MDM5 API Technology Proposal

1. OSGi Runtime

Equinox[1], the Eclipse runtime will be used as the application's core runtime. Following a short description of the Core & Compendium / Enterprise OSGi services that are planned to be used to build a modular, dynamic and distributed application.

1.1. Configuration Admin

The Configuration Admin will be used to deploy service component configurations to boot the application. Furthermore the Configuration Admin can be used to update already deployed service configurations at runtime. Service component configurations can be stored in one or multiple JSON or XML file(s).

1.2. Declarative Services (DS)

Declarative Services is the OSGi way of Dependency Injection. Any deployed service component configuration results in a service component instance. Such a service component instance is then dynamically instantiated and activated as soon as all of its mandatory service component dependencies are satisfied. A service component dependency states, that a service component instance of a certain type must be up and running, before the declaring service component can be activated. This very dynamic and complex part is completely manged by the OSGi Declarative Services. Therefore a service component must statically declare all of it's service component dependencies, regardless of whether they are mandatory or not. In some rare cases it is not possible to statically declare service component dependencies, then a *ServiceTracker* can be used to get notified as soon as the required service is up and running.

Dependency Injection Methods

Care should be taken when implementing OSGi DS related methods ((de-)activate, (un-)bindXX, etc.). The Java keyword *synchronized*, for example, should not be used to avoid increasing startup times while activating service component instances or binding service component dependencies to keep total boot time of the application as short as possible. Long running initialization tasks should be done in a background thread.

[1] https://www.eclipse.org/equinox/

A bound service (mandatory or optional) should always be stored in an *AtomicReference*. If more than one service is bound a 'copy on write' collection like *CopyOnWriteArrayList* should be used to store bound services. *AtomicReference* and *CopyOnWriteArrayList* will both ensure atomic service updates which are required in a dynamic environment where services can come and go at any time.

The Whiteboard Pattern

The Whiteboard Pattern[2] is the OSGi way of a Listener concept. Instead of resolving an event source and registering a Listener to it, the Listener itself is published under its implemented interface(s) in the OSGi service registry. The Event source on the other side declares a service component dependendcy on the listener interface it wants to work with. At runtime the Declarative Services will bind any available service component instance which is published under the listener interface to the event source service.

1.3. Remote Services

The OSGi service registry is used to register services under a subset of the implemented interfaces to be retrieved and used by others, the consumers. A consumer does not necessarily need to be registered as a service in the service registry but may use any of the registered ones.

Service Distribution

Services and their consumers are loosely coupled. Therefore a distribution provider may use this loose coupling to export a service by creating an endpoint for it. On the other side the distribution provider creates a proxy that accesses the endpoint of the exported service and registers it as an imported service in another framework service registry.

If an endpoint becomes unreachable due to network failures, the distribution provider is notified and consequently unregisters affected service proxies from the importing framework service registry. Finally, in the planned environment, Declarative Services comes into play and notifies all affected consumers of the detached remote services.

Furthermore if the service component configuration of an exported service is modified, any consumer, local or remote, will be notified about that change via Declarative Services or an associated *ServiceTracker*, if the service was retrieved that way.

[2] http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf

Service Discovery

In a distributed environment exported services, more specifically their endpoints, have to be discovered and announced to the distribution provider and vice versa. There is a very large number of ways how services, exported by other frameworks, can be discoverd.

Eclipse Communication Framework (ECF)

The Eclipse Communication Framework[3] is a fully-compliant and transport independent implementation of the OSGi Remote Services[4] standard which effectively extends an OSGi Framework with remote capabilities. It comes with various available Distribution and Discovery Providers listed in the table below.

Table 1. Available Providers in Eclipse ECF

Distribution Provider	Discovery Provider
ECF Generic - TCP-based protocol	EDEF[5] - XML file based discovery.
r-OSGi - Another TCP-based protocol.	Apache Zookeeper[6] - A popular server-based service discovery system that's part of the Apache Hadoop project.
Java Messaging Service (JMS)[7] - API for sending messages between clients.	Zeroconf[8] - A multicast-based protocol originally created by Apple for printer and other device discovery.
REST[9] - Access services using HTTP via GET, PUT, POST and DELETE actions.	Service-Location Protocol (SLP)[10] - An IETF service discovery standard.
XMPP - An open standard protocol allows exchanging messages and presence informations in real-time.	DNS-Service Discovery (DNS-SD)[11] - A discovery protocol based upon dynamic DNS.

