EChartsmachine code

Rapid Converged Telecom Application Development with E4SS, Grails and SailFin

Gregory W. Bond
AT&T Labs – Research

July 16, 2008

The latest version of this document is available from:
http://echarts.org

Abstract

ECharts for SIP Servlets (E4SS) is a development framework that enables rapid development of SIP protocol-based telecom applications for deployment on SIP servlet application servers like the open source SailFin server. Grails is a rapid web application development framework. This document provides a tutorial-style description of using E4SS, Grails and SailFin to implement, build and run a simple converged telecom application.

Contents

1 Introduction

ECharts for SIP Servlets (E4SS) is an open source development framework that enables rapid development of SIP protocol-based telecom applications for deployment on SIP servlet application servers like the open source SailFin server. However, the most compelling telecom applications interact with more than just the SIP protocol; interaction via HTTP enables a user to interact with a telecom application using a web interface. Telecom applications that interact with both SIP and HTTP are called *converged* applications. Software support for converged applications is currently in its nascent stage. The draft SIP Servlet 1.1 specification released by the JSR 289 working group includes basic support for converged applications. SailFin, which includes a Java EE application server, currently supports much of the specification, and E4SS builds upon this support by providing a light-weight convergence framework.

Beyond this level of support, converged application development also requires that it be possible to integrate an E4SS application with a given web application. Earlier this year I wrote an article explaining how E4SS could be integrated with a traditional HTTP servlet/Java Persistence API (JPA) web application to develop a converged click-to-dial application. Also, Peter Klein described on his blog how to use E4SS with JavaServer Faces/JPA to create a converged call screening application. While the web application frameworks used for these examples are effective, they don't utilize the latest technologies offered in rapid development frameworks like Ruby on Rails (RoR) or Grails. In particular, RoR and Grails provide powerful scripting language support (Ruby and Groovy respectively), high-level object-relational database support (ActiveRecord and GORM, respectively), and both support the DRY ("Don't Repeat Yourself") and CoC ("Convention over Configuration") principles which reduce coding effort and simplify maintenance.

While the prospect of integrating a framework like RoR or Grails with E4SS is attractive, these frameworks are intended for developing stand-alone web applications and don't readily accept input via non-HTTP means. This means integrating E4SS, where inputs arrive via SIP, presents a challenge. Furthermore, both frameworks use languages that differ from Java, the language of SIP Servlet development and E4SS. Although both Ruby (in the form of JRuby) and Groovy can be integrated with Java, Groovy does so more naturally since it is a Java-based scripting language and therefore inherits all of Java's built-in types and class libraries. In addition, Grails support for Java EE application servers is more mature than that for JRuby on Rails. For this reason I decided to look into how to integrate E4SS with Grails and, thanks to a helpful response to a query of mine on the grails-user mailing list, it turns out that this isn't difficult at all.

The remainder of this document is devoted to a tutorial-style description of using E4SS, Grails and SailFin to implement, build and run a simple converged telecom application: call forwarding. The example shows how a reusable E4SS feature, unconditional call forwarding (UCF), is integrated with a web application that supports provisioning the feature's subscriber data. I show how all of this is accomplished using the E4SS convergence framework, Grails and some "magic" code that provides a bridge between the E4SS feature interface and Grails.

2 Preliminaries

Before we get started with coding make sure your environment has all the requisite software for developing and running the example.

- Install SailFin build 40 or greater.
- Install Grails v1.0.3 or greater.
- Install the E4SS DK v2.4-beta or greater.
- Configure SailFin for E4SS.
- Install the CounterPath X-Lite SIP softphone.

3 Coding

Change working directories to wherever you want to create your Grails application and create a new Grails application for call forwarding:

```
shell> grails create-app callForwarding
```

Change working directories to the `callForwarding` directory that you just created. The reusable E4SS call forwarding feature utilizes the E4SS lightweight convergence framework to access a forwarding address for a feature subscriber.

