

OSGI

Bundle Advance:

- Background: Classes and Class Loaders
- Typical Class Loader Delegation
- Bundle Class Loading
- Bundle Dependencies
- A New Level of Visibility
 - Require-Bundle vs. Import-Package
 - Bundle Version Numbering

Service Layers – SCR

- Service Component
- Declarative Services & Extension Points
- Dynamic Services
- Dynamic Services Big Picture

Service Layers – SCR

- Service Providers
- Service Consumers
- Using Service Trackers

HTTP Service

- Web container embedded in Container
- Embed Osgi Container inside Web Container
- Advantages and disadvantages of both these architectures

Labs:

- Import / Required / Version
- Declarative Services - component I
- Service Tracker - component II
- Application using HTTPService
- Embed Osgi application in Web Container


Bundle Advance

What is Classloading?

- Classloaders are Java objects
- They are responsible for loading classes into the VM
 - Every class is loaded by a classloader into the VM
 - There is no way around
- Every class has a reference to its classloader object
 - **`myObject.getClass().getClassLoader()`**
- Originally motivated by Applets
 - To load classes from the server into the browser VM

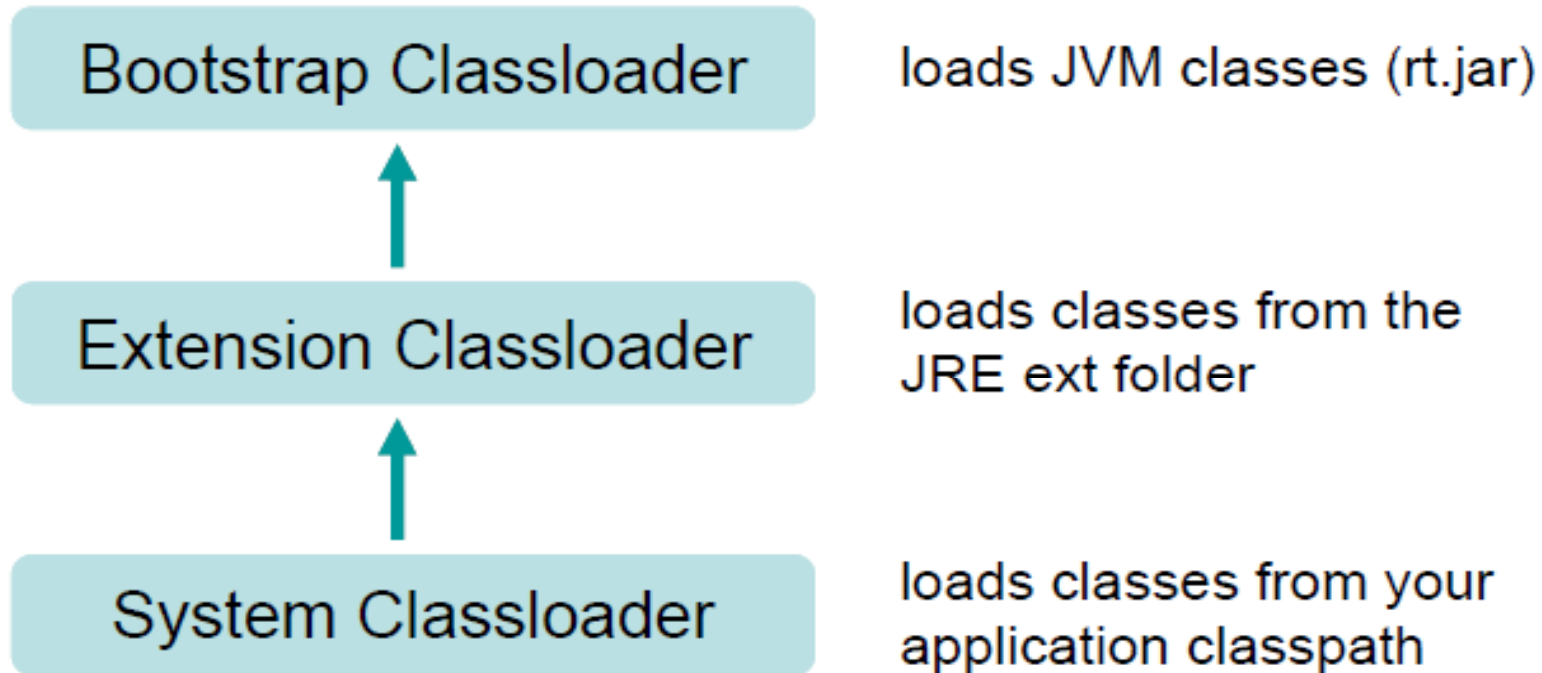
Implicit class loading

```
public class A {  
    public void foo() {  
        B b = new B();  
        b.sayHello();  
    }  
}
```



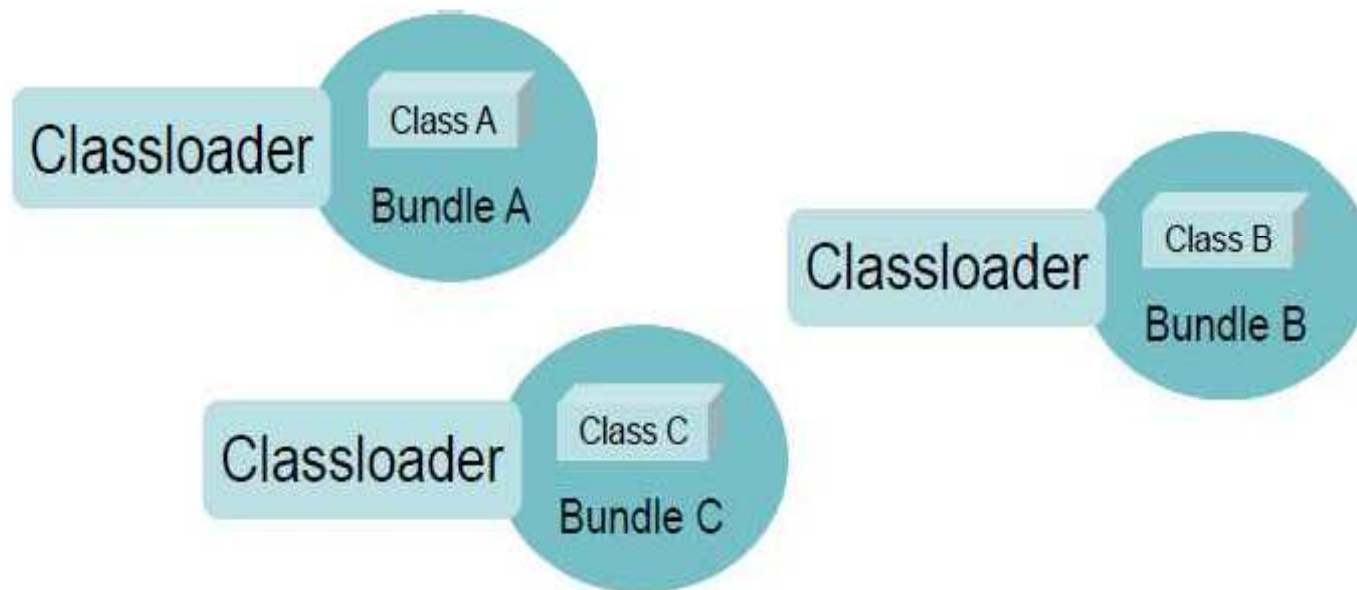
causes the VM to load
class B using the
classloader of A

