Getting started with Galasa Managers

Developing Galasa Guides

April 2023

James Davies

CICS TS Development Manager

Galasa Advocate

IBM

Table of Contents

1.0 Introduction *3*

2.0 Manager Lifecycle *4*

2.1 Manager Lifecycle Phases **4**

3.0 Simple Manager Code Structure *6*

4.0 Interfacing with the Lifecycle *7*

4.1 Initialising a Manager **8**

4.2 Using Dependant Managers **9**

5.0 Annotations *11*

5.1 MyManagerAnnotationField – The Manager Annotation **12**

5.2 AnnotationName.Java – The annotation interface **13**

5.3 IAnnotationName.Java – The Object Interface **15**

6.0 Implementing Injection Objects *16*

6.1 Managing resources **16**

6.2 Injecting into the test class **19**

7.0 Properties *20*

7.1 The structure of properties **21**

7.2 Accessing the configuration property store **23**

7.3 Defining a custom property **24**

7.4 Accessing Credentials **25**

8.0 Creating an SPI *26*

9.0 Resource Management *27*

Implementing Resource Management **27**

10.0 Deploying and Using Your Manager *29*

10.1 Scenario 1 – Application manager **29**

10.2 Scenario 2 – External Managers **30**

1.0 Introduction

The guide is intended to be a getting started with concepts for creating Galasa Managers introducing things like the lifecycle rather than an explicit how to guide to code a manager. This does not look to cover a comprehensive use and design of managers, but rather to give a starting point to most topics.

This document should provide context for laying out the project structure to a manager and explain some best practise and conventions for writing your own. However, you will quickly realise that the design of a manager is not a uniform layout and one that can vary quite dramatically based on the purpose and use case. Some examples of manager variation:

- Standard API Managers - A basic API object provided through annotation injection inside a test class. These are some of the most common managers and can be used for many purposes from repeating common tasks like 3270 logins, or provision testing resources for a particular test (e.g. a docker container).
- Service Provider Managers – Some managers are not explicitly called, but rather provide services like provisioning. Take the Openstack manager for example, this manager provides itself as a “Linux Provisioner”, which a tester or another manager can use by asking for the generic service, rather than a specific implementation.
- Interfacing Managers – A manager might not provide any implementation at all, but rather an interface for multiple implementations to use. A good example of this is a lot of the z/OS managers which can provide features like console commands or file handling through ZOSMF, RSE, OECONSUL, etc.
- Background Managers – A few managers can be used without any reference inside a test class, but at runtime. An example of this is the VTP recording manager.

This education looks to only cover some of the basics and will mostly concentrate on Standard API Managers. The Docker Manager is commonly used in the example as it is a good example of a simple API manager that covers a lot of the mechanisms.

2.0 Manager Lifecycle

Before looking at any code or how to implement a manager it is important to first understand the lifecycle of a manager. The Galasa framework is designed in such a way to provide hooks to enable functionality at many phases/stages of running a test class. This falls through into the architecture of managers which is provided the access to these hooks.

This manager lifecycle is part of the wider test execution lifecycle, fits withing this order of events:

1. A Galasa test is invoked by the CLI, Eclipse Plugin, Java Command, etc
2. Launches an OSGI Runtime (in this case Apache Felix).
3. This OSGI runtime installs all the Galasa framework, Galasa Managers, any extensions and the test code being executed (as OSGI bundles)
4. The Galasa Framework then starts executing, which starts the manager lifecycle. In this phase the framework provides the context to which managers are required for this test. This is done by offering each manager to observe all annotations within a class to look for related managers (this is covered more later). All relevant mangers perform an initialisation phase as they are required.
5. All initialised managers then continue with their lifecycle as normal.
6. The test case is then executed as expected.
7. Once the test case has been executed the managers which are active finish their lifecycle to provide resource clean-up. This ensures that Galasa leaves resources ready for other tests (but there is other protection built into managers in case a test case is ended unexpectedly).

2.1 Manager Lifecycle Phases

In the test lifecycle, you can see managers have an opportunity to initialise if they are required. If they are the follow phases exist within a managers life:

1. Initialise
2. Provision Generate
3. Provision Build
4. Provision Start
5. Start of test Class
6. Start of test method
7. End of test method
8. End of test class
9. Provision Stop
10. Provision Discard
11. Perform Failure Analysis
12. End of test run

As you may expect looking at this list, some phases like “Start of test method” are run more than once.
You may also notice that there are multiple “Provision” phases in the lifecycle. This breakdown into three different phases provides the opportunity to perform some phase of work in the correct order.

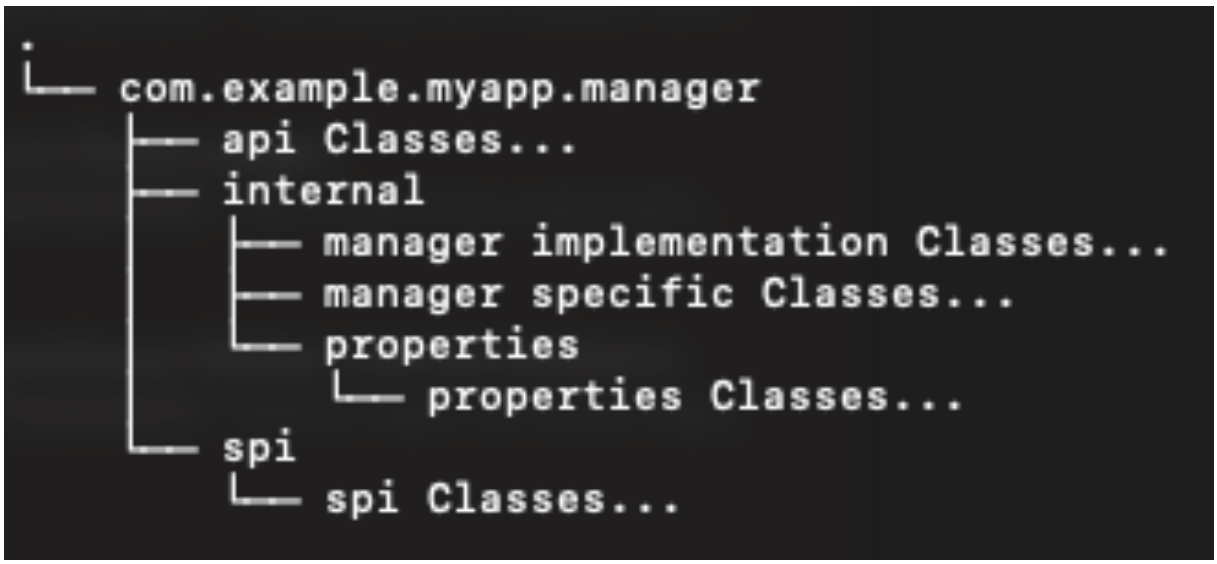
This is because managers can have dependencies on other managers (we cover this later), which can impact which manager needs to be initialised and provisioned first. There are also times when you want to split “provisioning” work into distinct phases.

