

OSGI

REST

- Introduction Rest - Jersey.
 - ✓ REST API

REST Security

- Apache Shiro overview
- Authentication
- Authorization

Testing OSGi based Applications

- OSGi Mocks
- Pax Exam 2.4
- Troubles shooting OSGi application with Karaf

Best Practices:

- OSGi Best Practices
- Designing web services APIs' for CRUD operations in REST resources.(URI pattern , bundle structure etc).

Labs:

- REST Web Services - Provider With Maven – Jersey
- REST Web Services - Consumer With Maven – Jersey
- REST Web Services - Security With Shiro Maven - Jersey
- PAX Exam and Troubleshooting

Designing and Implementing RESTful Web Services with JAX-RS

goals

- Understand principles of REST
- Survey RESTful tools and protocols
- Learn how to implement RESTful services with JAX-RS

REST Overview

- REpresentational State Transfer
 - <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> (Fielding, 2000)
- Web Architecture
 - *Components*: user agent, intermediary, server, browser, spider, gateway
 - *Connectors*: HTTP, HTTPS, FTP
 - *Data*: URI referring to HTML, XML, RSS

REST is an architectural style

- WWW is based on these principles
 - Starting from the `null` style, += a set of constraints:
- Separation of Concerns
- Statelessness
 - Scalability
- Cacheable
 - Reduce latency, reduce payload
- Uniform Interface
 - Decouples, improves visibility, independently evolve
- Information Hiding
 - Simplified clients, legacy encapsulation, load balancing
- Allow Code-on-Demand
 - Applets, JavaScript

REST in a Nutshell

- REST services are built around Resources
- REST services are Stateless
- REST uses a Uniform Interface
- Resources are manipulated through Representations
- Messages are Self-Describing
- Hypermedia As The Engine Of Application State

Uniform Resources

- *Resources* are domain interests
- Universal Semantics
 - Operations mean the same thing for every resource (GET, DELETE)
 - All Resources are identified by a single mechanism (URI)
- Manipulate resources by exchanging *Representations*

Uniform Methods

- URI is sufficient for working with resources
- Don't need a separate Resource Description Language
- Intermediaries can take advantage of caching, can better anticipate behavior

REST: Noun (Resource) Oriented

- About resources
- The operations are standard via HTTP
- Resources can be cached, bookmarked, saved via standard mechanisms

Customer

<http://example.com/customer/123>

<http://example.com/order/555/customer>

{POST, GET, DELETE}

JAX-RS

- Java API for RESTful Web Services
- Helps developers quickly write RESTful applications
- API Expressed in Annotations

REST and SOAP

- With REST, the semantics are specified entirely by the URI
- The SOAP envelope is the *beginning*
 - It's the extension point for a variety of specs (addressing, security, transactions, MTOM, WS-RM)
- With REST, the URI is the *end*

Orientation

- SOAP: oriented around verbs
 - (RPC, actions)
- REST: oriented around nouns
 - Resources

- A SOAP interface defines verb/noun combinations, after RPC
- It's about a *variety* of operations:

`addCustomer`

`updateCustomerAddress`

`getCustomer`

JAX-RS

Resources

- In JAX-RS, a Resource is a POJO
 - No interface to implement
 - Just express the matching URI
- `@Path`
 - The value is a relative path
 - The base URI is provided by the either
 - Deployment Context
 - Parent Resource

JAX-RS Methods

- If your method returns `void`, JAX-RS returns a 204 (successfully processed, no message body)
- Automatic encoding

– `list")` is identical
`@Path("product%20list")` .

HTTP Methods

Method	Purpose
GET	Read
POST	Create or update if ID is <i>not</i> known
PUT	Update or Create if ID is known
DELETE	Remove

Hello Jersey!

```
import javax.ws.rs.Path;
import javax.ws.rs.GET;
import javax.ws.rs.Produces;

@Path("/helloRest")
public class HelloRest
{
    @GET
    @Produces("text/html")
    public String sayHello()
    {
        return
            "<html><body><h1>Hello
            Jersey!</body></h1></html>";
    }
}
```

Modify web.xml to Use Adapter Servlet

```
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
<servlet>
<servlet-name>ServletAdaptor</servlet-name>
<servlet-class> com.sun.jersey.spi.container.servlet.ServletContainer
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>ServletAdaptor</servlet-name>
<url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

Uniform Interface

- **Annotate methods**
 - **@GET, @PUT, @POST, @DELETE, @HEAD**
- JAX-RS forwards to correct method based on request method

URI Templates

- At class level, assign a Root Resource with `@Path`
- Dynamic resources assigned using `@PathParam (paramName)`
- Can use Regular Expressions to match
 - `@Path ("products/{id}:[a-zA-Z][a-zA-Z_0-9] ")`
 - Non-matches return 404

URI Template Example

```
@Path("/products/{id}")
```

```
public class ProductResource {  
    @Context  
    private UriInfo context;  
    /** Creates a new instance of ProductResource */    public  
    ProductResource() { }
```

```
    @GET  
    @Produces("text/plain")  
    public String getProduct(@PathParam("id") int productId)  
    {  
        switch (productId) {  
            case 1: return "A Shiny New Bike";  
            case 2: return "Big Wheel";  
            case 3: return "Taser: Toddler Edition";  
            default: return "No such  
product";  
        }  
    }  
}
```

Variable Resources of the Same Type

- Map path elements using `@PathParam`:

