes the following topics:

2.1 Approaches to test automation across modern compute environments

In this section, we describe testing tools that are available across different environments. We discuss some of the current market requirements for business and development needs in each case.

Note: As we review different environments, it is important to remember the role of such environments and their testing needs differ.

2.1.1 Cloud

As one of the most modern environments that we see developers work in today, it is not surprising that we see the most natural uptake of continuous delivery and other modern methodologies.

The design and architecture of cloud-based applications better supports a “build up and tear down” deployment style. The small moving pieces or microservices that make up a cloud-based application are easily isolated and replicated, which eases some testing complexities.

Tools, frameworks, and practices for test automation are well established in this area. Components can be brought up, external calls and inputs can be mocked, and the core focus of a component can be tested. It is also possible to start other components of an application to further dig and test in an integrated style.

Test cases can grow in complexity with each component being deployable and managed in the same manner. It is the containerization of these applications or components that allow for these CI/CD methodologies to be so fast moving and dynamic.

Containerization of components and services also lends to the immutable nature of these pipelines. As a build progresses further through a development pipeline, rebuilding is not needed. This enables a much more granular level of development than a classical application because components of an application are not blocked by others.

Often, the underlying orchestration that is hosting these applications is Kubernetes, which is beyond the scope of this publication. However, you might want to familiarize yourself with some of the deployment methods it uses.

In Figure 2-1, we can see a common example of the following types of flows or pipelines developments that teams run through:

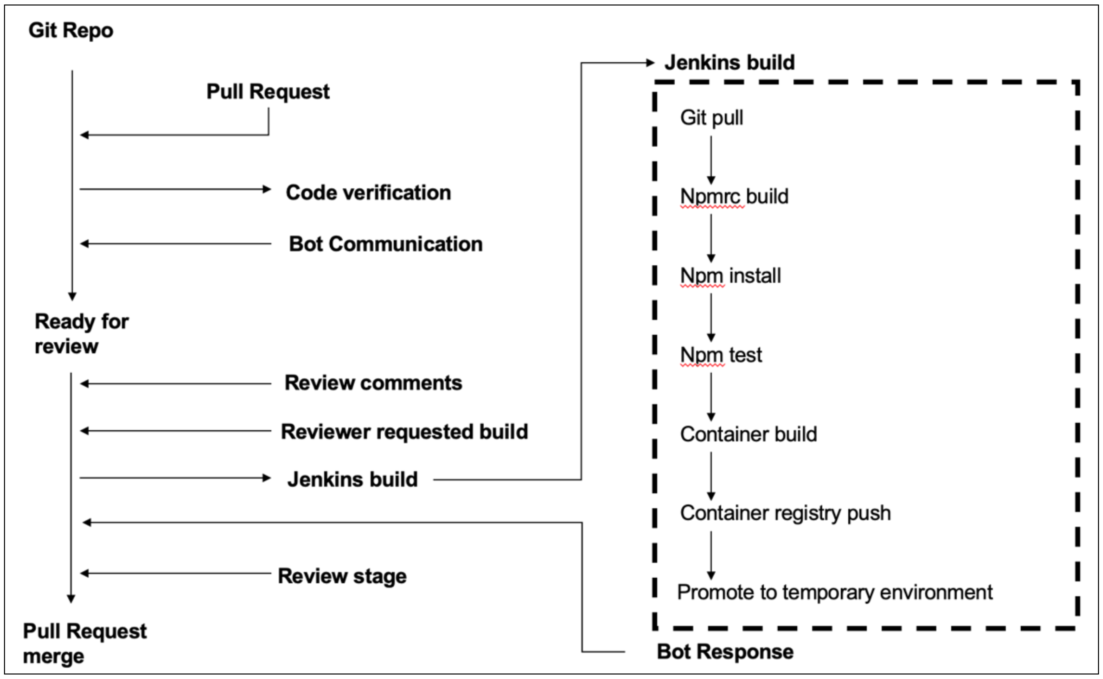


Figure 2-1 Common flows and pipelines

1. Pull request

   A developer submits their changesets that are ready for review and merge into the project.
2. Code verification

   A validation step to see whether the changesets meet criteria. This process can be as basic as validating that the changesets contain code changes.
3. Bot communication

   The automated system can tag the people that are required for review, as is commonly seen in many GitOps.
4. Review comments

   This process is normally a classical code review. Another developer or lead manually checks the code.
5. Reviewer requested build

   Usually another GitOps transaction to begin a build to check the code in a hosted test environment.
6. Jenkins build

   A pipeline that understands how to build, perform basic IVT style tests, and start a temporary test or review environment that is hosting the changes.
7. The final review section is for both the reviewer and the original developer for more testing and review that the changes are performing as expected. It is at this stage where more automated testing can also be directed at this environment.

However, this process typically is not the end of testing; instead, it is a quick verification that the change set is not going to break the application.

If the pull request is merged into the repository, any pre-production environment can update the application components to automatically pull in the new changes. Staging this application from the preview to pre-production does not require another build as the image can be progressed (see Figure 2-2).

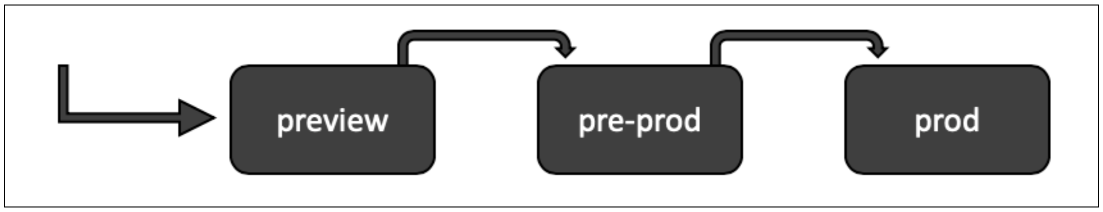


Figure 2-2 Staging

As we promote between these stages, repeated opportunities for testing are available. Typically, the more expensive a testing phase is, the further right it is occurring, which reduces the chances of smaller issues causing expensive delays.

However, hybrid cloud applications do not follow such a smooth experience. If some of the components of an application are running in a z/OS environment for performance, security, and scalability reasons, we cannot adopt some of these tools and techniques. These differing environments bring complexities when trying to integrate all components into the same pipeline.

We see less support for dynamic deployment of applications into a z/OS address space, but this is also not as realistic. This “building up and tear down” nature is not always the most efficient or practical for the design of the z/OS component.

2.1.2 Microservices

It is imperative to say that microservices and containers are not the same thing. However, no hard rules exist for what constitutes a microservice. Even so, a soft definition exists; that is, small as possible but as large as a necessary.

First, we define a microservice architecture as a set of loosely coupled services (usually REST), which together perform some business capability.

But how do you test microservices? We commonly see it broken down into the five areas that are shown in Figure 2-3.

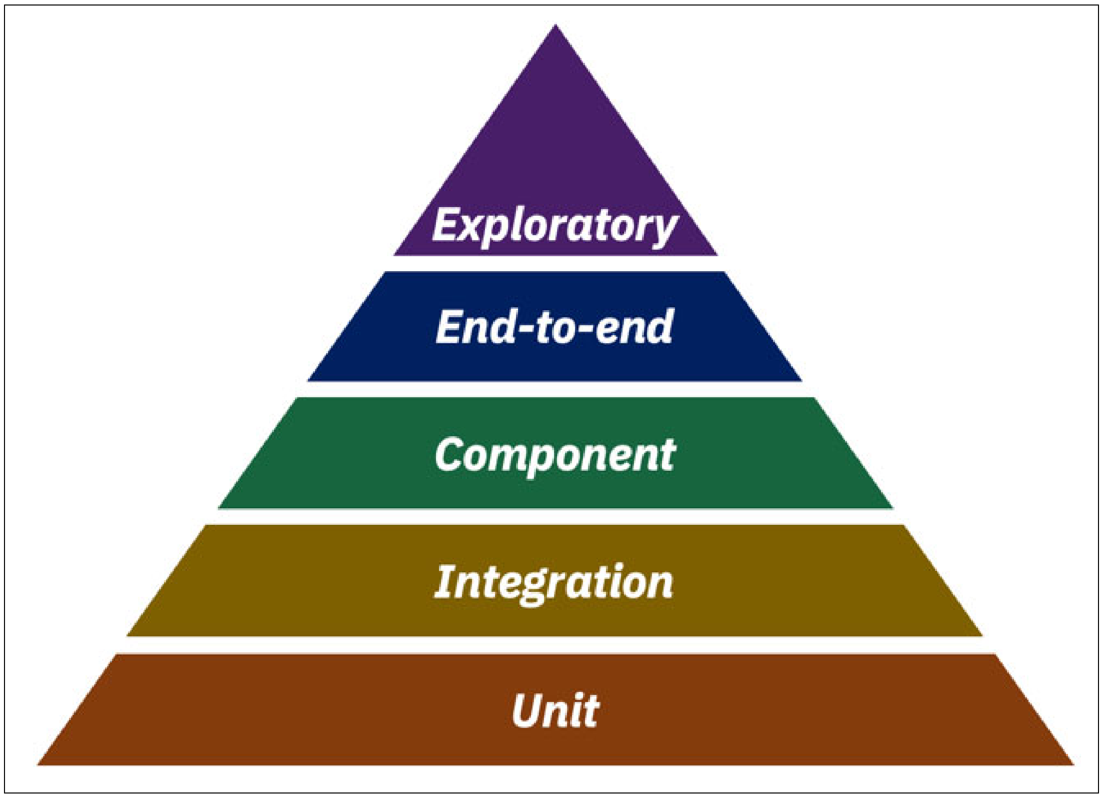


Figure 2-3 Testing microservices

These sections are broken down into Mike Cohn's testing pyramid to help visualize the number of tests we expect to see in each area:

- At the unit testing level, we expect to test the individual microservice at a developer level. The tools that are used differ from languages and frameworks.
- The integration level ensures that the collection of services interact in such a manner to produce the wanted business logic. Martin Fowler expresses for microservices that an integration test "exercises communication paths through the subsystem to check for any incorrect assumptions each module has about how to interact with its peers".[1]
- Component testing is about isolating the individual microservice to test, mocking all other network activity, including other microservices and databases. This allows us to quickly test the functions of a component; however, some larger application errors can be missed because component testing does not account for the asynchronous aspect of each service.
- End-to-end testing covers what component testing misses. Treating the entire application as a "black box" to truly test if the chain of microservices provides the business functions for which they are aiming.
- Exploratory testing is the final stage (and the most varying) because this testing is down to the application and tester searching for potential issues.

[1] https://martinfowler.com/articles/microservice-testing/#testing-integration-introduction

A microservice approach to an application has similar advantages to cloud applications in terms of testing. Having isolated components that can be deployed in a similar manner breaks down the components of an application that need to be tested. As such, we see similar pipeline approaches to test automation, but with the supported tools that depend on the applications architecture and design.

As with similarities, we also encounter the same problems with incorporating z/OS microservices into an application. The components are not as easily to test in isolation, with technologies surrounding mocking are greatly less supported. This is exacerbated by the fact any tool that can test the z/OS component generally is not the same technology that is used to test the rest of the application. This issue creates a distorted build and test ecosystem that feels like two halves.

This issue can result in the end-to-end and integration style testing to be left until later in the development cycle because of its difficulties. Finding defects this late in the cycle can lead to some costly development effort.

2.1.3 Heterogeneous and distributed systems

As more applications embrace a hybrid multi-cloud reality (that is, an application that is composed of services that are running across public, private clouds and on-premises hardware), it is unlikely that an enterprise application runs on a purely homogeneous environment. Although each component can be tested in isolation, any attempt to test across these systems requires the test to have first-class access to all the systems within the application. This challenge is not for only the framework that is running the tests because the tester must interact with various platforms to holistically test the application.

2.1.4 Mainframe and mission-critical

Unlike the cloud environment, this area is likely where we see the least adoption of modern testing methodologies and CI. But, is this result for a good reason?

Mainframes and mainframe applications are commonly well-established and adverse to rapid movement and risk. Failures within this area tend to be expensive and highly damaging, whether it is to the application or the reputation.

This issue did lead to a culture of “if it's not broke, don’t fix it,” which is an enemy to older applications. In such a risk adverse environment, how are applications tested and pushed through to production?

It is not uncommon to see application processes be heavily manual, especially on the testing front. Languages, such as Cobol, Assembler, and PL/X do not fit into many unit testing frameworks. Also, integration testing and above can be difficult to automate.

However, that does not mean we do not see any tools in this space, IBM’s Z Unit Test is an automated unit testing tool that addresses the difficulty in unit testing batch and CICS programs. In terms of producing pipelines, IBM’s UrbanCode® Deploy can automate some processes, such as defining environments and installing resources, which can then be incorporated as part of a CI/CD pipeline. This process does not always feel the most smooth or natural, with some parts of the process requiring specific and brittle scripts to perform work.

It is probably safe to say that the deployment and testing mechanism of z/OS applications cannot compete with the dynamic nature of cloud or microservices applications. The availability and cost of tools that supports and modernizes z/OS application development is a major driving factor in this area.

2.1.5 Web applications

Like microservices, testing web applications features its own requirements to ensure the usability of application. A typical checklist can consist of the following items:

- Functions testing ensures that all links behave as expected, forms can be interacted with and are operational, cookies are working, and so on.
- Usability testing covers the user’s experience, including navigation, ease of use, and content checking.
- Interface testing checks that all interactions between servers are performed, with errors being handled. Main interfaces to be tested are the web server and application, and the database and application interface.
- Compatibility testing checks the compatibility of different browsers and operating systems, platforms (mobile), and printing options.
- Performance testing checks apply load to the application to ensure it can handle large numbers of requests and scale to business needs.
- Security testing ensures that the application meets security needs and has no known vulnerabilities.

Selenium is a popular and somewhat industry standard. It is defined as an open source automated testing suite for web applications across different browsers and platforms and is commonly seen plugged into CI pipelines.

This area can be easier to involve z/OS components without effecting some of the testing methodologies. REST endpoints and web applications that are served from z/OS behave in a similar fashion to applications that are hosted elsewhere. Therefore, tools that can be used to test APIs and web applications work for a more hybrid environment.

However, the complexity that is found here is around validation. If we have a web-based API that is used to update an application; if this application is a z/OS application, how do we interact with the backing application and middleware to validate the responses that are provided from the API?

2.2 Systems that are difficult to test

A system’s difficulty to test depends on your definition of how something is tested. In the case of 3270 applications, is a test user logging on to a system and then following a script of commands? Entering data into fields and moving between windows is not difficult, but this example is common of an application that is difficult to test in automation. So, why is this difficult?

To write code that can communicate with 3270 windows, understanding the data streams that are sent back and forth is a complex task, and this understanding is needed before any test code is written. After such test code is written, interactions occur with datasets, batch jobs, application logic, and other middleware.

With every layer of complexity, code requirements to understand and interact with each service or component starts to become unmanageable. Unless all of your testers have intricate knowledge of every subsystem and service, writing even the most basic task becomes laborious.

These 3270 interactions can be a small fragment of the overall application. If we have a large-scale application that starts spanning multiple systems, or even multiple different environments for z/OS to cloud based technologies, this level of complexity moves from unmanageable to near impossible.

This example is an exaggeration because work to provide certain functions often is provided from our library of testing tools. However, finding the best tool for these hybrid cloud applications is an important task. It is unlikely that one tool is found that can cover every aspect of testing requirement.

2.3 What makes a good automated test?

A good automated test includes the following components:

- No hardcoding

  Observing tests that can fail from day to day because where the tests are run changed is something that should be avoided. Dynamic aspects of a test can be hostnames, ports, test data, and so on. If something can change or vary, a test should expect as such and compensate for it.

- Reporting

  When running in automation, you are not going to be monitoring a terminal output. Any failures that occur are going to be difficult to pinpoint if messaging and reporting are not well implemented. On a test failure, we want to see clear error messages, artifacts that were used, and logs from other services that were involved. This information leads to fast error analysis, and any automated test does not need to be run manually.

- Limit code duplication

  If we use a process that must be repeated for test setup or sign on processes, they must be extracted outside of the test and shared. This extraction prevents test material from becoming outdated and faulty if any updates are done.

- Immutability is required

  We do not want to see our tests passing in a test environment, but then not work or even run in our pre-production environment. If a test includes requirements, they must be part of the test, not something that must be configured or installed per environment.

- Isolation

  As testing scales, isolation becomes a larger factor. If we use a framework with multiple instances that are running tests, and by chance the same test is running congruently, we do not want to see one failing while the other one passes as they had a requisite on a unique resource.

- Cataloging

  Although cataloging is not inherently part of the test, it is important. Without test cataloging, it can be unclear what tests must be run. At times, you want to run your entire test corpus across your application, but this should not be every change set. If tests are cataloged into product areas or application functions, relevant tests can be run against changesets.

2.4 What makes a good automation tool, and where are other tools let down?

If we know what makes a good test, what makes a good test tool? The answer can depend on the requirements. A professional grade wrench is not going to help you to work with a screw, and the wrong automation is not going to meet business needs, regardless of its quality.

However, we want to see the following common traits within the tools we use:

- Flexibility

  A socket set is going to get you much farther than a single wrench. Specialist tools are not always required, but they exact a heavy toll.

