
salesforce
Fast Path to B2C Commerce
Developer Certification
Module 2
Cartridges & Controllers

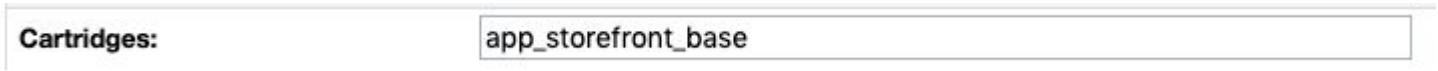
Module 2.1: Cartridges

- Objective
  - By the end of this module you will be able to ***define*** what a cartridge is, ***understand*** SFRA installation options, and ***understand*** the usage of the cartridge path
  - This module will cover:
    - SFRA base cartridge overview
    - Cartridges SFRA + All Optional Features installation option
    - Cartridge path meaning and usage
- What is this functionality and why does it matter for implementations?
  - Cartridges are the way code is uploaded to a B2C instance
  - Cartridge path affects how code gets executed in the site
  - SFRA provides two starting points: base site or complex

Module 2.1: What is a cartridge?

The main container to deliver code

- Contains directories specific to functionality that we will discuss during the webinars.
- SFRA cartridges are designed for modularity and reusability
- Your custom site code, 3rd party integrations, and LINK integrations are all delivered using cartridges
- An SFRA site is implemented at a minimum with the following:
  - A base cartridge: app_storefront_base
  - A modules folder: peer of cartridge folder, but NOT a cartridge
- Both must be deployed to your sandbox, but only the base cartridge needs to be in the cartridge path:



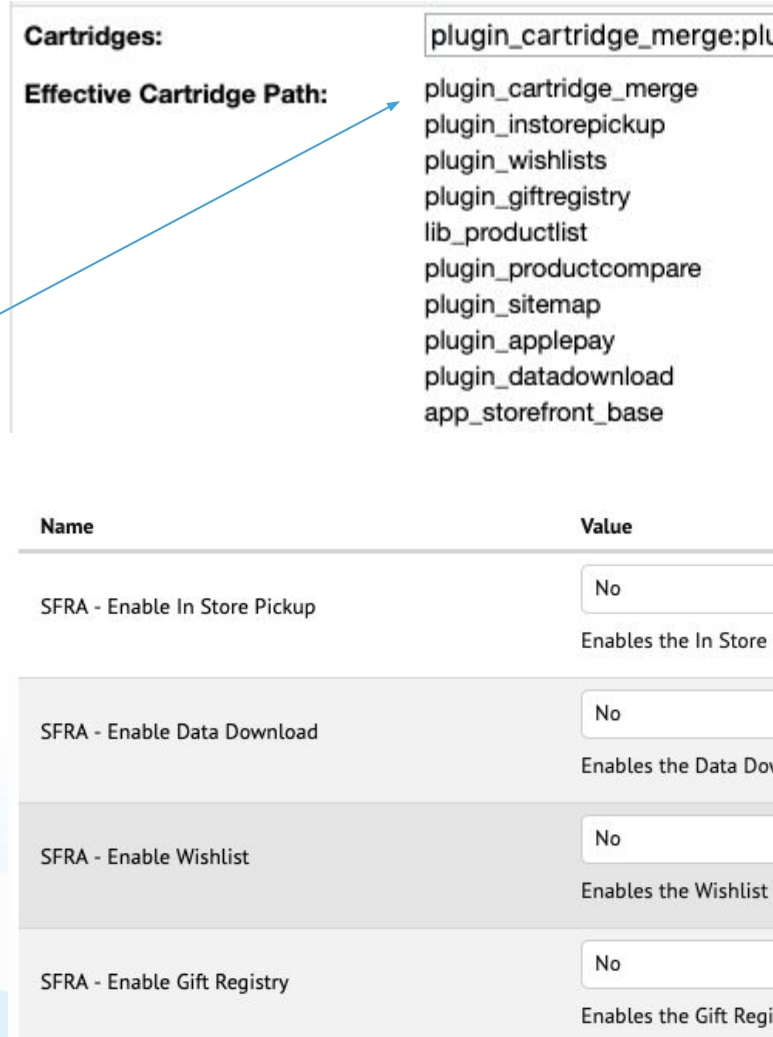
- Modules directory must be uploaded, but not added to cartridge path
- Don't modify the base cartridge: it is intended ONLY to be overriden by your custom cartridges. This is a best practice.
- For more documentation on cartridges, visit the documentation

