IBM

developerWorks.

# Rich Internet Applications with Grails, Part 1: Build a Web application using Grails and Flex

Skill Level: Intermediate

Michael Galpin
Software architect
eBay

24 Feb 2009

Rich Internet Applications (RIAs) promise the dynamism and functionality of desktop applications through the browser. One of the key characteristics is moving your presentation layer to the client and backing it with a robust RESTful service layer on the server. This idea is being popularized with buzzwords like SOUI (Service Oriented User Interface) and SOFEA (Service Oriented Front End Architecture). In this article, the first of a two-part series, you will see how simple it is to create a Web service back end using Groovy's Grails Web application framework, and you will hook it up to an RIA developed with Adobe's Flex framework.

## About this series

This series explores application architectures that use a Service Oriented Architecture (SOA) on the back end implemented with the Grails framework. Explore how much Grails simplifies creating Web applications in general and Web services in particular. This kind of back end can be easily hooked up to any pure client-side application. In Part 1, you will use Adobe Flex to create such an application that leverages the Flash Player. In Part 2, you will use the Google Web Toolkit to create the front end in pure JavaScript.

## Prerequisites

In this article you will build a Web application using Grails and Flex. The Grails framework is built on the Groovy programming language, a dynamic language for the Java™ platform. Familiarity with Groovy is great, but not completely necessary.


© Copyright IBM Corporation 2008. All rights reserved.

Knowledge of Java can be a good substitute, or even other dynamic languages like Ruby or Python. Grails 1.0.3 was used with developing this article (see Resources). Grails can work with numerous databases or application servers, but none is needed for this article—Grails comes with both. The front end is built using Flex, an application framework that uses the ActionScript programming language and runs on the Flash Player. Again, it is okay if you are not already familiar with Flex and ActionScript. Familiarity with Java and JavaScript will help in picking up Flex. Flex SDK 3.2 or higher is needed to compile the code in this article (see Resources). To run the application, you need the Flash Player Version 10.0 or higher.

## Architecture

Many organizations are pursuing a Service Oriented Architecture (SOA). This is often done to make your architecture more agile, allowing your business to more rapidly evolve. Of course, you probably have another pressing initiative in your organization: to modernize your user interfaces into a Rich Internet Application. It can be tough to deal with both buzz words, SOA and RIA. It turns out, however, that these two things work well together. You can use an SOA design to deploy services to your application servers. You can move all of your presentation logic to the client and leverage a powerful front-end technology such as Flex to create an RIA. That is exactly what you will do in this series, and you will start by creating a Web service using Grails.

## Web services

When many developers hear the term *Web service*, they think of one thing: SOAP (Simple Object Access Protocol). This has a negative connotation with many developers, as they think of SOAP as being a heavy and complicated technology. However, Web services do not have to be that way. REST (Representational State Transfer)-style Web services have gained popularity because of their simple semantics. They are easier to create and to consume. They can use XML, just like SOAP services, but this can be Plain Old XML (POX) with no fancy wrappers and headers like with SOAP. The Grails framework makes it very easy to create these kind of Web services, so let's get started with a Grails domain model.

### Grails domain model

Grails is a general purpose Web development framework. Most Web applications use a relational database to store and retrieve the data used in an application, and thus Grails comes with a powerful Object Relational Modeling (ORM) technology known as GORM. With GORM, you can easily model your domain objects and have them persisted to a relational database of your choice, without ever dealing with any SQL. GORM uses the popular Hibernate library for generating database-specific and optimized SQL, and for managing the life cycles of domain objects. Before getting in


© Copyright IBM Corporation 2008. All rights reserved.

to using GORM, let's quickly discuss the application you will create and what you need to model with GORM.

In the sample application, you will create a Web application that mimics the functionality of the popular site Digg (see Resources). On Digg, users can submit links to stories (Web pages). Other users can then read these stories and vote for or against them. You will capture all of this basic functionality in your application. It will let people submit and vote for stories anonymously, so you will not need to model users, just stories. With that in mind, here is the GORM model for a story in the example application, shown in Listing 1.

### Listing 1. The Story model

```
class Story {
    String link
    String title
    String
description
    String tags
    String
category
    int votesFor
    int
votesAgainst
}
```

That is all the code needed to model the domain object. You declare its properties and the types of those properties. This will allow Grails to create the table for you and dynamically create methods for both reading and writing data from that table. This is one of the major benefits of Grails. You only put data modeling code in one place and never have to write any boilerplate code for simple reads and writes. Now that you have the domain model in place, you can create some business services that use this domain model.

