

## Using Open Data Protocol (OData) with RESTful web APIs

REST is an architecture style for sending messages back and forth from client to server over HTTP. As there is no standard for querying and updating data, each client requires having specific implementation for each REST API. The **Open Data Protocol (OData)** defines a web protocol for querying and updating data via RESTful web APIs. In plain words, OData helps you to standardize data exchange over RESTful web services. Many companies including Microsoft, IBM, and SAP support OData today and it is governed by **Organization for the Advancement of Structured Information Standards (OASIS)**. At the time of writing this book, the latest release of OData is version 4.0.

---

Many API vendors started considering at OData for standardising their REST APIs, especially with release of OData version 4.0. A detailed discussion of OData is beyond the scope of this book. To learn more about OData, visit the official documentation page available at the following location: <http://www.odata.org/documentation>

---

### A quick look at the OData protocol

OData provides you with a uniform way of describing the data and the data model. This helps when you need to consume REST APIs from various vendors. Let us take a quick look at some of the core features offered in OData with some example.

### URI convention for OData based REST APIs

OData specification defines a set of recommendation for forming the URIs that identify OData based REST APIs. The URI for an OData service may take up to three parts as follows:

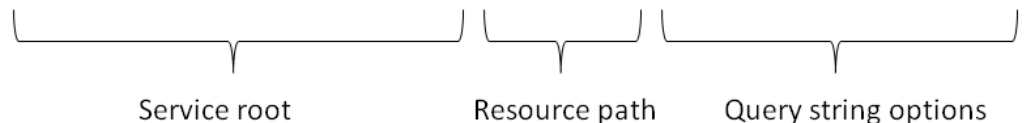
Service root: Identifies the root of an OData service

Resource path: Identifies the resources exposed by an OData service.

Query string options: The query string identifies the query options (built-in or custom) for the resource.

Here is an example:

`http://localhost:8080/hrapp/odata/Departments?$top=2&$orderby=Location`



The diagram illustrates the components of the URI `http://localhost:8080/hrapp/odata/Departments?$top=2&$orderby=Location`. Brackets below the URI identify three parts:   
1. **Service root**: `http://localhost:8080/hrapp/odata/`   
2. **Resource path**: `Departments`   
3. **Query string options**: `?$top=2&$orderby=Location`

**Insert image B04017\_086\_02.png**

### Reading resources

Resources from OData RESTful APIs are accessible via HTTP GET request. For instance the following GET request retrieves Departments entity collection from OData REST API server.

GET `http://localhost:8080/hrapp/odata/Departments` HTTP/1.1

The result that you may get from OData REST API server in response to the above call will be structured in accordance with OData protocol specification. This keeps client code simple and reusable.

To read individual resource element, you can pass the unique identifier for the resource as shown in the following code snippet. The following example reads details of department with the given id:

```
GET http://localhost:8080/hrapp/odata/Departments(10) HTTP/1.1.
```

## Querying data

OData supports various kinds of query options as query parameters. For instance `$orderby` can be used for sorting the query results. Here is an example:

```
GET http://localhost:8080/hrapp/odata/Departments?$orderby= DepartmentName
HTTP/1.1
```

Similarly, you can use `$select` option for limiting the attributes on entity resources returned by a REST API. Here is an example:

```
GET http://localhost:8080/hrapp/odata/Departments?
$orderby=DepartmentName&$select=DepartmentName,ManagerId HTTP/1.1
```

Some of the frequently used query options are listed below:

Query Option	Description	Example
\$filter	This option allows client to filter a collection of resources	<code>/Employees? \$filter=FirstName eq 'Jobinesh'</code>
\$expand	Include the specified resource in line with retrieved resources	<code>/Departments(10)?\$expand= Employees</code>
\$select	Include supplied attributes alone in the resulting entity	<code>/Departments?\$select=Name, LocationId</code>
\$orderby	Sort the query result by one or more attributes	<code>Departments?\$orderby= DepartmentName desc</code>
\$top	Returns only the specified number of items (from top) in the result collection	<code>Departments?\$top=10</code>
\$skip	How many items needs to be skipped from top while returning the result	<code>Departments?\$skip=10</code>
\$count	Total items in the result	<code>Departments/\$count</code>

## Modifying data

Updatable OData services provides standardized interface for performing following operations on entities exposed via OData services:

Create : Done via HTTP POST

Update: Done via HTTP PUT or HTTP PATCH.

