CRISPY v.1.1.0

Project Documentation

Crispy

28 August 2006

Table of Contents

1.1 Overview

Overview

CRISPY = Communication per Remote Invocation for different kinds of Services via ProxYs.

The intention for this project is a **very simple** API to call different kinds of services (provider/technology). Crispy's aims is to provide a single point of entry for remote invocation for a wide number of transports: eg. RMI, EJB, JAX-RPC or XML-RPC. It works by using properties to configure a service manager, which is then used to invoke the remote API. Crispy is a simple Java codebase with an API that sits between your client code and the services your code must access. It provides a layer of abstraction to decouple client code from access to a service, as well as its location and underlying implementation. The special on this idea is, that these calls are simple Java object calls (remote or local calls are transparent).

From Crispy supported transport provider are:

- Web-Service (JAX-RPC, for example Axis),
- XML-RPC (for example Apache XML-RPC),
- Burlap and Hessian (Caucho),
- RMI ,
- EJB (with JNDI lookup),
- JBoss Remoting ,
- REST (REpresentational State Transfer), a Crispy implementation with commons httpclient and
- Http invoker (http call with serializable Java objects), a Crispy implementation with commons httpclient
- CORBA (experimental)

The simplest case is a local Java call from Java object.

=> All calls can execute **synchronous** or **asynchronous** (see the user guide).

Details to the Crispy framework

What must I do, when I want to invoke a remote method/service. In this section you can get a comparsion from the native remote API (e.g. WebService and XML-RPC) and a Crispy call.

Example for remote invocation with the native API calls (without Crispy!)

- WebService (JAX-RPC):

```
Service service = (Service) ServiceFactory.newInstance().createService(null);
Call call = service.createCall();
call.setTargetEndpointAddress("http://localhost:9080/axis/services/Calculator");
call.setOperationName(new QName("add"));
QName paramXmlType = new QName(int.class.getName());
call.addParameter("arg0", paramXmlType, int.class, ParameterMode.IN);
call.addParameter("arg1", paramXmlType, int.class, ParameterMode.IN);

call.setReturnType(new QName(int.class.getName()));
Integer result = (Integer) call.invoke(new Integer[] {new Integer(1), new
Integer(2)});
System.out.println("1 + 2 = " + result);
```

- XML-RPC (Apache XML-RPC):

```
XmlRpcClient client = new XmlRpcClient("http://localhost:9090");
Vector param = new Vector();
param.add(new Integer(1));
param.add(new Integer(2));
Integer result = (Integer)
client.execute("test.crispy.example.service.Calculator.add", param);
System.out.println("1 + 2 = " + result);
```

Example for remote invocation with Crispy

Two steps to success:

1. Define properties to describe the invocation.
2. Do the remote invocation, how a local call.

=> The step two is always the same!

- WebService (JAX-RPC):

```
Properties prop = new Properties();
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:9080/axis/services");
prop.put(Property.EXECUTOR_CLASS, JaxRpcExecutor.class.getName());

ServiceManager manager = new ServiceManager(prop);
Calculator calc =  (Calculator) manager.createService(Calculator.class);
System.out.println("1 + 2 = " + calc.add(1, 2));
```

- XML-RPC:

```
Properties prop = new Properties();
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:9090");
prop.put(Property.EXECUTOR_CLASS, XmlRpcExecutor.class.getName());

ServiceManager manager = new ServiceManager(prop);
Calculator calc =  (Calculator) manager.createService(Calculator.class);
System.out.println("1 + 2 = " + calc.add(1, 2));
```

=> Properties can loaded with a implemetation from the *net.sf.crispy.PropertiesLoader*.

What ist the special on this framework?

- Very easy to use.
- Very simple and minimal to configure.
- You can call a remote method from Java object, like a local call.
- You don't need to know, how the (remote) technology work.
- You can easy change the technology (for example from XML-RPC to RMI).
- The services don't know a remote-interface or a RemoteException (how RMI).
- The parameter can be a complex object (in parts without programming a Serializabler (Marshalling) or Deserializabler (Unmarshalling))
- You can intercept methods before and after invocation (for logging, time stopping, ...).
- You can modify or extends method parameter and the result (transformation, set a authorization (login) token for the request in the background, ...).

Where you can set in Crispy?

If two or more communication partner exist:

- In Service Oriented Architecture (SOA).
- By Enterprise Service Bus (ESB).
- Client for Java Business Integration (JBI).
- Communication by a Rich Client Platform (RCP) with the server (Client/Server Architecture).
- Communication between service consumer (Client) and service provider (Server) (in the WWW or intranet).
- And so on.

Crispy extensions

You can Crispy combinate with other frameworks. At time exist follow extensions (you can read more under the menu point Extension):

- SpringFramework

- HiveMind
- PicoContainer
- OSGi
- AspectJ

Importend Hints

Before you started: Importend Hints

Questions?

You can read FAQs.

Samples?

You can here find samples.

Why Crispy?

This little story describe the problems and the solution by finding the right solution for a service invocation:

Developer

Crispy

I want to communicate with a remote service! What I have to do?	You have to find the right remote technology for your plans.
What is the right technology? I don't know!	I know this problem. A decision for a concrete technology is not easy, but important for a successful project (performance, security, interoperability and so on).
What must I do after the decision?	If you don't know the technology, you must learn, how the remote technology work.
I decided for a technology and I have implemented some services. Now, I find the remote invocations are to slow.	Change the remote technology!
The remote invocations are on many different places in my source code! The changes are very expensive!	You need a transparent remote invocation on a central place in your source code. Than you can easy change the service technology (how Crispy).

=> The Crispy story you can see in a Crispy Comic .

1.2 News/Plans

Crispy news and plans

Crispy's news

This version is a inofficial stable candidate. This mean, that the API is stable. Before Crispy get the official stable state, it will be more tested in a practical environment.

Crispy's plans

Plans for the core Crispy implementation:

- better support for the 'native' transporter (version 1.x)
- JAX-WS (version 1.x)
- Java Message Service (version 1.x)
- Socket based communication (no version)

New extensions:

- A GUI for easy test, configure and execute services (version 1.x).

1.3 Download

Download Crispy

=> You can here download Crispy .

Three kinds of download:

download	file	description
Source	crispy-source-${version}.zip	Sources from Crispy, tests and samples.
Standard	crispy-${version}.zip	Source plus binaries from Crispy, JavaDoc, JUnit.
Bin	crispy-bin-${version}.zip	Standard with external libraries (no source).

1.4 Documentation

Documentation

All what you can read about Crispy ...

- Importend Hints
- Quick Start
- How Crispy Work
- User Guide
- Developer Guide
- Crispy on the server

1.4.1 Importend Hints

Importend Hints

Not all service technology can the same. You can avoid problems with this hints.

- The transfer object (method parameters and return value) must be simple Java types or JavaBeans with below restriction:
  - Default constructor
  - setter/getter methods
  - setter/getter use interfaces by lists and maps (java.util.Collection, java.util.Map) and not the implementations (e.g. java.util.ArrayList, java.util.HashMap)
  - ...

  Example - class:

```
public class Customer {

	private String name = null;
	private List cars = null;

	public Customer() {
	}

	public String getName() {
		return this.name;
	}
	public void setName(String name) {
		this.name = name;
	}

	public List getCars() {
		return this.cars;
	}
	public void setCars(List cars) {
		this.cars = cars;
	}


}
```

- Prefer Datatypes:
  - java.lang.Integer
  - java.lang.Boolean
  - java.lang.String

    - java.lang.Double
    - java.lang.Date
- Don't use overloading from service methods:

```
public interface Calculator {

        // FALSE
        int add(int a, int b);
        long add(long a, long b);

        // make two different method names
        int addInt(int a, int b);
        long addLong(long a, long b);
}
```

- Every service methods must have one parameter at least and a return value:

```
public interface Echo {

        // false
        String ping();

        // right
        String echo(String echoString);
}
```

- How can transport logical/business/validation errors with service methods? The service method can throw a Exception that contains an additional class (container) for this errors:

```
public class ValidationError implements Serializable {

        private static final long serialVersionUID = 5930253049442253128L;

        private int code = 0;
        private String message = "No message available!";

        public ValidationError() { }
        public ValidationError(int pvCode, String pvMessage) {
                setCode(pvCode);
                setMessage(pvMessage);
        }

        public void setCode(int pvCode) { code = pvCode; }
        public int getCode() { return code; }

        public void setMessage(String pvMessage) { message = pvMessage; }
        public String getMessage() { return message; }

        public String toString() {
                return code + " - " + message;
        }
}

// and the Exception as container for the ValidationError

public class MyBusinessException extends RuntimeException {

        private static final long serialVersionUID = 3669006021606741955L;

        private List validationErrors = new ArrayList();

        public MyBusinessException() {}
        public MyBusinessException(String pvMessage) {
                super(pvMessage);
        }
        public List getValidationErrors() { return validationErrors; }
        public void setValidationErrors(List pvValidationErrors) {
validationErrors = pvValidationErrors; }

}

// and the business method signature example

public interface BusinessService {


        public Object businessServiceMethod(Object arg) throws
MyBusinessException;
}
```

- Don't use complex key objects by java.util.Map implementation by transfer objects.

1.4.2 Quick Start

Quick Start

Two Steps to success.

- **First step:** Create properties. The properties describe the behaviour and the kind of the service.
- **Second step:** Create a ServiceManager instance (this class is a factory for the services) and then create a concrete service instance. Invoke all methods on the service instance just like a call from Java objects.

The second step is always the same. Different are the properties.

A minimal example for a XML-RPC-Service with local server instance.

1. Download the XML-RPC libraries or download crispy-bin-${version}.zip .
   - commons-codec (XML-RPC)
   - commons-httpclient (XML-RPC)
   - xmlrpc (XML-RPC)
   - commons-logging (jakarta commons logging)
   - Crispy (Crispy)
2. Set libraries in the class path.
3. Invoke the sample.

```
import java.util.Properties;

import test.crispy.example.service.Echo;

import net.sf.crispy.Property;
import net.sf.crispy.impl.MiniServer;
import net.sf.crispy.impl.ServiceManager;
import net.sf.crispy.impl.XmlRpcExecutor;
import net.sf.crispy.impl.xmlrpc.MiniXmlRpcServer;

public static void main(String[] args) {

        MiniServer miniServer = null;

        try {
                // Server part
                miniServer = new MiniXmlRpcServer();
                miniServer.addService(Echo.class.getName(),
EchoImpl.class.getName());
                miniServer.addService(Calculator.class.getName(),
CalculatorImpl.class.getName());
                miniServer.start();
                // Client part
                Properties prop = new Properties();
                prop.put(Property.EXECUTOR_CLASS, XmlRpcExecutor.class.getName());

                ServiceManager manager = new ServiceManager(prop);
                Echo echo = (Echo) manager.createService(Echo.class);
                System.out.println("Echo: " + echo.echo("Hello Crispy"));

        } catch (Exception e) {
                e.printStackTrace();
        }
        finally {
                miniServer.stop();
        }
}
```

Examples

=> Details you can see on the User Guide or Developer Guide .

