OSGi Runtime Engineer Fieldbook

OSGi Runtime Engineer Fieldbook

> A current, plain-language guide to OSGi R8, Apache Karaf, Apache Felix, Eclipse Equinox, bnd, Declarative Services, and production operations.

\newpage

00. The OSGi ecosystem roadmap

OSGi is Java's mature **dynamic module and service system**. It gives every module a precise public boundary, resolves dependencies before code runs, and lets services appear or disappear while the process stays alive.

Explain it like I am five

Imagine a city made of Lego buildings:

- a **bundle** is one building;
- an exported package is a public door;
- an imported package is a door another building needs;
- a **service** is a job advertised in the town square;
- the **framework** is city hall: it checks doors, starts buildings, and keeps the town-square registry accurate.

A bundle cannot secretly see every other bundle. That rule is the source of OSGi's power—and of most beginner confusion.

Current baseline, August 2026

Layer	Current stable reference	What the number means
Specification	OSGi Core R8 and Compendium R8	Standard contracts
Apache Karaf	4.4.11	Managed runtime distribution; supports OSGi R8
Apache Felix Framework	7.0.5	Lightweight framework implementation
Eclipse Equinox	Eclipse Platform 4.40 stream	Framework plus Eclipse runtime ecosystem
bnd / Bndtools	7.3.0	Bundle build, analysis, resolution, testing, baselining

Felix subprojects have independent versions. For example, HTTP, SCR, Configuration Admin, Gogo, and Web Console do not inherit the Framework version. Always pin and verify each component rather than writing “Felix 7” for the whole stack.

The stack in one picture

```
Java VM → OSGi Framework → bundles + service registry → component model → runtime distribution → application
```

Felix and Equinox implement the Framework. Declarative Services (DS) implements a component model on top. Karaf packages a framework plus shell, features, configuration, logging, provisioning, and operations.

When OSGi fits

OSGi is strong when independently versioned modules, long-lived runtimes, plugins, device/edge software, in-process isolation, or controlled live reconfiguration matter. It is not automatically better than a plain Java application. Dynamic behavior has an operational cost.

Feynman check

Explain the difference between specification, framework implementation, component model, build tool, and runtime distribution without using their product names.

\newpage

01. Core mental model: modules, lifecycle, and

services

Three ideas define OSGi: **module boundaries**, **managed lifecycle**, and a **dynamic service registry**.

Modules

An OSGi bundle is a JAR with metadata in `META-INF/MANIFEST.MF`. The framework creates a class space from explicit imports and exports. A bundle sees its own private classes, selected Java platform packages, and packages wired by the resolver. It does not receive one flat application classpath.

Lifecycle

A bundle moves through states:

`INSTALLED → RESOLVED → STARTING → ACTIVE → STOPPING → RESOLVED`

`UNINSTALLED` is terminal for that bundle object. INSTALLED often means a requirement cannot be resolved. ACTIVE means the bundle's start operation completed—not that every business dependency is healthy.

Services

A service is an object registered under one or more interface names with properties. Consumers query by interface and LDAP-style filter. Services can arrive, be replaced, or disappear. Good code expresses this dynamics through Declarative Services instead of caching raw registry objects.

```
@Component(service = PricingService.class)
public final class DefaultPricingService implements PricingService {
  public Money price(Quote quote) { return quote.basePrice(); }
}
```

The DS runtime creates the component, publishes the service, binds references, and withdraws it safely.

Bundle versus component

A bundle is a deployment and class-loading unit. A DS component is a managed object inside a bundle. One bundle can contain many components; one component can publish several service contracts.

Static and dynamic

Package wiring is normally decided at resolve time and stays stable until a refresh. Service wiring is dynamic at runtime. Confusing those two timelines causes poor designs.

Feynman check

A package wire answers “where is this class loaded from?” A service reference answers “which object currently performs this job?” Explain why those are different questions.

\newpage

02. Bundle anatomy and manifest metadata

The manifest is the bundle's machine-readable contract.

