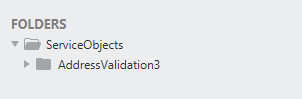
Magento is an extensible and powerful e-commerce platform that allows businesses to sell their products to consumers with a spiffy website and some nifty tools to drive it. The other nice thing about Magento is that nearly every aspect of it is customizable so that you can implement your specific business logic and rules to tailor the customer experience to your liking. Because of this, Service Objects products and Magento are a great match and can be easily integrated into your Magento Platform. We’ve done some of the initial legwork to illustrate how to integrate a web service call to one of our DOTS products. In this guide, we’ll illustrate how to call our DOTS Address Validation 3 service with a custom module in Magento. In order to follow along with this tutorial, you will need the following:

* A functioning Magento Instance (we’re using a fresh install of Magento 2.4 for this example)
* Working knowledge of Magento and the technologies it utilizes: PHP, Javascript, MySql etc
* A DOTS license key. Get a free key [here](https://www.serviceobjects.com/get-free-trial-key). We’ll be using a DOTS Address Validation 3 license key for this example.

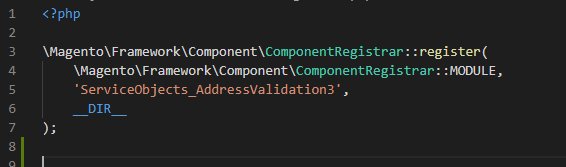
If you’re not familiar with the basics of creating a custom module in Magento, there is a piece of [documentation](https://devdocs.magento.com/videos/fundamentals/create-a-new-module/) that outlines the basic steps and process involved; we recommend you give it a read as it will show the general steps of creating a custom module. Part of our custom module will be a [plugin](https://devdocs.magento.com/guides/v2.4/extension-dev-guide/plugins.html) that fires before an address is saved into the database. We’ll highlight the different pieces necessary in the below tutorial. A link to the custom module we created will also be available at the end of this write up.

**Folder Structure**

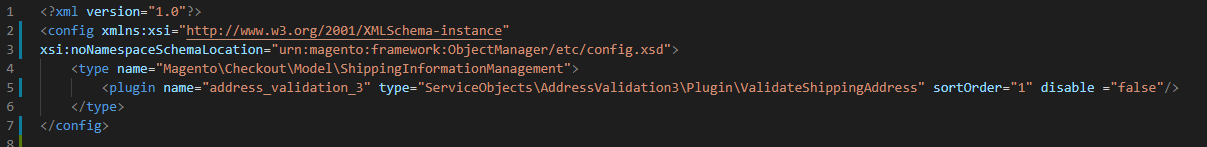
A big part creating a custom module is getting the folder structure right. For our custom module we’ll start by creating the following file structure



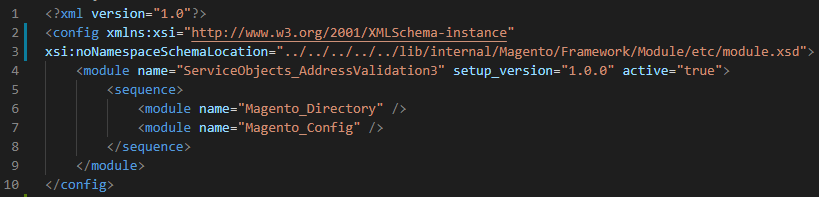
In this case, the text “ServiceObjects” is our Vendor name folder and AddressValidation3 is the name of our custom module. Next we’ll create a registration.php file in the AddressValidation3 folder. This file tells the Magento instance that we want to register a custom module. We fill the registration.php file with the following contents:



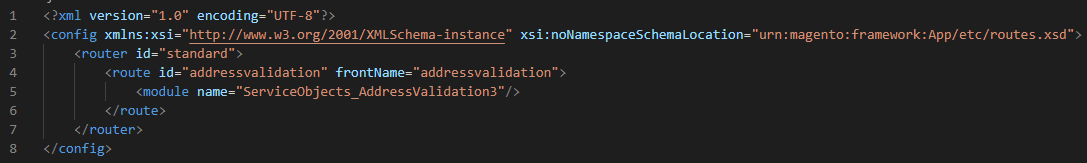
Next, we’ll create a folder called “etc” under the AddressValidation3 folder. Within this folder we are going to create two xml files, di.xml and module.xml The di.xml file will tell Magento which dependencies will need to be injected by the object manager. In our case this will be the ShippingInformationManagement class that is used by the default Magento\_Checkout module. This will also tell Magento to call our custom php file to extend the ShippingInformationManagement class. Note in the xml below we list the ValidateShippingAddress.php file and we will create that in a moment. We will add the following the di.xml file:



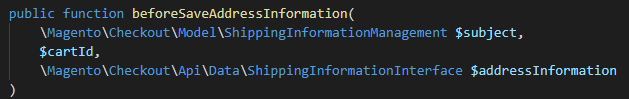
The module.xml file will tell Magento which will tell Magento what version of the custom module is being installed, along with the module name, and any dependencies that the module may need.