Source: SFRA Overview - ViewModels and Contollers

Module 2.1: Installing SFRA base + Optional Features

Available for installation every sandbox

- In Site Import/Export, you can install SFRA + optional cartridges
- This allows you to demo a more comprehensive site that is similar to the functionality SiteGenesis came with
- The cartridge path is longer to support all the cartridges for this version of SFRA deployment
- The merging of all these cartridges is done by the customization cartridge plugin_cartridge_merge
- If you add new cartridges that override functionality on the ones listed here, you would have to do your own customization logic
- You can turn on/off specific cartridges on this option by using site preferences



Source: SFRA Overview - ViewModels and Contollers

Module 2.1: Cartridge Path Considerations

The cartridge path controls lookup of files for your site

- The cartridge path defines what code is executed in your site
- The cartridge path is searched from left to right when a specific controller, script, model or ISML template is invoked via URL or code
- On the previous slide, the plugin_cartridge_merge is searched first, and therefore used to 'merge' any conflicts that the other cartridges introduced: 2 files modifying the same area of the page
- Cartridge path file lookup can be superseded by the use of the require() function:

relative to the current file	require('./shipping') or require('../../util')
relative to the current cartridge	require ('~/cartridge/scripts/cart')
from beginning of cartridge path	require('*/cartridge/scripts/util/array');
specific B2C Commerce API	require('dw/catalog/CatalogMgr');
specific module or cartridge:	require('server');

Module 2.1: Demo SFRA + Optional Cartridges

Recommended for most new implementations

1. Demo SFRA “All-in-One”
2. Enable functionality via site preferences
3. Show Wishlist functionality
4. Show Pick up in store (BOPIS) functionality
5. Remove the base cartridge from the path: test the site
6. Put it back: test the site
7. Remove the first cartridge: plugin_cartridge_merge
8. In the next module we will override functionality from the base cartridge by creating a custom controller inside a custom cartridge. This is a best practice!

Time to Test your Knowledge!

Module 2.1 - Questions from this webinar

1. Is the SFRA modules directory a cartridge
   a. true
   b. false
2. In your code, there is a line like this:
   ```
   var cache = require('*/cartridge/scripts/middleware/cache');
   ```
3. Your cartridge path contains the following:
   ```
   your_custom_cartridge:app_storefront_base
   ```
   If there is cache.js in the first cartridge and last cartridge, which one is executed?
4. If the code is changed to:
   ```
   require('~/cartridge/scripts/middleware/cache');
   ```
   In which cartridge is the cache.js file found?
5. What file does this code refer to: `require('server');`
   a. server.js in app_storefront_base
   b. server.js in the modules folder
   c. server.js in the modules/server folder
   d. the first server.js found in the cartridge path

Time to Test your Knowledge!

