SproutCore and ERRest

or Todos 06 - Building with ERRest

This tutorial is designed to supplement the "Todos" tutorial found on the SproutCore wiki. In the Todos tutorial, chapter six contains many examples from other frameworks on how to set up a Rest server to use as the backend for SproutCore. This tutorial will show you how to complete the tutorial using WebObjects and Project Wonder's ERRest framework. Along the way we'll try to hit on a couple of other concepts that may augment your general WebObjects knowledge.

Project Creation

To get started create a new Wonder application and names it TodosRestServer. In the proceeding steps please make the following choices:

1. For the package names you can specify something that makes sense for you. In the example we use org.objectstyle.app and org.objectstyle.components.
2. During the add referenced projects phase be sure to add the ERRest and the JavaMemoryAdaptor frameworks.

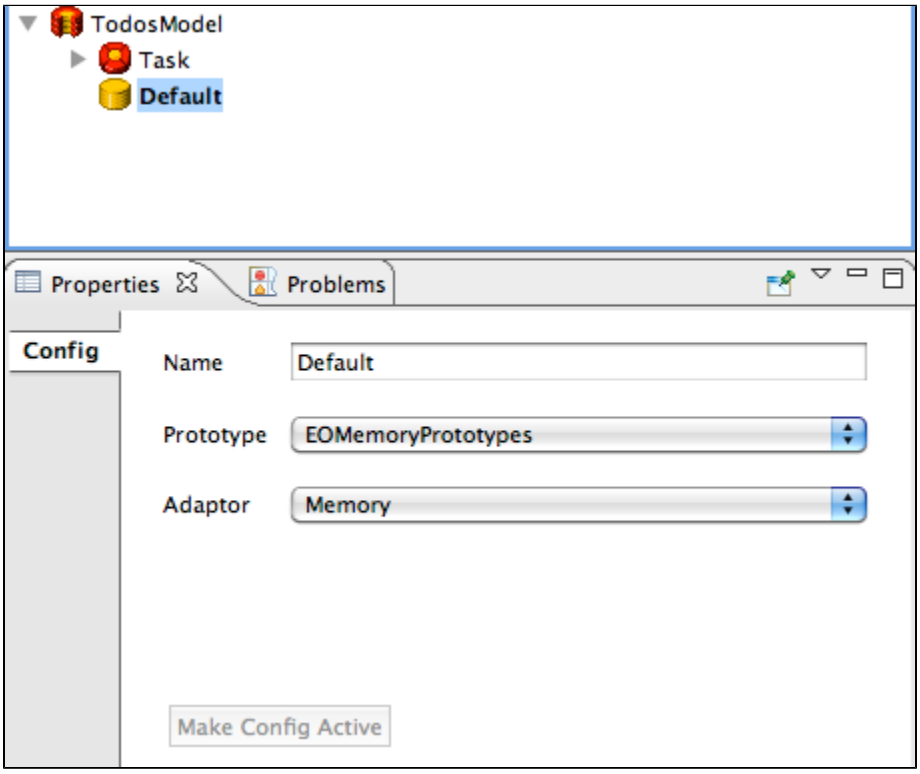
For more information on creating a Wonder Application please visit wiki.wocommunity.org

Creating the Model

The Todos tutorial has one basic type of data that we need to model. Please add a new model to the resources folder of your project. Name the model "TodosModel" and then open the model in Eclipse's Entity Modeler Perspective. Add an entity whose name and table name is "Task" and whose class name is org.objectstyle.model.Task. Then add two attributes to the Task entity:

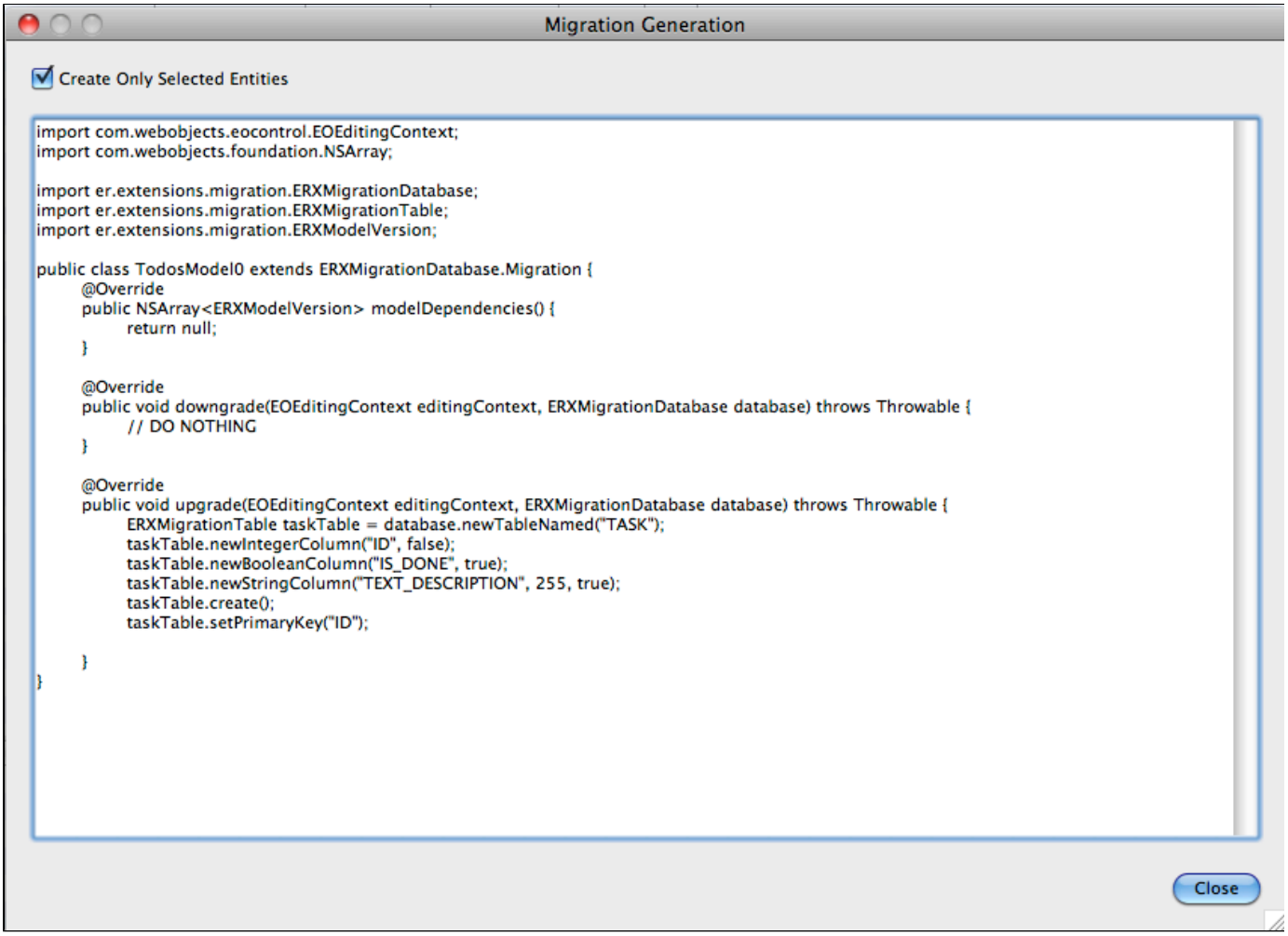
1. ***isDone*** of type Boolean
2. ***textDescription*** of type varchar255

Since we are using the model in conjunction with a tutorial there is no need to really persist the data. So we are going to use the JavaMemoryAdaptor framework to create the database. Open the model's configuration and tell it to use EOMemoryPrototypes and set the adaptor to Memory.



Create the Migrations File

Right click the Task entity and choose the option "Generate Migration". Copy the contents of the Modal Window.