### Business services

One of the benefits of an SOA is that it allows you to model your system in a very natural way. What are some of the operations you would like to perform? That is how you define the business services of the application. For example, you will want to be able to browse and search for stories, so you should create a service for this, as shown in Listing 2.

### Listing 2. Search service

```
class
SearchService {

    boolean
transactional =
false
```


© Copyright IBM Corporation 2008. All rights reserved.

```
    def list() {
Story.list()
    }

    def
listCategory(catName){
Story.findAllWhere(category:catName)
    }

    def
searchTag(tag){
Story.findAllByTagsIlike("%"+tag+"%")
    }
}
```

The first thing you might notice is the boolean flag `transactional`. Grails is built on top of numerous proven technologies, including Spring. It uses Spring's declarative transactions to decorate any service with transactions. If you were performing updates on data, or you wanted this service to be used by other transactional services, then you would want to enable this. You want this service to be a read-only service. It is not going to take the place of data access, since you have a domain model that already handles all of that.

The first service operation retrieves all stories. The second retrieves those for a given category. You use GORM's where-style finder for this. This lets you pass in a map of name/value pairs to form a `WHERE` clause in an SQL query. The third operation allows you to search for stories with a given tag. Here you use GORM's dynamic finders. This lets you incorporate the query clause as part of the name of the finder method. So to query on the tags column, you use `findAllByTags`. Also notice the `like` suffix; this is a case-insensitive `SQL LIKE` clause. This lets you find all stories where the tag parameter occurs as a substring of the tags attribute. For example, if a story had been tagged "BarackObama" it would show up in a search for "obama."

You have created three simple but powerful ways to search for stories. Notice how succinct the syntax for this is. If you are a Java programmer, just imagine how much you would normally write to do this. The search service is powerful, but there is nothing to search yet. You need another service for managing individual stories. I have called this the Story service, and it is shown in Listing 3.

### Listing 3. Story service

```
class
StoryService {

    boolean
transactional =
true

    def
create(story) {
story.votesFor =
0
```


© Copyright IBM Corporation 2008. All rights reserved.

```
story.votesAgainst
= 0
log.info("Creating
story="+story.title)
if(!story.save(flush:true)
) {
story.errors.each
{
log.error(it)
            }
        }
log.info("Saved
story="+story.title
+ " id=" +
story.id)
        story
    }
    def
voteFor(storyId){
log.info("Getting
story for
id="+storyId)
        def story
=
Story.get(storyId)
log.info("Story
found
title="+story.title
+ "
votesFor="+story.votesFor)
story.votesFor +=
1
if(!story.save(flush:true)
) {
story.errors.each
{
log.error(it)
            }
        }
        story
    }
    def
voteAgainst(storyId){
log.info("Getting
story for
id="+storyId)
        def story
=
Story.get(storyId)
log.info("Story
found
title="+story.title
+ "
votesAgainst="+story.votesAgainst)
story.votesAgainst
+= 1
if(!story.save(flush:true)
) {
story.errors.each
{
log.error(it)
            }
        }
        story
    }
}
```


© Copyright IBM Corporation 2008. All rights reserved.

The Story service has three operations. First, you can create a new story. Next, you have operations for voting for and voting against a given story, via the ID of the story. In each case, notice that you have done some logging. Grails makes it easy to log—just use the implicit log object and typical log4j-style methods: `log.debug`, `log.info`, `log.error`, and so on. The story is saved (either inserted or deleted) using the save method on the `Story` instance. Notice how you can check for errors by examining the errors property of the story instance. For example, if the story was missing a value for a required field, then this would show up as part of story.errors. Finally notice that this service is transactional. This will tell Grails (and thus Spring) to either use an existing transaction (if one is already present), or create a new one whenever any of these operations are invoked. This is particularly important for the voting operations, because in those you have to read the story from the database first and then update a column. Now that you have created your basic business services, expose them as an API by creating a Web service around them, as you'll see next.

### Exposing the API

As developers, we often think of APIs as code that we write that will be called by other developers. In a Service Oriented Architecture, this is still true. However, the API is not a Java interface or something similar; it is a Web service signature. It is the Web service that exposes the API and allows it to be invoked by others. Grails makes it easy to create a Web service—it is just another controller in a Grails application. Take a look at Listing 4 to see the API for the application.