3.1 Grails Domain Class

Now let's start coding, such as it is with Grails. Create the Subscriber domain class. This represents the feature's data schema.

```
shell> grails create-domain-class Subscriber
```

Using your favorite editor, open `grails-app/domain/Subscriber.groovy` and modify it to look like this:

```
class Subscriber {

    // subscriber name
    String name
    // name to forward to
    String forwardingAddress
}
```

3.2 Grails Controller

Create the feature's web interface so that Grail's scaffolding does the heavy lifting. This is done by creating a controller for the Subscriber domain class:

```
shell> grails create-controller Subscriber
```

Edit `grails-app/controllers/SubscriberController.groovy` to look like this:

```
class SubscriberController {

    def scaffold = Subscriber
}
```

3.3 E4SS Telecom Feature

So far we've created a normal Grails web-only application. To handle the telecom side of things we need to incorporate the reusable E4SS unconditional call forwarding feature into our application. The first thing to do is create the feature's jar file and add it, along with some supporting jar files, to the application. Before creating the jar file you must update your Java `CLASSPATH` to include SailFin's `javaee.jar` and `ssa-api.jar` file and you need to set the E4SS `EDK_HOME` environment variable.

```
shell> export CLASSPATH=SAILFIN_HOME/lib/javaee.jar: \
  SAILFIN_HOME/lib/ssa-api.jar:$CLASSPATH
shell> export EDK_HOME=[path to your E4SS installation]
shell> cd $EDK_HOME/features/ucf
shell> ant jar
```

`ucf.jar` is now in the current working directory. Copy the feature jar file and supporting E4SS library jar files to the application's lib directory. These jar files will be included in the application's war file when we create it later.

```
cp $EDK_HOME/features/ucf/ucf.jar $EDK_HOME/lib/approuter.jar \
    $EDK_HOME/lib/sip-sdp.jar $EDK_HOME/lib/echarts.jar \
    $EDK_HOME/lib/echarts-sipservlet.jar lib
```

3.4 Bridging E4SS and Grails

The next bit of code we need is an implementation of the E4SS unconditional call forwarding feature's SIP-to-Java interface. As described in the feature's documentation, this feature's interface defines a single method `getForwardingURI()` that takes the incoming call's INVITE request as a parameter and returns a forwarding URI. When an incoming call is received by the unconditional call forwarding feature it will call the method to determine where to forward the call. What we want is for the feature's interface implementation to query the Grails Susbcriber domain class we defined earlier. This necessitates bridging E4SS and Grails.

The approach we will use is provided by Burt Beckwith in this response to a query of mine on the grails-user mailing list. The overall approach is to create a Grails service with a method to obtain the info required by unconditional call forwarding SIP-to-Java interface implementation. At runtime, the SIP-to-Java interface implementation will call a utility method to obtain an instance of the service, and then call the service method to obtain a forwarding address.

3.5 Grails Service

If you're not already there, change working directories back to the Grails callForwarding application directory. Create the Java interface for the service by creating the file (and ancestor directories) `org/echarts/servlet/sip/applications/callForwarding/CallForwardingServiceInterface.java` under `src/java` and edit the file to look like this:

```
package org.echarts.servlet.sip.applications.callForwarding;

public interface CallForwardingServiceInterface {
    public String getForwardingName(String name);
}
```

Create a Grails service that implements the service interface.

```
shell> grails create-service CallForwarding
```

Edit `grails-app/services/CallForwardingService.groovy` so it looks like this:

```
import org.echarts.servlet.sip.applications.callForwarding.CallForwardingServiceInterface

class CallForwardingService implements CallForwardingServiceInterface {

    def String getForwardingName(String name) {
        Subscriber subscriber
        if ((subscriber = Subscriber.findByName(name)) == null)
            return null
        else
            return subscriber.forwardingAddress
    }
}
```