Create a service interface and implementation

```
public interface Calculator {

        public int add(int a, int b);
        public int subtract(int a, int b);
}
```

```
public class CalculatorImpl implements Calculator {

        public int add(int a, int b) { return a + b; }
        public int subtract(int a, int b) { return a - b; }

}
```

Create the service and work

```
// configuration, see step one
Properties prop = new Properties();

// the properties, dependent on the kind of call
...

// create ServiceManager (Factory) with properties from step one
ServiceManager serviceManager = new ServiceManager(prop);
```

```
// create service (interface)
Calculator calculator = (Calculator) serviceManager.createService(Calculator.class);

// simple Java object calls
int result = calculator.add(2, 3);
result = calculator.subtract(8, 2);
```

Concrete transport provider

Local Java call

```
// Map the interface Calculator to implementation CalculatorImpl
prop.put(Calculator.class.getName(),CalculatorImpl.class.getName());

ServiceManager lvServiceManager = new ServiceManager(prop);
Echo lvEcho = (Echo) pvServiceManager.createService(Echo.class);
System.out.println("Echo: " + lvEcho.echo("Hello Crispy!");
```

Alternative you can use a pseudo server:

```
MiniLocalObjectServer server = new MiniLocalObjectServer();
// add service interface and service implemantation
server.addService(Echo.class, new EchoImpl());
server.start();

Properties prop = new Properties();
prop.put(Property.STATIC_PROXY_CLASS, StaticLocalObjectProxy.class.getName());

ServiceManager lvServiceManager = new ServiceManager(prop);
Echo lvEcho = (Echo) pvServiceManager.createService(Echo.class);
System.out.println("Echo: " + lvEcho.echo("Hello Crispy!");
```

Http call with Java Serializer

Properties:

```
Properties prop = new Properties();
prop.put(Property.EXECUTOR_CLASS, HttpExecutor.class.getName());
prop.put(Property.REMOTE_URL_AND_PORT,
"http://localhost:8111/crispy/httpserializer");
```

Create your own HttpServlet:

```
public class MyHttpServlet extends net.sf.crispy.impl.http.HttpServlet {

  public void init(ServletConfig config) throws ServletException {
    addService(Echo.class.getName(), new EchoImpl ());
  }
}
```

Or test with the MiniHttpServer:

```
MiniServer miniServer = new MiniHttpServer();
miniServer.addService(Echo.class.getName(), EchoImpl.class.getName());
miniServer.start();
```

XML-RPC call

Properties:

```
// load an Executor, all calls are delegated to this class (proxy pattern)
prop.put(Property.EXECUTOR_CLASS, XmlRpcExecutor.class.getName());
// url and port of server
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:8080");
```

Create your own XmlServlet, overwrite the init method and register the servlet in the web.xml:

```
public class MyXmlRpcHttpServlet extends XmlRpcHttpServlet {

  ...

  public void init(ServletConfig config) throws ServletException {
    addService(MyServiceInterface.class.getName(), new MyServiceImpl());
  }

  ...
}
```

Alternative start the mini server and deploy a service implementation:

```
MiniXmlRpcServer miniServer = new MiniXmlRpcServer();
miniServer.addService(Echo.class.getName(), EchoImpl.class.getName());
miniServer.addService(Calculator.class.getName(), CalculatorImpl.class.getName());
miniServer.start();
```

RMI call

Properties dynamic RMI (a Crispy implementation):

```
// load an Executor, all calls are delegated to this class (proxy pattern)
prop.put(Property.EXECUTOR_CLASS, RmiExecutor.class.getName());
// url and port of server
prop.put(Property.REMOTE_URL_AND_PORT, "rmi://localhost:1099");
```

Properties static RMI, traditional RMI call:

```
// must extend java.rmi.Remote and throws java.rmi.RemoteException
prop.put(Property.STATIC_PROXY_CLASS, StaticRmiProxy.class.getName());
// url and port of server
prop.put(Property.REMOTE_URL_AND_PORT, "rmi://localhost:1099");
// register service with lookup name
prop.put(RemoteCalculator.class.getName(), RemoteCalculator.LOOKUP_NAME);
```

Or start the MiniRmiServer:

```
MiniRmiServer miniServer = new MiniRmiServer();

// by RmiExecutor you can add a service implementation
server.addService (Echo.class.getName(), new EchoImpl());

// OR
// by StaticRmiProxy you can add a service implementation
serever.addService (Echo.class.getName(), new EchoImpl());

miniServer.start();
```

EJB call

Properties:

```
// Map Service to JNDI-Name and Home interface properties and create proxy for EJBs
calls
prop.put(EjbService.class.getName(), EjbService.JNDI_NAME);
// map to the home-interface, with this interface can create the instance
prop.put(EjbService.class.getName() + Property.EJB_HOME_INTERFACE,
EjbServiceHome.class.getName());
// delegate calls to proxy
prop.put(Property.STATIC_PROXY_CLASS, StaticEjbProxy.class.getName());
```

WebService (JAX-RPC) call

Properties:

```
// load an Executor, all calls are delegated to this class (proxy pattern)
prop.put(Property.EXECUTOR_CLASS, JaxRpcExecutor.class.getName());
// url and port from server
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:8080/axis/services");
```

Start the server:

```
MiniServer miniServer = new MiniAxisServer();
miniServer.start();
```

Burlap and Hessian (Caucho) call

Properties:

```
// load Static-Proxy for Burlap implementation
props.put(Property.STATIC_PROXY_CLASS, StaticBurlapProxy.class.getName());
// url and port from server
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:8091/crispy/burlap");



// load Static-Proxy for Hessian implementation
props.put(Property.STATIC_PROXY_CLASS, StaticHessianProxy.class.getName());
// url and port from server
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:8091/crispy/hessian");
```

Create your own Burlap/Hessian-Servlet, overwrite the init method and register the servlet in the web.xml:

```
public class MyBurlapServlet extends BurlapServlet {

  public void init(ServletConfig config) throws ServletException {
    super.addCache(Echo.class.getName(), new BurlapSkeleton(new EchoImpl()));
  }
}

public class MyHessianServlet extends HessianServlet {

  public void init(ServletConfig config) throws ServletException {
    super.addCache(Echo.class.getName(), new HessianSkeleton(new EchoImpl(),
Echo.class));
```

```
  }
}
```

Or start the MiniCauchoServer:

```
MiniServer miniServer = new MiniCauchoServer();
miniServer.addService(Echo.class.getName(), EchoImpl.class.getName());
miniServer.addService(Calculator.class.getName(), CalculatorImpl.class.getName());
miniServer.start();
```

Crispy REST implementation

Properties:

```
// url and port from server
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:8092/crispy/rest");
prop.put(Property.EXECUTOR_CLASS, RestExecutor.class.getName());
```

Create your own RestServlet, overwrite the init method and register the servlet in the web.xml:

```
public class MyRestServlet extends RestServlet {

  public void init(ServletConfig config) throws ServletException {
    addService(Echo.class.getName(), new EchoImpl ());
    addService("service.echo", new EchoImpl ());
  }
}
```

Or start the MiniRestServer:

```
MiniServer miniServer = new MiniRestServer();
miniServer.addService(Echo.class.getName(), EchoImpl.class.getName());
miniServer.addService(Calculator.class.getName(), CalculatorImpl.class.getName());
miniServer.start();
```

Corba (experimental)

Properties:

```
// url and port from server
prop.put(Property.REMOTE_URL_AND_PORT, "127.0.0.1:1057");
prop.put(Property.STATIC_PROXY_CLASS, StaticCorbaProxy.class.getName());
```

MiniCorbaServer:

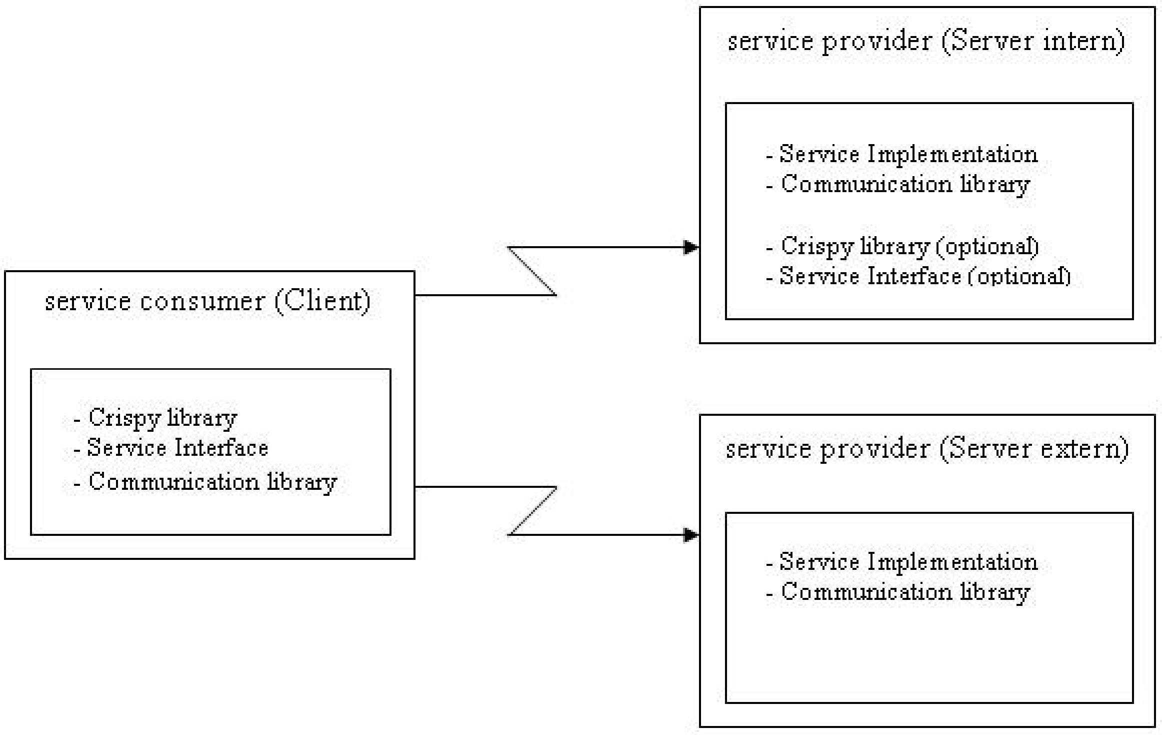
```
MiniServer miniServer = new MiniCorbaServer();
miniServer.addService("test.crispy.example.service.corba.Echo",
"test.crispy.example.service.CorbaEcho");
miniServer.start();
```

1.4.3 How Crispy Work

Relation between service consumer and service provider

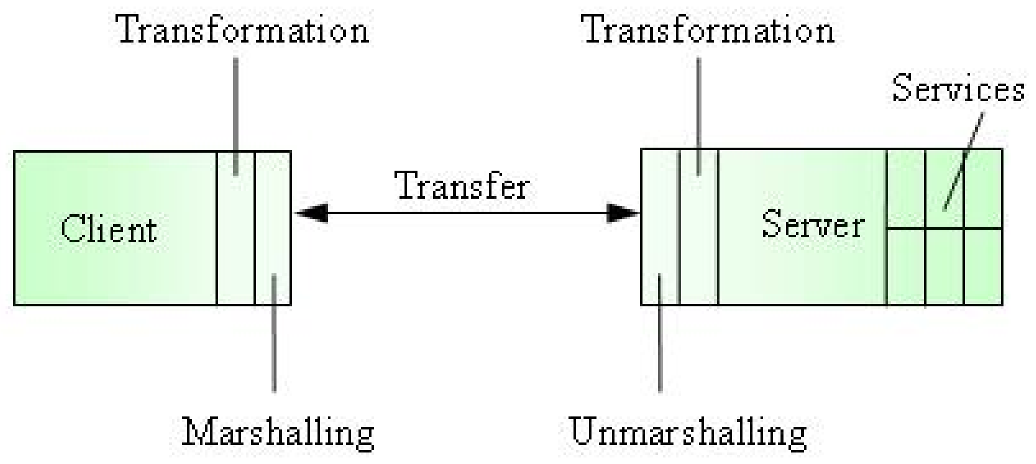
The division in Client and Server parts can be how on the picture:

part	description
Service interface	Descriptor for invocation, method signatuture with parameter und return type.
Service implementation	Implementation from the services on the server.
Crispy library	Library to easy communication with the services.
Communication library	Library for to useful technology.
Server (intern)	Server in the intranet. I can control the libraries.
Server (extern)	The server is in the internet (WWW), he is beyond our control.