### Listing 4. The API controller

```
import
grails.converters.*

class
ApiController {
    // injected
services
    def
searchService
    def
storyService

    // set the
default action
    def
defaultAction =
"stories"

    def stories =
{
        stories =
searchService.list()
        render
stories as XML
    }

    def submit =
{
```


© Copyright IBM Corporation 2008. All rights reserved.

```
        def story
= new
Story(params)
        story =
storyService.create(story)
log.info("Story
saved
story="+story)
        render
story as XML
    }

    def digg = {
        def story
=
storyService.voteFor(params.id)
        render
story as XML
    }

    def bury = {
        def story
=
storyService.voteAgainst(params.id)
        render
story as XML
    }
}
```

The first thing you will notice is the presence of the business services you created in the previous section. These services will be automatically injected by Grails. Actually, they were injected by the Spring framework, one of the technologies that Grails is built on. You simply follow the naming convention (`searchService` for the instance variable that will be used to reference an instance of the `SearchService` class and so on), and Grails does everything for you.

The API presents four operations (actions) that can be called: `stories`, `submit`, `digg`, and `bury`. In each case, you delegate to a business service, take the result of that call and serialize it to XML that you send to the client. Grails makes this rendering easy, just using the render function and an XML converter. Finally, notice that the `submit`, `digg`, and `bury` actions all use the `params` object. This is a hashtable of the HTTP request parameters and their values. For the `digg` and `bury`, you simply retrieve the `id` parameter. For the `submit` action, you passed the whole `params` object to the constructor of the `Story` class. This uses Grails data binding—as long as the parameter names match up to the property names of the class, Grails will set them for you. This is just another case of how Grails makes development easier. You have written a very minimal amount of code, but that was all you needed to create the services and expose them as Web services. Now you can create a rich presentation layer that uses these services.

## Presentation

You have used an SOA for the back end of the application. This will allow you to create many different kinds of presentation layers on top of it. You will first build a


© Copyright IBM Corporation 2008. All rights reserved.

Flash-based user interface using the Flex framework. However, you could just as easily use any other client-side presentation technology. You could even create a "thick" desktop client. However, there is no need, as you will be able to get a rich user experience from a Web client. Take a look at some simple use cases below, and build the user interface for them using Flex. Let's get started by listing all of the stories.

### Listing stories

To list the stories, you know what API to call from the back end. Or do you? When you created the back end, did you ever map any particular URLs to your API controller and its methods? Obviously you did not, but once again Grails will make your life easier. It uses convention over configuration for this. So, to call the stories action on the API controller for the digg application, the URL will be `http://<root>/digg/api/stories`. Since this is a RESTful Web service, this should be an HTTP `GET`. You can do this in your browser directly and get XML that looks something like the XML shown in Listing 5.

### Listing 5. Sample XML response

```
<?xml
version="1.0"
encoding="UTF-8"?><list>
  <story id="12">
<category>technology</category>
<description>This
session discusses
approaches and
techniques to
implementing Data
Visualization and
Dashboards within
Flex
3.</description>
<link>http://onflash.org/ted/2008/10/360flex-sj-
2008-data-visualization-and.php</link>
    <tags>flash,
flex</tags>
    <title>Data
Visualization and
Dashboards</title>
<votesAgainst>0</votesAgainst>
<votesFor>0</votesFor>
  </story>
  <story id="13">
<category>technology</category>
<description>Make
your code
snippets like
really nice when
you blog about
programming.</description>
<link>http://bc-squared.blogspot.com/2008/07/
syntax-highlighting-and-code-snippets.html</link>
<tags>programming,
blogging,
javascript</tags>
    <title>Syntax
```


© Copyright IBM Corporation 2008. All rights reserved.

```
Highlighting and
Code Snippets in
a blog</title>
<votesAgainst>0</votesAgainst>
<votesFor>0</votesFor>
  </story>
  <story id="14">
<category>miscellaneous</category>
<description>You
need a get
notebook if you
are going to take
good
notes.</description>
<link>http://www.randsinrepose.com/archives/2008/06/01/
sweet_decay.html</link>
<tags>notebooks</tags>
    <title>Sweet
Decay</title>
<votesAgainst>0</votesAgainst>
<votesFor>0</votesFor>
  </story>
  <story id="16">
<category>technology</category>
<description>If
there was one
thing I could
teach every
engineer, it
would be how to
market.
</description>
<link>http://www.codinghorror.com/blog/archives/001177.html</link>
<tags>programming,
funny</tags>
    <title>The
One Thing Every
Software Engineer
Should
Know</title>
<votesAgainst>0</votesAgainst>
<votesFor>0</votesFor>
  </story>
</list>
```

Obviously, the exact contents depend on what data you have saved in your database. The important thing to note is the structure of how Grails serializes the Groovy objects as XML. Now you are ready to write some ActionScript code for working with the services.