3.6 E4SS SIP-to-Java Interface

For the unconditional call forwarding SIP-to-Java interface implementation, create the file (and ancestor directories) `org/echarts/servlet/sip/applications/callForwarding/UCFServletToJavaImpl.java` under the `src/java` directory, and edit it to look like this:

```
package org.echarts.servlet.sip.applications.callForwarding;

import org.echarts.servlet.sip.features.ucf.UCFServletToJava;
import org.echarts.servlet.sip.EChartsProxyServletToJava;
import org.echarts.monitor.InfoEvent;
import org.echarts.servlet.sip.grails.util.SpringUtils;
import javax.servlet.sip.SipFactory;
import javax.servlet.sip.URI;
import javax.servlet.sip.SipURI;
import javax.servlet.sip.SipServletRequest;
import javax.servlet.sip.ServletParseException;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class UCFServletToJavaImpl extends EChartsProxyServletToJava
    implements UCFServletToJava {

    public URI getForwardingURI(final SipServletRequest request) {
        SipURI requestURI = (SipURI) request.getRequestURI();
        // get user name from incoming request
        final String name = requestURI.getUser();
        // get call forwarding service bean
        final CallForwardingServiceInterface callForwardingService =
            SpringUtils.getBean("callForwardingService");
        // get provisioned forwarding address for current name via service bean
        String forwardingAddress = (String) callForwardingService.getForwardingName(name);
        putEvent(new InfoEvent("forwarding address for " + name +
            " is: " + forwardingAddress));
        if ( forwardingAddress != null && forwardingAddress.length() > 0 ) {
            SipFactory sipFactory = null;
            try {
                // obtain SipFactory instance via JNDI lookup
                sipFactory = (SipFactory) new InitialContext().lookup("sip/SipFactory");
            } catch (NamingException e) {
                putEvent(new InfoEvent(e, "exception encountered looking up SipFactory"));
            }
            if (sipFactory != null) {
                try {
                    // create a SIP URI for the forwarding address
                    requestURI = (SipURI) sipFactory.createURI(forwardingAddress);
                } catch (ServletParseException e) {
                    putEvent(new InfoEvent(e,
                        "unable to create SIP URI using forwarding address: " +
                        forwardingAddress));
                }
            }
        }
        return requestURI;
    }
}
```

The code above uses “old fashioned” JNDI lookup to obtain the SipFactory instead of using resource injection because, try as I might, I couldn’t get SailFin’s resource injection to work.

3.7 Grails Service from Java

The last bit of code is a minor variation on what Burt Beckwith provided in his grails-user post (I added a missing import declaration). Under the `src/java` directory, create the file (and ancestor directories) `org/echarts/servlet/sip/grails/util/SpringUtils.java` and edit it to look like this:

```
package org.echarts.servlet.sip.grails.util;

import org.codehaus.groovy.grails.web.context.ServletContextHolder;
import org.codehaus.groovy.grails.web.servlet.GrailsApplicationAttributes;
import org.springframework.context.ApplicationContext;
import org.springframework.web.context.support.WebApplicationContextUtils;

public class SpringUtils {

    /**
     * Get a spring bean by name.
     */
    @SuppressWarnings("unchecked")
    public static <T> T getBean(final String name) {
        ApplicationContext ctx =
            (ApplicationContext)ServletContextHolder.getServletContext().
```

```
                getAttribute(GrailsApplicationAttributes.APPLICATION_CONTEXT);
        return (T) ctx.getBean(name);
    }
}
```

4 Configuration

Coding is now complete but we have to perform one configuration step prior to moving on to the build. Since our application will directly access SailFin's Derby database, you should edit the file `grails-app/conf/DataSource.groovy` to look like this:

```
dataSource {
    pooled = true
    driverClassName = "org.apache.derby.jdbc.EmbeddedDriver"
    dialect = "org.hibernate.dialect.DerbyDialect"
    username = "APP"
    password = "APP"
}
hibernate {
    cache.use_second_level_cache=true
    cache.use_query_cache=true
    cache.provider_class='com.opensymphony.oscache.hibernate.OSCacheProvider'
}
// environment specific settings
environments {
    production {
        dataSource {
            dbCreate = "create-drop"
            url = "jdbc:derby://localhost:1527/sun-appserv-samples;create=true"
        }
    }
}
```

This file specifies that our application will access the `sun-appserv-samples` database that is pre-configured with SailFin's Derby database. Note that we've excluded development and test data source environments here for simplicity's sake.

5 Build

5.1 WAR file

Since I haven't figured out how to configure the Java CLASSPATH for Grails builds, I copied the SIP Servlet API jar and the Java EE jar from SailFin's lib to Grails' lib:

```
shell> cp SAILFIN_HOME/lib/ssa-api.jar GRAILS_HOME/lib
shell> cp SAILFIN_HOME/lib/javaee.jar GRAILS_HOME/lib
```

Build the war file:

```
shell> grails war
```

Barring any compilation errors, `callForwarding-0.1.war` will be in the current directory.

