OSGi Alliance Community Event

WebLogic Event Server

Alexandre Alves
Agenda

• Problem domain
• WL-EvS: what is it and why it is needed
• Why the OSGi technology for WL-EvS?
• WL-EvS architecture using OSGi
• WL-EvS programming model using OSGi
• Lessons learned
• Challenges
• Next steps
Problem Domain

Event Processing Requirements
- Large volumes of streaming events
- Sub-millisecond response time
- Even under heavy load

- Financial Services
  - Algorithmic trading

- Transportation & Logistics
  - Asset management

- Telecommunications & Services

- Public Sector & Military
  - Intrusion detection systems
  - Military asset allocation

- Manufacturing
  - ‘Negative Working Capital’ inventory management

- Insurance
  - Responses to calamities - earthquake, flooding

- Algorithmic trading
- Asset management
- ‘Negative Working Capital’ inventory management
- Insurance
Agenda

- Problem domain
- **WL-EvS:** what is it and why it is needed
- Why the OSGi technology for WL-EvS?
- WL-EvS architecture using OSGi
- WL-EvS programming model using OSGi
- Lessons learned
- Challenges
- Next steps
What is WL-EvS (now Oracle CEP)?

- **Just Enough Application Server for Event Processing**
  - Event-based programming model
    - Application logic as event-driven beans
    - Declarative language for event processing
  - Event-based container features
    - Embedded Complex Event Processing engine
    - Streaming event adapters (e.g. HTTP pub/sub, stock market data feeds)
    - Event record/playback
    - Support for real-time response and high-performance
What is WL-EvS?

- Sensing
- Correlation
- Aggregation
- Filtering
- Respond
Why is WL-EvS needed?

- Existing programming models, such as enterprise Java Beans, are not applicable
  - Request-Response versus One-way (Sense-Respond)

- Existing infrastructures were not:
  - latency-sensitive (tend to focus on throughput)
  - jitter-free (i.e. real-time)
Agenda

- Problem domain
- WL-EvS: what is it and why it is needed
- **Why the OSGi technology for WL-EvS?**
- WL-EvS architecture using OSGi
- WL-EvS programming model using OSGi
- Lessons learned
- Challenges
- Next steps
Why the OSGi technology for WL-EvS?

• Business Drivers
  • Time to market
    • Needed to put together complete stack, but wanted to release within 1-2 years
  • Green Field
    • Needed to be able to quickly R&D different and innovative approaches to solve complex new problem
      • E.g. financial market demands processing of 100K+ events within single digit milliseconds
Why the OSGi technology for WL-EvS?

- Technology Drivers
  - Improved modularization
    - Better code reuse within product families
  - Improved maintainability and flexibility
    - De-coupling of application-code from infrastructure-code
    - De-coupling of service interface from service implementations
  - Improved pluggability, extensibility, and dynamism
    - Particularly around SPI
  - Portability
    - Should be able to swap kernel/backplane implementations
Agenda

- Problem domain
- WL-EvS: what is it and why it is needed
- Why the OSGi technology for WL-EvS?
- WL-EvS architecture using OSGi
- WL-EvS programming model using OSGi
- Lessons learned
- Challenges
- Next steps
Modularization

• Layered architecture where all components are implemented as bundles
  • Backplane layer: equinox
  • Base layer: thread management, logging, data-source, network connection management
  • Generic sub-system layer: configuration sub-system, security sub-system, monitoring sub-system
  • Domain-specific sub-system layer: CEP engine, stream management, event store
  • Application layer: WL-EvS applications
Modularization (2)

- Products being modularized to provide bundles
  - WLS provides over 300 bundles
  - Several third-party products were made into bundles
    - Jetty, ACL, antlr
- New products, such as WL-EvS, hosted on OSGi backplane and produce and consume bundles
  - WL-EvS consumes over 100 bundles
Modularization (3)

• OSGi bundles: explicit dependency model
  • Helped with modularization effort
    • Problems are surfaced sooner (e.g. friends-and-family APIs become apparent)
  • Helped with build process (don’t under-estimate the problem of building large systems)
Modularization (4)

• Versioning
  • Better release management
    • Allows individual bundles to be developed, tested independently, and hence the promotion of individual features
    • Quicker and smaller releases rather than huge multiple-year release-all-or-nothing
  • Runtime resilience
    • Dynamic patching of features
Modularization (5)

• Logistic problem: how does one integration-test/system-test the combination of hundreds of bundles?
  • Certain bundles tend to cluster together (e.g. net-io & work-manager)
  • Bundles are grouped into feature sets and tested as a unit
• End-user may specify feature-sets, and hence dynamically tuning runtime
Modularization (6)

- Feature-Sets
  - base: utilities, standard configuration, etc.
  - enterprise-base: stax, bea xmlbeans, jdo, persistence, transaction, etc.
  - enterprise: JNDI, JDBC, RMI, JTA, and a few more
  - http-jetty: Jetty 6.0 integrated with other mSA components
  - ws-glassfish: JAX-WS support from Project Glassfish and integration
  - spring: core Spring support, plus Spring/OSGi integration
  - security-css: Common Security Services
  - ede: CEP engine, stream-management
Service Oriented Implementation

- SOA deals with programming-in-the-large
  - Interaction between system components (e.g. WS-clients and WS-providers through WSDL)