### Data access

One of the nice things about moving all of the presentation tier to the client is that it is much cleaner architecturally. It is easy to follow a traditional model-view-controller (MVC) paradigm now, since nothing gets spread out between the server and client. So you start with a model for representing the data structure and then encapsulate access to the data. This is shown in the `Story` class in Listing 6.

### Listing 6. Story class


© Copyright IBM Corporation 2008. All rights reserved.

```
public class
Story extends
EventDispatcher
{
    private
static const
LIST_URL:String =
"http://localhost:8080/digg/api/stories";

    [Bindable]
public var
id:Number;
    [Bindable]
public var
title:String;
    [Bindable]
public var
link:String;
    [Bindable]
public var
category:String;
    [Bindable]
public var
description:String;
    [Bindable]
public var
tags:String;
    [Bindable]
public var
votesFor:int;
    [Bindable]
public var
votesAgainst:int;

    private
static var
listStoriesLoader:URLLoader;
    private
static var
dispatcher:Story
= new Story();

        public
function
Story(data:XML=null)
        {
            if
(data)
            {
id = data.@id;
title =
data.title;
link = data.link;
category =
data.category;
description =
data.description;
tags = data.tags;
votesFor =
Number(data.votesFor);
votesAgainst =
Number(data.votesAgainst);
            }
        }

}
```


© Copyright IBM Corporation 2008. All rights reserved.

Listing 6 shows the basics of the `Story` class. It has several fields, corresponding to the data structure you will get from the service. The constructor takes an optional XML object and populates its fields from that object. Notice the succinct syntax for accessing the XML data. ActionScript implements the E4X standard as a simplified way of working with XML. It is similar to XPath, but uses a syntax that is more natural for object-oriented programming languages. Also notice that each property has been decorated with `[Bindable]`. This is an ActionScript annotation. It allows a UI component to be bound to the field, so that if the field changes, the UI is updated automatically. Finally, notice the static variable, *listStoriesLoader*. This is an instance of `URLLoader`, an ActionScript class used for sending HTTP requests. It is used by a static method in the `Story` class for loading all of the stories via the API. This is shown in Listing 7.

### Listing 7. List Stories method

```
public class
Story extends
EventDispatcher
{
    public static
function
list(loadHandler:Function,
errHandler:Function=null):void
    {
        var
req:URLRequest =
new
URLRequest(LIST_URL);
listStoriesLoader
= new
URLLoader(req);
dispatcher.addEventListener(DiggEvent.ON_LIST_SUCCESS,
loadHandler);
        if
(errHandler !=
null)
        {
dispatcher.addEventListener(DiggEvent.ON_LIST_FAILURE,
errHandler);
        }
listStoriesLoader.addEventListener(Event.COMPLETE,
listHandler);
listStoriesLoader.addEventListener(IOErrorEvent.IO_ERROR,
listErrorHandler);
listStoriesLoader.load(req);
    }
    private
static function
listHandler(e:Event):void
    {
        var
event:DiggEvent =
new
DiggEvent(DiggEvent.ON_LIST_SUCCESS);
        var
data:Array = [];
        var
storiesXml:XML =
XML(listStoriesLoader.data);
        for (var
i:int=0;i<storiesXml.children().length();i++)
```


© Copyright IBM Corporation 2008. All rights reserved.

```
        {
            var
storyXml:XML =
storiesXml.story[i];
            var
story:Story = new
Story(storyXml);
data[data.length]
= story;
        }
event.data =
data;
dispatcher.dispatchEvent(event);
    }
}
```

The `list` method is what the controller will be able to invoke. It sends off an HTTP request and registers event listeners for when that request completes. This is needed because any HTTP request in Flash is asynchronous. When the request completes, the `listHandler` method is invoked. This parses through the XML data from the service, once again using E4X. It creates an array of `Story` instances and attaches this array to a custom event that is then dispatched. Take a look at that custom event in Listing 8.