The default setting

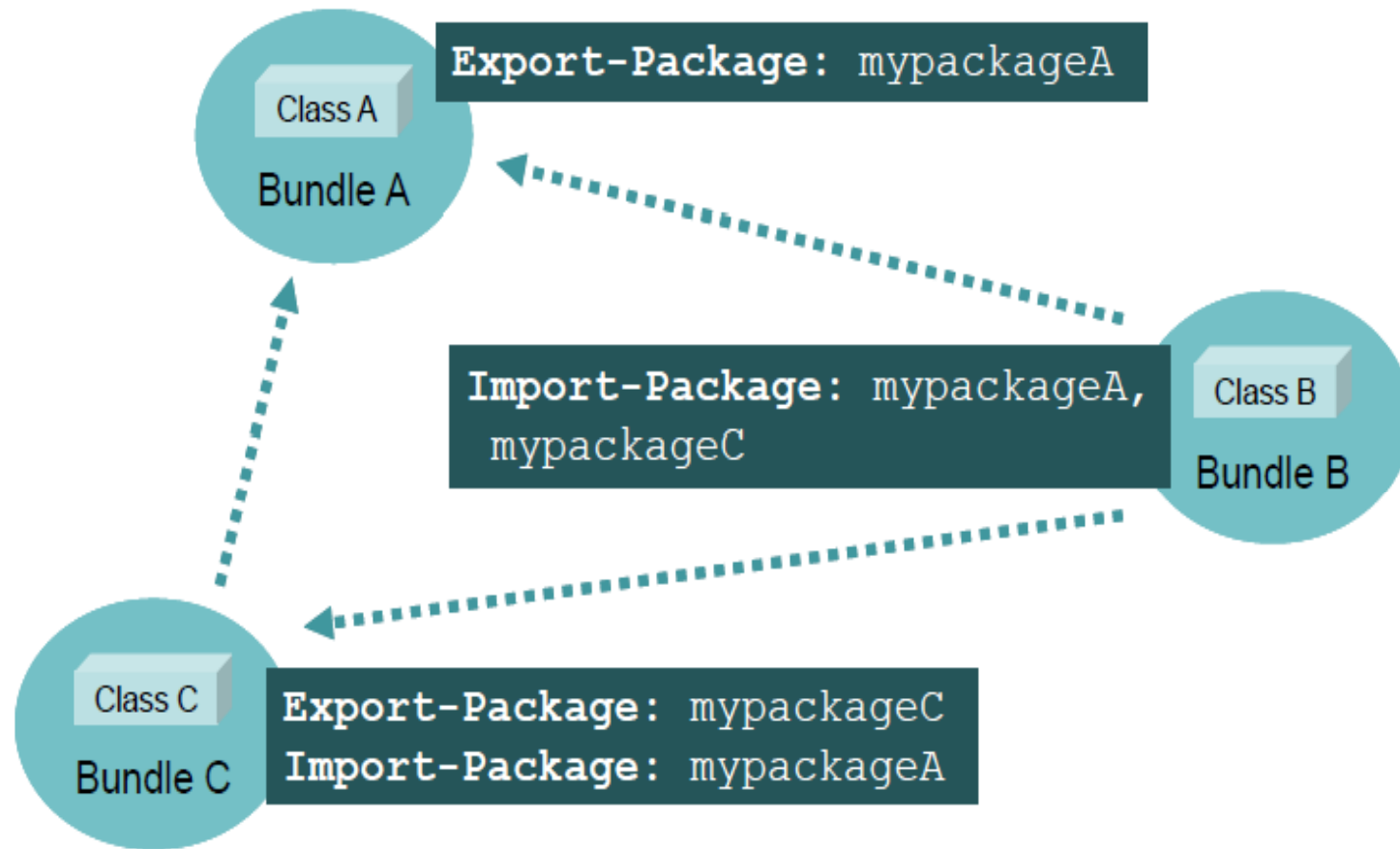


Classloader per bundle

- One classloader per bundle
 - ♦ Controls what is visible from the bundle
 - ♦ Controls what is visible from other bundles



Dependencies via delegation



The loading sequence

1. Try the parent for “java.” packages
2. Try the parent for boot delegation packages
3. Try to find it from imported packages
4. Try to find it from required bundles
5. Try to find it from its own class path
6. Try to find it from dynamic import

Restricting a bundle's export contract : Keeping packages private

Bundle-ClassPath: .,other-classes/,embedded.jar

Manifest-Version: 1.0

Bundle-ManifestVersion: 2

Bundle-Name: HelloWorldOSGi

Bundle-SymbolicName: HelloWorldOSGi

Bundle-Version: 3.0.0

Bundle-Activator: helloworldosgi.Activator

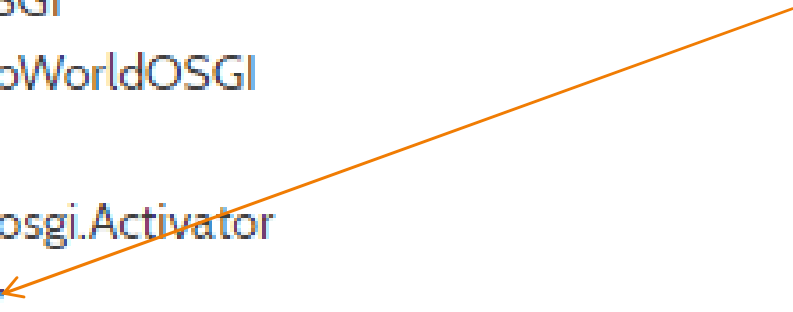
Bundle-ActivationPolicy: lazy

Bundle-RequiredExecutionEnvironment: JavaSE-1.6

Bundle-ClassPath: .,qciwsclient.jar

Import-Package: org.osgi.framework;version="1.3.0"

Setting this to *lazy* will tell the OSGi runtime that this plug-in should only be activated if one of its components, i.e. classes and interfaces are used by other plug-ins.



The provider bundle export the Java packages that are meant to be shared, and then have the consumer bundle import the Java packages that it needs.

To export Java packages

```
Manifest-Version: 1.0  
Bundle-ManifestVersion: 2  
Bundle-SymbolicName: helloworld  
Export-Package: manning.osgi.helloworld
```

To import a Java package

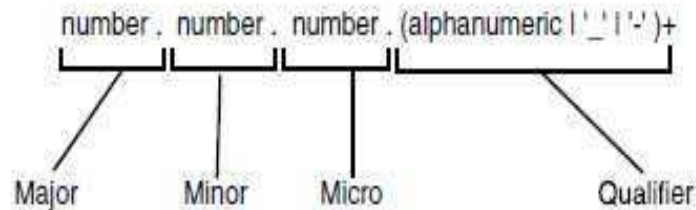
```
Manifest-Version: 1.0  
Bundle-ManifestVersion: 2  
Bundle-SymbolicName: manning.osgi.client  
Import-Package: manning.osgi.helloworld
```

Versioning bundles

OSGi provides native support for bundle versioning.

With OSGi, a bundle can specify a version for each package being exported, and conversely a bundle can specify version ranges when importing packages.

OSGi defines a version as a value of four parts:
major, minor, micro, and qualifier



OSGi defines a version as a value of four parts: major, minor, micro, and qualifier.

The first three take numeric values.

The last part takes alphanumeric digits, or `_`, or `-`.

By default, zero is used for the first three parts.

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: foo
Bundle-Version: 1.0.0
Export-Package: org.foo;version:="1.0.0.r4711"
```

```
Import-Package: org.foo;version:="[1.0,2.0)"
```

Restricting a bundle's export contract : Excluding classes from an exported package

```
Export-Package: manning.osgi.test;  
include:="Foo*, Bar"; exclude:=FooImpl
```

If more than one bundle needs to use a class, then the proper solution is indeed to create a bundle and export it. But you don't need to export the full package where the class resides; instead, you can pick and choose which classes are to be exported from a package. This is done using the `include` and `exclude` parameters of the `Export-Package` manifest header.

Require-Bundle

Bundle-SymbolicName: B1

Require-Bundle: B2; visibility:=reexport

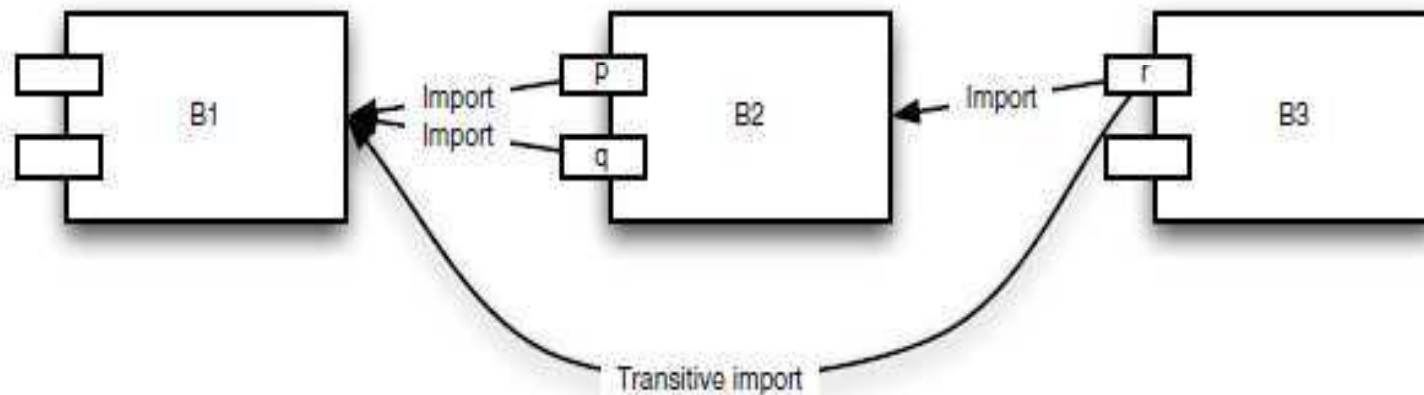


Figure Bundle B1 specifies bundle B2 as a required bundle. B1 imports B2's exported packages and B2's imported packages from bundle B3.

Dynamic imports

The OSGi framework defines the manifest header `DynamicImport-Package`, which can be used to specify Java packages that can be dynamically searched at runtime to load Java classes.

`DynamicImport-Package: org.apache.derby.jdbc`

Optional packages

In OSGi, you can specify a package that's being imported as optional. This means that the framework will attempt to wire this class during the resolve process, as the bundle is being installed, but it won't consider it an error if the package isn't found.

Import-Package:

```
org.apache.derby.jdbc;resolution:=optional
```

Fragment bundles

Fragment bundles are degenerated bundles that need to attach to a host bundle in order to function.

