CHAPTER 8

Integrating Drupal 8

In many organizations Drupal web sites often provide and consume content and services to and from external systems. Those systems may be other web sites, enterprise applications, or third-party services. Drupal has historically provided the ability to integrate with external systems through a combination of contributed modules, which have often been fraught with complexities that made it difficult at best to integrate with Drupal. Drupal 8 changes all of that with the inclusion of RESTful web services in core and they work beautifully.

This chapter focuses on

- Enabling and configuring RESTful web services in Drupal 8
- Using views to expose content to external sources

Using RESTful Web Services in Drupal 8

Before venturing into the RESTful web services in Drupal 8, I first describe what RESTful web services are and why you may be interested in using them on your Drupal site. Representational state transfer (REST), or RESTful web services, is one way of providing interoperability between computer systems on the Internet, such as a Drupal web site, and other systems that may interact with Drupal. REST-compliant web services allow requesting systems to access and manipulate resources stored in a system using a uniform and predefined set of stateless operations. Other forms of web services exist, and they expose their own arbitrary sets of operations such as WSDL and SOAP, but Drupal has standardized on REST as the preferred means of supporting web services.

Web resources were first defined on the World Wide Web as documents or files identified by their URLs, but today they have a much more generic and abstract definition encompassing everything or entity that can be identified, named, addressed, or handled, in any way whatsoever, on the Web. Images, videos, documents, and content are just a few examples of resources that may be accessed, updated, created, or deleted through a web service. In a RESTful web service, requests made to a resource's URI elicit a response that may be in formatted as XML, HTML, JSON, or other defined formats. The response may confirm that some alteration has been made to the stored resource, and it may provide links to other related resources or collections of resources. The operations provided by a RESTful web service align with the standard HTTP verbs of `GET`, `POST`, `PUT`, and `DELETE`. By making use of HTTP, which is a stateless protocol, and HTTP's standard operations, REST systems aim for fast performance, reliability, and the ability to grow. They employ reused components that can be managed and updated without affecting the system as a whole, even while it is running.

© Todd Tomlinson 2017
T. Tomlinson, *Enterprise Drupal 8 Development*, DOI 10.1007/978-1-4842-0253-1_8

RESTful web services in a Drupal environment provide the ability to:

- Query Drupal for content (nodes, taxonomy, users, and comments) stored on a site
- Create new content
- Update existing content
- Delete content

Due to the nature of the actions, as a site administrator you can choose whether to restrict access to RESTful services through standard HTTP authentication methods, such as requiring a user ID and password of a user who has an account on the Drupal site in order to perform any or all operations.

The benefits of RESTful web services on Drupal is that it opens the door to a virtually unlimited number of opportunities to provide information to external systems as well as the ability to create and maintain content that is sourced from systems outside of Drupal. For example, a manufacturing company that uses an ERP system as the definitive source of truth for information related to products might use a RESTful web service to update product descriptions, inventory levels, and pricing on its Drupal site where they sell their products. Conversely, an organization that sells products through a network of distributors could provide a RESTful web service that provides real-time access to current product information. The opportunities are limitless; it only requires that the external system support REST.

RESTful Modules in Drupal 8 Core

Drupal 8 ships with the basic modules required to support RESTful web services. All you need to do is enable the modules and configure them to support the types of transactions you want to support on your site. The modules provided in core are as follows:

- *HAL*: Serializes entities using Hypertext Application Language
- *HTTP Basic Authentication*: Provides the HTTP Basic authentication provider
- *RESTful Web Services*: Exposes entities and other resources as the RESTful web API
- *Serialization*: Provides a service for (de)serializing to and from formats such as JSON and XML

To enable the modules, navigate to Extend and scroll down until you see the web services section. For demonstration purposes, we enable all four modules (see Figure 8-1).

WEB SERVICES

☐	HAL	Serializes entities using Hypertext Application Language.
☐	HTTP Basic Authentication	Provides the HTTP Basic authentication provider
☐	RESTful Web Services	Exposes entities and other resources as RESTful web API
☐	Serialization	Provides a service for (de)serializing data to/from formats such as JSON and XML

Figure 8-1. *The RESTful modules in core*

Drupal core provides the basic architectural components required to support RESTful web services; however, as of when this chapter was written, there is not a user interface for configuring and managing the services created by the core modules. There is a contributed module that provides these capabilities, called the REST UI module (`drupal.org/project/restui`). To facilitate the creation and management of RESTFul web services, download and enable the module.

After enabling the module, navigate to the Configuration page and you'll see a new entry in the web services section called Rest. Click on the link and you will see a page that describes the enabled services as well as the other available services that may be enabled (see Figure 8-2).

REST resources ☆

Home » Administration » Configuration » Web services

REST resources

Here you can enable and disable available resources. Once a resource has been enabled, you can restrict its formats and authentication by clicking on its "Edit" link.

Enabled

RESOURCE NAME	PATH	DESCRIPTION	OPERATIONS
Content	/node/{node}	methods authentication: basic_auth, cookie formats: hal_json, json, xml formats authentication: basic_auth, cookie formats: hal_json, json, xml authentication authentication: basic_auth, cookie formats: hal_json, json, xml	Edit

Disabled

RESOURCE NAME	PATH	DESCRIPTION	OPERATIONS
Action	/entity/action/{action}		Enable
Base field override	/entity/base_field_override/{base_field_override}		Enable
Block	/entity/block/{block}		Enable
Comment	/comment/{comment}		Enable
Comment type	/entity/comment_type/{comment_type}		Enable
Contact form	/contact/{contact_form}		Enable
Contact message	/entity/contact_message/{contact_message}		Enable
Content type	/entity/node_type/{node_type}		Enable
Custom block	/block/{block_content}		Enable
Custom block type	/entity/block_content_type/{block_content_type}		Enable
Custom menu link	/admin/structure/menu/item/{menu_link_content}/edit		Enable
Customer	/admin/structure/customer/{customer}		Enable
Date format	/entity/date_format/{date_format}		Enable

Figure 8-2. *The list of available off-the-shelf services*

Retrieving Content Through REST

With the basics in place, this section demonstrates retrieving a node through a web service before configuring additional capabilities. To demonstrate accessing the services via REST, we need a tool that allows us to make HTTP `GET` requests. The Chrome Postman extension (`getpostman.com`) is an easy-to-use tool for performing REST operations. There are dozens of other tools for Chrome, Safari, and Firefox. Use the tool that you're most comfortable with. I'll use Postman throughout this chapter.

