



[java.com.sun/javaone](http://java.com.sun/javaone)

# Advanced Java™ NIO Technology-Based Applications Using the Grizzly Framework

JeanFrancois Arcand,  
Oleksiy Stashok,  
Sun Microsystems

TS-4883



Learn how easy it can be to write scalable client-server applications with the Grizzly Framework.



GOAL

# Agenda

- **What is the Project Grizzly**
- The Server Components
- The Client Components
- Asynchronous Read and Write Queue
- Synchronous Read and Write using Temporary Selectors
- Persisting your data using ThreadAttachment
- HTTP Modules
- NIO.2 Support
- Summary

# What is Project Grizzly

- Open Source Project on java.net, <https://grizzly.dev.java.net>
- Open Sourced under CDDL/LGPL license.
- Very open community policy.
  - All project communications are done on Grizzly mailing list. No internal, off mailing list conversations.
  - Project meetings open to anyone (public conference call).
- Project decisions are made by project member votes.
- Sun and non Sun committers.

# What is Project Grizzly

- Uses Java NIO primitives and hides the complexity programming with Java NIO.
- Easy-to-use high performant APIs for TCP, UDP and SSL communications.
- Brings non-blocking sockets to the protocol processing layer.
- Utilizes high performant buffers and buffer management.
- Choice of several different high performance thread pools.
- Ship with an HTTP module which is really easy to extend.
- Use the Framework, but **STILL** have a way to control the IO layer.

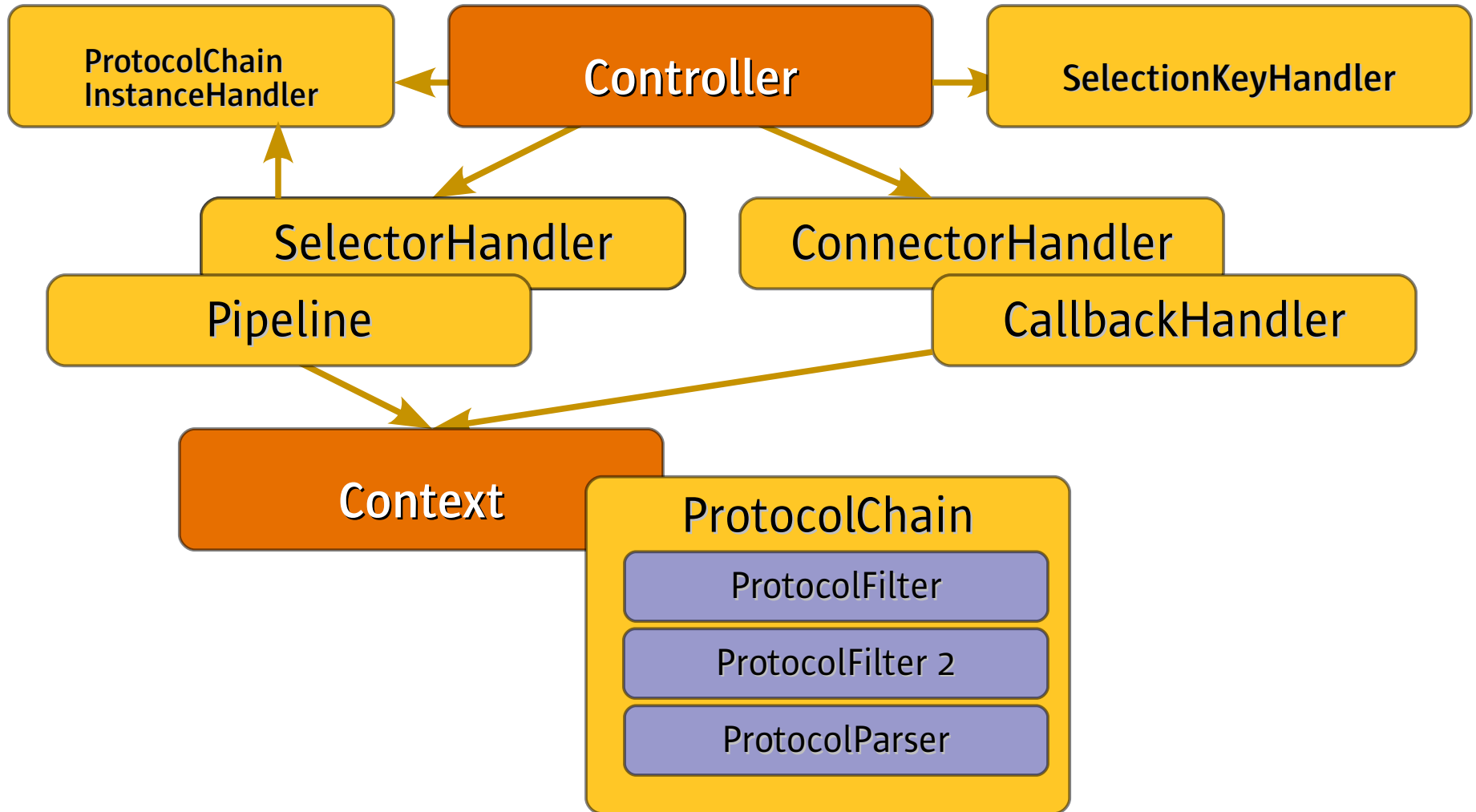
# Agenda

- **What is the Grizzly Framework**
- **The Server Components**
- **The Client Components**
- **Asynchronous Read and Write Queue**
- **Synchronous Read and Write using Temporary Selectors**
- **Persisting your data using ThreadAttachment**
- **HTTP Modules**
- **NIO.2 Support**
- **Summary**

## Example 1 – Server, how easy it is

- The Grizzly Framework bundles default implementation for TCP, UDP and TLS/SSL transports.
- Every supported transport SelectorHandler should be added to a Controller
- By default, you can use TCPSelectorHandler, UDPSelectorHandler and TLSSelectorHandler.
- You can also create your own by implementing the SelectorHandler interface.
- In Grizzly, **EVERYTHING** is customizable. If the default isn't doing what you need, customize it!

