OSGi as a Framework for Building a Product Line: Experience and Best Practices

Afkham Azeez [azeez@apache.org]

Ruwan Linton [ruwan@apache.org]

Agenda

- SOA Middleware Products
- Componentized SOA Platform
- Why OSGi?
- WSO2 Carbon
- Challenges faced
- Best practices

SOA Middleware Products

- Traditional SOA platforms are:
  - Overly complex
  - Heavyweight
  - Not internally consistent
  - Hard to learn
  - Acquired, not designed
  - The components don't fit well together
- The result is:
  - Centralizing SOA on a single hub based ESB
  - Relying on the “SOA team” to solve problems
  - Replacing centralized monolithic applications with
    - Centralized monolithic ESB and SOA platform

What is wrong with existing SOA Platforms?

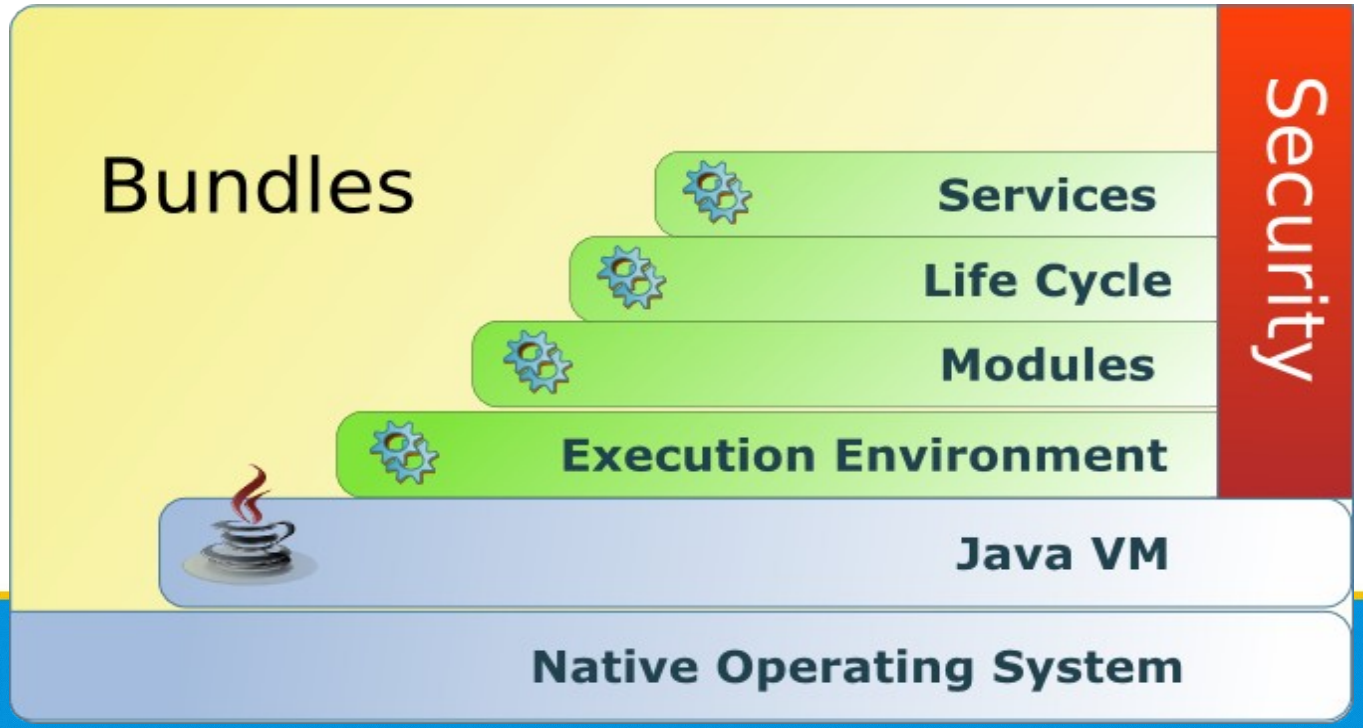
Agile Adoption	• Heavyweight, complex and bloated • Downloading an evaluation can take days
Skills	• SOA adoption is restricted because tools are so difficult to use
SOA Deployment	• Large SOA platform "components" are actually distinct products • Bigger installations encourage centralization
Continuity	• Adding new aspects of SOA (e.g. process choreography) means installing and learning a completely new product
Product Integration	• Customers waste time integrating vendor's products • Time that could be spent integrating real applications

Solution?

