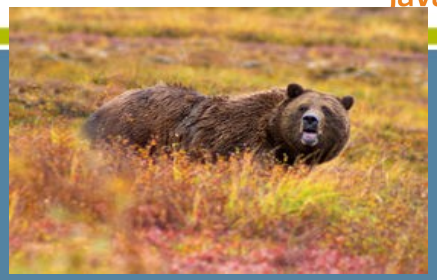


Embedding the Grizzly Framework

Jeanfrancois Arcand
Jerome Dochez
Senior Staff Engineers
SUN Microsystems
www.sun.com

BOF-4989

Goal of Your Talk

Learn about the Grizzly Framework

Grizzly is a flexible and very high performance Java technology-based new I/O (NIO) framework. This session will describe how the Grizzly Framework is designed, how it is used in several prject and how to create your own customizations of this scalable and extensible architecture.

Agenda

Introduction

What is Project Grizzly

Architecture

Glassfish V3

Demo

Q&A

Agenda

Introduction

What is Project Grizzly

Architecture

GlassFish V3

Q&A

Introduction

- In this presentation we will:
  - **describe Project Grizzly**
  - **giving a brief history of Grizzly**
  - **provide a architecture overview of Grizzly**
  - **tell you who is already using Grizzly, who is looking at Grizzly**
  - **Introduce GlassFish V3 project.**

Agenda

Introduction

What is Project Grizzly

Architecture

GlassFish V3

Q&A

Grizzly History

- Grizzly was born in 2004 under the GlassFish project, (https://glassfish.dev.java.net). This later became known as Grizzly 1.0
- HTTP over TCP/SSL was the first implementation
- Grizzly 1.0 shipped with Sun Java System Application Server 8.1 PE, 8.2 PE/EE and all GlassFish distributions, replacing native Sun WebServer runtime.
- Initially used to build an HTTP Web Server, replacing Tomcat's Coyote Connector and Sun WebServer 6.1

Grizzly History

- Grizzly 1.0 became extremely popular in 2006. Multiple protocol implementations were built on top of it
- But Grizzly 1.0 had HTTP protocol specific implementation details included in its transport logic
- The main class, SelectorThread, contained several artifacts specific to http such as file caching, request monitoring, etc

Grizzly History

- Several classes needed to be extended in order to use the framework
- Example: JettySelectorThread extends SelectorThread
- Example: SSLSelectorThread extends SelectorThread
- The Grizzly 1.0 mixed ‘extension’ and ‘implementation’

Grizzly History

- But, Grizzly 1.0 was still a good choice for nearly all TCP/HTTP based protocols.
- Several projects successfully utilized Grizzly 1.0:
  - JRuby On Grizzly
  - Alaska's HTTP BC component (OpenESB)
  - GlassFish v3/hk2
  - Phobos in NetBeans
  - Project Tango
  - Comet/Cometd
  - AsyncWeb on Grizzly
  - GlassFish v2
  - Sun Web 2.0 Developer pack (REST HTTP Server)

Grizzly History

- Grizzly 1.5 began development in 2006
- Currently under review and will release very soon
- Grizzly 1.5 objectives
  - Remove all dependencies to HTTP and/or GlassFish
  - All 1.0 applications must still work with 1.5
  - Support all tricks and tips learned during development of Grizzly 1.0 (performance, NIO performance traps, etc.)
  - **Keep it simple!!**
- Open Source Grizzly occurred February 6, 2007!
- Grizzly 1.5 started community release last week!