For example, we may want to run configuration changes (potentially with information provisioned from other managers) during “Provision Generate”, a custom build of an application during “Provision Build” and then the launching of an application during “Provision Start”.

These steps are generic in nature as each manager performs, runs and builds its own resources in different ways. It should be noted not all these phases need to be implemented, only the relevant ones for that manager.

3.0 Simple Manager Code Structure

When developing a manager there is a standard structure which is best practice:



The API classes mostly consist of Interfaces the tester will interreact with, Java annotation interfaces and customer exception classes.

Inside the internal package are all the classes we don't want to expose through the tester programming interfaces (TPI). This covers a lot of classes, from one to manage the Galasa hooks, for SPI implementations, TPI implementations and actual resource interactions.

It is normal practise for the main Galasa lifecycle controller class to be named *X*ManagerImpl.java where X is the name of the manager. The other classes inside here are implementations of the API classes, plus any additionally required code.

The SPI or systems programming interface is the home for interfaces which manager to manager interactions happen.

4.0 Interfacing with the Lifecycle

Like a number of Galasa components, managers utilise the OSGI framework and are defined within the runtime as service components. More specifically this is declared as a Component annotation which resolves the IManager.class.

The Galasa framework then also offers an AbstractManager class in which to extend to make development easier. Setting up a manager class would look something like this:

```
@Component(service = { IManager.class })
public class MyAppManagerImpl extends AbstractManager implements IMyAppManager{
    private static final Log logger = LogFactory.getLog(MyAppManagerField.class);

    public static final String NAMESPACE = "myapp";
    private IConfigurationPropertyStoreService cps;
    private IDynamicStatusStoreService dss;
```

> **Note**: *In this example there is a IMyAppManager interface also created, but this is not required for all managers. Typically, ManagerImpl Classes rarely have this. There is also reference to a MyAppManagerField.Class object which I will come back to when we discuss annotations.*

We also then declare a “NAMESPACE” for the manager. This is used for property management, as all CPS and DSS properties that belong to this manager should be prefixed with this value. You will notice this in any examples like “zos.XXX” or “docker.XXX” which are owned by the z/OS and Docker managers respectively.

> **Best Practise***: Any manager should only ever read its own properties in both the DSS and CPS. Any manager-to-manager integrations should be controlled through the SPI. This ensures safe management of resources throughout the ecosystem.*

4.1 Initialising a Manager

Initialising a manager refers to self-registering as required for this test and to progress through the rest of the manager lifecycle within the Galasa framework. This happens at the starting up of a test run each and is offered the chance to mark itself as required through the initialisation method.

Every manager created MUST override the initialise method to be able to register itself to the framework. The AbractManager super class also needs to be initialised with the test code and the framework to provide certain helper methods which we will explore shortly.

```
@Override
public void initialise(@NotNull IFramework framework, @NotNull List<IManager> allManagers,
        @NotNull List<IManager> activeManagers, @NotNull GalasaTest galasaTest) throws ManagerException {
    super.initialise(framework, allManagers, activeManagers, galasaTest);

    /**
     * Because galasa has support for the gherkin language, we can peform differnt actions per
     * language
     */
    if(galasaTest.isJava()) {
        List<AnnotatedField> ourFields = findAnnotatedFields(MyAppManagerField.class);
        if (!ourFields.isEmpty()) {
            youAreRequired(allManagers, activeManagers);
        }
    } else {
        throw new MyAppException("Test type provide not currently supported by this manager");
    }

    /**
     * My App manager specific properties for CPS and DSS
     */
    try {
        this.cps = framework.getConfigurationPropertyService(NAMESPACE);
        this.dss = framework.getDynamicStatusStoreService(NAMESPACE);
        MyAppPropertiesSingleton.setCps(cps);
    } catch (Exception e) {
        throw new MyAppException("Unable to request framework services", e);
    }
}
```

The next step in this initialisation step is to validate we are working with an expected test case type. For now we are checking that the test class is Java. Galasa has been setup for multiple language support (Java and Gherkin currently), so this is an important step to complete to ensure your manager responds appropriately.

Once we are working with the expected type of test, we can search for any annotations within the test class that THIS manager owns. This is done by search in the ManagerField Annotation which is a grouping mechanism used to group all relevant annotations within a single manager. This is covered in more detail later when describing how to define a Galasa annotation. The super class offers a utility method to help with this find Galasa annotations called findAnnotatedFields.

If we find an annotation in the test class which is owned by this manager, then we call the youAreRequired method. Currently you are required to override and implement this method yourself (but I think this should be added to the super class):

```
@Override
public void youAreRequired(@NotNull List<IManager> allManagers, @NotNull List<IManager> activeManagers)
    throws ManagerException {

    // Check to the another manager hasnt already added this manager into the required managers list.
    if (activeManagers.contains(this)) {
        return;
    }
    activeManagers.add(this);
```

You can see in this method we check to see if the manager is already in this list. This is because another manager may have already added this manager to the active managers as it may have a dependency on it.

The initialise phase can be used for initialising any other parameters but is generally limited to not performing any additional provisioning work. In this example we finish off by requesting an instance for a CPS and DSS from the framework. You can see we pass the namespace to limit access and writing of properties.

> **Note:** *A manager that does not provide a TPI can use CPS properties during initialise to understand if it is needed, the VTP manager is a good example of this.*

4.2 Using Dependant Managers

As has been mentioned in previous sections, Managers and interact with other Managers (and should!), to prevent re-implementation of any common tasks. But this needs to be considered when coding and activating a manager.