### Listing 8. DiggEvent

```
public class
DiggEvent extends
Event
{
    public static
const
ON_STORY_SUBMIT_SUCCESS:String
=
"onStorySubmitSuccess";
    public static
const
ON_STORY_SUBMIT_FAILURE:String
=
"onStorySubmitFailure";
    public static
const
ON_LIST_SUCCESS:String
=
"onListSuccess";
    public static
const
ON_LIST_FAILURE:String
=
"onListFailure";
    public static
const
ON_STORY_VOTE_SUCCESS:String
=
"onStoryVoteSuccess";
    public static
const
ON_STORY_VOTE_FAILURE:String
=
"onStoryVoteFailure";

    public var
```


© Copyright IBM Corporation 2008. All rights reserved.

```
data:Object = {};
    public
function
DiggEvent(type:String,
bubbles:Boolean=false,
cancelable:Boolean=false)
    {
super(type,
bubbles,
cancelable);
    }

}
```

Using a custom event class is a common paradigm in ActionScript development, because any server interactions have to be asynchronous. The controller can call the method on your model class and register its own event handlers that look for the custom events. You can decorate the custom event with extra fields. In this case you just added on a data field that is generic and can be reused. Now that you have looked at the model for the presentation tier, look at how it gets used by the controller.

### Application controller

The controller is responsible for coordinating calls to the model, providing the model to the view, and responding to events from the view. Take a look at the controller code in Listing 9.

### Listing 9. DiggController

```


public class
DiggController
extends
Application
{
    [Bindable]
    public var
stories:ArrayCollection;
    [Bindable]
    public var
subBtnLabel:String
= 'Submit a new
Story';

    public
function
DiggController()
    {
        super();
        init();
    }

    private
function
init():void
    {
Story.list(function(e:DiggEvent):void{
stories = new
ArrayCollection(e.data
```


© Copyright IBM Corporation 2008. All rights reserved.

```
as Array);});
    }
}
```

The controller extends the core Flex class `Application`. This is an idiomatic approach to Flex that lets you associate the controller to its view in a simple way, as you will see in the next section. The controller has a collection of stories that is bindable. This will let the collection be bound to UI controls. Once the application loads, it immediately calls the `Story.list` method. It passes an anonymous function or lambda, as a handler to the `Story.list` method. The lambda expression simply dumps the data from the custom event into the stories collection. Now take a look at how this is used in the view.

### The view

Earlier I mentioned that the controller extended the Flex `Application` class. This is the base class of any Flex application. Flex allows you to use MXML (an XML dialect) to declaratively create the UI. The controller is able to leverage this syntax, as shown in Listing 10.

### Listing 10. User interface

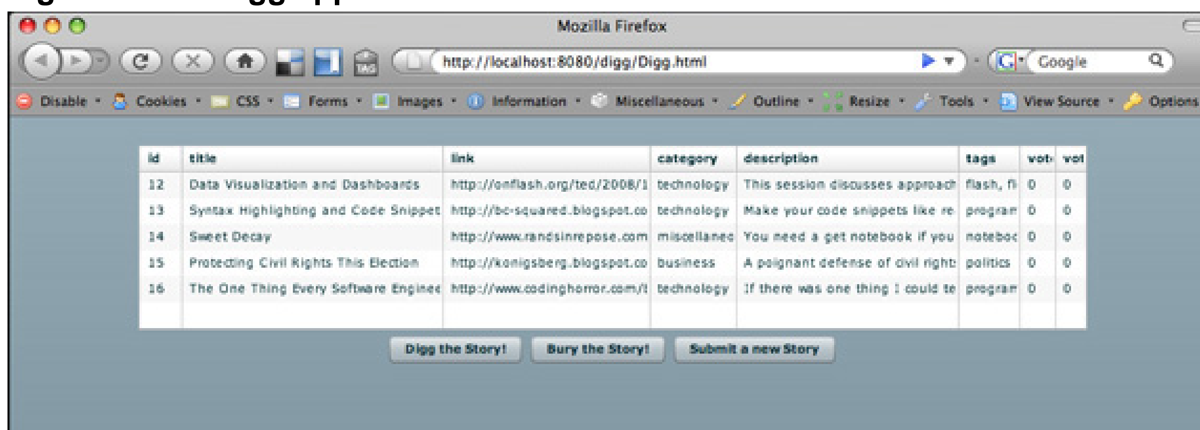
```
<?xml
version="1.0"
encoding="utf-8"?>
<ctrl:DiggController
xmlns:mx="http://www.adobe.com/2006/mxml"
layout="vertical"
xmlns:works="components.*"
xmlns:ctrl="controllers.*">
    <mx:Script>
        <![CDATA[
import
org.developerworks.digg.Story;

private function
digg():void
            {
this.diggStory(results.selectedItem
as Story);
            }
private function
bury():void
            {
this.buryStory(results.selectedItem
as Story);
            }
        ]]>
    </mx:Script>
    <ctrl:states>
        <mx:State
name="SubmitStory">
<mx:AddChild
relativeTo="{buttons}"
position="after">
<works:StoryEditor
successHandler="{this.submissionHandler}"/>
</mx:AddChild>
```