- OSGi Service Layer allows one to bring SOA concepts into the system component implementation level (e.g. programming-in-the-small)
OSGi Service Layer

- De-coupling of interface and implementation allows the selection of different implementation providers
  - Cache
    - Native implementation from WLS, Tangosol, etc
  - Web-containers
    - Jetty, Tomcat
  - Authentication/Authorization providers
    - LDAP, file-system
OSGi Service Layer (2)

- WL-EvS infrastructure features are composed of at least two bundles
  - API bundle exports public service interface
  - Implementation bundle registers service implementation with appropriate properties
- WL-EvS applications retrieve services from OSGi registry filtering on service reference properties
  - This is done transparently to user, using Spring-DM
OSGi Service Layer (3)

- Service Management
  - Services can be monitored at runtime
  - New services can be dynamically added to cope with demand
  - Offending services can be un-registered, and swapped dynamically
Extensibility

- WL-EvS provides Service Provider Interface
  - Vendors can plug-in different processing engines, cache providers, etc.
- SPI implemented using the Whiteboard pattern
  - Vendors implement service interfaces and register providers in the OSGi registry
  - Infrastructure selects services based upon ranking and properties
  - Simple, easy, and powerful (dynamic)
Portability

- OSGi bundle is the deployment unit for both infrastructure features and application code
  - Easier to re-use vendor bundles originally designed for other stacks (e.g. Apache Felix, Knopflerfish)
  - In theory, should allow us to swap the backplane implementation if needed
  - Allows us to leverage tooling for the authoring of bundles (e.g. Eclipse PDE, bnd)
  - Of course, this does not mean that one can deploy a WL-EvS application bundle into some other backplane
Agenda

- Problem domain
- WL-EvS: what is it and why it is needed
- Why the OSGi technology for WL-EvS?
- WL-EvS architecture using OSGi
- WL-EvS programming model using OSGi
- Lessons learned
- Challenges
- Next steps
Deployment unit

- CEP (Adapter, Stream, Processor)
- Spring Integration (WLEvS Tags)
- Management and Monitoring
- Core Integration Technologies
- Security, JMS, Jetty, etc
- Spring/OSGi Integration
- “Backplane”
- OSGi Runtime
Application Model

<<OSGi Service>>
Service1
Export

<<OSGi Service>>
Service2
Import

Data Source
Adapter
<<Source>>
Stream
Processor
Rule
Rule
Stream
POJO
<<Sink>>

OSGi Bundle/Spring Application Context

Event Processing Network (EPN)
Application Model (2)

Supports composition and layering of events
Programming Model

- Extended Spring-context XML configuration to support EDA concepts
  - New tags: `<wlevs:processor>`, `<wlevs:event-bean>`
- Spring-DM to assemble it all together, referencing infrastructure services transparently
  - `<wlevs:processor>` tag implementation retrieves `com.bea.wlevs.Processor` service from OSGi registry
Programming Model (2)

- As an extension to Spring-beans, there is seamless integration to legacy Spring-framework beans
- Users are free to interact directly with OSGi services through `<osgi:service>` and `<osgi:reference>` to implement complex scenarios
Schema-Driven

• WL-EvS is completely schema-driven
  • All server configuration can be done through XML files
  • XSD available for all features, even internal ones (e.g. `<netio>`)  
• Spring-DM used in several places:
  • Application assembly
  • Deployment configuration
• Management layer for all configuration is provided, which allows dynamic changes to configuration
Agenda

- Problem domain
- WL-EvS: what is it and why it is needed
- Why the OSGi technology for WL-EvS?
- WL-EvS architecture using OSGi
- WL-EvS programming model using OSGi
- Lessons learned
- Challenges
- Next steps
Lessons learned

• There are always opportunities for re-use
  • Re-use within organization
  • Re-use of standard services
    • HTTP Service
    • Service Tracker
    • Initial Provisioning
    • Declarative Services using Spring-DM
    • Start Level Service
• Modularize at all levels
  • WL-EvS programming model itself is a separate bundle, decoupled from other services, which means WL-EvS could in theory support other programming models, such as SCA, etc.
Challenges

• Mind-set:
  • Understand that it is more work to create a modular solution, but it pays off long-term

• Design-time:
  • Very large *Import-Packages*
    • Error-prone
  • Non-intuitive *Import-Packages*
    • Hard to get correct when reflection is used (e.g. Kodo)
Challenges (2)

• Runtime:
  • Hard to debug complex class-path resolving
    • `instanceof` just fails sometimes…
  • Service availability race-conditions
    • Client applications referencing to services that have not been bound it
    • Particularly a problem during start-up

• Certain features are missing or too hard to use:
  • Security, Configuration support, Transaction support
Next steps

• (Try to) leverage more of the standard services:
  • Event Admin Service, Conditional Permission Admin, Configuration Admin
• Better tooling, extensions to PDE
• Inter-process communication between OSGi runtimes
Conclusion

• Goals were achieved
  • Able to ship within 1 year of start of development
    • One of the first *domain-specific* application server in the market
  • Able to meet performance goals:
    • Processing 1M events/sec continuously with average latency under 70 microsecs (http://dev2dev.bea.com/pub/a/2007/12/event-server-performance.html)