How Crispy work

How is Crispy work? On the image you can see a graphical description of the work from Crispy. It is a classic Client - Server communication, where the data (paramter and the result) become marshalling and unmarshalling. For the marshalling, the most technology require, that the transfer object implement the interface *java.io.Serializable*. If it is not wanted, you can transform the transfer object to a simple data structure with the method *net.sf.crispy.util.Converter.makeSimple* on the client side. Before you call the service on the server side you make the original object structure from the client with the method *net.sf.crispy.util.Converter.makeComplex*.



Communication

Client:

1. The *ServiceManager* create a dynamic or static proxy object to make the communication with the server.
2. Call a method of the proxy object.
3. The call can optional transform the paramter (e.g. *net.sf.crispy.util.Converter.makeSimple*).
4. The proxy object serialize (marshall) the call.
5. The framework (technology) send the call to the server.

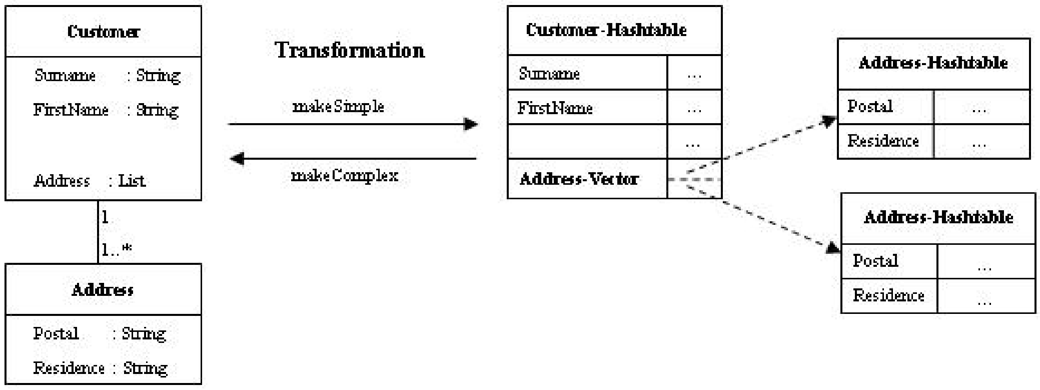
Server:

1. The server get the request.
2. The server deserialize (unmarshall) the client call.
3. When the client has transform the parameter, than transform the server they back (e.g. *net.sf.crispy.util.Converter.makeComplex*).
4. The server execute the method on the service implementation and send back the result to the client.

Transformation

Optional you can transform data on the client side, before they transfer (send to the server). On the server side transform back, before call the service method.

Complex Java Object are transform to the *java.util.Hashtable*. By the *Hashtable* are the keys the attribute names from the Java Object and the *Hashtable* values are the attribute values. Relationship are mapped to the *java.util.Vector*. A example you can see on the image.



1.4.4 User Guide

General

How can I use the different calls of services?

Client - Server

This framework is for the client side. Every client need of the other side a server. You can test the client with a mini server (implementation of *net.sf.crispy.impl.MiniServer* interface). Before you can use a implementation from a client, you have to start a equivalent server implementation.

For example for the RMI-client you start the server *net.sf.crispy.rmi.MiniRmiServer* or your own implementation.

Synchronous/Asynchronous execution

The default execution of the service is synchronous:

```
Properties prop = new Properties();
prop.put(Property.STATIC_PROXY_CLASS, StaticBurlapProxy.class.getName());
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:8093/crispy/burlap");

ServiceManager manager = new ServiceManager(prop);
Echo e = (Echo)  manager.createService(Echo.class);
...
```

The condition for a asynchronous execution is a implementation of the interface *net.sf.crispy.concurrent.AsynchronousCallback*. The implementation of this interface must be registered by the *ServiceManager*. The methods *handleResult* or *handleError* are calling, if the result of method execution is available.

You can switch to asynchronous execution of two kinds. The first way is a global property for all service executions:

```
Properties prop = new Properties();
prop.put(Property.STATIC_PROXY_CLASS, StaticBurlapProxy.class.getName());
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:8093/crispy/burlap");
// set a implementation of the interface:
net.sf.crispy.concurrent.AsynchronousCallback
prop.put(Property.ASYNCHRONOUS_CALLBACK_CLASS,
AsynchronousCallbackForTests.class.getName());
// maximum size of Threads
```

```
prop.put(Property.ASYNCHRONOUS_MAX_SIZE_OF_THREADS, "3");

ServiceManager manager = new ServiceManager(prop);
Echo e = (Echo)  manager.createService(Echo.class);
...
```

The second way is a local property for one service executions:

```

Properties prop = new Properties();
prop.put(Property.STATIC_PROXY_CLASS, StaticBurlapProxy.class.getName());
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:8093/crispy/burlap");

ServiceManager manager = new ServiceManager(prop);
AsynchronousCallback callback = new AsynchronousCallbackForTests();
// the last two args are a method-filter and maximum size of Threads
// the method ping is synchronous, all other Echo-methods are asynchronous
Echo e = (Echo)  manager.createService(Echo.class, callback, new String [] {"ping"}
, 8);
...
```

=> Hint: By Static Proxy must be set Dynamic Proxy for intercept all method calls.

```

Properties prop = new Properties();
prop.put(Property.STATIC_PROXY_CLASS, StaticCorbaProxy.class.getName());
prop.put(Property.DYNAMIC_PROXY_CLASS, Property.VALUE_FOR_JDK_DYNAMIC_PROXY);
```

To switch from asynchronous to synchronous, you can remove the *AsynchronousCallback* by the *ServiceManager*. A switch back to asynchronous execution is not possible.

```

Properties prop = new Properties();
...
ServiceManager manager = new ServiceManager(prop);
...
// after the remove of the AsynchronousCallback are all calls from the Echo class
synchronous.
manager.removeAsynchronousCallback(Echo.class);
```

The call order

For a better invocation control, you can add interceptor or/and modifier implementation (see section: Properties/Configuration). The call order of register interceptor and modifier implementation is:

1. call *net.sf.crispy.Modifier - modifyBeforeInvocation(InterceptorContext)*
2. call *net.sf.crispy.Interceptor - beforeMethodInvocation(InterceptorContext)*
3. execute remote invocation
4. if a Exception is thrown, then call - *net.sf.crispy.Interceptor onError(Throwable)*

5. call *net.sf.crispy.Modifier - modifyAfterInvocation(InterceptorContext)*
6. call *net.sf.crispy.Interceptor - afterMethodInvocation(InterceptorContext)*

The Modifier

Example:

```
public class MyBeforeModifier implements Modifier {

  public InterceptorContext modifyBeforeInvocation( InterceptorContext
interceptorContext) {
     String methodName = interceptorContext.getMethod().getName();
     if (methodName.equals("add")) {
        interceptorContext.setArgs(new Integer[]{new Integer(1), new Integer(2)});
     }
     return interceptorContext;
  }

  public InterceptorContext modifyAfterInvocation(InterceptorContext
interceptorContext) {
     return interceptorContext;
  }
}
```

The Interceptor

Example:

```
public class LogInterceptor implements Interceptor {

  protected static Log log = LogFactory.getLog (LogInterceptor.class);

  public void beforeNewInstance(Class interfaceClass) {
     log.info("beforeNewInstance: " + interfaceClass);
  }

  public void afterNewInstance(Class interfaceClass, Object proxyObject) {
     log.info("afterNewInstance: " + interfaceClass + " - " + proxyObject);
  }

  public void beforeMethodInvocation(InterceptorContext interceptorContext) {
     log.info("beforeMethodInvokation: " + interceptorContext);
  }

  public void afterMethodInvocation(InterceptorContext interceptorContext) {
     log.info("afterMethodInvokation: " + interceptorContext);
  }

  public void onError(Throwable throwable) {
     log.info("onError: " + throwable);
  }
```

Register Modifier/Interceptor

Example:

```
  Properties properties = new Properties();
  properties.put(Property.INTERCEPTOR_CLASS, LogInterceptor.class.getName());
  properties.put(Property.MODIFIER_CLASS, MyBeforeModifier.class.getName());
  IServiceManager serviceManager = new ServiceManager(properties);

  // OR
  IServiceManager serviceManager = new ServiceManager(...);
  serviceManager.addInterceptor(new LogInterceptor ());
  serviceManager.setModifier(new MyBeforeModifier ());
```

Example for interrupt method invocation with a Interceptor

The method invocation is interruptable, with the flag *InterceptorContext.setInterruptIncocation(true)*. This is helpful by a client security or by a client cache, where the remote methode don't execute. With the method *InterceptorContext.setResult(value)* can set the return value, without execute method. The return value can be a value or a Exception.

Example:

```
public class InterruptInterceptor implements Interceptor {

  ...

  public void beforeMethodInvocation(InterceptorContext interceptorContext) {
     Object result = MyCache.get(interceptorContext.getMethod(), 
interceptorContext.getArgs());
     if (result != null) {
             interceptorContext.setInterruptInvocation(true);
             interceptorContext.setResult(result);
     }
  }

  // Or a Exception

  public void beforeMethodInvocation(InterceptorContext interceptorContext) {
     String methodName = interceptorContext.getMethod().getName();
     if (MySecurityManager.checkPermission(methodName, getCurrentUser())) {
             interceptorContext.setInterruptInvocation(true);
             interceptorContext.setResult(new MySecurityException("No Permission to 
execute the method: " + methodName);
     }
  }

}
```

Filter for Interceptor

For a better control of Interceptors you can add a *InterceptorFilter*. With the Filter you can enable or disable the call from Interceptors. It is a possibility to control the Interceptors for a better performance or outline.

Example:

```
  ...

  ServiceManager serviceManager = new ServiceManager(properties);
  serviceManager.addInterceptor(new WaitInterceptor());
  serviceManager.addInterceptor(new StopWatchInterceptor());

  SimpleNameInterceptorFilter filter = new SimpleNameInterceptorFilter(new String[]
{"ping", "add"},
SimpleNameInterceptorFilter.FILTER_TYPE_METHOD);
  serviceManager.addInterceptorFilter(filter);

  Echo echo = (Echo) serviceManager.createService(Echo.class);
  // the Interceptor is filtered, the Interceptor is executed (stop time is greater
0)
  echo.ping();

  // the Interceptor is NOT executed
  echo.echo("Hello Crispy!");
```

Static vs Dynamic Proxy

See details in Developers Guide in section Proxy Pattern .

Static Proxy - Property.STATIC_PROXY_CLASS

The convention for the Static Proxy name class is: Static[transport provider]Proxy.

Transport	Static Proxy
Burlap and Hessian (Caucho)	*StaticBurlapProxy / StaticHessianProxy*
EJB	*StaticEjbProxy*
Static RMI	*StaticRmiProxy*
CORBA	*StaticCorbaProxy*
Local object calls	*StaticLocalObjectProxy*

Dynamic Proxy - Property.EXECUTOR_CLASS

The convention for the Dynamic Proxy name class is: [transport provider]Executer.