In this example I am going to add a dependency on the docker manager, starting with declaring an SPI object to interface with the docker manager. Using this structure allows the developer of a manager to provide or limit functionality of a manager compared to the TPI.

```
// Dependancy on the docker manager
private IDockerManagerSpi dockerManager;
```

> **Note:** *The SPI and TPI have different interfaces to allow for differing accesses and controls for manager-to-manager interaction or test to manager interaction respectively.*

We can then initialise this object within the youAreRequired method. This again uses a utility method call addDependantManager, which will locate the correct manager on your behalf.

```
@Override
public void youAreRequired(@NotNull List<IManager> allManagers, @NotNull List<IManager> activeManagers)
    throws ManagerException {

    // Check to the another manager hasnt already added this manager into the required managers list.
    if (activeManagers.contains(this)) {
        return;
    }
    activeManagers.add(this);

    // We then add out dependancy to this method, stating we have a dependancy on this manager also being active
    dockerManager = this.addDependentManager(allManagers, activeManagers, IDockerManagerSpi.class);
    if (dockerManager == null) {
        throw new MyAppException("The docker manager is not available, unable to initialise MyAppManager");
    }
}
```

Note: Both the list of provisioned managers and the provision dependent list of managers are listed in the run.log

---

There is another aspect we must be worried about, and that is provisional dependency. For example, if my application wants to provision “myapp” within a docker container during the “Provision Generate” phase, we need to ensure that the docker manager has performed it’s own “Provision Generate” first!

To do this we can override the areYouProvisionalDependantOn method to state any dependencies. For context the framework will call each manager’s areYouProvisionalDependantOn to create an ordered list of managers which it needs to work with. This will allow for any dependant managers to “Provision Generate” in the correct order:

```
@Override
public boolean areYouProvisionalDependentOn(@NotNull IManager otherManager) {
    if(otherManager instanceof IDockerManager) {
        return true;
    }
    return super.areYouProvisionalDependentOn(otherManager);
}
```

Best Practise: *Even though possible, a manager should NOT use other managers annotations for anything, if a manager needs information from another manager then it should be through the SPI for that manager*

4.3 Hooking into the lifecycle

Once a manager has successful initialised and has been correctly ordered for the provisioning phases, it can simply interface with these phases by overriding the correct method from the AbstractManager class. The full class can be found here

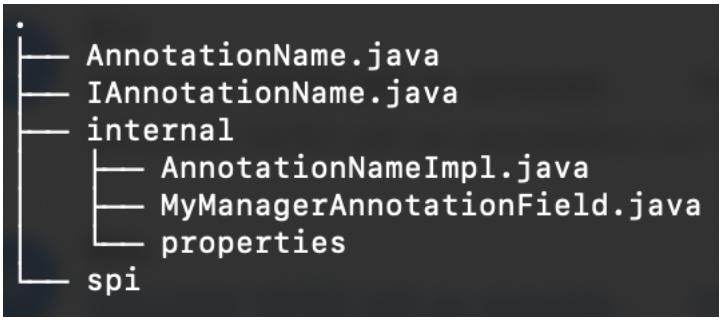
5.0 Annotations

The Galasa annotations are the testers programming interface (TPI) to interact with Galasa managers. Taking a more detailed look at an example:

```
@DockerContainer(image = "my-application:latest",
dockerContainerTag = "a", start = false)
public IDockerContainer container;
```

1. In the green is the annotation itself which hooks into a manager to provide a resource we can interact with.
2. In the red we have the metadata fields that can be added to the annotation.
3. In the blue we have the object type which is injected at runtime by Galasa that the tester will use.

To start creating one of these annotations there are several interfaces and classes that need to be created in the following locations:



5.1 MyManagerAnnotationField – The Manager Annotation

Let's take a look at this interface first, as it may not be obvious of the role of this annotation, but it is very important to ensure that all your TPI annotations are picked up by Galasa. Using the Docker manager as an example:

```
/**
 * Annotation to capture the other docker realted annotations from the test class
 *
 * @author James Davies
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE })
public @interface DockerManagerField {

}
```

This annotation interface is completely empty except for two other annotations:

1. @Retention – The retention policy states when the metadata from this annotation should be made available. In this case the Galasa framework uses all the annotations at RUNTIME.
2. @Target – This annotation can only be placed on other annotations and describes the type of annotation is being annotated. In this case an interface which comes under the ElementType of TYPE.

But what is this annotation being used for? It may sound a little strange, but we use this annotation to annotate our annotations within the TPI. It is done this way to group annotations together under a single manager to make initialisation of a manger much simpler. You'll see this in the next interface.

> **Note**: *The ManagerAnnotationField interface is created in the internal package as it is not to be used in any test code.*

5.2 AnnotationName.Java – The annotation interface

This is the actual definition for the annotation that the tester will be interfacing with from their test class. In many cases this annotation can have very little inside of it, but let's use the docker managers @DockerContainer annotation as an example to explore:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD })
@ValidAnnotatedFields({ IDockerContainer.class })
@DockerManagerField
public @interface DockerContainer {

    /**
     * The <code>dockerContainerTag</code> is used to identify the Docker Container to other Managers or Shared Environments.  If a test is using multiple
     * Docker Containers, each separate Docker Container must have a unique tag. If two Docker Containers use the same tag, they will refer to the
     * same Docker Container.
     * @return The tag for this container.
     */
    public String dockerContainerTag() default "PRIMARY";

    /**
     * The <code>image</code> attribute provides the Docker Image that is used to create the Docker Container.  The image name must not
     * include the Docker Registry as this is provided in the CPS.   If using a public official image from DockerHub,  then the
     * image name must be prefixed with <code>library/</code>, for example <code>library/httpd:latest</code>, the Docker Manager will
     * not default to the library namespace like the Docker commands do.
     * @return the name of the image.
     */
    public String image();

    /**
     * The <code>start</code> attribute indicates whether the Docker Container should be started automatically. If the
     * test needs to perform some work before the container is started, then <code>start=false</code> should be used, after which
     * <code>IDockerContainer.start()</code> can be called to start the container.
     * @return true if the docker container should be started automatically. false otherwise.
     */
    public boolean start() default true;

    /**
     * The <code>dockerEngineTag</code> will be used in the future so that a container can be run on a specific Docker Engine type.
     * You would not normally need to provide a Docker Engine tag.
     * @return The docker engine tag associate with this container.
     */
    public String dockerEngineTag() default "PRIMARY";