© Copyright IBM Corporation 2008. All rights reserved.

```
</mx:State>
</ctrl:states>
     <mx:DataGrid
id="results"
dataProvider="{stories}"
doubleClickEnabled="true"
doubleClick="openStory(results.selectedItem
as Story)"/>
     <mx:HBox
id="buttons">
<mx:Button
label="Digg the
Story!"
click="digg()"/>
<mx:Button
label="Bury the
Story!"
click="bury()"/>
<mx:Button
label="{this.subBtnLabel}"
click="toggleSubmitStory()"/>
     </mx:HBox>
</ctrl:DiggController>
```

Notice that the root document uses the "ctrl" namespace. This is declared to point to the "controllers" folder, and that is where you put your controller class. Thus everything from the controller class is available inside the UI code. For example, you have a data grid whose `dataProvider` property is set `stories`. This is the stories variable from the controller class. It directly binds to the DataGrid component. Now you can compile the Flex code into a SWF file. You just need a static HTML file for embedding the SWF. Running it will look something like Figure 1.

**Figure 1. The Digg application**




This is the default look and feel of a Flex application. You can easily style the colors using standard CSS like you would use for an HTML application. You can also customize the DataGrid component, specifying the columns, their order, and so on. The "Digg the Story!" and "Bury the Story!" buttons both call functions on the controller class. Also, notice that you wire-up a controller function to the double-click event for the DataGrid. The controller methods all use the model, just as you would expect in an MVC architecture. The final button, "Submit a new Story", uses several key features of Flex. Take a look at how it works.


© Copyright IBM Corporation 2008. All rights reserved.

## Submit a story

As you can see in Listing 10, clicking the "Submit a new Story" button invokes the `toggleSubmitStory` method on the controller class. The code is shown in Listing 11.

### Listing 11. The toggleSubmitStory method

```
public class
DiggController
extends
Application
{

    public
function
toggleSubmitStory():void
    {
        if
(this.currentState
!= 'SubmitStory')
        {
this.currentState
= 'SubmitStory';
subBtnLabel =
'Nevermind';
        }
        else
        {
this.currentState
= '';
subBtnLabel =
'Submit a new
Story';
        }
    }
}
```

This function changes the `currentState` property of the application to `SubmitStory`. Looking back at Listing 10, you can see where this state is defined. States allow you to add or remove components, or to set properties on existing components. In this case you add a new component to the UI. That component happens to be a custom component for submitting a story. It is shown in Listing 12.

### Listing 12. The StoryEditor component

```
<?xml
version="1.0"
encoding="utf-8"?>
<mx:VBox
xmlns:mx="http://www.adobe.com/2006/mxml"
width="100%"
height="100%">
    <mx:Form>
<mx:FormHeading
label="Submit a
```


© Copyright IBM Corporation 2008. All rights reserved.

```
New Story"/>
<mx:FormItem
label="What's the
URL?"
required="true">
<mx:TextInput
id="linkBox"
toolTip="Keep it
Short and Sweet"
text="{story.link}"/>
</mx:FormItem>
<mx:FormItem
label="Give it a
Title"
required="true">
<mx:TextInput
id="titleBox"
text="{story.title}"/>
</mx:FormItem>
<mx:FormItem
label="Pick a
Category"
required="true">
<mx:ComboBox
dataProvider="{Story.CATEGORIES}"
id="categoryBox"
selectedIndex="0"/>
</mx:FormItem>
<mx:FormItem
label="Give a
Short
Description">
<mx:TextArea
height="60"
width="320"
id="descripBox"
text="{story.description}"/>
</mx:FormItem>
<mx:FormItem
label="Tag It">
<mx:TextInput
id="tagBox"
text="{story.tags}"/>
</mx:FormItem>
<mx:Button
label="Submit
It!"
click="submitStory()"/>
    </mx:Form>
    <mx:Binding
source="linkBox.text"
destination="story.link"/>
    <mx:Binding
source="titleBox.text"
destination="story.title"/>
    <mx:Binding
source="categoryBox.selectedItem.data"
destination="story.category"/>
    <mx:Binding
source="descripBox.text"
destination="story.description"/>
    <mx:Binding
source="tagBox.text"
destination="story.tags"/>
</mx:VBox>
```

Listing 12 only shows the UI elements for the custom component. It is just a simple


© Copyright IBM Corporation 2008. All rights reserved.

form, but notice the binding declarations in it. This allows you to directly bind the UI form elements to a `Story` instance, called story. That is declared in a script block, as shown in Listing 13.