Identity and content

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: com.acme.billing.api
Bundle-Version: 2.3.1
Export-Package: com.acme.billing.api;version="2.3.0"
Import-Package: org.osgi.framework;version="[1.10,2)"
```

`Bundle-SymbolicName` identifies the bundle. `Bundle-Version` versions the artifact. Exported packages have their own API versions. Package versions—not Maven artifact versions—drive package resolution.

Import-Package

An import range describes compatible providers. `[1.2,2)` means at least 1.2 and below 2.0. An unbounded or overly wide import can wire to a binary- incompatible provider. A range locked to one micro version blocks safe patches.

Export-Package

Export only intentional API packages. Keep implementation packages private. Use `@ProviderType` for interfaces consumers implement rarely and `@ConsumerType` for interfaces consumers are expected to implement; bnd uses these roles during baselining.

Require-Bundle

`Require-Bundle` exposes a provider bundle's exported packages as a unit and couples to bundle identity. Prefer package imports for normal libraries. `Require-Bundle` remains relevant in ecosystems such as parts of Eclipse where bundle-level relationships are deliberate.

Resources and native code

Bundles can contain configuration, DS descriptors, localization, and native libraries. Class-relative resource lookup and bundle APIs have different visibility semantics. Native code clauses require careful OS, processor, and lifecycle handling.

Let bnd calculate metadata

Handwritten manifests drift. bnd analyzes bytecode to calculate imports, generates DS descriptors from annotations, and warns about split or unused relationships. Keep instructions intentional; inspect the generated manifest.

Feynman check

The Maven dependency says what was available while compiling. The OSGi import says what must be wired while resolving. Explain why shipping the first does not prove the second.

\newpage

03. Class loading, wiring, and uses constraints

OSGi creates a controlled class space for each bundle.

Delegation

Class lookup generally checks Java/platform boot delegation, the bundle's imports, required bundles, its own content, and dynamic imports according to framework rules. Exact details depend on the source of the request. Do not solve class-loading bugs by adding broad boot delegation.

Class identity

A Java type is identified by **binary name plus defining class loader**. Two bundles may contain the same class name yet create incompatible types. This explains failures that look impossible on a flat classpath.

Split packages

A split package appears in more than one bundle. It makes ownership unclear and can create inconsistent class spaces. Refactor toward one provider per package.

Uses constraints

An exported API package may expose types from another package. The resolver's `uses:=` constraint prevents consumer and provider from seeing incompatible providers of that related type.

If `payments.api` exposes `Money` from `money.api`, both sides must use a compatible wire for `money.api`. A uses-constraint error is protecting type identity, not being arbitrary.

Fragments

A fragment attaches content to a host bundle and has no independent lifecycle. Fragments can supply localization, tests, native code, or implementation patches, but increase coupling. Prefer services or normal bundles when possible.

DynamicImport-Package

Dynamic imports postpone package discovery until class loading. They reduce resolver evidence, hurt reproducibility, and turn deployment errors into runtime errors. Use only for deliberate extensibility mechanisms that cannot declare requirements upfront.

Diagnostic sequence

1. Read the unresolved requirement exactly.
2. Inspect provider exports and versions.
3. Ask `bundle:diag`, `headers`, or framework wiring APIs.
4. Trace uses constraints and duplicate providers.
5. Fix metadata or package ownership; do not add wildcard imports.

Feynman check

Why can two objects implement interfaces with identical text and still fail a cast? Because their interface classes came from different defining loaders.

\newpage

04. Resolver, requirements, capabilities, and start levels

The resolver turns declarative requirements into a consistent wiring graph.

Generic model

A resource provides **capabilities** and declares **requirements** in named namespaces. Package imports/exports are one namespace. Others describe bundle identity, execution environment, services, native code, or custom features.

```
Provide-Capability: com.acme.device;
  device.type=charger;
  version:Version=2.0
Require-Capability: com.acme.device;
  filter:="(&(device.type=charger)(version>=2.0))"
```

The resolver recursively finds resources whose capabilities satisfy mandatory requirements. A successful resolution is a graph, not just a list of JARs.

Effective directives

Some requirements matter at `resolve` time; others, commonly service requirements, are `effective:=active`. Build tools can use active requirements to assemble a complete application even when the framework does not wire them as package dependencies.

Optional requirements

Optional means resolution can succeed without the capability. Application behavior must still be correct without it. Do not use optional as a way to hide an unknown dependency.

Start levels

Framework and bundle start levels stage activation. They are an operational ordering tool, not a dependency mechanism. A component should express required services; it should not assume another bundle is ready because its numeric start level is lower.

bndrun

A `.bndrun` file declares the framework, execution environment, repositories, and initial requirements. bnd resolves these into `-runbundles`, producing a repeatable closure and reasons for every selected resource.

Feynman check

A resolver proves that declared needs can be supplied. It does not prove a database is reachable or a business rule is correct. Name the evidence needed for those separate claims.

\newpage

05. Service registry and dynamic collaboration

The service registry is OSGi's in-process collaboration layer.

Registration

A registration contains service interface names, an object, properties, a provider bundle, and lifecycle. Properties support discovery and ranking.