  It is not uncommon to see home-built solutions to a specific problem, which over the years became convoluted, difficult to maintain, and an uncomfortable experience to any non-experts and beginners. This issue often is the result of technologies changing, requirements shifting, and team members evolving.

  A tool that is flexible enough to integrate with other tools in a common way is going to help smooth even the busiest pipelines.

- Scalability

  Test requirements are not static. Everyone knows that the closer you get to release time, the more testing that is done. As an application grows, these requirements grow and shrink as the business needs it; so, too, should your automation.

  As the amount of work increases, workloads should scale horizontally, which prevents the long queuing, or slowed testing.

- Reliability

  No one wants to spend more times maintaining their pipelines, or worse, become stuck waiting for it, than they want to spend developing and testing. A developer wants to know that the change set they pushed is built and tested.

  The same applies to the test material. We do not want our testers' time taken up writing cases that are difficult to maintain and are specific.

  We prefer the cases that they do write and evolve with the application they are testing. We want to remove the hardcoding of endpoints and data and have them bound to the test at run times. We also want environments to be provisionally dynamically or bound to a dedicated testing environment and for the test to not care. If we see a test failure, we want to know it is because it caught a problem, not because a brittle test was run at the wrong time in the wrong place.

- Isolation

  As an automated system grows, so does the chance of collision if not accounted for. For example, we can have a test that pulls a piece of data from a data lake. If this piece of data is not protected, another test can also use the same data, which compromises both tests.

  What we want is our test run in logically isolated environments, and any interactions with other resources to be protected and locked from interrupts.

2.4.1 Common pitfalls of tools we see today

One of the largest pitfalls that is seen in many successful tools is a lack of awareness to the bigger picture. We see many tools that do “their thing” well and have an excellent understanding of their specific area, but then commonly have little way to integrate them with other tools or larger, more complex environments. For example, having an API testing tool that can hit a CICS web service that interacts with a back-end IBM Db2® database is useless unless it can validate with that backing Db2 data.

Extensibility also can be an issue. Trying to get a testing application to support some new tool can be an issue. Unless the development teams are responsive to upgrade requests, or you have access to the source to implement a solution yourself, the process can take time. This issue makes it difficult to always use the tools you want to use rather than the tools that are supported. The freedom to choose is important, especially when personalizing testing practices to an application.

A test tool is not a test framework. Without an umbrella structure uniting all of the tools that are used to test code, it can be difficult to orchestrate complex test cases while monitoring tests, runs, resources, environments, and so on. In-depth integration style testing becomes a long and arduous task without the suitable foundations.

2.5 Role of open source

What best for your test automation tools? Open source or a paid offering?

Both options feature different benefits. Open-source offerings have the advantage of cost, but sometimes lack in official support packages. Paid offerings are generally polished products, but can lack in extensibility or integration with other products. Therefore, the issue becomes on of what is most important for your environment and situation.

When we look at a modern pipeline, we are seeing specialist tools coming together to provide the production line for your code. No matter your preference of orchestration, build tool, code scanner, and so on, it is vital to find tools that can communicate with each other. However, nobody wants to be tied to a vendor because they use one of their products. Developers and engineers must choose to use the tools that are best suited for the task.

Consider Jenkins as an example, which is an open source automation server that can be used to construct complex pipelines that were successful. A plug-in nature that offers easy extensibility with many options made it simple to incorporate your favorite tool into your automation pipeline. This ease of extensibility, access to the source, and large number of provided plug-ins drives a community around the world. They constantly are pushing out the latest code to support even more tools.

It is the community you see around open source that realizes so many benefits.

Code that is open to many eyes of scrutiny discourage lazy or sloppy code because reputation and appearances are important to open source software. This issue also results in vulnerabilities that are spotted and patched with efficiency.

Another more unusual outcome of an open source offering is the number of experts that are willing to offer help that are outside the original development teams. A product that has a lively and thriving community is efficient and has the potential to move with an agility and directed evolution so keep up with the constant development of the industry.

These communities and open source software shaped modern day test automation pipelines, as shown in the example in Figure 2-4.

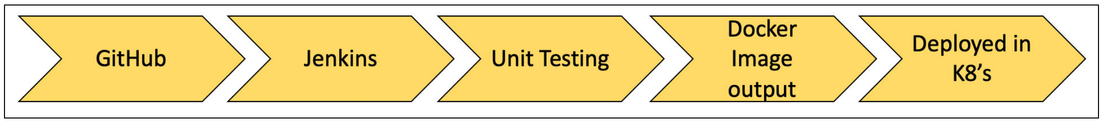


Figure 2-4 Test automation pipeline

This simple pipeline highlights the fact that GitHub, Jenkins, Docker, and Kubernetes are all open source.

2.6 Evolution of test tools

It is no surprise that testing has been around as long as programming and computers. However, what is surprising is that the term *bug* can be traced back as far as 1878, in letters written by Thomas Edison, explaining his mechanical system faults as bugs. But from that time, where has testing and testing tools gone since?

Through the late 1940s and early 1950s, we start to see some of the more famous examples of computer testing, such as the Turing Test, or Jurans Quality Control Handbook, which highlights even from the early development of a modern computer how important it is to test the results of output.

Focusing on test automation, we start to see a movement in the late 1980s with companies, such as Segue Software, and their test automation tools, such as SilkTest, being founded. However, if we look back to 1998, we see one of the names entering the field. STAF and STAX were a huge player in this area for the last 20 years and for good reason:

- It is an open source project that runs under the Eclipse Public License. This feature allowed people to take the source and implement in such a way to match their infrastructure.
- It used a compartmental design. It relied on this idea of creating reusable services to perform specific tasks, including process invocation, monitoring, and logging.
- It encouraged collaboration. Not only was STAF an open source project, it actively looked to its users to contribute to the project. This effort resulted in various supported languages and many features.

As popular as STAF and STAX were, it has not seen an update since 2016. So, what changed that caused people to find alternative methods?

In the last 5 years, an industry shift occurred toward containerization, especially as an architecture to creating applications in a modular fashion. As is the nature with containers, they are as small and simple as possible, which instantly makes tools, such as STAF, seem unappealing. Nobody wants the overhead of an agent on their containers.

In addition, some of the features that STAF offered to keep an infrastructure current and compliant are not required in a containerized world. When any maintenance is required, a new container image is produced and replaces the old container.

This concept of rapid deployment and tearing down of environments changed testing ideologies and methodologies. If we look at a modern style development setup, we commonly see CI/CD pipelines. Code that is flowing through these pipelines interact with several different tools at a number of stages. As a result, pipelining tools are all about extensibility, which we see in the form of plug-ins for automation servers, such as Jenkins.

Jenkins does not build or test any code, but its use collectively orchestrates the process. Jenkins uses a build tool, such as Maven; a Scanning tool, such as SonarQube; a test runner, such as JUnit, to take a developer’s code to a usable output.

Testing an enterprise application requires test capability across a range of test technologies, platforms, and practices. It is unlikely that any single piece of open or closed source software is likely to provide a high-quality solution to all of the requirements of an enterprise tester. However, the most promising open source solutions have shown that because of their open nature, they can integrate to provide capability that can span an entire enterprise application, each tool focusing on their particular set of capability.

As automated testing and the technologies that are used to enable enterprise applications mature and develop, new test tools are created to allow these applications to be tested. However, it is critical that these tools can integrate with existing test tools. Galasa acts as an integration test framework that allows all of these tools to integrate, share data, and allow a single interface for test execution, monitoring, and reporting.

3 Enterprise test automation

Thus far in this publication, we discussed the nature of enterprise testing and how open source can provide a flexible set of tools to allow you to construct a set of automated tests to validate the quality of your application.

However, writing the set of automated tests and checks that you need is not everything that you must consider; controlling the management, selection, scheduling, and reporting of those tests is as important.

Taking these points into consideration as early as possible helps guide your automation strategy and saves you from much rework later. This issue is not only test automation, it is enterprise test automation.

This chapter includes the following topics:

- 3.1, “Cataloging the test corpus” on page 24
- 3.2, “Scheduling tests” on page 27
- 3.3, “Test artifacts” on page 29
- 3.4, “Reporting tests” on page 30
- 3.5, “Summary” on page 31

3.1 Cataloging the test corpus

A *corpus* is a collection of written texts or a body of writing on a specific subject. By using this definition, it is fair to extend its definition to include a collection of written automated test code that is used to test your applications.

Understanding not only the quantity of the tests that are in your test corpus but what those tests do is key to managing your test automation. The simplest way of gaining this understanding is through a catalog of available tests. For each piece of test automation that is available, a catalog entry exists to describe that test and contains information, such as the following examples:

- Maintainer
- Versions of the software supported by the test
- Required environments
- Required data
- Priority
- Areas tested

The catalog should be used each time tests are run. The catalog is used to select the correct tests based on the area of code that was tested, the version of the software that is run, and the priority or importance of those tests.

It must be understood that the catalog is useful only if it is updated each time automated tests are created, updated, or deleted. Failing to keep the two synchronized degrades the capability of your tests to validate a build of the software because you are increasingly unable to find and run the correct set of tests.

The test corpus should be machine and human readable. This feature allows for engineers and continuous delivery pipelines to parse the corpus to understand which tests should be run to validate a specific build. Ideally, the corpus should be stored in a single format that can be directly parsed by software, but also used within a user interface for a test engineer to understand. Formats, such as XML or JSON, are useful for this purpose because they can be easily machine parsed and are human readable.

3.1.1 Importance of metadata

The information that is needed to create the test corpus catalog is best held within the test source code. As the source code is updated, the meta-information can be updated at the same time, which reduces the possibility that the two becoming unsynchronized.

It is also useful if the metadata can be codified within the programming language of the automated test. This process allows the programming language compiler to check that the meta-information is syntactically correct. This check makes entering the meta-information easier and reduces engineer errors, which increases the probability that the meta-information is accurate.

Meta-information that is encoded within the language of the automated test also can be extracted from the test as the automation is compiled. Extracted information then can be compiled into a test catalog directly from the test material. This process ensures that the test corpus and the catalog are synchronized. Whenever the test material is changed, it can be automatically rebuilt, and the test catalog regenerated and published.

At test run time, the annotations are examined to ensure that the test can be run against the level of the software that is under test. If the tested software violates the meta-information, the test framework ignores the test and warns the tester rather than attempting to run the test in an incorrect context. This feature alone can reduce the number of false negative test results.

For example, in Java we can use the annotation pattern to store the test meta-information. An annotation is created for each piece of meta-information that we want to store. This annotation is included in each test class. When the tests are built, the annotations can be extracted and used to validate the information in the test and then used to construct the test catalog.

To illustrate this concept, consider the requirement to detail the minimum and maximum versions of the software under test that a test can validate. Two annotations can be constructed:

- @MinimumRelease(version = Release.v110)
- @MaximumRelease(version = Release.v200)

These annotations define the minimum and maximum releases that are supported by the test class. The test then can be run against a build of the software between versions 1.1.0 and 2.0.0. Each annotation contains a parameter version that specifies a release of the software. The value of this parameter is an element from an enumeration. This enumeration enables the test engineer to choose the release from a list of options (constrained by the enumeration), rather than entering a string value.

A constraint exists between the two annotations. The value that is specified in `@MiniumumRelease` must be less than the value held in `@MaximumRelease`. Again, at build time, the compiled code can be interrogated to enforce this constraint, which ensures that the meta-information that is built from the test code is accurate.

When the tests are built, the meta-information from the tests can be extracted and used to construct the test catalog. This feature enforces that the meta-information is syntactically correct, logically consistent, and most importantly synchronized with the test code. This ensures that your CI pipeline always can run the most dated set of tests against new code release, which helps to ensure that your releases are of high quality.

3.1.2 Creating multiple indexes in the catalog

After the test catalog is constructed, it can be used in many ways; for example, searching for a test by name, selecting a set of tests that test a specific area of the product, or searching for tests that include a specific tag. For this search to be as efficient as possible, it makes sense to create multiple indexes against the catalog to allow it to be indexed in this way. For example, the following indexes are used in our test catalog:

- Test class name
- CICS release
- Testing area
- Core regression tag
- Owning group
- Test phase (that ism performance or load tests) that we do not always want to run in a pipeline

Multiple indexes allow the catalog to be browsed in multiple ways to support the type of testing that is required. For example, during early stages of the pipeline, we might want to run only the tests that run the functions of the software that was updated; therefore, we use the Testing area index to view all tests that are grouped by functional area.

Alternatively, during integration phases of the pipeline, we might want to run all the tests under the core regression index because this index contains all of the integration tests that test a selection of core function across the product.

Each index also can be nested within another index. This nesting allows for a fine-grained selection of tests to occur. An example is to select all the tests that test a specific area that support a specific version of CICS and that are owned by a specific team.

3.1.3 Binding tests to specific version of software

Including annotations within the test class to specify the versions of the software that the test class can support is important. Specifying versions helps you to quantify the number of new tests that were developed for a new release of the software. You might want to run tests that target new functions during the early stages of your integration pipeline, especially if you are experimenting with Test Driven Development type practices.

A single test class can support multiple versions of the software by specifying a range of versions within the minimum and maximum annotations. In this case, it is important that the test can obtain the specific version of the software that it is running against. This process allows the test class to make adjustments at runtime based on the software version, including differences in label names or extra data that is included in API responses.

Allowing a single test class to support multiple versions of the software reduces the requirement for multiple test classes potentially reducing duplicated code. However, if the changes that are needed to support multiple versions of the application are large, complex, or insert extra test methods, a separate test class is preferred.

Finally, binding your tests to the specific versions of the software that you are testing allows for tests within the test corpus to be scheduled to run against a specific release of the software. Tests that cannot test the required level of the software can be automatically deselected by the test framework or set to an IGNORED state. Without this safeguard, tests run and potentially produce false positive results, which prevents code from being promoted within an integration pipeline or incurs engineers wasting time investigating errors.

3.1.4 Binding tests to the function that they test

When a functional area of an application is updated or changed, the first set of tests you want to run are the tests that focus on that specific area. During unit testing, this issue is fairly trivial because the tests are likely to exist alongside the component that they exercise.

However, as we look at integration tests, it is not that simple. A change to a front-end component that is used to add a user to the application might include ramifications to the data processing logic that adds the user. Unit testing of the front-end component and testing the API contract between the two components can go some way to alleviating this risk.

Running all of the tests that test the integration between the two components allow you to ensure the quality of the whole solution, but these tests are unlikely to be in the same project as the front-end or back-end. Your CI pipeline must know where they are and how to run them.

Not all tests directly test all components of your applications; instead, they might focus only on a single or pair of components; however, they might indirectly exercise or drive a set of other capabilities that are more general or shared among other components. Identifying the areas that a test exercises or tests is useful to help you identify the set of tests you want to run at any specific time.

Code coverage is often used to do this and is useful when unit or function testing is conducted. However, when integration testing, it can become difficult to obtain coverage for all elements of the application, especially when they run on different platforms or are written in different languages. Also, given the more complex nature of the tests, it can be difficult to imply from code coverage data if a specific function was tested; that is, the results from the function were examined directly by the test to ensure correctness, or exercised only as part of the run of the test. We saw tests that included the following tags:

```
@AreasTested(area = Accounts_DEBIT)

@AreasTested(area = Accounts_TXFER)
```

These tags identify the areas of the product that are tested and the following tag also was used:

```
@AreasExercised(area = Accounts_LOGGING, Accounts_SCREENS)
```

These tags identify the areas of the product that is exercised by this test, although not directly exercised. This information can be used when scheduling tests to run all the tests that direct test a changed area and if they pass, run the tests that also exercise an area that was changed.

Having a good catalog of your test corpus that is up to date and both machine and human readable allow that information to be used not just during test scheduling, but through the entire test lifecycle. This in turn increases the value of the automated tests that are used.

3.2 Scheduling tests

A good set of reliable, automated tests are a valuable asset to a team. That value increases the more the tests can be used to validate different builds. In the early stages of building a CI pipeline for a hybrid application, you might be thinking only of the tests you want to run to validate a full build of the software; however, it is likely as your deployment matures that you want to extend those tests to validate different types of builds at different stages in your pipeline.

In this section, we discuss those different builds and how you can adjust your test automation during the early stages of adoption to capitalize on them as your test automation matures.

3.2.1 Multiple versions: Release, pre-release, and private builds

It is highly unlikely that you have a single build of your software. You likely to have a version of the software that is in production and a version that is under development. Each of these versions has a unique version number to identify them. The version of the software that a test can run against can be identified through annotations within the test source code.

However, it is common to have different build levels within the in-development release, such as best-so-far, integrated, test, and increment. These levels can be combined with the version number to give a complete picture of the status of this software build.

Although a test is likely not interested in the build level in the same way that it is interested in the version, it is a piece of information that is useful for scheduling and reporting the tests.

3.2.2 CI pipeline products and scheduling

As the collection of automated tests you use to validate the quality of your software increases, so does the requirement to allow your CI pipeline to intelligently and efficiently select and run a set of tests against a build of the software. Although CI products differ in style and implementation, they all share some requirements.