# Grizzly's Monster face



# Example: Simple log server over TCP

```
Controller controller = new Controller();
controller.setProtocolChainInstanceHandler(new
    DefaultProtocolChainInstanceHandler() {

    ProtocolChain protocolChain;
    // Stateless ProtocolChain
    public ProtocolChain poll() {

        if(protocolChain == null) {
            protocolChain = new DefaultProtocolChain();
            protocolChain.addFilter(new ReadFilter());
            protocolChain.addFilter(new LogFilter());
        }

        return protocolChain;
    }
});
```

# Example: Supporting TCP and UDP

```

Controller controller = new Controller();

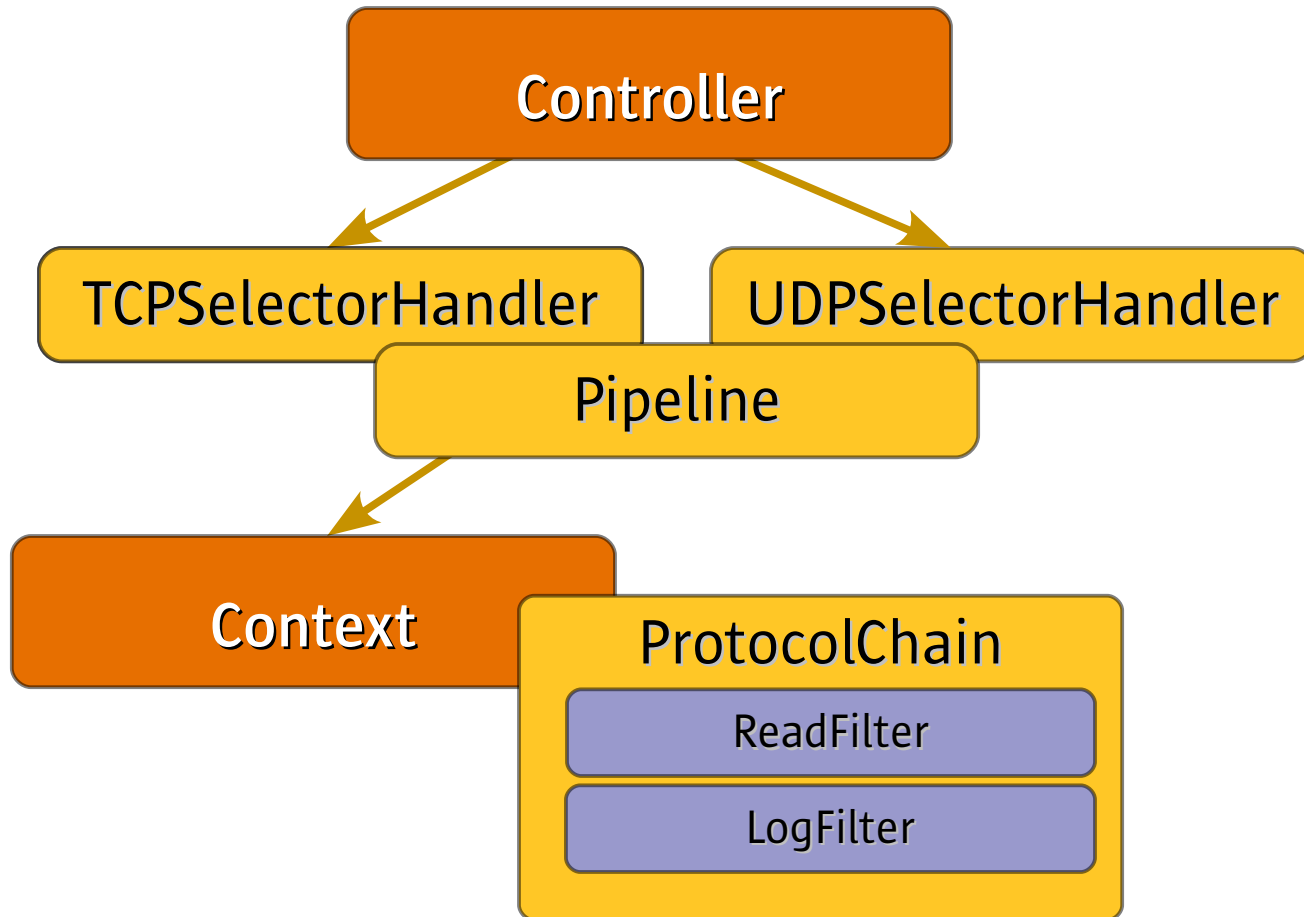
controller.addSelectorHandler(new TCPSelectorHandler());
controller.addSelectorHandler(new UDPSelectorHandler());
controller.setProtocolChainInstanceHandler(new
    DefaultProtocolChainInstanceHandler() {

    ProtocolChain protocolChain;
    // Stateless ProtocolChain
    public ProtocolChain poll() {

        if(protocolChain == null) {
            protocolChain = new DefaultProtocolChain();
            protocolChain.addFilter(new ReadFilter());
            protocolChain.addFilter(new LogFilter());
        }

        return protocolChain;
    }
});