5.2 SIP Deployment Descriptor

We also need to include a SIP deployment descriptor (sip.xml) in the war file. Create a WEB-INF directory and then create the file `WEB-INF/sip.xml` so that it looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sip-app PUBLIC "-//Java Community Process//DTD SIP Application 1.0//EN"
    "http://www.jcp.org/dtd/sip-app_1_0.dtd">

<sip-app>
  <display-name>Call Forwarding</display-name>

  <servlet>
    <servlet-name>callForwarding</servlet-name>
    <servlet-class>org.echarts.servlet.sip.features.ucf.UCFServlet</servlet-class>
    <init-param>
      <param-name>isRecordRoute</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>isSupervised</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>sipToJavaClassName</param-name>
      <param-value>
          org.echarts.servlet.sip.applications.callForwarding.UCFServletToJavaImpl
      </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>callForwarding</servlet-name>
    <pattern>
      <equal>
        <var>request.method</var>
        <value>INVITE</value>
      </equal>
    </pattern>
  </servlet-mapping>

</sip-app>
```

Note that SailFin does not currently support `false` values for `isRecordRoute` and `isSupervised` so we set their values to `true` even though this isn't required for the unconditional call forwarding feature.

Add the SIP deployment descriptor to the application war file:

```
shell> jar uf callForwarding-0.1.war WEB-INF/sip.xml
```

6 Deployment

Start up SailFin and SailFin's Derby database and deploy the application:

```
SAILFIN_HOME/bin/asadmin start-domain domain1
SAILFIN_HOME/bin/asadmin start-database
SAILFIN_HOME/bin/asadmin deploy callForwarding-0.1.war
```

You can confirm that the application deployed successfully if the last entry in the SailFin server log `SAILFIN_HOME/domains/domain1/logs/server.log` looks like this:

```
[# | 2008-07-10T21:45:15.103-0400 | INFO | sun-comms-appserver1.0 |
    javax.enterprise.system.container.sip |
    _ThreadID=17;_ThreadName=httpWorkerThread-4848-1; |
    Enabling the application Call Forwarding | #]
```

7 Test Run

To test run our application, Feathers McGraw will call Wallace. Since Wallace is a Call Forwarding subscriber has configured the feature to forward his calls to Gromit, then Gromit will receive the call.

7.1 Softphone Configuration

Prior to making a test call it is necessary to configure the softphones. I will describe how this is done on Mac OS X. Similar, platform-specific, steps must be performed for different operating systems. In a terminal window start one instance of an X-Lite phone:

```
shell> /Applications/eyeBeam.app/Contents/MacOS/eyeBeam
```

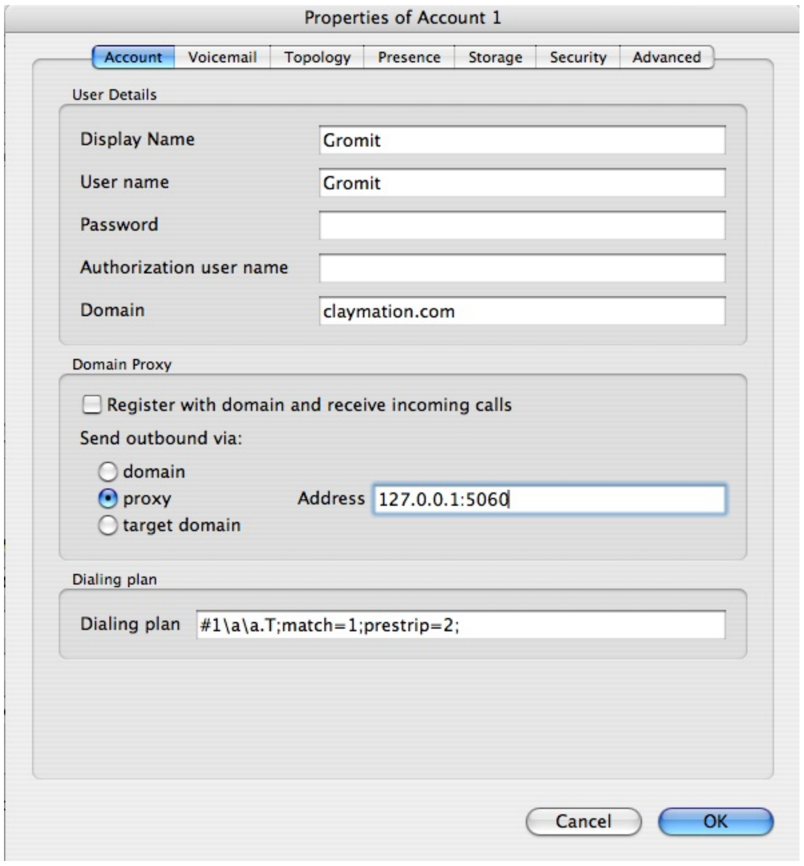
This softphone will be Gromit's. Choose the "SIP Account Settings..." menu option and click the "Add..." button to create a new SIP account. Edit the account to look like the screenshot shown in Figure 1. Note that the "Domain" field value is required, but it doesn't matter what the value is. Although Gromit's phone won't make outbound calls in this test, we'll go ahead and configure the account to proxy outbound calls to our local SailFin instance, which is listening for calls on port 5060.

Click "OK" to add the account and then enable the account by selecting the checkbox next to it in the SIP Accounts list as shown in Figure 2.

Click on "Close". Before we configure our next softphone we have to determine what port this softphone is listening on for incoming calls. This port number will be used in Wallace's forwarding address. To determine the port number on Mac OS X execute the following command:

```
shell> sudo lsof -i -P | grep -i "listen"
```

Executing the command will prompt you for your password and, assuming you have Administrator rights on your machine, you will be presented with a list that will include the following two entries:



User Details

Display Name	Gromit
User name	Gromit
Password	
Authorization user name	
Domain	claymation.com

Domain Proxy

☐ Register with domain and receive incoming calls

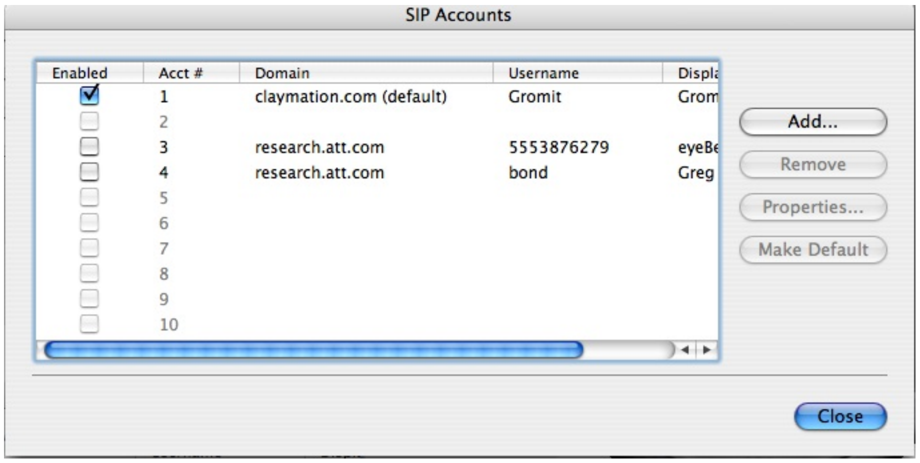
Send outbound via:

○ domain
● proxy — Address 127.0.0.1:5060
○ target domain

Dialing plan

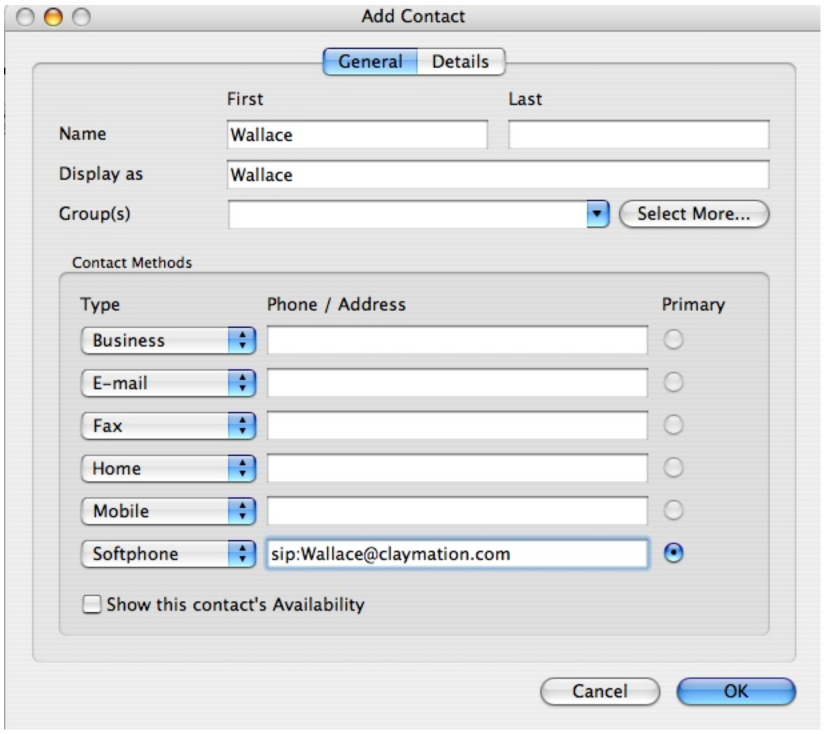
Dialing plan: #1\a\a.T;match=1;prestrip=2;

Figure 1: Gromit's account



Enabled	Acct #	Domain	Username	Displa
✓	1	claymation.com (default)	Gromit	Grom
☐	2			
☐	3	research.att.com	5553876279	eyeBe
☐	4	research.att.com	bond	Greg
☐	5			
☐	6			
☐	7			
☐	8			
☐	9			
☐	10			

Figure 2: Enabling Gromit's account


Add Contact

General | Details

	First	Last
Name	Wallace	
Display as	Wallace	
Group(s)		

Contact Methods

Type	Phone / Address	Primary
Business		○
E-mail		○
Fax		○
Home		○
Mobile		○
Softphone	sip:Wallace@claymation.com	●

☐ Show this contact's Availability

Figure 3: Adding Wallace to Feathers McGraw's contacts