The CI pipeline should not be running tests but instead calling upon the services of the test infrastructure to run and report tests on its behalf.

The CI pipeline should select a set of tests based on meta-information within the test catalog; for example, all of the tests that test a specific area of the product or were tagged with a specific label. This allows for the pipeline to run new tests as they are added into the test catalog without needing to have a static list manually updated.

When tests are run, the CI pipeline feeds the parameters that define the specific context of this test that is run into the test automation system, including the following examples:

- Software version
- Software build level
- Environment parameters that are specific to this run

These parameters should be merged with a set of generic parameters to produce the individual context for the requested tests to run within.

The CI pipeline also requests batches of tests to be run rather than individually so that the test execution engine runs the tests in parallel, which reduces test run time. The pipeline can then query this batch of tests to obtain test result data. Test result data should contain (but is not limited to) the following information:

- Test name
- Test result (including non-binary results, such as ignored)
- Unique identifier to obtain test artifacts

This set of information should be enough for the pipeline to understand what the next decision of the pipeline should be. Some tests might need to be rerun, tests that were ignored should be removed from the list. If tests failed, a defect should be raised in the defect tracking system. If all of the tests passed, the delivered software can be moved to the next stage in the pipeline.

3.2.3 Handling pipeline build releases

All of the information thus far in this chapter helps you to build a flexible CI pipeline that allows for changes that are delivered by a developer to be automatically built, tested, and integrated into your code base. Although the implementation of a CI pipeline for a product is out of the scope of this publication, it is important to note that as the CI pipeline is automatically building temporary builds of the product to be deployed and tested, these builds can be uniquely identified by using build versions. This uniqueness is then passed to the running tests and reporting systems.

3.2.4 Developer run tests

Although a mature CI pipeline is likely to be the primary user of test automation, ad hoc test run requests are always going to be necessary; therefore, it is important that every member of a delivery team can select and run any available test. These requests are not implemented through your CI product but instead require a separate UI or API to allow the user to request a test to be run.

3.3 Test artifacts

Now that we have a good set of cataloged tests that can be run against any level of the application by a CI pipeline or user, we must understand how to handle the outputs of that test. Each test that runs is going to produce two distinct sets of information.

The first set is the result data (pass/fail/ignore) that gives the overall result of this test run. This set of data also includes meta information about the run of the test, such as start/finish time and requester ID. It also includes information that is taken from the same meta information that is contained in the test source code that is used to construct the test catalog. This information can be in a JSON or XML format and is used to create a test dashboard that gives a holistic view of the quality of the product.

The second set of information consists of artifacts that are produced during test execution, such as logs and received output that can be used to audit that a test ran correctly or to aid in debugging when a test reports a failure.

Because the set of test artifacts depends on the type of application that is being tested and the technology that is used to conduct that testing, it is unlikely that a single approach to understanding the test artifacts is helpful. For example, a test driving a set of web UIs through Selenium produces a set of screen captures that are taken from the rendered web pages. A test that is running batch jobs on z/OS or driving a set of 3270 interfaces produces JCL spool output and renders terminal emulator output.Because of the nature of integration testing, it is likely that a single test might produce all of these artifacts.

A test engineer that wants to debug a test that produces this varying level of artifacts needs a consistent way of obtaining and interacting with all of this output. By using tools, it is likely that this information might be scattered across different test platforms and must gather it together in a single location. Similarly, the use of a test run log to understand the run of a test is also difficult if the run log is split across different test tools. Galasa provides a mechanism to combine all of the artifacts that are generated by a test to a single location and provide the UI to allow a tester to interact with each of the artifacts in a simple manner.

3.4 Reporting tests

In addition to test artifacts, a test run also produces a test run result that indicates whether this test passed or failed. This information is vital to understanding the quality of the software being tested. Without the correct ways of managing and analyzing this data, a risk exists that an enterprise is not obtaining the correct level of value from their automated test corpus.

3.4.1 Multiple actors contributing to a single test result

For the test result information to be used, it must be collected into a single datastore. If each test technology that is used wrote the result of a test to a different result store, these disparate stores must be combined to get an overall picture of the testing. For this combination of results to work, it must be needed for all the different test technologies to produce a test report that contained the same data. Without similar data reports from each test technology, the value of the overall report is reduced.

An integrated test exacerbates these issues because many test technologies work together to validate a test and no single test technology can be the sole decision maker of if a test passed or failed. This concept can be explained by using an example.

Consider an integrated test that runs a set of web UI interactions to drive a function. After it is complete, a z/OS batch job is used to verify that audit logs on the mainframe were correctly entered. Finally, a 3270 application is driven to ensure that older parts of the application are not regressed by the new function.

Although the web, batch, and 3270 parts of the test all ran and effectively passed their parts of the test, the test as a whole can fail if messages that are written by the application show that an ancillary transaction abended during the test run. Individually, each element of the test passed and it is only the extra checking by the framework of the state of the application at the end of the test that decided that an issue exists and that this test is marked as a failure.

The open source nature of test tools is important here because it allows an integration framework to easily run and combine the results from multiple test components into a single result, which reduces duplication and increases confidence in the test result.

3.4.2 Using the data in the test catalog to derive test metrics

It is not enough to simply say that this test class passed or failed. Instead, that data must also include much more data, primarily the data that is in the test catalog. If a test result record includes all of this data, it allows an analyst to split the result data around any of the data items that is included.

Grouping tests results depends on the area of function that they test, version of the software that they ran against, and any other piece of meta-information that is included within the test catalog. Run-time information allows the test results to be presented in as many ways as needed.

The same arguments are used when discussing an automated test catalog and its use to select tests at runtime can also be used here, if the catalog is built from the test source, used at runtime to select tests, and used to organize test results. Then, the information always is accurate and current.

3.4.3 When is a pass result not a pass result

In our experience, a test result is not always a binary pass or fail result. In a perfect world, a binary result shows that the software was free from known regressions or contained a problem that a defect can be raised against. However, as the application becomes more complex and is run across multiple platforms, a fail result might not always imply a genuine regression.

A test might fail for many reasons, none of which are necessarily associated with a regression in the software being tested; for example, network connectivity issues and ID expiring. In these cases, although the test failed, its failure should be differentiated from purely logic errors that are found during the test.

It also is not restricted to set up type problems. A test might complete correctly, but during the deprovisioning of a z/OS application, the log might show a system error that was not evident during the test. The tests appeared to pass, but the error in the log must be investigated.

These types of failures often require a different skill set to understand and remedy. A software developer might not be the best choice to debug a network connectivity or a user ID revocation issue. Instead, their skills are better placed to debug an issue within the target software. Therefore, it is important to separate these failures from genuine failures so that the CI pipeline and test analysts can drive the correct action from each failure.

These errors can be delineated as Environmental Errors; they do not count toward the pass/fail count, but exist as a separate error.

3.5 Summary

In this chapter, we discussed that running automated tests in the enterprise is more than using a tool or writing the automation. It requires a level of capability to span an entire hybrid cloud application.

4

Introducing Galasa

In Chapter 3, “Enterprise test automation” on page 23, we described why automated testing is so important and highlighted the challenges that are common with setting up automated testing in general, and in the z/OS space specifically.

These challenges were faced within our own organization and this chapter discusses how we overcame them by creating our own automated test framework. This framework was successful within IBM. It not only resolved the previously mentioned challenges, but also incorporated all of the key requirements that are discussed in Chapter 3, “Enterprise test automation” on page 23 that make enterprise test automation so vital. As a result of this internal success story, we decided to release an Open Source version of the framework, called Galasa.

This chapter includes the following topics:

4.1 Inspiration behind the name

Galasa is a type of Snout Moth. A moth was attached to the famous “bug” report by Grace Hopper where she recorded the “first actual case of a bug being found” after a moth had flown into a relay (see Figure 4-1)[1].

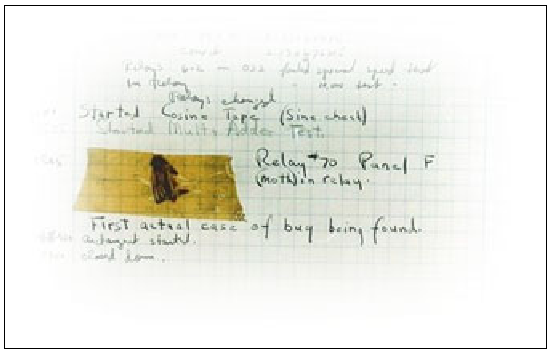

Started Cosine Tape (Sine check)
Started Multy Adder Test.
Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.

Figure 4-1 Galasa Snout Moth

Although we do not know whether it was a Galasa moth in the relay, we quite like the name.

4.2 Galasa overview

Galasa is an Open Source, deep integration test framework for hybrid cloud applications that allows teams to automate tests to run as part of a DevOps pipeline.

What makes Galasa a deep integration test framework is its ability to support tests that cross system boundaries and reach into remote layers that are inaccessible to other test tools. Many test tools allow you to test an endpoint, but they do not include Galasa’s ability for deep integration testing that enables you to explore end-to-end effects.

Galasa’s reliable, scalable, extensible framework for automating tests supports continuous delivery and frees testers up to focus on designing more sophisticated, high-value tests to root out hard-to-find application defects instead of spending unnecessary time on resource-intensive, repetitive testing.

Galasa uses plug-ins to popular integrated development environments (IDEs), such as Eclipse, to help with writing, running, and debugging tests. A Jenkins plug-in enables tests to be run as part of a Jenkins CI/CD pipeline.

For more information about Galasa and how you can contribute to the Galasa project, see this website.

[1] Source: `https://commons.wikimedia.org/wiki/File:H96566k.jpg`

4.3 Evolution of Galasa

The IBM CICS organization needed a test framework to enable them to meet key needs, including the ability to perform the following tasks:

- Automate tests and automate the checking of test results
- Limit code duplication by extracting repeatable processes outside of the test
- Manage the selection, scheduling, and reporting of tests by using a test catalog
- View diagnostics that enable the cause of a failure to be easily identified
- Isolate tests when testing at scale

There were many challenges to overcome, so we moved ahead and successfully completed our automation journey.

The framework needed to enable non-mainframe experts to perform the following tasks:

- Interact with a CICS region to start automating tests
- Run a Job Control Language (JCL) to run programs and utilities, and APIs
- Simulate 3270 traffic

Specific challenges for the CICS organization included the following factors:

- We use large systems that were built up over many years (multiple releases exist in service and a release under development), all being constantly updated and tested by a global team.
- We use many test technologies, including terminal interactions, batch jobs, web services, and selenium-based browsers.
- Although our tests primarily use CICS, we also must test integration points with other systems, including IMS, IBM DB2® and IBM WebSphere® MQ.

We needed a test framework that enabled us to run reliable, integrated, automated tests at scale. However, it is not good enough to write an automated version of your tests; the management, selection, scheduling, and reporting of those tests is also important. Out of these needs, the Java Automated Test (JAT) framework was created. The framework was inspired by JUnit, but with the focus on functional and deep integration testing rather than unit testing.

We used JAT within IBM CICS successfully for many years, which resulted in the following benefits:

- Our global development, service, and testing teams around the world are constantly developing five releases of CICS seamlessly.
- We complete the equivalent of 25 days of manual testing time in a single day, running about 5000 test classes.
- We can run the framework across 10 test LPARS.

Our experience of adopting and using JAT included the following key takeaways:

- The framework is scalable. The oscillations across the graph show that the framework easily handles non-constant workflow, scaling to handle high peaks at busy times.
- All tests do not have to be converted at once. The gradient's growth over time shows that the framework grows with your tests, which you can migrate gradually so that you never must stop developing.
- The framework is powerful. We can run over 70 thousand test classes in a single month.

Repeatable, scalable, and reliable integration tests that are running as part of our DevOps pipeline helped to reduce the delivery time of a new CICS feature from 18 to 3 months without any loss of confidence in quality. We wanted to share our success with other organizations and so began the process of creating an open-source version of the JAT framework, which perfected the architecture. An initial version of Galasa was released in 2019.

4.4 Galasa architecture

Galasa consists of the following major components:

- Test runner
- Collection of Managers
- Underlying core Galasa framework
- Galasa ecosystem

These components are described next.

4.4.1 Test runner

The Galasa test runner combines all of the parts of Galasa with your test material to run a test. To run a test, the test bundle is submitted to the test runner. The test runner can be hosted anywhere; for example, in a Kubernetes cluster, which can scale according to the processing needs of your test suites.

The test runner can run locally on your notebook or in the Galasa ecosystem. Therefore, the same test can be developed, tested, and debugged locally at speed and then run by the ecosystem when required. Regardless of whether the test runs locally or in the ecosystem, it can still access all of the resources that it needs.

For each test, the test runner starts an instance of the Galasa framework, which, alongside the test bundle, is used to run the test. The framework recognizes annotations that are used within the test code and uses this information to activate relevant Managers and can provision the resources that are needed by the test, such as environments, test data, or Docker containers. The test can then be run with all of the resources that it requires to run in place.

4.4.2 Managers

The test runner can access Galasa Managers. Managers make Galasa adaptable and extensible and provide a way of interacting with a specific tool or operating system-specific functions.

The main purpose of a Manager is to reduce the amount of boilerplate code within a test and provide proven tool interaction code. This feature makes the test code simpler and easier to write and understand and maintain because the focus of a test's development shifts to validating application changes, rather than marshalling environmental resources.

Different Managers can collaborate with each other to perform a joint task, including sharing information and getting other Managers to complete tasks for them.

For example, the Docker Manager enables containers to run on infrastructure Docker engines, which makes it easy to write tests that use container-based services. The test does not need to worry about where the Docker infrastructure is, its credentials, or its capacity because these issues are is all handled by the Manager.

The Docker Manager can be used by other Managers as a base for their own services. For example, the JMeter Manager can run a JMeter service inside a Docker container. By using the Docker Manager in this way, the test or administration team do not need to create dedicated JMeter resources.

The z/OS Batch Manager provides tests and Managers with the ability to submit, monitor, and retrieve z/OS batch jobs. For example, code to start z/OS MF is not required in the test. Instead, this code is written inside the Batch Manager and this Manager is started by using the annotation @ZosBatch within the test code. For more information about an example walkthrough of a test that uses the Batch Manager and a list of currently available and future Managers, see the Galasa website.

The work of the Managers is coordinated by the Galasa framework.

4.4.3 Framework

The Galasa framework sits on top of the Managers. The framework coordinates with the Configuration Property Store (CPS), Dynamic Status Store (DSS), Result Archive Store (RAS), API server, and Managers to allow a test to seamlessly run locally or in automation, against any environment, without needing to change the test code or to understand the context of its running.

Configuration Property Store

The CPS is used to define object properties; for example, the port on which a terminal connects to an application. Defining properties by using the CPS means that a tester does not need to know where the application to which they are connecting is running because this connection is configured in a Manager by the system administrator. Different properties can be set in the CPS to allow a test to run against multiple environments without changing the code that is inside the test.

Dynamic Status Store

The DSS is used by Managers and the Galasa framework to ensure that limits that are set in the CPS configuration are not exceeded. DSS properties change dynamically as tests are run to show the resources that are in use, shared, or locked by a test, so that workloads can be limited to avoid throttling. When running in automation, the DSS is shared by every instance of the framework.

Result Archive Store

The RAS stores all elements of a test, including the run log; stored artifacts, such as requests; and properties that are changed in the DSS. These elements can be used to help diagnose the cause of any failures that are encountered as a result of running the test, or to gather information about the test.

API server

The API server is an API mediation layer that sits between the Galasa framework and Galasa services, such as the properties services in the CPS. The API server acts as a central point from which to control the Galasa ecosystem.

Galasa's framework orchestrates all of these component activities and coordinates with the test runner to run tests. No code must be written to start a test; instead, the code that defines them is written as a set of one or more test classes and the methods within.

The Galasa framework automatically recognizes test definitions and activates the Managers that are required by the test and the test runner to run the test. It is unlikely that you want to change the framework or test runner code during the normal range of testing activities.

4.4.4 Galasa ecosystem

The Galasa *ecosystem* is the term that is given to the collection of components that are described in 4.4, “Galasa architecture” on page 36. The Managers, framework, DSS, RAS, CPS, and API server all exist within the ecosystem. This configuration enables automation and testers to submit tests, review test results, and examine test artifacts from a single endpoint.

The Galasa ecosystem is included as a set of containers that can be deployed to an OpenShift or Kubernetes-based cluster that can then scale to meet your demands. We supply operators with which you can deploy this ecosystem by using a single command.

4.4.5 Benefits of using the Galasa test framework

The use of a test framework includes the following benefits:

- The Galasa framework can be integrated easily with the rest of your pipeline, which enables you to take advantage of the following features that are provided by the framework:
  - A single API that can be used to run any test, regardless of the underlying technology.
  - A test catalog from which the relevant tests can be dynamically selected.
  - Checks that ensure the test environment is available and stable before tests are run.
  - Automatic monitoring of test results by the framework.
- The Galasa framework acts as a single point of control, which means that:
  - Test artifacts are centrally stored and indexed.
  - Test results are correlated, which ensures the quality of the release for all product components.
  - A single control is used for resource and test allocation and management, which is key to avoiding conflicts and corruption of data.
- The Galasa framework enables you to test at scale by supporting the following functions:
  - Late binding of the test material to the system under test. Late binding allows the same test to run against multiple environments without needing to change the test.
  - Horizontal scaling of tests without the need to change the underlying test code.
  - The ability to isolate tests so that multiple tests can run in parallel and are logically isolated by the framework.
  - It offers the following environment-agnostic capabilities:
    - Tests can run against multiple environments as code progresses through release stages.
    - The Galasa test engine allows tests to run locally for easier development and debugging, but also in the Galasa ecosystem for production testing at scale.