```



# Example: Why not TLS?

```

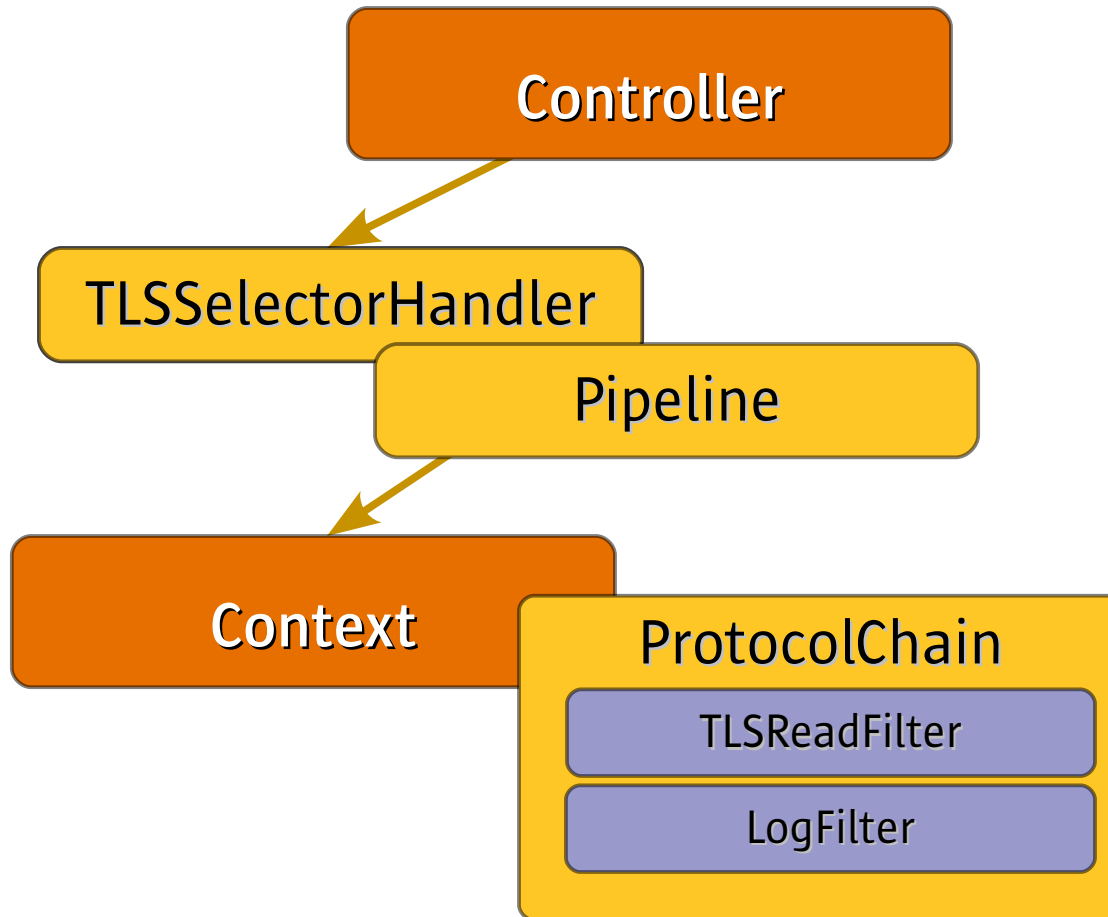
Controller controller = new Controller();

controller.addSelectorHandler(new TLSSelectorHandler());
controller.setProtocolChainInstanceHandler(new
    DefaultProtocolChainInstanceHandler() {

    ProtocolChain protocolChain;
    // Stateless ProtocolChain
    public ProtocolChain poll() {

        if(protocolChain == null) {
            protocolChain = new DefaultProtocolChain();
            protocolChain.addFilter(new TLSReadFilter());
            protocolChain.addFilter(new LogFilter());
        }

        return protocolChain;
    }
});
  
```



# Controller

- The main entry point when working with Grizzly.
- A Controller is composed of
  - SelectorHandler
  - ConnectorHandler
  - SelectionKeyHandler
  - ProtocolChainInstanceHandler
  - ProtocolChain
  - Pipeline
- By default, all these interfaces have ready to use implementations.
- All of these components are customizable using the Grizzly Framework

# SelectorHandler (optional)

- A SelectorHandler handles all `java.nio.channels.Selector` operations.
- The logic for processing of `SelectionKey` interest (`OP_READ`, `OP_WRITE`, etc) is usually defined using an instance of `SelectorHandler`
- Ex: `TCPSelectorHandler`, `UDPSelectorHandler`

```
public void preSelect(Context ctx)
    throws IOException;
```

```
public Set<SelectionKey> select(Context ctx)
    throws IOException;
```

```
public void postSelect(Context ctx)
    throws IOException;
```

## SelectionKeyHandler (optional)

- A SelectionKeyHandler used to handle the life-cycle of a SelectionKey. This object is responsible of canceling, registering or closing (timing out) SelectionKey(s).
- You can have one SelectionKey per SelectorHandler, or globally shared amongst them.

```
public void register(Iterator<SelectionKey> it, int  
selectionKeyOps) ;
```

```
public void expire(Iterator<SelectionKey> it) ;
```

```
public void cancel(SelectionKey key) ;
```

```
public void close(SelectionKey key) ;
```

# 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 Context.
- Most Grizzly based application only have to write an implementation of one ProtocolFilter.
- Individual ProtocolFilters can be assembled into a ProtocolChain.
- **The only classes you might have to write!**

```
public boolean execute(Context ctx) throws  
IOException;
```

```
public boolean postExecute(Context ctx) throws  
IOException;
```