Create a new package in your source folder and name it org.objectstyle.migrations. After creating the package select it and paste the migration file into it. Open the migration file (TodosModel0.java) and edit it to implement "IERXPostMigration" i.e.

```
public class TodosModel0 extends ERXMigrationDatabase.Migration implements
IERXPostMigration
```

Then add the following method to the class.

```
public void postUpgrade(EOEditingContext editingContext, EOModel model)
throws Throwable {
 String[] descriptions = new String[]{
  "Build my first SproutCore app",
  "Build a really awesome SproutCore app",
  "Next, the world!"
 };
 for(String description : descriptions) {
  Task task = Task.createTask(editingContext);
  task.setTextDescription(description);
  task.setIsDone(false);
 }
}
```

Open your properties file and uncomment the following two lines:

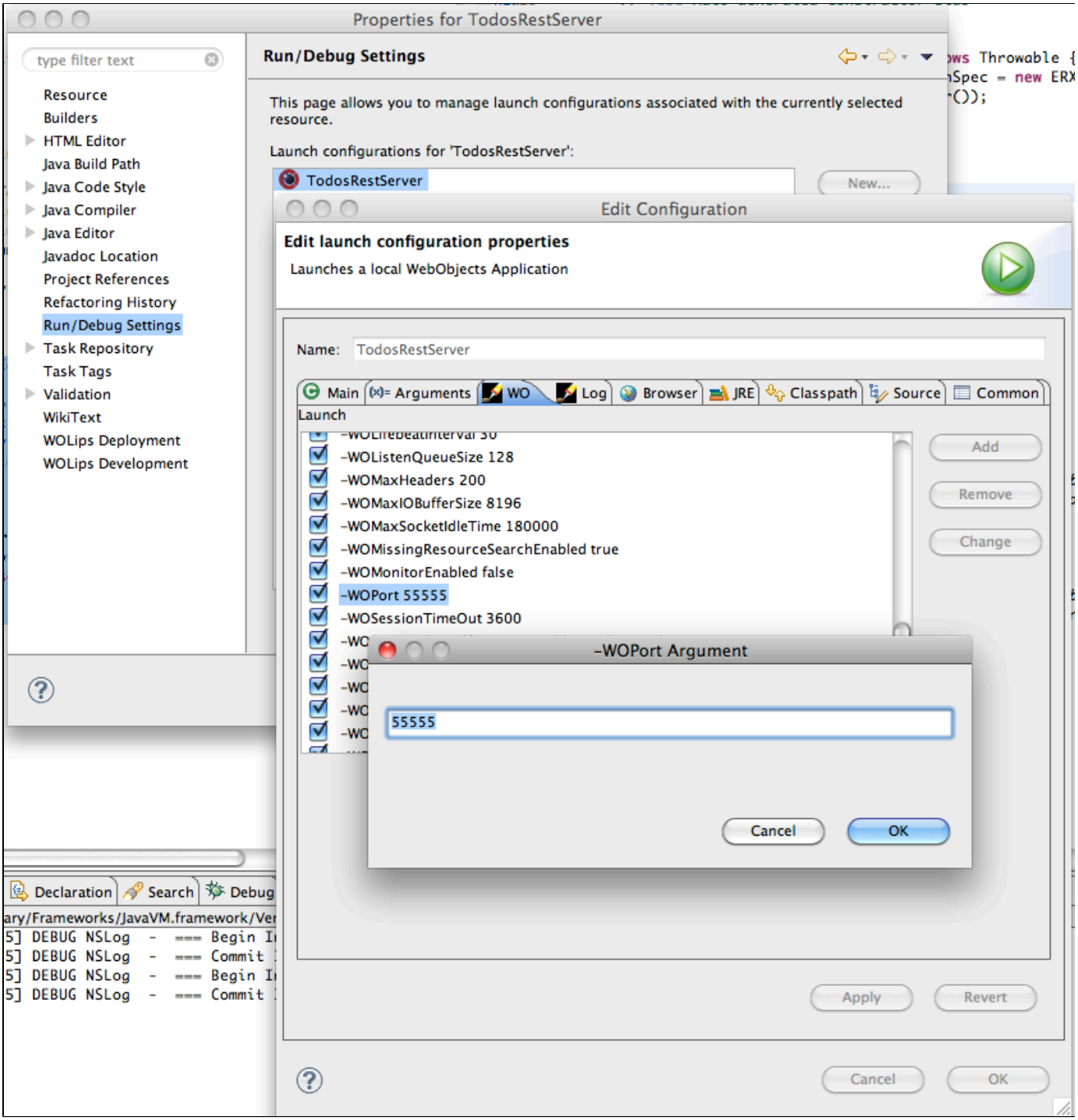
```
#Migrations
er.migration.migrateAtStartup=true
er.migration.createTablesIfNecessary=true
```

Now every time you start the application you will get the default dataset recommended by the SproutCore tutorial.