- Componentized SOA Platform
  - WSO2 Carbon Architecture based on OSGi

Why OSGi?

- It's a framework which specifies a modular architecture for dynamic component based systems.
- It can be used as the underlying core modularization technology.



Why OSGi?

- **It's a dynamic module system for Java.**
  - Bundle(== jar) is called the Modularization unit.
  - Meta data can be specified in manifest headers.
  - Separate classloader for a bundle.
  - A bundle can share or hide packages.
  - Allows multiple versions of the same class in a single VM. This solves jar hell problem.

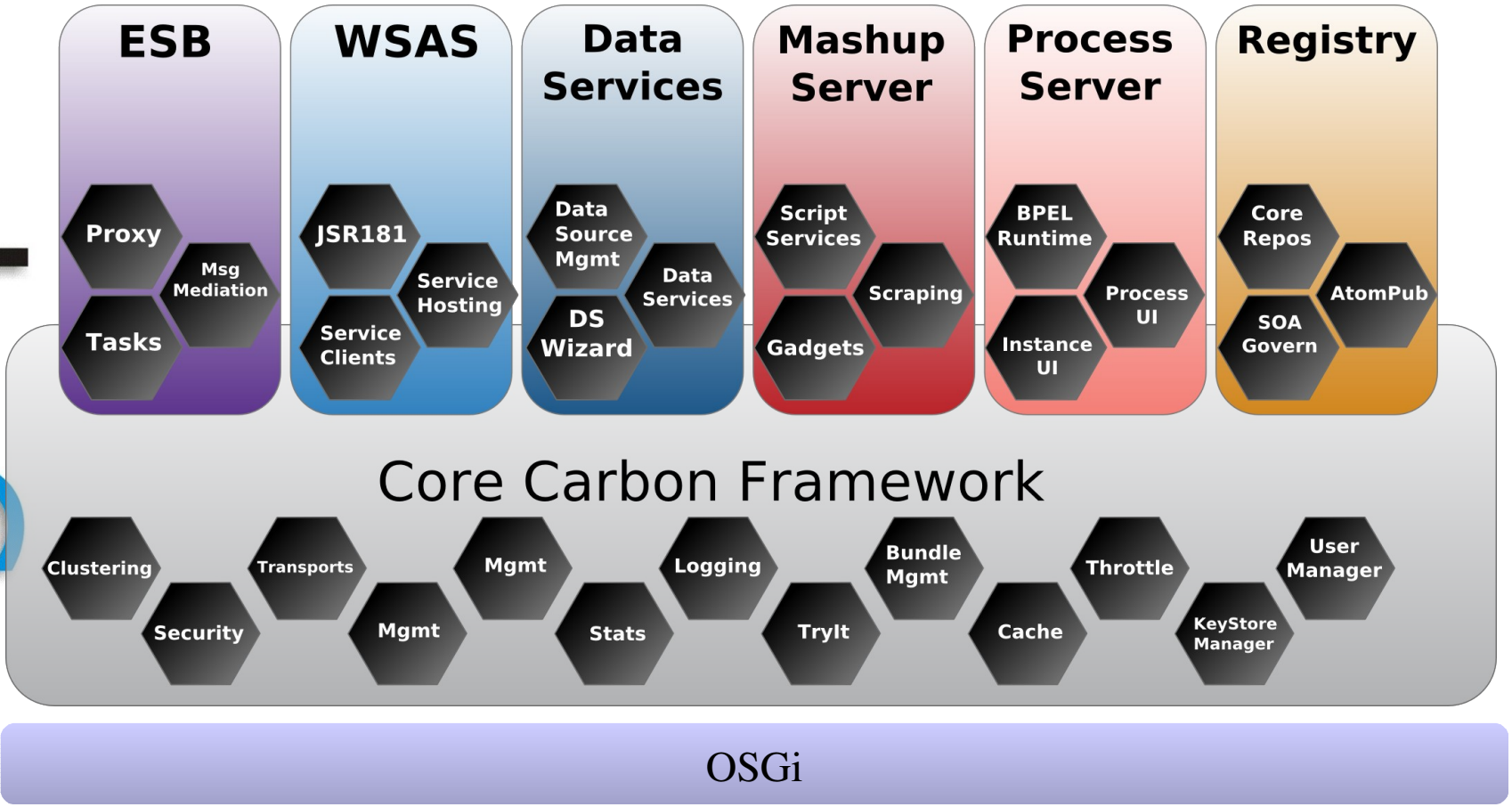
Why OSGi...