4.4.6 Test scheduling and monitoring

We discuss cataloging the test corpus in Chapter 3, “Enterprise test automation” on page 23 and how understanding what your tests do is key to managing your test automation. Galasa can dynamically construct a test catalog from your test classes. You can use the catalog to select test suites to run automatically as regression tests in future releases.

The Galasa framework can schedule large numbers of tests to run in parallel, in line with the resources that are available within a test environment, or in accordance with the availability of test “slots”. If all the slots are used, the test sits in a waiting state until a slot becomes available.

Monitoring large numbers of tests is simple because you can view the tests’ status at a glance by using Galasa’s dashboard. You also can use the run IDs that are associated with failed test runs to look up information in the RAS database to help diagnose what went wrong.

You also can use the dashboard’s filter to focus in on a set of tests; for example, tests that are associated with a specific area of the system under test or a test environment.

4.5 Galasa’s value

Galasa is different from other test tools in the following ways:

- Galasa is ideal for deep integration testing of hybrid applications on Z because it features first class support for z/OS and z/OS middleware.
- It incorporates a highly scalable automation system that can handle large volumes of complex enterprise-level tests.
- It is designed to enable low effort, robust, and fast automated testing for a rapid DevOps pipeline.
- It uses Eclipse plug-ins to run, debug, and view test output.

Galasa is not a unit test framework and it does not write tests automatically. It is also not a tool that designed to replace people.

Instead, it is designed for running and rerunning laborious, time-consuming manual tests. That type of testing is not the best use of a tester’s skills. Galasa runs those types of tests to give testers the time to start designing the complex test cases that are more likely to find defects.

5

Getting started with Galasa

This chapter contains information to get you started with running Galasa tests. Installation and configuration information and an introduction to the Galasa:Simbank platform also is provided. A methodology for how Galasa can be extended with your own tests also is described.

This chapter includes the following topics:

5.1 Role of Galasa:SimBank/Platform

Distributed with Galasa, SimBank is a component that simulates a mainframe application. It sits above another component that is called SimPlatform, which provides options for future growth.

As delivered, SimBank implements a sample banking application against which you can configure and run a set of provided tests in preparation for running your own tests against a mainframe application. You can also practice writing some new tests to run against the SimBank banking application.

By exercising the Galasa framework against SimBank, you can preempt much (but not all) of the work and learning that is necessary to eventually incorporate your own tests with a real mainframe environment. If the provided SimBank tests do not work, it is unlikely that you can run your own tests on a mainframe application.

SimBank helps you to learn Galasa's basic principles of operation before you need to learn how to connect Galasa to your own mainframe application-under-test.

5.2 Installation

Galasa installations can vary in complexity, depending on the context in which it is used. Invariably though, all first-time installations begin with the Eclipse integrated development environment (IDE) and the download and integration of the Galasa plug-in from a known update site. The Galasa plug-in is accompanied by SimBank (a demonstration application) that sits on top of a small middleware layer that is called SimPlatform (you might see its name in some console messages, but you otherwise do not need to interact with SimPlatform).

This section describes the most common initial installation scenario; that is, using Eclipse to install the Galasa plug-in with SimPlatform/SimBank on your local machine and preparing it to run an initial set of provided tests against a simulated mainframe application.

5.2.1 Prerequisites

Depending on how you use Galasa, several software prerequisites must be met, some or all of which might be installed already.

5.2.2 Java

Galasa tests and Managers are written in Java. You must install a Java version 8 Java SE Development Kit (but not a later version) to use it.

5.2.3 Eclipse

If you do not have an Eclipse installation, you can download a version of Eclipse that is suitable for your machine. Choose a package that supports your required level of Java development: Eclipse IDE for Java Developers or Eclipse IDE for Java EE Developers. If you are unsure, the Eclipse IDE for Java Developers should be fine, and you can always add any missing plug-ins if and when you discover you need them.

If you have a version of Eclipse installed, it should be at the version that is code-named Oxygen (released in June 2017) or later.

5.2.4 3270 terminal emulator

Galasa is packaged with SimBank, which is a simulated version of an application that helps you get acquainted with Galasa before connecting to a real mainframe to run your own tests. You need a 3270 terminal emulator to enable you to connect to and explore SimBank before running Galasa's provided suite of automated tests. Many such emulators are available, but IBM's Personal Communications is frequently used because it is IBM's Host on-Demand software, which includes support for Windows, Linux, and MacOS.

5.2.5 Installing the Galasa Eclipse plug-in

Complete the following steps to install the Galasa Eclipse plug-in:

1. Start Eclipse. If present, close any initial welcome window.
2. Select **Help** → **Install New Software** from the main menu.
3. Paste `https://p2.galasa.dev/` into the Work with field and press Enter.
4. Check the Galasa box in the main window, which ensures that Galasa and all of its child elements are ticked.
5. Follow the prompts to download and install the Galasa plug-in. You are asked to accept the terms of the license agreement and restart Eclipse to complete the installation.
6. After Eclipse restarts, you can verify that the plug-in is now available by observing the presence of a new Galasa option on the main menu between Run and Window. If you choose **Run** → **Run Configurations** from the main menu, you also see two new entries: Galasa and Galasa SimBank as available in the left side pane of the pop-up window.

5.3 Configuring Eclipse for Galasa

Choose **Galasa** → **Setup Galasa Workspace** from the main Eclipse menu. This Setup Galasa Workspace command creates some necessary configuration files. Your Eclipse console confirms its progress with some messages:

```
Setting up the Galasa workspace
Creating the ~/.galasa files
Created the ~/.galasa directory
Created an empty Bootstrap Properties file ~/.galasa/bootstrap.properties
Created an empty Overrides Properties file ~/.galasa/overrides.properties
Created an empty Credentials Properties file ~/.galasa/credentials.properties
Created an empty CPS Properties file ~/.galasa/cps.properties
Created an empty DSS Properties file ~/.galasa/dss.properties
The ~/.m2 directory exists
Created the ~/.m2/.settings.xml example file
Setup complete
```

Locate your user home directory and confirm it contains a `.galasa` folder. On Windows, the user home directory resembles: `C:\Users\<username>`, on MacOS or Linux, entering `cd ~` in a terminal takes you to your user home directory, whatever it was configured.

Edit a file called `overrides.properties` in your `.galasa` folder so that it contains the following lines:

```
simbank.dse.instance.name=SIMBANK
simbank.instance.SIMBANK.zos.image=SIMBANK
zos.image.SIMBANK.ipv4.hostname=127.0.0.1
zos.image.SIMBANK.telnet.port=2023
zos.image.SIMBANK.webnet.port=2080
zos.image.SIMBANK.telnet.tls=false
zos.image.SIMBANK.credentials=SIMBANK
zosmf.server.SIMBANK.images=SIMBANK
zosmf.server.SIMBANK.hostname=127.0.0.1
zosmf.server.SIMBANK.port=2040
zosmf.server.SIMBANK.https=false
```

Edit a file called `credentials.properties` in your `.galasa` folder so that it contains the following lines:

```
secure.credentials.SIMBANK.username=IBMUSER
secure.credentials.SIMBANK.password=SYS1
Your local Eclipse Galasa installation is now ready for some work.
```

5.4 SimBank example

In this section, we describe how to start and run the SimBank application.

5.4.1 Starting the SimBank application

If you started SimBank, select and run its run configuration. If not, complete the following steps:

1. Start Eclipse.
2. From the main menu, choose **Run** → **Run Configurations**.
3. In the Create, manage and run configurations window, right-click **Galasa SimBank** in the left pane and choose **New Configuration**.
4. Enter your preferred name for the run configuration in the *Name:* field (a relevant name such as *SimBank* is acceptable). Click **Apply** and then **Run**.

   After it is created, your run configuration is available for future runs. In a few seconds, the Eclipse Console window responds with a series of initialization messages, which in Windows looks like the following example:

   ```
   2019-10-21 14:24:37 INFO dev.galasa.simplatform.main.Simplatform main ...
   Simplatform started
   ```

   If you are a Mac or Linux user, the messages are almost identical.

The SimBank process is started, and is listening on port 2023 for Telnet connections, on port 2080 for web services connections and on port 2027 for Derby SQL connections. Neither web services nor Derby connections are explored further in this section.

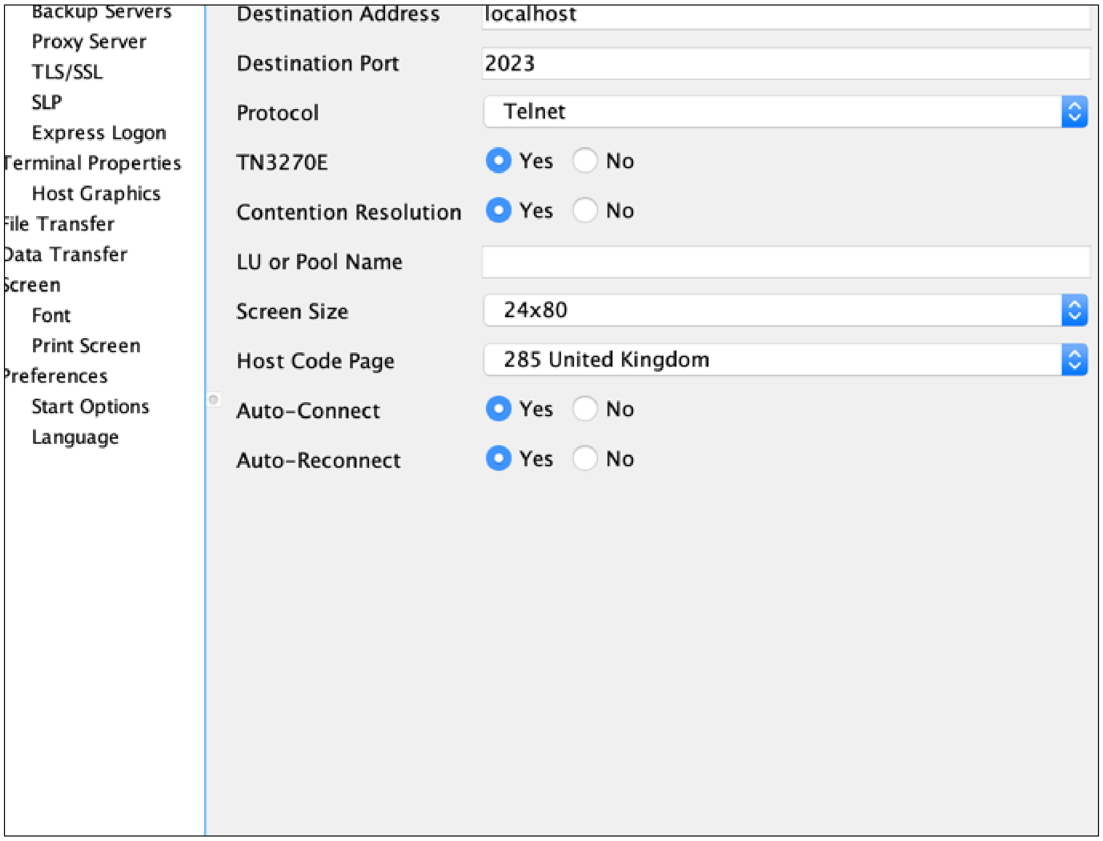
5.4.2 Exploring the SimBank application

When you start SimBank, its banking application listens on port 2023 for incoming client Telnet connections,. This process offers an opportunity to first connect to it manually to review and understand the (simulated) transactions it supports before subjecting it to Galasa's provided tests.

Logging in to the simulated application

Complete the following steps:

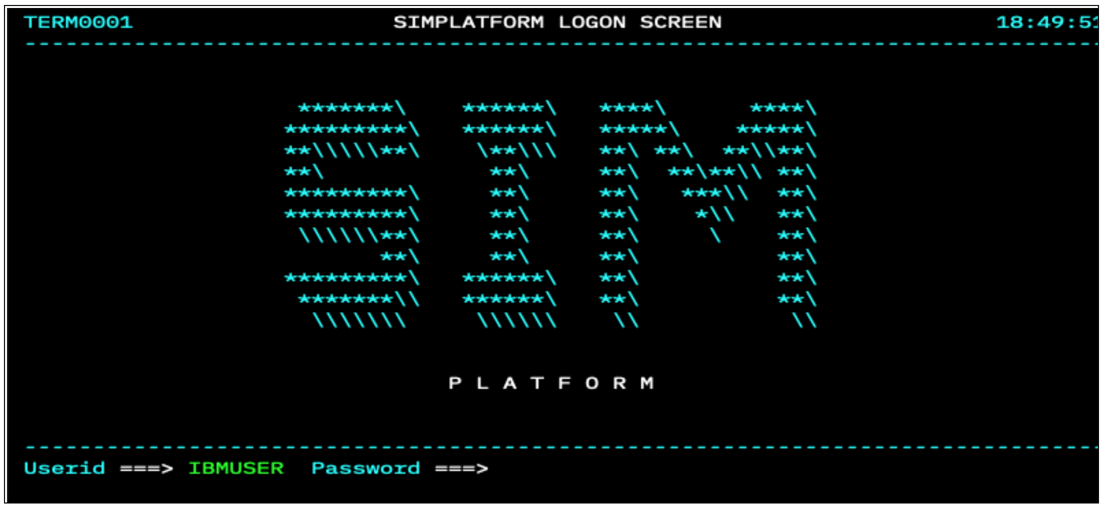
1. With Eclipse and the Galasa SimBank component still running, configure your 3270 terminal emulator to access port 2023 of localhost (or IP address `127.0.0.1` if the localhost alias is not set up) by way of the Telnet protocol. No SSL configuration is required (see Figure 5-1).



Destination Address	localhost
Destination Port	2023
Protocol	Telnet
TN3270E	[✓ Yes] No
Contention Resolution	[✓ Yes] No
LU or Pool Name	
Screen Size	24x80
Host Code Page	285 United Kingdom
Auto-Connect	[✓ Yes] No
Auto-Reconnect	[✓ Yes] No

Figure 5-1 Connection details

2. Connect to the listening Telnet service with your 3270 emulator and review the logon window (see Figure 5-2).

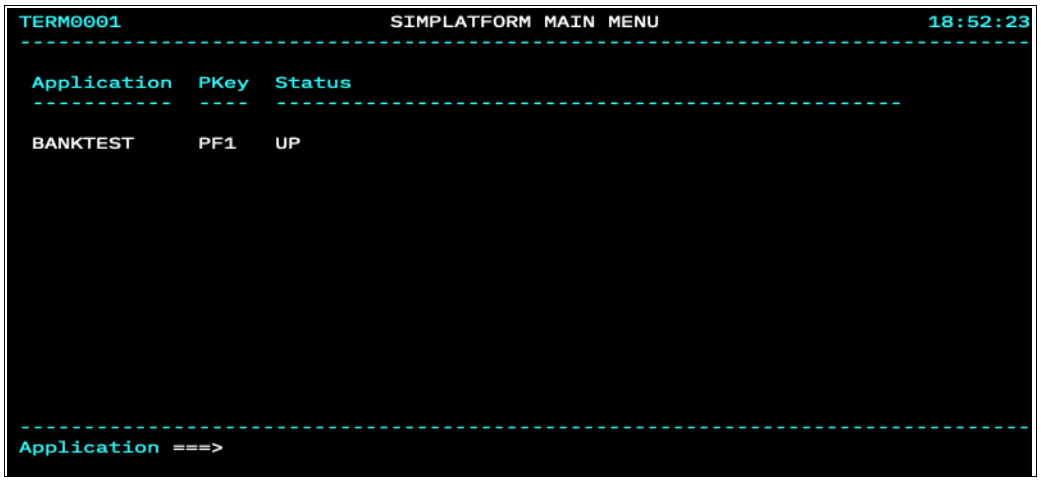


```
TERM0001                  SIMPLATFORM LOGON SCREEN                  18:49:5
---------------------------------------------------------------------------

                     *******\    ******\   ****\      ****\
                    *********\   ******\   *****\    *****\
                    **\\\\\**\    \**\\\   **\ **\  **\\**\
                    **\            **\     **\  **\**\\ **\
                    *********\     **\     **\   ***\\  **\
                    *********\     **\     **\    *\\   **\
                     \\\\\\**\     **\     **\     \    **\
                           **\     **\     **\          **\
                    *********\   ******\   **\          **\
                     *******\\   ******\   **\          **\
                      \\\\\\\    \\\\\\    \\           \\

                              P L A T F O R M

---------------------------------------------------------------------------
Userid ===> IBMUSER  Password ===>
```

Figure 5-2 SimPlatform main window

3. Ensure that the cursor is in the Userid field (if it is not, use the TAB key to position it). Enter the user ID `IBMUSER`. Press TAB to move the cursor into the Password field. Enter the password `SYS1` and press ENTER to log on and transfer to the SimBank main menu.