To execute a `GET` request, we use a node on the Drupal 8 instance, e.g., `node/1`, and in Postman, we use the URL of `node/1?_format=hal_json` to retrieve the JSON-formatted output from the RESTful web service. The result after executing the `GET` request is shown in Figure 8-3.

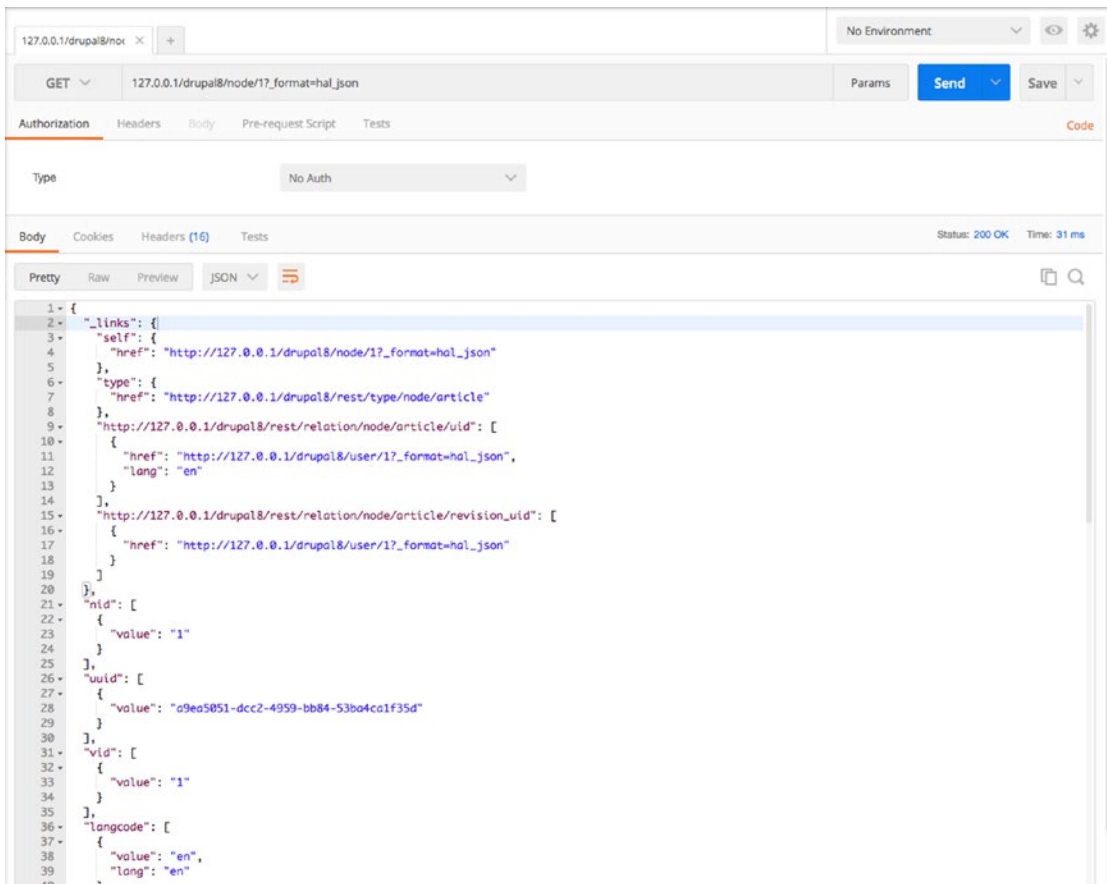

Figure 8-3. *The JSON results of a GET request*

Creating a Node Through REST

With the ability to connect to REST on Drupal 8 and retrieve content through the RESTful services, let's take the next step and configure REST to accept POST requests so that we can create and update content on the site.

The first step is to enable POST on the Content resource. By default, POST is disabled to protect the resources on your site from potentially unauthorized access. To enable POST, navigate to Configuration ➤ Services ➤ REST and, on the REST resources page, click on the Edit button for the Content Resource (see Figure 8-2). On the Settings for Resource Content page, check the box for POST and the boxes for hal_json, json, and xml in the Accepted Request Formats section, as well the basic_auth box in the Authentication Providers section (see Figure 8-4). Then click the Save Configuration button at the bottom of the page to complete the process.

Settings for resource *Content* ☆

Home » Administration » Configuration » Web services » Rest

Here you can restrict which HTTP methods should this resource support. And within each method, the available serialization formats and authentication providers.

☐ GET

Accepted request formats

☐ hal_json
☐ json
☐ xml

Authentication providers

☐ basic_auth
☐ cookie

☑ POST

Accepted request formats

☑ hal_json
☑ json
☑ xml

Authentication providers

☑ basic_auth
☐ cookie

Figure 8-4. *Enabling POST*

With POST enabled, it's time to create a new article node. Using Postman, we can set the following values:

- *Authorization*: Basic Auth and we use the user ID and password of a test user that we set up on the Drupal 8 site that has the administrator role.
- Headers:
    - `Authorization` is automatically set up for you when you enabled basic authorization in the previous step.
    - `X-CSRF-TOKEN` with a value that can be found by visiting `example.com/rest/session/token` (replace `example.com` with the domain name of your site). This is a secure token that provides another level of security on your site to external resources performing updates via REST.
    - `Content-Type` should be set to a value of `application/hal+json` (valid options are hal+json, json, or xml, depending on what you enabled when you set up POST on `admin/confg/services/rest`.