For more information on working with Migrations please visit Project Wonder's JavaDoc

Starting the Server on a Specified Port

SproutCore has a proxy mechanism that maps a request to a domain name/ip address and port number. By default when we debug our WebObjects applications in Eclipse it will choose a random port to run the application on. This will become annoying because we will have to keep telling our SproutCore app the new port number every time we restart the application. In addition you'll have to restart your SproutCore server for the change to take effect. To avoid the hassle we are going to tell Eclipse which port number we want to run our application on. Right click your application in eclipse's WO Explorer perspective and choose the "Properties" option. Navigate to the section "Run/Debug Settings" select the application, choose edit, select the "WO" tab and then find the property "WOPort". Change the property to use a port that makes sense for you. In the example it is set to "55555"



Generating the Model Classes.

Open the EOGenerator file (TodosModel.eogen) that was created in the Resources folder along side your EOModel. In the "Templates" section set the template folder to the location of your Wonder Entity Templates. Then specify "_WonderEntity.java" and "WonderEntity.java" as the values for the "Template" and "Subclass Template" fields respectively. Now Right Click the "TodosModel.eogen" file and choose "EOGenerate".

For more information about EOGenerator please visit: [EOGenerator on Object Style Wiki|]

Updating Application.java

Open Application.java in Eclipse. Update the Application's constructor to use DirectAction as the default request handler key and then add the ERXRouteRequestHandler for the Task entity i.e.

```
public Application() {
 ERXApplication.log.info("Welcome to " + name() + " !");
 /* ** put your initialization code in here ** */
 setDefaultRequestHandler(requestHandlerForKey(directActionRequestHandlerK
ey()));
 ERXRouteRequestHandler routeRequestHandler = new
ERXRouteRequestHandler(ERXRouteRequestHandler.WO);
 routeRequestHandler.addDefaultRoutes(Task.ENTITY_NAME);
 ERXRouteRequestHandler.register(routeRequestHandler);
}
```

ERRest utilizes the concept of convention over configuration. So when we tell the ERXRouteRequestHandler to **addDefaultRoutes** we are notifying that the request should use the convention defined methods to use in conjunction with the request's HTTP verb i.e. a GET request will map to indexAction and a POST request will map to createAction in the entity's rest controller that we are going to create in the next section.

Creating the Task Entity's Rest Controller

Create a new package to place your controller in i.e. "org.objectstyle.controllers". In that package create a new class named ***TaskController*** and have it extend ERXRouteController. It is important to note that the name of the controller has to be the entity's name + "Controller".

ERRest adds a lot of magic for you that makes setting up your request routes really easy. For the tutorial we need to create a route that will fetch the stored tasks. To accomplish this we first need to create the following methods:

```
public WOActionResults indexAction() throws Throwable {
 ERXRestFetchSpecification<Task> fetchSpec = new
ERXRestFetchSpecification<Task>(Task.ENTITY_NAME, null, null,
queryFilter(), null, 10);
 return response(fetchSpec, showFilter());
}


public ERXKeyFilter queryFilter() {
 return ERXKeyFilter.filterWithAllRecursive();
}


public ERXKeyFilter showFilter() {
 ERXKeyFilter filter = ERXKeyFilter.filterWithAttributes();
 filter.addMap(Task.TEXT_DESCRIPTION, new ERXKey<String>("description"));
 return filter;
}
```

When an HTTP GET comes into the server on the Task entity's route i.e.

GET /cgi-bin/WebObjects/TodosRestServer.woa/ra/Task

the ERRest framework will know to look at **indexAction** for the response because that is the method defined by the convention. In addition, ERRest allows you to configure which attributes should be exposed for such requests. Since we want to return all the entity's attributes, including the id, we need to specify that the **queryFilter** uses the method **filterWithAllRecursive**. When we defined the Task entity in the model we called our text field "textDescription" instead of "description" to avoid overriding Enterprise Object's description method. However, our SproutCore application expects an attribute named "description". Fortunately, this is easy to fix. In the **showFilter** method we add a filter that maps the attribute name to the format SproutCore anticipates.