- **OSGi services provides in-VM collaborative SOA model**
  - Each bundle can provide services to other bundles
  - Services are simply Java objects that implement a given interface
  - There is a Service Registry where you can find a service implementation
  - Services should be only links between components
  - Best known way to decouple components from each other.
  - This allows components to be substituted.
  - Significantly minimize class loading problems

Why OSGi..

- **It's Dynamic**
  - The OSGi component model is a dynamic model
  - Bundles can be installed, started, stopped, updated, and uninstalled without shutting down the whole system.

What is WSO2 Carbon?



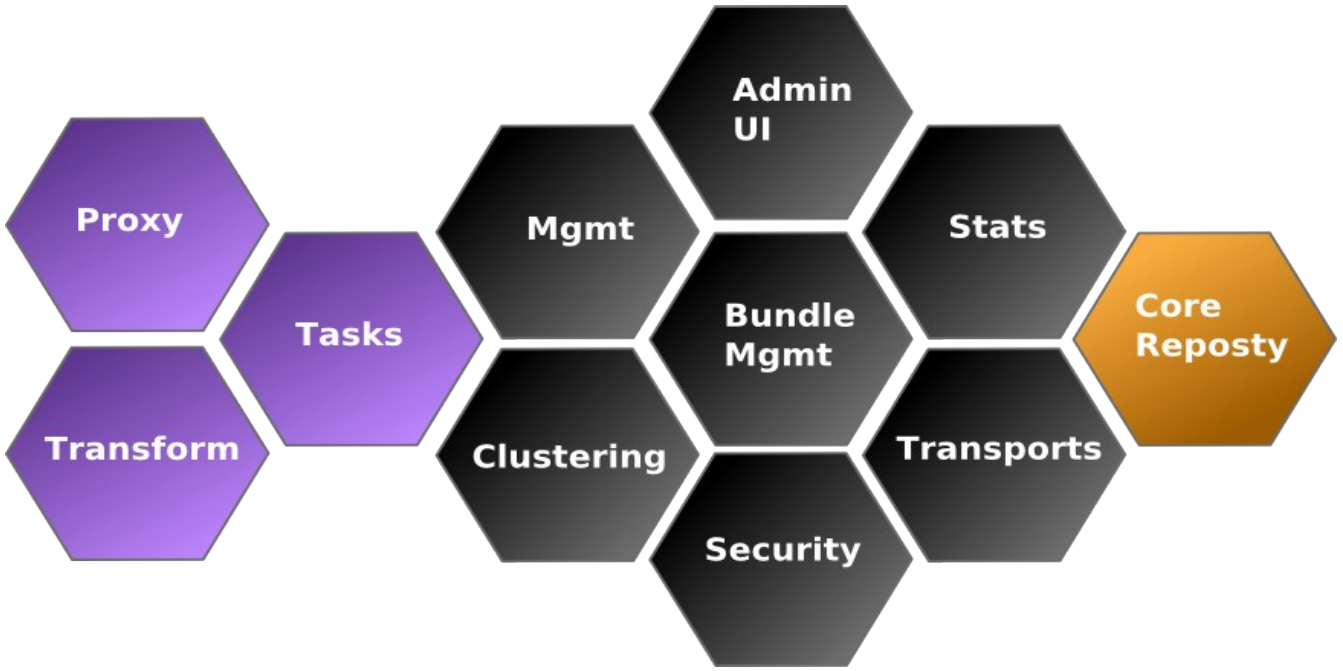
ApacheCon

What is WSO2 Carbon?