```
@Path("customer/{name}")
public class Customer {
    @GET
    String get(@PathParam("name") String
        name) { ... }
    @PUT
    Void put(@PathParam("name") String name,
        String value) { ... }
```


Regular Expressions in URI Template

`@Path("/products/{id: \\d{3}}")`

```
public class ProductResource {  
    public ProductResource() { }  
    @GET  
    @Produces("text/plain")  
    public String getProductPlainText(@PathParam("id") int productId)  
    {  
        return "Your Product is: " + productId;  
    }  
}
```

//constrained to 3 digits:

<http://localhost:8080/jrs/resources/products/555>

works <http://localhost:8080/jrs/resources/products/7>

returns 404

Accessing Query Parameters

- Use `@QueryParam` on your method parameter
- Optionally include `@DefaultValue`

`@GET`

`@Produces("text/xml")`

`public String`

`getProducts(`

`@PathParam("id") int productId,`

`@QueryParam("results")`

`@DefaultValue("5") int numResults)`

`– //.../resources/products?results=3`

Accessing Request Headers

```
@GET public String doGet (@Context
    HttpHeaders headers) {
    //list all incoming headers
    MultivaluedMap<String, String> h =
        headers.getRequestHeaders();
    for (String header : h.keySet()) {
        System.out.println(header + "=" +
            h.get(header));
    }
}
```

Accessing Other Parameter Types

- **@FormParam**
 - Extracts from a request representation of MIME media type "application/x-www-form-urlencoded" and conforms to the encoding specified by HTML forms
- **@MatrixParam**
 - Extracts from URL path segments
- **@HeaderParam**
- **@CookieParam**

Representation Formats

- Identified by media type
 - `text/xml`, `application/json`
- Content negotiation is automatically handled by JAX-RS
 - Annotate with `@Produces` or `@Consumes` to indicate static content capabilities

```
@Path("/emps")
public class EmployeeService {
    @GET
    @Path("{id}")
    @Produces("application/xml")
    public Employee getEmployee(@PathParam("id") int empId)
    { return emps.get(empId);
    }
}
```

//this example uses JAXB on the Employee POJO for XML:

```
@XmlRootElement(name="employee")
public class Employee { ...id, name }
```

Produces/Consumes

- **@Produces**
 - Specify the MIME media types of representations a resource can produce and send back to the client.
 - Applied at Class or Method level
- **@Consumes**
 - Specify MIME media types of representations a resource can consume that were sent by the client
 - Applied at Class or Method level
 - One method can consume more than one media type

Building a Response

- `ResponseBuilder` allows you to create a response that contains metadata instead of, or in addition to, an entity

```
String type = new  
    MimeTypeFileTypeMap().getContentType(image);  
return Response.ok(image, type).build();
```

Using JAXB to Provide XML View of Java

- `byte[]`
- `java.lang.String`
- `java.io.InputStream`
- `java.io.Reader`
- `java.io.File`
- `javax.activation.DataSource`
- `javax.xml.transform.Source`
- `javax.xml.bind.JAXBElement` and application-supplied JAXB classes (used for XML media types only)
- `MultivaluedMap<String, String>` for form content only (application/x-www-form-urlencoded)
- `StreamingOutput`

Different Representations of a Resource

- Typical HTTP Accept Request Header

Accept:

`text/html, application/xhtml+xml, application/xml; q=0.9, */*; q=0.8`

- Use HTTP Commons Client library:

```
private static void getXml() {  
    HttpClient client = new HttpClient();  
    GetMethod get = new GetMethod(RESOURCE_URL);  
    get.setRequestHeader("Accept", "text/xml");  
    try {  
        int httpStatus =  
            client.executeMethod(get); if  
            (HttpStatus.SC_OK == httpStatus) {  
                String xmlResponse =  
                    get.getResponseBodyAsString();  
                System.out.println("Xml Response: " +  
                    xmlResponse);  
            }  
    } //...
```

Responses

- `UriInfo`: get information about deployment context, request URI and the route to the resource
- `UriBuilder`: helps you construct resource URIs

Getting a Parameter Map with UriInfo

```
@GET public String get
    (@Context UriInfo ui) {
    MultivaluedMap<String, String> q =
        ui.getQueryParameters();

    MultivaluedMap<String, String> p =
        ui.getPathParameters();
```

UriBuilder

- Makes it easy to build new URIs or from scratch.
- Methods to work with all parts of URI: Path, Fragment, Matrix Param, query param, scheme
- To create **site#faq**:

```
UriBuilder.fromPath("{arg1}").fragment  
( "{arg2}") .build("site", "faq")
```

Adding Metadata to Responses

```
Response response = Response.noContent()  
.header("MY_KEY", "MY_VALUE")  
.cacheControl(cacheCtl)  
.expires(expy)  
.language(Locale.ENGLISH)  
.type(MediaType.TEXT_HTML)  
.build();
```

