OSGi meets Lambdas

Safe and powerful interaction with OSGi registry

Carlos Sierra Andrés

Liferay, Inc.

October 2017
What are we going to talk about?

We are going to discuss Aries Component DSL, a PoC library for interacting with OSGi registry in a functional way.

https://github.com/apache/aries/tree/trunk/component-dsl
Motivation

When creating JAX-RS reference implementation Whiteboard

- Dependencies filter depend on configuration.
- Dependencies filter depend on incoming services.
- Incoming services can specify any number of dependencies on their properties.
- In OSGi this dependencies can come and go at any time.
- Also dependencies can be *overwritten* with higher ranked services at any time.
bundleContext.registerService(
    MyResource.class,
    new MyResource(),
    new Hashtable<String, Object>(){
        put(
            "osgi.jaxrs.application.select",
            "(name=myapplication)");
        put(
            "osgi.jaxrs.extension.select",
            new String[]{
                "(name=extension1)",
                "(name=extension2)"
            });
    });
@Component(
    immediate=true,
    property={
        "osgi.jaxrs.application.select=(name=myapplication)",
        "osgi.jaxrs.extension.select=(name=extension1)",
        "osgi.jaxrs.extension.select=(name=extension2)",
        "osgi.jaxrs.whiteboard.target=(name=special)"
    }
)

service = MyResource.class

public class MyResource {
    ...
}

Possible implementations

- Component Frameworks:
  - Big runtimes
  - Difficult to express dependencies programatically based on incoming services

- Plain OSGi API:
  - ManagedServiceFactories + nested trackers.
  - Cumbersome to write.
  - Difficult to reason about.
  - Difficult to compose and reuse.
  - Difficult to modify.
bundleContext.registerService(
    ManagedServiceFactory.class,
    new ManagedServiceFactory() {
        public void updated(...) {
            new ServiceTracker<>() {
                public Object addingService() {
                    return new ServiceTracker<>() {...}
                }
                public void removedService() { close }
            }
            public void deleted(String s) { close }
        }
    },
    new Hashtable<>() {{
        put("service.pid", "configuration.pid");
    }}
);
What I would like to have

So I want:

- to declare the services and resources I need from the OSGi registry.
- to use them when they are ready, and only when they are ready.
- the services to clean up their mess when they go away.
- to be able to combine them with existing third party classes.
- to register new services.

I want dynamism but I DON’T want to deal with it!
Functional Programming to the rescue

- **Functors**: describe computations inside a context.
- **Monads**: express dependency between operations.
- **Applicative Functors**: combine functors with existing functions.

OSGi<T>
import static OSGi.*;

OSGi<Foo> service = services(Foo.class);

OSGi<ServiceReference<Foo>> sr =
    serviceReferences(Foo.class)

OSGi<ServiceRegistration<Foo>> registration =
    register(Foo.class, new Foo(), new HashMap<>());

OSGi<Dictionary<String, ?>> c = configurations("myPid");
As a functor

Functors allow to express computations in a context

Optional<Object> opt; opt.map(o -> o.toString());

Stream<Integer> numbers; numbers.map(x -> x * 2);

CompletableFuture<String> future; future.map(s -> s + "; Done!");

OSGi<Property> p = services.map(s -> s.getProperty());
Monads allow to express dependent computations

```java
Map<String, String> bigMap1;
Map<String, String> bigMap2;
Map<String, String> bigMap3;

Optional<String> final = Optional.ofNullable(bigMap.get("optional1"))
    .flatMap(s1 -> Optional.ofNullable(bigMap2.get(s1))
    .flatMap(s2 -> Optional.ofNullable(bigMap3.get(s2)))
);
Monads allow to express dependent computations

```
CompletableFuture<String> heavyTask1() {...};
CompletableFuture<String> heavyTask2() {...};
CompletableFuture<String> heavyTask3(
    String param1, String param2) {...};

CompletableFuture<String> future =
    heavyTask1().thenCompose(s1 ->
        heavyTask2().thenCompose(s2 ->
            heavyTask3(s1, s2);
        )
    );

