1. Overview

The extension library includes four types of REST services. You choose the type of REST service that meets your needs. This section introduces the different types of REST services. It also compares their capabilities to help you decide which type most applies to your needs.

The four types of REST services are as follows:

- The **Domino Data Service** is a built-in service that represents Domino data in JSON format. Once you install the extension library, you can begin using the data service without creating an XPage or adding any Java code to the Domino server.
- The **XPages REST Services Control** is a component you add to an XPage. Often you add other controls to the page that directly reference the REST Services control. For example, a Dojo Data Grid can reference a REST Services control on the same page. Once you save the page, the service also becomes available outside the page through a well-known URL. Depending on how you configure the control, the service represents data in JSON or XML format.
- A **Custom Database Servlet** is a Java class you add to a database design. The servlet handles incoming HTTP requests usually by delegating to one of the REST service classes in the extension library. This type of REST service requires detailed knowledge of the Java programming language, but you have complete control over the definition of the service.
- A **Custom Wink Servlet** is the most advanced type of REST service. You use the open source Apache Wink project to define your service. Your servlet is contained in a plug-in that is deployed directly to Domino's OSGi framework. This means your service is not tied to a single database. It can access any data you choose and represent it in any format you choose.

See Table 1.1 for a side-by-side comparison of the four types of REST services. See sections 2 through 5 for detailed descriptions of each type.

Table 1.1 Comparison of extension library REST services

	Component Type	**Container**	**Data Model**	**Data Format**	**Deployment**
Domino Data Service	OSGi plug-in	OSGi framework	Databases, views, documents, etc.	JSON	Installed with extension library
XPages REST Services Control	XPages custom control	XPage	Databases, views, documents, etc.	JSON or XML	You add a control to an XPage
Custom Database Servlet	Java class	Database	Databases, views, documents, etc.	JSON or XML	You add a Java class to a database design
Custom Wink Servlet	OSGi plug-in	OSGi framework	Good for a higher level of abstraction (e.g. calendar objects)	Any format you choose	You create the OSGi plug-in and add it to the Domino OSGi framework.

2. Domino Data Service

The Domino Data Service is now included in the XPages Extension Library. It represents the Domino object model (databases, views, view entries and documents) in JSON format. The data service lets you send HTTP requests to:

- Read the list of databases on the server.
- Read the list of views and folders in a database.
- Read the design of a view or folder.
- Read the entries in a view or folder.
- Create, read, update and delete documents.

The data service requires Domino 8.5.3 CD5 (or greater).

The following sections help you get started with the data service. For more information please see the Domino Data Service documentation included with the extension library.

2.1 Enabling the Data Service

2.1.1 Enabling the Data Service on a Server

After you install the extension library, the data service is loaded whenever the Domino HTTP task is started. However, an administrator typically doesn't want the data service to handle requests on every Domino server. You need to deliberately enable the data service in the appropriate Internet Site document.

To enable the data service:

1. Use a Notes client to open the server's public address book.
2. In the Domino Directory navigator, select Configuration – Web – Internet Sites.
3. Open the Internet Site document for your server.
4. Click the Edit Web Site action.
5. Select the Configuration tab.
6. At the bottom of the form, look for a section labeled Domino Access Services.
7. In the Enabled services field, add the Data keyword:

Domino Access Services

The following setting is a place holder for services provided by an external plug-in. See the Lotus Notes and Domino wiki for more information.

Enabled services: Data

8. Save your changes and restart the HTTP task.

NOTE: The above instructions assume you are using Internet Sites. If you are not using Internet Sites, you can enable the data service in the server document. See the Domino Data Service documentation for more information.

2.1.2 Enabling the Data Service for a Database

By default, the data service does not have access to each database. Just as you need to enable the data service for a server, you also need to deliberately enable the data service for a database.

> IMPORTANT: Before following the steps below, make sure you are using Notes 8.5.3 CD5 (or later). Also add the following line to the client's Notes.ini file:
>
> EnableDAS=1
>
> You need to make the above change only once and then restart your Notes client.

To enable the data service for a database:

1. Use the Notes client to open the database.
2. Select File – Application – Properties.
3. Click on the Advanced tab.
4. At the bottom of the properties box, look for a field labeled "Allow Domino Data Service". Change the field to "Views and documents". See Figure 2.1.
5. Close the Database properties box.

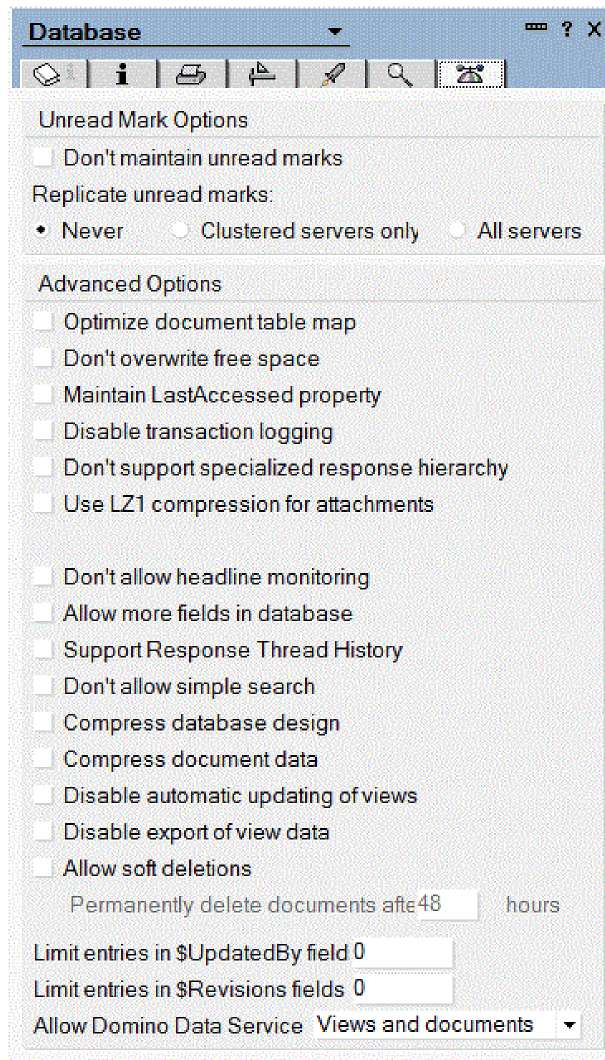


Figure 2.1 Database Properties Box

2.1.3 Enabling the Data Service for a View or Folder

By default, the data service does not have access to each view in a database. You need to deliberately enable the data service for a view or folder.

> IMPORTANT: Before following the steps below, make sure you are using Notes 8.5.3 CD5 (or later). Also add the following line to the client's Notes.ini file:
>
> EnableDAS=1
>
> You need to make the above change only once and then restart your Notes client.

To enable the data service for a view or folder:

1. Use Domino Designer to open the database.
2. Open the view or folder.
3. Press Alt-Enter to display the View properties box.

4. Select the Advanced tab.
5. Look for the checkbox labeled “Allow Domino Data Service operations”. Check the box as shown in Figure 2.2.
6. Close the View properties box.

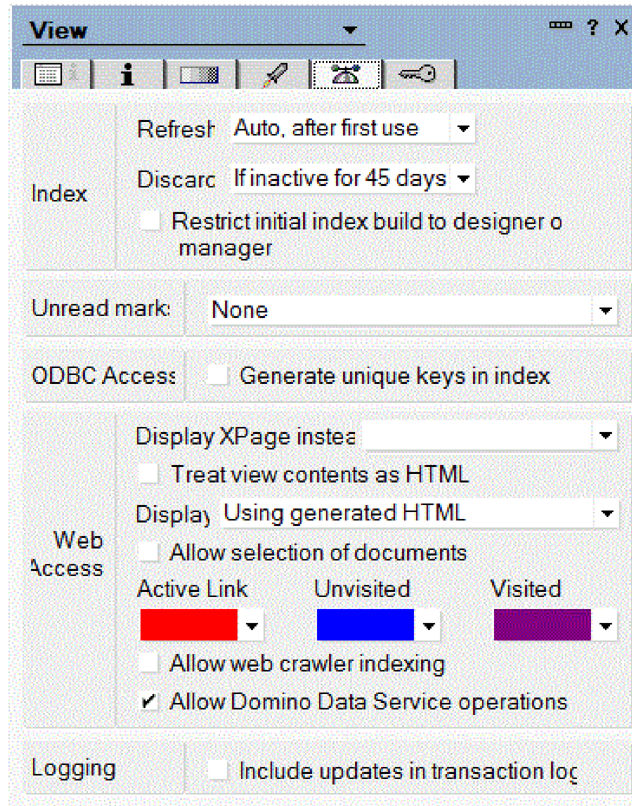


Figure 2.2 View Properties Box

2.2 Database Collection Resource

To get the list of databases on a server you send an HTTP GET request to the database collection resource URI:

http://{host}/api/data

The data service returns a response in JSON format like this:

```
[
  {
      "@title":"Administration Requests",
      "@filepath":"admin4.nsf",
      "@replicaid":"852573910361A2F4",
      "@template":"StdR4AdminRequests",
      "@href":"http:\/\/xyz.com:80\/admin4.nsf\/api\/data\/collections"
  },
  {
      "@title":"Java AgentRunner",
      "@filepath":"AgentRunner.nsf",
      "@replicaid":"8525671400725208",
```

```
        "@template":"",
        "@href":"http:\/\/xyz.com:80\/AgentRunner.nsf\/api\/data\/collections"
    },

    ...

    {
        "@title":"XPages Extension Library Demo",
        "@filepath":"XPagesExt.nsf",
        "@replicaid":"8525786700781FD3",
        "@template":"",
        "@href":"http:\/\/xyz.com:80\/XPagesExt.nsf\/api\/data\/collections"
    }
]
```


In other words, the response is an array of objects. Each object represents a database. For example:

- the @title property corresponds to the title of the database
- the @filepath property corresponds to the location of the database (relative to the Domino data directory)
- the @href property is the URI for the database's view collection resource

2.3 View Collection Resource

To get the list of views and folders in a database, you send an HTTP GET request to the view collection resource URI:

http://{host}/{database}/api/data/collections

The data service returns a response in JSON format like this:


```
[
    {
        "@title":"",
        "@folder":false,
        "@private":false,
        "@modified":"2011-04-29T13:02:19Z",
        "@unid":"45DF1A4D8B8C1027852578650065F565",

"@href":"http:\/\/xyz.com:80\/xpagesext.nsf\/api\/data\/collections\/unid\/45DF1A4D8B8C102785
2578650065F565"
    },
    {
        "@title":"TestCalendarOutline",
        "@folder":false,
        "@private":false,
        "@modified":"2011-04-29T13:02:20Z",
        "@unid":"F598C2D31E4E12F68525786500660B7E",

"@href":"http:\/\/xyz.com:80\/xpagesext.nsf\/api\/data\/collections\/unid\/F598C2D31E4E12F685
25786500660B7E"
    },

    ...

    {
        "@title":"AllContacts",
        "@folder":false,
        "@private":false,
        "@modified":"2011-04-29T13:02:20Z",
        "@unid":"CD40A953ABDE036A8525786500660C27",

"@href":"http:\/\/xyz.com:80\/xpagesext.nsf\/api\/data\/collections\/unid\/CD40A953ABDE036A85
25786500660C27"
    },
```

```
  {
      "@title":"Folder2",
      "@folder":true,
      "@private":false,
      "@modified":"2011-04-29T13:02:20Z",
      "@unid":"7B53FDDEE90C5E918525786500660E03",

"@href":"http:\/\/xyz.com:80\/xpagesext.nsf\/api\/data\/collections\/unid\/7B53FDDEE90C5E9185
25786500660E03"
  },
  {
      "@title":"Folder1",
      "@folder":true,
      "@private":false,
      "@modified":"2011-04-29T13:02:20Z",
      "@unid":"A01D1A3F6BEAE5588525786500660E22",

"@href":"http:\/\/xyz.com:80\/xpagesext.nsf\/api\/data\/collections\/unid\/A01D1A3F6BEAE55885
25786500660E22"
  }
]
```


The response is an array of objects. Each object represents either a view or a folder. For example:

- the @title property corresponds to the title of the view (or folder)
- the @folder property is true only for folders
- the @unid property correspond to the universal identifier for the view (or folder).

NOTE: The example above lists the views and folders in the XPages Extension Library sample database. This database is included with the extension library. The remaining examples are from the same database, but the data service works for other databases including mail files, team rooms, and so on.

2.4 View Design Resource

To read the design of a view or folder, you send an HTTP GET request to the view design resource URI:

http://{host}/{database}/api/data/collections/unid/{unid}/design

For example, the following URI corresponds to the AllContacts view in the extension library sample database:

http://xyz.com/xpagesext.nsf/api/data/collections/unid/CD40A953ABDE036A852578650066
0C27/design

The data service returns a response in JSON format like this:


```
[
  {
      "@columnNumber":1,
      "@name":"$0",
      "@title":"#",
      "@width":4,
      "@alignment":0,
      "@hidden":false,
      "@response":false,
      "@twistie":false,
      "@field":false,
      "@category":false
  },
  {
```

```
        "@columnNumber":2,
        "@name":"Id",
        "@title":"Id",
        "@width":10,
        "@alignment":0,
        "@hidden":false,
        "@response":false,
        "@twistie":false,
        "@field":true,
        "@category":false
    },
    {
        "@columnNumber":3,
        "@name":"FirstName",
        "@title":"First Name",
        "@width":10,
        "@alignment":0,
        "@hidden":false,
        "@response":false,
        "@twistie":false,
        "@field":true,
        "@category":false
    },
    ...
]
```

The response is an array of objects where each object represents a view column. For example:

- the @name property is the programmatic column name
- the @title property is the title displayed in the column header
- @width is the width of the column
- @field is true only if the column refers to a field on the corresponding document. It is false for computed values.

NOTE: This guide uses the UNID format of the view design resource URI. You may find it easier to use the name format instead. For example, you can send a GET request to:

http://xyz.com/xpagesext.nsf/api/data/collections/name/AllContacts/design

The response is the same as shown above.

2.5 View Entry Collection Resource

To get a list of entries in a view or folder, you send an HTTP GET request to the view entry collection resource URI:

http://{host}/{database}/api/data/collections/unid/{unid}

For example, the following URI corresponds to the AllContacts view in the extension library sample database:

http://xyz.com/xpagesext.nsf/api/data/collections/unid/CD40A953ABDE036A8525786500660C27

The data service returns a response in JSON format like this:

```
[
  {
"@href":"http:\/\/xyz.com:80\/xpagesext.nsf\/api\/data\/collections\/unid\/CD40A953ABDE036A85
25786500660C27\/unid\/AAE5C9A07AF9C1A7852578760048C0D6",
      "@link":
      {
```

```
          "rel":"document",

    "href":"http:\/\/xyz.com:80\/xpagesext.nsf\/api\/data\/documents\/unid\/AAE5C9A07AF9C1A785257
    8760048C0D6"
        },
        "@entryid":"1-AAE5C9A07AF9C1A7852578760048C0D6",
        "@unid":"AAE5C9A07AF9C1A7852578760048C0D6",
        "@noteid":"9AA",
        "@position":"1",
        "@read":true,
        "@siblings":201,
        "@form":"Contact",
        "Id":"CN=Adela Rojas\/O=renovations",
        "FirstName":"Adela",
        "LastName":"Rojas",
        "EMail":"adela_rojas@renovations.com",
        "City":"Paterson",
        "State":"NJ",
        "created":"2011-04-18T13:14:39Z",
        "$10":"Adela Rojas"
    },
    {

    "@href":"http:\/\/xyz.com:80\/xpagesext.nsf\/api\/data\/collections\/unid\/CD40A953ABDE036A85
    25786500660C27\/unid\/994A0071DF739926852578760048C169",
        "@link":
        {
          "rel":"document",

    "href":"http:\/\/xyz.com:80\/xpagesext.nsf\/api\/data\/documents\/unid\/994A0071DF73992685257
    8760048C169"
        },
        "@entryid":"2-994A0071DF739926852578760048C169",
        "@unid":"994A0071DF739926852578760048C169",
        "@noteid":"BF6",
        "@position":"2",
        "@read":true,
        "@siblings":201,
        "@form":"Contact",
        "Id":"CN=Adrienne Forbes\/O=renovations",
        "FirstName":"Adrienne",
        "LastName":"Forbes",
        "EMail":"adrienne_forbes@renovations.com",
        "City":"Newport News",
        "State":"VA",
        "created":"2011-04-18T13:14:41Z",
        "$10":"Adrienne Forbes"
    },

    ...

]
```


The response is an array of objects where each object represents a view entry. For example:

- the @href property is the URI for this view entry
- the @link property represents the URI for the corresponding document
- the @unid property is the universal identifier for the document
- Id, FirstName, LastName, etc. correspond to column values. The exact set of properties will depend on the specific view design.

NOTE: This guide uses the UNID format of the view entry collection resource URI. You may find it easier to use the name format instead. For example, you can send a GET request to:

http://xyz.com/xpagesext.nsf/api/data/collections/name/AllContacts

The response is the same as shown above.

When you get view entries, the response also includes a Content-Range header indicating how many entries are included. For example:

Content-Range: items 0-9/201

This header indicates the data service returned entries 0 through 9 from a total of 201 entries. To get the next 10 entries, you send a GET request with additional URL parameters:

http://xyz.com/xpagesext.nsf/api/data/collections/unid/CD40A953ABDE036A8525786500660C27?ps=10&page=1

In this example, the ps parameter specifies the page size and the page parameter specifies which page to get. In this case, you want to get the second page (page numbers are zero-based). The data service returns the second page of data and a new Content-Range header like this:

Content-Range: items 10-19/201

2.6 Document Resource

2.6.1 Reading a Document

To read a single document, you send an HTTP GET request to the document resource URI:

http://{host}/{database}/api/data/documents/unid/{unid}

NOTE: You can get a document resource URI from a view entry collection resource (see section 2.5).

The data service returns a response in JSON format like this:


```
{
"@href":"http:\/\/xyz.com:80\/xpagesext.nsf\/api\/data\/documents\/unid\/AAE5C9A07AF9C1A7852578760048C0D6",
    "@unid":"AAE5C9A07AF9C1A7852578760048C0D6",
    "@noteid":"9AA",
    "@created":"2011-04-18T13:14:39Z",
    "@modified":"2011-04-18T13:14:40Z",
    "@form":"Contact",
    "Id":"CN=Adela Rojas\/O=renovations",
    "FirstName":"Adela",
    "LastName":"Rojas",
    "City":"Paterson",
    "State":"NJ",
    "EMail":"adela_rojas@renovations.com"
}
```


The response is a single object representing the document. For example:

- @href is the document resource URI
- @unid is the universal identifier for the document
- @form is the name of the form used to create the document
- Id, FirstName, LastName, etc. correspond to items in the document. The exact set of items usually depends on the form used to create the document. Also, the set of items is usually different than the set of columns in a corresponding view entry.

2.6.2 Updating a Document

You can also use the data service to update a document. To do so, you send a PUT request to the document resource URI. When sending a PUT request, you must include a Content-Type header as shown below:

```
Content-Type: application/json

{
    "Id":"CN=Adela Rojas\/O=renovations",
    "FirstName":"Adela",
    "LastName":"Rojas",
    "City":"Newark",
    "State":"NJ",
    "EMail":"adela_rojas@renovations.com"
}
```

The above request changes Adela Rojas's city from Paterson to Newark. If the data service completes the request without errors, it returns an HTTP status code of 200 without a response body.

NOTE: When sending a PUT request, you don't need to include any "@ properties" like @unid and @href. These properties are considered metadata. The data service ignores any attempt to update metadata.

