Service common architecture

Service common is being restructured to enable services to plug in whatever common components are required at a more granular level. The new architecture has the following benefits:

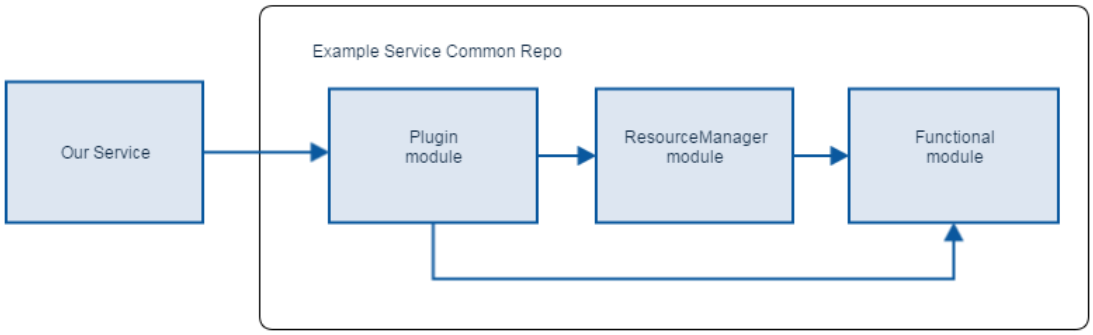
- services are able to plug in only the areas of common required
- services are able to specify which versions of a common area is required on an individual basis
- unnecessary dependencies between service common modules are avoided
- use of plugins means that complicated instantiation code can be removed from the services and placed within the common module
- less test coverage is required within the services themselves
- less code reuse is required within the services themselves
- services become smaller and clearer
- use of 'resource manager' modules within common enforce that common resources are accessed in a standard way

The new service common standard structure

Currently each service common module is held under the parent 'service common' repository. Since the modules are all packaged together at the same level, a service that wants to use the latest version of one module is forced to use this latest version of all modules, which can require unnecessary updates to services and cause dependency tangles.

Service common is being restructured to so that each area of service common functionality is housed in its own repository. A service can then plug in as many or as few areas of service common as needed, at the required version.

Each repository will contain a Plugin module, an optional Resource Manager module, and a Functional module.



Plugin module

The service plugin module will be responsible for constructing necessary items within our service (e.g. utilities defined in the Functional Module) and registering them within our service.

This is currently being done from within our application classes. It's not easily testable and unit tests need to be reproduced across each service to get coverage. Keeping this logic out of our services and the archetype should allow us to focus our testing and keep the code for the service much smaller. It should also greatly reduce the impact of changes to the Functional module.

Resource Manager module

This is an optional module which is responsible for storing and retrieving resources in a static manner, e.g. it could be used to retrieve and store information in a thread-context 'bucket'.

Functional Module

The Functional module holds the functionality to be offered by this area of common.

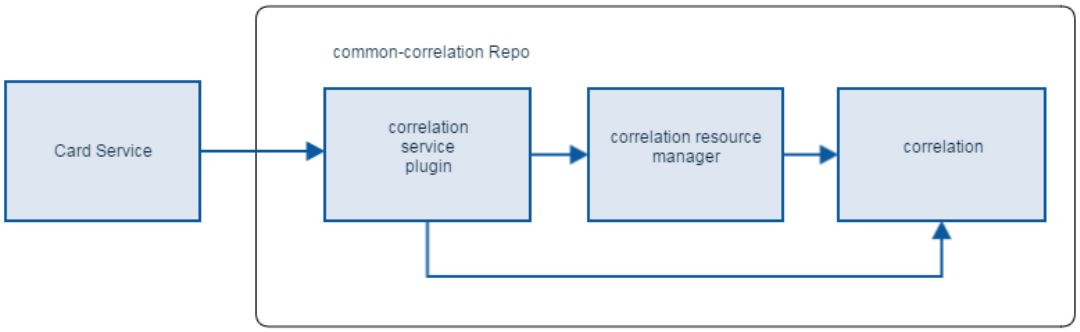
Permitted dependencies within service common repositories

The permitted dependencies between the three modules within a repository are illustrated in the diagram above. To summarise:

- The service plugin can know about the resource manager and the service common module.
- The resource manager can know about the service common module but **not** the service plugin.
- The service common module cannot know about either it's corresponding service plugin or resource manager.