String result = future.get();
```
As a monad

**Monads allow to express dependent computations**

```
OSGi<ServiceRegistration<POJO>> program = configurations("somePid").flatMap(
cfg ->

services(ServiceA.class, "(scope=" + cfg.get("scope") + ")").flatMap(
    sa ->

services(ServiceB.class, sa.getBFilter()).flatMap(
    sb ->

register(POJO.class, new POJO(sa, sb), new HashMap<>());
));
```
As a monad (normal indentation)

```java
OSGi<ServiceA> serviceAFromConfig(Dictionary<String, ?> cfg) {
    return services(
        ServiceA.class, "(scope=" + cfg.get("scope") + ")")
    );
}

OSGi<ServiceB> serviceBFromServiceA(ServiceA sa) {
    return services(ServiceB.class, sa.getBFilter());
}

OSGi<ServiceRegistration<POJO>> program =
    configurations("somePid").flatMap(
        cfg -> serviceAFromConfig(cfg).flatMap(
            sa -> serviceBFromServiceA(sa).flatMap(
                sb -> register(
                    POJO.class, new POJO(sa, sb), new HashMap<>())
            )));
```
Prominent Applicatives (CompletableFuture)

Applicatives allow to express computations that don’t depend on each other

```
CompletableFuture<String> heavyTask1() {...};
CompletableFuture<String> heavyTask2() {...};

heavyTask1().thenCombine(heavyTask2(), String::concat);
```

**NOTE**

We can see that Applicative express that there is no dependency between heavyTask1 and heavyTask2.
As an Applicative

Applicatives allow to express computations that don’t depend on each other

```java
public class POJO {
    public POJO(ServiceA a, ServiceB b, ServiceC c) {
        ...
    }
}

OSGi<ServiceA> sa = services(ServiceA.class);
OSGi<ServiceB> sb = services(ServiceB.class);
OSGi<ServiceC> sc = services(ServiceC.class);

OSGi<POJO> pojo = OSGi.combine(POJO::new, sa, sb, sc);

it produces the cartesian product of the parameters.
```
Syntax is a problem
On the bright side

- Each step is only executed when there are services or configurations available.
- Effects produced by an instance (service, configuration, etc.) are associated to it.
- When the service or configuration goes away, all the actions triggered by it are undone by the library.
So in our program

```java
OSGi<ServiceRegistration<POJO>> program =
    configurations("somePid").flatMap(
        cfg -> serviceAFromConfig(cfg).flatMap(
            sa -> serviceBFromServiceA(sa).flatMap(
                sb -> register(
                    POJO.class, new Pojo(sa, sb), new HashMap<>())));
```

If any goes away:

- Configuration instance
- ServiceA instance
- ServiceB instance

The corresponding POJO will be unregistered from the OSGi framework and trackers will be closed.
On the bright side

The expressions in our OSGi language are values in Java. We can manipulate the values to extend the program:

prepending or appending steps:

```java
public OSGi<?>> prependDependency(OSGi<?>> program) {
    return service(UndisclosedDep.class).then(program);
}
```
On the bright side

creating conditionals:

```java
public OSGi<ServiceB> chooseService(ServiceReference<?> sr) {
    Object filter = sr.getProperty("filter");

    if (filterProperty != null) {
        return services(ServiceB.class, filter.toString());
    }
    else {
        return just(() -> new ServiceB());
    }
}
```
On the bright side

or simply reusing them:

```
OSGi<ServiceA> sa = services(ServiceA.class);
OSGi<ServiceB> sb = services(ServiceB.class);

OSGi<POJO> pojo = combine(POJO::new, sa, sb);

pojo = prependDependency(pojo);

OSGi<?>> program = sa.flatMap(...)
```
Filtering and replaying

The language also allows to filter and replay already produced events. This is useful to implement, for instance, *highest* service filters or *highest service per property* filters.

```
OSGi<ServiceA> sa = service(ServiceA.class).filter(
    ServiceA::isEnabled);

OSGi<ServiceReference<ServiceA>> hsa = highest(
    serviceReference(ServiceA.class));

OSGi<ServiceReference<ServiceA>> hsProp = highestPer(
    serviceReference(ServiceA.class),
    sr -> sr.getProperty("key"));

hsa.flatMap(...)
hsProp.flatMap(...)
```
The builtin commands make sure they get cleaned when services go away. But when composing our own commands we might need to interleave effects when some services are available.