```
eyeBeam   14519  greg   14u  IPv4 0x4269c9c      0t0    TCP *:25644 (LISTEN)
eyeBeam   14519  greg   15u  IPv4 0x4442920      0t0    TCP *:25645 (LISTEN)
```

Make note of the first (even valued) port number listed, in this case 25644.

Using the same approach we used to configure Gromit's softphone, start a softphone in another terminal window and configure it for Feathers McGraw (Display Name and User Name should be "FeathersMcGraw").

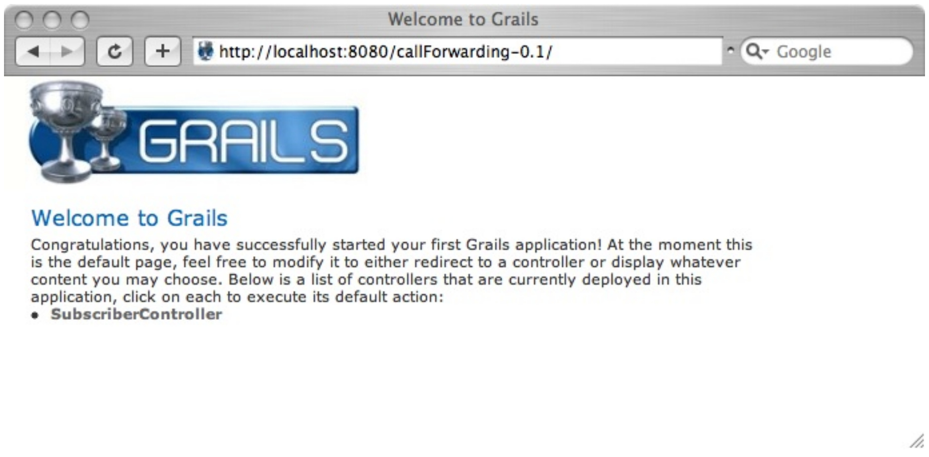
Now add an entry for Wallace in Feathers McGraw's contacts. If the Contacts drawer isn't displayed for Feathers McGraw's softphone (on the right-hand side of the softphone interface), then click on the small white triangle near the top right-hand side of the interface. In the "Contacts" drop-down menu, choose "Add Contact...". Add a softphone contact for Wallace as shown in Figure 3 and click "OK". Note that the domain name associated with Wallace's address is imaginary, but that doesn't matter because when Feathers calls Wallace the call forwarding feature will replace Wallace's address with a valid address for Gromit.

7.2 Call Forwarding Configuration

Now provision the data for Wallace's call forwarding feature. Open a browser window to our application's automatically generated home page shown in Figure 4.

Click on the SubscriberController link to see the page shown in Figure 5.