```
HTTP/1.1 204 No Content    Server: Apache-  
Coyote/1.1                must-revalidate,  
                           max  
                           -  
Cache-Control: no-store, no-transform, age=500  
Expires: Sat, 10 Oct 2009 16:41:49 GMT    MY_KEY: MY_VALUE  
Content-Language: en  
Date: Sat, 08 Nov 2008 16:41:49 GMT
```

Security

- Available via the `SecurityContext` from `@Context`
- Same as `security` in `HttpServletRequest` :
`@Path("cart")`

```
public ShoppingBasketResource  
    get(@Context SecurityContext sc) {  
    if(sc.isUserInRole("GoldMember")  
        { //...
```

Authentication in Jersey

- Need a database to store username, password and group information

```
CREATE TABLE users (  
    id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY  
    KEY, username VARCHAR(64) UNIQUE NOT NULL,  
    password VARCHAR(64) NOT NULL,  
);  
CREATE INDEX username ON users(username);  
  
CREATE TABLE groups (  
    id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  
    username VARCHAR(64) NOT NULL REFERENCES users(username) ON DELETE  
    CASCADE, groupname VARCHAR(64)  
);
```

Authentication in Jersey..

- Define a JDBC data source, connection pool and security realm in Glassfish

Authentication in Jersey..

- Define resource as follow:

```
@Path("dropbox")
public class DropBox
{
    @Context
    SecurityContext security;

    @POST
    public Response drop(InputStream data) {
        String username = security.getUserPrincipal().getName();
        ...
    }
}
```

Authentication in Jersey..

- web.xml

```
<security-constraint>
  <display-name>DropBox</display-name>
  <web-resource-collection>
    <web-resource-name>DropBox</web-resource-name>
    <description></description>
    <url-pattern>/dropbox</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
    <http-method>HEAD</http-method>
    <http-method>PUT</http-method>
    <http-method>OPTIONS</http-method>
    <http-method>TRACE</http-method>
    <http-method>DELETE</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>Have to be a USER</description>
    <role-name>USERS</role-name>
  </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>userauthn</realm-name>
</login-config>
<security-role>
  <description/>
  <role-name>USERS</role-name>
</security-role>
```

- include a suitable Authorization header in the request

Implementations

- Jersey
- Restlet
- JBoss RESTEasy
- Apache CXF
- Triaxrs
- Apache Wink

Tutorials

- Tutorial : REST Web Services - Provider With Maven - Jersey
- Tutorial : REST Web Services - Consumer With Maven - Jersey

Application Security With Apache Shiro

What is Apache Shiro?

- a powerful and easy-to-use Java security framework
- Performs :
 - authentication,
 - authorization,
 - cryptography,
 - and session management
- can be used to secure any application

Why would you use Apache Shiro?

- **Easy To Use**
- **Comprehensive**
- **Flexible**
- **Web Capable**
- **Pluggable**
- **Supported**

Core Concepts:

- Subject
- SecurityManager
- Realms

Subject

- the currently executing user“

```
import
org.apache.shiro.subject.Subject;
import org.apache.shiro.SecurityUtils;
...
Subject currentUser =
SecurityUtils.getSubject();
```

current user consists of, such as login, logout, access their session, execute authorization checks, and more

the Subject represents security operations for the current user,

SecurityManager

- manages security operations for *all* users.
- It is the heart of Shiro's architecture
- Set up the SecurityManager instance
 - web.xml (Shiro Servlet Filter)
 - text-based [INI](#) configuration.

Configuring Shiro with INI

```
[main]
```

```
cm = org.apache.shiro.authc.credential.HashedCredentialsMatcher
```

```
cm.hashAlgorithm = SHA-512
```

```
cm.hashIterations = 1024
```

```
# Base64 encoding (less text):
```

```
cm.storedCredentialsHexEncoded = false
```

```
iniRealm.credentialsMatcher = $cm [users]
```

```
jdoe = TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJpcyByZWZzb2
```

```
Asmith =Npbmd1bGFyIHhBhc3Npb24gZnJvbSBvdGhlciBhbXNoZWQsIG5vdCB
```

Realms

- Realm acts as the ‘bridge’ or ‘connector’ between Shiro and your application’s security data.
- a Realm is essentially a security-specific [DAO](#):
 - it encapsulates connection details for data sources
 - makes the associated data available to Shiro as needed.

Example realm configuration snippet to connect to LDAP user data store

[main]

IdapRealm = org.apache.shiro.realm.ldap.JndiLdapRealm

IdapRealm.userDnTemplate = uid={0},ou=users,dc=mycompany,dc=com

IdapRealm.contextFactory.url = ldap://ldapHost:389

IdapRealm.contextFactory.authenticationMechanism = DIGEST-MD5

Authentication

- the process of verifying a user's identity
- a three-step process.
 - Collect the user's identifying information, called *principals*, and supporting proof of identity, called *credentials*.
 - Submit the principals and credentials to the system.
 - If the submitted credentials match what the system expects for that user identity (principal), the user is considered authenticated. If they don't match, the user is not considered authenticated.

Subject Login

- //1. Acquire submitted principals and credentials:
AuthenticationToken token = new
UsernamePasswordToken(username, password);
- //2. Get the current Subject:
Subject currentUser = SecurityUtils.getSubject();
- //3. Login:
currentUser.login(token);

Handle Failed Login

- `//3. Login: try {
 currentUser.login(token
);
} catch (IncorrectCredentialsException ice) {
 ... } catch (LockedAccountException lae) { ...
} ... catch (AuthenticationException ae) {... }`

Authorization

- Authorization is essentially access control - controlling what your users can access in your application, such as resources, web pages, etc.

Authorization

Role Check

```
if ( subject.hasRole("administrator") ) {  
  
    //show the 'Create User' button  
} else {  
    //grey-out the button?  
  