[3] https://eclipse.org/ecf/
[4] https://osgi.org/download/r5/osgi.cmpn-5.0.0.pdf
[5] http://wiki.eclipse.org/File-based_Discovery_with_the_Endpoint_Description_Extender_Format
[6] http://zookeeper.apache.org/
[7] https://en.wikipedia.org/wiki/Java_Message_Service
[8] http://en.wikipedia.org/wiki/DNS-SD#Apple.27s_multicast_DNS.2FDNS-SD
[9] https://wiki.eclipse.org/Tutorial:_Creating_a_RESTful_Remote_Service_Provider
[10] http://en.wikipedia.org/wiki/Service_Location_Protocol
[11] http://en.wikipedia.org/wiki/DNS-SD#DNS-based_service_discovery

Distribution Provider	Discovery Provider
JGroups[12] - Allows to adopt the JGroup protocol to an ECF container	

An illustration of Eclipse ECF's architecture is shown below. One can use any of the available providers or implement the ECF Remote Service API for a custom distribution provider or the discovery provider API for a custom distribution provider. The OSGi Remote Service specification makes it very simple to distribute a service across multiple OSGi frameworks by simply adding some specific properties to the deployed service component configuration. A Framework can contain multiple distribution providers, each independently exporting and importing services.

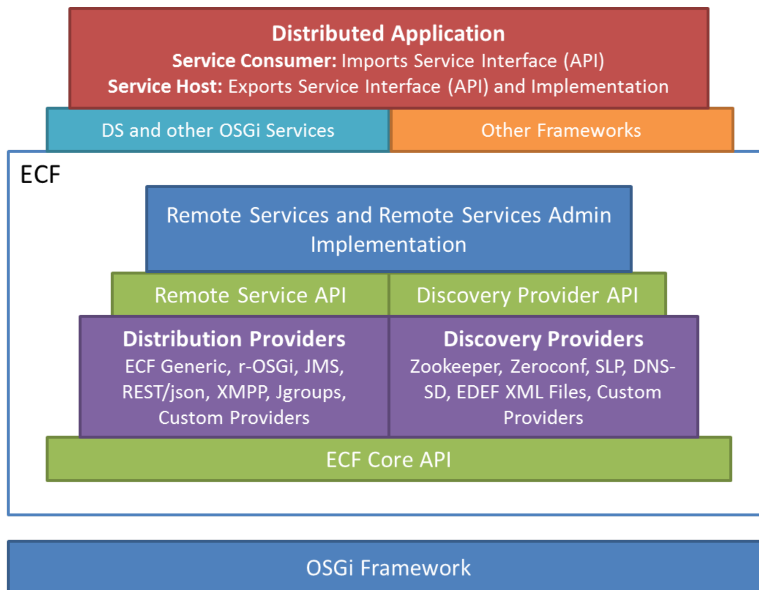

Figure 1. Eclipse ECF Architecture[13]

Example with Zookeeper discovery

The Eclipse ECF distribution comes with many examples which demonstrate some of the available distribution and discovery providers. We will take a deeper look at an example with asynchronous remote service access where the remote service is discovered via Zoodiscovery.

[12] https://wiki.eclipse.org/EIG:JGroups_provider
[13] https://wiki.eclipse.org/OSGi_Remote_Services_and_ECF

This example consists of 3 bundles:

- Service API Bundle exports the service interface and may export an appropriate interface, which remote consumers can use to access the service asynchronously. This bundle has to be loaded in both, host and consumer, frameworks.
- Service host implements the service interface and registers it in the service registry.
- Service consumer wants to uses the service and therefore requests it from the service registry.