Delete: Done via HTTP DELETE

## Relationship operations

OData supports linking of related resources. Relationships from one entity to another are represented as navigation properties. Following API reads employees in HR department.

[http://localhost:8080/hrapp/odata/Departments\("HR"\)/EmployeeDetails](http://localhost:8080/hrapp/odata/Departments()

OData service even allows you to add, update and remove the relation via navigation properties. Following example shows how you can use navigation properties to link employee with id = 1700 to IT department.

```
POST odata/Departments('IT')/Employees/$ref
OData-Version: 4.0
Content-Type: application/json;odata.metadata=minimal
Accept: application/json
{
  "@odata.id": "odata/Employees(1700)"
}
```

## Transforming JPA model in to OData enabled RESTful web services

You can use Apache Olingo framework for enabling OData services for your JPA model. Apache Olingo is an open source Java library that implements OData protocol. At the time of writing this book latest release of Olingo was based on OData Version 2.0 specifications, and the support for OData Version 4.0 were underway.

With Olingo framework you can easily transform your JPA Models into OData Services using OData JPA Processor Library.

---

The complete source code for this example is available in the Packt website. You can download the example from the Packt website link that we mentioned at the beginning of this book, in the Preface section. In the downloaded source code, see the project < [rest-chapter8-jaxrs](#)>/rest-chapter8-odata-service .

---

High level steps for enabling OData 2 services for your JPA model are listed below:

Build a web project for holding the RESTful web API components.

Generate JPA entities for the application as appropriate. Now let us see how we can use Apache Olingo framework for generating OData services for your JPA model

Add dependency to Olingo OData Library (Java) and OData JPA Processor Library. The complete list of jars are listed here:  
<http://olingo.apache.org/doc/odata2/tutorials/CreateWebApp.html>

Add a Service Factory implementation that provides a means for initializing OData JPA Processors and data model provider (JPA entity). You can do this by adding a class that extends `org.apache.olingo.odata2.jpa.processor.api.ODataJPAServiceFactory` as shown here:

```
//Imports are removed for brevity
public class ODataJPAServiceFactoryImpl extends
    ODataJPAServiceFactory {
    //HR-PU is the persistence unit configured in persistence.xml
    //which is used in JPA model
    final String PUNIT_NAME = "HR-PU";
    @Override
    public ODataJPAContext initializeODataJPAContext()
        throws ODataJPARuntimeException {
        ODataJPAContext oDataJPAContext = getODataJPAContext();
        oDataJPAContext.setEntityManagerFactory(
            JPAEntityManagerFactory.
```

```

        getEntityManagerFactory(PUNIT_NAME));
    oDataJPAContext.setPersistenceUnitName(PUNIT_NAME);
    return oDataJPAContext;
}
//Other methods are removed for brevity
}

```

Configure the web application by adding `CXFNonSpringJaxrsServlet` configuration to web.xml. Specify the service factory implementation class that you created in last as one of the init parameter for the servlet. The web.xml configuration may look like as shown here:

```

<servlet>
    <servlet-name>ODataEnabledJPAServlet</servlet-name>
    <servlet-class>
org.apache.cxf.jaxrs.servlet.CXFNonSpringJaxrsServlet
    </servlet-class>
    <init-param>
        <param-name>javax.ws.rs.Application</param-name>
        <param-value>
org.apache.olingo.odata2.core.rest.app.ODataApplication
        </param-value>
    </init-param>
    <init-param>
        <param-name>
org.apache.olingo.odata2.service.factory
        </param-name>
        <param-value>
com.packtpub.odata.ODataJPAServiceFactoryImpl
        </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>ODataEnabledJPAServlet</servlet-name>
    <url-pattern>/odata/*</url-pattern>
</servlet-mapping>

```

Now you can build the application and deploy it to a JAX-RS container such as GlassFish server.

To test the deployment, try accessing OData service as follows:

<http://localhost:8080/<appname>/odata>