```
Fragment-Host: manning.osgi.federated-  
database
```

Tutorial: Import / Required / Version.

Component Oriented Development in OSGi

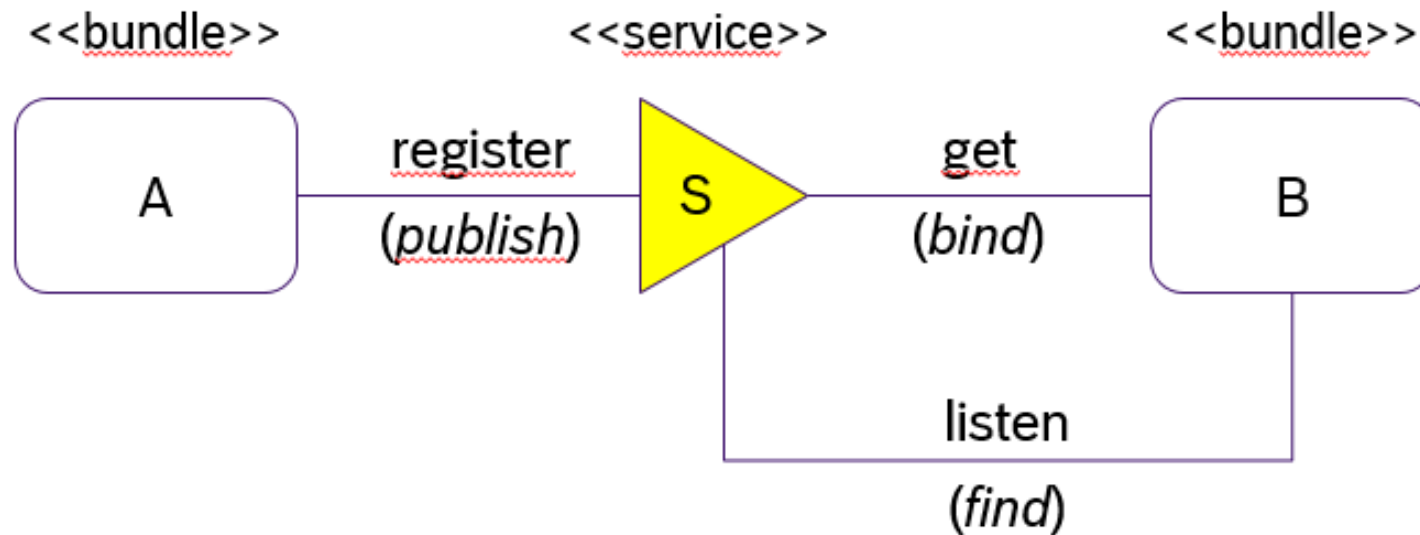
Declarative Services

The (Partial) Failure of Object Oriented

- One of the primary goals of object oriented programming (OOP) was, and still is, re-use.
- It has mostly failed in that goal.

Component Frameworks

- We would like to use a framework to ease implementation of components in OSGi.
- There is more than one to choose from!
- three popular choices:
 - Declarative Services;
 - Spring Dynamic Modules
 - Apache iPOJO.



This contract is implemented by the OSGi ServiceRegistry.

Declarative Services

With declarative services it is not necessary to register or consume services programmatically

Declarative Services

- There are significant improvements in DS in OSGi R4.2.
- Many of these changes are supported in Equinox 3.5M5+.
- We will use the R4.2 features, but mention when we do so.

Declarative Services

What does the SCR extender bundle do on our behalf?

- 1 Creates Components.
- 2 "Binds" them to services and configuration.
- 3 Manages the component's lifecycle in response to bound services coming and going.
- 4 Optionally, publishes our components as services themselves.

A Minimal Component Declaration

minimal.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">

  <implementation class="org.example.osgi.ds.HelloComponent"/>

</scr:component>
```

NB: the namespace is required in order to use R4.2 features. If not included then SCR will default to the prior version.

A Minimal Component Declaration

MANIFEST.MF

```
Bundle-SymbolicName: mybundle  
Bundle-Version: 1.0.0  
Service-Component: OSGI-INF/minimal.xml  
...
```

A Minimal Component Declaration

HelloComponent

```
package org.example.osgi.ds;  
  
public class HelloComponent {  
    public HelloComponent() {  
        System.out.println("HelloComponent created");  
    }  
  
    // ...  
}
```

Note: a Plain Old Java Object (POJO)! No OSGi API dependencies.

Building the Bundle

Internal Bundle Structure

```
minimal_ds.jar/  
  META-INF/  
    MANIFEST.MF  
  OSGI-INF/  
    minimal.xml  
  org/  
    example/  
      osgi/  
        ds/  
          HelloComponent.class
```

Running the Example

- We need to install the SCR bundles : Core and Util
- The SCR bundle needs to be started, but the util bundle does not.

Activation and Deactivation

- That was a very long-winded way to merely instantiate a class!
- This component is not useful because it cannot even do anything.
- However, DS allows us to define lifecycle methods.
- We can be notified when the component starts and stops.
- This allows us to do interesting things like start threads, open sockets, etc.

An Active Component

PollingComponent

```
public class PollingComponent {  
  
    private static final int DEFAULT_PERIOD = 2000;  
    private PollingThread thread;  
  
    protected void activate(Map<String, Object> config) {  
        System.out.println("Polling Component Activated");  
        Integer period = (Integer) config.get("period");  
        thread = new PollingThread(  
            period != null ? period : DEFAULT_PERIOD);  
        thread.start();  
    }  
    protected void deactivate() {  
        System.out.println("Polling Component Deactivated");  
        thread.interrupt();  
    }  
}
```

Activation and Deactivation

- Why didn't we just write this as a bundle activator??
 - This is a POJO!
 - Easier access to configuration { the Map parameter to activate.
 - Easier access to service registry { we will see this soon.

References to Services

- Using lower level APIs we must write a lot of “glue” code to bind to services.
- DS replaces the glue code with simple declarations.
- We declare references to services.

References to Services

Example reference Element

```
<reference name="LOG"
  interface="org.osgi.service.log.LogService"
  bind="setLog" unbind="unsetLog"/>
```

Reference Attributes

name	The name of the reference.
interface	The service interface name.
bind	The name of the "set" method associated with the reference.
unbind	The name of the "unset" method associated with the reference.

Optional Service Reference

Example reference Element

```
<reference name="LOG"  
  interface="org.osgi.service.log.LogService"  
  bind="setLog" unbind="unsetLog"  
  cardinality="0..1"/>
```

- The default cardinality was "1..1" meaning that we must have exactly one instance, i.e. the reference is mandatory.
- A cardinality of "0..1" indicates that either zero or one instance is okay, i.e. the reference is optional.
- Yes, "0..n" and "1..n" do exist.

Publishing a Service

- The final piece of the puzzle is publishing our component as a service itself.
- This is done with the `<service>` element in our XML descriptor.

Service Element

```
<service>  
  <provide interface="net...ContactRepository"/>  
</service>
```

- Provide multiple services simply by adding additional `<provide>` elements.

Service Properties

We can specify service properties using the `<property>` element. These properties are passed to the component in activation *and* published to the service registry.

Property Element

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">
  <implementation class="..." />
  <property name="foo" value="bar" />
  ....
</scr:component>
```

Lazy Service Creation

- By default, SCR creates “delayed services”.
- These are services that are registered in the service registry but the implementation object has not yet been instantiated.
- It will be instantiated “on-demand” when the first client attempts to actually use the service.

Immediate Services

- Delayed creation is the default for services, but it can be turned off.
- Service components can be declared with `immediate="true"` to make SCR instantiate them immediately (assuming their dependencies are satisfied).
- Non-service components are always “immediate”.

- Service API
- Service Provider
- Service Consumer

Components are created using *Declarative Services* (DS) using annotations – Interpret it to XML by bndtools

- This annotation identifies the annotated class as a service component.

```
@Component(  
    name = "ClusteringAgentServiceComponent",  
    immediate = true,  
    property = "Agent=hazelcast"  
)
```

- The @Reference annotation should be applied to a method which will be used as a “bind” method of a service component.
- references to other services made available to the component by the Service Component Runtime.

```
@Reference(  
    name = "http.service",  
    service = HttpService.class,  
    cardinality = ReferenceCardinality.MANDATORY,  
    policy = ReferencePolicy.STATIC,  
    unbind = "unsetHttpService"  
)
```

@Activate, @Deactivate, @Modified

are used with the respective methods that will be called when the service component state changes from one to another.

```
import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;
import org.osgi.service.component.annotations.ReferenceCardinality;
import org.osgi.service.component.annotations.ReferencePolicy;
import org.osgi.service.http.HttpService;
import org.osgi.service.http.NamespaceException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.wso2.carbon.transport.servlet.SampleServlet;

import javax.servlet.ServletException;

/**
 * This service component is responsible for retrieving the HttpService
 * OSGi service and register servlets
 */
```

```
@Component(  
    name = "org.wso2.carbon.transport.HttpServiceComponent",  
    immediate = true  
)  
public class HttpServiceComponent {  
  
    private static final Logger logger = LoggerFactory.getLogger(HttpServiceComponent.class);  
  
    private HttpService httpService;  
  
    @Activate  
    protected void start() {  
        SampleServlet servlet = new SampleServlet();  
        String context = "/sample";  
        try {  
            logger.info("Registering sample servlet : {}", context);  
            httpService.registerServlet(context, servlet, null,  
                                     httpService.createDefaultHttpContext());  
        } catch (ServletException | NamespaceException e) {  
            logger.error("Error registering servlet", e);  
        }  
    }  
}
```

```
@Reference(  
    name = "http.service",  
    service = HttpService.class,  
    cardinality = ReferenceCardinality.MANDATORY,  
    policy = ReferencePolicy.STATIC,  
    unbind = "unsetHttpService"  
)  
protected void setHttpService(HttpService httpService) {  
    this.httpService = httpService;  
}  
  
protected void unsetHttpService(HttpService httpService) {  
    this.httpService = null;  
}  
}
```