Transport	Dynamic Proxy / Executer
XML-RPC	*XmlRpcExecutor*

Transport	Dynamic Proxy / Executer
JAX-RPC	*JaxRpcExecutor*
REST	*RestExecutor*
JBoss Remoting	*JBossRemotingExecutor*
Dynamic RMI	*RmiExecutor*

Server name/port and invocation strategy

All transport provider required a url to invoke the remote invokation. In this url is coded the server name and server port and the service and the service method. The server name and port is describe in the properties:

```
Properties prop = new Properties();
prop.put(Property.REMOTE_URL_AND_PORT, "http://www.myserver.de:8080");
prop.put(Property.INVOCATION_STRATEGY,
ClassPlusMethodInvocationStrategy.class.getName());
```

The service name and method is coded with the *net.sf.crispy.InvocationStrategy*, for Example:

```
public class ClassPlusMethodInvocationStrategy implements InvocationStrategy {

  public Object convert(Map propertyMap) {
     String clazz = (String) propertyMap.get(InvocationStrategy.CLASS_NAME);
     String method = (String) propertyMap.get(InvocationStrategy.METHOD_NAME);
     return clazz + "." + method;
  }

}
```

Transport	URL plus Invocation Strategy	Example
Burlap and Hessian (Caucho)	*UrlAndPort* + "/" + *InterfaceName*	http://www.myserver.de:8080/crispy/hessian/test.crispy.example.servic
EJB	with properties	-
Static RMI	*UrlAndPort* + "/" + *LookupName*	http://rmi://myserver:1099/RemoteEcho
Local object calls	local Java Object calls	-
XML-RPC	*NameSpacePlusMethodInvocationStrategy*	http://myserver:9090/test.crispy.example.service.Calculator.add
JAX-RPC	*UrlPlusClassNameInvocationStrategy*	http://myserver:80/axis/services/Calculator
REST	*RestInvocationStrategy*	http://myserver:8095/crispy/rest?class=test.crispy.example.service.Calc

Transport	URL plus Invocation Strategy	Example
JBoss Remoting	*ClassMethodMapInvocationStrategy*	socket://myserver:5411
Dynamic RMI	-	rmi://myserver:1099/RmiInvokationHandler

Properties/Configuration to describe the communication

General Properties, independent of the choice of the client technology. All properties are optional and can combinate jointly.

All properties declaretions are Strings!

- **DYNAMIC_PROXY_CLASS** - *Property.VALUE_FOR_JDK_DYNAMIC_PROXY* (is default) or *Property.VALUE_FOR_CGLIB_DYNAMIC_PROXY* or your own implementation of *net.sf.crispy.DynamicProxy*
- **INTERCEPTOR_CLASS, INTERCEPTOR_CLASS_2, INTERCEPTOR_CLASS_3** - You can add interceptor (*net.sf.crispy.Interceptor*) classes. For example *net.sf.crispy.interceptor.LogInterceptor* for print logs.
- **MODIFIER_CLASS** - You can add modifier class with implementation from the *net.sf.crispy.Modifier* interface. In this class, you can modify the parameter befor the method is executed or you can convert the result after the execution. With the class *net.sf.crispy.impl.ModifierChain* you can link together several Modifier in a chain. The output from the first Modifier is the input for the next (second) Modifier in the chain and so on.
- **INVOCATION_STRATEGY** and **INVOCATION_STRATEGY_NAMESPACE** - A remote call has the url and port from the server. The url has different construction. A XML-RPC url may be: *http://localhost:8080/sf.net.service.Echo.echo -> [url] + [class] + [method]* or *http://localhost:8080/services.echo -> [url] + [namespace] + [method] (INVOCATION_STRATEGY_NAMESPACE correspond with [namespace]).*
- **WITH_CONVERTER** - Convert parameter before invoke method (*net.sf.crispy.utilConverter.makeSimple()*) and result after invoke methode (*net.sf.crispy.utilConverter.makeComplex()*).
- **SECURITY_PASSWD** and **SECURITY_USER** - Are properties for the future to manage security. For example, to get a identify Token for the next calls.
- **DEBUG_MODE_ON** - By search error or analyse the system in detail. This option can extend with set the logger to debug mode.
- **NULL_VALUE** - If object properties can be `null` and the transport not supported null values (XML-RPC for example) you can substitute the null value with this property.
- **ASYNCHRONOUS_CALLBACK_CLASS** - A class name of a implementation of the interface: *net.sf.crispy.concurrent.AsynchronousCallback*. With set this properties, all executions are asynchronous. (**ASYNCHRONOUS_MAX_SIZE_OF_THREADS**, with this extension you can influence the maximum size of Threads.)

Properties define in the source code

Property example:

```
Properties prop = new Properties();
prop.put(Property.DYNAMIC_PROXY_CLASS, Property.VALUE_FOR_CGLIB_DYNAMIC_PROXY);

prop.put(Property.INTERCEPTOR_CLASS, LogInterceptor.class.getName());
prop.put(Property.INTERCEPTOR_CLASS_2, StopWatchInterceptor.class.getName());

prop.put(Property.SECURITY_USER, "user");
prop.put(Property.SECURITY_PASSWD, "passwd");
```

Load property from file

Properties can create of two kinds. It is possible to do the properties in a file or the second possibillity is to create the properties in the programming sources.

Are the properties in a file, than can load this properties of different kinds:

- *net.sf.crispy.properties.ClassPathPropertiesLoader*: the porperty file can find in the java class path.

```
PropertiesLoader loader = new ClassPathPropertiesLoader("example.properties");
```

- *net.sf.crispy.properties.ClassPropertiesLoader*: the property file is in the same package, how the class.

```
PropertiesLoader loader = new ClassPropertiesLoader(this.getClass(),
"example.properties");
```

- *net.sf.crispy.properties.FilePropertiesLoader*: the file can load how every file with the file and directory path.

```
PropertiesLoader loader = new FilePropertiesLoader("c:/temp/example.properties");
```

- *net.sf.crispy.properties.UrlPropertiesLoader*: file can load with a url to the file.

```
PropertiesLoader loader = new
UrlPropertiesLoader("file://c:/temp/example.properties");
```

If properties are loaded, then can the *ServiceManager* get the properties and work:

```
PropertiesLoader loader = new ClassPathPropertiesLoader("example.properties");
ServiceManager manager = new ServiceManager(loader);
Calculator calculator = (Calculator) serviceManager.createService(Calculator.class);
```

```
int result = calculator.add(2, 3);
```

A Example for a property file:

```
crispy.prop.server.url=http://localhost:9090
crispy.prop.executor.class=net.sf.crispy.impl.XmlRpcExecutor
```

Different Clients

In this section i will describe the special properties for the client invocation.

Http call with Java Serializer

Properties:

- *EXECUTOR_CLASS* - The executor class. *net.sf.crispy.impl.HttpExecutor*
- *REMOTE_URL_AND_PORT* (optional) - The server URL and port. Default value is *http://localhost:8111/crispy/httpserializer*

Example:

```
Properties prop = new Properties();
prop.put(Property.EXECUTOR_CLASS, HttpExecutor.class.getName());
prop.put(Property.REMOTE_URL_AND_PORT,
"http://www.my.de:8111/crispy/httpserializer");
```

Hint: All transfer object must implement the interface *java.io.Serializable*. You can deal with this restriction by set the *Property.WITH_CONVERTER* of *true*. This option is used be non great transfer objects.

XML-RPC

Properties:

- *EXECUTOR_CLASS* - The executor class. *net.sf.crispy.impl.XmlRpcExecutor*
- *REMOTE_URL_AND_PORT* (optional) - The server URL and port. Default value is *http://localhost:8080*

Example:

```
Properties prop = new Properties();
prop.put(Property.EXECUTOR_CLASS, XmlRpcExecutor.class.getName());
prop.put(Property.REMOTE_URL_AND_PORT, "http://www.my.de:8081");
```

Server: If you want to use complex parameter objects (for example a customer with addresses), you have to add the handler *net.sf.crispy.xmlrpc.XmlRpcConvertHandler*. This handler make the marshalling and unmarshalling (how the *MiniXmlRpcServer*).

RMI

Crispy know two kinds of RMI invocation:

- **Static:** Is the traditional method. The service interface must implement the *java.rmi.Remote* interface and the method must thrown *java.rmi.RemoteException*. The implementation extends the *java.rmi.server.UnicastRemoteObject* and is bind on the RMI server (Naming.rebind([LookUp-Name]).
- **Dynamic:** Is a Java Interface without restriction. The call is wrapped in the *net.sf.crispy.impl.rmi.RmiInvokationHandler*.

Static RMI:

Properties:

- *STATIC_PROXY_CLASS* - The proxy creater class. *net.sf.crispy.impl.StaticRmiProxy*
- *Service-Name and LookUp-Name* The service is mapped to lookup-name.
- *REMOTE_URL_AND_PORT* (optional) - The server URL and port. Default value is *rmi://localhost:1099*

Example:

```
Properties prop = new Properties();
prop.put(Property.STATIC_PROXY_CLASS, StaticRmiProxy.class.getName());
prop.put(Property.REMOTE_URL_AND_PORT, "http://www.my.de:1099");
// LookUp - name for Naming.rebind()
prop.put(RemoteCalculator.class.getName(), RemoteCalculator.LOOKUP_NAME);
```

Dynamic RMI:

Properties:

- *EXECUTOR_CLASS* - The executor class. *net.sf.crispy.impl.RmiExecutor*
- *REMOTE_URL_AND_PORT* (optional) - The server URL and port. Default value is *rmi://localhost:1099*

Example:

```
Properties prop = new Properties();
prop.put(Property.EXECUTOR_CLASS, RmiExecutor.class.getName());
prop.put(Property.REMOTE_URL_AND_PORT, "http://www.my.de:1099");
```

Server: If you want to use complex parameter objects, you have to add the handler *net.sf.crispy.rmi.RmiInvokationHandlerImpl*. This handler make the marshalling and unmarshalling (how the *MiniRmiServer*).

EJB

Properties:

- *JNDI_NAME* - The interface is mapped to the JNDI name.
- *EJB_HOME_INTERFACE* - The home interface, create the instance.
- *STATIC_PROXY_CLASS* - The executor class. *net.sf.crispy.impl.StaticEjbProxy*

Example (*EjbService* extends *EJBObject*):

```
Properties prop = new Properties();
prop.put(EjbService.class.getName(), EjbService.JNDI_NAME);
prop.put(EjbService.class.getName() + Property.EJB_HOME_INTERFACE,
EjbServiceHome.class.getName());
prop.put(Property.STATIC_PROXY_CLASS, StaticEjbProxy.class.getName());
```

Server: Crispy don't have a server (container) implementation. You can test your EJB service with MockEJB or with a J2EE container implementation.

JAX-RPC

Properties:

- *TypeMappingFactory.PROPERTY_TYPE_MAPPING_FACTORY, PROPERTY_TYPE_MAPPING_FACTORY_2, PROPERTY_TYPE_MAPPING_FACTORY_3* - With the implementation from this interface, you can register Serializer and Deserializer to transfer complex Java object. The properties for a BeanSerializer is: *TypeMappingFactory.PROPERTY_TYPE_MAPPING_FACTORY, BeanTypeMappingFactory.class.getName() + "," + Customer.class.getName() -> [TypeMappingFactory] + , + [BeanClass]*
- *REMOTE_URL_AND_PORT* (optional) - The server URL and port. Default value is *http://localhost:80*

Example:

```
Properties prop = new Properties();
prop(Property.EXECUTOR_CLASS, JaxRpcExecutor.class.getName());
prop(Property.REMOTE_URL_AND_PORT, "http://localhost:8080/axis/services");
prop(TypeMappingFactory.PROPERTY_TYPE_MAPPING_FACTORY,
BeanTypeMappingFactory.class.getName() + "," + Customer.class.getName());
prop(TypeMappingFactory.PROPERTY_TYPE_MAPPING_FACTORY_2,
BeanTypeMappingFactory.class.getName() + "," + Address.class.getName());
```

Burlap and Hessian (Caucho)

Properties:

- *STATIC_PROXY_CLASS* - The static proxy class: *net.sf.crispy.impl.StaticBurlapProxy*.
- *REMOTE_URL_AND_PORT* (optional) - The server URL and port. Default value is *http://localhost:8091/crispy/burlap* or *http://localhost:8091/crispy/hessian*
- *DYNAMIC_PROXY_CLASS* - (optional) The dynamic proxy class: *Property.VALUE_FOR_JDK_DYNAMIC_PROXY* This is necessary, if you want to use a *net.sf.crispy.Interceptor* or *net.sf.crispy.Modifier*.