# ProtocolParser

- A specialized ProtocolFilter that knows how to parse bytes into protocol unit of data.
- Simply protocol implementation in Grizzly. Split protocol parsing state into steps:
  - start processing a buffer
  - enumerate the message in the buffer
  - and end processing the buffer.
- The buffer can contain 0 or more complete messages, and it's up to the protocol parser to make sense of that.

```
public T getNextMessage();
```

```
public boolean hasNextMessage();
```

```
public void startBuffer(ByteBuffer bb);
```

```
public boolean releaseBuffer();
```

# ProtocolChain (optional)

- 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 filters”, that can be combined into a “protocol chain”.

```
public boolean addFilter(ProtocolFilter pf);
```

```
public boolean removeFilter(ProtocolFilter pf);
```

```
public void execute(Context context) throws Exception;
```

# ProtocolChainInstanceHandler (optional)

- A ProtocolChainInstanceHandler is where one or several ProtocolChains are created and cached.
- A ProtocolChainInstanceHandler decides if a stateless or stateful ProtocolChain needs to be created:
  - Stateless: One ProtocolChain instance shared amongst Threads.
  - Statefull: One ProtocolChain instance per Thread

```
public ProtocolChain poll();
```

```
public boolean offer(ProtocolChain pc);
```

# Pipeline (optional)

- An interface is used as a wrapper around any kind of thread pool.
- The best performing implementation is the default configured Pipeline.
- Can have one Pipeline per SelectorHandler or a shared one amongst them.

```
public void execute(E task) throws  
                                PipelineFullException;  
  
public E waitForIoTask() ;
```

# Agenda

- **What is the Grizzly Framework**
- **The Server Components**
- **The Client Components**
- **Asynchronous Read and Write Queue**
- **Synchronous Read and Write using Temporary Selectors**
- **Persisting your data using ThreadAttachment**
- **HTTP Modules**
- **NIO.2 Support**
- **Summary**

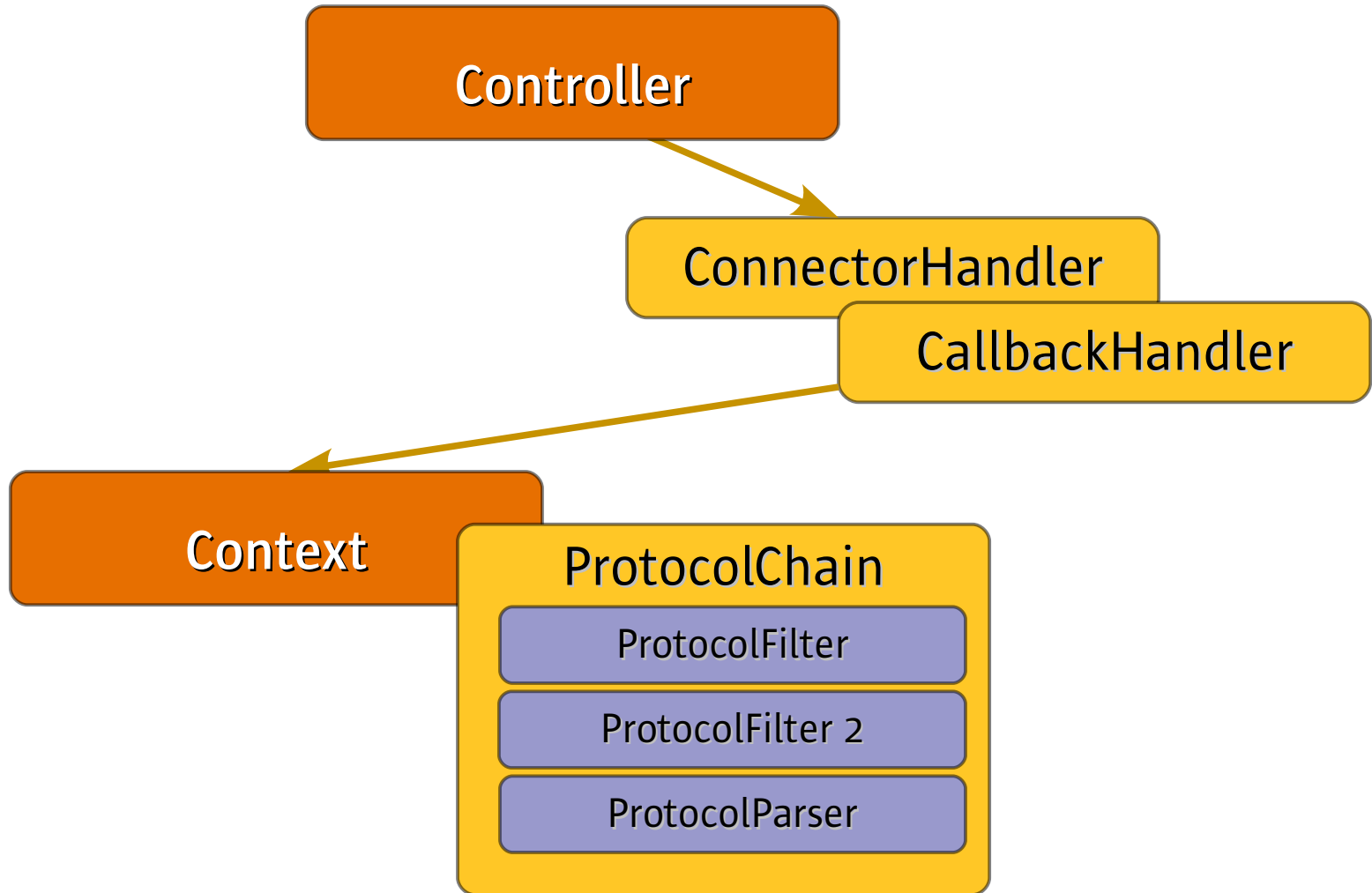
## Example 2 – Client: synchronous mode

```
Controller ctrl = new Controller();
ctrl.addSelectorHandler(new TCPSelectorHandler(true));
startGrizzly(ctrl);

ConnectorHandler connection =
    ctrl.acquireConnectorHandler(Protocol.TCP);

connection.connect(new InetSocketAddress(host, port));
connection.write(outputByteBuffer, true);
connection.read(inputByteBuffer, true);
connection.close();
ctrl.releaseConnectorHandler(connection);
```