}
```

Permission Check

```
if ( subject.isPermitted("user:create") ) {  
    //show the 'Create User' button  
} else {  
  
    //grey-out the button?  
}
```

Instance-Level Permission Check

```
if ( subject.isPermitted("user:delete:jsmith") )  
{  
    //delete the 'jsmith' user  
} else {  
    //don't delete 'jsmith' }
```

Session Management

a consistent Session API usable in any application and any architectural tier.

Subject's Session

```
Session session = subject.getSession(); Session session =  
subject.getSession(boolean create);
```

Session methods

```
Session session = subject.getSession();  
session.getAttribute("key", someValue); Date start =  
session.getStartTimeStamp(); Date timestamp =  
session.getLastAccessTime(); session.setTimeout(millis);  
...
```

Cryptography

the process of hiding or obfuscating data so prying eyes can't understand it.

Hashing

JDK's MessageDigest

```
try {  
    MessageDigest md =  
    MessageDigest.getInstance("MD5");  
    md.digest(bytes);  
    byte[] hashed = md.digest();  
} catch (NoSuchAlgorithmException e) {  
    e.printStackTrace();  
}
```

Cryptography

Ciphers :cryptographic algorithms that can reversibly transform data using a key.

Apache Shiro's Encryption API

```
AesCipherService cipherService = new AesCipherService();  
cipherService.setKeySize(256);  
//create a test key:  
byte[] testKey = cipherService.generateNewKey();  
  
//encrypt a file's bytes: byte[]  
encrypted =  
    cipherService.encrypt(fileBytes, testKey);
```

Web Support

Shiro ships with a robust web support module to help secure web applications.

ShiroFilter in web.xml

```
<filter>
  <filter-name>ShiroFilter</filter-name>
  <filter-class>
    org.apache.shiro.web.servlet.IniShiroFilter
  </filter-class>
  <!-- no init-param means load the INI config from
    classpath:shiro.ini -->
</filter>
<filter-mapping>
  <filter-name>ShiroFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

URL-Specific Filter Chains

Path-specific Filter Chains

[urls]

/assets/** = anon

/user/signup = anon

/user/** = user

/rpc/rest/** = perms[rpc:invoke], authc

/** = authc

Web Session Management

Default Http Sessions

the methods `subject.getSession()` and `subject.getSession(boolean)` Shiro will return Session instances backed by the Servlet Container's `HttpSession` instance.

Shiro's Native Sessions in the Web Tier

Lab : Web services Security

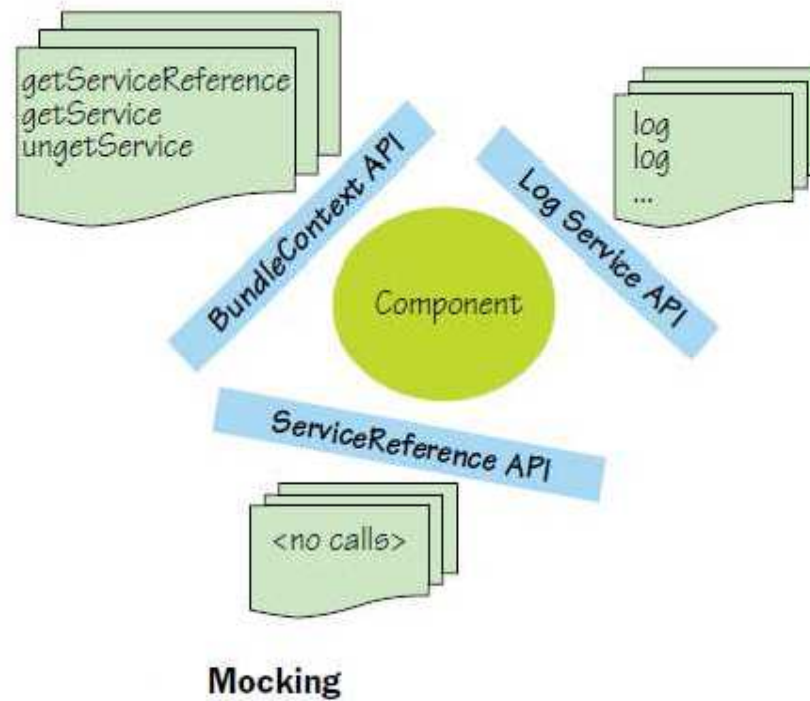
Testing OSGI applications

Mocks & PAX Exam

Mocking OSGi

- *mock objects : to test portions of code without requiring a complete system.*
- A mock object is basically a simulation, not a real implementation.
- a powerful technique
- Verify that right methods are called in the appropriate order

Mocking - Example



Mocking API

EasyMock

- 1 *Mock—Create prototype mock objects*
- 2 *Expect—Script the expected behavior*
- 3 *Replay—Prepare the mock objects*
- 4 *Test—Run the code using the mock objects*
- 5 *Verify—Check that the behavior matches*