```
<components xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">  
    <scr:component immediate="true" name="org.wso2.carbon.transport.HttpServiceComponent" activate="start">  
        <implementation class="org.wso2.carbon.transport.internal.HttpServiceComponent"/>  
        <reference name="http.service" interface="org.osgi.service.http.HttpService" cardinality="1..1"  
            policy="static" bind="setHttpService" unbind="unsetHttpService"/>  
    </scr:component>  
</components>
```

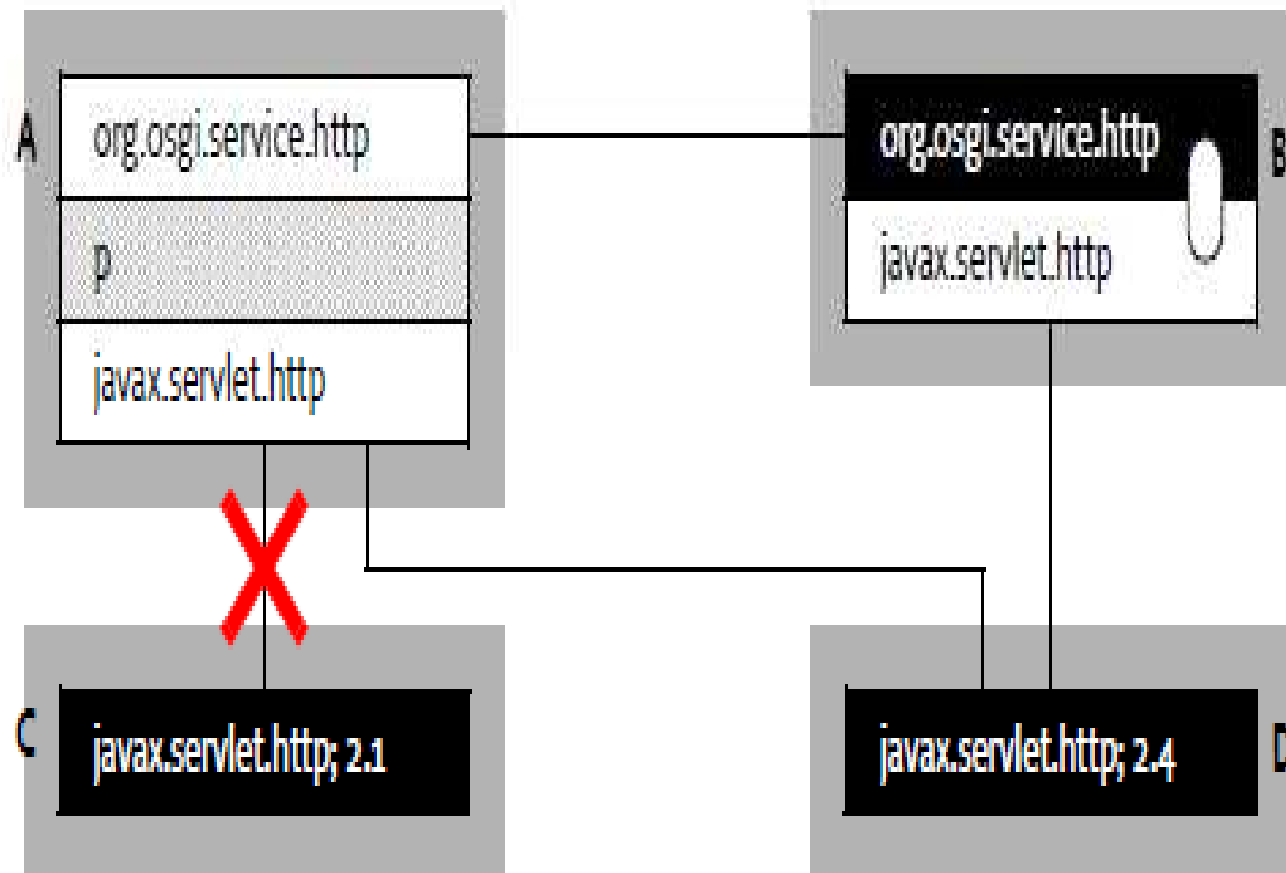
```
@Activate  
void activate( BundleContext context) {  
    this.context = context;  
}  
  
|  
@Deactivate  
void deactivate() {  
    if ( this.registration != null)  
        this.registration.unregister();  
}
```


Tutorial • Declarative Services

Advance OSGI I

- all classes reachable from a given bundle's class loader
- It can contains:
 - The parent class loader (normally java.* packages from the boot class path)
 - Imported packages
 - Required bundles
 - The bundle's class path (*private packages*)
 - Attached fragments

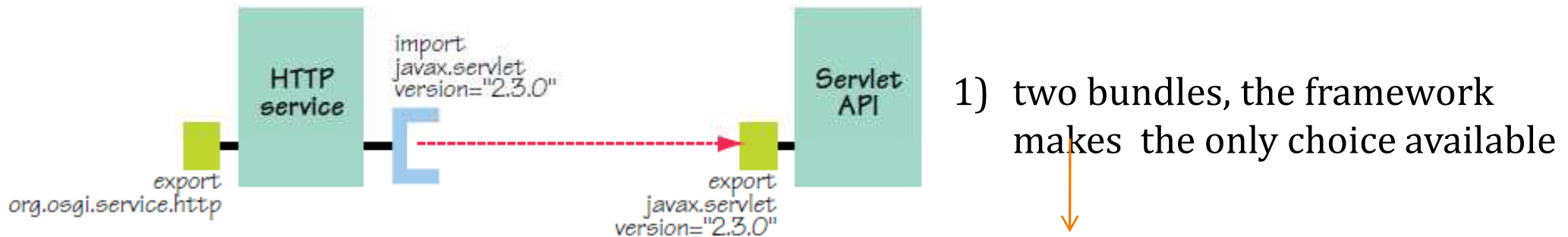
Uses directive in B, forces A to use javax.servlet.http from D



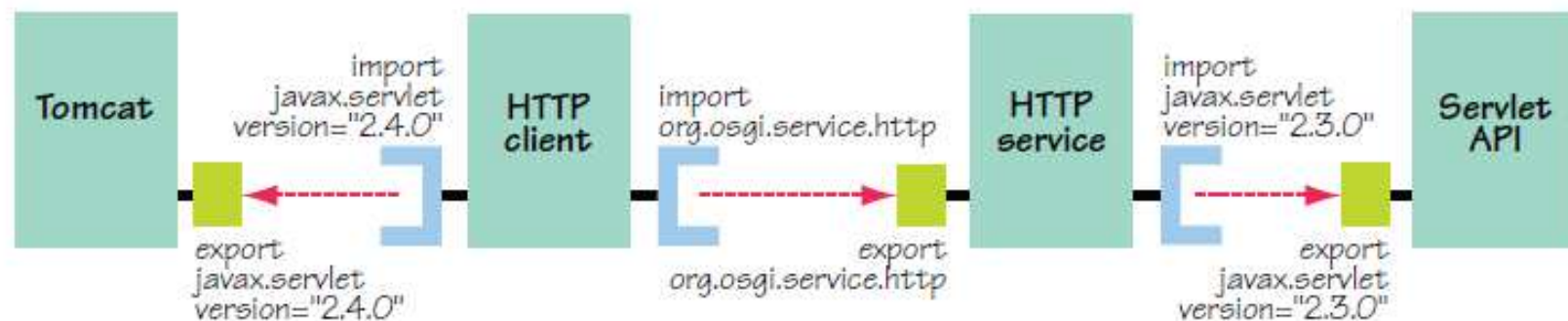
client-dependency resolution

```
package org.osgi.service.http;
import javax.servlet.Servlet;
public interface HttpService {
    void registerServlet(String alias, Servlet servlet, HttpContext ctx);
}
```

```
Export-Package: org.osgi.service.http; version="1.0.0"
Import-Package: javax.servlet; version="2.3.0"
```



2) install two more bundles into the framework



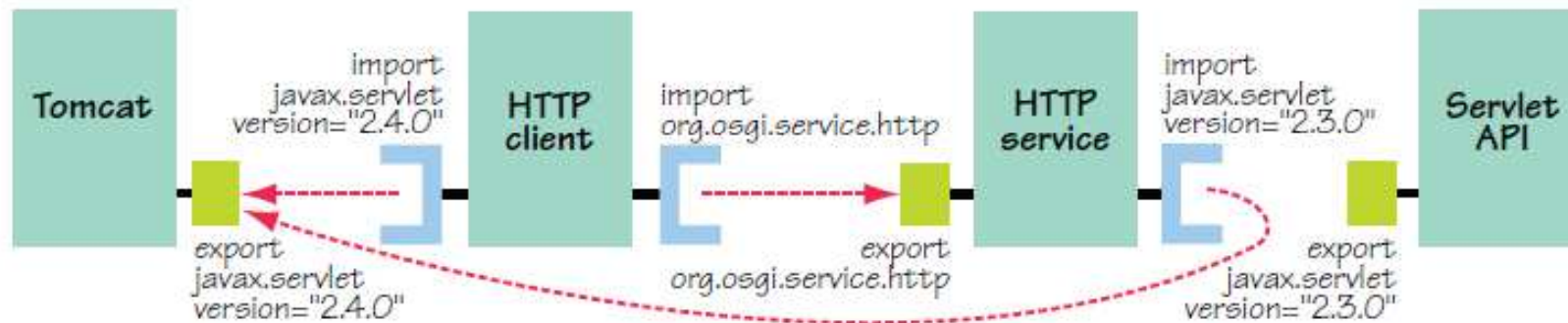
At execution time, this results in class cast exceptions when the HTTP service and client bundles interact

Consistent dependency