```java
OSGi<ServiceA> program = services(ServiceA.class).effects(
    sa -> doSomething(sa),
    sa -> undoSomething(sa)
);
```

```java
OSGi<Void> program = onClose(() -> doSomeCleaning());
```
Concurrency

- The language will make sure that, if an effect has been executed, the counter effect is also executed. If the effect is never executed the counter effect is discarded. Although effects order is guaranteed by the monad implementation, counter effects order of execution can’t be guaranteed.

- The implementation is lock free and *should* be safe to use concurrently.
private class MyComponent {

    private final Dictionary<String, ?>> configuration;
    private final Foo mandatory;
    private final ArrayList<ServiceForList> servicesForLists;
    private ServiceOptional optional = null;

    public MyComponent(
        Dictionary<String, ?> configuration, Foo mandatory) {

        this.configuration = configuration;
        this.mandatory = mandatory;
        servicesForLists = new ArrayList<>();
    }
}
public Dictionary<String, ?> getConfiguration() {
    return configuration;
}

public Foo getMandatory() {
    return mandatory;
}

public ServiceOptional getOptional() {
    return optional;
}

public ArrayList<ServiceForList> getServiceForLists() {
    return _serviceForLists;
}
public void setOptional(ServiceOptional optional) {
    this.optional = optional;
}

public void addService(ServiceForList serviceForList) {
    _serviceForLists.add(serviceForList);
}

public void removeService(ServiceForList serviceForList) {
    _serviceForLists.remove(serviceForList);
}
OOP dependency management

- Static dependencies are requested in the constructor.
- Multiple cardinality dependencies are added and removed through add/remove methods.
- Dynamic dependencies can be added and removed using setters.
- Optional properties can be signaled with `Optional` or `null`.
- No need for `@PostConstruct` or other lifecycle annotations, when the class is instantiated it is ready to use.
Component Framework

```java
OSGi<MyComponent> components = combine(
    MyComponent::new,
    configurations("org.components.MyComponent"),
    services(Foo.class));

OSGi<?> program =
    components.flatMap(comp ->
        all(
            dynamic(
                highestService(ServiceOptional.class),
                comp::setOptional, c -> comp.setOptional(null)),
            dynamic(
                services(ServiceForList.class),
                comp::addService, comp::removeService),
            ignore(register(Component.class, comp, new HashMap<>()))
        ));
```

Carlos Sierra Andrés (Liferay, Inc.)
OSGi meets Lambdas
October 2017
OSGi<MyComponent> components = combine(
    MyComponent::new,
    configurations("org.components.MyComponent"),
    highestService(services(Foo.class)));

OSGi<?> program =
    components/effects(
        comp -> comp.setOptional(new ServiceOptional()),
        comp -> comp.setOptional(null)
    ).flatMap(comp ->
        all(
            dynamic(
                services(ServiceForList.class),
                comp::addService, comp::removeService),
            ignore(
                register(
                    Component.class, comp, new HashMap<>()))
        ));
Did I say syntax is a problem?!?
On the bright side

- It is a thin layer on top of OSGi services API. We have almost all the power with much fewer code.
- Cartesian products and dependent trackers with Stream-like API.
- Syntax aside, the DSL allows to structure your code like steps of a script.
- No annotations:
  - So you can reuse existing business classes or use third party classes directly.
  - No lifecycle annotation (@PostConstruct) misinteractions
- It is embedded in Java:
  - we get help from the compiler unlike XML based frameworks.
  - we can create macros or define new functions composing the language primitives using plain Java functions.
- Statements are treated as values, so we can manipulate them and improve reusability.
Running

```java
OSGi<MyComponent> components = ...

OSGiResult result = components.run(bundleContext);
...
result.close();
```

To run the described program we only need to provide a valid `bundleContext`.

*Beware of effects and sharing mutable structures!*