- The Body value will be a valid JSON-formatted object that maps to the fields required to create an article. For demonstration purposes, we create an article with a `title` and body:

```
{
  "_links": {
      "type": {
          "href":"http://127.0.0.1/drupal8/rest/type/node/article"
       }
   },

   "title":[{
      "value":"My new article created through REST"
   }],

   "body":[{
     "value": "This is an article body that was created through the REST POST
     method"
   }],

   "type":[{
     "target_id":"article"
   }]
  }
```

We paste the JSON object into the body field on Postman.

The next step is to perform the post. Select POST from the list of available methods and enter the URL on the site that is used by REST to create a new node. In this case:

http://127.0.0.1/drupal8/entity/node?_format=hal_json

Note the addition of `?_format=hal_json` to the end of the URL. This instructs REST on Drupal to process the incoming POST as a hal+json request.

After you click the Send button, Postman responds with the status (in this case `201 Created`) and the results that were returned from the POST (see Figure 8-5).

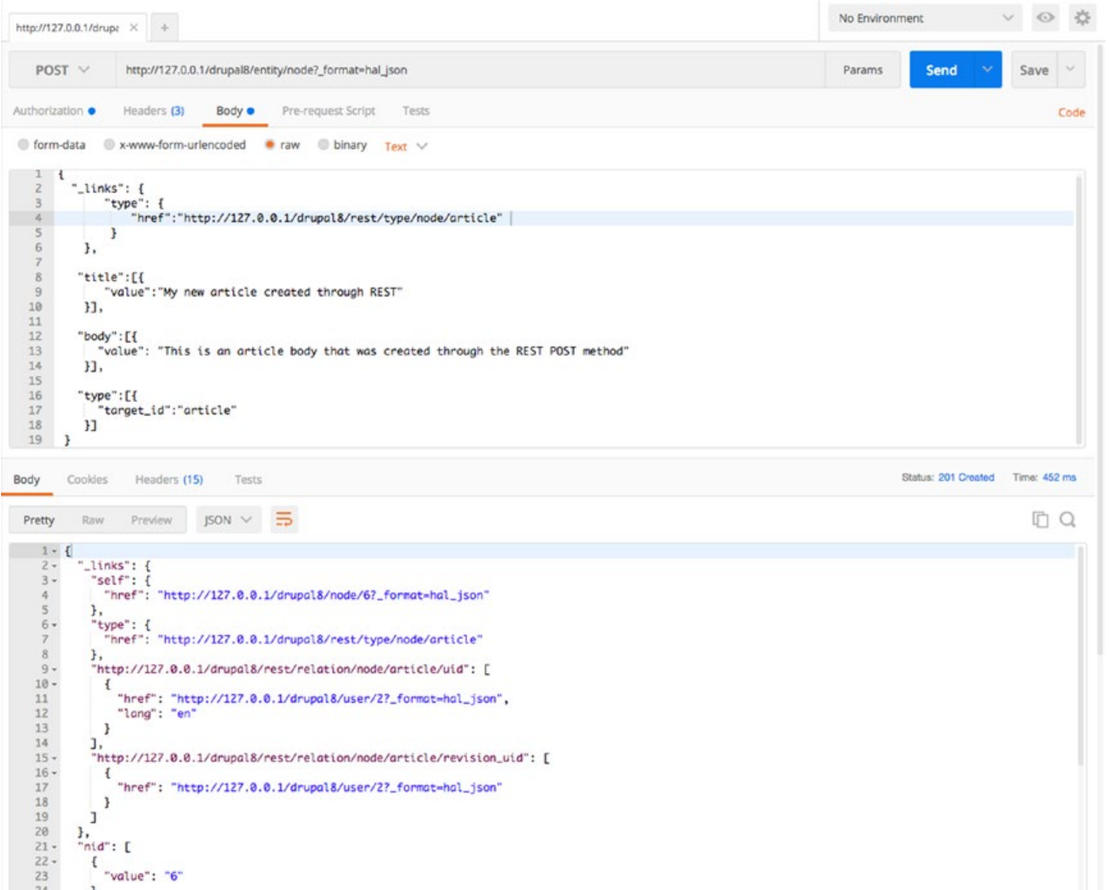

Figure 8-5. *Creating a node through Postman*

To verify that the article was successfully created, navigate to Content to see if the new node is listed. As shown in Figure 8-6, it appears in the list with the title that we created through the POST.

	TITLE	CONTENT TYPE	AUTHOR	STATUS	UPDATED	OPERATIONS
☐	My new article created through REST	Article	test	Published	01/02/2017 - 10:10	Edit
☐	Sample Article	Article	admin	Published	01/01/2017 - 19:47	Edit

Figure 8-6. *The new article appears in the list*

If you view the new article, you'll see that the body content was also successfully created (see Figure 8-7).

Figure 8-7. *The new article was created through REST with a title and body*

Updating and Deleting a Node Through REST

You can also update existing nodes and delete nodes using REST. To perform updates, you need to enable the PATCH method on the REST configuration page for content. Navigate to Configuration ➤ REST and, on the REST Resources page, click on the Edit button for the Content resource. On the Setting for Resource Content, check the boxes for PATCH and the Request Formats and Authentication Providers, as shown in Figure 8-8.

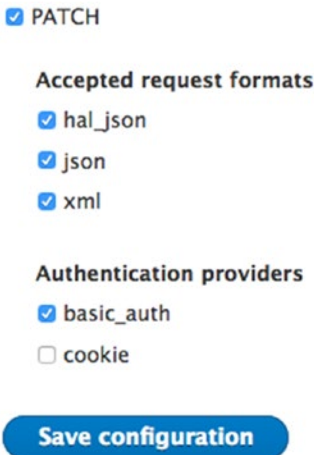

Figure 8-8. *Enabling PATCH*

With PATCH enabled, the next step is to return to Postman to specify the changes that you want to make to your article. We can update both the `title` and body fields by changing the word "added" to the word "updated," as shown in the following JSON object.

```
{
  "_links": {
       "type": {
           "href":"http://127.0.0.1/drupal8/rest/type/node/article"
        }
    },

   "title":[{
       "value":"My new article updated through REST"
    }],
```

```
  "body":[{
    "value": "This is an article body that was updated through the REST POST method"
  }],

  "type":[{
    "target_id":"article"
  }]
}
```

Now update the URL in the Postman interface to reflect that we want to update the node with a node ID of 6 and change the method to PATCH, as shown in Figure 8-9.