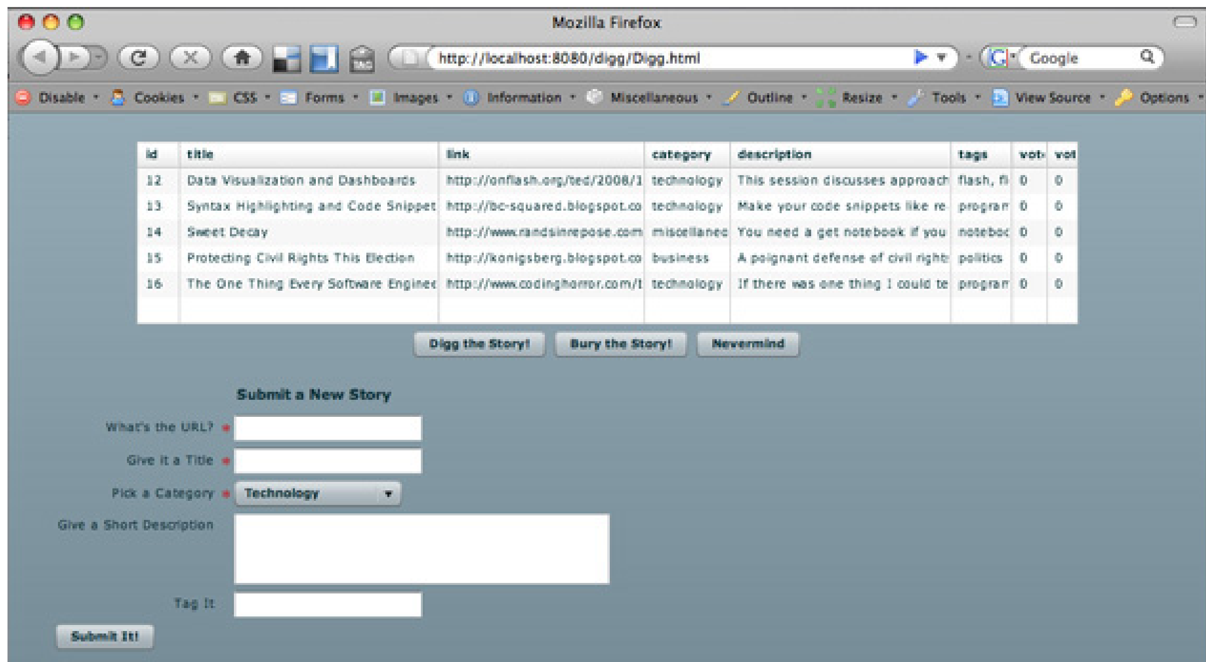
### Listing 13. Script for StoryEditor

```
<?xml
version="1.0"
encoding="utf-8"?>
<mx:VBox
xmlns:mx="http://www.adobe.com/2006/mxml"
width="100%"
height="100%">
    <mx:Script>
        <![CDATA[
import
mx.controls.Alert;
import
org.developerworks.digg.*;

[Bindable]
private var
story:Story = new
Story();
public var
successHandler:Function;

private function
submitStory():void
            {
story.addEventListener(DiggEvent.ON_STORY_SUBMIT_SUCCESS,
successHandler);
story.addEventListener(DiggEvent.ON_STORY_SUBMIT_FAILURE,
errorHandler);
story.save();
// reset
story = new
Story();
            }

private function
errorHandler(e:Event):void
            {
Alert.show("Fail!
: " +
e.toString());
            }
        ]]>
    </mx:Script>
</mx:VBox>
```

The script block acts as the controller code for the component. You could easily put both into a separate controller class for the component, since both styles are common in Flex. Going back to Figure 1, you can click on the "Submit new Story" button, and it will show the custom component, as shown in Figure 2.

### Figure 2. Custom component displayed


© Copyright IBM Corporation 2008. All rights reserved.




You can add a new story using the component. It will call the back-end service, which will return XML for the story. That is, in turn, converted back to a `Story` instance and added to the stories collection that is bound to the DataGrid, making it automatically appear in the UI. With that, the presentation code is complete and hooked up to the back-end service.

## Summary

In this article you have seen how easy it is to create a Web service with Grails. You never had to write any SQL to create a database or to read and write from it. Grails makes it easy to map URLs to Groovy code and to invoke services and create XML for the Web service. This XML is easily consumed by a Flex front end. You have seen how to create a clean MVC architecture on the front end and how to use many sophisticated features of Flex, like E4X, data-binding, states, and custom components.