Create prototype objects

```
BundleContext context =  
    createStrictMock(BundleContext.class);  
ServiceReference serviceRef =  
    createMock(ServiceReference.class);  
LogService logService =  
    createMock(LogService.class);
```

Script the expected behavior

```
expect (context.getServiceReference(LogService.class.getName()))  
    .andReturn(serviceRef);
```

```
expect (context.getService(serviceRef))  
    .andReturn(logService);
```

```
logService.log (and (geq (LogService.LOG_ERROR,  
    leq (LogService.LOG_DEBUG)), isA (String.class)));
```

initialize your mock objects

```
replay(context, serviceRef,  
        logService);
```


Use your mock objects

```
BundleActivator logClientActivator = new Activator();  
logClientActivator.start(context); try {  
    Thread.sleep(1000);  
} catch (InterruptedException e) {}  
logClientActivator.stop(context);
```

the expected behavior

```
verify(context, serviceRef, logService);
```

Pax Exam

- a test framework for OSGi bundles and OSGi based applications.
- based on the principles of launching a specified OSGi Framework setup.
- support a wide range of different OSGi frameworks.
- supports both JUnit 3 and 4,

Basic approach

1. Prepare the OSGi container.
2. Deploy the selected bundles.
3. Create a test bundle on the fly.
4. Deploy and execute the tests.
5. Shut down the container.

- A probe carries tests only.
- may obtain context information via Parameter injection

```
import static org.ops4j.pax.exam.CoreOptions.*;
```

```
org.ops4j.pax.exam.Option
```

Test Containers

- TestContainer implementations:
 - PaxRunnerTestContainer
 - NativeTestContainer

Maven - Configuration

```
<dependency>
  <groupId>org.ops4j.pax.exam</groupId>
  <artifactId>pax-exam-container-native</artifactId>
  <version>${paxexamversion}</version>
  <scope>test</scope>
</dependency>
```

```
<dependency>
  <groupId>org.ops4j.pax.exam</groupId>
  <artifactId>pax-exam-container-paxrunner</artifactId>
  <version>${paxexamversion}</version>
  <scope>test</scope>
</dependency>
```

Pax Runner Test Container

- launch a new Pax Runner instance as "TestContainer"
- Benefits:
 - New JVM means you have full control over which JVM will be used and what the options look like
 - Vast amount of additional options: use `scan*()` and `profile()` options which simplify your setup.
- Drawbacks:
 - May be slower (new Process launched). More computation happening crunching the arguments forth and back.
 - RMI Communication happening. This has been stabilized compared to Pax Exam 1. But its still networking with all its implications.
- Use it:
 - when you need the Pax Runner specialties

Native Test Container

- simply add at one compatible OSGi framework

```
<dependency>
  <groupId>org.apache.felix</groupId>
  <artifactId>org.apache.felix.framework</artifactId>
  <version>${felixversion}</version>
  <scope>test</scope>
</dependency>
```

Benefits:

- Really fast

- Flawless debugging experience

Drawbacks:

- Less options compared to the Pax Runner Test Container.

- Even if we could, this test container will not try to copy all the options the Pax Runner Container gives you. (like the profiles/scanner feature)

Use it:

- by default.

Example

```
@RunWith(JUnit4TestRunner.class)
@ExamReactorStrategy(AllConfinedStagedReactorFactory.class)
public class SampleTest {

    @Inject
    private HelloService helloService;

    @Configuration
    public Option[] config() {

        return options(
            mavenBundle("com.example.myproject", "myproject-api", "1.0.0-SNAPSHOT"),
            bundle("http://www.example.com/repository/foo-1.2.3.jar"),
            junitBundles(),
            equinox().version("3.6.2")
        );
    }

    @Test
    public void getHelloService() {
        assertNotNull(helloService);
        assertEquals("Hello Pax!", helloService.getMessage());
    }
}
```

Pax Exam Maven Dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.ops4j.pax.exam</groupId>
  <artifactId>pax-exam-sample-empty</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>
    <exam.version>2.5.0</exam.version>
    <url.version>1.4.0</url.version>
  </properties>

  <dependencies>

    <dependency>
      <groupId>org.ops4j.pax.exam</groupId>
      <artifactId>pax-exam-container-native</artifactId>
      <version>${exam.version}</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.ops4j.pax.exam</groupId>
      <artifactId>pax-exam-junit4</artifactId>
      <version>${exam.version}</version>
      <scope>test</scope>
    </dependency>
```

Hint: double-click to select code

```
<dependency>
  <groupId>org.ops4j.pax.url</groupId>
  <artifactId>pax-url-aether</artifactId>
  <version>${url.version}</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.apache.felix</groupId>
  <artifactId>org.apache.felix.framework</artifactId>
  <version>3.2.2</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-core</artifactId>
  <version>0.9.20</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>0.9.20</version>
  <scope>test</scope>
</dependency>

</dependencies>
```

Pax Exam Maven Dependencies...

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.5.1</version>
      <configuration>
        <source>1.6</source>
        <target>1.6</target>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

Pax Runner Container Example