```
package org.osgi.service.http;  
import javax.servlet.Servlet;  
public interface HttpService {  
    void registerServlet(String alias, Servlet servlet, HttpContext ctx);  
}
```

Export-Package: org.osgi.service.http; version="1.0.0"

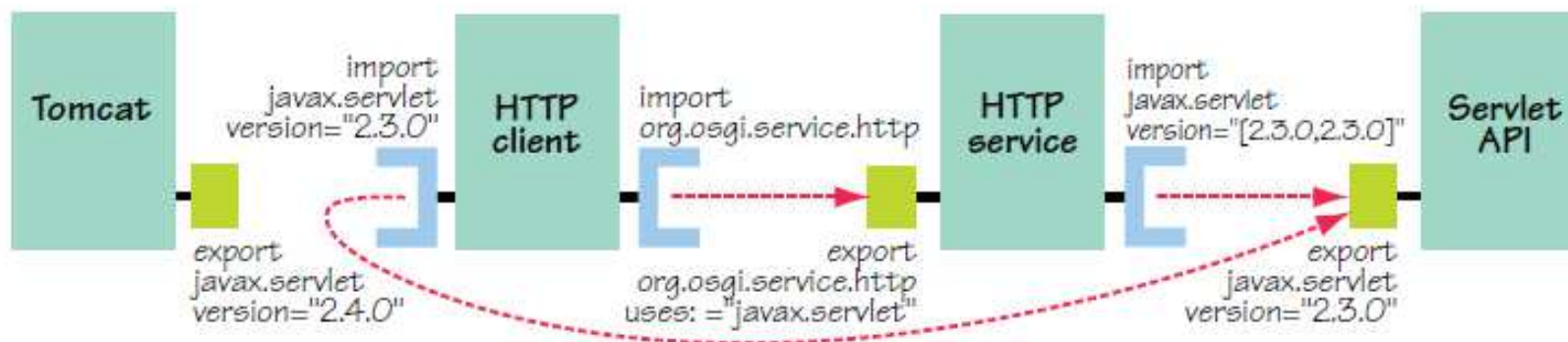
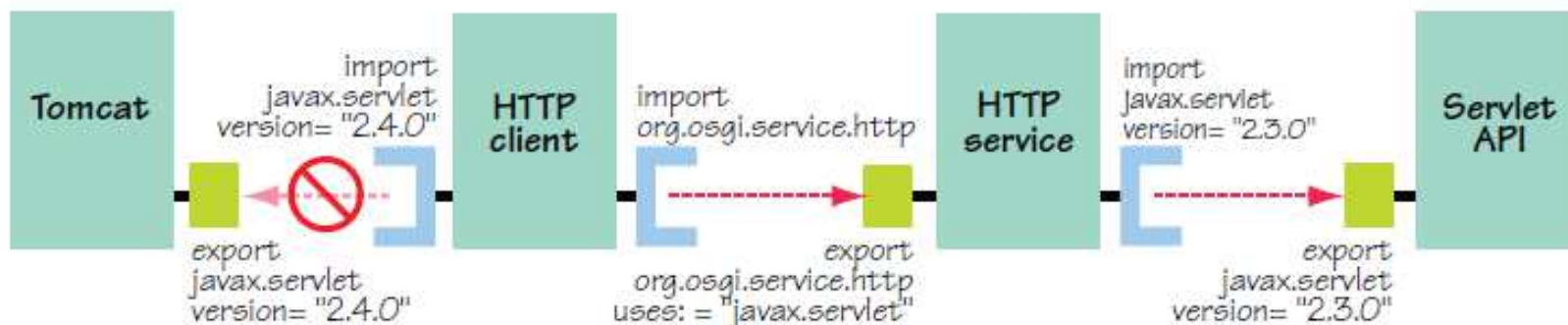
Import-Package: javax.servlet; version="2.3.0"



If you install all four bundles together, the framework resolves the dependencies in a consistent way using its existing rules

Consistent dependency - Uses