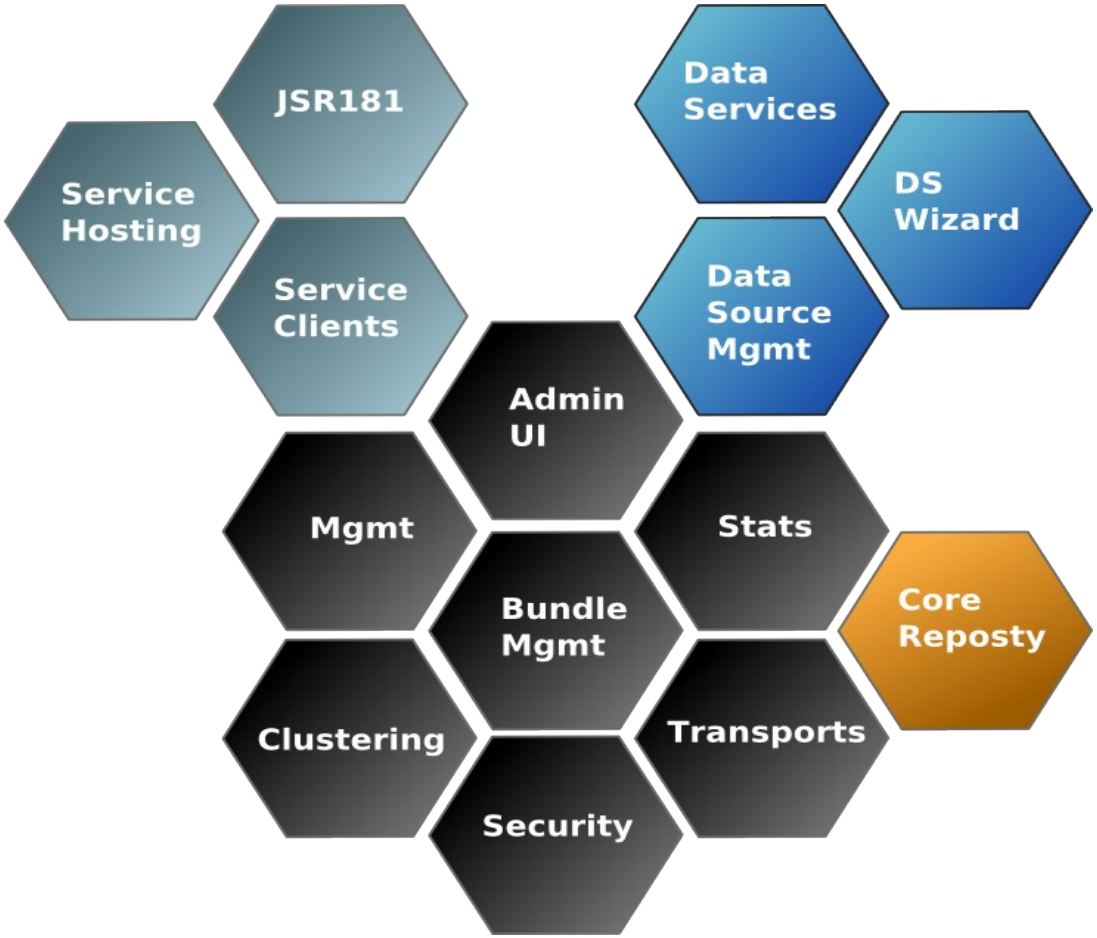
- A framework based on OSGi
  - Uses Equinox by default
- Still very closely based on Apache technology
  - Apache Axis2, Apache Synapse, Apache ODE
  - Apache Tomcat, Apache Axiom, and many other core libraries
- A well defined component model for Enterprise Middleware