Agenda

Introduction

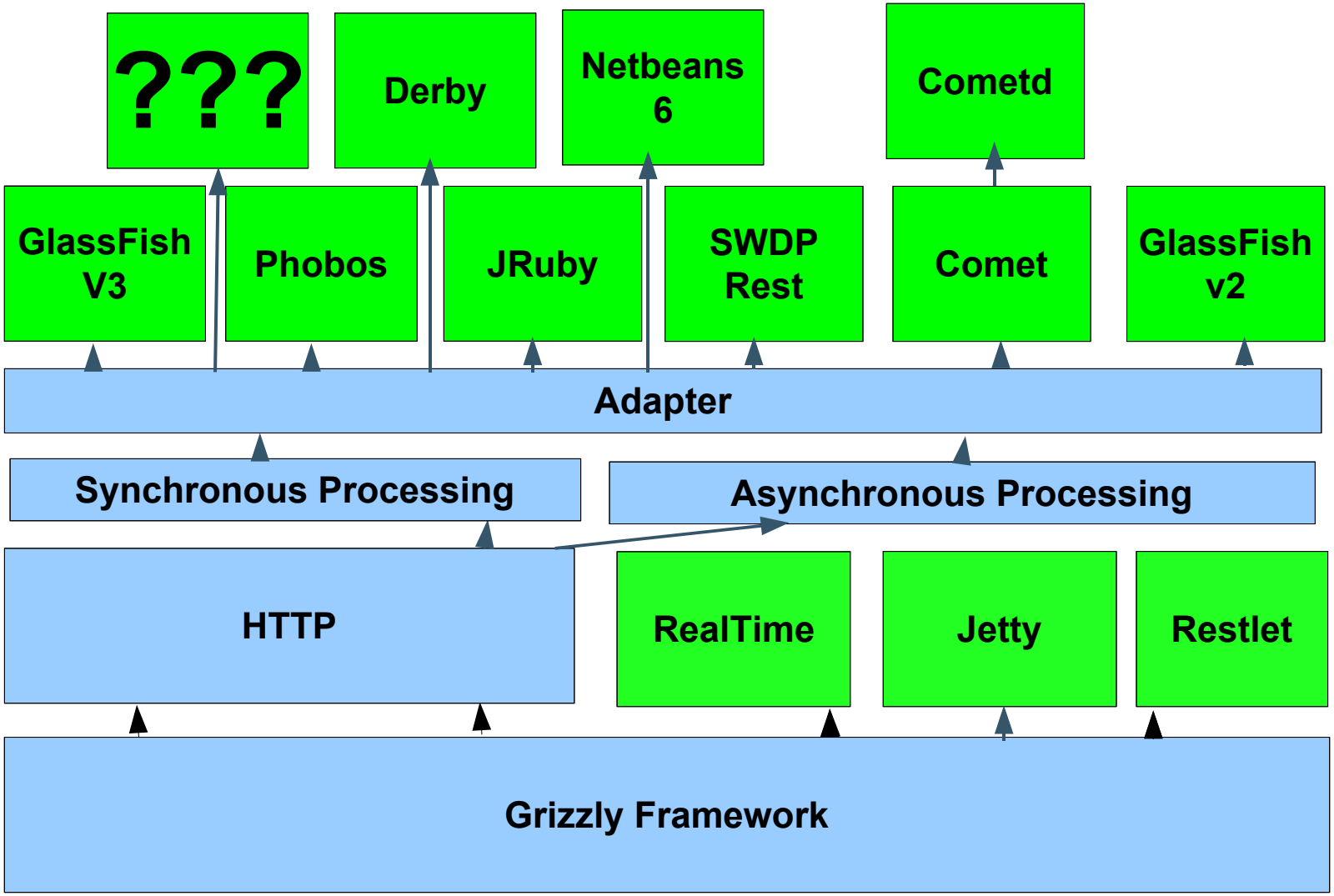
What is Project Grizzly

Architecture

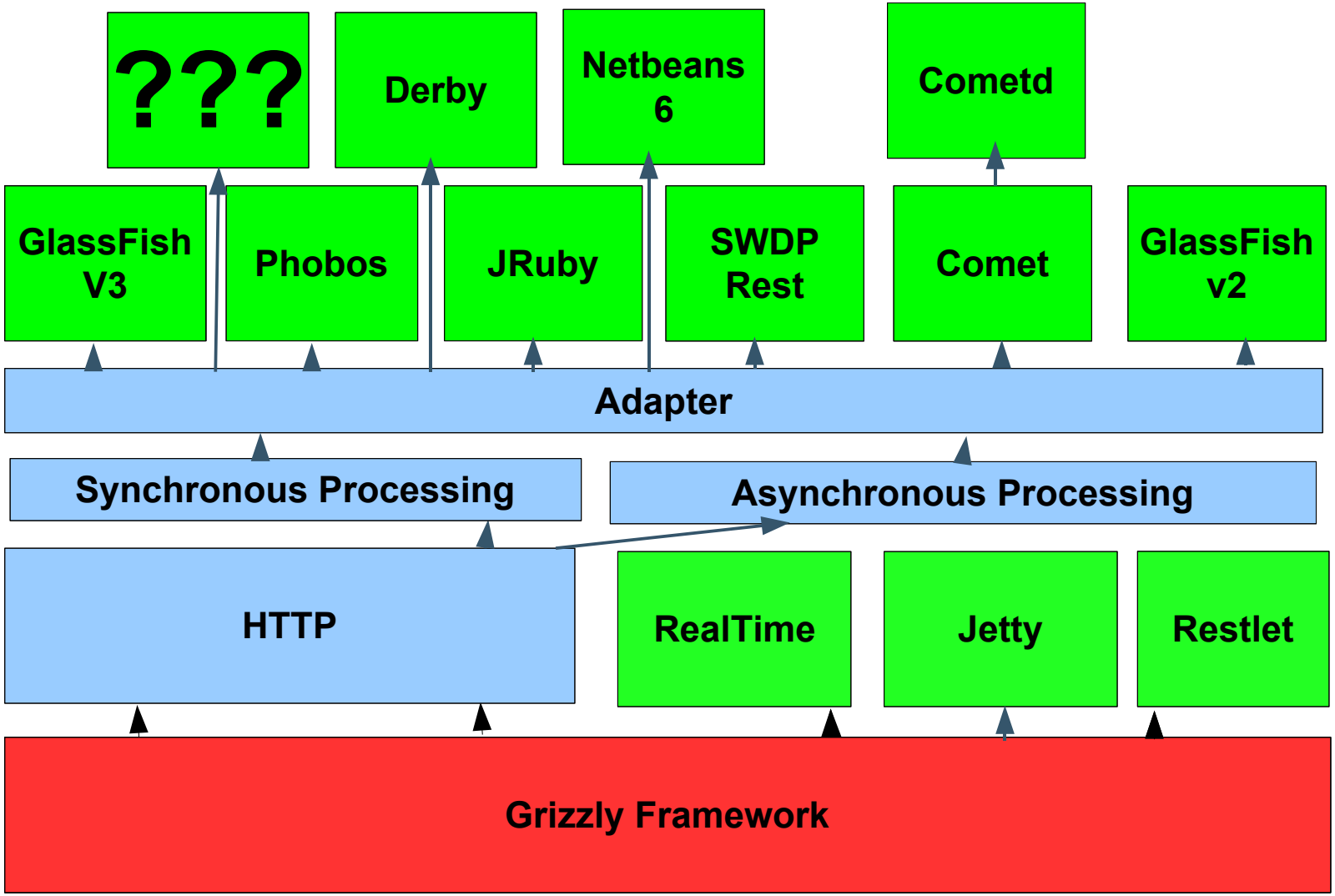
GlassFish V3

Q&A

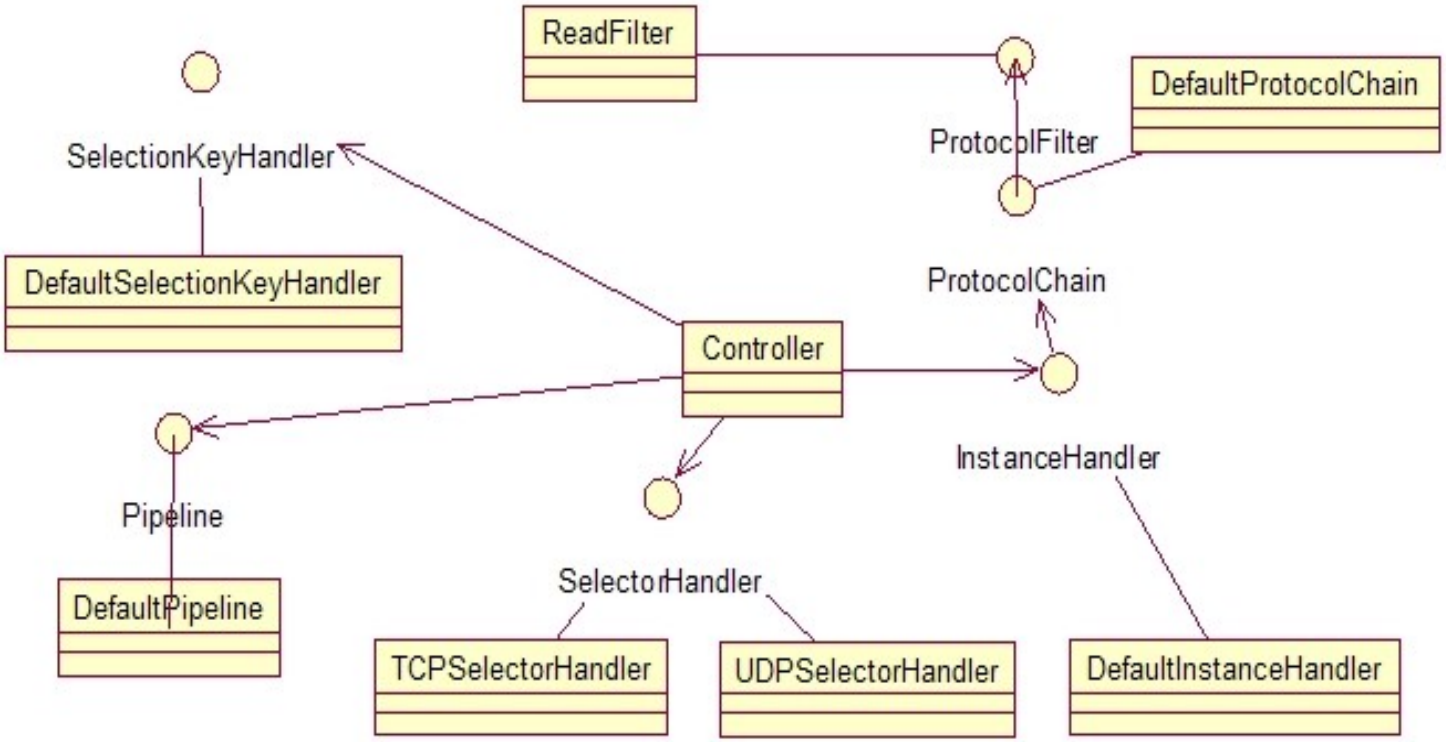
Architecture



Architecture - Grizzly Framework



Grizzly Class Diagram



Controller

- Main entry point when using the Grizzly Framework -- Controller
- A Controller is composed of
  - SelectorHandler
  - SelectionKeyHandler
  - ProtocolChainInstanceHandler
  - ProtocolChain
  - Pipeline
- All of these components are configurable using the Grizzly Framework