Usually when you update a document, you want to execute the business logic contained in a specific form. To do so, you send a PUT request with additional URL parameters:

http://xyz.com/xpagesext.nsf/api/data/documents/unid/AAE5C9A07AF9C1A785257876004 8C0D6?form=Contact&computewithform=true

In this example, the form parameter specifies the Contact form and the computewithform parameter is true, instructing the data service to execute the Contact form's business logic.

2.6.3 Deleting a Document

To delete a document, you send an HTTP DELETE request to the document resource URI. If the data service deletes the document without errors, it returns an HTTP status code of 200.

2.7 Document Collection Resource

To create a new document, you send an HTTP POST request to the document collection resource URI:

http://{host}/{database}/api/data/documents

When sending a POST request, you must include a Content-Type header as shown below:

```
Content-Type: application/json

{
    "FirstName":"Bob",
    "LastName":"Dylan",
    "City":"Hibbing",
    "State":"MN",
    "EMail":"bdylan@renovations.com"
}
```

If the data service is able to create the document without errors, it returns an HTTP status code of 201. The response also includes a Location header identifying the URI of the new document resource:

```
Location: http://xyz.com/.../api/data/documents/unid/3249435909DCD22F852578A70063E8E5
```

Usually when you create a new document you want to execute the business logic contained in a specific form. To do so, you send a POST request with additional URL parameters. For example:

http://xyz.com/xpagesext.nsf/api/data/documents?form=Contact&computewithform

See section 2.6.2 for more information on the form and computewithform parameters.

Sometimes you want to create a document that is a response to another document. To do this, you send a POST request with a parentid parameter. For example:

http://xyz.com/xpagesext.nsf/api/data/documents?form=Discussion&computewithform=true &parentid=440FA99B2F0F839E852578760048C1AD

If you POST a request to the above URI, the data service creates a response to the document with an UNID of 440FA99B2F0F839E852578760048C1AD.

2.8 Troubleshooting the Data Service

The following sections list some common error conditions you might encounter. To help you troubleshoot the problem, each error condition is accompanied by one or more possible causes.

2.8.1 Bad Request

You send a request to a resource and the data service responds with an HTTP status code of 400 (Bad Request). Usually this is caused by a bad URL parameter or, in the case of a POST or PUT, a poorly formatted request body. For example, consider a GET request to this URI:

http://xyz.com/xpagesext.nsf/api/data/collections/unid/CD40A953ABDE036A852578650066 0C27?ps=10&page=x

The data service responds with a status code of 400 because page=x is invalid. Often the response body includes details about the error condition:

```
{
    "code":400,
    "text":"Bad Request",
    "message":"Invalid parameter page: x",
    "type":"text",
    "data":"java.lang.Exception: Invalid parameter page: x ...
}
```

2.8.2 Unauthorized

You send a request to a resource and the data service returns an HTTP status code of 401 (Unauthorized). This usually occurs for one of two reasons:

1. You are sending an anonymous request (no user credentials) to a protected resource. In this case, you can correct the problem by supplying the appropriate user credentials with each request. The Domino server supports both basic authentication (HTTP

Authorization header) and session authentication. See the appropriate Domino web server documentation for more about these authentication schemes.

2. The authenticated user has insufficient access to the resource. In this case, you can correct the problem by changing the access control list (ACL) for the database.

A response body in HTML format usually accompanies an HTTP status code of 401. The response body may help you troubleshoot the error:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Error</title></head>
<body text="#000000">
<h1>Error 401</h1>You are not authorized to perform this operation</body>
</html>
```

2.8.3 Forbidden

You send a request to a resource and the data service returns an HTTP status code of 403 (Forbidden). There can be many causes for this error:

- The data service could be disabled for this server. See section 2.1.1 for instructions for enabling the data service on a server.
- The data service could be disabled for this database. See section 2.1.2 for instructions for enabling the data service for a database.
- The data service could be disabled for this view (or folder). See section 2.1.3 for instructions for enabling the data service for a view.

2.8.4 Resource Not Found

You send a request to a resource and the data service returns an HTTP status code of 404 (Not Found). This can be caused by an improperly formatted URI. For example, the following two URIs are incorrect:

http://xyz.com/xpagesext.nsf/api/data/collection/name/AllContacts
http://xyz.com/xpagesext.nsf/api/data/collections/unid/AllContacts

The first URI is wrong because the collection path element should be plural (collections). The second URI is wrong because AllContacts is not a valid UNID.

Often, the data service will include additional details in the response body. For example, this response indicates the data service cannot find the specified view:

```
{
    "code":404,
    "text":"Not Found",
    "message":"The URI must refer to an existing view."
}
```

If you have checked the URI syntax and the Domino web engine is still returning an HTTP status code of 404, you should verify the data service plug-ins are active in the OSGI runtime of the Domino server. From the Domino server console, type the following command

> tell http osgi ss

The response will be a list of OSGI plug-ins and their status. Verify that the following plug-ins are included:

com.ibm.domino.das_8.5.3.yyyymmdd-hhmm.jar
com.ibm.domino.services_8.5.3.yyyymmdd-hhmm.jar
com.ibm.wink_8.5.3.yyyymmdd-hhmm.jar

If these plug-ins are not included, the data serice has not been installed correctly on the Domino server. Please see the extension library installation procedure.

2.8.5 Could Not Connect

You send a request to a resource and the message "Error: Could not connect to server" is returned. Confirm the HTTP task is running on the Domino server. From the Domino server console enter the command:

> show tasks

Confirm the following line appears:

HTTP Server Listen for connect requests on TCP Port:80

Please see the Domino Web Server documentation for additional information about troubleshooting HTTP connection problems.

2.8.6 Internal Server Error

You send a request to a resource and the data service returns an HTTP status code of 500 (Internal Server Error). This should happen very rarely, if at all. As described in the previous sections, the data service is designed to respond with status codes that help you pinpoint the cause of the problem. An HTTP status code of 500 could be caused by a software defect. Please report these errors to IBM.

3. XPages REST Services Control

3.1 Accessing Domino Data Service from XPages REST Services Control.

As stated previously, all the Domino Data Services resources can be accessed from the XPages REST Services Control. The data is exposed using the **pathinfo** property of the REST services control. From the REST Services Control you can select one of many services including the following data services:

- xe:databaseCollectionJsonService - Read the list of databases on the server.
- xe:viewCollectionJsonService - Read the list of views and folders in a database.
- xe: viewJsonService - Read the design of a view or folder.
- xe: viewJsonService - Read the entries in a view or folder.
- xe:documentJsonService - Create, read, update and delete documents.

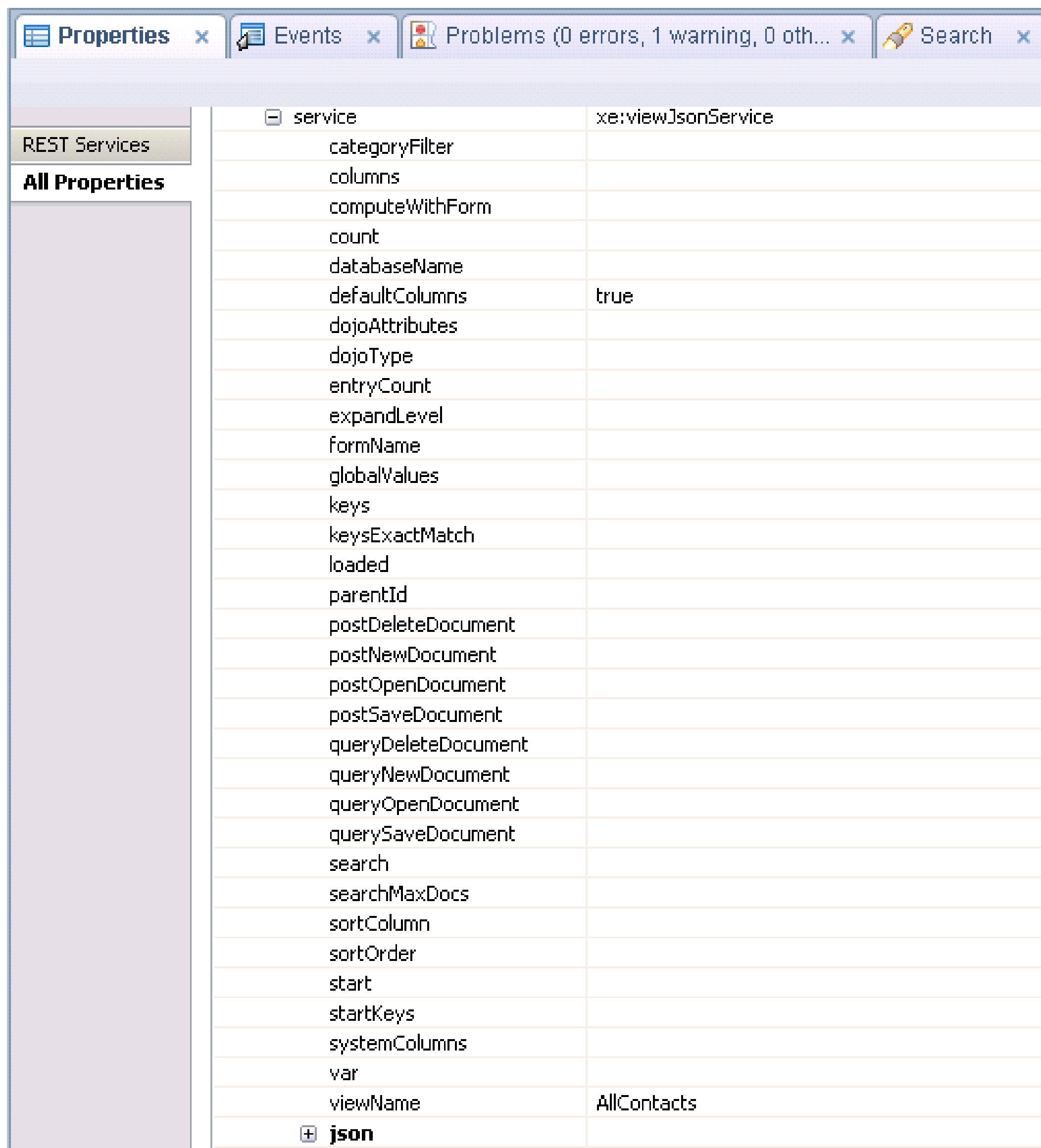


Figure 3.1.1 ViewJsonService Parameters

Depending on the REST Services Control you select you will receive additional properties that map to the parameters supported by the service as documented in Section 2. For example, the Form field can be set as a URL parameter **form** or a property **formName** (see Figure 3.1.1). In most cases the name of the service property matches the parameter.

3.2 Tutorial

In this section we will walk through the steps on how to build and reference an XPage that uses the REST Services Control to access the Domino Data Services. In this tutioral we will access the same resource described in Section 2.2 Database Collection Resource.

1. Use Domino Designer to open a database or create a new database.
2. From the menu in Domino Designer select Create > Design > XPage.