An example: common-correlation

We can look at how the Card Service uses common-correlation as an example of the new structure for a common repo.



The correlation resource manager manages the access to a Mapped Diagnosis Context (MDC) for getting and setting of correlation id. N.B. Other common repos that require a correlation id should use this correlation resource manager rather than accessing the MDC directly; the correlation resource manager can be used by other services separately from the other modules in the common-correlation repo.

There is a requirement for the functional correlation module to provide a filter that extracts the correlation id from inbound HTTP requests and adds it to the MDC. The above dependency restrictions mean that the correlation module that contains the filter cannot know directly about how and where to store the correlation id. For this reason, the implementation code within the functional correlation module makes use of functional interfaces.

The Plugin module does know about the correlation id storage mechanism through its dependency on the correlation resource manager. When the Card Service starts up it uses the Plugin module to instantiate and register the correlation module filter.The plugin module provides the actual implementation code for the functional interface, and passes it to the filter at construction.

The following pseudocode illustrates how a filter might be defined within the correlation module to set a correlation ID on to an MDC from inbound http requests:

```
class InboundCorrelationFilter {

    //correlationSetter is a functional interface also defined in the Correlation
Module
    correlationSetter ;

 //constructor will be passed an implementation of correlationStorer,
    //typically defined as a lambda or method reference when the filter is constructed

 InboundCorrelationFilter( correlationSetter )
 {
     this.correlationSetter  = correlationSetter ;
 }

 //this method will access the MDC at runtime
 void filter(Request request)
 {
   correlationSetter.set( correlationId )
    }

}
```

In the above example, the plugin might construct and register the filter like this:

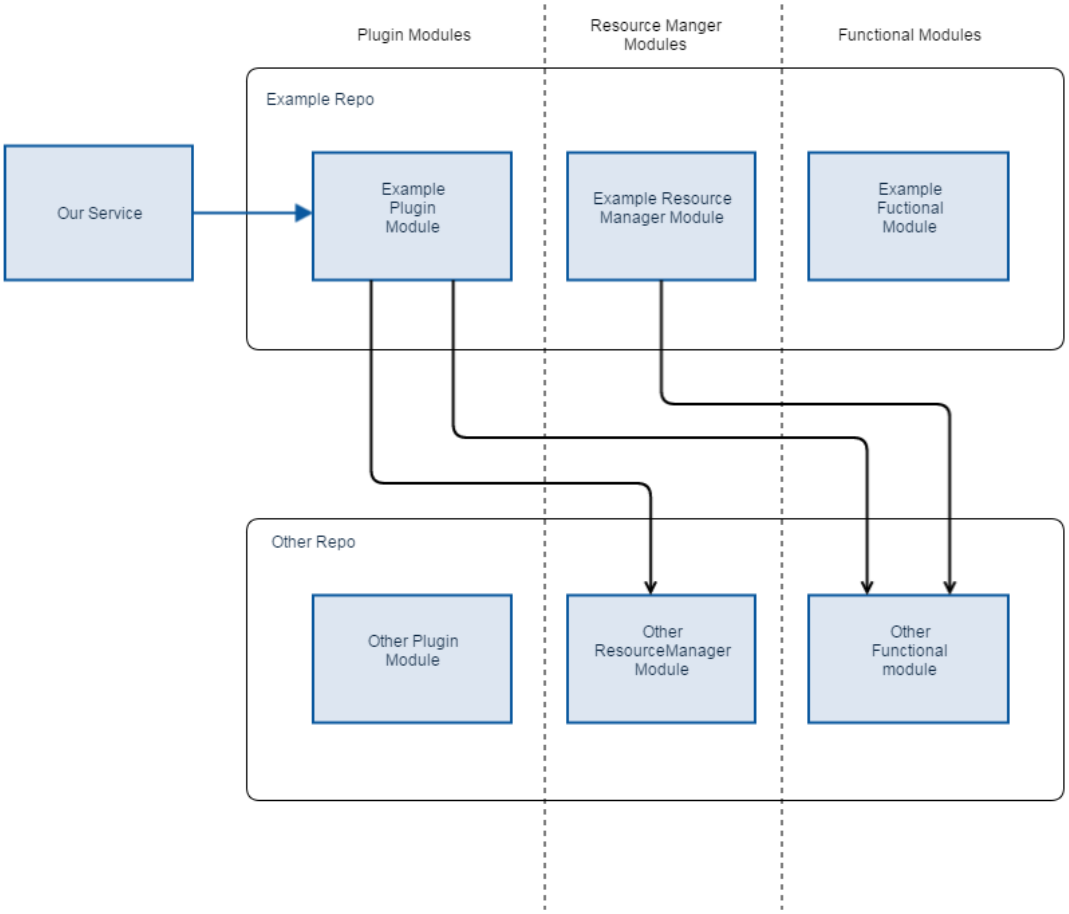
```
environment.jersey().register( new InboundCorrelationFilter(
CorrelationResourceManager::set ) );
```