```
(&(objectClass=com.acme.PaymentGateway)(region=eu)(secure=true))
```

Filters use LDAP syntax. Treat property names and value types as API.

Selection

When several services match, service ranking and service ID define ordering. Prefer explicit target filters representing business meaning. Ranking is useful for defaults and replacement, but invisible ranking wars are fragile.

Scope

Services can be singleton, bundle-scoped, or prototype-scoped. Scope answers whether consumers share the same service object. It does not replace thread- safety design.

Dynamics

A provider can stop after a consumer starts. Holding a service object beyond its valid reference lifecycle creates stale collaboration. DS reference policies handle this:

- static: component restarts when a bound service changes;
- dynamic: bind/unbind occurs while the component remains active;
- reluctant: keep a suitable current binding;
- greedy: move to a better candidate.

Choose policy from business continuity and component invariants, not from a desire to avoid restarts.

Registry anti-patterns

- service locator calls spread through business code;
- caching `ServiceReference` or raw service objects indefinitely;
- publishing implementation classes instead of contracts;
- storing large mutable application state as service properties;
- assuming exactly one provider without declaring cardinality.

Feynman check

The registry is a live phone book. DS is the receptionist that updates your call when entries change. Explain why manually polling the phone book spreads infrastructure concerns.

\newpage

06. Declarative Services in depth

Declarative Services (DS) is the default component model for modern OSGi.

Component lifecycle

SCR—the Service Component Runtime—reads component descriptions, waits for required references and configuration, creates the instance, injects references, calls activation, publishes services, and later deactivates it.

```
@Component(
  service = Reconciler.class,
  configurationPid = "com.acme.reconciler"
)
public final class DefaultReconciler implements Reconciler {
  private Ledger ledger;

  @Reference
  void bindLedger(Ledger ledger) { this.ledger = ledger; }

  @Activate
  void activate(ReconcilerConfig config) { /* validate */ }
}
```

Use constructor injection for mandatory unary references when your DS level and tooling support it. Use bind/unbind methods when dynamic replacement is a real requirement.

Cardinality

`1..1` is one mandatory service; `0..1` optional; `1..n` at least one; `0..n` any number. Multiple cardinality can inject a collection, map, tuple, or component service objects depending on the field option.

Immediate and delayed

A delayed component activates on first service use, reducing startup work. Immediate components activate as soon as satisfied. Components without a service are normally immediate. Avoid making everything eager.

Configuration policy

`OPTIONAL`, `REQUIRE`, and `IGNORE` decide whether configuration is necessary. Typed configuration annotations turn string dictionaries into a checked contract. Validate invariants during activation so invalid instances never publish.

Factory components

A factory configuration can create multiple independently configured component instances—for example one connector per charging-network tenant.

Tooling

bnd generates descriptors and requirements from DS annotations. At runtime, inspect `scr:list`, `scr:info`, satisfied references, component failure reasons, and configuration PIDs before editing code.

Feynman check

A bundle can be ACTIVE while a DS component is unsatisfied. Explain this by separating bundle lifecycle from component prerequisites.

\newpage

07. Configuration Admin, Metatype, and Configurator

Configuration Admin (ConfigAdmin) stores and delivers configuration dictionaries to managed services and DS components.

PID and factory PID

A PID identifies one configuration. A factory PID creates many named configurations and therefore many component instances. Keep PIDs stable like an API; renaming them breaks operations and upgrades.

Typed configuration

```
@ObjectClassDefinition(name = "Charger connector")
public @interface ConnectorConfig {
  String endpoint();
  int timeout_ms() default 3000;
```