Core Carbon Framework

Enterprise Service Bus

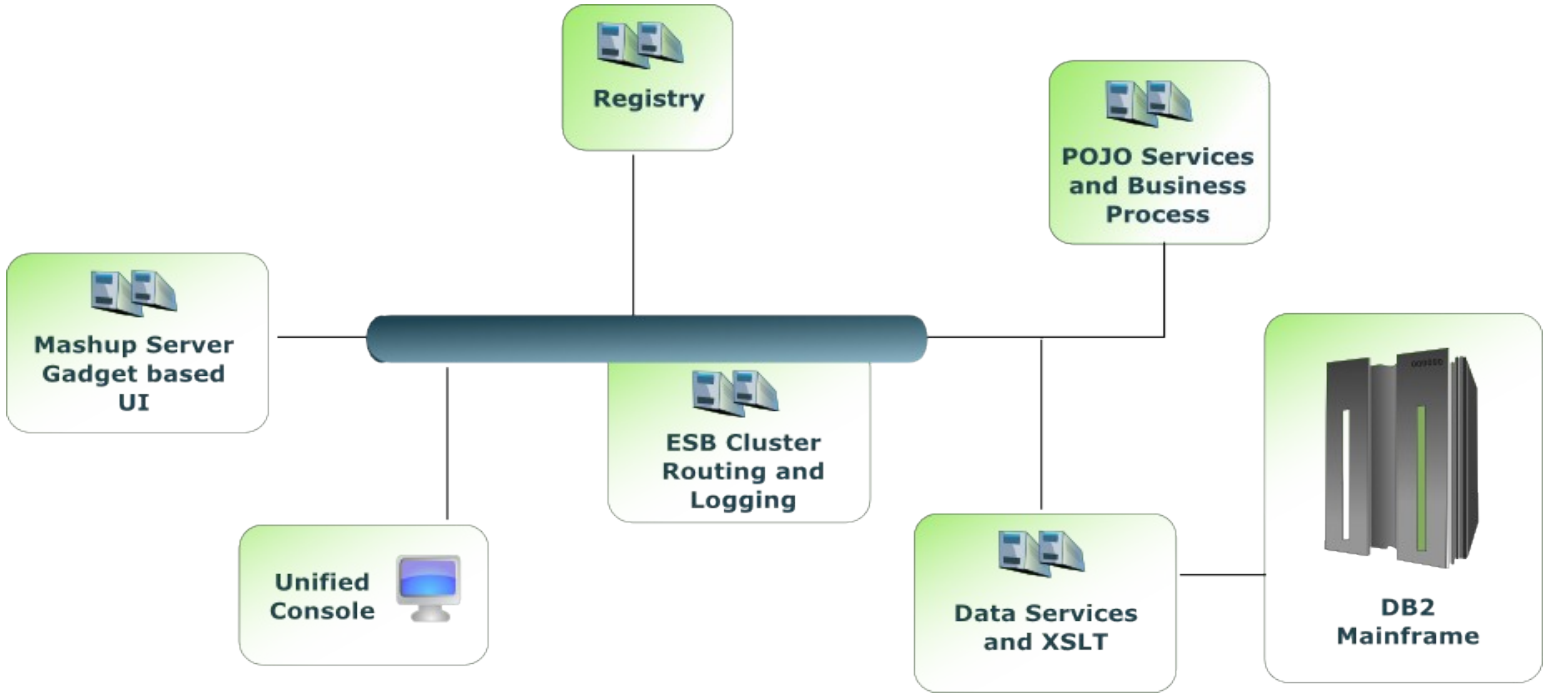


Web Services Application Server

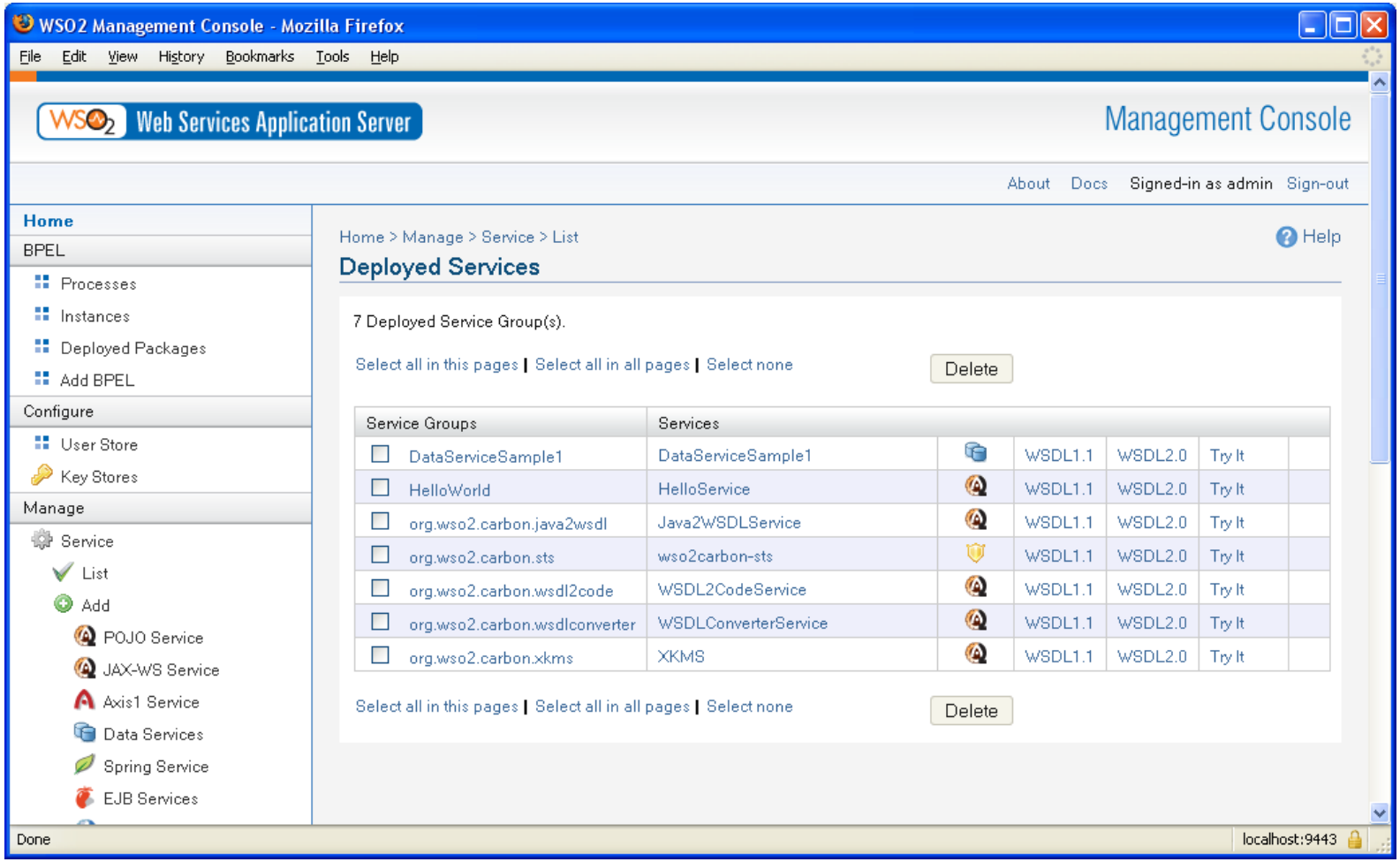


The Full Component Repository

A distributed example



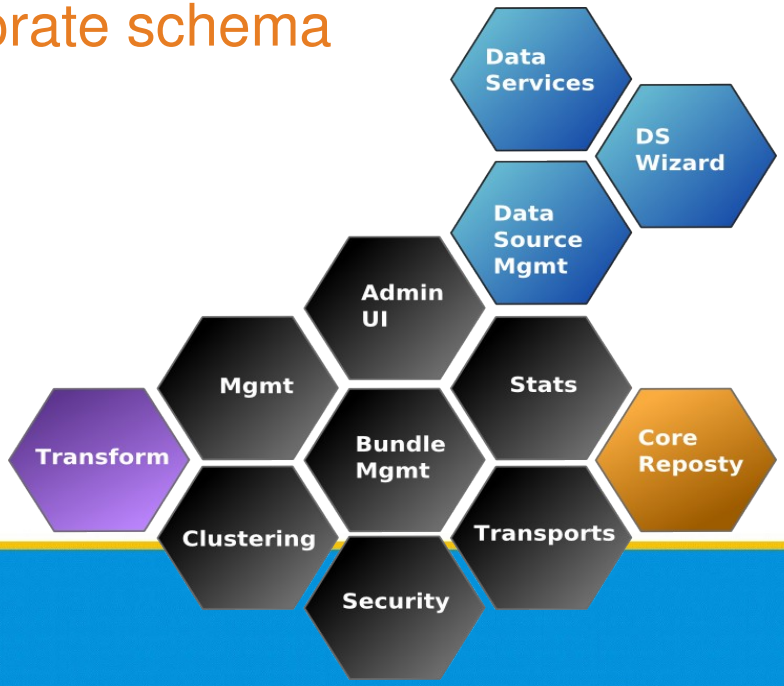
Unified management console



Can be run independently and connect to any JVM instance

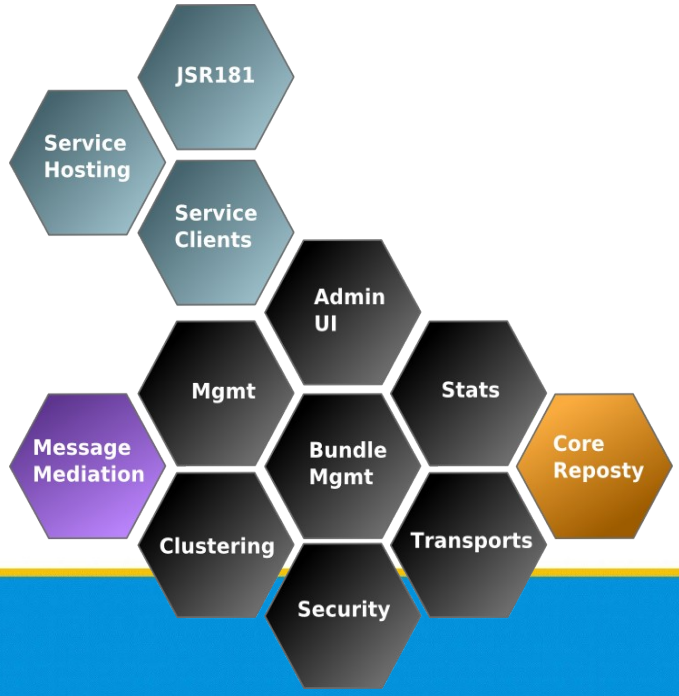
Use case: Data services & mediation

- Daniel is a Database Administrator
  - He has used WSO2 Data Services to expose his Data as Secure Web Services across the Enterprise
  - Now he has been asked to implement an existing schema
  - By adding the Mediation component and using XSLT or E4X he can effectively convert the existing service interface to match the corporate schema



Use case: POJO service and logging

- Helen is a POJO Service developer
  - She has an existing POJO service
  - The analytics department would like her to send a copy of every service request to them
  - She could write a new Handler in Java, but she discovers it takes 5 minutes to deploy and test the ESB Clone Mediator



What is right with Carbon?

Agile Adoption	• Start with the components you need, and grow as needed
Skills	• Based on industry standards • Common patterns can be applied across products
SOA Deployment	• Choose the components you need • Total platform is <150Mb (1/4 the size of Oracle) • Designed for Distributed Deployment with a central registry
Continuity	• Can install new function into an existing deployment • The same Admin UI works for all products