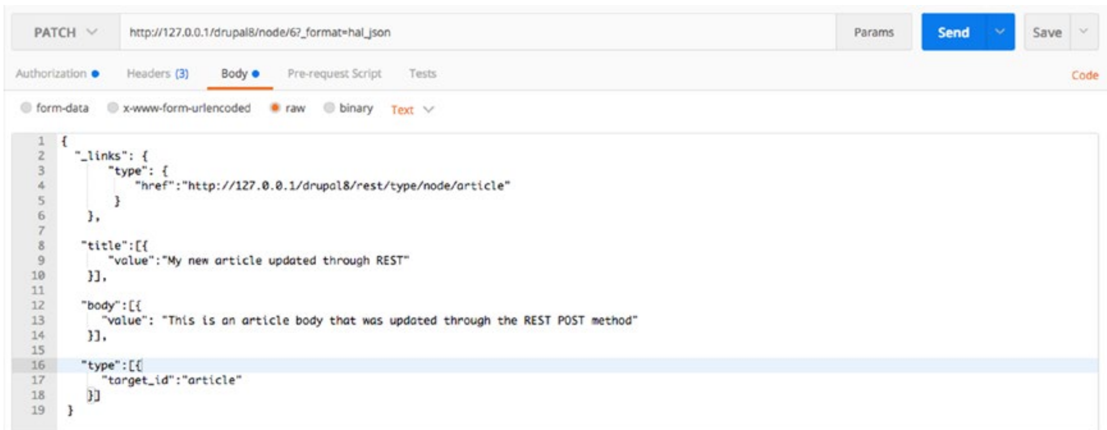

Figure 8-9. *Using PATCH in Postman to update an existing node*

After clicking the Send button, visit your Drupal 8 site and view the existing node. You can see that the changes were successfully made (see Figure 8-10).

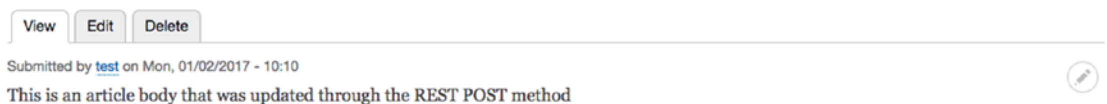

Figure 8-10. *The updated article*

To delete an existing content item, first navigate to Configuration ➤ REST and update the Content Resource to accept the Delete method. Follow the same steps as we did for PATCH, checking the same set of options.

After enabling Delete, return to Postman and simply change the method from PATCH to Delete, leaving the URL as it was for PATCH. You should clear out the body field in Postman, as no values are required. Click the Send button and then visit your Drupal 8 site, where you'll see that the node you previously created is now deleted.

Using REST for Other Entity Types

The examples in the previous section focused on using the REST interface to perform actions on content. You may also use REST to perform similar actions on other entities and objects on a Drupal 8 site. Visit the `admin/config/services/rest` page to see a list of all the other available resources, such as taxonomy terms, comments, blocks, menus, etc., that may be accessed through REST. To use REST to query, update, and delete taxonomy terms, for example, enable the Resource taxonomy term and configure the various methods.

Generating Lists of Content Using Views and REST

The previous sections demonstrated using `GET` to retrieve a single content item from Drupal through REST. Retrieving a single content item is a valid use case; however, a more common use case is to retrieve a list of content items, for example, a list of all articles on a Drupal site. The process for creating a list-based RESTful web service is to employ views as the mechanism for generating the list and for responding to the web services request.

For demonstration purposes, we use the Devel module to generate 50 articles on our Drupal 8 site. If you haven't used the Devel module and its content-creation tools, I suggest that now would be a good time to try it out. Download and install Devel and all of its submodules from `drupal.org/project/devel`. After downloading and enabling the Devel and Devel Generate modules, navigate to Configuration ➤ Development ➤ Generate content. Check the box to Generate Articles and leave the defaults for the remainder of the options. Finish the process by clicking the Generate button at the bottom of the page. You can verify that the articles were created by visiting the Content page, where you'll see a long list of articles.

With the content in place, it's time to create the view. Navigate to Structure ➤ Views and click on the Add View button. We name the view RESTful Article List and update the settings to Show Content of type Article and sorted by Unsorted. Leave the options Create a Page and Create a Block unchecked. Check the Provide a REST export option at the bottom of the page and enter `rest/articles/list` in the REST export path. You can continue the process by clicking the Save and Edit button (see Figure 8-11).

Figure 8-11. *The articles list REST View*

On the RESTful article list (content) page, you can already see that the view is generating JSON objects for the articles on the Drupal 8 site, but there are a few changes that we need to make before saving the view and testing it through Postman. The first change is to enable basic authentication so that access is restricted to those who have permissions to view articles through REST. To enable authentication, click the No Authentication Is Set option in the second column of the view in the Path Settings section. After clicking the list, Drupal displays the available authentication methods—basic_auth and user. For this example, we click the basic_auth option and then save the changes by clicking the Apply button.

The second change that we need to make is to remove the limit of only 10 articles returned by the view. Click the Display a Specified Number of Items option in the Pager section, selecting Display All Items in the List of Options. Then click the Apply button to update the view. The practice site has a limited number of articles, so returning all articles won't create a performance issue. If you have a site with a large number of articles, you may want to consider limiting the number returned by the view.

After making the changes, click the Save button. The view at this point is ready to use and is configured, as shown in Figure 8-12.