Back in SproutCore

First we need to update the *BuildFile* file to include the proxy route to our Rest server i.e.

```
proxy "/cgi-bin/WebObjects/TodosRestServer.woa/ra/", :to =>
"localhost:55555"
```

If you take a look closely at our route you might have noticed we have appended the route with ".sc". ERRest has a concept called formatters that returns data in a format that the request expects i.e. xml, json, plist etc... In fact, there is even a special formatter for SproutCore that uses the extension ".sc". The SproutCore formatter includes a mapping that translates the "id" field to "guid" which is what SproutCore expects.

Now we need to add the route to our fetch function in *data_sources/task.js*.

```
 fetch: function(store, query) {
        if (query === Todos.TASKS_QUERY) {
  var url = '/cgi-bin/WebObjects/TodosRestServer.woa/ra/Task.sc';
  SC.Request.getUrl(url).json().notify(this, 'didFetchTasks', store,
query).send();
  return YES;
 }
 return NO;
},
```

One more important change we need to make is more of a gotcha. In the Todos.TaskDataSource object we need to change the function didFetchTasks. On the line:

```
store.loadRecords(Todos.Task, response.get('body').content);
```

change **content** to **records**. So it looks like:

```
store.loadRecords(Todos.Task, response.get('body').records);
```

```
Note: A patch has been submitted to change the wrapper from records to
content.  So in future versions of ERRest this step may become unnecessary.
```

The second gotcha. ERRest has a bug where it returns attributes in underscore format instead of camel case (which SproutCore expects). To fix this bug open the file ERXRestFormat.java and update the static variable SPROUTCORE to the following:

```
public static ERXRestFormat SPROUTCORE =
ERXRestFormat.registerFormatNamed(new ERXJSONRestParser(), new
ERXSproutCoreRestWriter(),
 new ERXRestFormatDelegate("guid", "type", "nil", true, false), "sc",
"application/sc");
```

This bug has been submitted and may be fixed in the version of Wonder you are using.

Creating New Records

Please return to the Task entity's controller, TaskController.java. Following the ERRest convention we now need to create a method to use in conjunction with the HTTP POST verb:

```
public WOActionResults createAction() {
 Task task = create(updateFilter());
 editingContext().saveChanges();
 return response(task, showFilter());
}
```

The one interesting part to note is in the fetch method (**indexAction)** we had a filter (**showFilter)** that mapped "textDescription" to "description". Because we now need to convert "description" to "textDescription" we need to create a filter that works in reverse i.e.

```
public static ERXKeyFilter updateFilter() {
 ERXKeyFilter filter = ERXKeyFilter.filterWithAttributes();
 filter.addMap(new ERXKey<String>("description"), Task.TEXT_DESCRIPTION);
 return filter;
}
```

Now restart your application and then refresh your SproutCore application's browser. You should now be able to use the "Add Task" button. To test if your new tasks have been created just refresh the browser again.

That completes the WebObjects supplement to the Todos tutorial.

Going Beyond the Tutorial - Updating and Deleting Records

The first step in managing updates and deletes is to create the corresponding actions in our TaskController.java file. Open TaskController.java in Eclipse and add the following methods.

```
 public WOActionResults updateAction(@PathParam("task") Task task) throws
Throwable {
 update(task, updateFilter());
 editingContext().saveChanges();
 return response(task, showFilter());
}


public WOActionResults destroyAction(@PathParam("task") Task task) throws
Throwable {
 task.delete();
 editingContext().saveChanges();
 return response(task, showFilter());
}
```

When using the default routes the convention for updating and deleting records is to create methods named "updateAction" and "deleteAction" respectively. Note, ERRest supports JSR 311 in the example we demonstrate this capacity by passing in a parameter to both methods that will automagically look up and find the proper object to update/delete based upon the information provided in the request. This is basically a short hand way of writing the following method:

```
public WOActionResults updateAction() {
 Task task = (Task)routeObjectForKey("task");
 update(task, updateFilter());
 editingContext().saveChanges();
 return response(task, showFilter());
}
```