Example:

```
// load Static-Proxy for Burlap implementation
props.put(Property.STATIC_PROXY_CLASS, StaticBurlapProxy.class.getName());
// url and port from server
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:8091/crispy/burlap");


// load Static-Proxy for Hessian implementation
props.put(Property.STATIC_PROXY_CLASS, StaticHessianProxy.class.getName());
// url and port from server
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:8091/crispy/hessian");
```

Hint to Mapping the Servlet (for a hessian example) => the url pattern must end with the ***:

```
<servlet-mapping>
        <servlet-name>crispy-servlet</servlet-name>
        <url-pattern>/crispy/hessian/*</url-pattern>
</servlet-mapping>
```

JBoss Remoting

Properties:

- *EXECUTOR_CLASS* - The executor class: *net.sf.crispy.impl.JBossRemotingExecutor*.
- *REMOTE_URL_AND_PORT* (optional) - The server URL and port. Default value is *socket://localhost:5400* (possible protocols are *socket, sslsocket, servlet, rmi, http*).

Example:

```
// load Executor for JBoss remoting implementation
props.put(Property.EXECUTOR_CLASS, JBossRemotingExecutor.class.getName());
// url and port from server
prop.put(Property.REMOTE_URL_AND_PORT, "socket://localhost:5400");
```

CORBA

Properties:

- *STATIC_PROXY_CLASS* - The static proxy class: *net.sf.crispy.impl.StaticCorbaProxy*.
- *REMOTE_URL_AND_PORT* (optional) - The server URL and port. Default value is *127.0.0.1:1057*

Example:

```
Properties prop = new Properties();
prop.put(Property.STATIC_PROXY_CLASS, StaticCorbaProxy.class.getName());
// no protocol!
prop.put(Property.REMOTE_URL_AND_PORT, "myhost.de:8081");
```

1.4.5 Developer Guide

Extend this framework

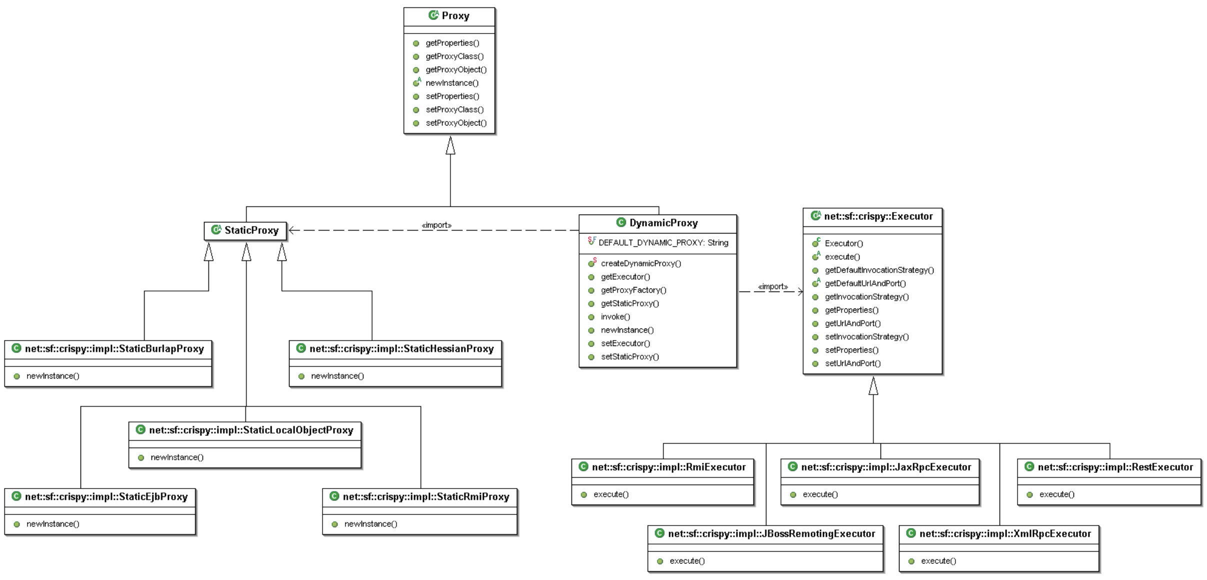
How can I implement additional calls of services?

Basics

The central units from this framework are:

- *ServiceManager* - Is a factory to create the service
- *Properties* - To configure the Service Manager
- *Service Interface* - Plain java class
- *Proxy/Executor*

Class Diagramm



Proxy Pattern

The framework distinguish two kinds of *Proxies*:

- static proxy and
- dynamic proxy

The differents between this two proxies are:

- The *static proxy* is used, when the technology generate stubs and skeletons for the communication

(RMI for example) or when the technology is responsible to generate the proxy object (EJB for example).

- The *dynamic proxy* is used, when the developer has to implement the communication and call the method from the framework (XML-RPC or Web-Service for example).

Static Proxy

If you want create your own implementation with the static proxy, you have to extend the class *net.sf.crispy.StaticProxy*.

The extended static proxy class, is responsible for the method *newInstance*. The *StaticEjbProxy* for example makes in this method a JNDI lookup.

Dynamic Proxy

The framework support two implementation of *Dynamic Proxies*:

- The *Property.VALUE_FOR_JDK_DYNAMIC_PROXY* - It is a implementation from the *java.lang.reflect.Proxy* from jdk. This proxy can only extend interfaces.
- The *Property.VALUE_FOR_CGLIB_DYNAMIC_PROXY* - It is a implementation with CGLIB . This Proxy can extend interfaces and non final classes.

The *DynamicProxy* class needn't to extend. If you want to create an additional implementation, you have to extend the *net.sf.crispy.Executor* class. You have to implement the abstract method *execute* and the method *getDefaultUrlAndPort*. The *execute* method make the call to the remote server.

When the *DynamicProxy* is not enough, than can you make your own implementation. You must implement the methods:

- *newInstance* - Create the proxy for the *Service Interface*.
- *invokeIfExecutorIsNull* (optional) - This method is for the case, that none *Executor* is well know.

Test the own implementation of compatibility

After the implementation from your own extension, you can test the compatibility. This is possible with the *net.sf.crispy.util.compatibility.CompatibilityKit*:

```
CompatibilityKit compatibilityKit = new CompatibilityKit();
compatibilityKit.addProperty(Property.EXECUTOR_CLASS,
XmlRpcExecutor.class.getName());
compatibilityKit.makeAllTests("XmlRpc", new MiniXmlRpcServer());
```

You have to add the *Executor* or the *StaticProxy* class to the properties. If the *CompatibilityKit* has to manage a server instance, then one parameter is an instance of the *net.sf.crispy.impl.MiniServer* interface. This parameter is optional. If is no server set, the server has to be started externally.

for example:

```
(classpath:
commons-codec.jar;commons-logging.jar;commons-httpclient.jar;servlet.jar;
xmlrpc.jar;crispy.jar)
java net.sf.crispy.impl.xmlrpc.MiniXmlRpcServer
```

If the test is not successful, then a *CompatibilityException* is thrown.

1.4.6 Crispy on the server

Crispy goes to the server

By the start from this project, Crispy was a project for the client side. After one year development exist a new standpoint. Client side without the server side is not the best idea. In the future will Crispy go to the server side.

The future goals for release greater than 1.1.0 are:

- Better Exception handling by server side (business or technical Exceptions)
- Support for server side InterceptorHandler. Can intercept before and after method execution. (tracing, monitoring, security and so on)
- Support for statefule services. The service know the caller.
- ...

Example for server side Interceptor

This example is showing on a XML-RPC transport provider, how can add interceptors on a XML-RPC servlet.

The idea is to create a instance of the *InterceptorHandlerCreator*, where all interceptors are created and to *InterceptorHandler* added. Before a service method is excecuted, is a new *InterceptorHandler* created and all added interceptors are calling.

```
public class XmlRpcExampleServiceEndpoint extends XmlRpcHttpServlet {

        private static final long serialVersionUID = 167096795226621025L;

        public SubXmlRpcServiceEndpoint() {
                super();
                // add services
                addService(Echo.class.getName(), new EchoImpl());
                // set method for creating your own InterceptorHandlerCreator
                setInterceptorHandlerCreator(createInterceptorHandlerCreator());
        }

        private InterceptorHandlerCreator createInterceptorHandlerCreator() {

                return new InterceptorHandlerCreator() {

                        public InterceptorHandler
createNewInterceptorHandlerInstance() {
                                InterceptorHandler lvHandler =  new
InterceptorHandler();
                                lvHandler.addInterceptor(new LogInterceptor());
                                lvHandler.addInterceptor(new
StopWatchInterceptor());
                                return lvHandler;
                        }

                };

        }

}
```

=> Is this a good idea and an easy way to add a interceptor/modifier?

Get your feedback to the forum on sourceforge

Under construction ...

1.5 Crispy REST Guide

Principles

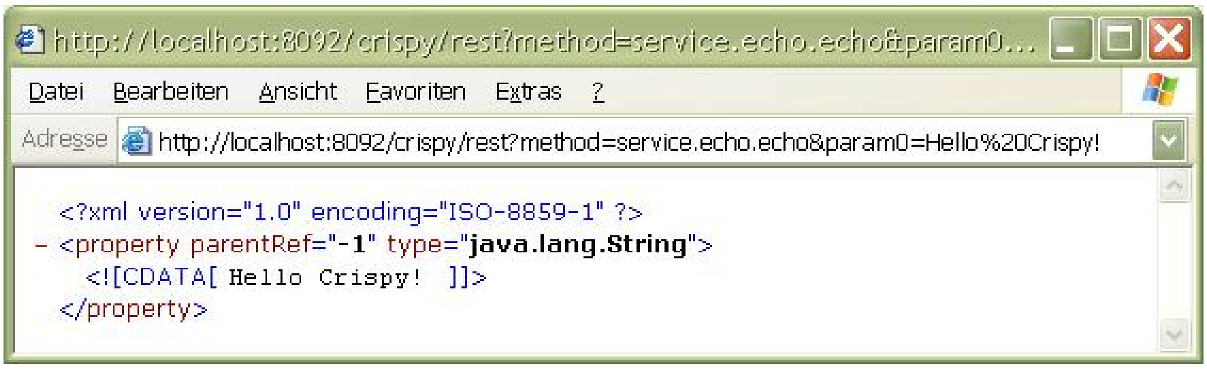
Principles are from wikipedia

REST's proponents argue that the web has enjoyed the scalability and growth that it has as a result of a few key design principles:

- A stateless client/server protocol: each HTTP message contains all the information necessary to understand the request. As a result, neither the client nor the server needs to remember any communication-state between messages. In practice, however, many HTTP-based applications use cookies and other devices to maintain session state (some of those practices, like URL-rewriting, are not RESTful).
- A set of well-defined operations that apply to all pieces of information (called resources): HTTP itself defines a small set of operations, the most important of which are GET, POST, PUT, and DELETE.
- A universal syntax for resource-identification: in a RESTful system, every resource is uniquely addressable through the resource's URL (attempts to define a higher level of abstraction, the URI, have met little success, though the name "URI" will often appear in specifications and other literature).
- The use of hypermedia both for application information and application state-transitions: representations in a REST system are typically HTML or XML files that contain both information and links to other resources; as a result, it is often possible to navigate from one REST resource to many others, simply by following links, without requiring the use of registries or other additional infrastructure.

Crispy REST guide

The special on the REST (REpresentational State Transfer) implementation is, that the invocation can you do about the URL. A example can you see on the picture.


http://localhost:8092/crispy/rest?method=service.echo.echo&param0...
Datei
Bearbeiten
Ansicht
Favoriten
Extras
?
Adresse

http://localhost:8092/crispy/rest?method=service.echo.echo&param0=Hello%20Crispy!

```
  <?xml version="1.0" encoding="ISO-8859-1" ?>
- <property parentRef="-1" type="java.lang.String">
    <![CDATA[ Hello Crispy!  ]]>
  </property>
```

In this simple example you can see a call from the echo-method with the parameter 'Hello Crispy!'. This a very easy way to test the server.