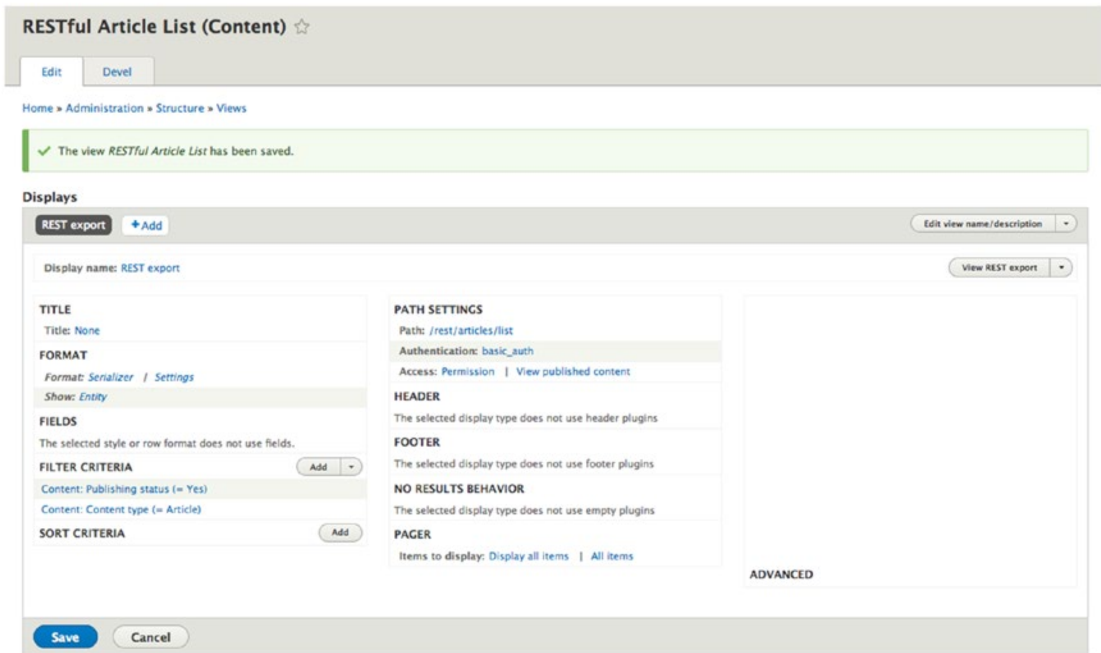

Figure 8-12. *The article listing RESTful web services view*

Return to the Postman tool and leave the authorization and header values as they were in previous examples. (Note: If you left the Postman tool and are returning, you'll need to re-enter the values for authorization and headers before continuing the process. See the previous example for creating a `POST` request for an article for the appropriate values.) We need to update the URL to reflect the URL set in the view, which is in this case is `http://127.0.0.1/drupal8/rest/articles/list?_format=hal_json` (replace `http://127.0.0.1/drupal8` with the appropriate value for your site). Then we update the method to `GET` and click the Send button to retrieve the values from the view (see Figure 8-13).

GET http://127.0.0.1/drupal8/rest/articles/list?_format=hal_json Params Send Save

Authorization Headers (3) Body Pre-request Script Tests Code

	Key	Value
☑	Authorization	Basic dGVzdDp0ZXN0
☑	X-CSRF-Token	RxQ7nUu_OnUvEplgNmBCtbaBalv-xw8Slxza_QckrrM
☑	Content-Type	application/hal+json
☑	key	value
	key	value

Body Cookies Headers (15) Tests Status: 200 OK Time: 1485 ms

Pretty Raw Preview JSON


```
[
  {
    "_links": {
      "self": {
        "href": "http://127.0.0.1/drupal8/node/7?_format=hal_json"
      },
      "type": {
        "href": "http://127.0.0.1/drupal8/rest/type/node/article"
      },
      "http://127.0.0.1/drupal8/rest/relation/node/article/uid": [
        {
          "href": "http://127.0.0.1/drupal8/user/2?_format=hal_json",
          "lang": "en"
        }
      ],
      "http://127.0.0.1/drupal8/rest/relation/node/article/revision_uid": [
        {
          "href": "http://127.0.0.1/drupal8/user/2?_format=hal_json"
        }
      ],
      "http://127.0.0.1/drupal8/rest/relation/node/article/field_image": [
        {
          "href": "http://127.0.0.1/drupal8/sites/default/files/2017-01/generateImage_bWoNFD.gif",
          "lang": "en"
        }
      ],
      "http://127.0.0.1/drupal8/rest/relation/node/article/field_tags": [
        {
          "href": "http://127.0.0.1/drupal8/taxonomy/term/1?_format=hal_json",
          "lang": "en"
        }
      ]
    },
    "nid": [
      {
        "value": "7"
```


Figure 8-13. *The results of executing the article listing RESTful view*

You can expand on the capabilities of this view by adding contextual filters to restrict the list of nodes to specific criteria. For example, I could update the view to accept a contextual filter of the content ID and limit the response only to that node with that ID. To do so, you add a contextual filter of ID, as shown in Figure 8-14.

Configure contextual filter: Content: ID

This display does not have a source for contextual filters, so no contextual filter value will be available unless you select 'Provide default'.

WHEN THE FILTER VALUE IS *NOT* AVAILABLE

Display all results for the specified field
Provide default value
Type
Content ID from URL
Show "Page not found"
Display a summary
Display contents of "No results found"
Display "Access Denied"

EXCEPTIONS

Skip default argument for view URL
Select whether to include this default argument when constructing the URL for this view. Skipping default arguments is useful e.g. in the case of feeds.

WHEN THE FILTER VALUE *IS* AVAILABLE OR A DEFAULT IS PROVIDED

Override title
Specify validation criteria

ADMINISTRATIVE TITLE

MORE

Apply Cancel Remove

Figure 8-14. *Adding a contextual filter to the RESTful view*

After updating the view, return to the Postman interface and update the URL to include a node ID of an article on the Drupal 8 site. We execute the request by clicking the Send button and we will see in the results that only the node with an ID of 8 was returned in the results (see Figure 8-15).