```
  boolean enabled() default true;
}
```

Metatype annotations describe configuration, types, defaults, labels, and validation hints. DS can map configuration directly to this interface.

Update behavior

An update may modify an active component, deactivate/reactivate it, or create a new factory instance. Understand the component's modified method and whether a half-applied configuration is possible.

Configurator

The OSGi Configurator specification defines JSON resources that seed configurations with ranking and ownership rules. This is useful for immutable application assembly while still allowing controlled runtime overrides.

Karaf configuration

Karaf commonly materializes ConfigAdmin data through files under `etc/`, shell commands, environment/property substitution, or external provisioning. FileInstall watches changes. Treat configuration files as managed state: version, secure, back up, and test their update semantics.

Secrets

ConfigAdmin is not automatically a secret vault. Avoid printing secret values in Web Console, shell output, logs, support snapshots, or source control. Resolve short-lived secrets through a dedicated service where feasible.

Feynman check

Configuration is input to a component, not the component itself. Explain why a configuration update may make one component disappear and another appear.

\newpage

08. Events, logging, promises, and asynchronous flows

OSGi supplies several collaboration styles; use each for its real semantics.

Event Admin

Event Admin sends topic-based events with properties. Synchronous delivery propagates on the caller's thread; asynchronous delivery decouples time but is not a durable message broker.

Use Event Admin for in-process notifications. Do not claim it survives process crashes, gives exactly-once delivery, or replaces Kafka/JMS.

Typed Event

Typed Event provides type-safe event models while retaining OSGi dynamics. Define ownership, ordering, error handling, and whether listeners may block.

Log Service

The OSGi Log Service associates entries with bundles, services, and log levels. Log Reader and Log Stream enable observation. Bridge carefully to SLF4J or a runtime backend to avoid loops and duplicate bindings.

Promises and Push Streams

Promise represents a future result with success/failure composition. Push Stream models asynchronous streams with buffering and backpressure concepts. They are useful when asynchronous APIs are genuine—not as decoration over blocking work.

Threading rules

- document the thread that calls consumers;
- never hold framework or component locks across slow work;
- close streams and executors with lifecycle;
- bound queues and concurrency;
- preserve diagnostic context across async boundaries.

Feynman check

An event tells listeners something happened now. A durable message records work that must survive failure. Explain which one an invoice settlement requires.

\newpage

09. HTTP Whiteboard, Jakarta REST, and web integration

Whiteboard patterns let applications publish web endpoints as services.

HTTP Whiteboard

Servlets, filters, listeners, and resources register with properties declaring patterns and context selection. The runtime tracks their lifecycle.

```
@Component(service = Servlet.class, property = {
  "osgi.http.whiteboard.servlet.pattern=/health/*",
  "osgi.http.whiteboard.servlet.asyncSupported=true"
})
public final class HealthServlet extends HttpServlet { }
```

Context helpers isolate applications and security policies. Target filters bind servlets to the intended context. Failures often come from property type, pattern, context selection, or DTO failure reasons—not from the servlet code.

Jakarta REST Whiteboard

Jakarta REST resources and applications can also be registered as services. Use standard whiteboard properties for application base, resource role, and extensions. Track the runtime's Jakarta namespace and compatible servlet stack.

Karaf web stack

Karaf 4.4.x uses Pax Web and supports feature-based provisioning. Its exact Jetty, Servlet/Jakarta, and whiteboard versions are part of the runtime compatibility matrix. Do not independently upgrade one web bundle in production without resolving and testing the complete feature.

Security

Authenticate at an explicit context/filter boundary and authorize again in the business service. Configure TLS, forwarded headers, upload limits, timeouts, error responses, and management endpoint isolation.

Feynman check

The whiteboard is a noticeboard: a servlet pins up “I handle `/health/*`.” The runtime installs it into the matching context. Explain how removing the service removes the route.

\newpage

10. bnd, Bndtools, Maven, Gradle, and baselining

bnd is the core tool for creating and analyzing OSGi artifacts. Bndtools adds an Eclipse workspace and runtime experience; Maven and Gradle plugins integrate the same analysis into ordinary builds.

Build by analysis

bnd reads compiled bytecode, determines referenced packages, calculates imports, applies export instructions, generates DS/Metatype descriptors, and verifies metadata.