Parts from the URL are:

```
http://localhost:8092/crispy/rest?method=service.echo.echo&param0=Hello Crispy!
```

URL	description
http://localhost:8092/	The URL and port from the server.
crispy	The web context.
rest	The servlet mapping string.
method	Mapping string for the server to find a service object and method.
param0	Parameter for the service method.

Second possibile way to make a call with the URL:

```
http://localhost:8092/crispy/rest?class=test.crispy.example.service.Echo
                        &method=echo&param0=Hello Crispy!
```

URL	description
http://localhost:8092/	The URL and port from the server.
crispy	The web context.
rest	The servlet mapping string.
class	The name of register service class.
method	The service method.
param0	Parameter for the service method.

REST invocation with Crispy

Start the mini server:

```
MiniServer server = new MiniRestServer();
try {
        server.addService("test.crispy.example.service.Echo",
"test.crispy.example.service.EchoImpl");
        server.addService("service.echo", "test.crispy.example.service.EchoImpl");
        server.addService("test.crispy.example.service.Calculator",
"test.crispy.example.service.CalculatorImpl");
```

```
        server.addService("service.calculator",
"test.crispy.example.service.CalculatorImpl");

        server.start();

        // all calls

} catch (Exception e) {
        e.printStackTrace();
}
finally {
        server.stop();
}
```

Problems with Crispy REST

Not supported REST Datatypes:

- Array's -> change Array's with java.util.Vector

1.5.1 Install

Install Crispy REST

Install description:

- Copy the Crispy, commons-httpclient and commons-logging libraries in the WEB-INF/lib directory.
- To install the Crispy REST implementation you must register the RestServlet in the web.xml file from the web application.

1. Register the REST-Servlet:

```
<servlet>
        <servlet-name>rest-crispy-servlet</servlet-name>
        <display-name>REST Crispy Servlet</display-name>
        <description>A servlet to communicate with REST technology</description>
        <servlet-class>net.sf.crispy.impl.rest.RestServlet</servlet-class>
</servlet>
```

2. Mapping the REST-Servlet:

```
<servlet-mapping>
        <servlet-name>rest-crispy-servlet</servlet-name>
        <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

1.5.2 Publish a service

Publishing a new REST service

1. Extend the *net.sf.crispy.impl.rest.RestServlet* Servlet.
2. Overwrite the *init()* method.

Example:

```
public class MyRestServlet extends RestServlet {

        public void init() throws ServletException {
                // addService(key, value);
                // key: Mapping string for the server to find a service object and
method.
                // value: implementation from the service
                addService(Echo.class.getName(), new EchoImpl ());

                // The string is to lower case!
                // key = 'service.' + class-name without package
                addService("service.echo", new EchoImpl ());
        }
}
```

1.6 Samples

Samples

This section show concrete samples to try and test the Crispy frameworke.

- Jira Sample
- Thesaurus lexicon (University of Leipzig, Germany) Sample
- ...

This samples show what do you can do with Crispy and how Crispy work.

If you download Crispy, you can find more samples in the *sample* folder.

You know a application, where can see the power of Crispy? Let me know.

1.6.1 Issue Tracking Jira

Jira Sample

This sample demonstrate the access to services from the issue tracking system JIRA . It is the first test (experiment) with the JIRA version 3.2.x.. On the Jira web side you can see the description of a example .

Advantages from the Crispy-JiraService

- The same call from Web-Service and XML-RPC.
- The return value from XML-RPC are concrete (Jira) objects (otherwise are it Vector or Hashtable).
- All calls are simple Jave Object calls.
- The Login-Token must never used. The token is only use in background and not to see for the user.

Quick Start

A simple and rapid way to use a JIRA client is to use the class *net.sf.crispy.sample.jira.Run*

1. Edit the property: jira.properties (the file is, where the class *net.sf.crispy.sample.jira.Run* is)
2. Execute the main-method in the class *net.sf.crispy.sample.jira.Run*.

The conrete steps are:

1. The properties are:

```
# the user to login
crispy.sample.user=soaptester
# the password to login
crispy.sample.passwd=soaptester
# a issue to search
crispy.sample.issue=TST-2184
# when the service is a Web-Service
crispy.sample.url=http://atlassian01.contegix.com:10083/rpc/soap
# OR
# when the service is a XML-RPC-Service
# crispy.sample.url=http://atlassian01.contegix.com:10083/rpc/xmlrpc
```

2. Run the main-method:

```
java -classpath [libs] net.sf.crispy.sample.jira.Run
```

Libs for the JIRA-client:

- *atlassian-jira-classes* - are in the /WEB-INF/classes form the JIRA install dir.
- *atlassian-jira-rpc-plugin-1.5.4.jar* - documentation and JIRA RPC client .
- *commons-lang-2.0.jar* - download
- *log4j-1.2.8.jar* - download
- *ofbcore-entity-2.1.1.jar* - download
- *ofbcore-share-2.1.1.jar* - download
- *osuser-1.0-dev.jar* - download

Libs for XML-RPC-Client :

- *commons-codec-1.2.jar*
- *commons-httpclient-2.0.2.jar*
- *servlet.jar*
- *xmlrpc-1.2-b1.jar*
- *commons-logging.jar*

Libs for Web-Service-Client Axis :

- *axis.jar-1.2.1.jar*
- *commons-discovery-0.2.jar*
- *jaxrpc.jar*
- *saaj.jar*
- *wsdl4j-1.5.1.jar*
- *commons-logging.jar*

Your own application

1. Create Properties:

```
Properties props = new Properties();
props.put(Property.REMOTE_URL_AND_PORT,
http://atlassian01.contegix.com:10083/rpc/soap);
```

2. Create ServiceManager and Service instance:

```
IServiceManager serviceManager = new JiraServiceManager(props,
JiraServiceManager.SOAP_SERVICE);
JiraService jiraService = (JiraService)
serviceManager.createService(JiraService.class);
```

3. Login on the jira-server:

```
jiraService.login("user", "passwd");
```

4. Invoke service-methods:

```
RemoteUser ru = jiraService.getUser("user");
System.out.println("User: " + ru.getName() + " "
                                + ru.getFullname()
                                + " " + ru.getEmail());

RemoteProject remoteProjects[] = jiraService.getProjects();
for (int i = 0; i < remoteProjects.length; i++) {
        System.out.println(remoteProjects[i].getId() + " "
                                + remoteProjects[i].getName() + " "
                                + remoteProjects[i].getDescription() + " "
                                + remoteProjects[i].getKey() + " "
                                + remoteProjects[i].getLead() + " "
                                + remoteProjects[i].getProjectUrl());
}
```

5. After all calls you can logout:

```
jiraService.logout();
```

1.6.2 Thesaurus lexicon

Thesaurus lexicon (University of Leipzig, Germany) Sample

Thesaurus lexicon (University of Leipzig, Germany) Homepage .

```
// create a special ServiceManager (with URL and Serializer/Deserializer
configuration
IServiceManager manager = new WortschatzServiceManager();

// simple ping to test the connection
Thesaurus thesaurus = (Thesaurus) manager.createService(Thesaurus.class);
System.out.println("PING: " + thesaurus.ping());

// thesaurus request for the german word leer, with max 5 response word
List lvResultList = thesaurus.execute("leer", 5).getResultList();
System.out.println("- Thesaurus -");
System.out.println(lvResultList);

Similarity similarity = (Similarity) manager.createService(Similarity.class);
System.out.println("- Similarity -");
System.out.println(similarity.execute("leer", 2).getResultList());

Wordforms wordform= (Wordforms) manager.createService(Wordforms.class);
System.out.println("- Wordforms -");
System.out.println(wordform.execute("leer" ,2).getResultList());

Sentences sentences = (Sentences) manager.createService(Sentences.class);
System.out.println("- Sentences -");
System.out.println(sentences.execute("leer", 2).getResultList());

Frequencies frequencies = (Frequencies) manager.createService(Frequencies.class);
System.out.println("- Frequencies -");
System.out.println(frequencies.execute("leer").getResultList());

Synonyms synonyms = (Synonyms) manager.createService(Synonyms.class);
System.out.println("- Synonyms -");
System.out.println(synonyms.execute("leer", 2).getResultList());
```

1.7 Extension

Extensions

This section show extensions for the Crispy frameworke.

- SpringFramework
- HiveMind
- PicoContainer
- OSGi
- AspectJ
- ...

You know a extensions for Crispy? Let me know.

1.7.1 SpringFramework

SpringFramework

SpringFramework Homepage .

A short example, how you can invoke with the SpringFramework a XML-RPC service. First you create a instance from the *ApplicationContext* to load the configuration file. The context create in a second step a proxy object for the service interface. The proxy object delegate all calls from the service interface method to the *net.sf.crispy.impl.XmlRpcExecutor*. The *XmlRpcExecutor* make the remote calls to the MiniServer and send back the result.

```
MiniServer server = new MiniXmlRpcServer(8080);
try {
        server.addService(Echo.class.getName(), EchoImpl.class.getName());
        server.addService(Calculator.class.getName(),
CalculatorImpl.class.getName());

        server.start();

        ApplicationContext ctx = new
ClassPathXmlApplicationContext("/spring_example_1.xml");
        Calculator calculator = (Calculator) ctx.getBean("calculator");
        System.out.println("add: 5+7 = " + calculator.add(5, 7));
        System.out.println("subtract: 9-3 = " + calculator.subtract(9, 3));

} catch (Exception e) {
        e.printStackTrace();
}
finally {
        server.stop();
}
```

In the example you load a configuration file (spring_example_1.xml). This file is description here:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC
   "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">


<beans>

        <bean id="calculator"
class="net.sf.crispy.extension.spring.CrispyFactoryBean">
                <property name="props">
                  <props>
```

```
                    <prop key="crispy.prop.static.proxy.class">
                      net.sf.crispy.impl.XmlRpcExecutor
                    </prop>
                  </props>
                </property>
                <property name="serviceInterface">
                  <value>test.crispy.example.service.Calculator</value>
                </property>
                <property name="serviceUrl">
                  <value>http://localhost:8080/</value>
                </property>
        </bean>


</beans>
```

1.7.2 HiveMind

HiveMind

HiveMind Homepage .