Permitted dependencies across different service common repositories

Managing module dependencies across separate modules should follow the same pattern as dependencies within a single functional area, but they should not be able to know about modules at the same level.

- The service plugin can know about other resource modules and other functional modules but **not** other service plugins
- The resource manager can only know about functional modules
- The service common functional module cannot know about service plugins, resource managers or other functional modules

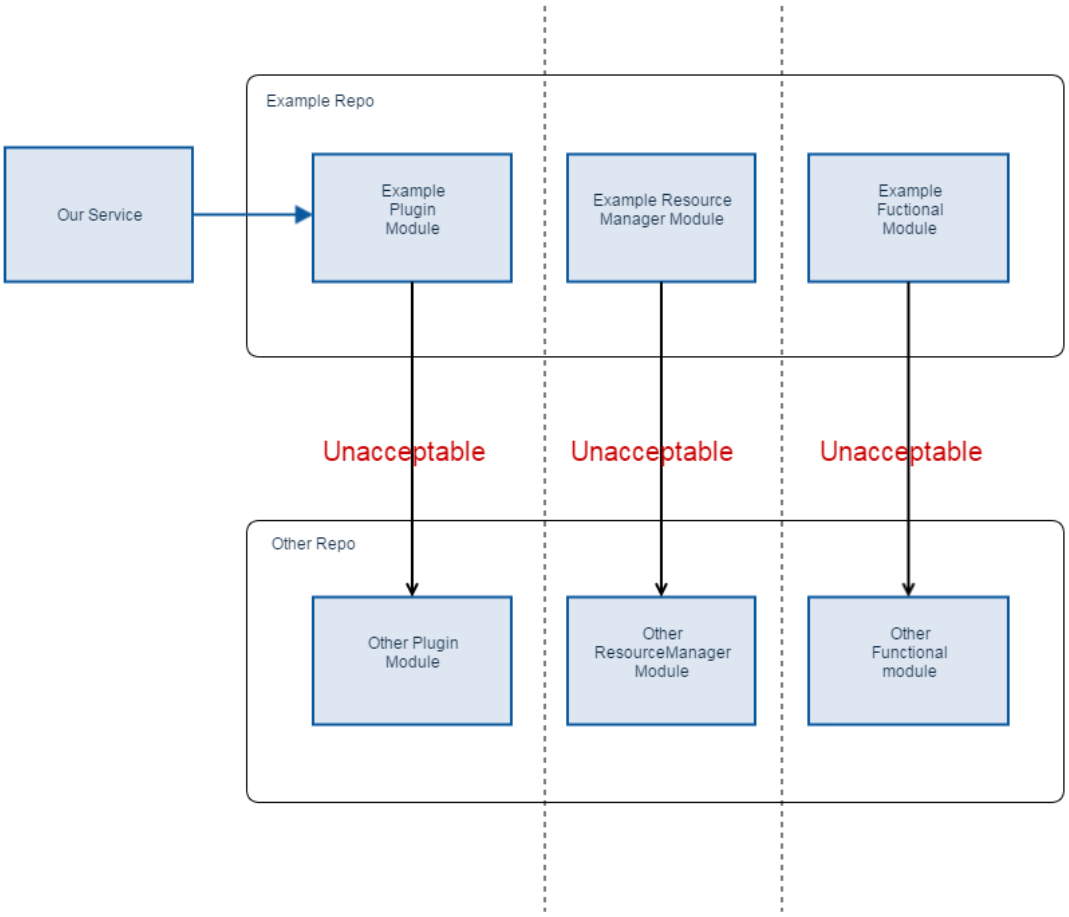
The diagram below illustrates the acceptable dependencies across the module 'swim lanes':