}
```

This class is also annotated with several annotations:

1. @Retention – The retention policy states when the metadata from this annotation should be made available. In this case the Galasa framework uses all the annotations at RUNTIME.
2. @Target – This annotation can only be placed on other annotations and describes the type of annotation is being annotated. In this case an interface which comes under the ElementType of TYPE.
3. @ValidAnotatedFields – This annotation validates that the object that has been annotated within the test class is the expected class to be reflected.
4. @DockerManagerField – This is the ManagerAnnotationField that we have just defined. This is the grouping mechanism for all the annotations that this manager owns.

Note: *An annotation is usually on a TYPE within the class. However, there are some annotations that are used to annotate the class which do not inject a variable into the test.*

If you remember in the initialisation method, we used a FindAnnotatedField method using the ManagerAnnotationField to locate all the related annotations. This is how the @DockerContainer, @DockerContainerConfig, etc annotations all activates the Docker manager.

The @DockerContainer annotation also holds other metadata which we can define in the interface. The dockerContainerTag for example is a string metadata which tags the container that will be instantiated with a particular tag. As with the dockerContainerTag these values can be defaulted.

We can see an example of this again here:

```
@DockerContainer(image = "my-application:latest",
dockerContainerTag = "a", start = false)
public IDockerContainer container;
```

5.3 IAnnotationName.Java – The Object Interface

The interface for the annotation then needs to describe what will be offered through the TPI. Any tester using this annotation will receive an object of this type and the methods offered through this interface. Here is a small part of the IDockerContainer interface as an example:

```
public interface IDockerContainer {
    /**
     * Fetch the Resource Object representing the Docker Image of this container.
     *
     * @return a {@link IDockerImage} for this container - never null
     */
    public IDockerImage getDockerImage();

    /**
     * Returns a map of all the exposed ports of the container and the real host ports
     * they have been mapped to.   An exposed port can be mapped to more than one host port.
     * The exposed port in the format used by docker, eg if tcp port 80 is exposed by the image, then
     * the port will be mapped to <code>"tcp/80"</code>.  The {@link InetSocketAddress} will contain the
     * ip address of the host.
     *
     * @return a map of the exposed ports,  never null
     * @throws DockerManagerException
     */
    public Map<String, List<InetSocketAddress>> getExposedPorts() throws DockerManagerException;
    /**
     * A convenience method to obtain the first socket of an exposed port.   Normally
     * an exposed port will only have one real socket, so this will usual way of obtaining the
     * socket of an exposed port.
     * @param exposedPort - the name of the exposed port - eg <code>"tcp/80"</code>
     * @return {@link InetSocketAddress} of the first real port, or null if not exposed or not mapped
     */
    public InetSocketAddress getFirstSocketForExposedPort(String exposedPort);
    /**
     * A convenience method to obtain a random socket for an exposed port that has been mapped to
     * more than one host socket.  Similar to {@link #getFirstSocketForExposedPort(String)}.
     *
     * @param exposedPort - the name of the exposed port - eg <code>"tcp/80"</code>
     * @return {@link InetSocketAddress} of a random real port, or null if not exposed or not mapped
     */
    public InetSocketAddress getRandomSocketForExposedPort(String exposedPort);

    /**
     * Start the Docker Container.  The Docker Manager does not validate that the
     * container is down, so if up it will throw an exception with the remote api failure, likely to
     * be NOT MODIFIED.
     *
     * @throws DockerManagerException
     */
    public void start() throws DockerManagerException;