SelectorHandler

- A SelectorHandler handles all java.nio.channels.Selector operations. One or more instance of a Selector are handled by SelectorHandler.
- The logic for processing of SelectionKey interest (OP_ACCEPT,OP_READ, etc.) is usually defined using an instance of SelectorHandler.
- This is where the decision of attaching an object to SelectionKey occurs.

SelectionKeyHandler

- A SelectionKeyHandler is used to handle the life cycle of a SelectionKey.
- Operations likes cancelling, registering or closing of SelectionKeys are handled by a SelectionKeyHandler.

InstanceHandler

- An InstanceHandler is where one or several ProtocolChain(s) are created and cached.
- An InstanceHandler decides if a stateless or stateful ProtocolChain needs to be created.
- Note: InstanceHandler is being renamed to a ProtocolChainInstanceHandler for improved clarity

Pipeline

- An interface used as a wrapper around any kind of thread pool.
- There are several implementation of Pipelines in Grizzly 1.5.
- The best performing implementation is the default configured Pipeline.

ProtocolChain

- A ProtocolChain implements the "Chain of Responsibility" pattern (for more info, take a look at the classic "Gang of Four" design patterns book).
- The ProtocolChain API models a computation as a series of "protocol filter" that can be combined into a "protocol chain".

ProtocolFilter

- A ProtocolFilter encapsulates a unit of processing work to be performed, whose purpose is to examine and/or modify the state of a transaction that is represented by a ProtocolContext.
- Individual ProtocolFilter(s) can be assembled into a ProtocolChain.

ProtocolFilter

- The API for ProtocolFilter consists of a two methods:

  execute(Context)

  postExecute(Context)

- which are passed a "protocol context" containing the dynamic state of the computation