```
Export-Package: org.osgi.service.http;  
  uses:="javax.servlet"; version="1.0.0"  
Import-Package: javax.servlet; version="2.3.0"
```

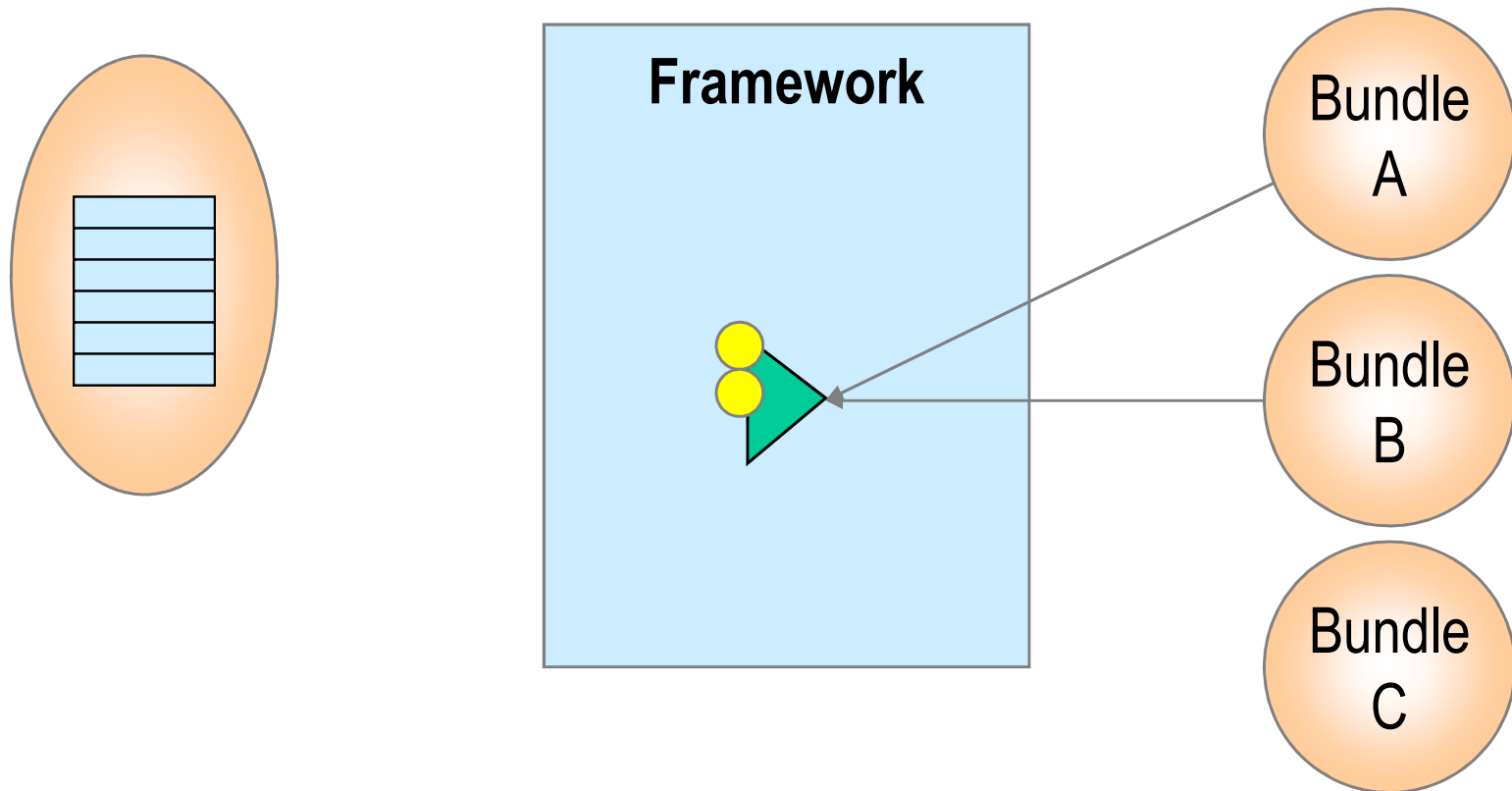


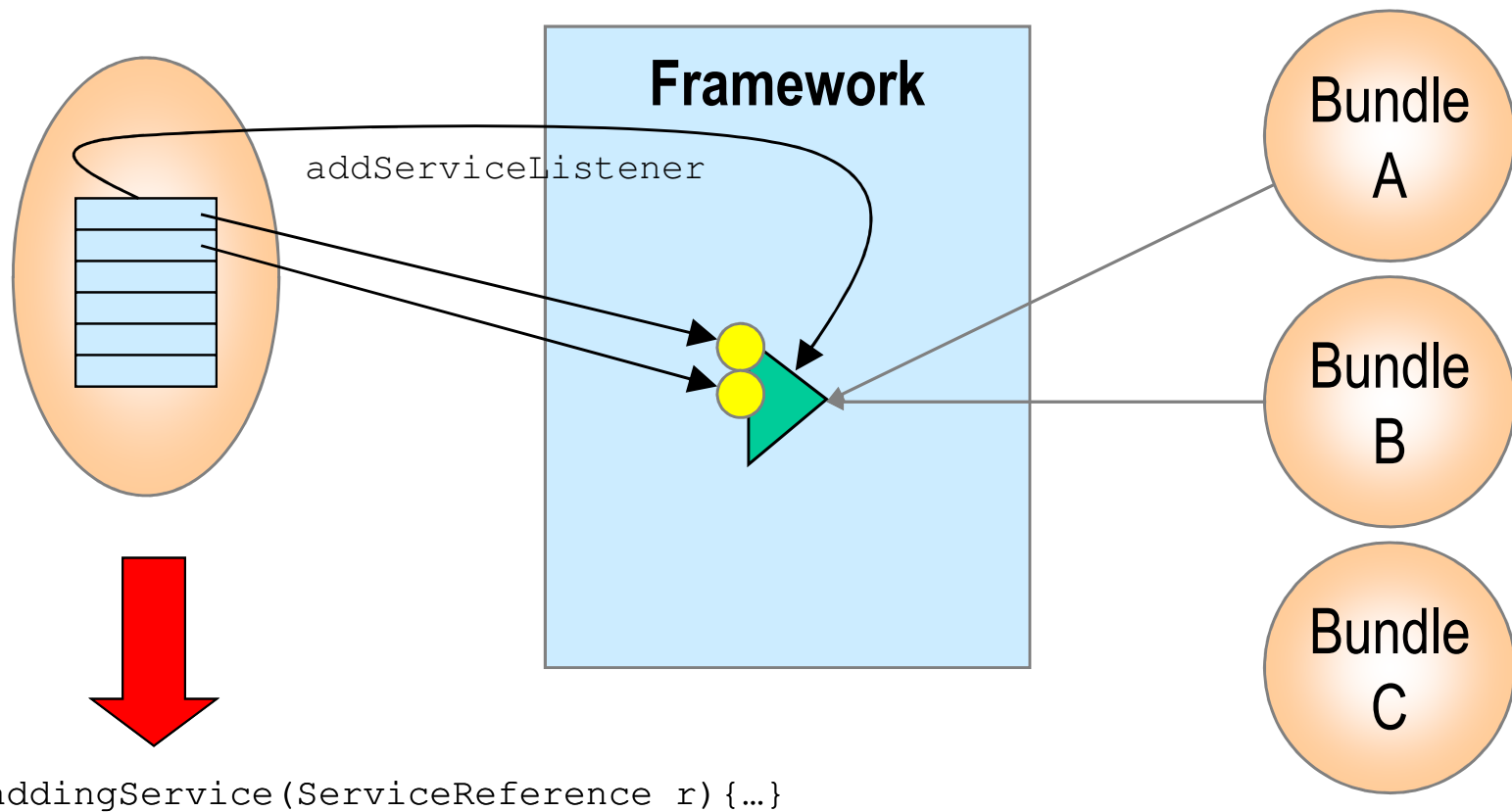
- OSGi services must always be considered volatile, and may disappear or become unusable at any time.
 - Always **expect RuntimeException** when calling a service
 - Use the white-board model
 - Use **ServiceTracker**

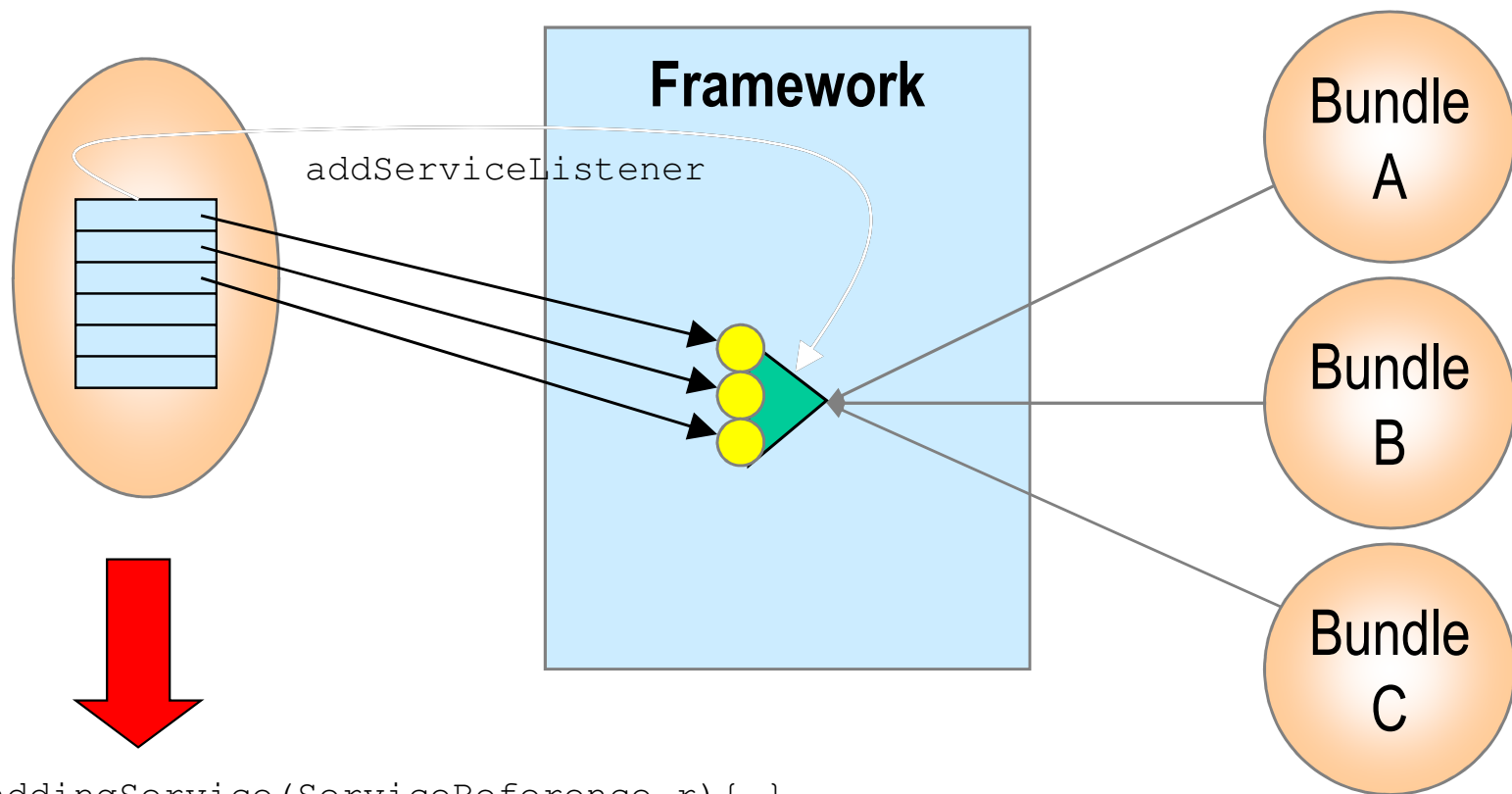