    /**
     * Start the Docker Container with a provided galasa DockerContainerConfig. This will stop and remove
     * any previous containers.
     *
```

At this point, these 3 classes are enough to define an annotation that would resolve to a tester in their Java test class. In Section 6.0 we cover how to now implement this and have the resulting implementation injected into the test class.

6.0 Implementing Injection Objects

As you might expect, we can now implement the interface within the internal package of our manager.

> **Note:** *The implementation has the naming convention to match the interface in the external packaged but placed in the internal package as this should NOT be accessible to the tester.*

At this point the implementation will be specific to the annotation being designed. It is within these classes and any additional classes which we expect the “actual” code to interface with the tool/technology/platform/application/etc.

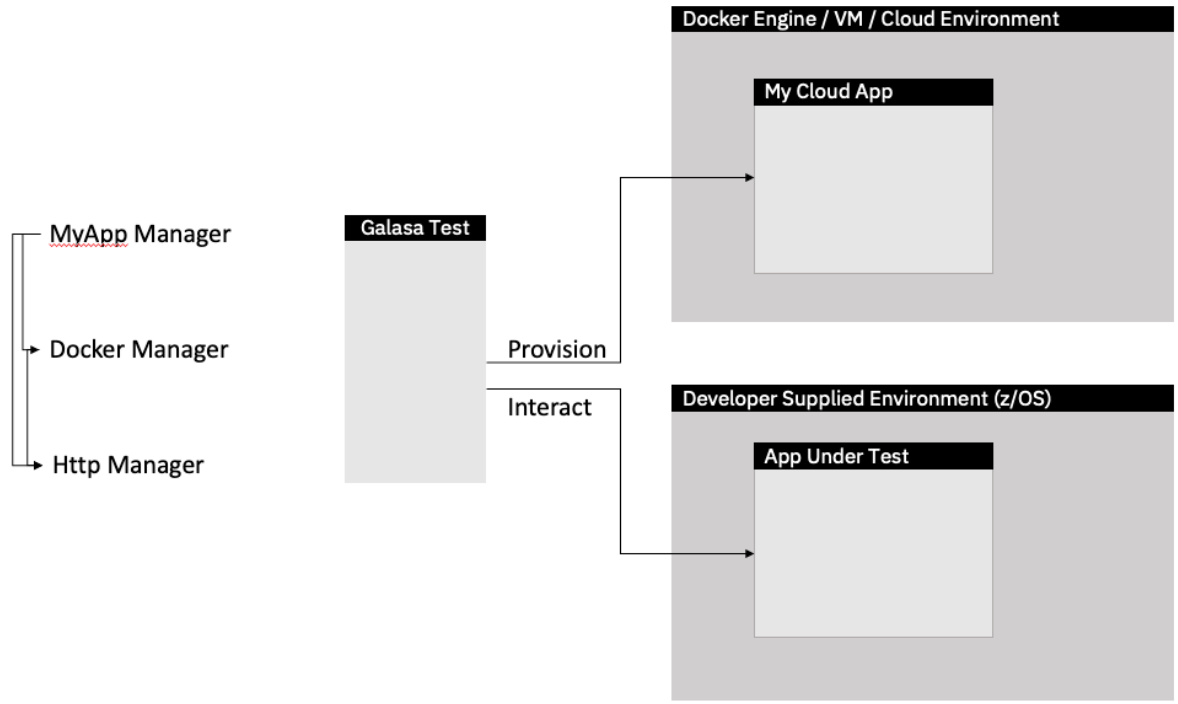
When we are interacting with any technology, there is commonly some resource(s) attached or related too it. We can use the Galasa framework and lifecycle to hook into management of these resources to ensure any environment that is provisioned or created can also be deprovisioned or destroyed. This is vital for maintaining a good health of all available testing resources.

Using the docker manager again as an example, we don’t want tests to be able to setup containers only for them to be orphaned once the test has finished. We also don’t want them to be abandoned in the case of a test case exiting abnormally. Let’s look at managing resources from the Manager lifecycle now, abnormal exits we will come back to in Section 9.

6.1 Managing resources

Managing resources specific to a manager are the responsibility of the author of the manager, but in this section, we will cover what needs to created and configured to allow Galasa to help with resource management.

A good place to start is covering what needs to be covered and what doesn’t, as this is important to ensure no replication or clashing. In this example we are going to pretend that “MyApp” Manager helps a tester provision the cloud portion of my application inside a docker container while we test out z/OS application changes. Our manager also understands how to interact and configure both components of the application. In terms of dependencies between managers to make this happen it might look a little something like this:



The MyApp Manager might use the docker manager to create the container, but then the Http Manager to perform Restful calls to the application. The docker manager also uses the http manager to talk to the docker engine API to create the container.

The docker manager imposes restrictions on the system via CPS properties to limit the number of docker containers that can be brought up on a single docker engine. This means MyApp manager does not have to be actively monitoring containers as a resource as its not within its remit.

However, we may still want to impose our own limits within the MyApp manager. For the sake of this example, let's say we only want 1 My Cloud App container running at any one time.

In this picture "MyApp" manager only needs to manage a "slot" to control the number of My Cloud App containers running at one time. The docker manager will impose any other container limits, and the http manager will control and http clients.

> **Note:** *Remember there could be 1000's of tests running in parallel, so every limit is important. When a restriction is hit, a manager should throw a ResourceUnavailableException so the Galasa Test controller knows to requeue*.*

This makes writing manager much easier, as we only need to be concerned with OUR resources, and not that of any other interlinked technologies might be using.

** When the test is being run as part of an Ecosystem.*

But what is a “slot”?

If you are unfamiliar with the term, a slot is a concept used to throttle or limit something by imposing a requirement on a virtual resource. For example, a docker engine might be capable of running 50 containers at one time, but to ensure that the engine isn’t oversubscribed by other users we might want to limit Galasa to only use 25 at any one time. In the next section we walk about CPS properties and how they are used to dynamically configure something like a slot.

So, to continue with the example we are using, we want “MyApp” manager to limit the whole Galasa Ecosystem of tests running to only have one instance of the container running and bound to one test at a time.

To do this we must interact with the Dynamic Status Store (DSS) to communicate the state of any running tests resources in relation to the entire state of the Galasa Ecosystem. If you look back at Section 4.1 I included a line to instantiate the DSS from the framework given a namespace. This DSS instance can then be used to pass state of this test back to the ecosystem whenever appropiate. This ensures that if we ever provision a resource, we state that within the ecosystem for other tests to check against. This way we can enforce limits on the number of resources in use at the same time.

> **Note**: *Galasa itself uses this slot mechanism to limit the number of parallel runs that can happen in the Ecosystem.*

This guide doesn’t want to dig into implementation to heavily but more express important component, but if you wish to see how the docker manager does this slot management, see here.

A normal flow of events for a manager during “Provision Generate”:

1. Reserve a resource slot within the DSS.
2. Tag the slot in the DSS with a specific run name. (Another DSS property)
3. If reserved successful, provision any resource we are tracking.
4. Let test execution complete.
5. Complete a clean-up of resource.
6. If clean-up was successful release the slot
7. Clean-up DSS properties

> **Best Practise**: *The DSS instance returned from the framework allows for putSwap. These should be used to avoid race conditions when reserving slots.*

6.2 Injecting into the test class

Assuming we have managed to provision our resources within the 3 provisioning phases, we now want to inject the object created back to the test class. This is done with Java reflection, with the AbstractManager class again offering helper methods.

For simple managers that only provision simple resources, the generateAnnotatedFields method from the AbractManager class can be called at provisionGenerate with the ManagerAnnotationField class.

If generateAnnotatedFields is called, then Galasa looks through this package for methods that are annotated like below to call them appropriately.

```
@GenerateAnnotatedField(annotation = MyApp.class)
public IMyApp generateMyApp(Field field, List<Annotation> annotations) throws MyAppException {
    MyApp myAppAnnotation = field.getAnnotation(MyApp.class);

    return new MyAppImpl(myAppAnnotation.tag(), myAppAnnotation.version());
}
```

For more complex cases where a manager might be provisioning multiple objects that are related in some way, the provision generate and other provision phases must be manually used to complete work. The Docker manager is a good example of a more complex case:

```
@Override
public void provisionGenerate() throws ResourceUnavailableException, ManagerException {
    logger.info("Registering Docker registries");
    registerDockerRegistires();
    logger.info("Finding all Docker related annotations");
    generateDockerFields();
}
```

Instead of calling the generateAnnotatedFields method during provision generate to allow Galasa to automatically populate the annotations in the test, the manager manually does these steps to make sure everything is done in the right order. For the Docker manager, we want to make sure the engine can pull from a specific registry before trying to pull a container image. I don't want to explain (as these are implementation specific to docker) the two methods registerDockerRegistires and generateDockerFields, but they can be found here.

In the linked manager above, this demonstrates manually checking the annotation type, fields from the annotation and provisioning resources. You will also notice that they still use the Abstract Manager to register the annotated fields (registerAnnotatedField method). This is to allow Galasa to perform the injection of the objects into the test class. You will notice a method called fillAnnotatedFields where this happens (which is called by the test invocation).

7.0 Properties

In this section I want to cover fundamentals of CPS and DSS properties as well as how to create manager properties.

Before moving into how properties are structured or how to define custom Manager properties, it is important to understand why we use them, the difference between the CPS and the DSS, and how they help Galasa scale.

The ***Configuration Property Store*** (CPS) as you may expect is the place where all the configurations for the Galasa runtimes and Galasa test resources are stored. Everything from hostnames, port number, run name prefixes, etc are defined here. These are all generally quite static or defaulted values.

The ***Dynamic Status Store*** (DSS) is intended to hold the state of the system(s). Every test class and most manager write to the DSS to show information about tests run status, the state of any resources allocated by a test and heart beats. As a user, you are likely to never need to interact with the DSS, but as a developer of Manager you will need to often.

How each of these stores are implemented changes depending on how Galasa is being invoked. When we run a local Galasa test, it is common to interact with local files found in “~/.galasa” directory. However, this causes serval problem when we come to scale Galasa for both the DSS and the CPS.

If we require every user of Galasa to store their own copy of the CPS locally on their workstation there is an issue with keeping the configuration up to date. If any testing resource is taken offline or edited in anyway, all changes would have to be reflected in every CPS file for every user. The DSS also has significantly less value if hosted locally. Ideally, we want every test engine running to use the same store to ensure safe resource allocation and protection. If every user is using their own DSS, there is potential collision issues many resources since none of the engines would be sharing system states.

This is normally the major justification for the use of a Galasa Ecosystem!

When a user is linked up to an Ecosystem, the Galasa runtime is then retrieving all CPS properties and updating state in a DSS that is shared. This way if something is updated in either CPS or DSS, it is reflected in every runtime. At the current version this is implemented for both as a ETCD cluster. Depending on setup these can be the same or separate cluster for both.

Within the Ecosystem, or calls to the Ecosystem allows Galasa tests to safely scale to many runtimes in parallel without fear of test collision and corruption.

It is still offered for any tester or developer to override any set properties using the local file “~/.galasa/overrides.properties” when running in a ecosystem.

7.1 The structure of properties

Many managers provide their own properties to allow them to be appropriately configured. You can find documentation for many of them here. But you may notice that all properties have a similar structure.

For example let's look at how to define a z/OS image for Galasa to connect too. Before we edit the CPS, we must decide a name for the new image, which I will call "IMAGE1". I have used all capitals for this name for a very specific reason. When we are looking at properties in the CPS, they are dot separated values with lower and upper-case components. The upper-case components are used to show which part of the property is dynamic, while the lower case is used to define the static parts. For example:

```
"zos.image.IMAGE1.ipv4.hostname=127.0.0.1"
```

In red shows which manager "owns" this property.
Green shows the components which the manager is expecting to be static.
Blue shows the component that the manager will search from.

This means I could define a separate hostname for a different image by using a new name when defining this property. However, I can then continue to define this image with other relevant properties.

```
"zos.image.IMAGE1.ipv4.hostname=127.0.0.1",
"zos.image.IMAGE1.telnet.port=2023",
"zos.image.IMAGE1.webnet.port=2080",
"zos.image.IMAGE1.telnet.tls=false",
"zos.image.IMAGE1.credentials=CREDS"
```

In this case the manager will be looking for specific properties that have been set, using IMAGE1 as the key. This image is now defined to Galasa, but not yet actually useable in a test yet. This is due to the environment agnostic nature of Galasa.

In a test case, the tester should not be requesting a specific environment, but rather a capability to be able to do something. For example, I might want an environment that has "MyApp" installed. Ideally the test class should have an annotation like:

```
@ZosImage(imageTag="MYAPP")
public IZosImage testImage;
```

Assuming IMAGE1 has "MyApp" installed and is suitable for this test, we need to offer this image as capable. This can be done in one of two ways, as a DSE (Developer Supplied Environment) or as a cluster of tagged images.

Looking at the DSE, this is done with another property:

```
“zos.dse.tag.MYAPP.imageid=IMAGE1”
```

This is very commonly provided with the “~/.galasa/overrides.properties” to link directly to a specific instance. This makes it very easy to link directly to a specific instance, rather than a set of available images.

In automation we tend not to provide a specific image, but rather a cluster of images (as this is meant to be scalable):

```
“zos.tag.MYAPP.clusterid=MYAPPIMAGES”
```

We can then define this cluster of images that have the “MyApp” capability:

```
“zos.cluster.MYAPPIMAGES.images=IMAGE1,IMAGE2,IMAGE3”
```

This will allow Galasa to link the tag in the test a real set of available images to select an appropriate image to run the tests on.

Setting them up this way allows for people to override or adjust where images are running both locally and in automation. Hopefully this example showed you how we can structure properties within a manager.

> ***Note**: This example is from a CPS property, but the same structure is used for DSS properties. These however are still controlled and set from Managers based of RUNID as the key.*

7.2 Accessing the configuration property store

Accessing both the CPS and the DSS is important within a manager, and can be done through the Galasa framework. If you remember back to Section 4.1 when initialising a manager, we request a DSS and CPS instance with a namespace. The namespace is initial prefix in any property. In the example a moment ago, “zos” is the namespace which relates to the owning manager.

To make property easier to define and access within the Manager code, we actually define a Property singleton that can be used as a OSGi service component:

```
@Component(service = DockerPropertiesSingleton.class, immediate = true)
public class DockerPropertiesSingleton {

    private static DockerPropertiesSingleton singletonInstance;
    private static void setInstance(DockerPropertiesSingleton instance){
        singletonInstance = instance;
    }

    private IConfigurationPropertyStoreService cps;

    @Activate
    public void activate() {
        setInstance(this);
    }

    @Deactivate
    public void deacivate() {
        setInstance(null);
    }

    public static IConfigurationPropertyStoreService cps() throws DockerManagerException {
        if (singletonInstance != null) {
            return singletonInstance.cps;
        }

        throw new DockerManagerException("Attempt to access manager CPS before it has been initialised");
    }

    public static void setCps(IConfigurationPropertyStoreService cps) throws DockerManagerException {
        if (singletonInstance != null) {
            singletonInstance.cps = cps;
            return;
        }

        throw new DockerManagerException("Attempt to set manager CPS before instance created");
    }
}
```

Creating a service singleton makes it easier to write and access custom properties within the Manager without the need of passing around a CPS object. Section 7.3 will demonstrate creating a custom property and how to use it.

7.3 Defining a custom property

Creating a custom property is a simple class that extends the CpsProperties class and implementing a single static “get” method. The CpsProperties class provides nice utility functions to make defaulting and response handling easy. Looking at an example for the Docker manager for setting the number of containers that can be run on a single engine:

```
public class DockerSlots extends CpsProperties {
    public static String get(DockerEngineImpl dockerEngine) throws DockerManagerException {
        try {
            String maxSlots = getStringNulled(DockerPropertiesSingleton.cps(), "engine", "max.slots", dockerEngine.getEngineId());

            if (maxSlots == null) {
                throw new DockerManagerException("Value for Docker Engine max slots not configured for the docker engine: "  + dockerEngine);
            }
            return maxSlots;
        } catch (ConfigurationPropertyStoreException e) {
            throw new DockerManagerException("Problem asking the CPS for the max slots for the docker engine: "  + dockerEngine, e);
        }
    }

}
```

Here we can see that the property takes in an engine object as each engine will have its own limits. Pulling the id from the engine, the getStringNulled method searches for properties called:

> “`docker.engine.<engineID>.max.slots`” – 1st choice
> “`docker.engine.max.slots`” – 2nd choice

The CpsProperties class searches for multiple options (provided infixes are provided) to allow for clever interactions. For example, “`docker.engine.max.slots`” can be set for defaulting all other engines, while “`docker.engine.<engineID>.max.slots`” for a specific objects.

Any class within the manager then access the CPS property with a simple call:

> “`DockerSlots.get(engine)`”

Notice that no CPS object required to pull the data.

> Note: If we had passed more infixes to the getStringNulled method, more keys would have been checked. AKA:
>
> `getStringNulled(Singleton.cps(), “example”, “property”, “one”, ”two”, ”three”)`
>
> Would search for:
> “namespace.example.one.two.three.property”
> “namespace.example.one.two.property”
> “namespace.example.one.property”
> “namespace.example.property”

7.4 Accessing Credentials

If you are familiar with the components of an ecosystem you might be aware that the “Credential Store” can be its own service, or an additional namespace within an etcd cluster. But like the CPS and DSS, the CREDS is another service provided from the framework:

```
this.credService = framework.getCredentialsService();
```

This service can then be used to access credential objects from whatever underlying implementation has been used. It is important to use this service to ensure that no secrets get added to logging and outputs.

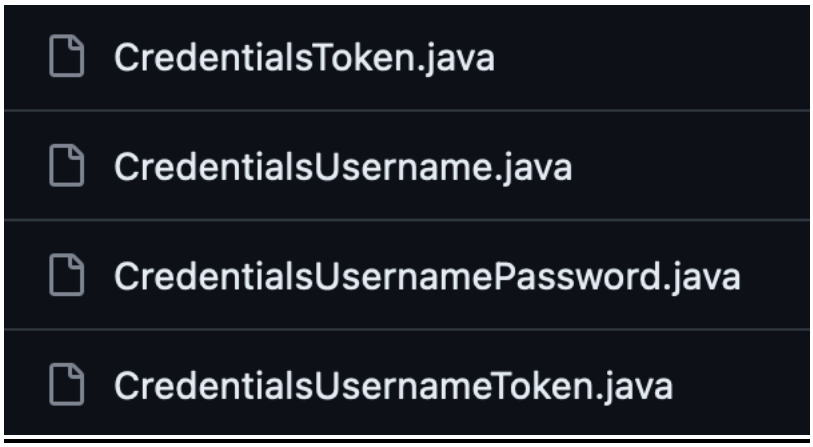
To retrieve any credentials, we can then request from the credsService:

```
credService.getCredentials(credKey);
```

The credsKeys is itself a CPS property to provide a reference to the credentials. This means we can configure tests to use certain systems and credentials very easily.

However, the returned object from the credService is of type `ICredentials.` This is intended to provide all required components of any credentials (aka username password, username token, just token, etc). This means you will have to either directly cast to the correct Credential type or use “instanceOf” to determine the correct behaviour.

At time of writing there are 4 supported credential types:

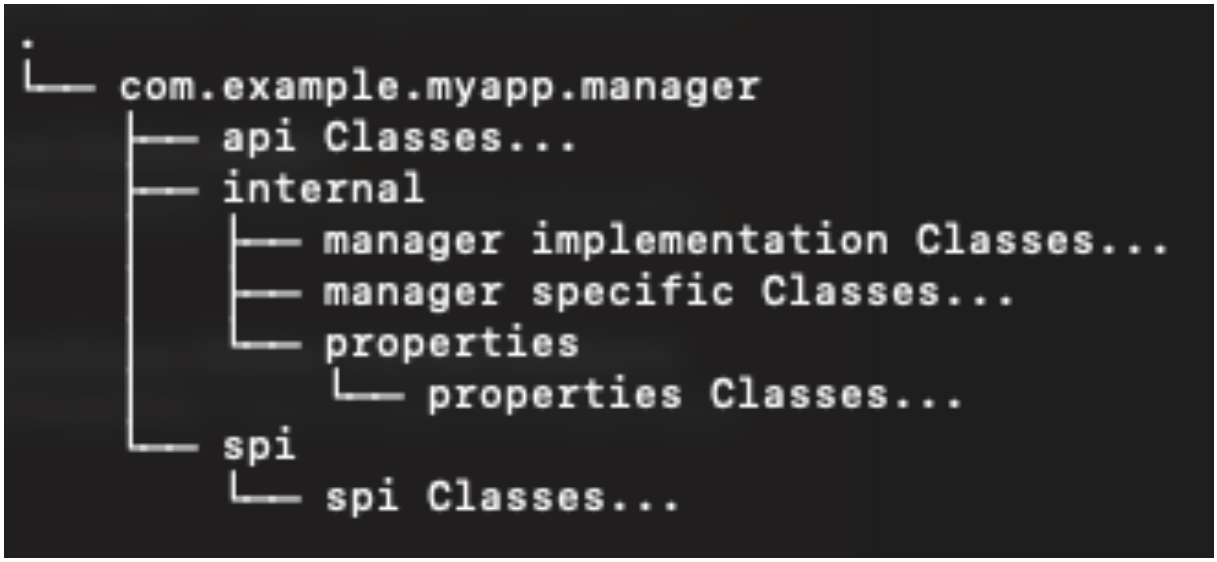


8.0 Creating an SPI

To avoid any duplication of code between managers, managers should expose a Systems Programming Interface (SPI) to allow other managers to utilise any functionality. For example, the “MyApp” manager does not want to implement its own REST calls to the docker engine to bring up an image. Instead, it should ask the Docker Manager to create an image based of information like image name.

The advantage of interacting through other Manager SPI’s is not only just to re-use functionality already created but the other manager, but also to off load resources management to the other manager as well. So, if any test with MyApp ended abnormally, it would be the docker manager that would “clean” up the stranded container of MyApp. The MyApp manager would only be looking to clean up a “slot” one the docker manager had cleaned up the container.

To create the interface that other managers will used, we typically place this in a directory called “spi”:



The actual SPI implementation still resides in the internal package.

9.0 Resource Management

Resource management is one of the big features within Galasa as once we start automating things, we automatically break things too. This can be anything from outages, to abnormal ends of tests but breakages like these end the manager lifecycle early and leave messy and dirty resources lying around.

Within the Galasa Ecosystem is another microservice called the Resmon (short for resource monitoring) whose only job is to maintain the health of test resources.

This resource protection and clean-up is implemented directly into a manager, as every manager is responsible for its own properties and resources.

Implementing Resource Management

Like other components in Galasa, a manager can provide a OSGi service component to the Resmon, by annotating a class as a IResourceManagementProvider. This works as the Resmon microservice is a Galasa framework running in a different operational mode, which loads all the managers up and searched of components that offer resource management.

Since manager may be managing multiple resources, we normally create a class to handle all the resource monitors. Looking at the Docker manager for example:

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

    private IFramework                          framework;
    private IResourceManagement                 resourceManagement;
    private IDynamicStatusStoreService          dss;
    private IConfigurationPropertyStoreService  cps;

    private DockerSlotResourceMonitor           slotResourceMonitor;
    private DockerContainerResourceMonitor      containerResourceMonitor;
    private DockerVolumeResourceMonitor         volumeResourceMonitor;

    /**
     * Initialses the resource management of the docker slots.
     *
     * @param framework
     * @param resourceManagement
     * @throws ResourceManagement
     */
    @Override
    public boolean initialise(IFramework framework, IResourceManagement resourceManagement)
            throws ResourceManagerException {
        this.framework              = framework;
        this.resourceManagement     = resourceManagement;

        try {
            this.cps = framework.getConfigurationPropertyService("docker");
            this.dss = framework.getDynamicStatusStoreService("docker");
        } catch(DynamicStatusStoreException e) {
            throw new ResourceManagerException("Could not initialise Docker resource monitor, due to the CPS:  ", e);
        } catch(ConfigurationPropertyStoreException e) {
            throw new ResourceManagerException("Could not initialise Docker resource monitor, due to the DSS:  ", e);
        }
        slotResourceMonitor = new DockerSlotResourceMonitor(framework, resourceManagement, dss, this, cps);
        containerResourceMonitor = new DockerContainerResourceMonitor(framework, resourceManagement, cps, dss);
        volumeResourceMonitor = new DockerVolumeResourceMonitor(framework, resourceManagement, dss, this, cps);
        return true;

    }
```