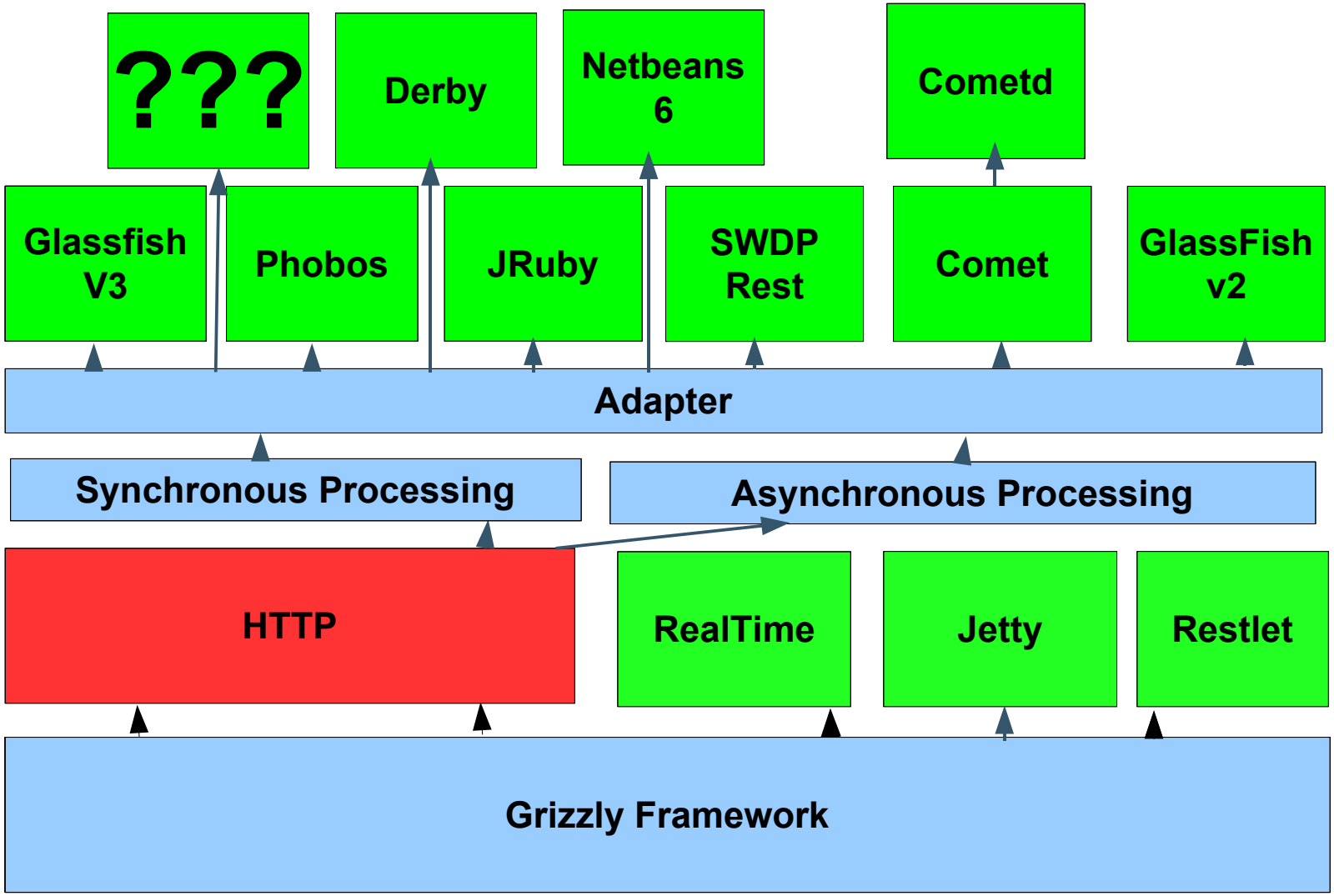
Example 1 - TCP

- By default, the Grizzly Framework bundles default implementation for TCP and UPD transport. The TCPSelectorHandler is instantiated by default.
- As an example, supporting the TCP protocol should only consist of adding the appropriate ProtocolFilter like:

Example – 1 TCP (Cont.)

```
Controller con = new Controller();
con.setInstanceHandler(new DefaultInstanceHandler(){
  public ProtocolChain poll() {
      ProtocolChain protocolChain = protocolChains.poll()
      if (protocolChain == null){
         protocolChain = new DefaultProtocolChain();
         protocolChain.addFilter(new ReadFilter());
         protocolChain.addFilter(new LogFilter());
}
      return protocolChain;
   }
});
```