# Grizzly's Client Monster face



## Example 2 – Client: asynchronous mode

```
Controller ctrl = new Controller();
ctrl.addSelectorHandler(new TCPSelectorHandler(true));
startGrizzly(ctrl);
ConnectorHandler connection =
    ctrl.acquireConnectorHandler(Protocol.TCP);
connection.connect(new InetSocketAddress(host, port),
    new CustomCallbackHandler(connection));
connection.write(giantByteBuffer, false);

// continue, while ByteBuffer will be written
// asynchronously
```

## Example 2 – Client: asynchronous mode (cont.)

```

public class CustomCallbackHandler implements
    CallbackHandler<Context> {
    private ConnectorHandler connection;

    CustomCallbackHandler(ConnectorHandler connection){
        this.connection = connection;
    }

    public void onWrite(IOEvent<Context> ioEvent) {
        connection.write(giantByteBuffer, false);
        if (!giantByteBuffer.hasRemaining()) {
            notifyAsyncWriteCompleted();
        }
    }

    // Skip other CallbackHandler methods implementation...
}

```

# ConnectorHandler

- Client side connection unit, which implements basic I/O functionality: connect, read, write, close.
- Provides possibility of working in both synchronous and asynchronous modes (e.g. blocking or non blocking read/write).
- Asynchronous operation execution could be controlled by CallbackHandler with user custom code, or automatically by framework, using asynchronous read and write queues.
- Grizzly Framework contains default TCP,TLS UDP ConnectorHandler implementations.

# CallbackHandler

- Handles client side asynchronous I/O operations: connect, read, write. Corresponding CallbackHandler method will be called, when NIO channel is ready to perform the operation.
- Asynchronous events could be either processed inside CallbackHandler or propagated to a ProtocolChain.
- When processing asynchronous event inside CallbackHandler, be careful with SelectionKey interests registration.

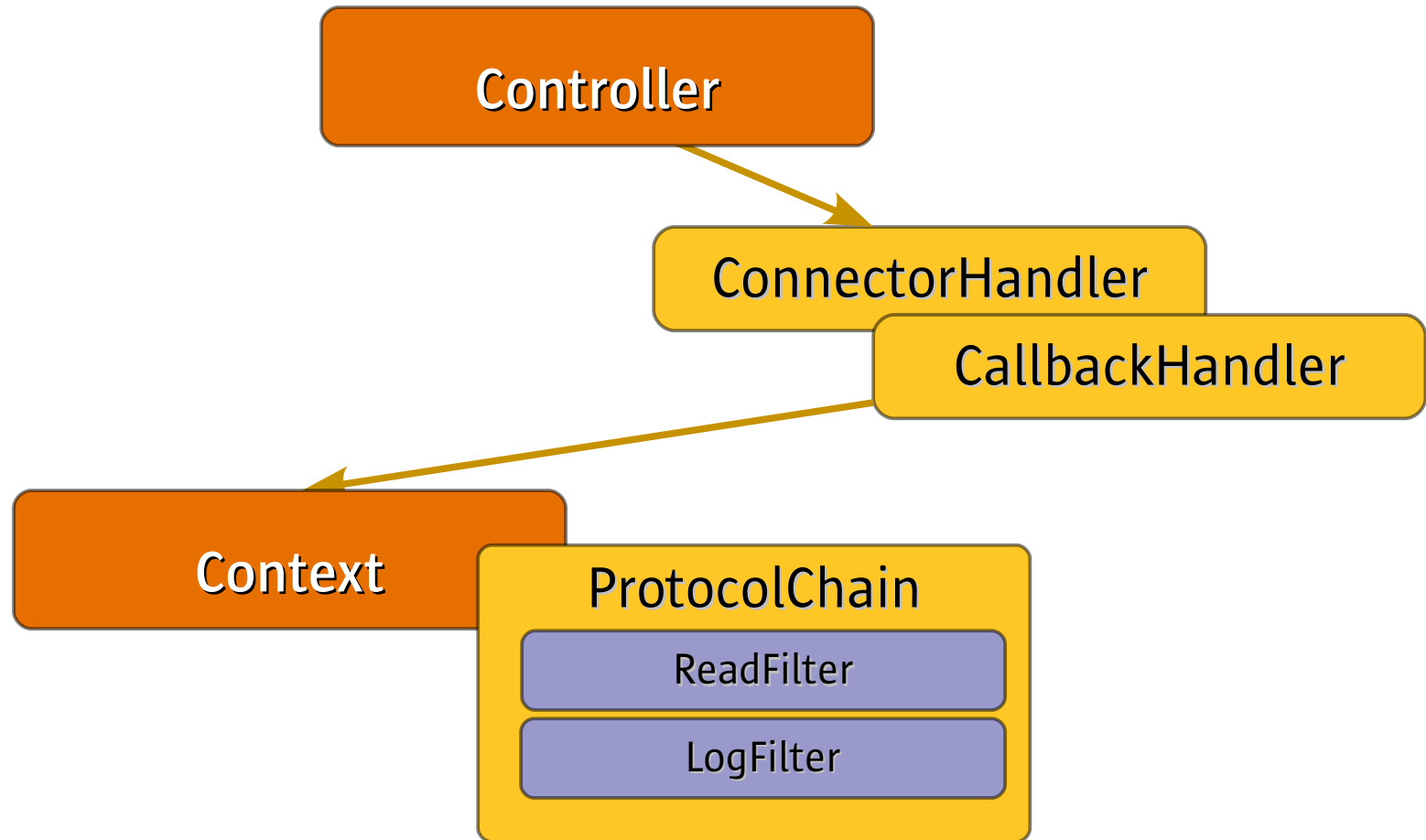
```
public void onConnect(IOEvent<E> ioEvent);
```

```
public void onRead(IOEvent<E> ioEvent);
```

```
public void onWrite(IOEvent<E> ioEvent);
```