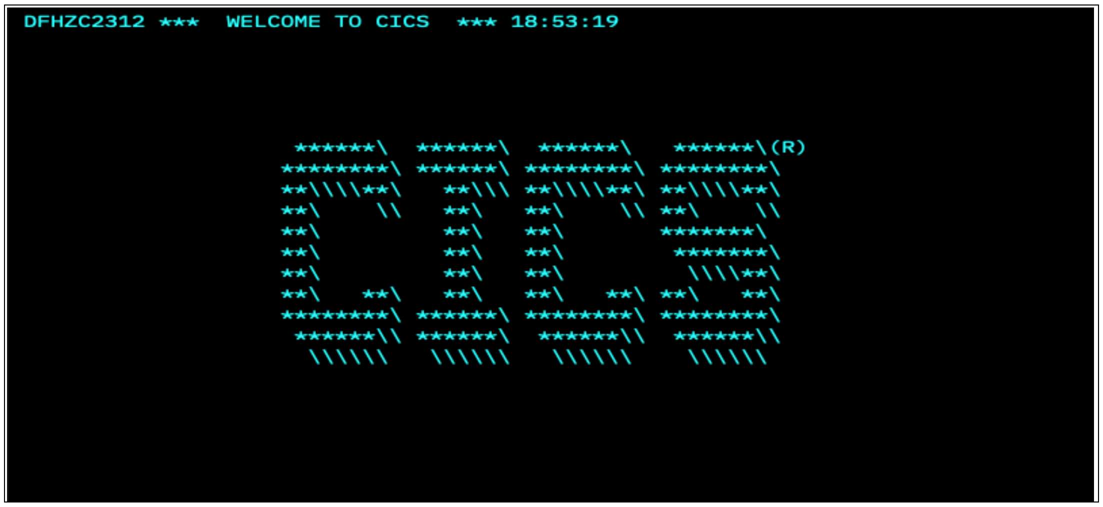
4. Press TAB to move the cursor into the Password field. Enter the password `SYS1` and press your terminal emulator's ENTER key to log on and transfer to the SimBank main menu (see Figure 5-3).



```
TERM0001                  SIMPLATFORM MAIN MENU                   18:52:23
--------------------------------------------------------------------------

 Application  PKey  Status
 -----------  ----  ----------------------------------------------

 BANKTEST     PF1   UP










--------------------------------------------------------------------------
Application ===>
```

Figure 5-3 BANKTEST window

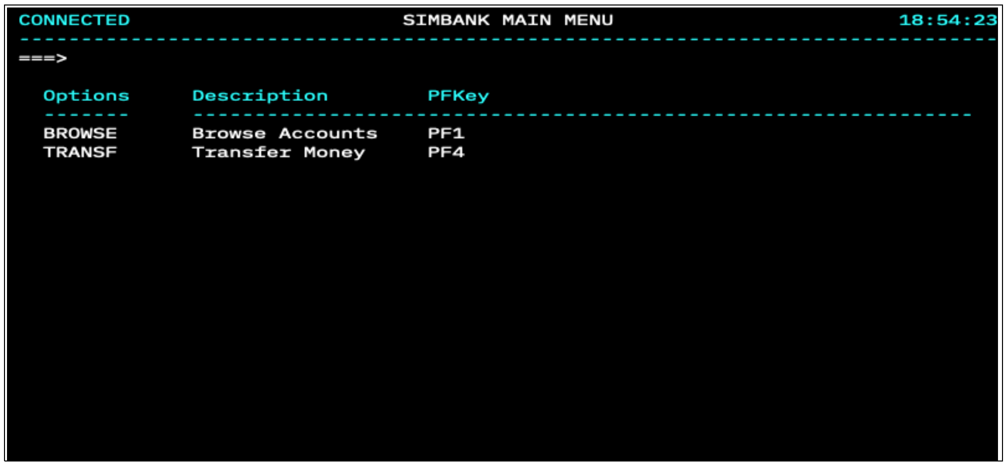
5. Press PF1 (Figure 5-4).



```
DFHZC2312 ***  WELCOME TO CICS  *** 18:53:19

                  ******\   ******\   ******\    ******\(R)
                 ********\ ******\ ********\ ********\
                 **\\\\**\   **\\\ **\\\\**\ **\\\\**\
                 **\     \\   **\   **\     \\ **\     \\
                 **\          **\   **\        *******\
                 **\          **\   **\         *******\
                 **\          **\   **\          \\\\**\
                 **\    **\   **\   **\    **\ **\    **\
                 ********\ ******\ ********\ ********\
                  ******\\ ******\  ******\\  ******\\
                   \\\\\\   \\\\\\   \\\\\\    \\\\\\
```

Figure 5-4 Main CICS window

6. Press your terminal emulator's CLEAR SCREEN key.
7. Enter the transaction name BANK and press your terminal emulator's ENTER key again get to the SimBank main menu (see Figure 5-5).



```
CONNECTED                    SIMBANK MAIN MENU                     18:54:23
-----------------------------------------------------------------------------
===>

  Options       Description        PFKey
  -------      ---------------------------------------------------------
  BROWSE        Browse Accounts    PF1
  TRANSF        Transfer Money     PF4
```

Figure 5-5 SimBank options

As you progress through this process, Eclipse logs selected events to its console:

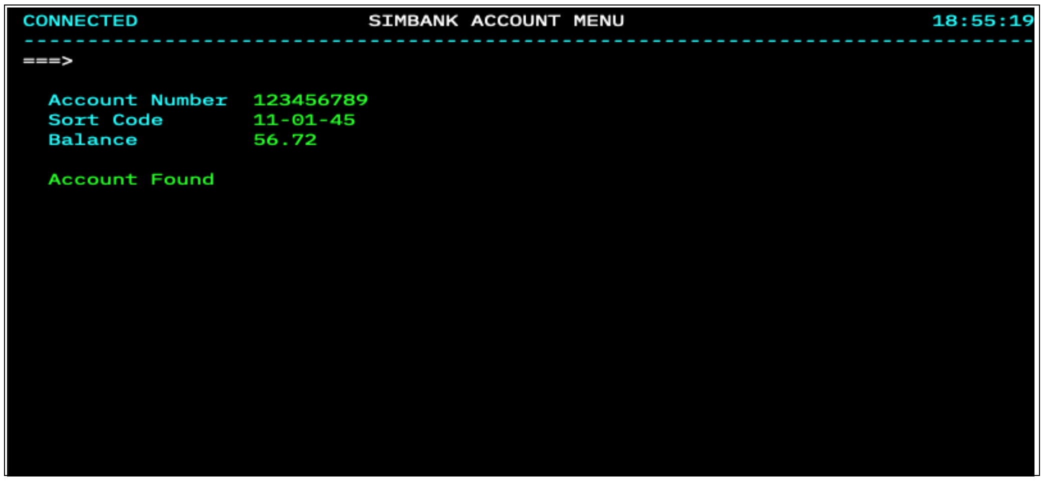
```
2019-08-16 09:26:39 INFO dev.galasa.simplatform.t3270.screens.AbstractScreen
buildScreen Building Screen: SessionManagerLogon
2019-08-16 10:26:08 INFO dev.galasa.simplatform.saf.SecurityAuthorizationFacility
authenticate User: IBMUSER authenticated
2019-08-16 10:26:08 INFO dev.galasa.simplatform.t3270.screens.AbstractScreen
buildScreen Building Screen: SessionManagerMenu
2019-08-16 10:30:10 INFO dev.galasa.simplatform.t3270.screens.AbstractScreen
buildScreen Building Screen: CICSGoodMorning
2019-08-16 10:36:19 INFO dev.galasa.simplatform.t3270.screens.AbstractScreen
buildScreen Building Screen: CICSClearScreen
2019-08-16 10:38:54 INFO dev.galasa.simplatform.t3270.screens.AbstractScreen
buildScreen Building Screen: BankMainMenu
```

This is an example of log output that can be useful when running tests.

Browsing account information

Complete the following steps:

1. From the SimBank main menu, press PF1, which takes you to the account menu window.
2. Press TAB until the cursor is in the Account Number field. Enter `123456789` and press ENTER. The account details are populated, which show that the account number `123456789` is 56.72 in credit (see Figure 5-6).



```
CONNECTED                     SIMBANK ACCOUNT MENU                         18:55:19
-------------------------------------------------------------------------------
===>

  Account Number   123456789
  Sort Code        11-01-45
  Balance          56.72

  Account Found
```

Figure 5-6 Account menu

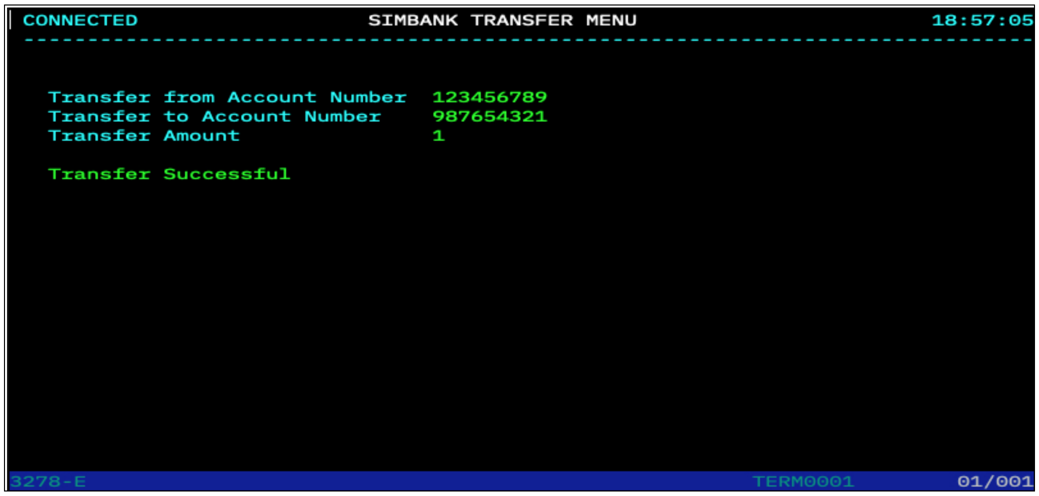
3. Press PF3 to return to the account menu window.

Transferring funds between accounts

Complete the following steps:

1. From the SimBank main menu, press PF4, which takes you to the SimBank transfer menu.
2. Press TAB until the cursor is in the `Transfer from Account Number` field and enter `123456789`.
3. Press TAB until the cursor is in the `Transfer to Account Number` field and enter `987654321`.

4. Press TAB until the cursor is in the `Transfer Amount` field and enter `1`, see (Figure 5-7).



```
 CONNECTED                        SIMBANK TRANSFER MENU                          18:57:05
 ------------------------------------------------------------------------------------------


   Transfer from Account Number   123456789
   Transfer to Account Number     987654321
   Transfer Amount                1

   Transfer Successful






3278-E                                                               TERM0001      01/001
```

Figure 5-7 Transfer menu

5. Press ENTER. A `Transfer Successful` message appears. A log message also appears in the Eclipse Console window:

   ```
   2019-08-16 13:50:53 INFO dev.galasa.simplatform.application.Bank transferMoney
   Transfering  1.0 from account: 123456789 to account: 987654321
   ```

6. Press PF3 and once again browse the `123456789` account to verify that its total credit decreased by the transferred 1.00, and that the `987654321` account increased by the same amount.

   Note: SimBank also offers a web services interface on port 2080. Although it is not described here, it is used by two of the provided tests, as described in the next section.

Having explored SimBank manually, it is a good time to run some or all of a small collection of automated tests that are provided with SimBank. Step-by-step instructions about how to run the tests are described in the Galasa documentation. For more information about the tests, see 5.5, “Different layers of testing” on page 50.

Note: Before the tests are run, remember to initialize the Galasa framework by selecting **Galasa** → **Initialize Galasa Framework**. Show the Galasa Results view by selecting **Window** → **Show View** → **Other** → **Galasa Results**.

5.5 Different layers of testing

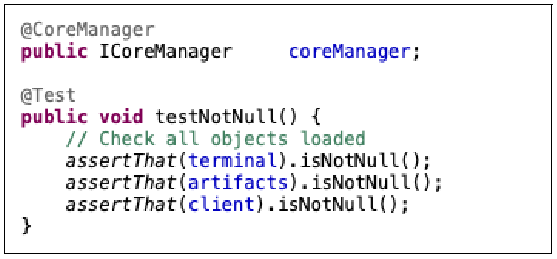
The example tests of SimBank provide an example of the logical progression of tests as a tester becomes more familiar with Galasa. The tests show a standard ‘Hello World’ example to provide familiarity with the nuances of how the tests are formatted, and move on to a more logically worked out example to test more complicated actions. Ultimately, a test is re-created while displaying the capabilities of the framework that shows how the difficulty of provisioning is removed from the tester.

In this section, we examine three tests for SimBank.

5.5.1 SimBankIVT

The first test class SimBankIVT is composed of two tests. The first is a simple “not null” test to ensure that the framework provisioned all the required objects correctly. The second provides example of simple actions within Galasa:SimPlatform.

The not null test is important in showing how the framework populates objects and how the provisioning annotations are laid out above the test. This follows the simple structure of an annotation for the object, followed by a public interface of the object to be populated. It is important that the interface object is public because it must be visible to the framework. Checking that each of these provisioned objects is not null ensures that all objects are created correctly before continuing with other tests (see Figure 5-8).



```
@CoreManager
public ICoreManager     coreManager;

@Test
public void testNotNull() {
    // Check all objects loaded
    assertThat(terminal).isNotNull();
    assertThat(artifacts).isNotNull();
    assertThat(client).isNotNull();
}
```

Figure 5-8 testNotNull()

The core manager, as shown in Figure 5-8, is one of the most basic managers. It allows the tester to interact directly with the framework to gain information from the run (for example, the run name), or to affect the framework during the run (for example, setting specific strings as confidential so they do not appear in run logs).

The second test uses the terminal to complete a simple navigation to the SimBank application. This test is used to show off a few core concepts of the Galasa framework.

First, the use of the Configuration Property Store (CPS) to define properties of objects allows the terminal to connect to the SimBank application. Through changing the CPS, you can define the telnet port to which the terminal connects. This change allows for the tester to have no knowledge about where the application they are connecting to is running because this connection is configured by a system administrator.

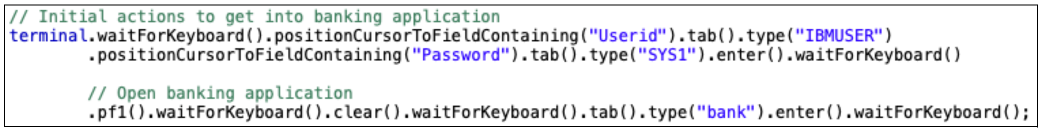
This test also demonstrates basic testing behavior of the use of the objects and provides assertions to pass or fail the test. In this case, the tester logs in to Galasa:SimPlatform and navigates to the SimBank application. From there, the test asserts that the application is on the correct terminal window.

5.5.2 BasicAccountCreditTest

The second test, BasicAccountCreditTest, is used to show a more complicated integration test between the terminal and web services of SimBank. All of the actions in this test are literal, which is similar to that seen in the second test of the IVT; therefore, it requires the tester to navigate windows by using the terminal during the test.

The test navigates to the Simbank application, and retrieves the balance of a specific test account through more navigation from the terminal. A web request is then sent to the SimBank application to update the balance of the test account. The balance is then retrieved again and tested against the original amount to test that the amount sent was sent correctly.

This test demonstrates literal testing, with the environment, account, and application in this case being hardcoded objects that the tester must set up in advance. Navigating the windows within the application is also hardcoded by using the terminal object. From the first few lines of the test, it can be seen that this can produce complicated and unclear code, as shown in Figure 5-9.



```
// Initial actions to get into banking application
terminal.waitForKeyboard().positionCursorToFieldContaining("Userid").tab().type("IBMUSER")
        .positionCursorToFieldContaining("Password").tab().type("SYS1").enter().waitForKeyboard()

        // Open banking application
        .pf1().waitForKeyboard().clear().waitForKeyboard().tab().type("bank").enter().waitForKeyboard();
```

Figure 5-9 Use of the terminal object

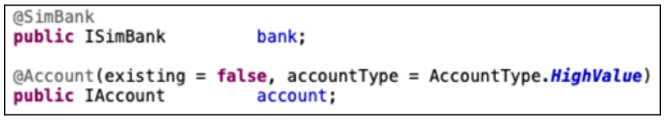
This code must be copied and pasted across all test classes that use the application. It also requires a large change to all the tests if the architecture of the application were to ever change.

This test provides an example of how testing can be done without fully using the scalability of the managers in Galasa. It also is an example of how scripted testing in the way that interactions with the terminal can be performed.

5.5.3 ProvisionedAccountCreditTests

The final example test (ProvisionedAccountCreditTests) provides the same test as the BasicAccountCreditTest, but utilizes an application manager specifically for the SimBank application to make coding tests simpler and produce clearer code.