```
ServiceReference sr = bc.getServiceReference(HttpService.class.getName());  
HttpService http = (HttpService)bc.getService(sr);  
http.registerServlet(...)|
```

Using Service Trackers

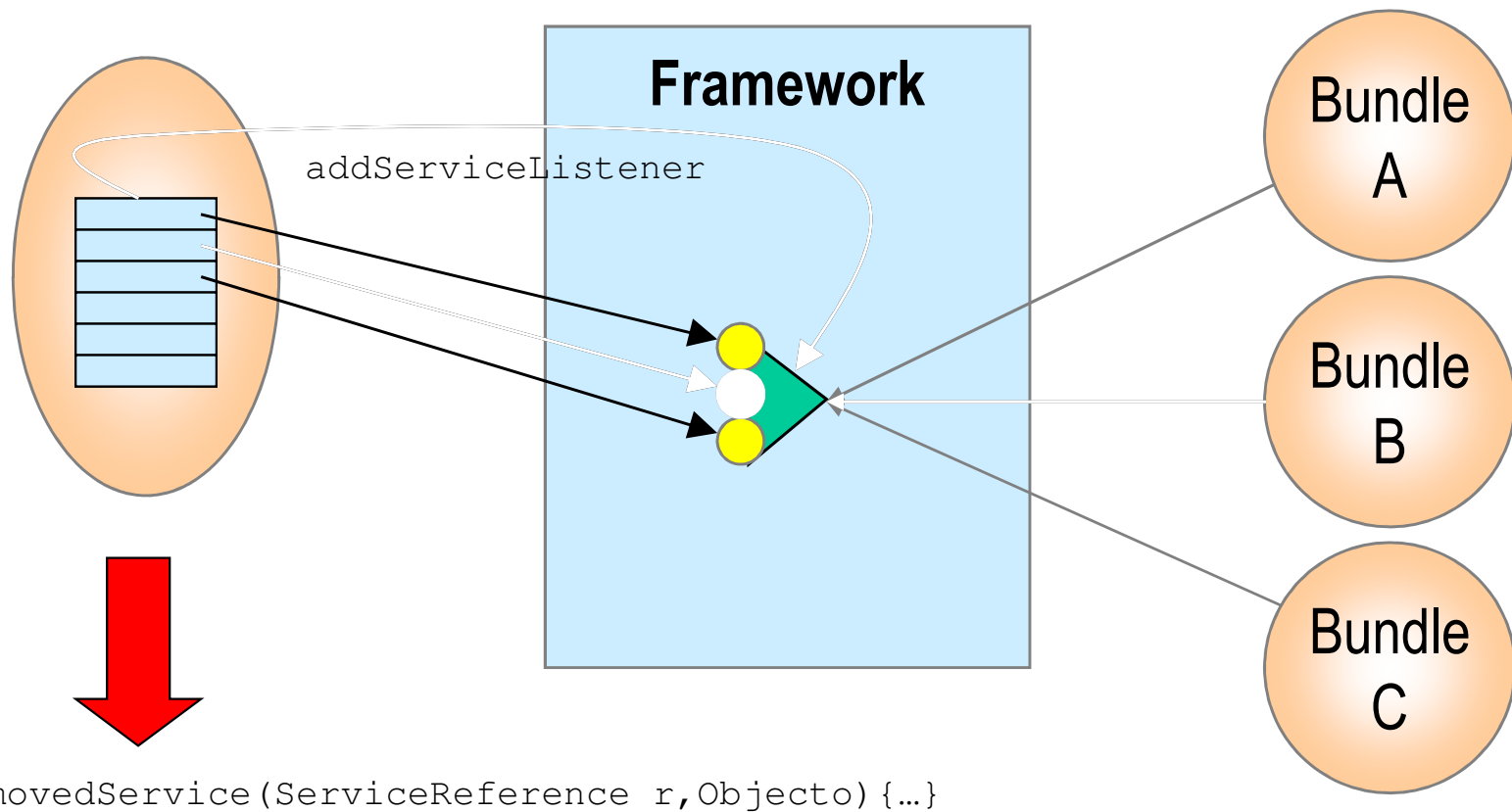
- Finding services for each message is kind of expensive.
- The ServiceTracker in org.osgi.util.tracker package is intended to simplify this task
- A service tracker maintains a list of services based on:
 - A filter
 - A specific class
- It reports any existing or new services as well as any services that become unregistered
 - Object addingService
 - void modifiedService
 - void removedService
- The service tracker is used to track channels and store them in a Map

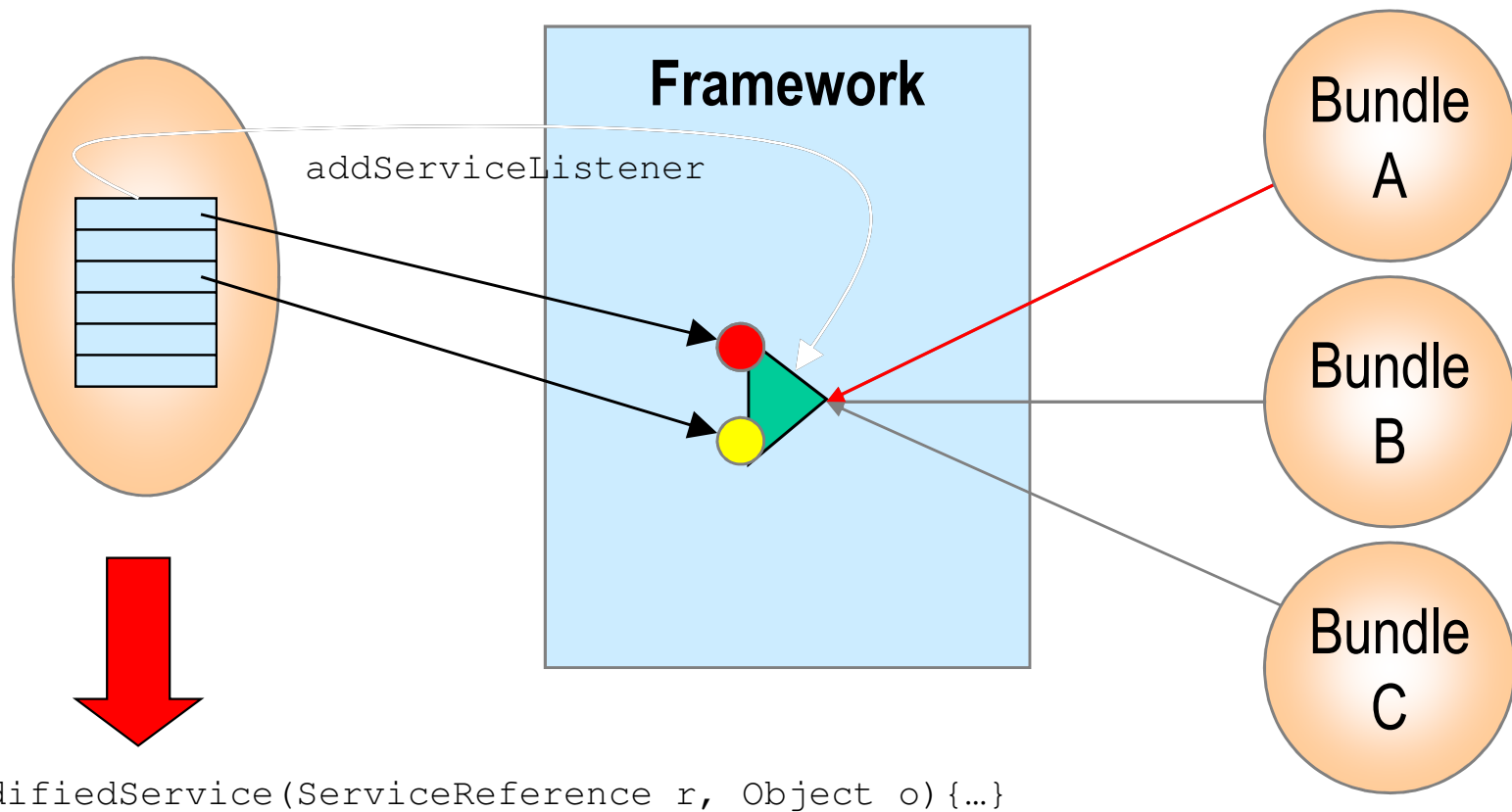






```
Object addingService(ServiceReference r) {...}
```





ServiceTracker example

```
ServiceTracker logTracker =  
    new ServiceTracker(bc,  
        LogService.class.getName(), null);  
  
logTracker.open();  
  
((LogService) logTracker.getService()).doLog(  
    ...);
```

The tracker guarantees to hold all currently available services

Tutorial: Service Tracker - component II

Compendium Services

- In addition to the core services, the OSGi Alliance defines a set of non-core standard services called the *compendium services*.
- *Whereas the core services are typically available* by default in a running OSGi framework, the compendium services aren't.
- these services are provided as separate bundles by framework implementers or other third parties and typically work on all frameworks.

- Log Service
- HTTP Service

Log Service

- provide a common logging service for an OSGi framework.
- two services:
 - service end
 - service provider end

Log Service provides a simple logging facade, with various flavors of methods

Log Interface

- `log(int level, String message, Throwable exception)`

Usage:

```
LogService log = null;  
ServiceReference ref = context.getServiceReference(  
    LogService.class.getName());  
if (ref != null)  
{  
    log = (LogService) context.getService(ref);  
}
```

Log Service Example

```
public class Activator implements BundleActivator {  
    LogService m_logService;  
  
    public void start(BundleContext context) {  
        ServiceReference logServiceRef =  
            context.getServiceReference(LogService.class.getName());  
  