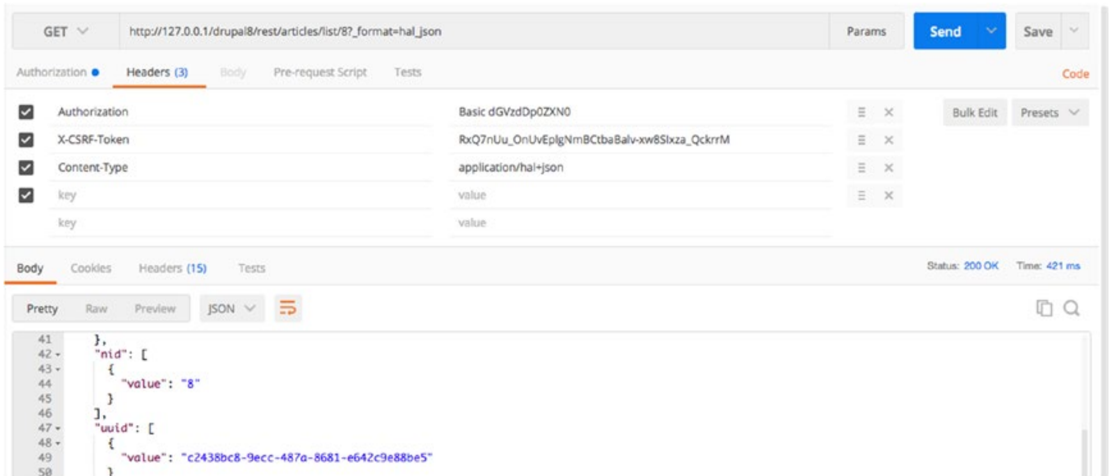

Figure 8-15. *Executing a limited search through the RESTful view*

You could expand on this example to restrict the list to articles tagged with a specific taxonomy term or any other criteria that your use case requires. You may also create views to generate lists of other content types, as well as any other lists you can create using views.

Generating Output in Other Formats

In the previous examples, we used `hal_json` as the format that was returned by the RESTful web service. Views also provide the ability to export results in JSON and XML. To change the format, visit the view and click on the Settings link in the Format section. Check the boxes shown on the Rest Export: Style Options page (see Figure 8-16).

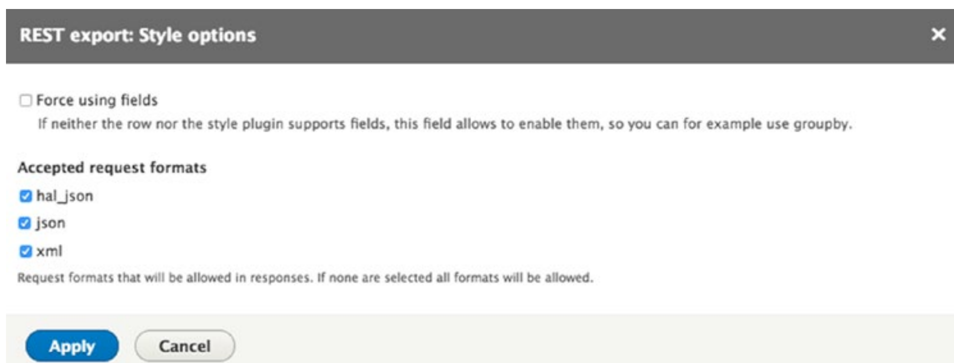

Figure 8-16. *Other supported output formats*

After changing the supported formats, return to Postman and update the URL to reflect the different formats. We can test the XML output first by changing the end of the URL from `_format=hal_json` to `_format=xml`. The resulting output is shown in Figure 8-17.

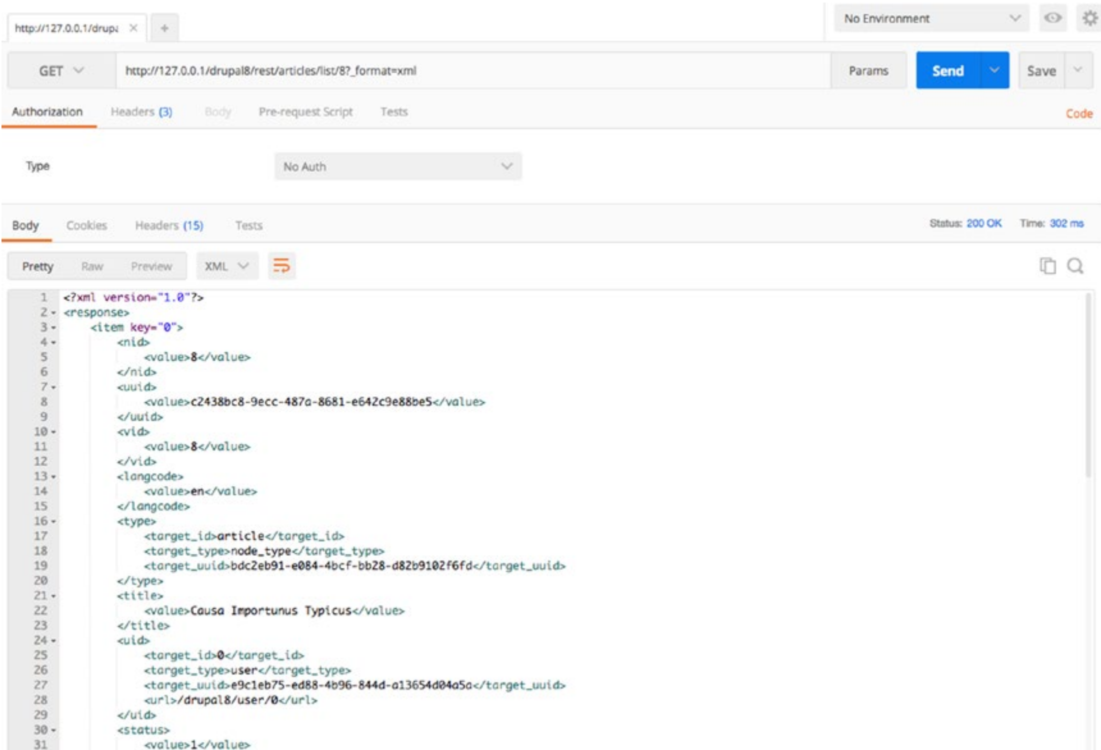

Figure 8-17. *Output from the view as XML*

Using Views to Expose Content to External Sources

The previous section demonstrated using views and REST to expose lists of content to a REST client through a RESTful web service. While REST is a prevailing standard and is supported by nearly every platform in the market, there may be instances where REST isn't possible and a simplified approach for consuming content from your Drupal 8 site is required. The solution in this case has been around for years and that is generating RSS or OPML feeds with views. with RSS (Rich Site Summary or Really Simple Syndication) and OPML (Outline Processor Markup Language). The client imply needs to be able to consume the output generated by visiting a URL.