As can be seen at the top of the test, some new objects are provisioned. An ISimBank is created by using information in the CPS to provide the address and ports of the application, and an IAccount is created, which includes configuration in the annotation (see Figure 5-10).



```
@SimBank
public ISimBank         bank;

@Account(existing = false, accountType = AccountType.HighValue)
public IAccount         account;
```

Figure 5-10 New objects

Because these annotations are in the test, the Galasa framework creates the SimBank Manager to provision these objects. The use of an application manager in this case provides a way for the technicalities of the application to be hidden and reduces the amount of prior knowledge that is required by the tester. This feature allows tests to be more quickly and easily produced.

The SimBank Manager also provides functions to create accounts for the SimBank application as the tester requires. These functions allow for less narrow testing across various accounts with varying balances.

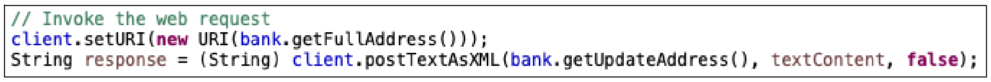
The provisioned objects feature methods to make writing tests less tedious and produce cleaner and easier to read code.

Looking at the previous test, obtaining the balance of an account required the tester to navigate through windows by using the terminal and required an understanding of how the windows are laid out to get there.

With the provisioned account, this process is done through the getBalance() method and requires no previous knowledge of how the application is laid out. Any changes to the application's architecture must be made within the manager to get the balance instead of each test that requires it. This is similar for the ISimBank object.

The web address that is used in the web services section of the test was read from the ZosImage that contains information from the CPS in the previous test. In this test, the ISimBank object has the information about the application provided to it by the manager and the getFullAddress() can be used to retrieve it instead of hardcoding the required port.

This example shows how the configuration of provisioned systems does not need to be known by the tester and can be fully configured in the CPS. Different properties that are set in the CPS allow for one test to be tested against multiple different environments without having to change the source code and allows for tests to be immutable (see Figure 5-11).



```
// Invoke the web request
client.setURI(new URI(bank.getFullAddress()));
String response = (String) client.postTextAsXML(bank.getUpdateAddress(), textContent, false);
```

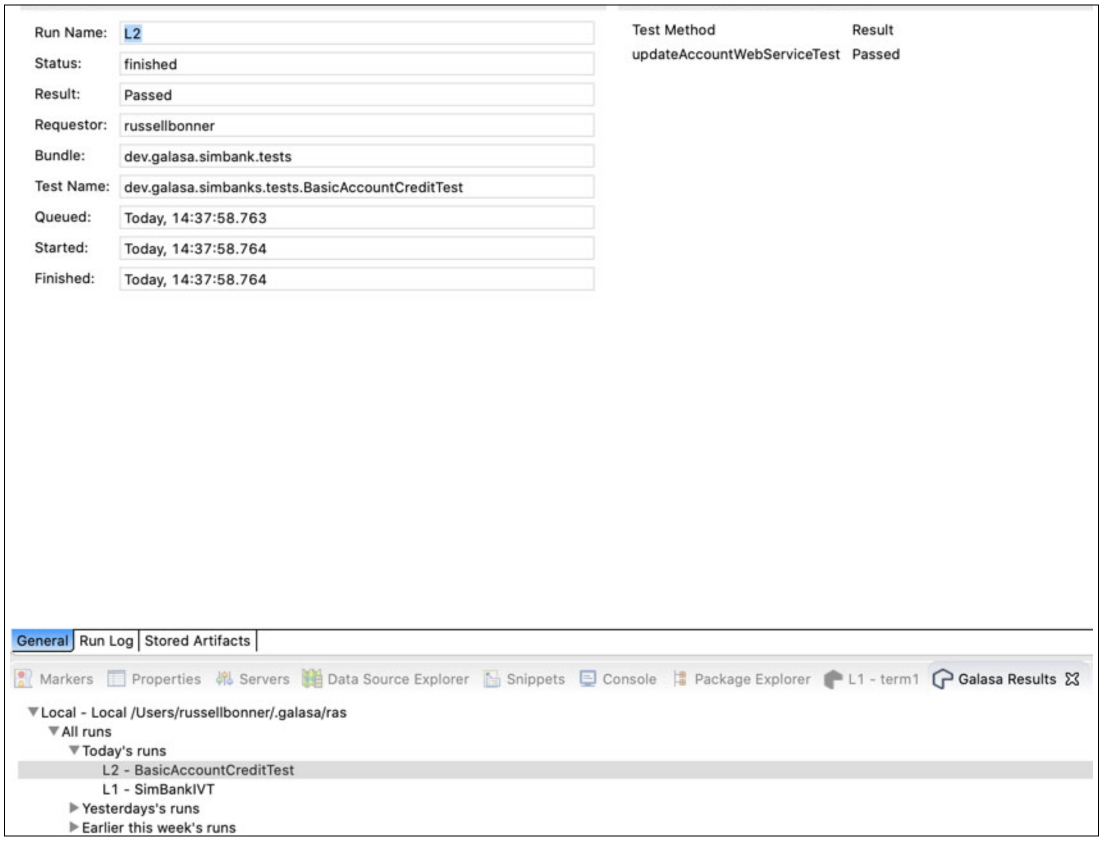
Figure 5-11 Start the web request

This test also an example how elements of the test can be stored in the Result Archive Store (RAS) and be viewed after the test through the Galasa Results tab in Eclipse, or through the `~/.galasa/ras/` path. The RAS is used in this case to store information that can be useful after the test completes to help with debugging the test or provide more information about the test. This test stores the web service request and response XML.

Each of these different levels of testing provides an example of how to gain familiarity with the Galasa Testing Framework and move to more complicated provisioned tests. They should also introduce aspects of the CPS and RAS in a practical context and why they are important.

5.5.4 Reviewing the test results

Results of the test runs can found in the Galasa Results view in Eclipse (see Figure 5-12).



Run Name:	L2
Status:	finished
Result:	Passed
Requestor:	russellbonner
Bundle:	dev.galasa.simbank.tests
Test Name:	dev.galasa.simbanks.tests.BasicAccountCreditTest
Queued:	Today, 14:37:58.763
Started:	Today, 14:37:58.764
Finished:	Today, 14:37:58.764

Test Method	Result
updateAccountWebServiceTest	Passed

```
▼Local - Local /Users/russellbonner/.galasa/ras
  ▼All runs
    ▼Today's runs
        L2 - BasicAccountCreditTest
        L1 - SimBankIVT
    ▶Yesterdays's runs
    ▶Earlier this week's runs
```

Figure 5-12 Reviewing test results

5.6 Extending Galasa:SimBank Tests with own test

After getting familiar with the provided example tests, extending the tests with the SimBank application with your own tests helps in understanding how to write tests for other systems.

When writing SimBank tests, the tester follows the same layout of tests as the provided examples by creating a not null test to ensure that all of the objects that are provisioned correctly and then, moving on to writing the test fully by using the terminal to navigate windows. Finally, the manager is used to hide unnecessary code.

Because the transfer window of the SimBank application is not used in the example tests, it can be used to write tests against the application.

5.6.1 Simple IVT

As with the example tests, a good place to start is understanding which objects are needed for the test and set up the annotations and interfaces for these objects. A not null test should also be written, similar to the one in the SimBankIVT to ensure that each object is provisioned correctly.

To get a better understanding of how the application is laid out, in addition to the not null, the tester should use a terminal object to navigate to the transfer window of the SimBank application.

These two tests provide experience with provisioning objects through Galasa, and an understanding how to use the objects in a specific environment. For example, the terminal in this environment requires a waitForKeyboard() whenever the window changes to allow the application to catch up (as seen in the example tests). These two simple tests make up tests similar to SimBankIVT and ensure that the tester understands the basics of the framework before writing a test.

5.6.2 Bank transfer test

The test for transferring money is composed of three stages: First, the balance of two separate accounts is checked and stored. Second, an amount is transferred from one account to the other. Finally, check that the correct amount was transferred between the accounts by comparing the current balances to the previous balances.

Obtaining the balance of each account can be done by navigating through the application to the Browse window. (An example of how this process is done is in the BasicAccountCreditTest from the provided examples.)

Big Decimal is used as a type to store the balance because it provides more accurate arithmetic when comparing the balances. The accounts `123456789` and `987654321` can be used for this test because they are hardcoded into the SimBank application.

After the balances are obtained, the tester navigates by using the terminal to the transfer window so that money can be transferred between the two accounts. The test should then enter the two account numbers and an amount to transfer between the two. An assertion can be made here to ensure that the correct message is displayed in the window to notify the user that the transfer was successful.

Then, the balance of the two accounts is taken again and compared with the amount that was transferred and their original value. During the test, objects, such as the Logger and StoredArtifactRoot, can be used to let the tester log aspects of the test to the run log or store information to the RAS because they can provide more information about a test run. Through this process, the tester learns how to write a simple Galasa test on hard-coded values and gains familiarity with the application.

5.6.3 Using a Manager

Moving on from writing out a full test, the test can be improved to make use of the SimBankManager to provision different accounts for more testing variety and reduce the code that is required to traverse the application. Because the manager can provision accounts that are not hard-hardcoded, this feature provides more reliability and robustness in the test because it is not testing only one case.

An example of how to provision an account is shown in the ProvisionedAccountCreditTests test from the provided examples. By using this process, you can provision one account with a high balance to transfer from, and one account with a low balance to which to transfer.

Also, the use of account objects helps the tester delete the code that is used to get the account balance because the method getBalance() provides this function. No function is available for transferring money in the SimBankManager, but this function can be added to the ISimBank and SimBankImpl objects.

Explore the manager code that is provided with the example tests to see how the getBalance in AccountImpl.java works. This same function can be used to navigate to the transfer window, enter both account numbers, and complete the transfer. Adding such functions to the manager allow for more tests to be quickly created because that the code that is required to transfer funds only ever must be written in the manager.

This test can be extended easily to test fail cases as well, such as transferring funds that are not available or transferring to an account that does not exist and making sure that the suitable error messages are present. Producing a test and using and updating a manager provides an insight into how the provisioned objects of managers work and how they can be written for other environments.

Galasa:SimPlatform provides a simple application in which a tester can get to grips with the Galasa testing framework to gain experience and an understanding before writing tests and managers for their own system.

6 Creating Galasa tests

This chapter introduces the practical task of writing your own Galasa tests on a local machine. Starting with a simple test, it progresses through several examples that increasingly use the set of Managers that are distributed with the Galasa system.

Note: It assumed you understand the integrated development environment (IDE) that you selected to use with Galasa (Eclipse was chosen as the first supported IDE). Detailed user interface manipulation is not discussed here. For more information, including how to install and run tests, see this web page.

To make things simple, the examples that are used in this chapter are written as though you were adding them to a Galasa SimBank project. Therefore, do not set up your own project first.

When you are comfortable with the workflow and technical details that involved in local testing, you can integrate your test libraries with your CI/CD pipeline.

This chapter includes the following topics:

6.1 Overview of your first Galasa test

The Galasa equivalent of “Hello world!” might be a simple test that checks that one string is a substring of another. Although it does not test anything useful, it demonstrates how to organize your early Galasa code (see Example 6-1).

Example 6-1 MostBasicTest.java

```
package dev.galasa.simbanks.tests;
import static org.assertj.core.api.Assertions.assertThat;
import dev.galasa.Test;
@Test
public class MostBasicTest {
    /**
     * Test that "Deep integration testing"
     * contains "Deep"
     */
   @Test
   public void helloWorldTest() {
   assertThat("Deep integration testing").contains("Deep");
   }
}
```

Whenever you create a test or modify an existing test in Galasa, you must run its containing project as a Maven Install. Then, you can create and run a run configuration for the test.

When you run this first example, your console log contains a stream of messages, some of which confirm that the test method `helloWorldTest` ran and passed:

```
---- *** Start of test method
dev.galasa.simbanks.tests.MostBasicTest#helloWorldTest,type=Test
----
11/01/2020 10:48:03.241 INFO dev.galasa.framework.GenericMethodWrapper.invoke -
Ending
----
*** Passed - Test method
dev.galasa.simbanks.tests.MostBasicTest#helloWorldTest,type=Test
*****************************************************************
```

Despite the simplicity of this first test, consider the following important points:

- To use the Galasa framework, you must import `dev.galasa.Test`.
- The `@Test` annotation must be applied to every class and method that you want the Galasa framework to process.
- Every class and method that you want to be processed by the Galasa test framework should be declared as `public`.
- You do not have to instantiate the main test class or specifically run any of your test methods because this process is handled by the core Galasa framework and its test runner. Therefore, you do not have to provide a `main` entry point.
- Galasa works well with the `assertj` assertion library.

6.2 Moving to the next level with Managers

Enterprise-scale integration testing is different (by more than only a degree) from local or unit testing. Some of the differences include the following examples:

- Enterprise-scale testing often involves testing system flows across heterogeneous mixes of hardware and software.
- The complexity of the entire system under test is often so great that it defeats understanding by any single person, even an expert.
- Provisioning code can get in the way of test code; therefore, it becomes more difficult to change the provisioning environment and test environment.
- At scale, testing becomes a social activity and it is increasingly important to ensure isolation of test environments when multiple test suites are running in parallel.
- Automation is key to a testing team’s ability to manage a growing catalog of regression tests.

Managers are a Galasa abstraction that mitigates these challenges, which makes them more manageable. They offer a means to separate provisioning code from test code, and (perhaps more importantly) a way to partition the range of human skills and knowledge that is required to coordinate the exhaustive testing of an enterprise-scale application.

Managers are often written by expert niche specialists with extensive knowledge of a specific area (for example, z/OS batch operations). Managers are used by testers who want to include powerful features into their tests but do not want to (or cannot) write those features themselves.

Informally, you can think of Galasa Managers as libraries of bundled test functions that are available to test writers to call upon as required.

6.3 Writing a test by using a Galasa Manager

Several Managers are delivered with the base Galasa distribution, and more are being added as the development effort progresses.

This section explores a second simple test in which the `HTTPClientManager` is used to read the contents of Google’s home page and test the existence of a small fragment of text (see Example 6-2).

Example 6-2 BasicHttpClientTest.java

```
@Test
public class BasicHttpClientTest {

 @HttpClient
 public IHttpClient client;
 public static HttpClientResponse<String> resp;
 @Logger
 public Log logger;
 @Test
 public void testNotNull() {
  // Check all objects loaded
  assertThat(client).isNotNull();
  assertThat(logger).isNotNull();
  }
  @Test
  public void checkSite() throws
  HttpClientException, URISyntaxException {
   client.setURI(new URI("https://google.com"));
   resp = client.getText("/");
   assertThat(resp.getContent()).contains("I'm Feeling      Lucky");
   logger.info("Status code: " + resp.getStatusCode());
  }
}
```

6.3.1 Understanding BasicHttpClientTest.java

For brevity, the package declaration and imports were omitted in Example 6-2 on page 59. Your IDE provides the necessary assistance if you need it. Remember also that the example tests that are provided with Galasa (for example, `SimBankIVT.java`) are a good reference.

First, the Manager is declared within the body of the main test class (see Example 6-3).

Example 6-3 Declaring a Manager

```
@HttpClient
public IHttpClient client;
```

When it comes to declaring Managers in Galasa, a universal pattern is used:

- Include an annotation that names the relevant Manager. This annotation is processed by the Galasa framework to insert code behind the scenes and instantiate the Manager before the test is run (in this case, `@HttpClient`).
- Declare a public Interface object that acts as a proxy within your code to the instantiated Manager (in this case, `IHttpClient`).

Next, a static `HttpClientResponse` object is declared. This object is a template class and we chose the `String` variant because we intend to store the result of an HTTP GET as a `String`.

The `@Logger` annotation and `logger` declaration provides us with the capability to insert messages into the Galasa console output.

The `testNotNull` test method is an example of defensive programming that is designed to ensure that all declared Managers or their proxies (`client` and `logger`) instantiated correctly.

The main test method `checkSite()` is uncluttered and simple to understand. It sets the target URI for the client, retrieves the root page of the target URI by using `resp=client.getText("/")` and asserts that the page's contents contain the wanted String: "I'm Feeling Lucky".

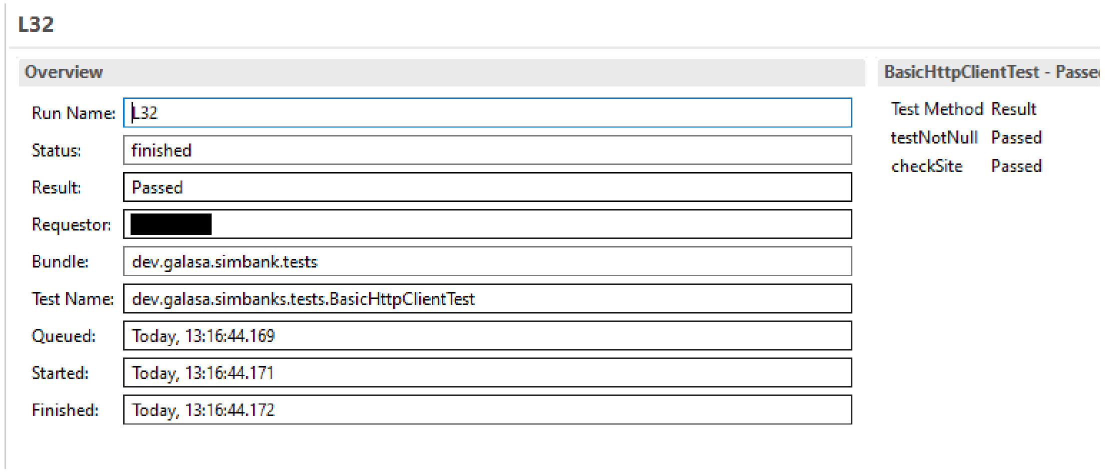
The key takeaway here is that we did not have to write any of the code that is involved in provisioning an HTTP request. Instead, we used a declared Manager that did not need to be instantiated. This ease of use extends across the range of Galasa Managers, including those Managers that can interact with z/OS.