```
<dependency>
  <groupId>org.ops4j.pax.exam</groupId>
  <artifactId>pax-exam-container-paxrunner</artifactId>
  <version>${exam.version}</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.ops4j.pax.runner</groupId>
  <artifactId>pax-runner-no-jcl</artifactId>
  <version>${runner.version}</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.ops4j.pax.exam</groupId>
  <artifactId>pax-exam-junit4</artifactId>
  <version>${exam.version}</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.ops4j.pax.exam</groupId>
  <artifactId>pax-exam-link-mvn</artifactId>
  <version>${exam.version}</version>
  <scope>test</scope>
</dependency>
```



```
<dependency>  
  <groupId>ch.qos.logback</groupId>  
  <artifactId>logback-core</artifactId>  
  <version>0.9.29</version>  
  <scope>test</scope>  
</dependency>
```

```
<dependency>  
  <groupId>ch.qos.logback</groupId>  
  <artifactId>logback-classic</artifactId>  
  <version>0.9.29</version>  
  <scope>test</scope>  
</dependency>
```

Tutorial Testing With PAX Exam

REST - Guidelines

URI Design:

- be concise.
- be easy to remember.
- un-ambiguously identify the target resource.
- have only nouns as part of URI
- Modular.

Versioning

- pass version information in 'Accept' request header
 - Accept: application/xml;version=1.0

REST – Guidelines..

- Granularity:
 - Services should be coarse grained
- Request/Response mime types:
 - Service should use standard HTTP headers to consume/produce different mime types.
 - content-type and accept headers: application/xml and application/json

REST – Guidelines..

- Caching:
 - Services should cache only successful GET requests
 - for cachable responses, server should set HTTP 'Vary' header

REST – Guidelines..

- Logging
 - Logging should follow one log per request pattern
- Error Handling
 - Service should stick to [HTTP Status Codes](#) for communicating success/failure
 - include a list of user error messages in the response body

REST – Guidelines..

- Document should include (for each service method):
 - HTTP Method
 - URI
 - Accept and Content-Type HTTP Request Headers
 - All possible HTTP Response codes
 - Any custom Headers
 - Sample Response
 - Sample request body for PUT, POST requests
 - Schema (xsd files for each request and response)

OSGi Best Practices!

OSGi Best Practices!

Learn how to prevent common mistakes and build robust, reliable, modular, and extendable systems using OSGi™ technology

Introduction to OSGi Technology

The Dynamic Module System for Java™ Platforms

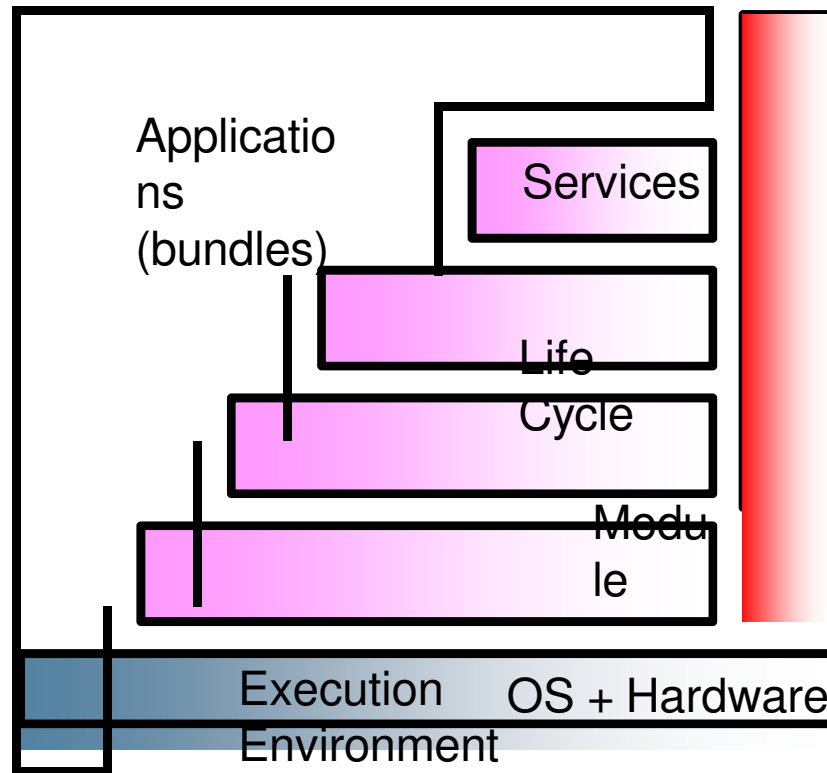
- It's a module system for the Java platform
 - Includes visibility rules, dependency management and versioning of bundles, the OSGi modules
- It's dynamic
 - Installing, starting, stopping, updating, uninstalling bundles, all dynamically at runtime
- It's service oriented
 - Services can be registered and consumed inside a VM, again all dynamically at runtime
- A specification of the OSGi Alliance, a non profit organization <http://www.osgi.org>

OSGi Technology Key Benefits

The Dynamic Module System for Java™ Platforms

- Avoids Java Archive (JAR) file hell
- Reuse code “out of the box”
- Simplifies multi-team projects
- Enables smaller systems
- Manages deployments local or remotely
- Extensive tool support
- No lock in, many providers of core technology including many open source
- Very high adoption rate

OSGi Layering



Portable Code

Problem

- You compile your code using source level 1.3 on a Java 5 platform compiler, assuming you are safe to run on older VMs
- But then it fails to run when you deploy to a Java platform 1.3 or CDC/Foundation 1.0 environment
- It turns out that despite your 1.3 source level, you were still linked to new parts in the Java 5 class library