In Part 2 of this series, explore using a JavaScript-based UI for the service using the Google Web Toolkit.


© Copyright IBM Corporation 2008. All rights reserved.

# Downloads

| Description | Name | Size | Download method |
|---|---|---|---|
| Example Grails application | digg.zip | 504KB | HTTP |
| Example Flex app source | digg-flex-src.zip | 10KB | HTTP |

Information about download methods


© Copyright IBM Corporation 2008. All rights reserved.

# Resources

### Learn

- "Apache Geronimo on Grails" (Michael Galpin, developerWorks, July 2008): The application created in this article can be deployed to any application server. See an example in this developerWorks article.
- "Mastering Grails: Build your first Grails application" (Scott Davis, developerWorks, January 2008): Get an introduction to creating Grails applications.
- "Mastering Grails: GORM: Funny name, serious technology" (Scott Davis, developerWorks, February 2008): See the power of GORM in action.
- "RESTful Web services and their Ajax-based clients" (Shailesh K. Mishra, developerWorks, July 2007): Learn about combining RESTful Web services, like the ones developer here, with Ajax applications.
- "Introducing Project Zero: RESTful applications in an SOA" (Roland Barcia and Steve Ims, developerWorks, January 2008): See how REST based services fit in perfectly in a SOA system.
- Grails manual: Read this for Grails questions.
- "Integrating Flex into Ajax applications" (Brice Mason, developerWorks, July 2008): Read about an example of Flex and Ajax working together.
- ASDocs for Flex: Check out this language reference for the classes in ActionScript and Flex.
- "Mastering Grails: Changing the view with Groovy Server Pages" (Scott Davis, developerWorks, March 2008): Don't like the UI of this app? Learn all the ways to change it.
- "Practically Groovy: Reduce code noise with Groovy" (Scott Hickey, developerWorks, September 2006): Are you a fan of the succinctness that Groovy provides? Learn all about its concise syntax.
- "Practically Groovy: Mark it up with Groovy Builders" (Andrew Glover, developerWorks, April 2005): Groovy is another promising language that compiles to Java bytecode. Read about creating XML with it in this developerWorks article.
- Grails.org: The best place for Grails information is the project's site.
- The Grails manual: Every Grails developer's best friend.
- A series of rough benchmarks: See how Grails has significant advantages over Rails by advantages being on top of Java, Hibernate, and Spring.


© Copyright IBM Corporation 2008. All rights reserved.

- "Understand Geronimo's deployment architecture" (Hemapani Srinath Perera, developerWorks, August 2005): Deploying an application on Geronimo is easy, but there is a lot that goes on. Learn all about what it takes to deploy an application on Geronimo.
- "Remotely deploy Web applications on Apache Geronimo" (Michael Galpin, developerWorks, May 2006): Find out how to deploy your Grails applications to remote instances of Geronimo.
- "Invoke dynamic languages dynamically, Part 1: Introducing the Java scripting API" (Tom McQueeney, developerWorks, September 2007): Learn about other alternative languages running on the JVM in this developerWorks article.
- "Build an Ajax-enabled application using the Google Web Toolkit and Apache Geronimo" (Michael Galpin, developerWorks, May 2007): See how Geronimo can be used with the Google Web Toolkit.
- Visit Digg.com.
- developerWorks Web development zone: Expand your site development skills with articles and tutorials that specialize in Web technologies.
- developerWorks Open source zone: Visit us for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM products.
- developerWorks podcasts: Listen to interesting interviews and discussions for software developers.
- developerWorks technical events and webcasts: Stay current with developerWorks technical events and webcasts.

### Get products and technologies

- Grails: This article uses Grails version 1.0.3
- Get the Flex 3 SDK.
- Get Adobe Flash Player Version 10 or later.
- Java SDK: This article uses Java SE 1.6_05.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.
- Download IBM product evaluation versions, and get your hands on application development tools and middleware products from DB2, Lotus, Rational, Tivoli, and WebSphere.

### Discuss

- Participate in developerWorks blogs and get involved in the developerWorks


© Copyright IBM Corporation 2008. All rights reserved.

community.

## About the author

Michael Galpin
Michael Galpin has been developing Web applications since the late '90s. He holds a degree in mathematics from the California Institute of Technology and is an architect at eBay in San Jose, CA.

## Trademarks

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.


© Copyright IBM Corporation 2008. All rights reserved.