6.3.2 Running BasicHttpClientTest

When you run this test, your console output contains (among others) the following lines that indicating that the `testNotNull and checkSite` test cases ran successfully, that the `BasicHttpClientTest` class passed, and that the wanted log output was generated (in this case, a status code of 200):

```
*** Passed - Test method
BasicHttpClientTest#testNotNull,type=Test
*** Passed - Test method
BasicHttpClientTest#checkSite,type=Test
*** Passed - Test class BasicHttpClientTest
INFO BasicHttpClientTest.checkSite - Status code: 200
```

The Galasa Results tab of your IDE shows some of the important characteristics of your recent test runs. The exact data that it contains depends on which Managers you use in your tests; however, it contains at a minimum the information that is shown in Figure 6-1.



L32

Overview

Run Name:	L32
Status:	finished
Result:	Passed
Requestor:	[redacted]
Bundle:	dev.galasa.simbank.tests
Test Name:	dev.galasa.simbanks.tests.BasicHttpClientTest
Queued:	Today, 13:16:44.169
Started:	Today, 13:16:44.171
Finished:	Today, 13:16:44.172

BasicHttpClientTest - Passe

Test Method	Result
testNotNull	Passed
checkSite	Passed

Figure 6-1 Galasa results

If this tab is not visible, in Eclipse, choose **Window** → **Show View** → **Other** and then, **Galasa** → **Results**. If you see the message `Framework not initialized`, choose **Galasa** → **Initialize Galasa Framework** from the main menu.

6.4 Managers provided with and planned for Galasa

The use of Managers is the real key to the power of Galasa. Many Managers are provided with Galasa and more are planned or are in development at the time of this writing.

The current or planned Galasa Managers are listed in Table 6-1.

Table 6-1 Current and planned Galasa Managers

IBM AIX® Manager	Artifact Manager	Artifactory Manager
CECI Manager	CEMT Manager	CICS TS Manager
Docker Manager	ElasticLog Manager	Galasa EcoSystem Manager
GitHub Manager	GitLab Manager	HTTP Client Manager

IMS DB Manager	IMS TM Manager	IP Network Manager
ISPF Manager	JMeter Manager	Kubernetes Manager
Liberty Manager	Linux Manager	MongoDB Manager
Nexus Manager	OpenLDAP Manager	OpenStack Manager
RTC Manager	Selenium Manager	TSO Manager
UrbanCode Manager	WebSphere Application Server Manager	Windows Manager
WSIM Manager	z/OS Batch Manager	z/OS Connect Manager
z/OS Console Manager	z/OS DB2 Manager	z/OS File Manager
z/OS Manager	z/OS IBM MQ Manager	z/OS PT Manager
z/OS Security Manager	IBM z/VM® Manager	

For more information and an updated status, see this web page.

6.5 Using multiple Managers

It is common for multiple Managers to collaborate within a single test scenario. A good example of this collaboration might be the Docker Manager IVT (Installation Verification Test), which is available at this web page.

This example uses the `HTTPClientManager` with the Docker Manager.

Three of the important declarations are shown in Example 6-4.

Example 6-4 Three declarations

```
@DockerContainer(image = "library/httpd:latest", dockerContainerTag = "a", start =
false)
public IDockerContainer container;
@ArtifactManager
public IArtifactManager artifactManager;
@HttpClient
public IHttpClient httpClient;
```

The `@DockerContainer` annotation references the latest version of Docker's `HTTPd` image, and the corresponding container is declared as an `IDockerContainer`.

`@ArtifactManager`, which provides a templating service and access to resources within test bundles, is used to provision some template HTML for pushing to a Docker container.

`@HttpClient` enables the test to visit HTML endpoints that are exposed by the HTTPd server that is running on a live Docker container.

First, check that the Docker Manager was provisioned (see Example 6-5).

Example 6-5 checkDockerContainerNotNull()

```
@Test
public void checkDockerContainerNotNull() throws DockerManagerException {
        assertThat(container).as("Docker Container").isNotNull();
}
```

Then, the container is recycled to ensure it restarts if bounced (see Example 6-6).

Example 6-6 startandStopContainer()

```
@Test
public void startAndStopContainer() throws DockerManagerException {
  logger.info("Stopping the Docker Container");
  container.stop();
  logger.info("Starting the Docker Container");
  container.start();
}
```

By using the Artifact Manager, a file is stored in the container so that it can be retrieved later. An **ls** command is issued to ensure that it exists on the file system (see Example 6-7).

Example 6-7 storeFilesInContainer()

```
@Test
public void storeFilesInContainer() throws DockerManagerException,
TestBundleResourceException {
  IBundleResources bundleResources =
artifactManager.getBundleResources(getClass());
  // Retrieve the test html file
  InputStream isHtml = bundleResources.retrieveFile("/test1.html");
  // Store it in the container
  container.storeFile("/usr/local/apache2/htdocs/test1.html", isHtml);
  // Check it is there via ls command
  IDockerExec exec = container.exec("/bin/ls", "-l",
"/usr/local/apache2/htdocs/test1.html");

  assertThat(exec.waitForExec()).as("The waitForExec finished true").isTrue();

  String cmdResult = exec.getCurrentOutput();
  logger.info("Result from ls:-\n" + cmdResult);
  assertThat(cmdResult).contains("-rw");
}
```

The file is retrieved by using an `httpClient`, and its contents verified (see Example 6-8).

Example 6-8 retrieveHtml()

```
@Test
public void retrieveHtml() throws DockerManagerException, HttpClientException,
URISyntaxException {
  InetSocketAddress exposedPort =
container.getFirstSocketForExposedPort("80/tcp");

  assertThat(exposedPort).as("Correctly retrieved the exposed port").isNotNull();
  URI uri = new URI("http://" + exposedPort.getHostName() + ":" +
exposedPort.getPort());
  httpClient.setURI(uri);
  String html = httpClient.get("/test1.html");
  assertThat(html).as("Checking the HTML container the Galasa constant
text").contains("Galasa Docker Test");
}
```

The container log is examined to ensure that the file was retrieved (see Example 6-9).

Example 6-9 retrieveContainerLog()

```
@Test
public void retrieveContainerLog() throws DockerManagerException {
  String log = container.retrieveStdOut();
  logger.info("Container Log:-\n" + log);
  assertThat(log).as("checking that the test1.html was retrieved and
logged").contains("\"GET /test1.html HTTP/1.1\" 200");
}
```

Finally, the file is retrieved to ensure it contains the expected text (see Example 6-10).

Example 6-10 retrieveFile()

```
@Test
public void retrieveFile() throws DockerManagerException, IOException {
  String htmlTest1 =
container.retrieveFileAsString("/usr/local/apache2/htdocs/test1.html");

  assertThat(htmlTest1).as("check we can pull back the file").contains("<h1>Galasa
Docker Test</h1>");
}
```

The main takeaway here is to demonstrate that multiple Managers can be effectively used to deliver a single testing objective.

7

Extending Galasa

Galasa is highly extendable. The main point of extension is the Managers, which are the true power of Galasa, not the actual framework, and are the focus of this chapter.

Another way of extending Galasa is to provide alternatives to the default “Stores”:

- Configuration Property Store (CPS)
- Dynamic Status Store (DSS)
- Result Archive Store (RAS)
- Credential Store (CREDS)
- Certificate Store (CERTS)

You consider only extending the Stores if the default providers, etcd3, CouchDB and File, are not suitable, or banned, for your environment.

This chapter describes the use of the Stores but does *not* explain how to extend them.

Writing Galasa tests does not require much experience with Java because it leans toward a scripting use of Java. However, a thorough understanding of Java is required when writing a Manager. Inheritance and reflection are two Java topics that must be understood. An understanding of Maven and OSGi also is beneficial.

This chapter includes the following topics:

7.1 What is a Galasa Manager

The term *Manager* comes from the concept of managing a product or tool on behalf of a test. Although the role of the Manager is to remove boiler plate code from tests and provide a proven, reliable interaction with the product or tool, a Manager can be written to do anything that is required during the lifecycle of a test.

Managers are normally run during the lifecycle of a test if the test indicated that it needs that Manager. However, Manager code can be run at other places in the Galasa ecosystem, as shown in the following examples:

- During resource management where resources are cleaned and monitored in a background task
- Contributing to the Eclipse UI, an example is the z/OS 3270 Manager that provides live views and playbacks of 3270 terminal windows that were recorded during a test run
- Providing context to the web UI, the z/OS Manager indicates which z/OS image a test is running on
- Providing assistance in the web UI for setting up configuration properties the Manager requires during a test run

With these examples in mind, Managers in Galasa include the following possible use cases:

- Provision and clean up CICS TS regions for use in a test.
- Provide the test access to central or embedded resources; for example, files.
- Log the result of a test to a dashboard.
- Influence the result of a test based on whether a GitHub issue is still open; that is, if an issue is still open, mark a failing test as Passed-Defect instead of Failed because we are aware of the failure.
- Modify the capacity of test resources based on its status; for example, reduce the test capacity of a z/OS image if the CPU% is above a threshold.
- Provide failure analysis on a failed test, a Manager looks for specific events in a test and provides clues about why a test failed in the test result.
- Log a failure analysis to a dashboard so that Galasa administrators can see whether a pattern evolves and rectify possible infrastructure issues.

7.2 Why should I write a Galasa Manager?

It is likely that more than 100 public Galasa Managers will be available by the time Galasa becomes stable. Most of these managers are written by the IBM Galasa team based on client and internal requirements. Many those Managers are written to help the CICS TS development team test CICS TS.

However, the IBM Galasa team cannot write Managers for all possible products and tools that are available. Also, Managers are included in only the Galasa offering if it is deemed that they are of use to many teams.

It is beneficial to write a custom Manager for many reasons, including the following examples:

- Provisioning and interaction with a client application.
- Removing boiler plate code from tests (even if it is not directly related to a product or application).

- Lacking a public Manager that deals with the product or tools that you want to integrate into the test.
- Applying site standards to a Manager, your data set or file names might be different from what the Manager expects. This is assuming that the Manager provided the means to override specific settings.
- Providing site-specific methods of provision resources; for example, the z/OS D&T Manager provides a standard way of provisioning z/OS D&T instance that you can override with a new Manager by using your site standards.

7.2.1 Is it easy to develop a new Manager?

The short answer is yes.

With the Galasa plug-in installed into your integrated development environment (IDE), developing and testing your new Manager and seeing how it interacts with your test is relatively easy. It is no different than developing a test, and you can use the power of the IDE to set breakpoints, step through the code, and view variables.

The difficulty exists when you want to make your Manager available to the Galasa automation system. Unfortunately, you must know about Maven repositories, OSGi Bundle Repositories (OBR), possibly Docker, and build pipelines. No one-size-fits-all solution exists; however, in this chapter, we describe the steps to provide the Manager to automation runs (assuming the standard toolkits are available).

The best way to learn how to develop a Manager is to find a Manager that provides similar services and review the source code and `pom.xml`.

7.2.2 Concepts needed when writing a Manager

You must be familiar with many concepts, rules, and information before you can start to write a Manager; however, you must embrace two important concepts rules.

The first and most important concept is Galasa and its Managers were developed and written to make the life of the tester easier when dealing with complex integration testing. At most, it should take the tester only three lines of code to provision the most complicated integrated testing environment. The fact that it might take multiple managers working together should not matter.

The second important concept is that a Manager should specialize in its area and not duplicate the work of another Manager. For example, if your new application Manager requires a Docker container as part of provisioning of the test environment, it should not communicate with the Docker Engine directly. The new Manager should ask the Docker Manager to do the work. If the Docker Manager does not include the functions that are needed, the Docker Manager should be extended rather than implement the code yourself.

7.3 TPI, SPI, and API

A Manager provides three types of interfaces for tests, Managers, or web services to use. Because application programming interface (API) is an overused term, Galasa differentiates between the different types of APIs available:

- Testing Programming Interface (TPI)

  Items in the TPI are intended to be used by tests and managers within the testing lifecycle. They often can be found in the base package of the manager; for example, the TPI of the Artifact Manager, `dev.galasa.artifact`. Tests should only ever use the TPI; never the SPI or API.

- Services Programming Interface (SPI)

  SPI items are for Manager to Manager interfaces during the testing lifecycle and in areas, such as resource management. SPI tends to feature more powerful methods that you do not want to be exposed to the test.

- API

  The API is used for web services that can be called from the Galasa web UI or from external services, such as Jenkins. Visual Studio Code use many of the Core APIs to communicate with the Galasa ecosystem; Eclipse uses the SPI because we can load the framework into Eclipse.

7.4 Manager lifecycle during a test run

Using the power of OSGi, when a test bundle is loaded by the framework, all referenced Manager and utility bundles are automatically loaded. This process is achieved by reviewing the test bundle's MANIFEST.MF to see what other bundles are required. However, only OSGi bundles are loaded into the test run. To be a Manager, the Manager class must register as a OSGi IManager service by using the following code:

```
@Component(service = { IManager.class })
public class ArtifactManagerImpl extends AbstractManager
```

The Artifact Manager implements the IManager interface by way of the AbstractManager. It also registers to the OSGi framework as an IManager service by using the `@Component` annotation.

When the framework test runner starts, it asks OSGi for all services that are registered as an IManager service. When the test runner finds all of the Managers, it begins the lifecycle.

Note: The Managers are loaded because they are used by any test within the bundle; however, they cannot be used for the specific test that is run.

The IManager interface contains all the methods that are called by the framework test runner during the test lifecycle. Most the methods exist in AbstractManager; therefore, they do not need to be included in a new Manager.

The following methods are called by the framework in the order that they are presented here:

- `extraBundles()`

  This method provides the Manager a way to request optional bundles to be loaded before any of the Managers are initialized. The z/OS Manager is one of the few Managers that requires this function and uses it to load implementations of the batch and file interfaces.

  The z/OS Manager uses zOS/MF to access batch jobs and files on the mainframe. Alternatives to zOS/MF are available in Galasa and the z/OS Manager can be configured to load those alternatives by way of a CPS property.

- `initialise()`

  This method is the opportunity for the Manager to decide whether it wants to initialize itself for this test run. This process normally is achieved by examining the test class if it uses any of the Manager's annotations. If the Manager wants to be part of the lifecycle from this point, it must add itself to the activeManagers list.

  The order in which the Managers are initialized can be considered random.

  If the Manager requires the services of another Manager, it should search the `allManagers` list, look for required Managers, and call their `youAreRequired()` method.

- `youAreRequired()`

  This method is called by other Managers to indicate that they require the services of this Manager. It is possible that this method is called before `initialize()`.

- `areYouProvisionalDependentOn()`

  After the Managers are initialized, the framework attempts to sort the them in order of provisioning. The CICS TS Manager needs the z/OS Manager to provision a z/OS Image before a CICS TS Region can be provisioned; therefore, the CICS TS Manager is provisionally dependent on the z/OS Manager and should return true.

  The framework throws an exception if a loop exists in the dependency chain.

- `anyReasonTestClassShouldBeIgnored()`

  This method provides the option to mark the test as `Ignored`, which can be used to bypass tests that are not applicable for a specific version of an application. The Manager should return a human readable reason after which the framework updates the run result with `Ignored` and ends the test run.

- `provisionGenerate()`

  The Manager should provision resources for the test to use in this method, but should not build those resources. It is intended that `provisionGenerate()` runs quickly to determine whether all of the resources are available before any Manager attempts to build those resources.

  How provisioning is achieved is up to the Manager and often is dependent on the configuration properties that are provided by the tester. Provisioning can be building a CICS TS region from scratch or providing an applid of a CICS TS region in a pool.

  The Manager decides how resources are to be provisioned and allocates resource names at this point. If sufficient resources are not available, the Manager throws the `ResourceUnavailableException`. The framework then shuts down the Managers and places the test back on the run queue if it is an automated run and is ready to retry when the resource are available.

  The Manager avoid building or deploying resources during this method unless it is the only way to obtain a resource ID or ensure that the resources are available. This process avoids expensive and time-consuming build operations if some of the resources that are needed by the test are unavailable.

- `provisionBuild()`

  After the resources are allocated, this method is called to build those resources. It can be assumed that all of the resources that are required by test are available so that we can perform those time-consuming build operations.

  If a a problem occurs during the build operation, the Manager throws an exception. Errors in this method and `provisionGenerate()` cause the test to be marked as `Environment Failed` because this marking helps differentiate between a true test or build failure.