Creating an RSS or OPML-based view is relatively simple. Go to Structure ➤ Views and click on the Add View button. On the Add View page, we enter `feeds` as the name of the view and leave the rest of the page set to the default values. Click the Save and Edit button to continue.

On the feeds (content) page, we have to make only a few changes in order to generate an RSS feed:

- In the Displays section, click the Add button and select Feed from the list of options.
- In the Feed Settings section, update the path by clicking on the No Path Is Set option. For demonstration purposes, we enter feeds/content.
- We change the page from Display a Specified Number of Items |10 items to Display All Items then save the changes.

After saving the view, we visit the URL we entered in the path settings and will see the output of the view (see Figure 8-18).

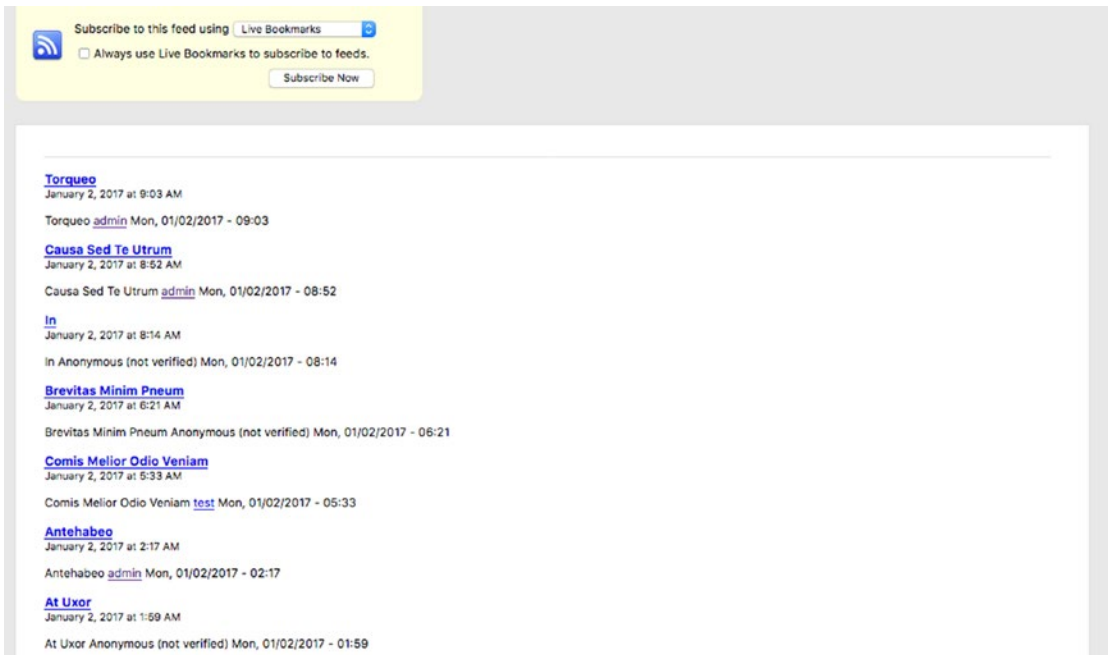

Figure 8-18. *The output of an RSS feed-based view*

To generate an OPML-based feed, simply change the Format value from RSS Feed to OPML.

You may also want to explore the views data export module (`drupal.org/project/views_data_export`) as an alternative to REST, RSS, and OPML. Views data export provides the ability to generate CSV, XLS, DOC, TXT, and XML files from views.

Creating Custom RESTful APIs

The previous examples demonstrated using off-the-shelf Drupal modules for providing RESTful APIs. There may be instances where the Drupal 8 Core REST modules and views do not provide you with the functionality you need and custom development is the only option. This simple example demonstrates how to create a custom RESTful web service that responds to a `GET` request. While a simple example, it demonstrates the skeleton of a custom module that you could then expand upon to meet your specific needs.

Creating the Custom Module

We create the new custom module, `demo_rest_api`, in the `/modules/custom` directory on our Drupal 8 site. Set up the directory structure as follows:

```
└── demo_rest_api
    └── src
        └── Plugin
            └── rest
                └── resource
```

The first file that we create is the .info.yml file for the module. In the module's root directory, create a file named demo_rest_api.info.yml and, in that file, place the following code:

```
name: demo_rest_api
type: module
description: A demo module that creates a REST endpoint
core: 8.x
package: Custom
```

The next file that we create is a plugin to handle the REST API that the module will provide. A plugin is a small piece of functionality that may be swapped in and out of your Drupal site. Plugins that perform similar functionality are called plugin types. For more information on plugins, visit drupal.org/docs/8/api/plug-api.

Plugins are stored in the src/Plugin directory and are grouped by plugin type. In this case, the plugin that we're creating is for REST so we'll create a subdirectory in the Plugin directory named rest. In the src/Plugin/rest directory, we'll create a resource directory, which is where the actual plugin code will reside. We call this plugin DemoResource and define it in a file named DemoResource.php.