Changes back in SproutCore

The todos tutorial expects that the backend will always return an object to a PUT request. However, ERRest always returns an array. So, on the client we need to update our methods to include the function **objectAt** so that we get the object SproutCore anticipates. For reference please see the completed Task Data Source File.

```
//
============================================================================

// Project:   Todos.TaskDataSource

// Copyright: ©2010 My Company, Inc.

//
============================================================================

/*globals Todos */
/** @class
(Document Your Data Source Here)
@extends SC.DataSource
*/
```

```
sc_require('models/task');

Todos.TASKS_QUERY = SC.Query.local(Todos.Task, {
 orderBy: 'isDone,description'
});

Todos.TaskDataSource = SC.DataSource.extend(
/** @scope Todos.TaskDataSource.prototype */
{


fetch: function(store, query) {
        if (query === Todos.TASKS_QUERY) {
  var url = '/cgi-bin/WebObjects/TodosRestServer.woa/ra/Task.sc';
  SC.Request.getUrl(url).json() .notify(this, 'didFetchTasks', store,
query).send();
  return YES;
 }
        return NO;
},

didFetchTasks: function(response, store, query) {
        if (SC.ok(response)) {
            store.loadRecords(Todos.Task, response.get('body').content);
            store.dataSourceDidFetchQuery(query);
        } else {
            store.dataSourceDidErrorQuery(query, response);
        }
},


retrieveRecord: function(store, storeKey) {
        if (SC.kindOf(store.recordTypeFor(storeKey), Todos.Task)) {
  var url = '/cgi-bin/WebObjects/TodosRestServer.woa/ra/Task' +
store.idFor(storeKey) + '.sc';
  SC.Request.getUrl(url).json().notify(this, 'didRetrieveTask', store,
storeKey).send();
  return YES;
        } else return NO;
},


didRetrieveTask: function(response, store, storeKey) {
 if(SC.ok(response)) {
  var dataHash = response.get('body').content.objectAt(0);
  store.dataSourceDidComplete(storeKey, dataHash);
 } else store.dataSourceDidError(storeKey, response);
},


createRecord: function(store, storeKey) {
```

```
 if(SC.kindOf(store.recordTypeFor(storeKey), Todos.Task)) {
  var url = '/cgi-bin/WebObjects/TodosRestServer.woa/ra/Task.sc';
  SC.Request.postUrl(url).json().notify(this, this.didCreateTask, store,
storeKey).send(store.readDataHash(storeKey));
  return YES;
 }
 return NO; // return YES if you handled the storeKey
},


didCreateTask: function(response, store, storeKey) {
 if(SC.ok(response)) {
  var data = response.get('body');
  if(data) data = data.content.objectAt(0);
  store.dataSourceDidComplete(storeKey, null, data.guid); // update url
 } else store.dataSourceDidError(storeKey, response);
},

updateRecord: function(store, storeKey) {
 if(SC.kindOf(store.recordTypeFor(storeKey), Todos.Task)) {
  var url = '/cgi-bin/WebObjects/TodosRestServer.woa/ra/Task/' +
store.idFor(storeKey) + '.sc';
  SC.Request.putUrl(url).json().notify(this, this.didUpdateTask, store,
storeKey).send(store.readDataHash(storeKey));
  return YES;
 } else return NO; // return YES if you handled the storeKey
},


didUpdateTask: function(response, store, storeKey) {
 if(SC.ok(response)) {
  var data = response.get('body');
  if(data) data = data.content.objectAt(0);
  store.dataSourceDidComplete(storeKey, data);
 } else store.dataSourceDidError(storeKey);
 },


destroyRecord: function(store, storeKey) {
 if(SC.kindOf(store.recordTypeFor(storeKey), Todos.Task)) {
  var url = '/cgi-bin/WebObjects/TodosRestServer.woa/ra/Task/' +
store.idFor(storeKey) + '.sc';
  SC.Request.deleteUrl(url).json().notify(this, this.didDestroyTask, store,
storeKey).send();
  return YES;
 }
},


didDestroyTask: function(response, store, storeKey) {
 if(SC.ok(response)) {
  store.dataSourceDidDestroy(storeKey);
 } else store.dataSourceDidError(response);
```

```
  }
  });
```