  If possible, avoid starting the built resources at this point; instead, wait for `provisionStart()`, which is called after all Managers successfully build their resources.

- `provisionStart()`

  After all the managers allocated and build their resources, this method is called to start those resources if applicable; for example, a CICS TS region or Docker container.

- `startOfTestClass()`

  This method is called when an instance of the test class is created and the test is about to start. This method can be used for timing, logging, or metrics.

- `fillAnnotatedField()`

  This method is called before every test method to allow the Manager to fill the values of field variables in the test class instance with the resources that were provisioned. If the tester decided to overwrite a field value, every field is filled whenever this method is called.

- `anyReasonTestMethodShouldBeIgnored()`

  Similar to `anyReasonTestClassShouldBeIgnored`, this method gives the Manager the option for individual test methods to be ignored if they are not applicable for this test run.

- `startOfTestMethod()`

  This method is called before every test method in the order they exist in the source code. The method is called for `@Test`, `@Before`, `@After`, `@BeforeClass`, and `@AfterClass` methods.

- `endOfTestMethod()`

  This method is called after every test method finishes. This method can present the opportunity for the Manager to perform more test validation, change the test result, or perform more logging.

  If the Manager wants to perform more validation of the test (for example, check for transaction abends) and detects a test failure, the Manager throws an exception.

  The Manager can change the result of the test method, which is normally `passed`, `failed`, or `ignored`. The manager can return a new string that is based on the exception or result; for example, `failed_with_defect` if the failure is known and the GitHub issue is still open.

- `testMethodResult()`

  This method is called when the result of the test method is resolved. This method is intended for logging or failure analysis purposes.

- `endOfTestClass()`

  As with `endOfTestMethod()`, this method can be used to perform post test validation and influence the test result.

- `testClassResult()`

  As with `testMethodResult()`, this method can be used to perform logging or failure analysis.

- `provisionStop()`

  At this point, the test finished and the Manager stops any resource that was started for this test run; for example, stop any running Docker container.

- `provisionDiscard()`

  Now that all the resources are stopped, the Manager deletes or discards those resources; for example, delete the Docker containers that were created from this run.

- `peformFailureAnalysis()`

  This method is the opportunity for any Failure Analysis Managers to examine logs or stored artifacts to determine possible causes of a test failure. The analysis result can be logged with the test or written to a remote server. For example, it is possible now to interrogate remote metrics to determine the state of systems to see if a reason exists as to why a test timed out (perhaps the CPU was at 100% for a prolonged period).

- `endOfTestRun()`

  Marks the end of the test run, possible uses are metrics and logging.

- `shutdown()`

  Requests that the Manager clean up any resources it used. Although the Java JVM is stopped after this point, it is still better to close SSH connections, and so on.

7.5 Lifecycle patterns

In this section, we examine lifecycle patterns.

7.5.1 initialise() and youAreRequired() methods

Because these methods can be called in any order and `youAreRequired()` can be called multiple times, the Manager must record if it is required so that it does not initialize itself multiple times.

In `youAreRequired()`, first check to see if the Manager instance exists in the `activeManagers()` list:

```
if (activeManagers.contains(this)) {
            return;
        }
```

If it does not exist, the instance must be added to the `activeManagers` list and a `required` boolean field must be set to `true`.

The `initialise()` method is called only once; therefore, check the `required` flag to see if `youAreRequired()` was called. If not, the Manager checks for annotations and fields.

7.5.2 Provision generation and filling fields

The AbstractManager features a few methods that make provision generation and filling the fields easier, although it can appear a bit confusing at first.

Review the Artifact Manager and the `@ArtifactManagerField` annotation. This is used to annotate TPI annotations so that utility methods can find annotations without hard coding them in the code. The `@ArtifactManager` and `@BundleResources` are annotated with this annotation.

By sing this pattern, a simple call to the AbstractManager `findAnnotatedFields(ArtifactManagerField.class)` returns all fields in the test object that are annotated by using this Manager's annotations.

By using the same technique during `provisionGenerate()`, a call to the AbstractManager `generateAnnotatedFields(ArtifactManagerField.class)` assists in generating the annotated fields. The `generateAnnotatedFields` looks for all Manager annotated fields and searches for a method in the Manager that matches the field profile, as shown in the following example:

```
    @GenerateAnnotatedField(annotation = BundleResources.class)
    public IBundleResources fillBundleResources(Field field, List<Annotation>
annotations) {
        return new BundleResourcesImpl(field.getDeclaringClass(),
getFramework());
    }
```

This method is called for every field in the test object that is annotated with `@BundleResource` and is type `IBundleResource`.

By using this pattern, the AbstractManager also deals with the `fillAnnotatedField()` call.

7.6 Configuration Property Store

As the name indicates, the Configuration Property Store (CPS) is the Galasa framework and the Managers are configured. The default Galasa ecosystem uses etcd3 as a key/value store so that the same configuration can be shared across the testing teams. In the future, alternatives to etcd3 will be made available, possibly mariadb, db2, couchdb, and so on.

An simple example of a configuration property is shown here:

```
zos.image.MV2C.ipv4.hostname=winmvs2c.hursley.ibm.com
```

This property is used by the z/OS Manager to give the IPv4 host name of the MV2C z/OS Image (equivalent properties for other platforms are available).

The namespace of the property is `zos`, which means the property belongs to the z/OS Manager. No other Manager can read from or write to this namespace. If another Manager wants to know the value of the property, it must use the z/OS Manager's TPI/SPI. This is required so that the z/OS Manager can change the format of the properties without fear of breaking other Managers.

All name spaces must start with a lowercase letter, a - z. All properties must have at least three qualifiers; for example, `a.b.c`. Reserved name spaces are available within Galasa; for example, framework, dss, and secure, which can be accessed by way of the framework/Manager SPIs.

When choosing a namespace for a new Manager, ensure that it does not conflict with an existing Manager and is appropriate. If the Manager is for internal use or is for a specific team, suffix the namespace with `-team` or `-int` to avoid future conflicts.

Property names should be lowercase, except for resource IDs or tag names that should be uppercase. By using this convention, it is easier to locate properties that are related to a resource; for example, MV2C. After the namespace, the format of the property is up to the Manager.

It is intended that the contents of the CPS remain reasonably static. If a Manager must update a property many times (for example, recording how many resources are used in a test), this information should be stored in the DSS that is designed for rapid updating.

7.6.1 Accessing properties in the CPS

One of the first things a Manager should do during initialization is to obtain an instance of `IConfigurationPropertyStoreService` as shown in the following example:

```
IFramework.getConfigurationPropertyService("zos")
```

An `IConfigurationPropertyStoreService` is returned for accessing or updating properties in the `zos` namespace. A Manager never accesses a namespace that does not belong to it. Future enhancements to the Galasa framework night enforce this rule.

The `IConfigurationPropertyStoreService` includes an extended getProperty method that makes it easier to obtain site-wide defaults or instance-specific settings, as shown in the following example for a zos Manager:

```
   getProperty("image","ssh.port","PLEX2","MV2C")
   will search for properties in the following order:-
   zos.image.PLEX2.MV2C.ssh.port
   zos.image.PLEX2.ssh.port
   zos.image.ssh.port
```

The use of a default port number can set site-wide, sysplex-wide, or specific information for an image, without the Manager having to perform multiple `getProperty()` calls.

All property gets are recorded during a testing lifecycle and can be found in the test run's stored artifacts in a file called `framework/cps_record.properties`. If an attempt was made to access a property and that property is missing, this process also is recorded.

7.7 Dynamic Status Store

The Dynamic Status Store (DSS) is used to store the dynamic status of the Galasa ecosystem and Managers. The default Galasa ecosystem uses etcd3 as its data store (as does the CPS). In fact, the default ecosystem uses the same etcd3 instance.

The properties usage is the same as the CPS in that all properties are owned by a Manager that uses a namespace; however, all properties are also prefixed with `dss.` to avoid collisions with CPS properties.

The DSS is designed for rapid updating of properties to achieve the scalability of running hundreds of tests in parallel. To avoid database locking issues within the DSS, all updates are done by the atomic updates pattern and Managers should be written to expect updates to fail and retry automatically.

Note: This area is intended to be developed further to make common usage patterns easier; therefore, this might change. The z/OS Manager always provides good examples of the use of the DSS.

To help explain how to use the DSS, the following example is a common usage pattern that uses a fictitious Manager that uses pseudo code as checks for nulls were omitted.

Manager `robin` wants to limit how many automated tests can be run in parallel in the Galasa ecosystem. By default, the maximum number of parallel tests that can use the `robin` resource should be four slots. In Galasa, we use the term *slot* as a generic name for reserving resources and providing a means of associating a test run with a resource allocation.

The first thing the `robin` Manager must do is obtain a DSS namespace instance, which is similar to the CPS:

```
IDynamicStatusStoreService dss =
IFramework.getDynamicStatusStoreService("robin")
```

All properties that are now accessed by way of the `IDynamicStatusStoreService` are prefixed with “`dss.robin.`”.

During the `provisionGenerate() lifecycle` call, we must reserve a slot for use by the test. The property we use to keep the current count of in-use of slots is `robin.inuse.slots`; therefore, we must get the current count from the DSS:

```
int currentSlots = Integer.parse(dss.get("inuse.slots"));
Check that it does not exceed the maximum:-
If (currentSlots >= 4) {
     throw new FrameworkResourceUnavailableException("Robin slots exceeded");
}
```

If no free slots are available, a `FrameworkResourceUnavailableException` is thrown so that the automated test run is queued again and retries after a random time.

Now, we must increment the current slots and create a slot resource:

```
HashMap<String,String> slotResource = new HashMap<>();
slotResource.put("slot." + uniqueId, framework.getRunName());
int newSlots = currentSlots + 1;
dss.putSwap("inuse.slots", Integer.toString(currentSlots),
Integer.toString(newSlots), slotResource);
```

In putSwap, the DSS checks that the property `dss.robin.inuse.slots` is still set to the `currentSlots` value. If it is, DSS now sets the `dss.robin.inuse.slots` to `newSlots` and creates the new DSS property `dss.robin.slot.uniqueId`.

If a `DynamicStatusStoreException` is thrown, it is likely that another test updated `dss.robin.inuse.slots` in the milliseconds that this test incremented and attempted to update. This issue is highly likely if you are running hundreds of tests in parallel. If the property was updated, the Manager loops round and restarts at the `dss.get("inuse.slots")` and attempts to increment again.

This example shows a simplified use of the DSS. How to get a `uniqueId` was not explained because it is dependent on the Manager’s usage. Also, other checks and balances were shown. Looking at Managers shows how they use the DSS and provides good examples.

When the test is finished, the Manager decrements the `inuse.slots` and deletes the slot property by using a similar technique that is shown in our example. This process ensures that the `inuse.slots` never falls below zero.

As more Managers are written, more DSS usage patterns are discovered. It is the intention of the Galasa team to create an easy way of maintaining the DSS properties of a Managers.

However, what happens if you lose a test while it is allocated a slot? A test can be “lost” in multiple ways, including the server it was running on was powered off or a user stopped a locally run test by using Ctrl+C or the “big red button”. In this instance, the slot never is freed, and you eventually run out of slots.

For this reason, all Manager that use the reserve slots or similar patterns should contribute to the Resource Management service, as explained next. The `robin` Manager can use the `dss.robin.slot.uniqueId` DSS property to find resources that are still allocated to a deleted run and clean up the slot and in-use slots properties.

7.8 Resource management

Galasa provides a way for the ecosystem to self-clean resources for test runs that were lost. This process is called the *resource management server*. This server runs as part of the Galasa ecosystem to which Manager can contribute.

Normally, the framework and Manager periodically check their resources to see if anything must be cleaned up. Usually, only one resource management server is running in an ecosystem at a time; however, if this server cannot keep up with the amount of work, more instances can be started.

Managers that contribute to the resource management server must be aware that multiple instances might attempt to clean up the same resource at the same time and should be coded to cope with that possibility. It is for this reason that atomic updating of the DSS was selected to avoid the possibility of multiple clean up instances deadlocking.

The resource management server can be used for activities other than clean up; for example, the z/OS Manager can examine the average CPU % of a test image and adjust the maximum number of tests that are allowed to use the image in parallel as suitable. Anything a Manager wants to do on a repeated basis outside of test life cycle is a candidate for running in the resource management server.

Managers that want to contribute a task to the resource management server must provide a `IResourceManagementProvider` OSGi service, as shown in the following example:

```
@Component(service= {IResourceManagementProvider.class})
public class ZosResourceManagement implements IResourceManagementProvider
```

At start, the resource management server examines the supplied OSGi Bundle Repositories (OBRs) for any bundle that provides an `IResourceManagementProvider` service and automatically loads the bundle.

The z/OS Manager provides a good use case of checking z/OS Image slots every 20 seconds to see if any of them can be cleaned up.

The resource management server provides a `ScheduledExecutorService` that Managers can use to register time-based tasks, rather than Manager maintain their own schedulers.

The `IResourceManagementProvider` provides a `runFinishedOrDeleted(runName)` method that is called whenever a test finishes or deletes if a test is lost. This feature allows the timely cleaning up of resources rather than waiting for polling. Experience shows that a poll every 20 seconds is a long time to wait for clean up if you are running hundreds of tests in parallel that are competing for the same resources.

7.9 Building Managers

At the time of this writing, Maven is the preferred tool for build Managers and tests; although in the future, alternatives, such as Gradle will be developed.

Managers are OSGi Bundles. The use of the maven-bundle plug-in with bundle packing builds the necessary MANIFEST.MF file and the OSGi component XML files for you.

After the Manager is built and deployed to a Maven repository, it must be included in one or more OSGi Bundle Repositories that Galasa was configured to use.

7.10 OSGi bundle repositories

Galasa was designed with scalability in mind so that development shops can run hundreds or thousands of tests in parallel. To achieve that level of testing throughput, many servers must be setup and configured to run Galasa tests. These servers can be Kubernetes nodes, Docker Engines, or bare metal servers, on and off of IBM Z.

To avoid the difficulty of distributing many test and Manager bundles to many engine servers, the bundles are stored in a central Maven repository and are pulled by the Galasa test runner when needed.

When a test run starts, the test runner is given the bundle symbolic name and the test class name. The test runner must discover which Maven artifact contains the test bundle. This is where Galasa’s use of the OSGi Bundle Repository (OBR) comes into play.

When a test runner starts, it is given a test class and bundle name, and a list of OBRs and Maven repositories to search. The test runner loads all the OBRs that are provided from the Maven repositories.

It then searches the loaded OBRs, in order, for the first occurrence of the test bundle name. This bundle entry contains the Maven artifact ID as a URI, and all the OSGi import dependencies and any exports.

The test runner asks the OSGi framework to load the test bundle. OSGi examines the bundle definition and attempts to resolve all the dependencies whether that is Manager or utility bundles. By using the Maven artifact ID URI, a Galasa URI extension then attempts to pull the Maven artifact from the Maven repositories.

If the Maven artifact includes a -SNAPSHOT version, the Galasa URI extension pulls the Maven artifact whenever a test runs. This process deviates slightly from the normal Maven usage where a -SNAPSHOT artifact is updated daily. Galasa's usage ensures the latest test bundle is used where tests are actively developed and deployed.

To build an OBR for use in Galasa, a Maven project with the packaging of galasa-obr must be created and deployed to a Maven repository. All dependencies of this project are included in the OBR if it includes a scope of compile. This also includes dependencies of dependencies.

8

Summary

Relying on a set of automated tests that can be run against your application is a key part of any organization’s DevOps journey. The value of validating a build of software automatically without manual intervention is compelling because it frees up testers from the monotony of rerunning tests, which allows them to focus on creating tests that can find more bugs. However, getting to that end-point can seem difficult with various tools and processes to learn.

Initially, an organization might pick a single tool to create a subset of the automation they need, such as a framework (Junit), or a technology-specific tool, such as Jmeter. These tests initially provide value to the customer, but eventually the limitations of the tools become a problem; Junit is for small fast unit tests only, not larger scale integration testing, and JMeter can run a HTTP workload, but cannot validate the data later. Therefore, more tools are added, although doing so allows more tests to be created and an integration issue starts to build.

So, how can all these tools work together to provide a holistic test framework?

Galasa was built as an integration test framework, to test applications that span multiple platforms as part of a hybrid multi-cloud and integrate all of the test tools that are needed to test such an application. As a result, you have a single test catalog, single endpoint to run tests, and a single UI that is used to review the reports from those tests. These enterprise-level features are key to unlocking the value of your automation and allowing you to deliver your DevOps journey.

You can get started with Galasa without accessing your own mainframe. Sample sample tests and simulated mainframe applications are included with which you can get started with Galasa.

You also can expand on those tests to build your own tests against your own application. Those tests can be run locally, or in the Galasa ecosystem and run as part of a DevOps pipeline.

Galasa is an open source product under an Eclipse Public License V2. Anyone can try, adopt and experience success from Galasa. It also allows us to build a community of interested users around Galasa. This community can share best practise, fix bugs and add features.

This IBM Redpaper publication is accurate as of this writing. Galasa continues to grow and gain new features and capabilities. For the most update information, see the Galasa website.

REDP-5614-00

ISBN 073845933x