Product Integration	• The whole platform was designed to be consistent • The products integrate using the Registry out-of-the-box

Powered by OSGi

- OSGi is the underlying core modularization technology
  - Shipping with Eclipse Equinox by default
  - Can be supported on Spring dm Server, Felix and Knoplerfish if required
- Uses the core OSGi approach to support plugging in new functions in a managed way
- Any OSGi bundle can be deployed (e.g. ActiveMQ)
- New components can be deployed into an existing installation
- Used for both server runtime as well as for componentized management console
- Customers can write and deploy their own OSGi components

But more than just OSGi

- Analogies:
  - Eclipse uses OSGi but is much more – it defines how you plug Tool components into a Tooling Framework
  - Microsoft Office uses DLLs, but is much more – it defines how you can share a graph component across multiple document types
- Carbon uses OSGI, but it also defines how you build a consistent SOA platform
  - New service types plug into the console
  - Security, throttling, stats, TryIt, all work on any service component
- Not just using OSGi for componentizing a single product, but rather *entire* middleware platform

Challenges

- Difficult to convert existing code into OSGi bundles.
  - Legacy jars use class loaders heavily, hence most likely to fail in an OSGi environment.
- Getting the “start level” of bundles right is hard.
- OSGi originally designed for embedded devices
  - Needs little more tuning to fit into the server runtime.