```
Bundle-SymbolicName: com.acme.charging.core
Export-Package: com.acme.charging.api;version=1.4.0
-dsannotations: *
-metatypeannotations: *
-reproducible: true
```

Avoid `Import-Package: *` as a thoughtless handwritten manifest. In bnd, a final `*` instruction means “include remaining calculated imports” and is often appropriate after intentional overrides.

Resolution

Use `.bndrun` requirements plus repositories to calculate `-runbundles`. Commit or reproduce the resolution and fail CI when the declared runtime no longer resolves.

Semantic versioning and baselining

bnd compares a new API with a previous bundle:

- compatible addition usually increments minor;
- breaking API change increments major;
- implementation-only fix increments micro.

Provider and consumer interface roles affect compatibility advice. Baselining turns version discipline into build evidence.

Maven and Gradle

Use current bnd plugins, pin versions, keep generated manifests inspectable, and run integration tests against the produced JAR—not just IDE output. Maven coordinates are repository identity; they do not replace OSGi metadata.

Reproducibility

Pin repositories, plugin versions, execution environment, framework, and runtime closure. Sign releases, generate an SBOM, and compare the deployed digest to CI output.

Feynman check

bnd does not “make a JAR modular” by magic. It discovers bytecode relationships and writes a contract. You still decide the public API and acceptable ranges.

\newpage

11. Testing bundles, components, resolution, and dynamics

An OSGi test strategy has layers because plain Java correctness and runtime wiring are different claims.

Plain tests

Keep business services ordinary Java and test them without a framework. Mock or fake service contracts at the object boundary. These tests are fast and explain domain behavior.

Metadata tests

Inspect generated manifests, DS descriptors, Metatype, capabilities, and API baselines. A build should fail on unresolved package imports, accidental exports, or semantic-version violations.

Framework integration

Launch Felix or Equinox with the actual runtime closure. Verify:

- bundles resolve and reach intended states;
- DS components become satisfied;
- configuration creates and updates instances;
- services bind, unbind, and replace correctly;
- HTTP endpoints appear and disappear;
- stopping a provider does not corrupt consumers.

The OSGi Test project supplies JUnit 5 extensions and assertions for framework objects. Pax Exam remains common in Karaf and integration suites. bnd testing can launch resolved test runs.

Failure drills

Tests should stop a provider during work, update configuration, introduce two candidates with ranking, refresh an exported package, deny a permission where supported, exhaust a bounded dependency, and restart from persisted state.

Resolver tests

Resolve the application from clean repositories in CI. Test both the chosen closure and expected failures, so a missing capability produces a useful diagnostic.

Feynman check

A unit test proves an object. A resolver test proves declared compatibility. A framework test proves dynamic wiring. A production probe proves the deployed instance. Do not substitute one for another.

\newpage

12. Apache Felix: framework and subprojects

Apache Felix is a compact OSGi Framework implementation and a family of independently released OSGi service implementations and tools.

Framework

Felix Framework 7.0.5 implements the OSGi R8 framework APIs. It can run as a distribution or be embedded via the standard `FrameworkFactory` API.

```
ServiceLoader.load(FrameworkFactory.class)
  .findFirst()
  .orElseThrow()
  .newFramework(configuration);
```

Embedding makes lifecycle, storage, parent class loading, and shutdown your application's responsibility.

Important subprojects

- SCR: Declarative Services runtime;
- Config Admin, Metatype, Configurator;
- Gogo runtime/shell/commands;
- FileInstall: watches deployment/configuration directories;
- Web Console: administrative plugins;
- HTTP Service and Whiteboard implementations;

- Resolver, Repository, Converter, Health Checks, Inventory;
- Maven Bundle Plugin and related bnd-based build tooling.

Versions are independent. Read each release note and compatibility range.

Framework configuration

Choose a persistent storage directory, storage-clean policy, beginning start level, boot delegation only when required, and system package extras only for deliberate host APIs. Give each test/runtime its own cache.

Gogo

Gogo is a small command processor used by Felix and Karaf. Commands are OSGi services, so they appear dynamically. Treat remote shells as privileged administration surfaces.

Web Console

Web Console is excellent for inspecting bundles, services, DS, configuration, logs, and system information. In production, require strong authentication, TLS, network restriction, auditing, and plugin minimization.

Feynman check

Felix Framework is the engine. Felix SCR, HTTP, ConfigAdmin, and Web Console are replaceable parts installed around it. Explain why their versions differ.

\newpage

13. Eclipse Equinox, PDE, p2, applications, and products

Equinox is the OSGi Framework used by Eclipse and an ecosystem of runtime services. The Eclipse Platform 4.40 release stream is the current stable baseline in August 2026.

Equinox framework

The standalone framework lives in `org.eclipse.osgi`. Equinox implements OSGi Core and adds mature diagnostics, extension registry integration, launcher, application model, security history, and platform behaviors.

Extensions versus services

Eclipse extension points are declarative contributions read by the extension registry. OSGi services are dynamic runtime objects. Eclipse applications often use both. Prefer services for dynamic collaboration; use extension points where the product ecosystem defines that contract.

PDE

Plug-in Development Environment (PDE) develops bundles/plugins, features, target platforms, applications, and products. A target platform pins what the workspace compiles and resolves against. Never let “whatever is installed in my IDE” become the implicit production target.

p2

p2 is Eclipse's provisioning system. Installable Units describe capabilities, requirements, artifacts, touchpoints, and update metadata. Repositories can assemble and update products. p2 identity/version semantics interact with OSGi bundle metadata but are not identical to Maven.

Products and applications

An Eclipse application is an executable contribution. A product defines branding, launcher, configuration, features/plugins, and update behavior. Feature-based products curate installable units; bundle-based products list plugins directly.

Diagnostics

Use the OSGi console, bundle wiring, DS/component views, PDE dependency analysis, p2 director logs, state area, and error log. Avoid deleting the entire configuration/state directory before capturing evidence.

Feynman check

Equinox is the module engine; PDE builds the product; p2 installs and updates it; extension points contribute declared features; services provide live objects.

\newpage

14. Apache Karaf 4.4: provisioning and operations

Apache Karaf is a managed OSGi runtime distribution. Version 4.4.11 is the current stable release and its 4.4 line supports OSGi R8 and Java 11+.

What Karaf adds

- local and SSH shell;
- features repositories and application provisioning;
- configuration in `etc/`;
- logging, audit, JAAS-based security facilities;
- hot deployment;
- instance and wrapper tooling;
- integration with Pax Web, Aries, CXF, Camel, Decanter, and Cellar.

Features

A feature is a named, versioned provisioning recipe listing bundles, configs, libraries, requirements, and dependent features.