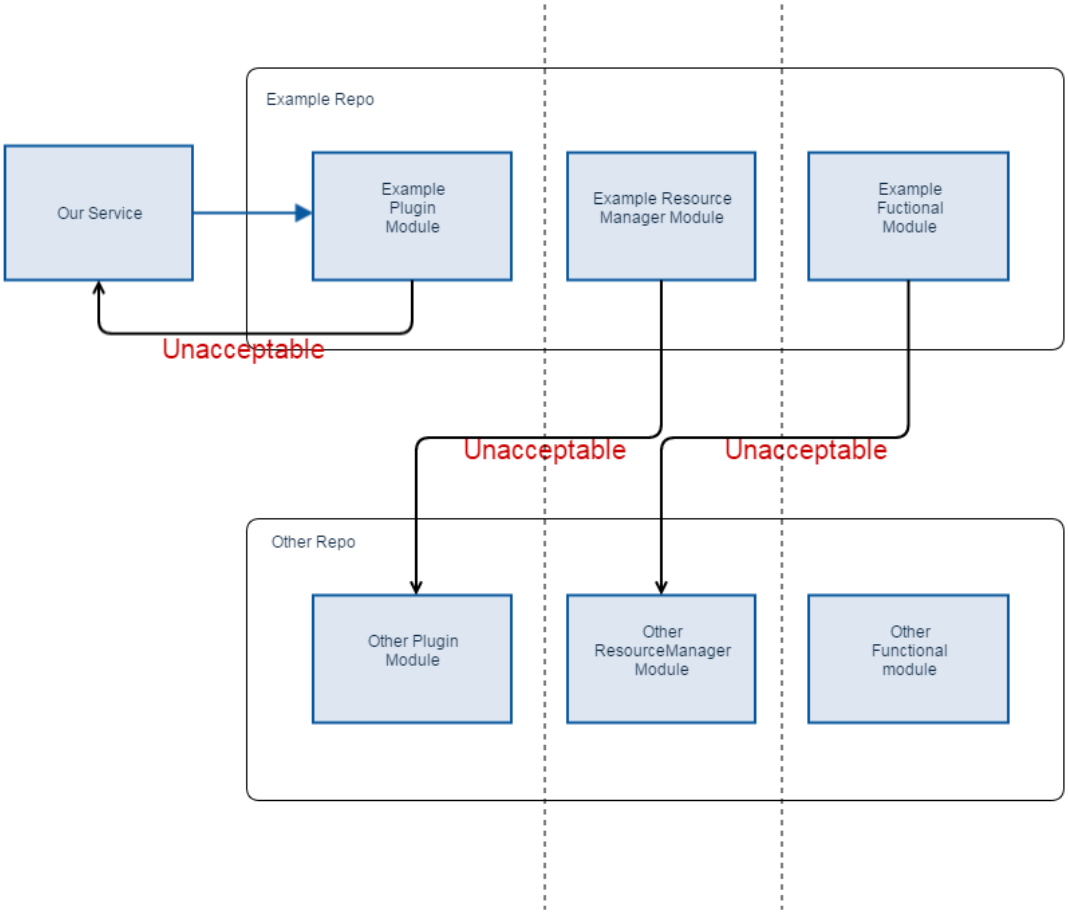
(Note: An example of functional dependencies between repositories can be seen in the interaction between common correlation and common rabbit within the common rabbit documentation).

The following diagram illustrates the forbidden dependencies between modules within the same 'swim lane'. This reinforces the following concepts:

- There should be no reason why a plugin module would require a dependency on another plugin module
- Resources should not be managed by more than one resource manager module (although a plug in module can use multiple resource manager modules where required)
- Functional modules should have no dependencies on other functional modules; their functionality should be kept distinct to what their repo advertises



The following diagram illustrates that dependencies are only permitted in one direction, avoiding circular dependencies and other dependency tangles.


Example Repo
Our Service
Example Plugin Module
Example Resource Manager Module
Example Fuctional Module
Unacceptable
Unacceptable
Unacceptable
Other Repo
Other Plugin Module
Other ResourceManager Module
Other Functional module