That alone is a working example of providing and consuming a service in one framework. To make the service remotely available in other frameworks it is at least required to add some properties to the service host. Then the chosen discovery provider has to be configured. On the consumer side additional code will be required to access the service asynchronously.

Adding properties to the service host

The host publishes its implementation of the service in the service registry. It is not required to change that implementation. Only its service component configuration requires some additional properties:

- **service.exported.interfaces** This property takes a list of interfaces under which this service will be available on the consumer-side. The value "*" stands for all.
- **service.exported.configs** A list of configuration types that will be used to export the service. In this example we will use the ECF Generic container and set this property to: "ecf.generic.server".
- **ecf.exported.containerfactoryargs** This property is an ECF specific one and provides additional container arguments which we set to "ecftcp://localhost:3787/server".
- **ecf.exported.async.objectClass** This property is ECF specific too and specifies the interface for asynchronous access (optional).

The properties listed above can automatically be generated while deploying service component configurations for each service component which is marked as remote available.

Zookeeper Discovery

As described in the previous chapters, with the modified service component host configuration, the OSGi Remote Service will automatically generate an endpoint for it. The next step is to configure the Zookeeper discovery, so that another framework can discover the exported

service and provide it to its registered services for consumption as a proxied local service. In this example Zoodiscovery is configured via virtual machine arguments:

- **zoodiscovery.autoStart** If this property is defined Zoodiscovery will start automatically.
- **zoodiscovery.flavor** This property defines the network nodes providing exported service endpoints. It takes following example values for host and consumer (multiple nodes can be defined as well):
    - Host-side: zoodiscovery.flavor.standalone=localhost:2002;clientPort=2001
    - Consumer-side: zoodiscovery.flavor.standalone=localhost:2001;clientPort=2002

Zoodiscovery supports three different flavors (standalone, centralized and replicated). A standalone flavor provides only its local endpoints to other nodes. A centralized flavor provides the endpoints of all nodes from one central server (single point of failure). The replicated flavor is more stable than the centralized one, because the endpoints are replicated over all nodes. If one node fails, then only its local endpoints will be shut down. In case of network failures Zoodiscovery keeps tyring to reconnect automatically, regardless of the chosen flavor.

Asynchronous service access

A remote service consumer, in most cases, does not care whether any of its bound services is remote or not. Nevertheless it is simple to determine whether the service is local or not. A distribution provider will always add properties to the service component's configuration while importing the service, among them the property **service.imported** will be set. Is this property defined in the service component's configuration, then this service is remote, otherwise not. Sometimes it is required to access a remote service asynchronously. For this purpose an appropriate interface must be available. Consider the following service interface in the API bundle.

Simple service interface

```
public interface AnyService {

    public String doSomeThing(String from);

}
```

To access an implementation of that service asynchronously, ECF expects an interface for asynchronous access. As previously described this interface has to be defined in the service component's host configuration with the property **ecf.exported.async.objectClass**. The distribution provider on the consumer side will provide a proxy that can be cast into the asynchronous interface. How such an interface could look is shown below:

Simple asynchronous service interface

```
// Options 2 and 3 are mutually exclusive - return type is the only difference!
public interface AnyServiceAsync {

    // 1 The IAsyncCallback interface is ECF specific
    public void doSomeThingAsync(String from, IAsyncCallback<String> callback);

    // 2 Requires polling for non blocking access
    public Future<String> doSomeThingAsync(String from);

    // 3 Since Java 8 - more advanced asynchronous access
    public CompletableFuture<String> doSomeThingAsync(String from);

}
```

As one can see the *AnyServiceAsync* interface defines 3 different options accessing the same method *doSomeThing* of the service interface *AnyService*:

1. The *IAsyncCallback* interface causes a dependency on ECF specific bundle(s).
2. The *Future* is immediately returned but requires some kind of polling to determine whether the request is finished or not.
3. Java 8 introduced the *CompletableFuture*. One can define a *Runnable* which is executed as soon as the asynchronous call has finished → polling is not required.

