Automatically Managing Service Dependencies in an OSGi Environment

Marcel Offermans

Paris, France
About me...

- Marcel Offermans
- Senior Software Engineer at luminis®
- Our mission is to provide knowledge and products to organisations who create software intensive products, to help them adopt software technology innovations.
- We use OSGi at the core of the architecture for manageable, embedded systems.
Agenda

- Dependencies in OSGi
- Goals for a dependency manager
- Architecture, illustrated by examples
- Conclusions
Dependencies in OSGi

- Package dependencies, which in R4 have been extended with requiring, fragment and extension bundles. These are all resolved in the module layer.
- Service dependencies, which are resolved in the service layer.
Service Dependencies

- Need to be managed at runtime
- The OSGi framework offers basic tools:
  - Service listener
  - Service tracker
- Third party tools:
  - Service binder
  - ...probably there are more :)
Goals for the Dependency Manager

- Minimize the amount of code that needs to be written.
- Provide a clean separation between the service implementation and “glue” code.
- Be dynamic. Allow the programmer to add services and dependencies at any time.
Types of dependencies

• Required dependencies, which need to be resolved before the service can work at all.

• Optional dependencies, which are used when available but are not essential for the service to work.
**Standard use case**

- A service with two required dependencies and one optional one.

```java
public class Activator extends DependencyActivatorBase {
    public void init(BundleContext bc, DependencyManager dm) {
        dm.add(createService()
            .setInterface(MyService.class.getName(), null)
            .setImplementation(MyServiceImpl.class)
            .add(createServiceDependency()
                .setService(HttpService.class)
                .setRequired(true))
            .add(createServiceDependency()
                .setService(SomeOtherService.class)
                .setRequired(true))
            .add(createServiceDependency()
                .setService(LogService.class)
                .setRequired(false)));
    }

    public void destroy(BundleContext bc, DependencyManager dm) {
    }
}
```
Standard use case

- Implementation instantiated lazily, invokes callbacks as part of life-cycle: init, start, stop, destroy
- Dependencies are injected using reflection
- Null object pattern used for optional dependencies

```java
class MyServiceImpl implements MyService {
    private HttpContext httpService;
    private SomeOtherService someOtherService;
    private LogService logService;

    public void start() {
        logService.log(LogService.LOG_INFO, "Starting");
    }
    public void stop() {
        logService.log(LogService.LOG_INFO, "Stopping");
    }
}
```
public class Activator implements BundleActivator {
    private BundleContext context;
    private ServiceRegistration registration;
    private AudioBroadcaster audioBroadcaster;
    private AudioSource audioSource;
    private AudioEncoder audioEncoder;
    private ServiceTracker audioSourceTracker;
    private ServiceTracker audioEncoderTracker;
    private ServiceTracker logTracker;

    public void start(BundleContext context) throws Exception {
        this.context = context;
        audioSourceTracker = new ServiceTracker(context, AudioSource.class.getName(), customizer);
        audioEncoderTracker = new ServiceTracker(context, AudioEncoder.class.getName(), customizer);
        logTracker = new ServiceTracker(context, LogService.class.getName(), null);
        logTracker.open();
        audioSourceTracker.open();
        audioEncoderTracker.open();
    }

    public void stop(BundleContext context) throws Exception {
        audioSourceTracker.close();
        audioEncoderTracker.close();
        logTracker.close();
    }

    private ServiceTrackerCustomizer customizer =
            new ServiceTrackerCustomizer() {
                public Object addingService(ServiceReference reference) {
                    Object service = context.getService(reference);
                    setService(reference, service);
                    return service;
                }
                private void setService(ServiceReference reference, Object service) {
                    Object objectclass = reference.getProperty(Constants.OBJECTCLASS);
                    if (objectclass instanceof String) {
                        String name = (String) objectclass;
                        setNamedService(service, name);
                    } else if (objectclass instanceof String[]) {
                        String[] names = (String[]) objectclass;
                        for (int i = 0; i < names.length; i++) {
                            setNamedService(service, names[i]);
                        }
                    }
                    // register service if necessary
                    if ((registration == null) && (audioSource != null) && (audioEncoder != null)) {
                        // instantiate the implementation and pass the services
                        audioBroadcaster = new AudioBroadcasterImpl(audioSource, audioEncoder, logTracker);
                        registration = context.registerService(AudioBroadcaster.class.getName(), audioBroadcaster, null);
                    }
                    // unregister service if necessary
                    if (((audioSource == null) || (audio Encoder == null)) && (registration != null)) {
                        registration.unregister();
                        registration = null;
                        audioBroadcaster = null;
                    }
                }

                private void setNamedService(Object service, String name) {
                    if (AudioEncoder.class.getName().equals(name)) {
                        audioEncoder = (AudioEncoder) service;
                    } else if (AudioSource.class.getName().equals(name)) {
                        audioSource = (AudioSource) service;
                    }
                }

                public void modifiedService(ServiceReference reference, Object service) {
                    // do nothing
                }

                public void removedService(ServiceReference reference, Object service) {
                    setService(reference, null);
                    context.ungetService(reference);
                }
            };
    
    public void init(BundleContext ctx, DependencyManager manager) throws Exception {
        manager.add(createService() .setInterface(AudioBroadcaster.class.getName(), null) .setImplementation(AudioBroadcasterImpl.class));
        manager.add(createServiceDependency() .setService(AudioSource.class, null) .setRequired(true))
            .add(createServiceDependency() .setService(AudioEncoder.class, null) .setRequired(true))
            .add(createServiceDependency() .setService(LogService.class, null) .setRequired(false));
    }

    public void destroy(BundleContext ctx, DependencyManager manager) throws Exception {
    }
}
Tracking dependencies

- Setting up a service with an optional dependency that can track multiple dependent services:

```java
public class Activator extends DependencyActivatorBase {
    public void init(BundleContext bc, DependencyManager dm) {
        dm.add(createService()
            .setImplementation(DeviceTracker.class)
            .add(createServiceDependency()
                .setService(Device.class)
                .setAutoConfig(false)
                .setCallbacks("addDevice", "removeDevice")
                .setRequired(false)
            );
    }

    public void destroy(BundleContext bc, DependencyManager dm) {
    }
}
```
Tracking dependencies

• Implementation:

```java
public class DeviceTracker {
    private List devs = new ArrayList();
    public void addDevice(ServiceReference ref, Object srv) {
        devs.add(srv);
    }
    public void removeDevice(ServiceReference ref, Object srv) {
        devs.remove(srv);
    }
}
```
Other features:

- Injection of BundleContext and ServiceRegistration
- New services and dependencies can be added or removed dynamically
- Callbacks are configurable and will look for methods with “suitable” signatures
- Service listeners allow you to track the state of a service
- Manager allows for addition of customized dependencies (so you're not limited to service dependencies)
Conclusions

• Clean separation between service implementation and dependency management, you can use a POJO if you want

• Dynamic nature of dependencies has proven to be useful in several scenarios

• Substantial code reduction is realized when compared to using service trackers
Further info

- **Contacting me:**
e-mail: marcel.offermans@luminis.nl
ICQ: 22100024
Skype: marcel_offermans

- **Article:**

- **Development site:**
https://opensource.luminis.net/confluence/x/PwE