```
java.lang.NoSuchMethodError: java.lang.StringBuffer:  
method  
append(Ljava/lang/StringBuffer;)Ljava/lang/StringBuffer;  
not found
```

Portable Code

Best Practice

- Compile your code against the minimum suitable class libraries
- OSGi specification defines Execution Environments (EE)
 - OSGi Minimum—Absolute minimum, suitable for API design
 - Foundation—Fairly complete EE, good for most applications; Used for Eclipse
 - JAR files available from OSGi website
- Java platforms are backward compatible so you should always compile against the lowest version you are comfortable with
 - New features are good, but there is a cost!
 - At least think about this

Proper Imports

Problem

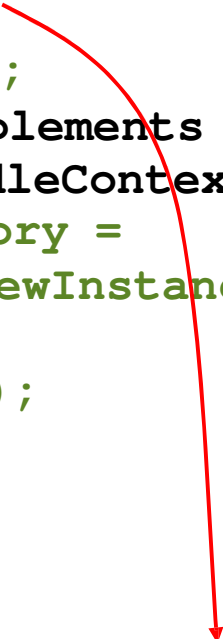
- You develop and test your bundles on an OSGi Service Platform that you have configured yourself
- Your colleague tries these bundles on another OSGi Service Platform and complains of a **ClassNotFoundException** in your bundles

Proper Imports

Problem

Code:

```
import org.osgi.framework.*
import ;
    javax.xml.parsers.*;
public class Activator implements BundleActivator
{ public void start(BundleContext ctxt) {
    SAXParserFactory factory =
        SAXParserFactory.newInstance();
    SAXParser parser =
        factory.newSAXParser();
}
}
```



*Missing an import for
javax.xml.parsers in
the manifest*

Manifest:

```
Import-Package:
org.osgi.framework
```

Proper Imports

Best Practice

- Do not assume that everything in the Java Runtime Environment (JRE) will be available to your bundle
 - Only java.* packages are reliably available from the boot class path.
- Your bundle must import all packages that it needs
 - Except: java.* does not need to be imported
- Why?
 - Enables bundles to provide substitute implementations of JRE implementation release software version packages.
- The **org.osgi.framework.bootdelegation** system property may be set differently on different configurations, so you should never rely on its setting

Minimize Dependencies

Problem

- You find an interesting bundle and want to use it
- You install it in an OSGi framework
- You find it has dependencies on other bundle
- So you find and install those bundles
- Those bundles end up depending on still other bundles ...
 - Ad nauseum ...

Minimize Dependencies

Best Practice

- Use **Import-Package** instead of **Require-Bundle**
 - Require-Bundle can have only one provider— the named bundle
 - Import-Package can have many providers
 - Allows for more choices during resolving
 - Has a lower fan out, which gain adds up quickly
- Use version ranges
 - Using precise version numbers gives the dependency resolver less choice
- Design your bundles
 - Don't put unrelated things in the same bundle
 - Low coupling, high cohesion

Hide Implementation Details

Problem

- You wrote a bundle that has a public API and associated implementation code
 - This implementation code defines public classes because it needs to make cross-package calls and references
- You exported all the packages in your bundle
- In the future, you release an update to the bundle with the same public API but a vastly different implementation
- You then get an angry call because you broke some customer's code
 - And you told them not to use the implementation packages ...

Hide Implementation Details

Best Practice

- Put implementation details in separate packages from the public API
 - `org.example.foo` - exported API package
 - `org.example.foo.impl` - private implementation package
- Do not export the implementation packages
 - Export and/or import the public details while keeping the implementation details private
 - **Export-Package:** `org.example.foo; version=1.0`

Avoid Class Loader Hierarchy Dependencies

Problem

- You are designing a multimedia system and want to allow other bundles to provide plugin codecs
- Your design requires them to pass names of the codec classes which you load via **Class.forName**
 - Either by method call or configuration file
- This design works in a traditional tree based class loader model since the multimedia system's class loader has visibility to the codec classes
- However, in an OSGi environment, the multimedia system gets **ClassNotFoundExceptions** since it does not have visibility to the codec classes

Avoid Class Loader Hierarchy Dependencies

Best Practice

- Better to use a safe OSGi model like services or the Extender Model to have bundles contribute codecs
 - More dynamic, you can add new services on the fly by installing bundles
- Workaround for using **Class.forName**
 - Use **DynamicImport-Package: *** and have the contributing bundles export their codec package
 - This may work but can result in unintended side effects since your bundle may import packages it did not expect

Avoid Start Ordering Dependencies

Problem

- You develop a bundle that uses the Http Service and get the service in your BundleActivator

```
public class HttpService implements BundleActivator {  
  
    public void start(BundleContext ctxt) {  
  
        ServiceReference ref =  
ctxt.getServiceReference(HttpService.class.getName());  
        http = ctxt.getService(ref);  
        http.registerServlet(); }  
  