- No support for the new servlet-api specs even in the latest OSGi R4 spec
- No built in support for JSPs in the model
- Primitive tooling for debugging OSGi issues

Migrating to OSGi

- First create a big OSGi bundle including all your project code and dependent jars.
- Use a BundleActivator to call the main class.
- Get your project working.
- gradually break down the big bundle into manageable bundles.
  - remove all the dependent jars and use their OSGified versions.
  - If possible, break down your project into reusable components(bundles)
- You will get so many issues.
- Solve them and keep it working!!
- Use Ant/Maven bundle plugin to create bundles.

Use proper package imports

- Import all the packages your bundle needs
- Use version when importing packages
  - This avoids version conflicts.
  - Fixed versions vs. version ranges.
- User Import-Package instead of Require-Bundle
  - Import-Package can have many providers
  - Require-Bundle can only have a single provider.
- Avoid using Class.forName
  - a lots of ClassNotFoundExceptions.
  - Use OSGi services to handle these dynamic collaborations between bundles.
  - As a workaround for Class.forName, use DynamicImport-Package. But this is not the recommended approach.

Few more best practices..

- Avoid start ordering dependencies.
  - Starting order of bundle may differ on different framework.
  - Use ServiceTracker utility or Declarative Services
- Use Whiteboard pattern.

Summary

Adapt your middleware to your enterprise architecture,
not your architecture to the middleware

Thank you!

Questions??