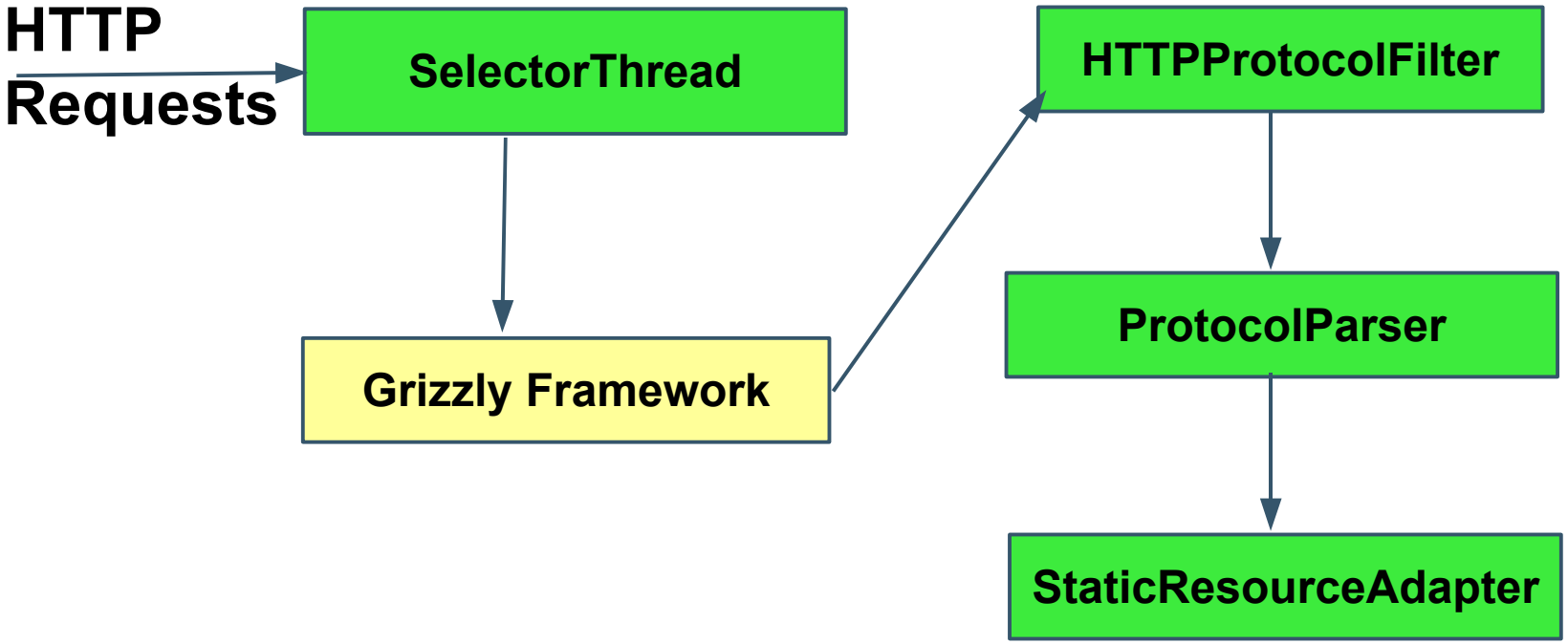
Architecture – HTTP layer



Grizzly HTTP layer

- Lightweight HTTP 1.0/1.1 based server
- Extremely easy to embed.
- Small footprint.
- Performance is extremely good, but to see it you need to come to:
  - > **Session TS-2992 Tricks and Tips with NIO, Using the Grizzly Framework**
  - > **Free Grizzly T-shirt!!!**
- **Good performance apply to both Synchronous processing and Asynchronous Processing**

Example: Grizzly Web Server



Grizzly HTTP layer

- Easy to embedded. Only have to interact with one object: SelectorThread
- Write an implementation of com.sun.grizzly.tcp.Adapter class.
- The Adapter is the glue code between the HTTP layer and the program that embed Grizzly.
- In the following example, the default StaticResourcesAdapter is used

Example – 1 Static Resource Web Server

```
SelectorThread selectorThread = new SelectorThread();
selectorThread.setPort(port);
selectorThread.setAdapter(
                        new StaticResourcesAdapter());
selectorThread.setWebAppRootPath(folder);
selectorThread.initEndpoint();
selectorThread.startEndpoint();
```