Click on the "New Subscriber" link and create an entry for subscriber name Wallace whose forwarding address is sip:Gromit@127.0.0.1:25644, where 25644 is the listening port number for Gromit's softphone that we determined earlier. See Figure 6 for a screenshot. Click on the "Create" button to save the entry.

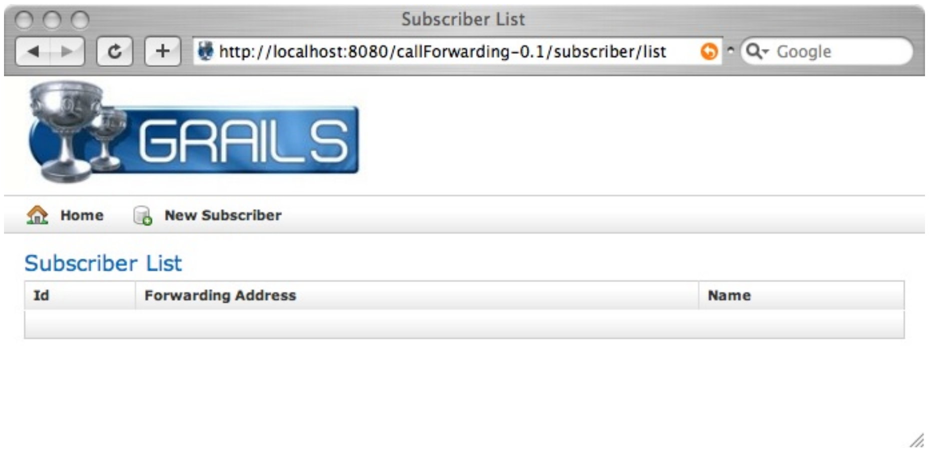


Welcome to Grails

Congratulations, you have successfully started your first Grails application! At the moment this is the default page, feel free to modify it to either redirect to a controller or display whatever content you may choose. Below is a list of controllers that are currently deployed in this application, click on each to execute its default action:

- SubscriberController

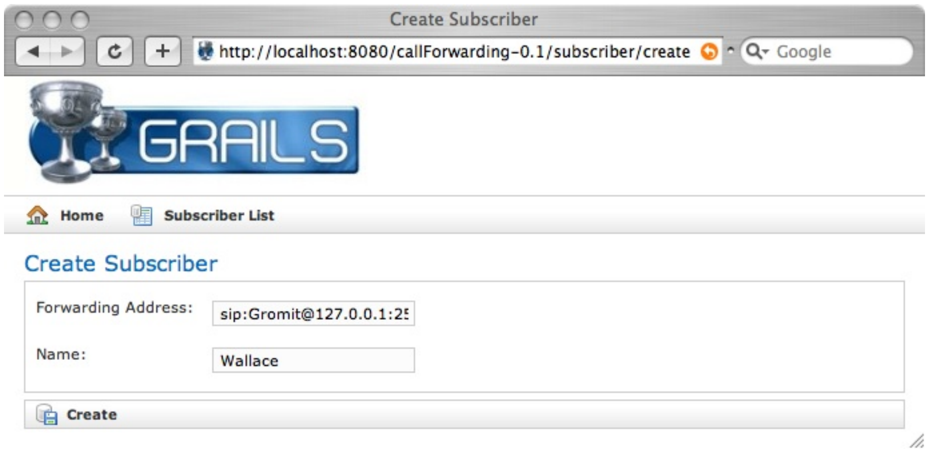
Figure 4: Call Forwarding home page



Subscriber List

Id	Forwarding Address	Name

Figure 5: Call Forwarding Subscriber controller page

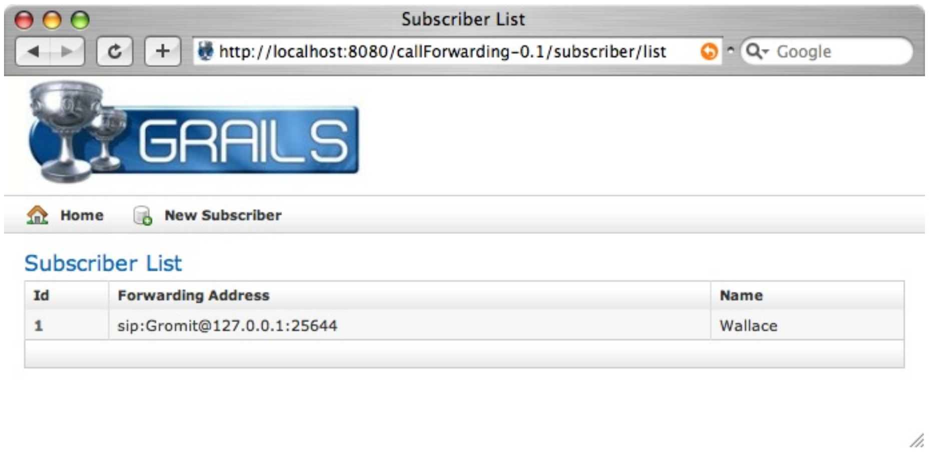


Create Subscriber

Forwarding Address: sip:Gromit@127.0.0.1:25

Name: Wallace

Figure 6: Call Forwarding create Subscriber page


Subscriber List

http://localhost:8080/callForwarding-0.1/subscriber/list

Home   New Subscriber

Subscriber List

Id	Forwarding Address	Name
1	sip:Gromit@127.0.0.1:25644	Wallace

Figure 7: Call Forwarding Subscriber list page

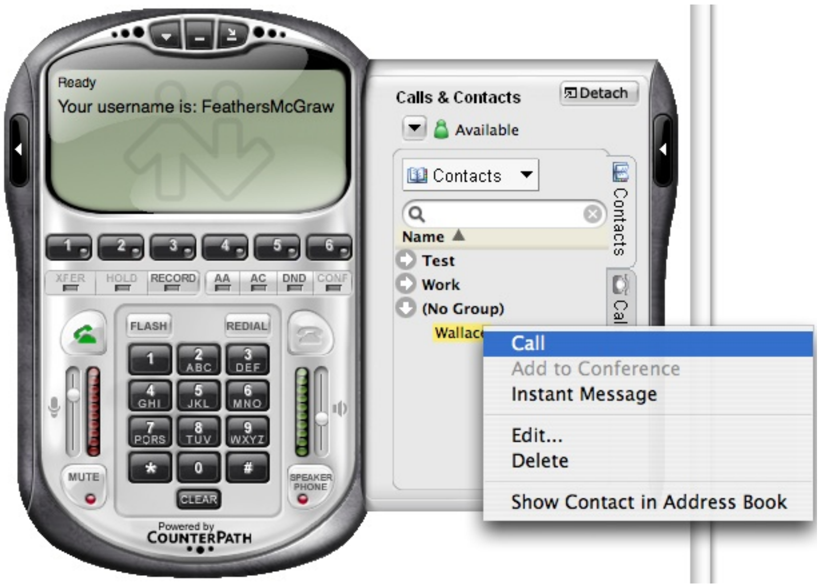


Figure 8: Feathers calls Wallace

Now the "Subscriber List" view should look like the screenshot in Figure 7.

7.3 Test Call

Wallace's call forwarding feature is provisioned so now we're ready to make a test call from Feathers McGraw to Wallace. Using Feathers McGraw's softphone, right-click on Wallace's contact name choosing "Call" from the contextual menu as shown in Figure 8.

If all goes well, then Gromit's softphone should ring. Accept the call using Gromit's softphone and you have an end-to-end connection between Feathers McGraw and Gromit! If Gromit's softphone doesn't ring then take a look in `SAILFIN_HOME/domains/domain1/logs/server.log` for clues.

7.4 Clean Up

To undeploy the application and stop SailFin after you've finished testing:

```
shell> SAILFIN_HOME/bin/asadmin undeploy callForwarding-0.1
shell> SAILFIN_HOME/bin/asadmin stop-database
shell> SAILFIN_HOME/bin/asadmin stop-domain domain1
```

8 Concluding Remarks

This article has shown how to rapidly develop a converged telecom application using E4SS and Grails. While the example application is a simple one, the same approach can be used for more complex applications where the telecom component is either provided in the form of a reusable E4SS feature as it was in this article, or as a custom E4SS feature written with the ECharts language and E4SS machine fragments.

Converged applications can then be composed using the SIP Servlet 1.1 application routing mechanism to form arbitrarily complex converged services. This modular approach helps to manage the complexity of service development and maintenance. In this context, I am currently experimenting with incorporating a shared set of GORM domain classes across composed converged applications developed using E4SS and Grails. This is a likely subject for a future report.