A short example, how you can invoke with the HiveMind framework a XML-RPC service. First you create a instance from the *org.apache.hivemind.Registry* to load the configuration file *hivemind.xml* in the META-INF folder. In the second step create a proxy object for the service interface. The proxy object delegate all calls from the service interface method to the *net.sf.crispy.impl.XmlRpcExecutor*. The *XmlRpcExecutor* make the remote calls to the MiniServer and send back the result.

```
MiniXmlRpcServer server = new MiniXmlRpcServer();
try {
        server.addService(Echo.class.getName(), EchoImpl.class.getName());
        server.addService(Calculator.class.getName(),
CalculatorImpl.class.getName());

        server.start();

        Registry registry =  RegistryBuilder.constructDefaultRegistry();
        Calculator calculator = (Calculator) registry.getService(Calculator.class);
        System.out.println("2 + 3 = " + calculator.add(2, 3));

        Echo echo = (Echo) registry.getService(Echo.class);
        System.out.println("Echo: " + echo.echo("Hello Crispy!"));

} catch (Exception e) {
        e.printStackTrace();
}
finally {
        server.stop();
}
```

In this example you load a configuration file (hivemind.xml). This file is description here:

```
<?xml version="1.0"?>
<module id="net.sf.crispy.extension.hivemind" version="1.0.0"
package="hivemind.examples">

<configuration-point id="ConfigCrispyFactoryRMI" schema-id="CrispySchema"/>
<configuration-point id="ConfigCrispyFactoryXMLRPC" schema-id="CrispySchema"/>

  <schema id="CrispySchema">
    <element name="property" key-attribute="key">
      <attribute name="key"/>
      <attribute name="value" required="true"/>

      <conversion class="net.sf.crispy.extension.hivemind.KeyValue"/>
    </element>
  </schema>


<contribution configuration-id="ConfigCrispyFactoryXMLRPC">
  <property key="crispy.prop.server.url" value="http://localhost:9090"/>
  <property key="crispy.prop.executor.class"
value="net.sf.crispy.impl.XmlRpcExecutor"/>
</contribution>

<contribution configuration-id="ConfigCrispyFactoryRMI">
  <property key="crispy.prop.server.url" value="rmi://localhost:1099"/>
  <property key="crispy.prop.executor.class"
value="net.sf.crispy.impl.RmiExecutor"/>
  <property key="test.crispy.example.service.Echo"
value="test.crispy.example.service.EchoImpl"/>
  <property key="test.crispy.example.service.Calculator"
value="test.crispy.example.service.CalculatorImpl"/>
</contribution>



<service-point id="calc" interface="test.crispy.example.service.Calculator">
        <invoke-factory  service-id="CrispyFactory">
                <construct class="net.sf.crispy.extension.hivemind.CrispyFactory" />
        </invoke-factory>
</service-point>

<service-point id="echo" interface="test.crispy.example.service.Echo">
        <invoke-factory  service-id="CrispyFactory">
                <construct class="net.sf.crispy.extension.hivemind.CrispyFactory" />
        </invoke-factory>
</service-point>

<service-point id="CrispyFactory"
interface="org.apache.hivemind.ServiceImplementationFactory">
          <invoke-factory>
            <construct class="net.sf.crispy.extension.hivemind.CrispyFactory"
initialize-method="init">
                    <!-- !!!!!!!!!!! set the configiguration for the Crispy Factory
!!!!!!!!!!! -->
                <set-configuration property="properties"
configuration-id="ConfigCrispyFactoryXMLRPC"/>
                <!-- set-configuration property="properties"
configuration-id="ConfigCrispyFactoryRMI"/ -->
            </construct>
          </invoke-factory>
</service-point>

</module>
```

1.7.3 PicoContainer

PicoContainer

PicoContainer Homepage .

A short example, how you can invoke a remote service with the PicoContainer with a XML-RPC service provider. First you register a *CrispyComponentAdapterFactory* and a *ClassPropertiesLoader* by the *DefaultPicoContainer* . Than you generate a proxy service object. With this this proxy, you can work how a local java object.

```
MiniXmlRpcServer server = new MiniXmlRpcServer();
try {
        // add the service to the server
        server.addService(Echo.class.getName(), EchoImpl.class.getName());
        // start the server
        server.start();

        // first example
        // register the CrispyComponentAdapterFactory and a PropertiesLoader
        DefaultPicoContainer pico = new DefaultPicoContainer(new
CrispyComponentAdapterFactory());
        pico.registerComponentImplementation(new
ClassPropertiesLoader(PropertiesLoaderTest.class,
                                                "test-xml-rpc2.properties"),
PropertiesLoader.class);
        // register the service interface
        pico.registerComponentImplementation(Echo.class);

        // get a proxy object from the service interface
        Echo echo =  (Echo) pico.getComponentInstance(Echo.class);
        System.out.println("Echo: " + echo.echo("Hello Girl!"));



        // second example
        DefaultPicoContainer pico2 = new DefaultPicoContainer();
        pico2.registerComponentImplementation(ClassPropertiesLoader.class,
                                                ClassPropertiesLoader.class,
                                                new Parameter[] {
                                                new
ConstantParameter(PropertiesLoaderTest.class),
                                                new
ConstantParameter("test-xml-rpc2.properties") } );
        pico2.registerComponentImplementation(ServiceManager.class);
        Echo echo2 =  (Echo) pico2.getComponentInstance(Echo.class);
        System.out.println("Echo: " + echo2.echo("Hello Boy!"));

} catch (Exception e) {
        e.printStackTrace();
}
```

```
finally {
        server.stop();
}
```

1.7.4 OSGi

OSGi

OSGi Homepage .

This section describe a proposal, how can you use Crispy in a OSGi envionment (how knopflerfish , oscar or equinox).

Crispy offer two classes. The first class ist the service provider *net.sf.crispy.extension.osgi.CrispyActivator*. The second class is a example for a client, the service consumer *net.sf.crispy.extension.osgi.CrispyClientActivator*.

The service provider - CrispyActivator

By the osgi specification describe the *META-INF/Manifest.mf* the service properties. For example:

```
Manifest-Version: 1.0
Bundle-Description: Crispy - OSGi extension
Bundle-Name: CrispyActivator
Bundle-Classpath: ., commons-logging.jar, crispy-with-tests-0.6.2.jar,
commons-proxy-0.1-SNAPSHOT.jar
Bundle-Activator: net.sf.crispy.extension.osgi.CrispyActivator
Import-Package: org.osgi.framework, net.sf.crispy, net.sf.crispy.impl,
test.crispy.example.service
Bundle-Vendor: Crispy
Bundle-SymbolicName: CrispyActivator
Export-Package: test.crispy.example.service, net.sf.crispy.impl, net.sf.crispy
Bundle-Version: 1.0.0
```

This implementation from the *org.osgi.framework.BundleActivator* interface, the *CrispyActivator*, will internal create a instance of the *ServiceManager*. The *ServiceManager* can be describe with a property file. The location from this file can be in the *META-INF* directory or in the same package how the *CrispyActivator* class, with the name *service.properties*. For example:

```
crispy.prop.dynamic.proxy.class=jdk_proxy
crispy.prop.server.url=rmi://localhost:1099
crispy.prop.executor.class=net.sf.crispy.impl.RmiExecutor
test.crispy.example.service.Echo=test.crispy.example.service.EchoImpl
test.crispy.example.service.Calculator=test.crispy.example.service.CalculatorImpl
crispy.osgi.services=test.crispy.example.service.Echo,test.crispy.example.service.Calculator
```

The source from the *CrispyActivator*:

```
public class CrispyActivator implements BundleActivator {

 private static BundleContext bundleContext = null;

 public void start(BundleContext pvContext) throws Exception {
          bundleContext = pvContext;
        URL lvUrl =
pvContext.getBundle().getResource("META-INF/service.properties");
        PropertiesLoader lvPropertiesLoader = null;
        // first try to load the properties
        if (lvUrl != null) {
          lvPropertiesLoader = new UrlPropertiesLoader(lvUrl);
        }
        // second try to load the properties
        if (lvPropertiesLoader == null) {
          lvPropertiesLoader = new ClassPropertiesLoader(CrispyActivator.class,
"service.properties");
        }

        Properties lvProperties = lvPropertiesLoader.load();
        ServiceManager lvServiceManager = new ServiceManager(lvProperties);
        createServices(lvProperties, lvServiceManager);

        bundleContext.registerService(IServiceManager.class.getName(),
lvServiceManager, new Properties());
 }

 public void stop(BundleContext pvContext) throws Exception {
        bundleContext = null;
 }

 private void createServices(Properties pvProperties, IServiceManager
pvServiceManager) throws Exception {
        String lvServiceClassStrings = (String)
pvProperties.get("crispy.osgi.services");
        if (lvServiceClassStrings != null) {
          String lvClassStr[] = lvServiceClassStrings.split(",");
          for (int i = 0; i < lvClassStr.length; i++) {
            Class lvServiceClass = Class.forName(lvClassStr[i]);
            Object lvServiceProxy = pvServiceManager.createService(lvServiceClass);
            bundleContext.registerService(lvClassStr[i], lvServiceProxy, new
Properties());
          }
        }
 }
```

The service consumer - CrispyClientActivator

By Client is the description in the *META-INF/Manifest.mf* too. For example:

```
Manifest-Version: 1.0
Bundle-Name: CrispyClient
Bundle-ClassPath: ., crispy-with-tests-0.6.2.jar, commons-logging.jar,
commons-proxy-0.1-SNAPSHOT.jar
Bundle-Activator: net.sf.crispy.extension.osgi.CrispyClientActivator
```

```
Import-Package: org.osgi.framework, test.crispy.example.service
Bundle-SymbolicName: CrispyClient
Bundle-Version: 1.0.0
```

A possible implementation from the service consumer:

```
public class CrispyClientActivator implements BundleActivator {

 public void start(BundleContext pvContext) throws Exception {
   // first a echo example
   String filter = "(objectclass=" + Echo.class.getName() + ")";
   ServiceReference[] srl =  pvContext.getServiceReferences(null, filter);
   Echo lvEcho = (Echo) pvContext.getService(srl[0]);
   System.out.println("Echo: " + lvEcho.echo("Hello Crispy ..."));

   // second a calculator example
   filter = "(objectclass=" + Calculator.class.getName() + ")";
   srl =  pvContext.getServiceReferences(null, filter);
   Calculator lvCalculator = (Calculator) pvContext.getService(srl[0]);
   System.out.println("Add 5.3 + 6.3 = " + lvCalculator.add(5.3, 6.4));
 }

 public void stop(BundleContext pvContext) throws Exception { }

}
```

Importend:

The *Bundle-ClassPath* must contain all libraries. In this example can you see the required libraries. The *Import-Package* part describe the packages from the required services.

1.7.5 AspectJ

AspectJ

AspectJ Homepage .

This section describe a proposal, how can you use Crispy in a AspectJ envionment

It are two problems to solve:

1. Where can find *java.util.Properties* to configure the Crispy *net.sf.crispy.impl.ServiceManager*
2. Who interfaces are the Services.

For this two problems exist differents solutions. This is the reason, to divide this problem in a abstract aspect and in one possible extension from the abstract aspect. Both are proposal.

First, you can see the abstract aspect *AbstractServiceBuilder*:

```
public abstract aspect AbstractServiceBuilder {

        protected IServiceManager serviceManager = null;

        /**
         * Create a configuration for a ServiceManager-Instance (first problem).
         */
        public abstract Properties createProperties();

        /**
         * Find all service interfaces for the ServiceManager to
         * create the Service-Proxy-Instance (second problem).
         */
        public abstract pointcut findServices();

        /**
         * The Advice.
         */
        Object around() : findServices() {
                 serviceManager = new ServiceManager(createProperties());
                 FieldSignature fieldSignature = (FieldSignature)
thisJoinPoint.getSignature();
                 Class serviceClass = fieldSignature.getFieldType();
                 Object serviceProxy = serviceManager.createService(serviceClass);
                 return serviceProxy;
        }
}
```

Second, you can see the concrete implementation from the *AbstractServiceBuilder*, the *ServiceBuilder*:

```
public aspect ServiceBuilder extends AbstractServiceBuilder {
        /**
         * Capture all read access to fields in the
         * net.sf.crispy.extension.aspectj.AspectJExample class, where
         * fields from the package test.crispy.example.service.
         */
        public pointcut findServices() :
                get (test.crispy.example.service.*
net.sf.crispy.extension.aspectj.AspectJExample.*);

        /**
         * Load Properties from a file, where the class is.
         */
        public Properties createProperties() {
                String propFileName = "aspect-test.properties";
                Class propClass = this.getClass();
                PropertiesLoader propertiesLoader = new
ClassPropertiesLoader(propClass, propFileName);
                Properties properties = propertiesLoader.load();
                return properties;
        }
}
```

A example class, where the aspects be effective (this example starts your own *MiniServer*):

```
public class AspectJExample {

        private Echo echo = null;
        public Calculator calculator = null;

        public String echo (String echoString) {
                return echo.echo(echoString);
        }


        public static void main(String[] args) {

                MiniServer miniServer = new MiniRmiServer(1099);
                miniServer.addService("test.crispy.example.service.Echo",
"test.crispy.example.service.EchoImpl");
                miniServer.addService("test.crispy.example.service.Calculator",
"test.crispy.example.service.CalculatorImpl");
                miniServer.start();

                try {
                        AspectJExample aspectJExample = new AspectJExample();
                        System.out.println("Echo: " + aspectJExample.echo("Hello
Crispy-AspectJ-Echo ..."));
                        System.out.println("Calculator-add (2+3): " +
aspectJExample.calculator.add(2, 3));
                } catch (Exception e) {
                        e.printStackTrace();
                }
                finally {
                        miniServer.stop();
                }
        }
}
```

The properties:

```
crispy.prop.server.url=rmi://localhost:1099
crispy.prop.executor.class=net.sf.crispy.impl.RmiExecutor
```

1.8 Articles

Articles

Every can write about Crispy. Let it me know and i publishing your article.