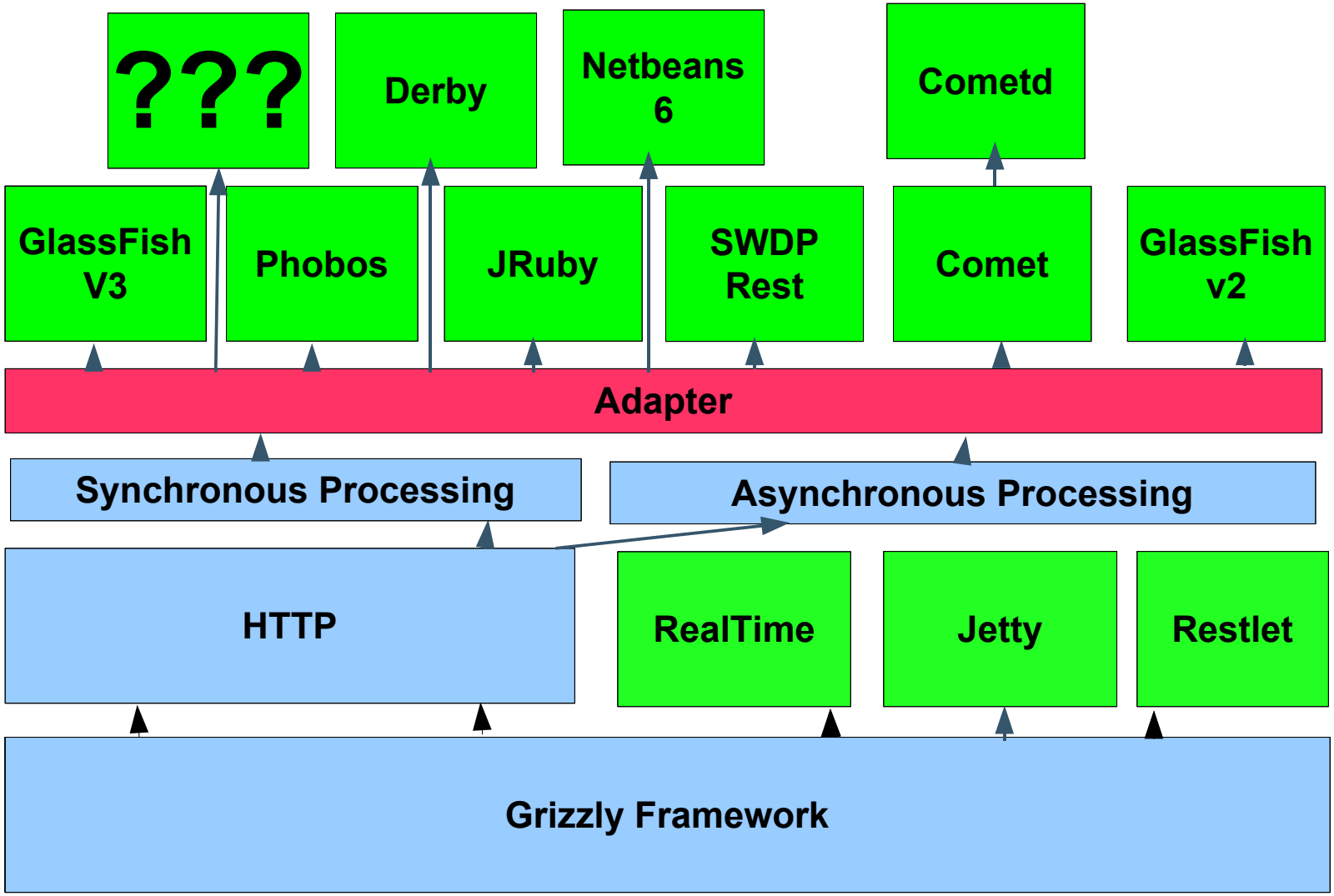
Asynchronous Request Processing

- Allow for “parking” a request; a type of “continuation” at the request processing level
- The goal is to be able to build, on top of Grizzly, a scalable ARP implementation that doesn't hold one thread per connection, and achieve as closer as possible the performance of synchronous request processing (SRP).

Example – 2 Asynchonous Request Processing

```
SelectorThread selectorThread = new SelectorThread();
selectorThread.setPort(port);
selectorThread.setWebAppRootPath(folder);
selectorThread.setAdapter(
                        new StaticResourcesAdapter());
AsyncHandler asyncHandler = new DefaultAsyncHandler()
asyncHandler.addAsyncFilter(new CometAsyncFilter());
selectorThread.setAsyncHandler(asyncHandler);
selectorThread.initEndpoint();
selectorThread.startEndpoint();
```

Architecture – Adapter



Architecture - Adapter

Main entry point for most of HTTP based server

- Most Grizzly 1.0 implementation write their own com.sun.grizzly.tcp.Adapter implementation.
  - Project Phobos in Netbeans
  - Netbeans 6 Embedded Web Server
  - JRuby on Grizzly
- Simple Interface

```
public void service(Request req,Response res);
```

- Request contains all HTTP information like:
  - Method: GET/POST/TRACE
  - Headers: content-length, content-type, etc.
- Works at the bytes level.

Source: Please add the source of your data here

Example – StaticResourceAdapter

```
public void service(Request req, final Response res)  {
   MessageBytes mb = req.requestURI();
   ByteChunk requestURI = mb.getByteChunk();
   String uri = req.requestURI().toString();
    ....
    res.setStatus(200);
    res.setContentType(ct);
    res.sendHeaders();
    ....
    res.doWrite(chunk);
    res.finish()
}
```

Architecture - Adapter

- But this approach is problematic if you need to embedded more than one http based implementation because you needs one adapter per implementation
  - One for Phobos
  - One for Comet
  - One for JRuby on Rail
- They cannot listen to the same http port!
- Adapter notes cannot be shared.
- Solution: GlassFish V3 project!