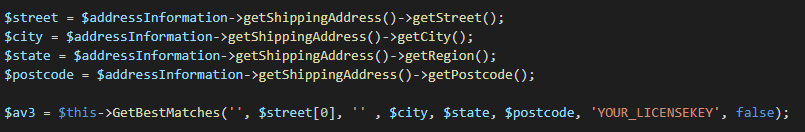
We’ll also need to create a routes.xml file for the custom controller we’ll need to build in the later part of this tutorial. To do this, create a folder named “frontend” under the etc folder and create a file called routes.xml within it. You can put the following contents in it:



Now that the proper xml files are defined, we can create the ValidateShippingAddress.php file and associated method that will be called when a user attempts to save a shipping address. To start, we will create a folder called “Plugin” under the AddressValidation3 folder and within the Plugin folder the ValidateShippingAddress.php will be placed. Within the created php file we will create a method called beforeSaveAddressInformation. For this case we are calling out method *before* the saveAddressInformation method gets called in the default Magento Checkout method. The signature of the beforeSaveAddressInformation method must match the original saveAddressInformation. Our method signature will be as follows:

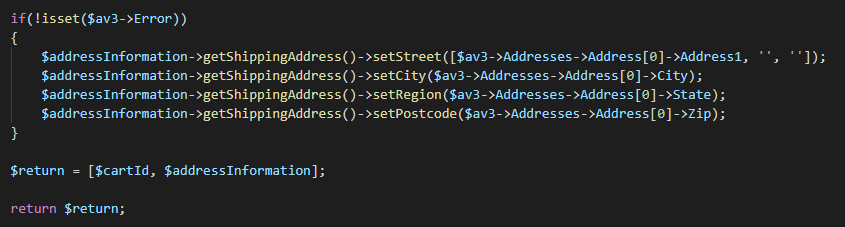


The $addressInformation object will ultimately hold the values that we want to validate, and we’ll need to access those to send them to the DOTS Address Validation 3 service. To do this, we’ll pull out the shipping address information from this object and send them to the DOTS AV3 service using our [php class library](https://www.serviceobjects.com/blog/ease-of-integration-with-code-libraries/).

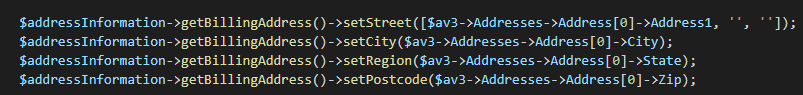


Our class libraries are used to call our webservices using our recommended best practices. This includes [failover configuration](https://www.serviceobjects.com/support/failover-configuration) to ensure maximum uptime in the event that the primary Service Objects data center is not reachable or producing unexpected results.

Now that the code has called the AV3 web service, we can set the updated values and return the result to continue normal operation of the default Checkout Module

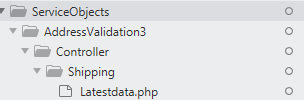


Note: In the code mentioned above, we simply check if the AV3 service returned an error. If it did return an error, we will skip updating the address as there would be nothing to update. This would be a great place to put in some custom business logic that applies to your specific application. For example, if you want to make use of the different [DPV](https://docs.serviceobjects.com/display/devguide/AV3+-+DPV+Codes%2C+DPV+Notes%2C+and+Corrections) values our service returns you could relay to the front end user that an address was missing a valid apartment number if the service returned a DPV 3 or 4. In this case, we’re just updating the shipping address but we could just as easily update the billing address with the following code if desired:



Now that we have returned the updated data in our plugin, the validated shipping information will be saved in the database.

Now we can move onto putting in a custom controller that will allow our updated address information to be sent to the frontend, so the updated shipping address is displayed on the final checkout page. To do this we will create a folder called “Controller” in the AddressValidation3 folder and within the Controller folder we’ll create another folder called “Shipping”. Under the Shipping folder we will create a file called Latestdata.php which will be an endpoint called by the front end to return the updated address information from the database. The folder structure should be as follows:

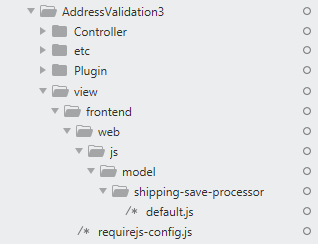


Within the Lastestdata.php file we can put the following code which return return a JSON object with the updated address information for the frontend.

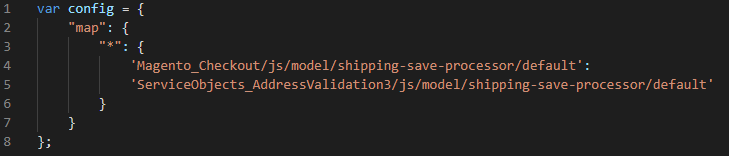


This code looks up the updated address information by the active $cardId field and returns the updated shipping address.