```
<feature name="charging-api" version="1.4.0">
  <feature>scr</feature>
  <bundle>mvn:com.acme/charging-api/1.4.0</bundle>
  <bundle>mvn:com.acme/charging-core/1.4.0</bundle>
</feature>
```

Feature resolution selects a compatible closure. Pin repositories and feature versions. Test installation from an empty Karaf, not a developer cache.

KAR and deploy directory

A KAR packages a features repository and artifacts for offline deployment. The `deploy/` directory is convenient but implicit watching can surprise production operations. Prefer an explicit, audited provisioning path.

Shell diagnostics

Start with:

```
bundle:list
bundle:diag <id>
headers <id>
```

```
services
scr:list
scr:info <component>
config:list
feature:list -i
log:tail
```

Upgrades

Treat Karaf, framework, Pax Web, logging, features, Java, and applications as one tested compatibility set. Use immutable base distributions and externalized state. Rehearse config migration, data compatibility, feature rollback, and clean-cache recovery.

Security

Restrict SSH and JMX, rotate credentials and host keys, audit commands, limit Web Console, protect `etc/`, verify artifacts and signatures, and isolate management from public application traffic.

Feynman check

Karaf is not a different module standard. It is an operator-friendly house built around an OSGi framework and selected service implementations.

\newpage

15. Apache Aries, Blueprint, transactions, and persistence

Apache Aries provides enterprise OSGi implementations and integration modules.

Blueprint

Blueprint is a standardized dependency-injection container designed for dynamic services. XML declares beans, service publications, references, and lifecycle. An extender discovers `OSGI-INF/blueprint/*.xml`.