3. In the New XPage dialog enter "MyXPage" for the Name field as shown in Figure 3.2.1.

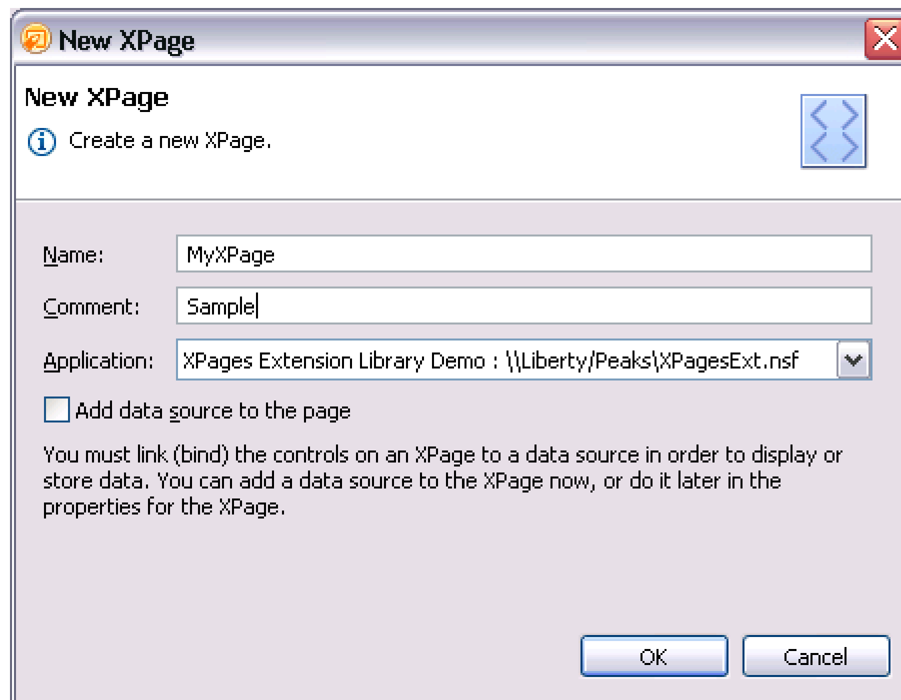


Figure 3.2.1

4. From the menu in Domino Designer select Create > Extension Library > Rest Services
5. Select the Rest Services control as shown in Figure 3.2.2.

Figure 3.2.2

6. From the Properties tab select All Properties and enter “myPathInfo” for the pathInfo property as show in Figure 3.2.3.

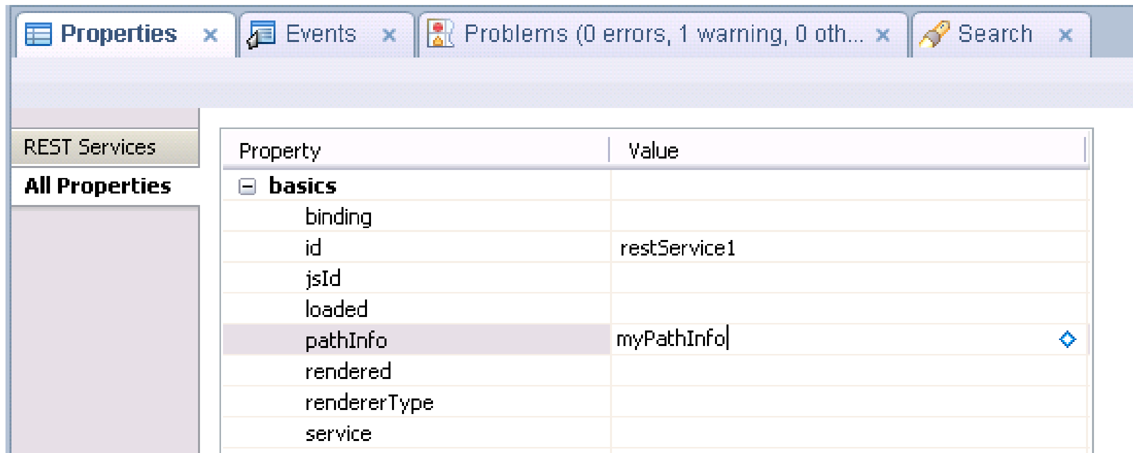


Figure 3.2.3

7. From the Properties tab select All Properties and select “xe:databaseCollectionJsonService” for the service property as show in Figure 3.2.4.

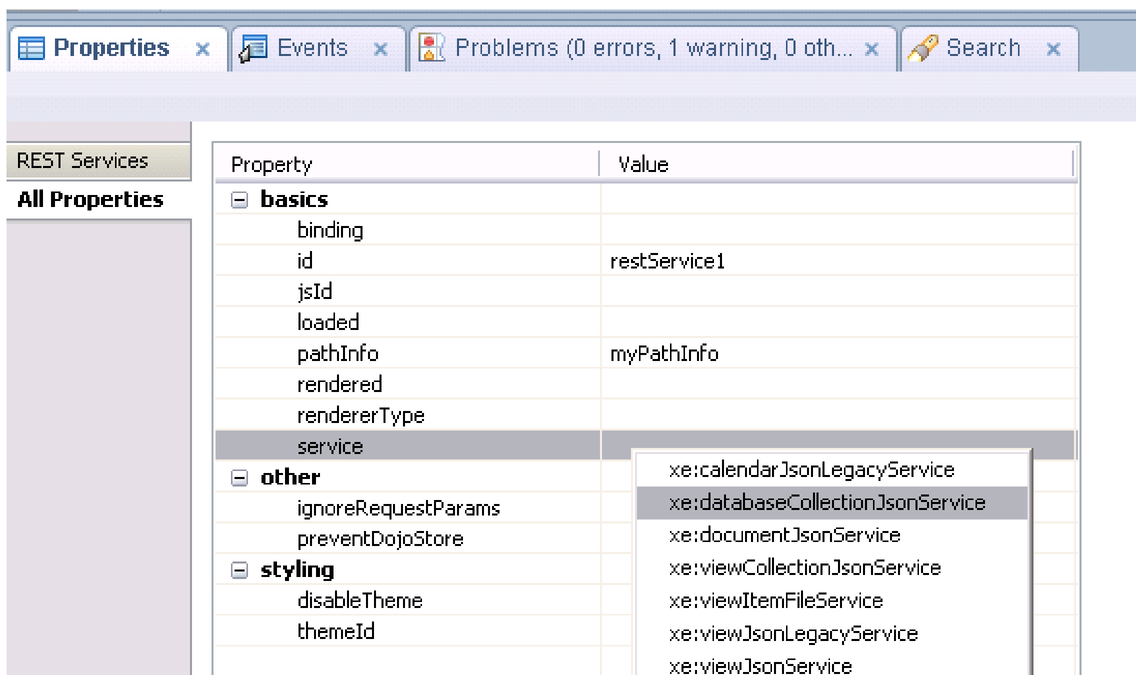


Figure 3.2.4

8. From the menu in Domino Designer select File > Save (Ctrl+S).
9. (Optional) You can now view the source code in the XPage

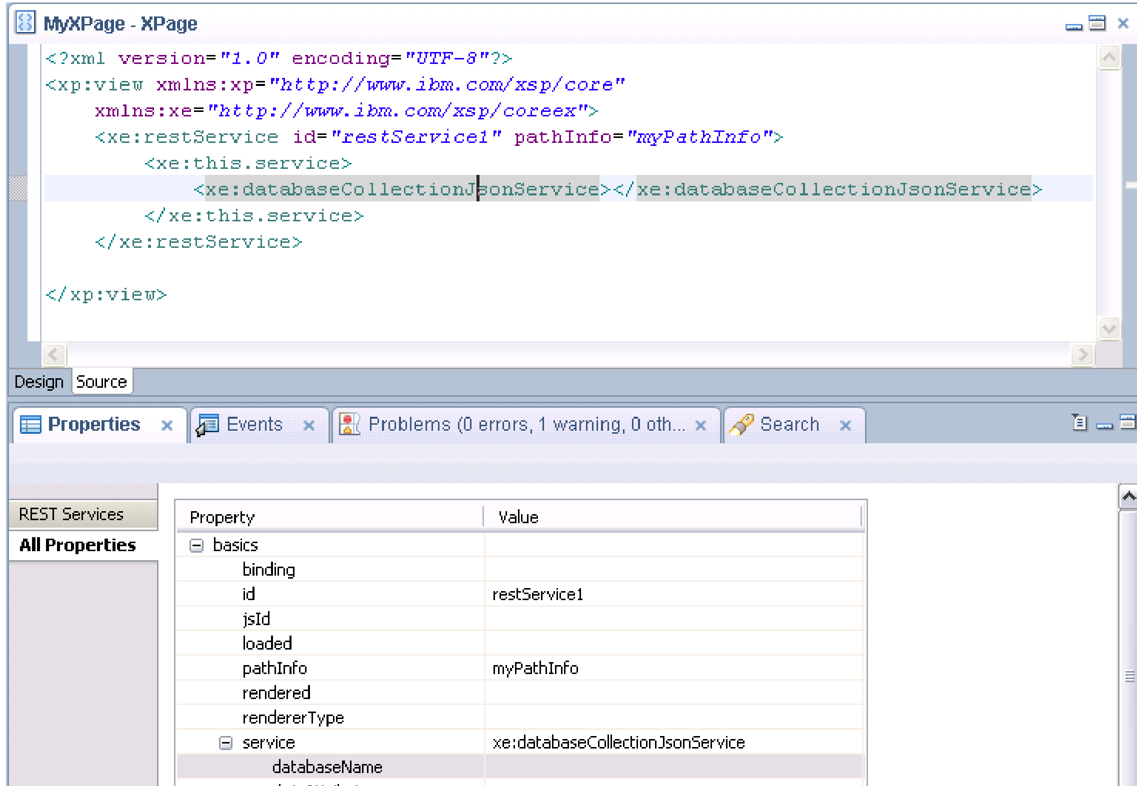


Figure 3.2.5

10. From a browser enter the URL:
    http://xyz.com/xpagesext.nsf/MyXpage.xsp/myPathInfo
11. The response in JSON will look similar to the response described in Section 2.2 Database Collection Resource.

3.3 Domino Data Service from XPages Samples and Additional Documentation

Another good resource is the sample database “XPagesExt.nsf “ that is included with the XPages Extension Library. This sample application called XPages Extension Library Demo includes a REST tab that has several samples that demonstrate the REST Data Service.

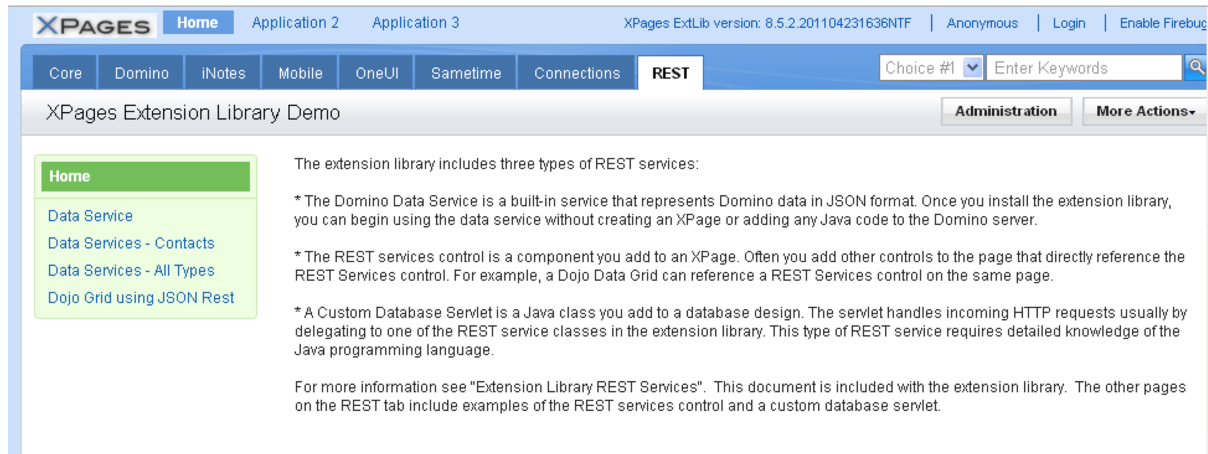


Figure 3.3.1 REST Samples Home Page

3.3.1 Data Services Sample Page

This page contains an example that demonstrates all the Domino Data REST Services. A button is used to launch a URL that references each REST Services control as shown in Figure 3.3.2.

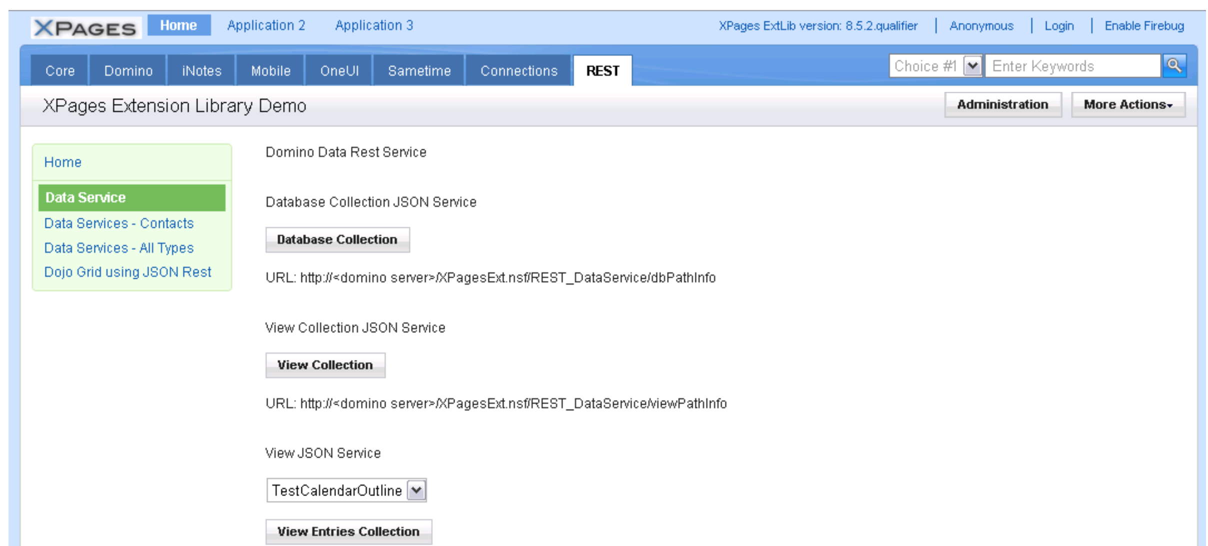


Figure 3.3.2 Data Services Sample Page

3.3.2 Dojo Grid Sample Page

This page contains an example that demonstrates a Dojo Data Grid referencing a REST Services control on the same page as shown in Figure 3.3.3. The REST Services control uses xe:viewJsonService to access the “AllContacts” view. The data in the grid can be updated and then saved back to the database.

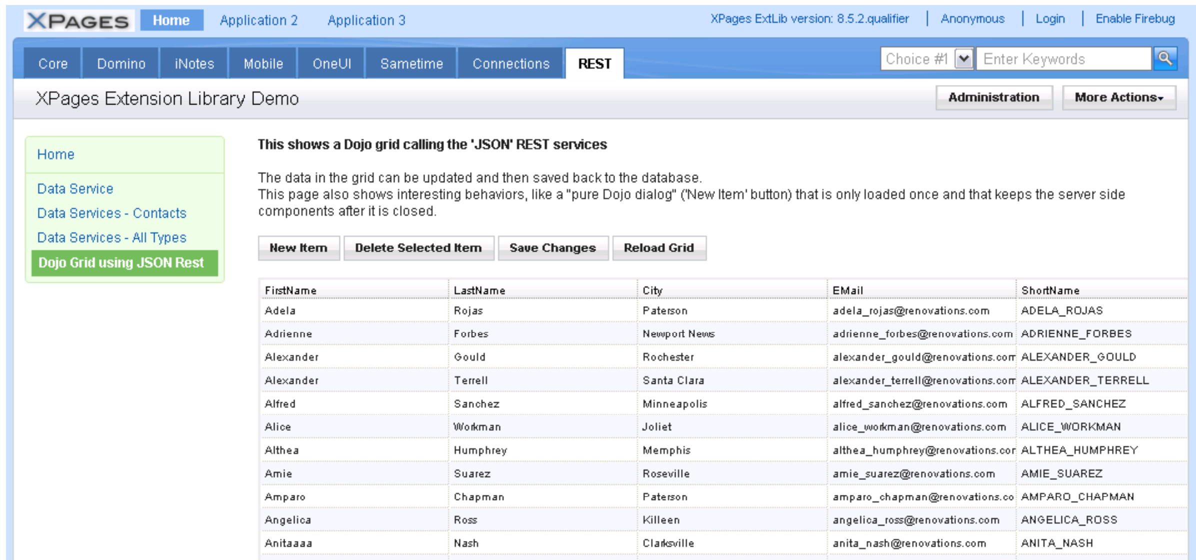


Figure 3.3.3 Dojo Grid Sample Page

3.3.3 XPages REST Control Design Pattern

You can find additional information on the design pattern in section “Separating UI and data using complex types” in *XPages Extension Library Documentation*.

4. Custom Database Servlet

A custom database servlet is a set of Java classes you add to a Notes database. These classes handle HTTP requests for a URL in the following format:

http://{host}/{database}/xsp/services/{servicepath}

You can add more than one logical service to a single custom servlet. Each service class has a unique URL because each class is bound to a different {servicepath} segment.

You have complete control over how your custom servlet handles HTTP requests. However, the easiest way to get started is to delegate HTTP requests to one of the REST service classes included in the extension library. These are the same classes used by the REST services control described in section 3. For example, you can use RestViewJsonService to quickly build a service that handles RESTful requests for a particular view. Unlike the REST services control, you don't have to create an XPage to build a service, but you do need to understand Java.