        m_logService = (LogService) context.getService(logServiceRef);  
        startTestThread();  
    }  
    public void stop(BundleContext context) {  
        stopTestThread();  
    }  
}
```

Finds single best Log Service ←

← **Starts Log Service test thread**

← **Stores instance in field (bad!)**

`m_logService.log(LogService.LOG_INFO, "ping");`

HTTP Service

- HTTP Service
- Web Applications – (WAB).

HTTP Service

- A means for bundles to expose servlets or resources to be accessed through Http.
- to provide content in HTML, XML .
- The bundles register their content and servlets in a dynamic manner.

HttpService

- The HttpService implementation :
 - initialized the registered servlets.
 - interfacing with the outside world.
 - delegating requests to the corresponding servlet
 - providing the resulting content back to the requesting party.

HttpService

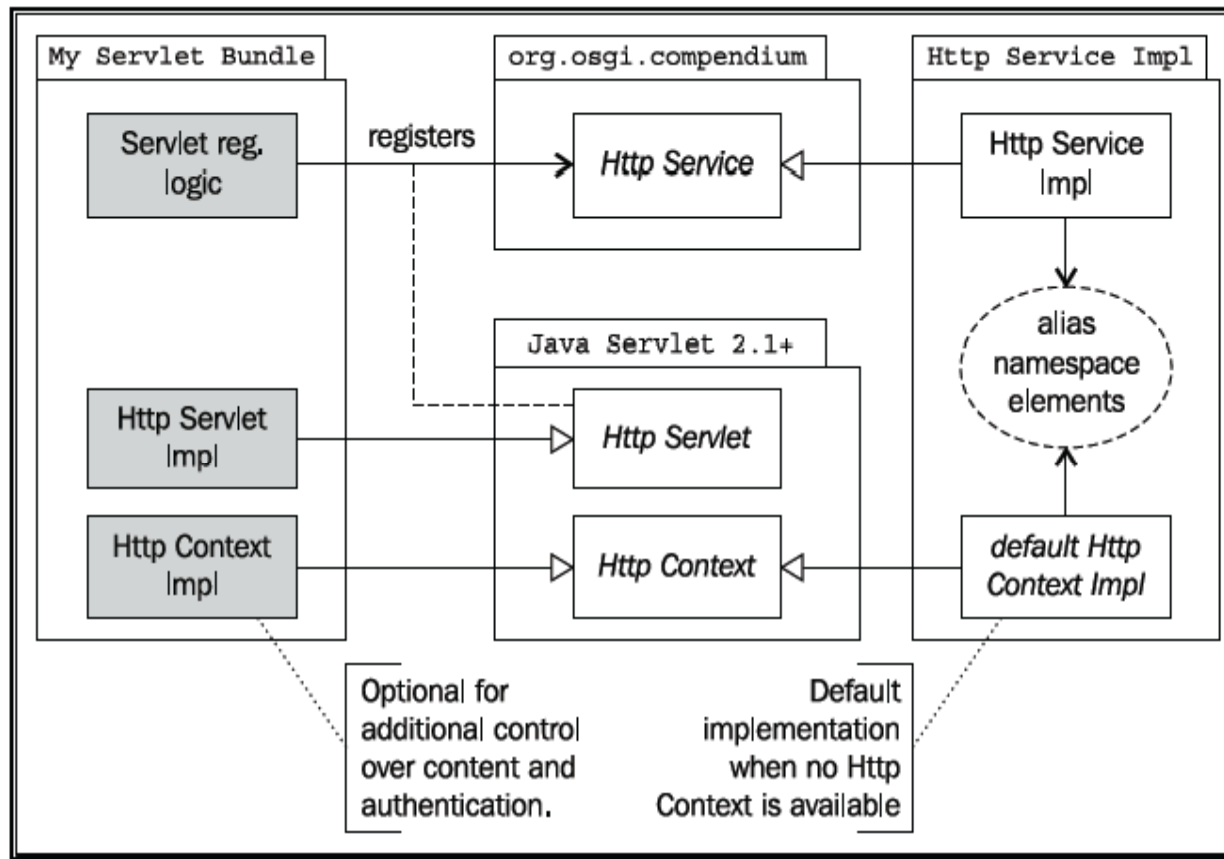
Ex:

alias /myServlet

<http://localhost:8080/myServlet>.

The customization of the base URL is part of the configuration of Http Service implementation.

HTTPService - Component structure



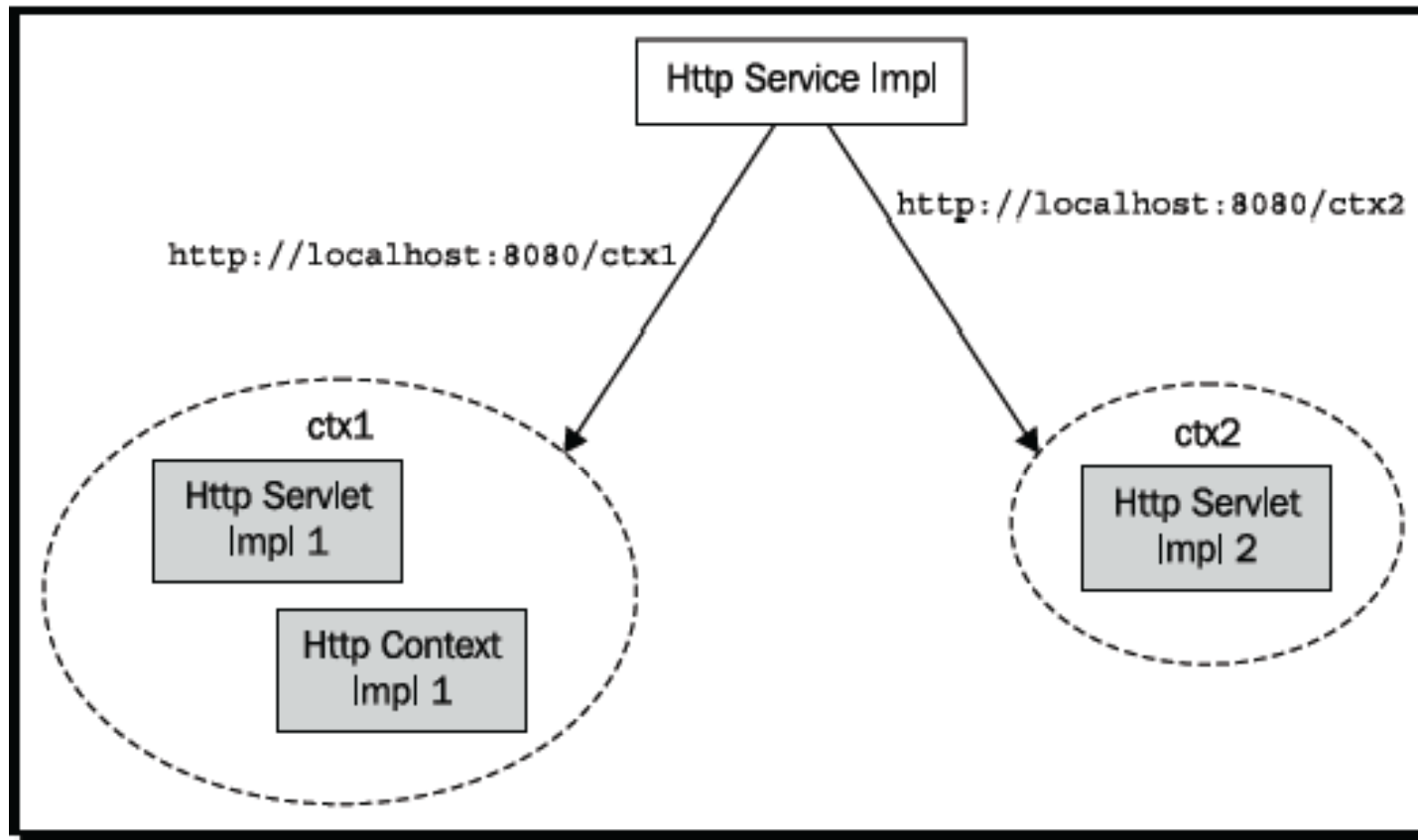
- find the HTTP Service like any other OSGi service

```
String name = HttpService.class.getName();
ServiceReference ref = ctx.getServiceReference(name);
if (ref != null) {
    HttpService svc = (HttpService) ctx.getService(ref);
    if (svc != null) {
        // do something
    }
}
```

Registration – HTTPService

```
Hashtable initParams = new  
    Hashtable();  
initParams.put("paramName",  
    "paramValue");  
getHttpService().registerServlet(    "/alias",  
new MyServlet(), initParams,  
    null);
```

The servlet alias must be unique in the context of the Http Service!



HttpService - unregister

- `getHttpService().unregister("/alias");`

- The HTTP Service is registered by an implementation bundle.
- The client has no control over the port or URL on which the service is running
- etc/jetty.xml
- `org.osgi.service.http.port`—Specifies the port used for servlets and resources accessible via HTTP. The default value is 80.
- `org.osgi.service.http.port.secure`—Specifies the port used for servlets and resources accessible via HTTPS. The default value is 443.

WAB

Web Applications specification

```
Bundle-SymbolicName: com.google.gwt.sample.stockwatcher
Bundle-ClassPath: WEB-INF/lib/gwt-servlet.jar,WEB-INF/classes
Include-Resource: war
Import-Package: \
    com.google.gwt.benchmarks;resolution:=optional,\
    junit.framework;resolution:=optional,\
    *
Web-ContextPath: /stockwatcher/stockPrices
```

Accessing the BundleContext from within a servlet

```
@Override  
public void init() throws ServletException {  
    ctx = (BundleContext) getServletContext()  
        .getAttribute("osgi-bundlecontext");  
}
```

override the `init()` and `destroy()` methods of `javax.servlet.GenericServlet`

OSGi for Web Applications

- Embed a server in Equinox/Karaf
 - forced to use OSGi HTTP Service
 - great for new web applications built from scratch.
 - complicates the development effort required for making an existing WAR file work with this setup.
- **Embed Equinox /Karaf in an existing servlet container**
 - Can use web.xml
 - Easy to use for existing web application.

- Using Jetty: Quick and Easy
- Using WebContainer (like Apache Tomcat)

- Run a Client and connect to the server
 - `bin/client feature:list`
 - `feature:list`

```
-----
```

State	Version	Name	Repository	Description
[uninstalled]	[4.0.9]	wrapper	standard-4.0.9	Provide OS integration
[uninstalled]	[4.0.9]	obr	standard-4.0.9	Provide OSGi Bundle Repository (OBR) support
[uninstalled]	[4.0.9]	config	standard-4.0.9	Provide OSGi ConfigAdmin support
[uninstalled]	[4.0.9]	region	standard-4.0.9	Provide Region commands
[uninstalled]	[7.5.4.v20111024]	jetty	standard-4.0.9	Provide Jetty engine support
[uninstalled]	[4.0.9]	http	standard-4.0.9	Implementation of the OSGi HTTP Service
[uninstalled]	[4.0.9]	http-whiteboard	standard-4.0.9	Provide HTTP Whiteboard pattern support
[uninstalled]	[4.0.9]	war	standard-4.0.9	Turn Karaf as a full WebContainer
[uninstalled]	[4.0.9]	deployers	standard-4.0.9	Provide Karaf deployer
[uninstalled]	[4.0.9]	kar	standard-4.0.9	Provide KAR (KARaf archive) support
[uninstalled]	[4.0.9]	webconsole-base	standard-4.0.9	Base support of the Karaf WebConsole
[uninstalled]	[4.0.9]	webconsole	standard-4.0.9	Karaf WebConsole for administration and monitoring
[uninstalled]	[4.0.9]	ssh	standard-4.0.9	Provide a SSHd server on Karaf

Tutorial

- Http Service implementations