# Client and server side logic unification

- There are use cases, where both client and server sides should process requests similar way, for example CORBA or SIP.
- CallbackHandler are able to redirect a request processing to a ProtocolChain, so high level protocol logic, which is implemented in ProtocolChain filters will be accessible from client side API too.



## Example 3 – Client and server side logic unification

```
public class CallbackHandlerToProtocolChain implements
    CallbackHandler<Context> {
    public void onRead(IOEvent<Context> context) {
        Context context = ioEvent.attachment();
        context.getProtocolChain().execute(context);
    }

    // Skip other CallbackHandler methods implementation...
}
```

# Agenda

- **What is the Grizzly Framework**
- **The Server Components**
- **The Client Components**
- **Asynchronous Read and Write Queue**
- **Synchronous Read and Write using Temporary Selectors**
- **Persisting your data using ThreadAttachment**
- **HTTP Modules**
- **NIO.2 Support**
- **Summary**

# Asynchronous write queue

- Provides user the easy way for sending data asynchronously and be notified, when operation will be either successfully completed or interrupted due to I/O error.
- Asynchronous write queue is thread safe. Data from different buffers will never be intermixed.
- Asynchronous write queue processors provide easy and pluggable way to compress, encrypt output data just before it will be sent over NIO channel.
- Grizzly Framework has default asynchronous write queue implementation for UDP and TCP protocols.

# Asynchronous read queue

- Provides user the easy way for receiving data asynchronously and be notified, when operation will be either successfully completed or interrupted due to I/O error.
- Asynchronous read queue is thread safe. Next buffer will start to fill up only after current one will be completed.
- Asynchronous read queue processors provide easy and pluggable way to decompress, decrypt input data just after it was read from NIO channel.

## Asynchronous read queue (cont.)

- Asynchronous read conditions provide user possibility to set custom “read complete” condition. By default asynchronous read will consider completed, when input ByteBuffer will become full.
- Grizzly Framework has default asynchronous read queue implementation for UDP and TCP protocols.

## Example 6 – Asynchronous queue: write

```

class AsyncQueueEchoFilter implements ProtocolFilter {
    public boolean execute(Context ctx)
        throws IOException {
        ByteBuffer buffer = getByteBuffer(ctx);
        // we pass null for callback notificator and
        // data processor. Last true parameter means buffer
        // will be cloned before added to the queue
        ctx.getAsynQueueWritable().writeToAsyncQueue(
            buffer, null, null, true);
    }

    public boolean postExecute(Context ctx)
        throws IOException {
        return true;
    }
}

```

## Example 6 – Asynchronous queue: read

```
ConnectorHandler connection =
    ctrl.acquireConnectorHandler(Protocol.TCP);
connection.connect(new InetSocketAddress(host, port));

// request notification, when byte buffer will be full
AsyncReadCallbackHandler notificationHandler =
    new CustomAsyncReadCallbackHandler();
connection.readFromAsyncQueue(bigByteBuffer,
    notificationHandler);
```

## Example 6 – Asynchronous queue: read (cont.)

```

class CustomAsyncReadCallbackHandler implements
    AsyncReadCallbackHandler {

    public void onReadCompleted(SelectionKey key,
        SocketAddress srcAddress, ByteBuffer buffer) {
        processReadData (buffer) ;
    }

    public void onIOException (IOException ioException,
        ...
    }
}
  
```

# Agenda

- **What is the Grizzly Framework**
- **The Server Components**
- **The Client Components**
- **Asynchronous Read and Write Queue**
- **Synchronous Read and Write**
- **Persisting your data using ThreadAttachment**
- **HTTP Modules**
- **NIO.2 Support**
- **Summary**

# Synchronous Read and Write

- There is some situations where registering a Channel on more than one Selector improve performance:
  - Instead of registering a SelectionKey using the Selector used by the acceptor thread (the Selector than handled the OP\_ACCEPT), get a Selector from a pool and try to read/write more bytes using it.

```

readSelector = SelectorFactory.getSelector();
tmpKey = socketChannel.register
        (readSelector, SelectionKey.OP_READ)
tmpKey.interestOps(tmpKey.interestOps() |
    SelectionKey.OP_READ);
int code = readSelector.select(readTimeout);
tmpKey.interestOps(tmpKey.interestOps() &
    (~SelectionKey.OP_READ));
while (count > 0){
    count = socketChannel.read(byteBuffer);
}
  
```

# Synchronous Read and Write (Con't)

- Grizzly offers two utility classes for supporting synchronous read and write:

- `com.sun.grizzly.util.InputReader`

```
{
```

```
    InputReader syncReader = new InputReader();  
    // Block at most 5 seconds.  
    syncReader.read(bb, 5, TimeUnit.SECONDS);
```

```
}
```

- `com.sun.grizzly.util.OutputWriter`

```
{
```

```
    // Block at most 5 seconds.  
    OutputWriter.write(bb, 5, TimeUnit.SECONDS);
```

```
}
```

# Agenda

- **What is the Grizzly Framework**
- **The Server Components**
- **The Client Components**
- **Asynchronous Read and Write Queue**
- **Synchronous Read and Write using Temporary Selectors**
- **Persisting your data using ThreadAttachment**
- **HTTP Modules**
- **NIO.2 Support**
- **Summary**

# ThreadAttachment

- Between OP\_READ and OP\_WRITE, some protocol needs to keep some states (remaining bytes inside a byte buffer, attributes, etc.).
- In Grizzly, the default is to associate one Byte Buffer per Thread. That means a byte buffer cannot be cached as its Thread can always be re-used for another transaction.
- To persist the byte buffer amongs transactions, a Thread Attachment can be used:

```

{
    WorkerThread wt =
        (WorkerThread) Thread.currentThread();
    ThreadAttachment = wt.detach();
}
  
```

# ThreadAttachment (Con't)

- What happens is internally, all attributes associated with the WorkerThread are 'detached', and new instance recreated
  - Warning: A new ByteBuffer will be created!!
- The ThreadAttachment can be attached to a SelectionKey, and next time an OP\_READ/OP\_WRITE happens, you are guarantee the value will be re-used.

```

{
    ta.setAttribute(keepAliveCount, 10);
    ctx.getSelectionKey().attach(ta);
    ...
    ThreadAttachment ta = (ThreadAttachment)
        ctx.getSelectionKey().attachment();
    int keepAliveCount = (int)ta.getAttribute();
}
  
```

# Agenda

- **What is the Grizzly Framework**
- **The Server Components**
- **The Client Components**
- **Asynchronous Read and Write Queue**
- **Synchronous Read and Write using Temporary Selectors**
- **Persisting your data using ThreadAttachment**
- **HTTP Modules**
- **NIO.2 Support**
- **Summary**

# HTTP modules

- The Grizzly Framework also have an HTTP framework that can be used to build Web Server
  - This is what GlassFish v1|2|3 build on top of.
  - More specialized modules are also available like Comet (Async HTTP).
- Simple interface to allow customization of the HTTP Protocol
  - **GrizzlyRequest**: A utility class to manipulate the HTTP protocol request.
  - **GrizzlyResponse**: A utility class to manipulate the HTTP protocol response.
  - **GrizzlyAdapter**: A utility class to manipulate the HTTP request/response object.

# HTTP protocol module (Con't)

```
public class FileAdapter extends GrizzlyAdapter{
    public void service(GrizzlyRequest req,
                       GrizzlyResponse res){
        String fileUri = req.getRequestURI();

        FileChannel file = ....;
        fileChannel.transferTo(...);

        res.flush();
        res.close();
    }
}
```

## Example: Jersey ([jersey.dev.java.net](http://jersey.dev.java.net)).

- Jersey is the open source JAX-RS (JSR 311) Reference Implementation for building RESTful Web services.
- By default, Jersey ship with support for Grizzly HTTP WebServer :
  - Implement the GrizzlyAdapter interface, use the GrizzlyRequest and Response object to handle the http protocol.
- In less than 20 minutes, Jersey was running with Grizzly!