1.4. Event Admin

Services in OSGi can use the Event Admin to fire (a)synchronous events, other services can listen to without any knowledge where those events came from (Whiteboard Pattern). Consumers implement an event handler interface and publish it in the service registry. Events in turn consist of a topic and event properties. The Event Admin uses the topic to deliver the event only to those who are interested in that topic. Eclipse ECF provides a distributed version of the Event Admin. An example how this works is available in the Eclipse Wiki[14].

2. Java 8

Java 8 is the latest stable release which introduced new features like lambda expressions, stream API, *Optional*, *CompletableFuture*, default methods in interfaces and JavaFX which replaces Swing as the new standard for rich client development.

[14] https://wiki.eclipse.org/EIG:Distributed_EventAdmin_Service

2.1. The stream API in conjunction with lambda expressions

Lambda expressions allow us to treat functionality as a method argument. In the following example a list of persons is processed to print given names of persons older than 18:

```
List<Person> persons = new ArrayList<>();
// .. populate list and list given names of persons older than 18

// ..  using stream API
persons.stream().filter(p -> p.getAge() > 18).forEach(p ->
 System.out.println(p.getGivenName());

// .. if the collection is very large process it parallel
persons.stream().parallel().filter(p -> p.getAge() > 18).forEach(p ->
 System.out.println(p.getGivenName());

// .. the classic way
for(Person p : persons) {
    if(p.getAge() > 18) {
        System.out.println(p.getGivenName());
    }
}
```

Working on collections with the stream API in conjunction with lambda expressions is very expressive, and therefore easy to read and maintain.

2.2. *Optional* might help to avoid occurences of *NullPointerException*

The *Optional* is a utility class that wraps any kind of an object. It is an excellent visual feedback to the developer, that the contained object is not always available.

```
List<Person> persons = new ArrayList<>();
// .. populate list and find any person who is 14 years old.

Optional<Person> person14yOldOpt = persons.stream().filter(p -> p.getAge()
 == 14).findAny();

// a person who is 14 years old may not exist
if(person14yOldOpt.isPresent()) {
    Person p = person14yOldOpt.get();
    // do something...
}
```

Working with services in a dynamic environment like OSGi, where services can come and go at any time, might easily result in *NullPointerException*, if a check for null is missed (relevant when services are updated at runtime or disappear due to network failures). Providing them wrapped in an *Optional* might help to avoid *NullPointerException* in advance.

2.3. Functional-style callbacks using *CompletableFuture*

CompletableFuture is useful in various scenarios. One of them is asynchronous access on remote running services. Some examples are shown in the listing below.

```
// .. execute a task as soon as another one is finished
CompletableFuture.runAsync(() -> System.out.println("task 1")).thenRunAsync(() ->
 System.out.println("task 2"));

// .. execut a task when others are finished
List<CompletableFuture<?>> cfs = new ArrayList<>();
// .. populate list
CompletableFuture.allOf(cfs.toArray(new
 CompletableFuture[cfs.size])).thenRunAsync(() -> System.out.println("all tasks
 successfully completed"));
```

2.4. Default methods in Interfaces

Finally it is possible to provide default method implementations in interfaces. Consider following example. The interface *Simple* defines default implemented methods to get and set a name. Both use the abstract method which individually has to be implemented each time this interface is used.

```
public interface Simple {

    Map<String, Value> getValues();

    // it is not possible to mark default methods as final though it is implemented
    default String getName() {
        return getValues().get("Name");
    }

    // any (abstract) class implementing this interface may override
    default void setName(Value value) {
        getValues().put("Name", value);
    }

}
```

With Java 8 it is no longer necessary to implement various interfaces in one abstract base class and let multiple implementations extend that class. This is sometimes done to reduce implementation overhead or enforce identical behaviour across implementations. With default methods in interfaces it is possible to build interfaces on top of each other. This allows to implement more or less complex functionality in one place and share it among different classes (very useful when designing POJOs).

3. Build System

We would prefer to use Maven as the build system for the application. While developing the MDM5 prototype we used the tycho[15] plugin to build OSGi bundles with the Manifest-first approach (dependencies are resolved by looking at corresponding Manifest headers.) and have therefore experience with it. Nevertheless we are open to give Gradle a try.

[15] https://eclipse.org/tycho/