4.1 Adding a Custom Servlet to a Database

Let's assume you have an existing application you would like to access via a REST service. You could enable the Domino Data Service (section 2), but you want to customize some of the responses. The REST services control (section 3) is a more customizable option, but you don't necessarily want to create an XPage just to create a REST service. If you are already familiar with Java, a custom database servlet is probably the best fit.

This section includes the steps required to add a custom servlet to an existing database. The example is an application called Store.nsf. The database includes a Products view that lists all of the products in the store. You want to access the Products view with a custom REST service.

1. Open the database in Domino Designer.
2. In the navigator on the left, find the Application Properties element for the database. Double click to open Application Properties.

3. In the XPage Libraries section of the Advanced Properties tab, enable com.ibm.xsp.extlib.library as shown below:

Advanced Properties

Unread Marks

Maintain unread marks

Replicate: Never

Space Savers

Compress database design

Compress document data

Use LZ1 compression for attachments

Use Domino Attachment and Object Service

Advanced Options

Optimize document table map

Don't overwrite free space

Maintain LastAccessed property

Disable transaction logging

Limit entries in $UpdatedBy fields: 0

Limit entries in $Revisions fields: 0

Features

Allow more fields in database

Don't allow simple search

Allow soft deletions

Permanent document deletion interval (hours): 48

Support Response Thread History

Don't support specialized response hierarchy

Disable automatic updating of views

Disable export of view data

Don't allow headline monitoring

XPage Libraries

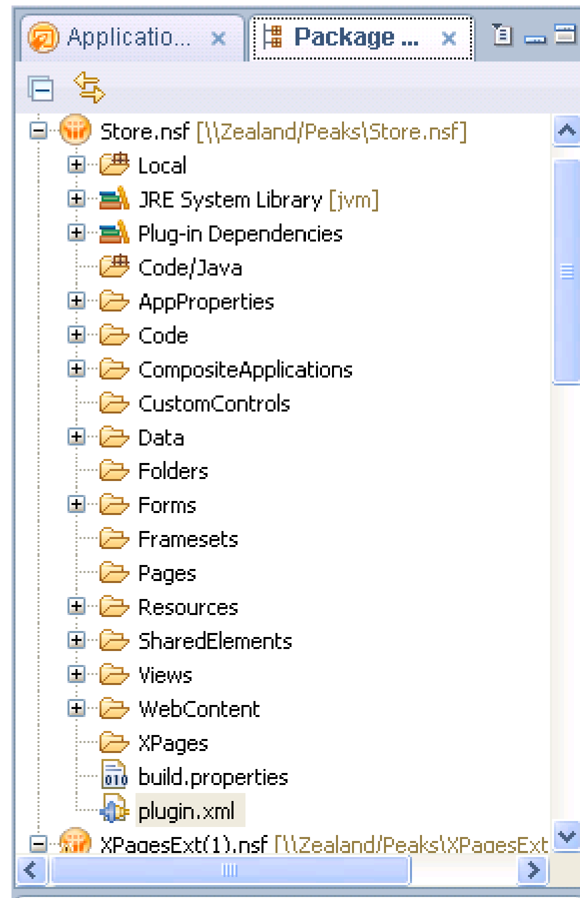
Select the libraries of extended XPage controls to use in this application.

Library ID

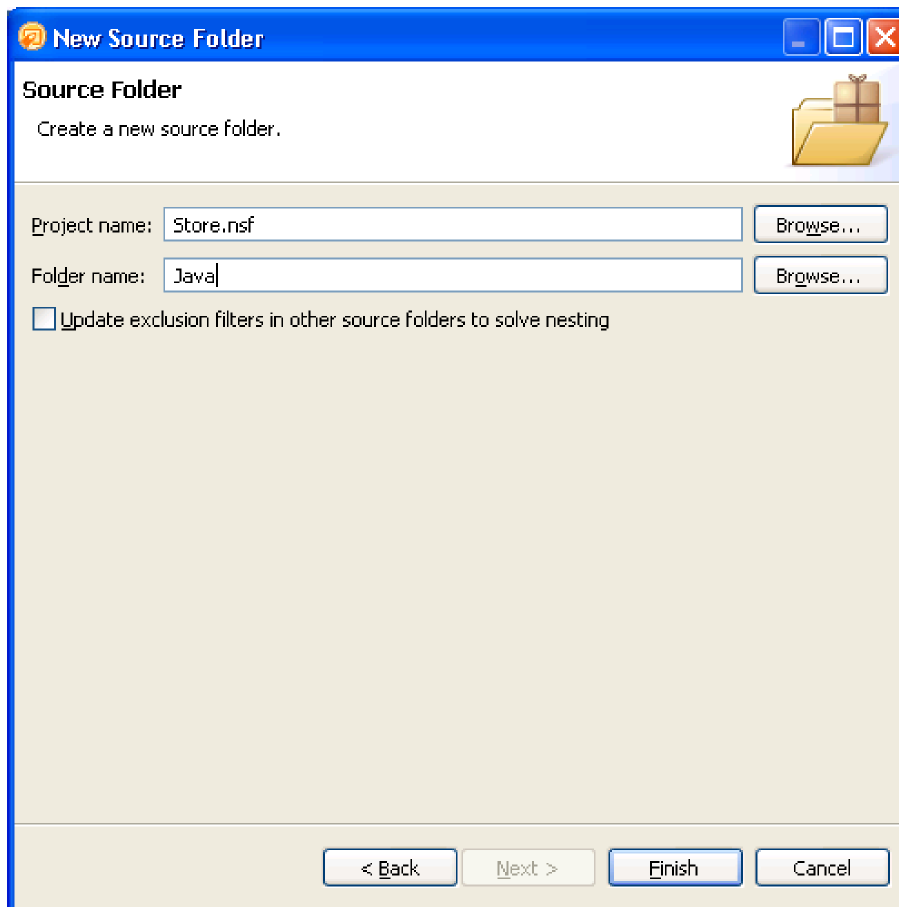
com.ibm.xsp.extlib.library

4. Save the Application Properties design element.

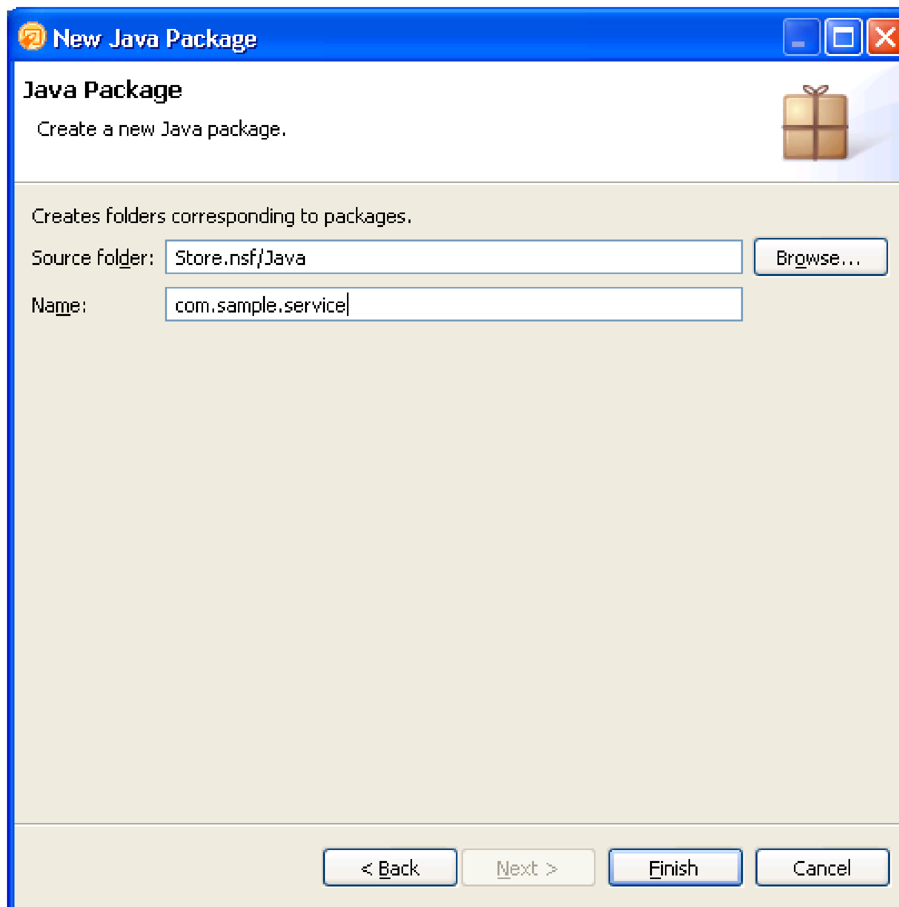
5. Open the Eclipse package explorer (Window – Show Eclipse Views – Package Explorer). You should see your database's design elements in the package explorer like this:



6. Create a new Java source folder. For example, right click on the database and select New – Other – Java – Source Folder. In the New Source Folder dialog, add “Java” to the Folder name box and click Finish.



7. Add a new package to the Java source folder. For example, right click on the Java folder and select New – Other – Java – Package. In the New Java Package dialog, enter a package name and click Finish.



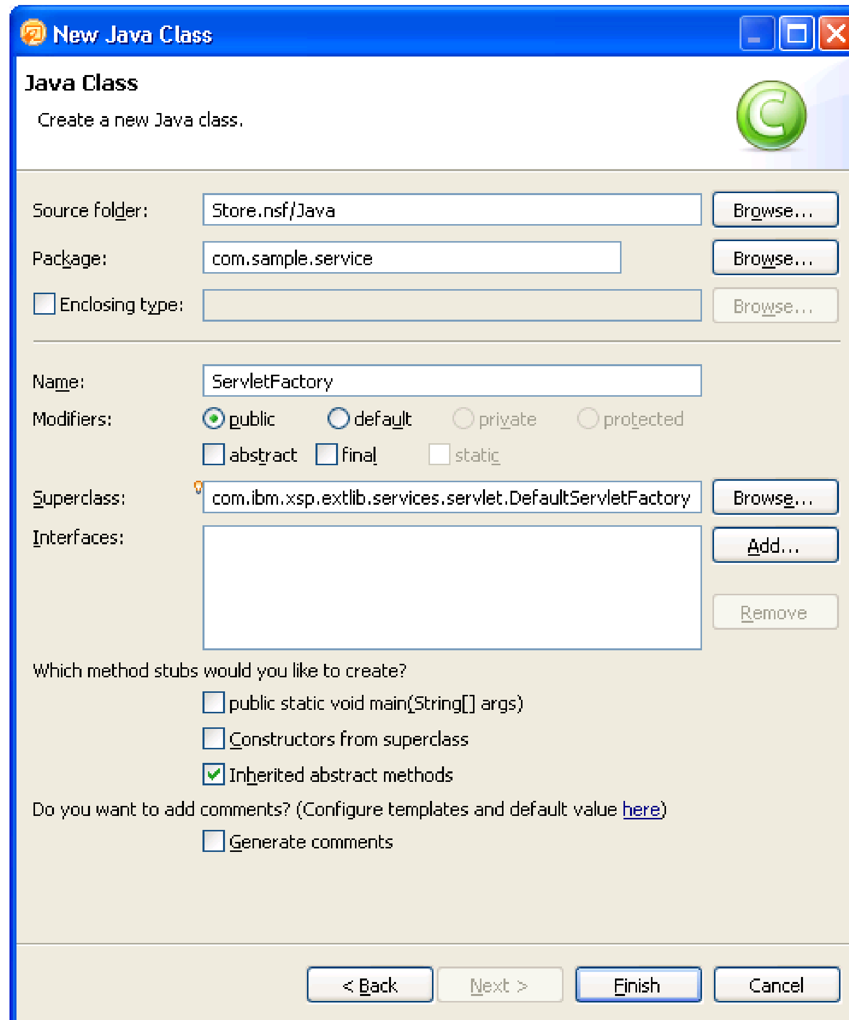
8. Add a ServletFactory class to the new package. For example, right click on the package and select New – Other – Java – Class. In the New Java Class dialog, enter the following in the Name box:

   ServletFactory

   Enter the following in the Superclass box:

   com.ibm.extlib.services.servlet.DefaultServletFactory

   Then click Finish.



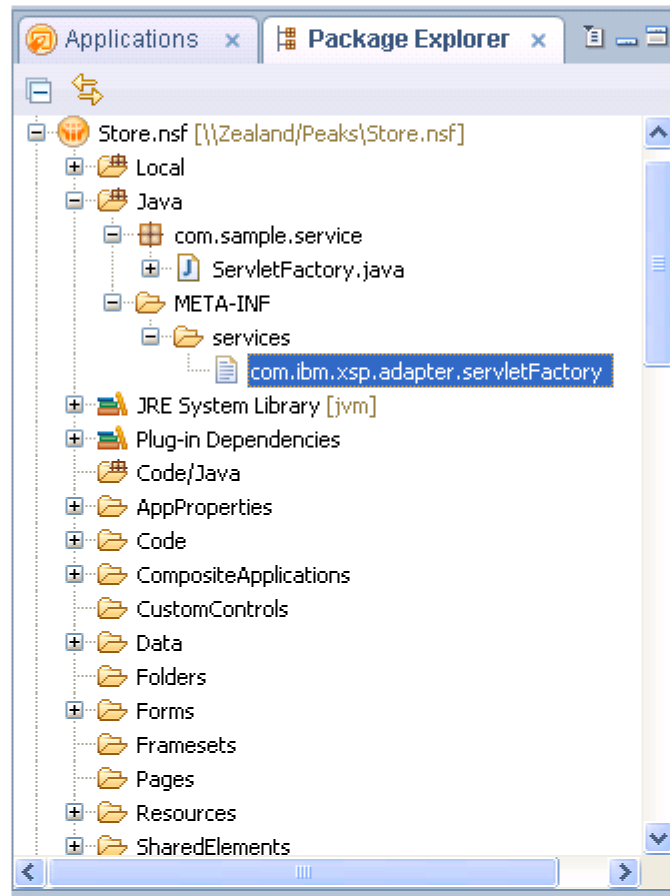
9. Replace the contents of ServletFactory.java with the code from Source Listing 4.1. We'll explain the Java code later. For now, just note the following line assumes your database includes a view called Products:

```
p.setViewName("Products");
```

   If you would like to use a different view, change the code accordingly.

10. Register the servlet factory with XPages. To do this, add a file named com.ibm.xsp.adapter.servletFactory to a folder named META-INF\services. The

contents of the file should be as shown in Source Listing 4.2. When you are done the project should look like this:



At this point you can test your servlet. The easiest way to do this is to use a browser to send an HTTP GET request to the servlet URL. Type the following URL into your browser's address bar:

http://{host}/Store.nsf/xsp/services/Products

Your service should return a response like this:


```
[
  {
      "@entryid":"1-F73088139B943195852578B20066F538",
      "@unid":"F73088139B943195852578B20066F538",
      "@noteid":"8FA",
      "@position":"1",
      "@read":true,
      "@siblings":3,
      "@form":"Product",
      "ProductName":"GT-2160",
      "Company":"Asics",
      "ProductPrice":"$89.99"
  },
  {
```

```
        "@entryid":"2-95D7A6F090DF96F7852578B2006DB6E2",
        "@unid":"95D7A6F090DF96F7852578B2006DB6E2",
        "@noteid":"8FE",
        "@position":"2",
        "@read":true,
        "@siblings":3,
        "@form":"Product",
        "ProductName":"ProGrid Ride",
        "Company":"Saucony",
        "ProductPrice":"$89.99"
    },
    {
        "@entryid":"3-95B3CA2BDEE839FF852578B20066E92D",
        "@unid":"95B3CA2BDEE839FF852578B20066E92D",
        "@noteid":"8F6",
        "@position":"3",
        "@read":true,
        "@siblings":3,
        "@form":"Product",
        "ProductName":"Wave Rider",
        "Company":"Mizuno",
        "ProductPrice":"$89.99"
    }
]
```


Of course the exact response depends on the design and contents of the Products view in your database.

Source Listing 4.1. ServletFactory.java

```
package com.sample.service;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ibm.domino.services.ServiceEngine;
import com.ibm.domino.services.rest.das.view.RestViewJsonService;
import com.ibm.domino.services.rest.das.view.impl.DefaultViewParameters;
import com.ibm.xsp.extlib.services.servlet.DefaultServiceFactory;
import com.ibm.xsp.extlib.services.servlet.DefaultServletFactory;
import com.ibm.xsp.extlib.services.servlet.ServiceFactory;

public class ServletFactory extends DefaultServletFactory {

    private static ServiceFactory createFactory() {
        DefaultServiceFactory factory = new DefaultServiceFactory();

        factory.addFactory("Products", new ServiceFactory() {
            public ServiceEngine createEngine(HttpServletRequest httpRequest,
                 HttpServletResponse httpResponse) throws ServletException {
                    DefaultViewParameters p = new DefaultViewParameters();
                    p.setViewName("Products");
                    p.setGlobalValues(DefaultViewParameters.GLOBAL_ALL);
                    p.setSystemColumns(DefaultViewParameters.SYSCOL_ALL);
                    p.setDefaultColumns(true);
                    // Set the default parameters
                    p.setStart(0);
                    p.setCount(4);
                    return new RestViewJsonService(httpRequest,httpResponse,p);
                }
            }
        );

        return factory;
    }

    public ServletFactory() {
        super("services","Extension Library Services Servlet",createFactory());
    }
}
```

Source Listing 4.2. com.ibm.xsp.adapter.servletFactory

```
com.sample.service.ServletFactory
```

4.2 Explanation of Sample Code

As shown in the preceding sample code (listings 4.1 and 4.2), a custom database servlet requires multiple Java classes. It requires a servlet factory, usually a subclass of com.ibm.xsp.extlib.services.servlet.DefaultServletFactory. It requires a service factory, in this case an instance of com.ibm.xsp.extlib.services.servlet.DefaultServiceFactory, and it requires an instance of com.ibm.domino.services.ServiceEngine. Each of these classes has a distinct purpose and lifecycle.

You register your servlet factory by adding a line to the com.ibm.xsp.adapter.servletFactory file as shown in listing 4.2. Your servlet factory is created just once for multiple HTTP requests. In

our example, the servlet factory constructor calls the createFactory method to create and initialize an instance of DefaultServiceFactory. The servlet factory passes the service factory to its super constructor.

Take a closer look at the createFactory method in listing 4.1. After creating an instance of DefaultServiceFactory, it calls the addFactory method to add an anonymous instance of ServiceFactory. This anonymous class must override the createEngine method, which is called every time the extension library receives an HTTP request for your servlet. In our example, the createEngine method simply delegates to RestViewJsonService. In other words, all HTTP requests are handled by RestViewsonService.

Now let’s imagine you want to access another view – for example, a view named Vendors -- with the same servlet. To do this you can change createFactory to call addFactory a second time. The following code would add a Vendors service to the same servlet:

```
factory.addFactory("Vendors", new ServiceFactory() {
    public ServiceEngine createEngine(HttpServletRequest httpRequest,
         HttpServletResponse httpResponse) throws ServletException {
             DefaultViewParameters p = new DefaultViewParameters();
             p.setViewName("Vendors");
             p.setGlobalValues(DefaultViewParameters.GLOBAL_ALL);
             p.setSystemColumns(DefaultViewParameters.SYSCOL_ALL);
             p.setDefaultColumns(true);
             // Set the default parameters
             p.setStart(0);
             p.setCount(4);
             return new RestViewJsonService(httpRequest,httpResponse,p);
        }
    }
);
```

In other words, you can create one custom servlet that contains more than one logical service. Each service has its own path as defined by the first parameter of the addFactory method.

4.3 Customizing the Service

Until now our custom database servlet just uses RestViewJsonService to handle HTTP requests. When you send a GET request to the service URL, the response is determined completely by the implementation of RestViewJsonService. This section provides some examples of how you can customize a service. The examples all use RestViewJsonService, but the same principles apply to other subclasses of RestServiceEngine including:

- RestDatabaseCollectionJsonService
- RestDocumentJsonService
- RestViewCollectionJsonService

The simplest way to customize RestViewJsonService is to modify the parameters used to initialize the service. For example, listing 4.1 includes this line:

```
p.setSystemColumns(DefaultViewParameters.SYSCOL_ALL);
```

This tells the service to emit all system columns including UNID, note ID, position, read mark, etc. for each entry in the view. An example of an entry with all system columns is as follows:

```
{
    "@entryid":"1-F73088139B943195852578B20066F538",
    "@unid":"F73088139B943195852578B20066F538",
    "@noteid":"8FA",
```

```
        "@position":"1",
        "@read":true,
        "@siblings":3,
        "@form":"Product",
        "ProductName":"GT-2160",
        "Company":"Asics",
        "ProductPrice":"$89.99"
    }
```

You can reduce the number of system columns, by changing the input to setSystemColumns(). For example, the following line tells the service to just include UNID and position:

```
p.setSystemColumns(DefaultViewParameters.SYSCOL_UNID |
        DefaultViewParameters.SYSCOL_POSITION;
```

After making this change to the code in listing 4.1, your service formats each entry like this:

```
{
    "@entryid":"1-F73088139B943195852578B20066F538",
    "@unid":"F73088139B943195852578B20066F538",
    "@position":"1",
    "ProductName":"GT-2160",
    "Company":"Asics",
    "ProductPrice":"$89.99"
}
```

Another way to customize your view service is to add one or more columns to each entry. Let's say all products from one manufacturer are on sale. You want to indicate which products are on sale in the service response. To do this, you create a list of RestViewColumn objects (in this case there is just one object in the list). Then you use DefaultViewParameters.setColumns() to add the column list to the service.

The following changes to createFactory are an example:

```
private static ServiceFactory createFactory() {
    DefaultServiceFactory factory = new DefaultServiceFactory();

    final List<RestViewColumn> customColumns = Arrays.asList( (RestViewColumn)
        new DefaultViewColumn("onsale") {
            @Override
            public Object evaluate(RestViewService service, RestViewEntry entry)
                            throws ServiceException {
                String company = entry.getColumnValue("Company").toString();
                Boolean v = null;
                if ( "Asics".equals(company) ) {
                    v = new Boolean(true);
                } else {
                    v = new Boolean(false);
                }

                return v;
            }
        }
    );

    factory.addFactory("Products", new ServiceFactory() {
        public ServiceEngine createEngine(HttpServletRequest httpRequest,
                            HttpServletResponse httpResponse) throws ServletException {
            DefaultViewParameters p = new DefaultViewParameters();
            p.setViewName("Products");
            p.setGlobalValues(DefaultViewParameters.GLOBAL_ALL);
            p.setSystemColumns(DefaultViewParameters.SYSCOL_UNID |
                    DefaultViewParameters.SYSCOL_POSITION);
            p.setDefaultColumns(true);
            p.setColumns(customColumns);
```

```
            // Set the default parameters
            p.setStart(0);
            p.setCount(4);
            return new RestViewJsonService(httpRequest,httpResponse,p);
        }
    });

    return factory;
}
```

The above changes produce the following response. Notice the new onsale property for each view entry:


```
[
  {
      "@entryid":"1-F73088139B943195852578B20066F538",
      "@unid":"F73088139B943195852578B20066F538",
      "@position":"1",
      "ProductName":"GT-2160",
      "Company":"Asics",
      "ProductPrice":"$89.99",
      "onsale":true
  },
  {
      "@entryid":"2-95D7A6F090DF96F7852578B2006DB6E2",
      "@unid":"95D7A6F090DF96F7852578B2006DB6E2",
      "@position":"2",
      "ProductName":"ProGrid Ride",
      "Company":"Saucony",
      "ProductPrice":"$89.99",
      "onsale":false
  },
  {
      "@entryid":"3-95B3CA2BDEE839FF852578B20066E92D",
      "@unid":"95B3CA2BDEE839FF852578B20066E92D",
      "@position":"3",
      "ProductName":"Wave Rider",
      "Company":"Mizuno",
      "ProductPrice":"$89.99",
      "onsale":false
  }
]
```


The most advanced way to customize your service is to subclass one of the JSON content classes included with the extension library. To do this you must first understand that each of the REST service classes delegates some of its work to a JSON content class. For example:

- RestViewJsonService delegates to JsonViewEntryCollectionContent.
- RestDatabaseCollectionJsonService delegates to JsonDatabaseCollectionContent.
- RestDocumentJsonService delegates to JsonDocumentContent.

Therefore you can customize your service by subclassing one of the JSON content classes and overriding one or more methods. As a trivial example you can make all JSON property names lower case by subclassing JsonViewEntryCollectionContent. Source listing 4.3 shows a class that extends JsonViewEntryCollectionContent. To use this class you make the following change to createFactory:

```
//return new RestViewJsonService(httpRequest, httpResponse, p);
return new RestViewJsonService(httpRequest,httpResponse,p, new JsonContentFactory(){

    public JsonViewEntryCollectionContent createViewEntryCollectionContent(
                View view, RestViewService service) {
        return new CustomViewEntryCollectionContent(view, service);
    }

});
```

In other words instead of using the three-parameter constructor for RestViewJsonService, you use the four-parameter constructor. The fourth parameter is an instance of JsonContentFactory that knows how to create an instance of CustomViewEntryCollectionContent.

After these changes, your service produces the following response. Notice the JSON properties are now all lower case:

```
[
  {
      "@entryid":"1-F73088139B943195852578B20066F538",
      "@unid":"F73088139B943195852578B20066F538",
      "@position":"1",
      "productname":"GT-2160",
      "company":"Asics",
      "productprice":"$89.99",
      "onsale":true
  },
  {
      "@entryid":"2-95D7A6F090DF96F7852578B2006DB6E2",
      "@unid":"95D7A6F090DF96F7852578B2006DB6E2",
      "@position":"2",
      "productname":"ProGrid Ride",
      "company":"Saucony",
      "productprice":"$89.99",
      "onsale":false
  },
  {
      "@entryid":"3-95B3CA2BDEE839FF852578B20066E92D",
      "@unid":"95B3CA2BDEE839FF852578B20066E92D",
      "@position":"3",
      "productname":"Wave Rider",
      "company":"Mizuno",
      "productprice":"$89.99",
      "onsale":false
  }
]
```

Throughout this section we have made changes to the ServletFactory class shown in listing 4.1. See listing 4.2 for the final version of ServletFactory.java.

Source Listing 4.3. CustomViewEntryCollectionContent.java

```
package com.sample.service;

import java.io.IOException;

import lotus.domino.View;

import com.ibm.domino.services.ServiceException;
import com.ibm.domino.services.content.JsonViewEntryCollectionContent;
import com.ibm.domino.services.rest.das.view.RestViewNavigator;
import com.ibm.domino.services.rest.das.view.RestViewService;
import com.ibm.domino.services.util.JsonWriter;

public class CustomViewEntryCollectionContent extends
                JsonViewEntryCollectionContent {

    public CustomViewEntryCollectionContent(View view, RestViewService service) {
        super(view, service);
    }

    protected void writeColumn(JsonWriter jsonWriter, RestViewNavigator navigator,
                        int colIdx, String colName, Object colValue)
                        throws IOException, ServiceException {
        super.writeColumn(jsonWriter, navigator, colIdx,
                colName.toLowerCase(), colValue);
    }
}
```

Source Listing 4.4. ServletFactory.java (final version)

```
package com.sample.service;

import java.util.Arrays;
import java.util.List;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import lotus.domino.View;

import com.ibm.domino.services.ServiceEngine;
import com.ibm.domino.services.ServiceException;
import com.ibm.domino.services.content.JsonContentFactory;
import com.ibm.domino.services.content.JsonViewEntryCollectionContent;
import com.ibm.domino.services.rest.das.view.RestViewColumn;
import com.ibm.domino.services.rest.das.view.RestViewEntry;
import com.ibm.domino.services.rest.das.view.RestViewJsonService;
import com.ibm.domino.services.rest.das.view.RestViewService;
import com.ibm.domino.services.rest.das.view.impl.DefaultViewColumn;
import com.ibm.domino.services.rest.das.view.impl.DefaultViewParameters;
import com.ibm.xsp.extlib.services.servlet.DefaultServiceFactory;
import com.ibm.xsp.extlib.services.servlet.DefaultServletFactory;
import com.ibm.xsp.extlib.services.servlet.ServiceFactory;

public class ServletFactory extends DefaultServletFactory {

    private static ServiceFactory createFactory() {
        DefaultServiceFactory factory = new DefaultServiceFactory();

        final List<RestViewColumn> customColumns = Arrays
                .asList((RestViewColumn) new DefaultViewColumn("OnSale") {
                    @Override
                    public Object evaluate(RestViewService service,
                            RestViewEntry entry) throws ServiceException {
```

```
                        String company = entry.getColumnValue("Company")
                                .toString();
                        Boolean v = null;
                        if ("Asics".equals(company)) {
                            v = new Boolean(true);
                        } else {
                            v = new Boolean(false);
                        }

                        return v;
                    }
                });

        factory.addFactory("Products", new ServiceFactory() {
            public ServiceEngine createEngine(HttpServletRequest httpRequest,
                    HttpServletResponse httpResponse) throws ServletException {
                DefaultViewParameters p = new DefaultViewParameters();
                p.setViewName("Products");
                p.setGlobalValues(DefaultViewParameters.GLOBAL_ALL);
                p.setSystemColumns(DefaultViewParameters.SYSCOL_UNID
                        | DefaultViewParameters.SYSCOL_POSITION);
                p.setDefaultColumns(true);
                p.setColumns(customColumns);
                // Set the default parameters
                p.setStart(0);
                p.setCount(4);
                // return new RestViewJsonService(httpRequest, httpResponse, p);
                return new RestViewJsonService(httpRequest, httpResponse, p,
                        new JsonContentFactory() {

                            public JsonViewEntryCollectionContent
                                createViewEntryCollectionContent(
                                    View view, RestViewService service) {
                                return new CustomViewEntryCollectionContent(
                                        view, service);
                            }

                        });
            }
        });

        return factory;
    }

    public ServletFactory() {
        super("services", "Extension Library Services Servlet", createFactory());
    }
}
```

5. Custom Wink Servlet

To be completed in a future release of the extension library.