- Transform complex object graph in a simple structure for the XML-RPC interface (German)
- Crispy article (German)
- ...

1.9 FAQ's

Frequently Asked Questions

General

1. Is Crispy for the client, for the server or both?
2. What is a service?
3. What is a service technology?
4. How can I set a Proxy?

Question to Crispy Framework

1. Which properties are set with the *java.util.Properties* and which about the *ServiceManager*?
2. How can I activate the logger?

General

General

Is Crispy for the client, for the server or both?

Crispy is precedenced for the client.

The server must now Crispy when:

- Crispy extend the service technology (for example, by XML-RPC or RMI (dynamic)).
- By the REST (REpresentational State Transfer) implementation.

What is a service?

A service has two parts: a interface and a implementation from the interface.

Example:

```
public interface Calculator {
        public int add(int a, int b);
        public int subtract(int a, int b);
}
```

```
public class CalculatorImpl implements Calculator {
        public int add(int a, int b) { return a + b; }
        public int subtract(int a, int b) { return a - b; }
}
```

What is a service technology?

A service technology is the kind to communicate from the client to the server

Examples:

- XML-RPC,
- RMI,
- WebService,
- REST
- or other

How can I set a Proxy?

Proxy host and port:.

```
// host
System.setProperty("http.proxyHost", proxy);
// port
System.setProperty("http.proxyPort", port);
```

Question to Crispy Framework

Question to Crispy Framework

Which properties are set with the *java.util.Properties* and which about the *ServiceManager*?

All properties to the Service technology must set with properties and the other properties can you set. For example: *Property.EXECUTER_CLASS* or *Property.REMOTE_URL_AND_PORT*.

Properties to extend Crispy, can set about the *ServiceManager*. For example: *Property.INTERCEPTOR_CLASS* or *Property.MODIFIER_CLASS*.

```
// with properties
Properties prop = new Properties();
// this can you set in properties and with the ServiceManager
prop.put(Property.INTERCEPTOR_CLASS, StopWatchInterceptor.class.getName());
prop.put(Property.REMOTE_URL_AND_PORT, "http://localhost:9090");
prop.put(Property.EXECUTOR_CLASS, XmlRpcExecutor.class.getName());

// with ServiceManager
ServiceManager serviceManager = new ServiceManager(prop);
serviceManager.setModifier(new MyModifier());
serviceManager.addInterceptor(new MyInterceptor ());
```

How can I activate the logger?

By problems you can activate the logger, to get more details about the working of Crispy. It are two possibilities. You can turn on the jdk logger API with:

```
        net.sf.crispy.util.Util.initJdkLogger();
```

This class initiate the jdk logger with the properties *jdk14-logging.properties*. This file is in the package *net.sf.crispy.util*.

The second possibility is the apache Log4J logging API. This API can you activate with:

```
        PropertyConfigurator.configure("...");
        // replace ... with the configuration property file
```

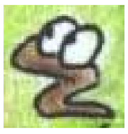
1.10 Comic

Crispy Comic - Actors

Crispy

Erik Question

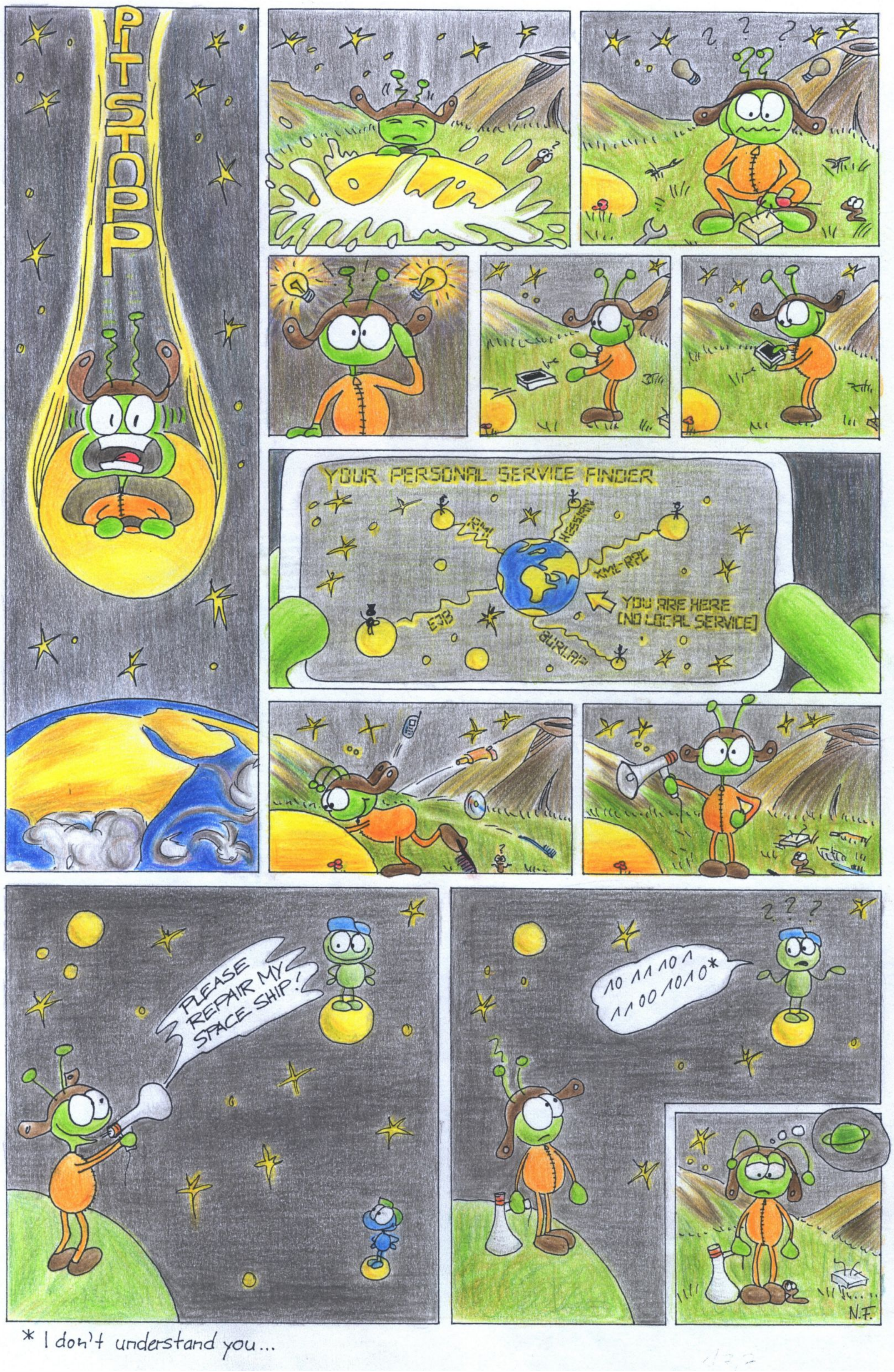
Spacy

Service

Wormy

Crispy 3D

1.10.1 Part one

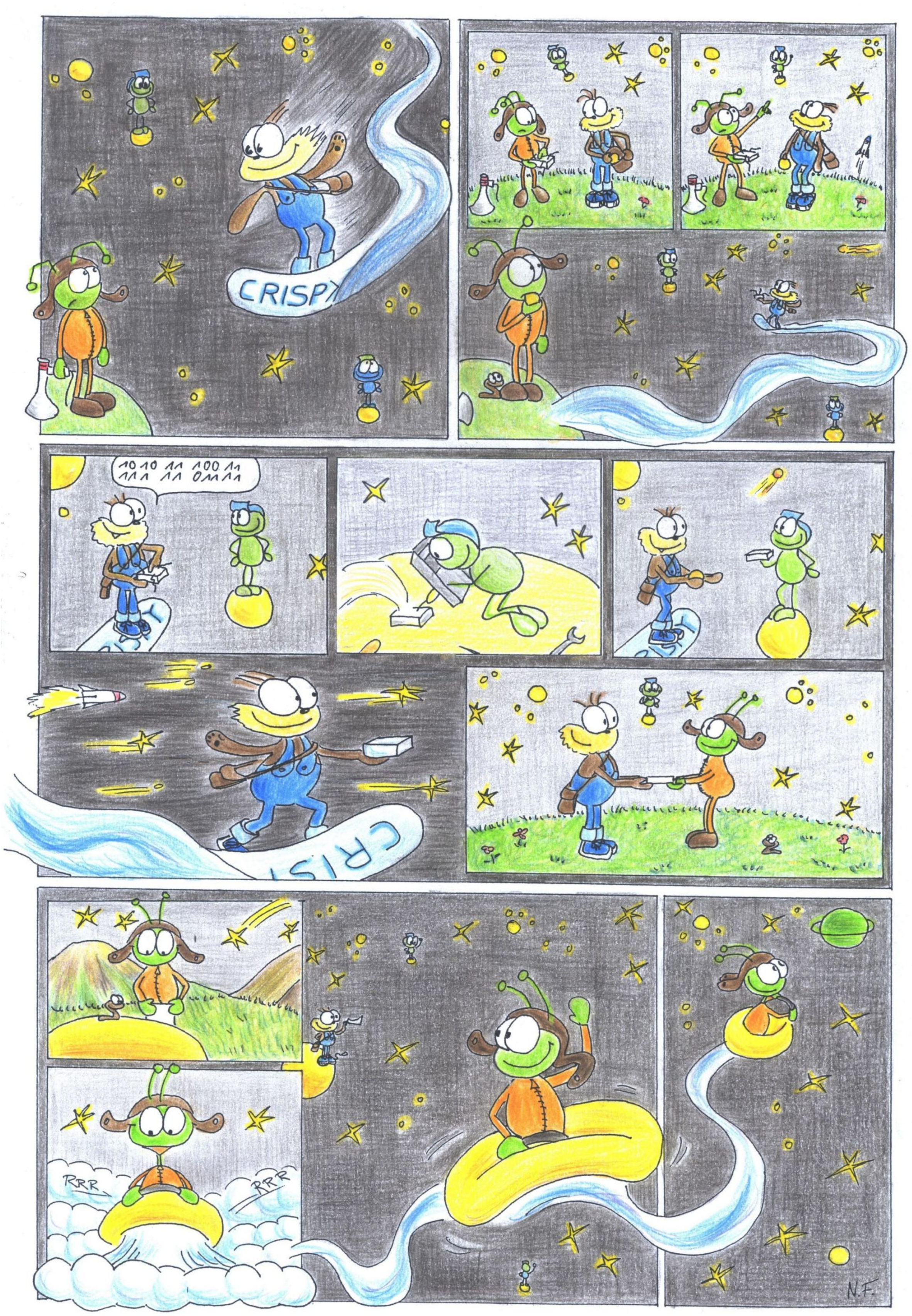
Part one



* I don't understand you...

1.10.2 Part two

Part two


CRISPY
1010 11 100 11
111 11 0111 11
CRISPY
CRISPY
RRR
RRR
N.F.