Note: *It is important to remember however that this component does NOT go through the standard Manager lifecycle so must manage its own DSS and CPS interactions.*

In this example we have 3 resources to be monitoring and maintaining:

1. Docker containers
2. Docker Volumes
3. Galasa Docker slots

Once the monitor is initialised it will call the start method for normal operation:

```
@Override
public void start() {
    this.resourceManagement.
    getScheduledExecutorService().
    scheduleWithFixedDelay(slotResourceMonitor, this.framework.getRandom().nextInt(20), 20, TimeUnit.SECONDS);

    this.resourceManagement.
    getScheduledExecutorService().
    scheduleWithFixedDelay(containerResourceMonitor, this.framework.getRandom().nextInt(10), 10, TimeUnit.SECONDS);

    this.resourceManagement.
    getScheduledExecutorService().
    scheduleWithFixedDelay(volumeResourceMonitor, this.framework.getRandom().nextInt(10), 10, TimeUnit.SECONDS);
}
```

This provides us the opportunity to customise the level of precision in which we need to performing checks. We use a random number of seconds to perform the first check to avoid huge parallelism with other managers, as the Resmon may be running 100's of resource monitors in parallel.

This class will then cause check and maintenance on a regular cadence to ensure good health in the ecosystem. The classes that implement the monitors will be heavily specific to what task they are performing, but are required to implement "Runable" and override the run method:

```
@Override
public void run() {
    logger.info("Docker container resouce check started");

    updateDockerEngines();

    for (String engine : this.dockerEngines.keySet()) {
        List<String> containers = getOrphanedContainers(engine, this.dockerEngines.get(engine));
        logger.info("Engine " + engine + " has " + containers.size() + " orphaned containers found");
        if (containers.size() > 0) {
            killContainers(containers, this.dockerEngines.get(engine));
        }
    }
    logger.info("Docker container resource check finished");
}
```

10.0 Deploying and Using Your Manager

At this point we have discussed some more of the code design and architecture of a manager, but not how to run a manager. Thank fully this is very simple and the same concepts as running a test.

Since Galasa runs in a OSGi container and all the managers are annotated as OSGi service components, the bundles which contain a manager just need to be installed to be utilised. This opens a large amount of flexibility on how to do this, so let's go through a few scenarios.

10.1 Scenario 1 – Application manager

This is not true in all cases, but in some cases I might want to design a manager that will provide repeated tasks for managing my application. Whether this is logging into the application, handling data provision, etc.

In some circumstances this manager may only affect a very small amount of tests and only a single bundle. So very much like the Simbank examples, we can pair our tests and application manager in the same bundle so could look something like this:

The manager and tests alike would be defined in the OBR project so all the required bundles get installed into the runtime.

10.2 Scenario 2 – External Managers

What I mean by external managers is a set of managers which live outside of the Galasa projects, so are not included in the core runtime.

In this case any new manager I want to add would not be included in the Galasa uber OBR, so will have to be manually passed at runtime. Much like the example above in Scenario one, we would have a project that groups all our managers into a single build with an OBR module included. The generated OBR would then have to be passed to the Galasa runtime at started to ensure that the bundles get added. Any test wanting to include these managers would still have to add typical Maven/Gradle dependencies to include them, as they wont be in the Galasa BOM.

There is however the option to separate the OBR and managers build. Simply the OBR build just requires the Galasa plugins to build a Galasa OBR, and then any maven/gradle dependencies on the managers. The outputted OBR artifact is then what is passed to the Galasa runtime for installation of all required bundles.