

# Adding a Web Service Step by Step Guide for Core Developers (REST 1.x)

This guide walks a core developer through adding a new set of web service methods for a core object. It is a similar process for module developers, but for that setup, see [Adding a Web Service Step by Step Guide for Module Developers \(REST 1.x\)](#)

## Get the Code

The module can be checked out through maven at <https://github.com/openmrs/openmrs-module-webservices.rest>

You can build the module like any other mavenized module by doing an **mvn package** in the root module package.

## Read the Conventions

The first thing to do is to read through the conventions section on the [REST Web Services API For Clients](#). This outlines the very basics of the how to name the urls, design the objects, etc.

**Note:** When creating a REST resource and sub-resource URI and name respectively, the absolute and relative path names should be written in all-lowercase ASCII letters. Avoid capital letters, camel caps to mention a few. The norm is to use lower case letters.

Example:

```
/ws/rest/**/conceptclass
```

## Testing a URL

### Through the webapp

With the webservices.rest module installed, there is a **Test** page linked to from on the **Administration** page.

From this page you can make jquery POST/PUSH/GET/DELETE web services calls to the server and see the results.

The default values are to do a GET on a person. Check your database for a valid person UUID and try the query. (The post data does not matter in this case)

### From command line with CURL

A valid curl call with authentication:

```
curl -u admin:password http://localhost:8080/openmrs/ws/rest/person/b3f64e4c-860c-11e0-94eb-0027136865c4
```

## Adding new Web Service URLs

Our best-practices may change over the course as we learn new things, but [PatientResource](#) and <https://source.openmrs.org/browse/~br=trunk/Modules/webservices.rest/trunk/omod/src/main/java/org/openmrs/module/webservices/rest/web/controller/PatientController.java?hb=truePatientController> should always be up-to-date best-practice examples that you can use as patterns.

This example will use the **Location** object in the OpenMRS API.

### Create a new LocationResource object for the object

See also code for [LocationResource](#)

The resource objects are used for the automatic translation into json/xml.

1. In the **org.openmrs.module.webservices.rest.resource** package, create a new class that extends `MetaDataDelegatingCrudResource<Location>`.
  - a. public class `LocationResource` implements `MetaDataDelegatingCrudResource<Location>`
    - i. The `MetaDataDelegatingCrudResource` and `DataDelegatingCrudResource` class are helper classes for "data" and "metadata" `OpenmrsObjects`.
    - ii. The `DelegatingCrudResource` super class can be used for any other object type
    - iii. Its also possible to not have a resource class for some simple rest urls. See the `SessionController` class.
2. Add **@Resource("location")** annotation to the class.
  - a. This tells the framework to put the resource at uri location: `/ws/rest/location/uuid`
  - b. By convention, resource names are all lower case.
3. Add **@Handler(supports = { Location.class }, order=0)** annotation to the class.
  - a. This allows the webservices.rest module to easily and automatically find our resource.

- b. The module can now look for "any Handler for class *Location* of type *CrudResource*" and use that for the json/xml translation
- c. The "order" element means this class has lowest precedence and can be overridden by a module-provided resource for *Location*
- 4. Expose properties that are on the *Location* object through the resource:
  - a. Create a method named "DelegatingResourceDescription **getRepresentationDescription**(Representation rep)"
  - b. Switch on the "rep" argument (or use if/else)
  - c. Create a DelegatingResourceDescription according to the rep, using `addProperty` for the properties that you need
  - d. Tip: Use `description.addProperty("auditInfo", findMethod("getAuditInfo"))` for **not** creator/dateCreated/changedBy/dateChanged properties
  - e. Tip2: the `MetaDataDelegatingCrudResource` class defines the "ref" representation for you.
- 5. **DO NOT** add `locationId` to the `getRepresentationDescription` method. That is an internal number that should never be exposed over web services
- 6. Adding a new property (`fullAddress`) that is **not on the Location object** (only done here as an example. `Patient.name` is a better example)
  - a. There is no `Location.getFullAddress()` method, so automatic translation of this into json/xml will fail in the webservicess module.
  - b. Add `description.addProperty("fullAddress", findMethod("getFullAddress"))` to the `getRepresentationDescription` method
  - c. Add a method **SimpleObject getFullAddress(Location location)** on the **LocationResource** class.
    - i. This method is called anytime a user requests to see the "fullAddress" property in the json/xml
  - d. The implementation of the method can be anything using the *Location* object. Something like:
    - i. `return location.getAddress1() + " " + location.getAddress2() + " " + location.getCityVillage() + " " + location.getStateProvince()`
- 7. Add a `newDelegate()` method
  - a. This is used by the parent class to know how to construct our object (our delegate)
- 8. Add a `save(Location location)` method
  - a. This is used by the parent class to know how to persist our object to the database. This is typically just `Context.getSomeService().saveMyObject(object)`
- 9. Add a `getByUniqueId(String s)` method
  - a. The parent class calls this for all getters. You should look up your object by uuid. If your object can also be looked up by a unique name, you can do that in this method as well as the uuid.
  - b. This method is what allows your object to be a spring "Converter". So other WS methods can refer to your object by uuid/name as well
  - c. **DO NOT** look up by primary key. Primary keys should not be exposed over web services
- 10. Add a `purge(Location location, RequestContext context)` method
  - a. You can get the reason (if needed) out of the request context
- 11. Add a `doGetAll(RequestContext context)`
  - a. This should return a list of every location objects that is still valid (not retired)
  - b. The context object has parameters like "limit", "maxresults", "start", etc
- 12. Add `doSearch(String query, RequestContext context)`
  - a. This should return a list of location objects with a fuzzy search on the query.
  - b. The context object has parameters like "limit", "maxresults", "start", etc
  - c. There is a helper class named `ServiceSearcher` that you can use to help with the paging, index, etc
- 13. If you have a non-standard setter method:
  - a. **TODO**
- 14. Sub resources (like patient name)
  - a. **TODO**

#### Create the LocationController to house the REST URLs

See also code for the [LocationController](#)

The class for registering REST urls is a normal Spring annotated controller.

1. Create `LocationController` in the `org.openmrs.module.webservices.rest.web.controller` package and be sure to extend the **BaseCrudController** **er<LocationResource>** class.
2. Add the **@Controller** annotation to the class
  - a. This is so that Spring can find and register our controller
3. Add the **@RequestMapping(value = "/rest/location")** annotation to the class.
  - a. This is a shortcut so we don't have to put `"/rest/location"` into each one of our method annotations
  - b. It also allows us to use the parent class (`BaseCrudController`) to automatically define further urls
  - c. Our urls will actually be located at `"/openmrs/ws/rest/location"`
  - d. The mapping request should use the name of the resource (not the name of its class) and hence should be in all lower case
4. (Optional) Add the **public SimpleObject searchByAddress(@RequestParam("address") String address, HttpServletRequest request, HttpServletResponse response) throws ResponseException** method to allow for a custom search by address.
  - a. Annotate it with **@RequestMapping(method = RequestMethod.GET, params = "address")** and **@ResponseBody**.
  - b. Add the **@WSDoc("Fetch all-none retired locations with the given address")** annotation explaining the purpose of the method and the expected parameter.
5. That's it! The parent class deals with adding the GET/POST/DELETE methods for our location object with the right conventions