Source: Please add the source of your data here

Agenda

Introduction

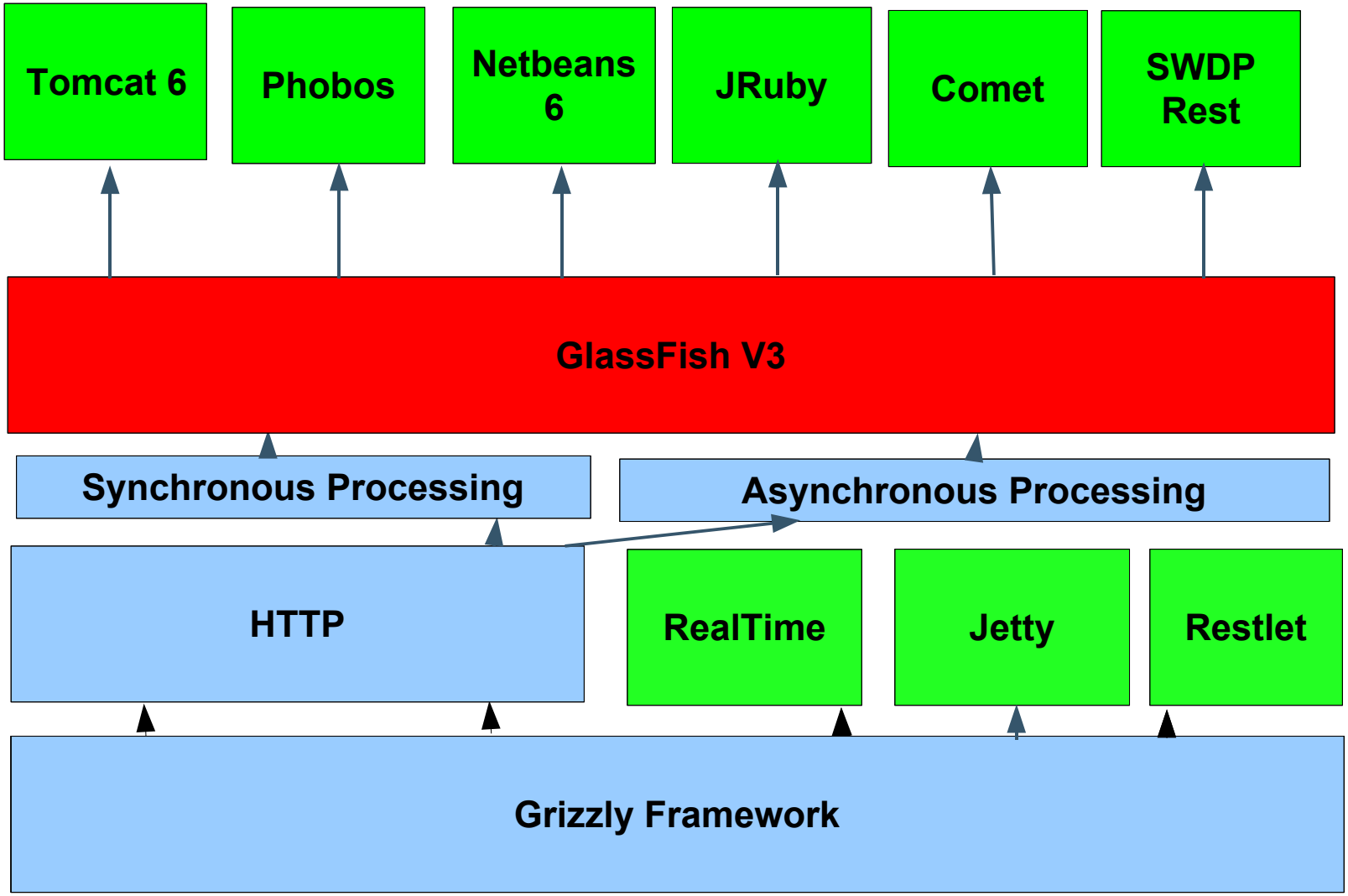
What is Project Grizzly

Architecture

GlassFish V3

Q&A

Architecture – GlassFish V3



Advantages

- Same performance
- Same port, different context
- Adapter Notes management (caching)
- ThreadLocal storage management
- Common administration : deploy, undeploy...
- Container loading/unloading
- Adapter boilerplate reduced
- Intra-adapter communication

Application adapter

- In GlassFish V3, each application can register its adapter.
- Adapter have context root
- Requests are dispatched based on the registered context roots
- Registration/Unregistration of Adapter instances is automatically handled by the runtime
- GlassFish has no knowledge of the target container type, Adapter is the interface

GlassFish V3 Adapter

```
@Contract
public interface Adapter extends com.sun.grizzly.tcp.Adapter {

    /**
     * Returns the context root for this adapter
     * @return context root
     */
    public String getContextRoot();
}
```

Containers

- Containers are the runtime for application.
- Each application type has a corresponding container
- Implementation of a container are discovered by GlassFish through :
  META-INF/services/com.sun.enterprise.v3.api.Container
- If using HK2 and maven 2, packaging is greatly simplified.

Container

```
public @interface Container {
    /**
     * Defines the short name for the container type.
     * @return the container type
     */
    String type();
    /**
     * @return the deployer class name
     */
    String deployerImpl();
}
```

Example : RoR

```
@Service
@Container(type="jruby",
   deployerImpl="com.sun.enterprise.rails.RailsDeploy
   er", infoSite="http://jruby.dev.java.net")
public class RailsContainer implements
   ContractProvider, PostConstruct, PreDestroy {
...
}
```

Rails Adapter

- Thanks to Naoto Takai !

```
import com.sun.grizzly.rails.RailsAdapter;
import com.sun.grizzly.rails.RubyObjectPool;

public class RailsApplication extends RailsAdapter
    implements ApplicationContainer {

    public String getContextRoot() ...
    }
```

Summary

- The Project Grizzly is extendable:
  - At the TCP/UDP level
  - At the HTTP level
- Easy to embed
  - Less that 10 lines.
  - Small footprint (~500k)
- Can support multiple extension via the h2k project.

Call to Action

- Download HK2/GlassFish v3 and experience the fastest web container on the planet
- Join Project Grizzly and be added to Project Grizzly mailing lists
- Join Project HK2 and be added to Project HK2 mailing lists

Where to find more infomation

- Project Grizzly home page
  https://grizzly.dev.java.net
- Project HK2 home page
  https://hk2.dev.java.net/
- Jeanfrancois Arcand's blog
  http://weblogs.java.net/blog/jfarcand
- Jerome Dochez's blog
  http://blogs.sun.com/dochez/
- Project Grizzly mailing lists,
  dev@grizzly.dev.java.net and/or
  users@dev.grizzly.java.net

Q&A

Optional Speaker Names Here