```xml
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <reference id="ledger" interface="com.acme.api.Ledger"/>
  <bean id="reconciler" class="com.acme.impl.Reconciler">
    <property name="ledger" ref="ledger"/>
  </bean>
  <service ref="reconciler" interface="com.acme.api.Reconciler"/>
</blueprint>
```

For new annotation-friendly services, DS is usually simpler and more directly tooled. Blueprint remains important in Karaf, Camel, CXF, and established enterprise systems.

Transactions

Aries transaction control coordinates local or XA resource work through OSGi services. Define transaction boundaries in application services. XA can solve specific atomicity needs but adds recovery, heuristic, and operational cost. Outbox/idempotency patterns may be a better distributed boundary.

JPA and JDBC

OSGi persistence must handle provider, entity, datasource, weaving, class loading, and lifecycle. Use the current Data Service/JDBC and transaction contracts where suitable. Test against the exact framework and provider set.

CDI

OSGi CDI integrates CDI component models with the service registry. Do not mix DS, Blueprint, and CDI in one module without a clear ownership and lifecycle model.

Feynman check

The registry connects services. DS, Blueprint, or CDI decides how objects are created and connected to that registry. Choose one primary component model per area.

\newpage

16. Remote Services and distributed OSGi

OSGi Remote Services lets a local service be exported and represented by a proxy in another framework. Remote Service Admin (RSA) handles endpoints, topology, discovery, import, export, and intents.

Local illusion, remote reality

A remote proxy may implement the same Java interface, but calls cross a network. Latency, partial failure, serialization, authentication, compatibility, and retries now matter. Never design a chatty local interface and export it unchanged.

Endpoint model

Export properties describe which service to export, supported configurations, and intents such as security. Discovery distributes endpoint descriptions. Import creates a local service proxy whose availability follows the endpoint.

Providers

Eclipse Communication Framework (ECF) and Apache CXF Distributed OSGi are common implementations. Choose from protocols, discovery, security, interoperability, maintenance, and operational tooling—not only annotation convenience.

Contract design

- coarse operations with explicit deadlines;
- versioned, portable DTOs;
- idempotency for retried commands;
- bounded payloads;
- authentication and authorization on both sides;
- telemetry with endpoint identity;
- circuit breaking or admission control outside the framework contract.

Topology

Decide whether exports are automatic or centrally controlled. Prevent a test service from being exported because it happens to match a broad property.

Feynman check

Remote Services can make a remote endpoint look like a service in the registry. It cannot make the network behave like a method call.

\newpage

17. Production operations, security, and observability

Dynamic modularity needs disciplined operations.

Health model

Report separately:

- process/JVM liveness;
- framework state and unresolved bundles;
- DS unsatisfied components;
- required service availability;
- configuration validity;
- business readiness;
- external dependency health.

A bundle count alone is not health. A deliberately inactive optional bundle is not a failure.

Metrics and logs

Measure service bind/unbind events, component activation failures, resolver errors, bundle state changes, configuration updates, HTTP routes, feature install duration, thread pools, heap/GC, and user outcomes. Control cardinality: bundle symbolic name is useful; service ID per request is usually not.

Support snapshot

Capture framework/runtime version, Java build, installed features, bundle states, unresolved diagnostics, wirings, DS states, sanitized configuration, recent logs, threads, memory summary, artifact digests, and timestamps. Redact secrets and personal data.

Security boundaries

Modern Java no longer relies on Security Manager as a general sandbox. OSGi metadata gives class-space encapsulation, not hostile-code isolation. Use process/container boundaries for untrusted code.

Protect provisioning repositories, signatures, remote shell, JMX, Web Console, HTTP management, configuration, bundle cache, logs, and dumps. Patch the JDK and every independently versioned OSGi component.

Safe update

1. resolve and integration-test the closure;
2. confirm API baselines and config migration;
3. stage with the real state shape;
4. quiesce work if live update is unsafe;
5. install/update, refresh only required wirings;
6. verify components and business probes;
7. roll back with compatible data/config.

Feynman check

Dynamic update is a capability, not a requirement. Explain when a controlled process restart is safer than refreshing bundles in place.

\newpage

18. Real-life scenario: ChargeGrid

ChargeGrid operates 70,000 EV chargers from multiple vendors. Sites remain available during network loss, tariffs change by region, and operators deploy connector fixes without rebuilding the whole control station.

Requirements

- each vendor protocol is replaceable;
- core billing never imports vendor implementation packages;
- connector configuration can create one instance per site;
- offline commands queue with bounded storage;
- duplicated messages never create duplicate charges;
- operators can diagnose one site without exposing credentials;
- updates are signed, staged, observable, and reversible.

Bundle map

```
chargegrid.model.api       records, IDs, money, states
chargegrid.connector.api   ChargerConnector service contract
chargegrid.session.core    charging-session state machine
chargegrid.tariff.api      TariffService contract
chargegrid.vendor.ocpp     OCPP connector implementation
chargegrid.vendor.legacy   legacy serial connector
chargegrid.persistence     checkpoint and idempotency store
chargegrid.http            HTTP/Jakarta REST whiteboard resources
chargegrid.operations      health, commands, support snapshot
```

API bundles export small packages. Implementation bundles export nothing. `session.core` imports only model and service API packages.

Dynamic services

Each vendor bundle publishes `ChargerConnector` with properties:

```
protocol=ocpp
protocol.version=2.0.1
region=eu
secure=true
```

The station coordinator has a multiple DS reference and indexes connectors by declared protocol. Removing one connector stops new assignments, drains bounded work, checkpoints state, and then unbinds.

Configuration

A factory PID creates one `SiteConnector` component per charging site. Typed Metatype declares endpoint, site ID, TLS profile, timeout, and concurrency. Secrets are references to a vault-backed `CredentialService`, not literal passwords in ConfigAdmin.

Correctness

Every command has a durable `(siteId, commandId)` idempotency key. Session state changes and outbound intent commit together. A dispatcher retries safe commands with deadlines and jitter. Reconciliation compares station counters, accepted events, and billed energy.

Karaf assembly

Karaf 4.4.11 provides the managed runtime. A versioned ChargeGrid features repository installs SCR, ConfigAdmin, HTTP, logging, APIs, then implementations. Provisioning occurs from a signed internal repository with no public shell.

Failure walkthrough

When the OCPP service disappears:

1. DS unbinds the connector from the coordinator.
2. New OCPP work is rejected or queued within a fixed site quota.
3. Existing calls reach a deadline; no lock spans the network.
4. Idempotency makes a later retry safe.
5. Health shows the affected protocol and site count.
6. The legacy connector and core tariff services remain active.

Upgrade

A connector API addition passes bnd baseline as a minor-compatible change. The new provider is installed beside the old one with a higher service ranking only for a canary site filter. Metrics and reconciliation prove correctness before the remaining configurations move. Rollback restores ranking and config; the data format remains backward compatible.

Architecture verdict

OSGi fits because connector diversity, long-lived edge runtimes, controlled plugin boundaries, and partial updates are primary requirements. If ChargeGrid were one stateless cloud API redeployed as a unit, a plain

modular Java service might be simpler.

Feynman check

Trace one “start charging” command through HTTP service, session core, connector selection, durable idempotency, vendor service, event result, and telemetry. Then explain what happens at every step if the connector vanishes.

\newpage

19. Decision guide, migration, interview, and glossary

The strongest OSGi engineer knows both how to use it and when not to.

Choosing the runtime

Need	Good starting point
Small embedded/hosted dynamic framework	Felix Framework
Eclipse RCP, IDE tooling, p2 product	Equinox + PDE/p2
Managed server runtime, shell, features	Karaf
Bundle creation and resolution	bnd/Bndtools or bnd Maven/Gradle plugins
Annotation-first components	Declarative Services
Existing Karaf/Camel/CXF XML assembly	Blueprint where already appropriate

OSGi versus JPMS

JPMS gives Java-platform modules and strong static readability. OSGi gives versioned packages, resolver metadata, lifecycle, and dynamic services. They can coexist but duplicate module descriptors, class loading, reflection, and build tooling require deliberate testing. Do not promise automatic dual modularity.

Migration path

1. identify package ownership and remove split packages;
2. separate API from implementation;
3. add bnd analysis without changing runtime;
4. baseline public APIs;

5. introduce services at volatile boundaries;
6. resolve a small application closure;
7. add DS and framework integration tests;
8. move operations/configuration only when the value is clear.

Interview frame: BUNDLE

- **B**oundaries: exported APIs and private implementation
- **U**ses/wiring: requirements, capabilities, versions
- **N**amed services: contracts, properties, cardinality
- **D**ynamics: bind/unbind, lifecycle, configuration
- **L**aunch/provision: bndrun, features, p2, runtime
- **E**vidence: resolution, tests, health, logs, rollback

Glossary and abbreviations

Term	Plain meaning
OSGi	Dynamic module and service specifications for Java
Bundle / BSN	Modular JAR / Bundle-SymbolicName
Framework	Runtime implementing OSGi Core
Wiring	Resolved connection between requirement and capability
DS / SCR	Declarative Services / its Service Component Runtime
CM / Metatype	Configuration Admin / configuration-description standard
RSA	Remote Service Admin
HTTP WB / JAX-RS WB	HTTP / Jakarta REST Whiteboard
bnd / bndrun	Build-analysis tool / runtime-resolution description
Gogo	Dynamic command shell used by Felix and Karaf
KAR	Karaf archive containing features and artifacts
PDE / p2	Eclipse plug-in development / provisioning system
RCP	Eclipse Rich Client Platform
API / SPI	Consumer-facing contract / provider extension contract
PID / factory PID	Configuration identity / multi-instance configuration identity
LDAP filter	Parenthesized service/capability matching expression
EE	Java execution environment requirement

Term	Plain meaning
SLA / SLO	Agreement / measurable reliability objective
SBOM	Software bill of materials

Official references

- OSGi Core Release 8
- OSGi Compendium Release 8
- OSGi Working Group
- Apache Karaf downloads
- Apache Felix documentation
- Eclipse Equinox downloads
- bnd documentation
- Apache Aries Blueprint

Final Feynman challenge

Explain one ChargeGrid command from bundle resolution to service selection, configuration, remote I/O, failure, telemetry, update, and rollback. Anything you cannot explain without the word “magic” is your next lab.