}
```

- Your bundle works fine on your workstation but fails with a `NullPointerException` on the call to `getService` when integrated into the build

Avoid Start Ordering Dependencies

Best Practice

- Do not assume that you can always obtain a service during initialization
 - Bundles can start in different orders on different systems and you usually do not have control over the order
- Use **ServiceTracker** to track services and respond to their publication by subclassing or via a **ServiceTrackerCustomizer**
- Use a declarative service model like OSGi Declarative Services or Spring OSGi

Handle Service Dynamism

Problem

- You develop a bundle with a servlet
- You get the HttpService and register your servlet
- After deployment, you receive problem reports that your servlet seems to vanish after working for a while
- It turns out the HttpService was unpublished temporarily when the HttpService bundle was stopped and restarted during an update
- Your bundle did not react and re-register the servlet


Handle Service Dynamism

Best Practice

- A service is a dynamic entity and can be unpublished after you get it
 - A bundle must respond to the lifecycle of a dependent service
- The OSGi framework provides an API to handle these dynamics but they are rather low level
- There are helpers, based on this API, like:
 - Service Tracker and Service Activator Toolkit (SAT)
 - Declarative models like Declarative Services, iPOJO and Spring OSGi

Whiteboard Pattern

Problem

- You design a service provided by your bundle to use the familiar `addListener` and `removeListener` methods
- In practice, you find that other bundles forget to call `removeListener` when they stop or you stop, or forget to call `addListener` when you restart
- Both bundles need special code to track the other bundle or events are not properly delivered
- The OSGi `LogReaderService` design is an example of this problem 


Whiteboard Pattern

Best Practice

- Design your API to have the listener registered as a service
 - Simple
 - More robust
 - Leverages the OSGi service model and its life cycle model awareness
- The event source tracks the listener services and calls them when there is an event to deliver
- This is called the Whiteboard Pattern
 - It can be considered an Inversion of Control pattern
- The OSGi **EventAdmin** design is an example of this best practice

Extender Model

Problem

- You design a Help System where other bundles contribute help content to your bundle
- The other bundles need to track the Help System bundle and contribute their Help content
- The Help System bundle must clean up when the bundles that contribute Help content are stopped
- This problem of tracking bundle life cycles is much like the one solved by the Whiteboard Pattern
 - But there is another pattern to address this use case
- The OSGi HttpService design is an example of this problem 

Extender Model

Best Practice

- The bundle being "extended" specifies a data schema
- Contributing bundles define this data in their bundle
- The extender bundle will track the bundles via certain life cycle event and process the data, if present
 - This can include loading classes from the contributing bundle
- Extenders have more advantages
 - Lazy—less time pressure on startup and less memory later
 - More robust in case of failures—extender bundle can make consistent and policy driven choices
- Many bundles use this pattern
 - Declarative Services, iPOJO, Spring OSGi and Eclipse Extension Point Registry

Avoid OSGi Framework API Coupling

Problem

- You wrote your code and packaged it in a bundle
- Your code publishes an OSGi service for other bundles to use and also uses services provided by other bundles
- Your code uses the OSGi service layer API in quite a number of classes and is now coupled to the OSGi API
- You no longer can easily use your code in a non-OSGi environment

Avoid OSGi Framework API Coupling

- Best Practice
- Write your code as POJOs (Plain Old Java Objects)
- Program against interfaces, not concrete classes
- Isolate the use of OSGi API to a minimal number of classes
- Let these coupled classes inject dependencies into the POJOs
- Make sure none of your domain classes depend on these OSGi coupled classes
- Use an OSGi ready IoC container like Declarative Services or Spring OSGi to express these dependencies in a declarative form
 - Let the IoC containers handle all of the OSGi API calls

Return Quickly from Framework Callbacks

Problem

- You work in a large team building an enterprise OSGi based system
- Each developer develops their part of the system in a modular fashion and does extensive and continuous unit testing
- When all bundles are put together for integration test, a week before deadline, it takes too long to bring up the whole system
- It turns out that each bundles spent a long time in their activator and the cumulative effect on the complete system was significant

Return Quickly from Framework Callbacks

Best Practice

- Bundle developers have a tendency to do too much up front activation
- 1s per bundle (think DNS name lookup)
 - => One minute with 60 bundles
 - => Five minutes with 300 bundles
- Lazy is good
 - See new lazy activation features in Release 4 Version 4.1
- Framework callbacks need to return quickly
- If you need to do something that takes some time then either:
 - Use eventing, or
 - Spin off a background thread to perform the long running work

Thread Safety

- **Problem**
- You develop a bundle and test it extensively
- However when deployed in the field with a set of other bundles, your bundle fails with exceptions in strange places
- Ultimately you realize that these other bundles are triggering events
 - Which your bundle receives and processes
 - But the events are being delivered on many different threads
- Time to consult a concurrency expert...

Thread Safety

Best Practice

- In an OSGi environment, framework callbacks to your bundle can occur on many different threads simultaneously
- Your code must be thread-safe!
 - Callbacks are likely running on different threads and can occur really simultaneously
 - Do not hold any locks when you call a method and you do not know the implementation, they might call back to bite you
 - Java platform monitors are intended to protect low level data structures; use higher level abstractions with time outs for locking entities
 - In multi-core CPUs, memory access to shared mutable state must always be synchronized

Thank YOU!

You can reach me → life.skolur@gmail.com