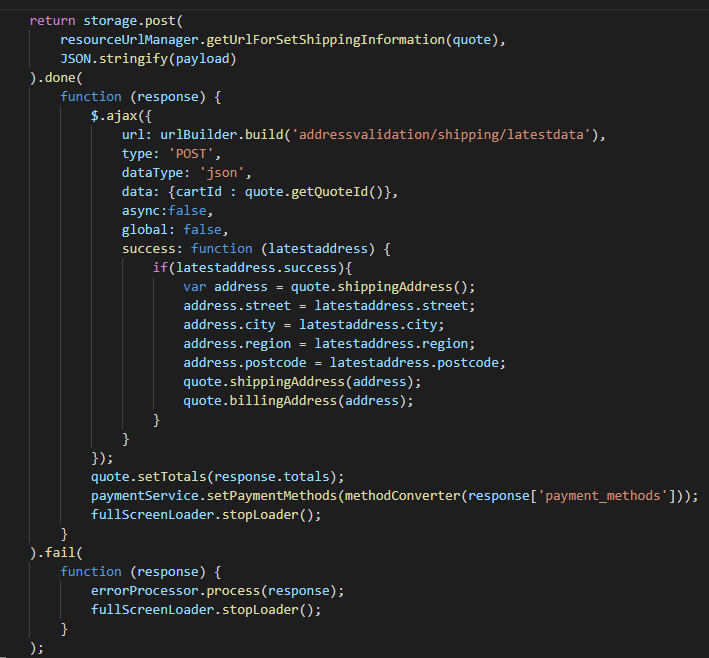
For this code to be called, we’ll need to add a call to this controller on the front end by overriding some of the JavaScript that handles the rendering of the address. To do this we will need to create some more folders and sub folders under the AddressValidation3 folder. Your folder structure should look like the following:



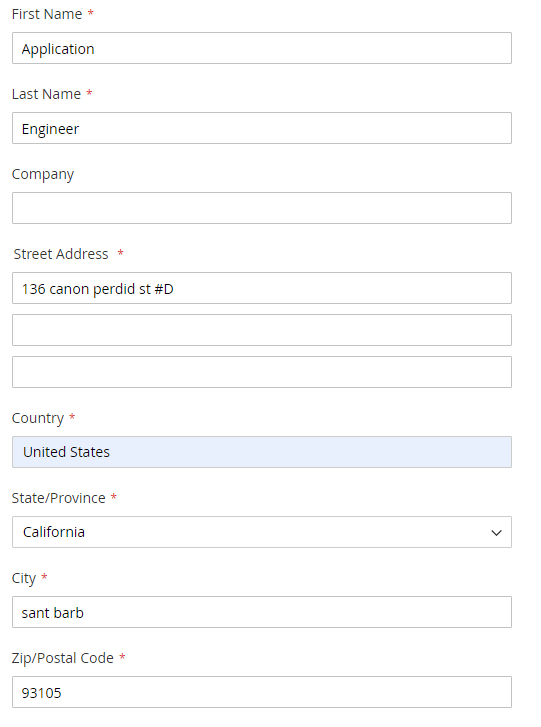
The requires-config.js file is under the frontend folder and tells Magento what default .js file that we want to overwrite. In this case it is the default.js file under the shipping-save-processor folder. The requires-config.js file should contain the following contents for our tutorial:



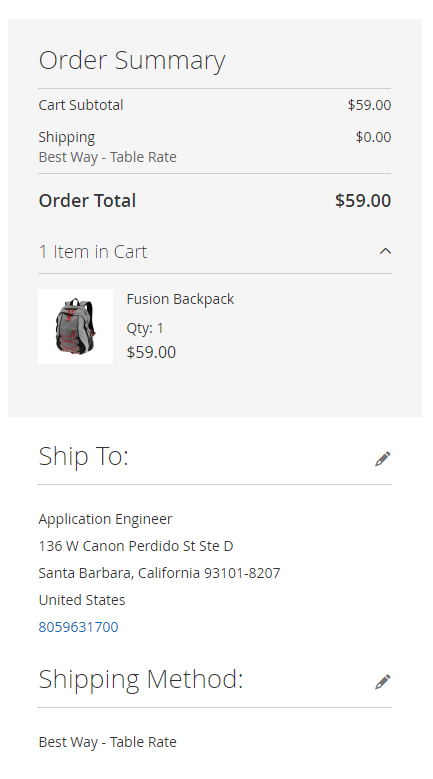
In the updated default.js file we have in our project; we’ll add the following bit of code that will call the custom controller we just created and set the shipping address on the output.



Now that we have everything in place, we’re ready to install the custom module and show off the results! To install the custom module, place custom module in the app/code/ directory of your Magento instance and run the “bin/magento setup:upgrade” command to have Magento install the module. If everything goes smoothly, your module is ready to test out! To do this we’ll just do a quick order of one of the default products that come with the sample data installation of Magento. On the shipping address screen we’ll enter Service Object’s office address that will include some errors and unstandardized versions of the address.



After we select next and go to the next screen, we will see that the input address we had has been corrected and standardized in the checkout screen. Notice the correction to the city, zip code and street name correction.



Viola! Now we have some validated address info on the output and our custom Magento module is finished! As mentioned earlier in the tutorial, it can be downloaded here***{insert download link for the code}*** Your specific integration will likely be different and may require some special consideration; please reach out to us if you have any questions and we’ll be happy to make recommendations for your specific integration needs!