Place the following code in the src/Plugin/rest/resource/DemoResource.php file:

```
<?php

namespace Drupal\demo_rest_api\Plugin\rest\resource;

use Drupal\rest\Plugin\ResourceBase;
use Drupal\rest\Plugin\ResourceInterface;
use Drupal\rest\ResourceResponse;

/**
  * Provides a Demo Resource
  *
  * @RestResource(
  *   id = "demo_rest",
  *   label = @Translation("Demo Rest endpoint"),
  *   uri_paths = {
  *     "canonical" = "/demo/rest"
  *   }
  * )
  */

class DemoResource extends ResourceBase {

  /**
   * Responds to entity GET requests.
   * @return \Drupal\rest\ResourceResponse
   */
  public function get() {
    $response = ['myresponse' => 'Hello, this is a rest service response from Drupal 8'];
    return  new ResourceResponse($response);
  }
}
```

The code is relatively straightforward:

- We specify the namespace so Drupal knows where the `DemoResource` class resides.
- We include the components that we need to construct the class from `Drupal\rest\`.
- Next, in the `docblock`, we specify the ID of my `RestResource`, which is the label that appears in the RestUI interface, and the path that the RESTful API can be accessed from.
- The class `DemoResource` extends the base class of `ResourceBase`, which is part of the REST architecture included with Drupal core.
- The single function that we provide is `get()`. This function does one thing, it formats a response message that is sent back to the client that called the function. We could also provide other functions such as `delete` and `patch`.

After saving the files, navigate to the Extend page and enable the module. After enabling the module, assuming you have installed the RestUI module, go to Configuration ➤ REST. Scan through the list of resources until you find your Demo REST endpoint (as defined in the docblock in the `DemoResource.php` file). Enable it and then configure the endpoint, as shown in Figure 8-19.

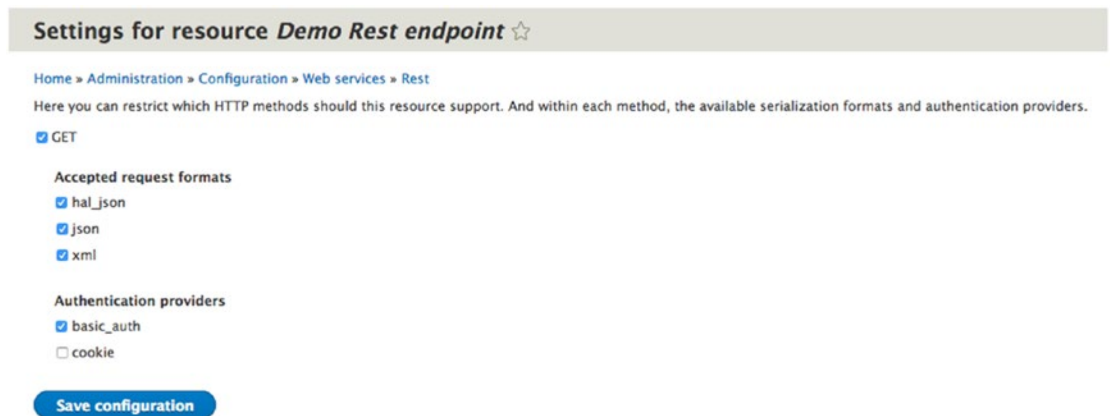

Figure 8-19. *Configuring the Demo REST endpoint*

With the RESTful API enabled and configured, you're ready to test it. We use Postman as the means for testing the endpoint. Set up the headers by specifying the `Content-Type` as `application/json` and set up basic authorization where you specify the user name and password from your Drupal 8 site. Then enter the appropriate URL—in this case `example.com/demo/rest?_format_json` (replace `example.com` with your site's domain name). Choose `GET` from the list of methods to execute and send the request. The results of this test are shown in Figure 8-20. The response is `"myresponse"` with the value as set in the `GET` function.

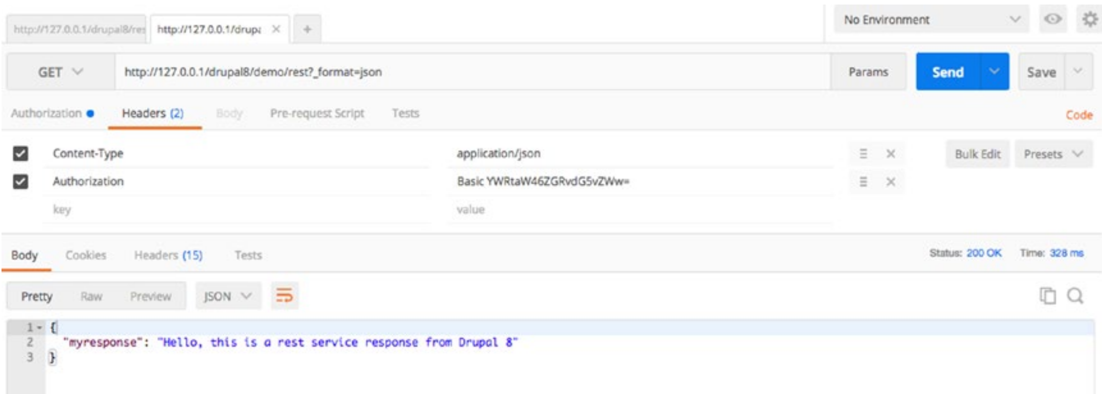

Figure 8-20. *The response from the Get request*

While it is a simple example, it demonstrates the minimum viable code required to create a custom RESTful API in Drupal 8. You may use this as a template to create custom services that meet your specific requirements.

Other Integration Options

There are other integration options that provide you with the capabilities required to support unique needs, such as importing content into your new site through a module such as the Feeds module (`drupal.org/project/feeds`). While commonly used to import content during the process of migrating a site to Drupal, the Feeds module also provides the ability to run periodic imports of content from external third-party services (via URL and a structured feed such as XML, or via a comma-separated value—CSV—file).

If Feeds does not meet your needs, you may also consider writing a custom module that consumes as RESTful web service from another source and performs the required transformations on that information before storing it in the Drupal database.

Summary

Drupal 8's off-the-shelf RESTful web services capabilities provide an easy-to-use approach for integrating your Drupal site with other web sites, enterprise applications, mobile applications, and third-party services. You may choose to use the off-the-shelf capabilities of Drupal core or you may want to write your own custom web services using the capabilities in Drupal core as the foundation for your custom module. The options and opportunities are virtually limitless.

The next chapter explores improving the user experience for your site administrators and content authors, often referred to as the *forgotten* users. They are typically left to the end of the project when there is little budget, time, and resources. Creating a usable backend offers a payback of potentially huge dividends over the life of your web site.