## Example: Jersey (jersey.dev.java.net).

```
public void service(GrizzlyRequest request,
                   GrizzlyResponse response) {
    try {
        requestAdaptor = new GrizzlyRequestAdaptor(
            application.getMessageBodyContext(), request);

        responseAdaptor =
            new GrizzlyResponseAdaptor(response,
                                       application.getMessageBodyContext(),
                                       requestAdaptor);

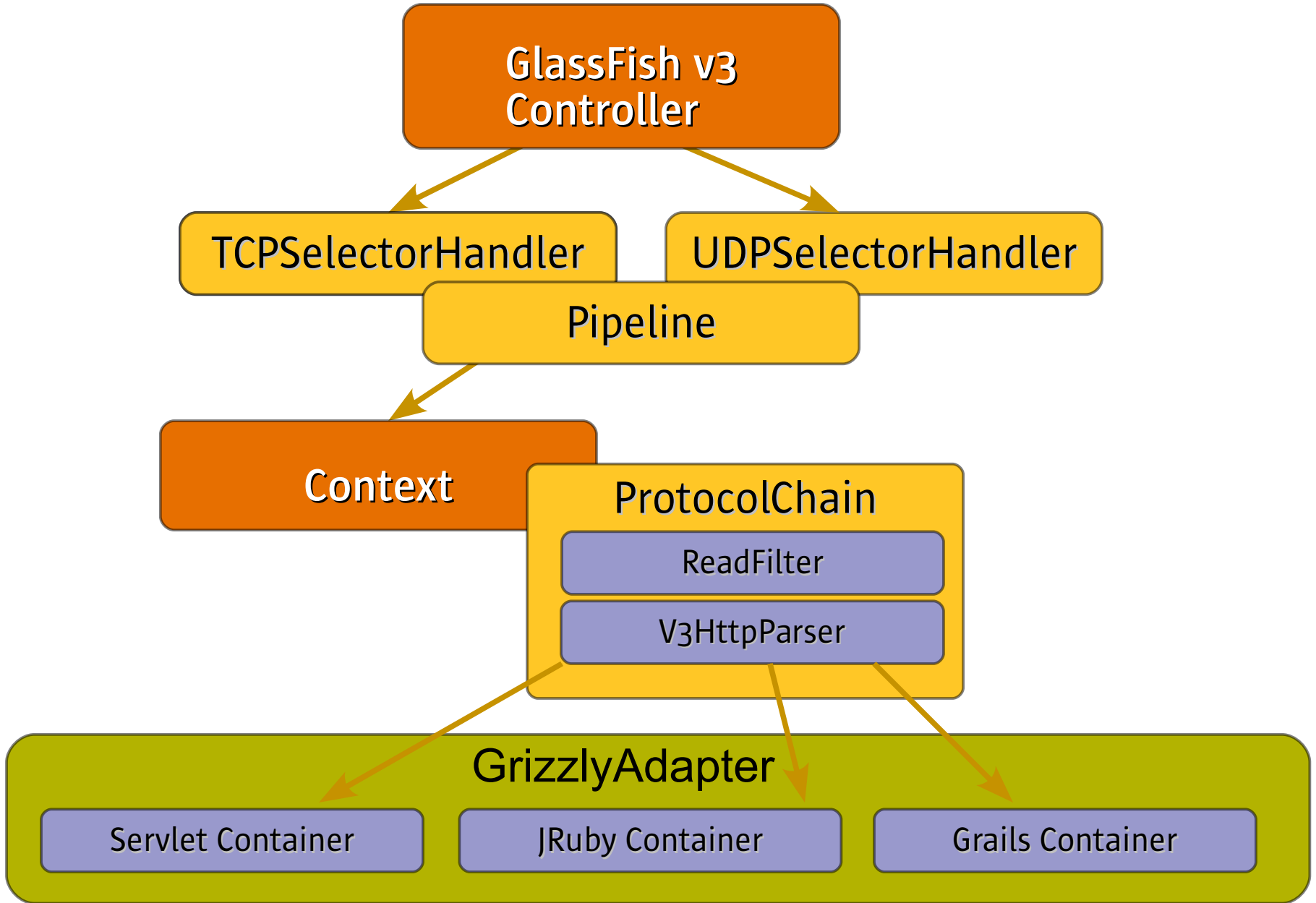
        application.handleRequest(requestAdaptor,
                                   responseAdaptor);
    }
}
```

# Example: Jersey (Con't).

```
protected void commitStatusAndHeaders() throws IOException {
    response.setStatus(this.getStatus());

    for (Map.Entry<String, List<Object>> e :
        this.getHttpHeaders().entrySet()) {
        String key = e.getKey();
        for (Object value: e.getValue()) {
            response.addHeader(key,value.toString());
        }
    }

    String contentType = response.getHeader("Content-Type");
    if (contentType != null) {
        response.setContentType(contentType);
    }
}
```



# GlassFish v3 Demo

DEMO

# Agenda

- **What is the Grizzly Framework**
- The Server Components
- The Client Components
- Asynchronous Read and Write Queue
- Synchronous Read and Write using Temporary Selectors
- Persisting your data using ThreadAttachment
- HTTP Modules
- **NIO.2 Support**
- Summary

## NIO.2: Asynchronous I/O (jsr-203)

- NIO.2 is:
  - An API for asynchronous (as opposed to polled, non-blocking) I/O operations on both sockets and files.
  - The completion of the socket-channel functionality defined in JSR-51, including the addition of support for binding, option configuration, and multicast datagrams.
- Starting in Grizzly 1.8.0, Asynchronous I/O is supported if you are running JDK 7.
- You can easily mix non blocking with asynchronous I/O.

## NIO.2: Asynchronous I/O (JDK 7.0)

- Grizzly supports NIO.2. Switching an application from using NIO.1 to NIO.2 is quite simple
- Bytes are delivered the same way, independently of NIO.1 or NIO.2:

```
// Instead of TCPSelectorHandler  
controller.addIOHandler(new TCPAIOHandler());
```

```
//Instead of using ReadFilter  
AIOReadFilter readFilter = new AIOReadFilter();  
protocolChain.addFilter(readFilter);
```

# Agenda

- **What is the Grizzly Framework**
- **The Server Components**
- **The Client Components**
- **Asynchronous Read and Write Queue**
- **Synchronous Read and Write using Temporary Selectors**
- **Persisting your data using ThreadAttachment**
- **HTTP Modules**
- **NIO.2 Support**
- **Summary**

# Summary

- **Project Grizzly is a simple NIO framework.**
  - With less than 50 lines, you can create power client or server side components.
  - Support NIO.1 and NIO.2.
  - You have control of the framework: everything can be customized!
- **HTTP module makes it easy to build WebServer extension**
  - Jersey, blog-city.com, Netbeans, Ning, Red Hat REST Impl, RESTlet, GlassFish v1|2|3, Sailfin Load Balancer, JXTA, and many many more!
- **Healthy community: get a response in less than 2 hours :-)**

# For More Information

- Grizzly:
  - <http://grizzly.dev.java.net>
- Jeanfrancois's blog:
  - <http://weblogs.java.net/blog/jfarcand/>
- Alexey's blog:
  - <http://blogs.sun.com/oleksiys/>
- Project Grizzly mailing lists,
  - [dev@grizzly.dev.java.net](mailto:dev@grizzly.dev.java.net)
  - [users@dev.grizzly.java.net](mailto:users@dev.grizzly.java.net)

# THANK YOU



Jeanfrancois Arcand  
Oleksiy Stashok,  
Sun Microsystems

TS-4883