Module 2.1 - Questions from pre-work

1. What does the dw.json file allow you to do?
   a. It is a legacy demandware file, not used in SFRA
   b. It allows upload of cartridges via npm uploadCartridge
   c. It logs you into your SB without having to remember username and password
2. If there are 2 code versions in your SB, which one is a true statement?
   a. The cartridge path contains all cartridges from the active version
   b. Cartridges uploaded to one version automatically get copied to other versions
   c. During execution, the cartridge path looks for cartridges in the active version
   d. Cartridge path and versions are totally unrelated
3. The cartridge path controls the behaviour of your site? True/False
4. Cartridges can only be uploaded using UX Studio: True/False
5. What is a best practice?
   a. Create your custom code in a cartridge, and put that cartridge in front of app_storefront_base in the cartridge path?
   b. Copy app_storefront_base with a new name, make all modifications there

Module 2.2: Controllers

- Objective
  - By the end of this module you will be able to define **what** a controller is, **understand** the meaning of routes, exports and middleware functions, and create a new controller that **appends** to a controller on the base
- This module will cover:
  - Controller functionality
  - Routes, exports, middleware functions
  - Extending a controller using append and prepend
- What is this functionality and why does it matter for implementations?
  - Controllers represent the URLs that are invoked in the storefront
  - Custom controllers are the best way to extend the base SFRA controllers without copying code

Module 2.2: What is a Controller?

Controllers are server-side scripts that handle storefront requests. Controllers manage the flow of data in your application, and create ViewModels to process each storefront request as a route and generate an appropriate response. For example, in a storefront application, clicking a category menu item or entering a search term triggers a controller that renders a page.

Controllers are written in JavaScript and B2C Commerce script and must conform to the CommonJS module standard.

A controller's file extension can be either `.ds` or `.js`.

Controllers must be located in the `controllers` folder at the top level of the cartridge. Exported methods for controllers must be explicitly made public to be available to handle storefront requests.

Source: SFRA Overview - ViewModels and Contollers

Module 2.2: Routes and Exports

Routes are the storefront URL endpoints

- Routes represent the specific URL endpoints that the storefront uses:

  In the Home.js controller, the Home-Show route is created with this code:

```
server.get('Show', consentTracking.consent, cache.applyDefaultCache, function (req, res, next) {
    ...
    next();
}, pageMetaData.computedPageMetaData);
```

- The Home-Show route is invoked via this storefront URL (the homepage):

  https://<your_instance>.net/on/demandware.store/Sites-RefArch-Site/en_US/ **Home-Show**

- Every controller is a CommonJS module, which is the convention that SFRA follows
- `module.exports = server.exports();` makes all the routes available on the URL
- If you don't export the routes, they are not available to the storefront

Source: SFRA Overview - ViewModels and Contollers

Module 2.2: Middleware Functions

Using the server module routes

Replacement for the guard functionality that existed in SiteGenesis Controllerss

Provides a different approach to extensibility that is new to SFRA

Registering routes using middleware functions:

- No middleware function: Home-ErrorNotFound

```
server.get('ErrorNotFound', function (req, res, next) {
    res.setStatusCode(404);
    res.render('error/notFound');
    next();
});
```

- Using middleware functions: Home-Show

```
server.get('Show', consentTracking.consent, cache.applyDefaultCache, function (req, res, next) {
    res.render('/home/homePage');
    next();
});
```

- middleware functions execute as a chain
- each function receives req, res and next as params

- consentTracking.consent presents the tracking consent modal that appears the first time you browse the homepage
- cache.applyDefaultCache caches the homepage for a period of time:
  - This replaces the use of the <iscache> tag
  - There are other versions for different caching periods

Module 2.2: Usage of req, res, next in routes

Let's look at the code inside a route

Each step of a middleware chain is a function that takes three arguments: `req`, `res`, and `next`, in this order.

`req` stands for request, it contains information about the server request that initiated execution. The `req` object contains user input information. The `req` argument parses query string parameters and assigns them to the `req.querystring` object.

`res` stands for response and contains functionality for outputting data back to the client.

Examples:

- `res.cacheExpiration(24)`: Sets cache expiration to 24 hours from now.
- `res.render(templateName, data)`: Outputs an ISML template back to the client and assigns data to `pdict`.
- `res.json(data)`: Prints a JSON object back to the screen. It's helpful in creating AJAX service endpoints that you want to execute from the client-side scripts.
- `res.setViewData(data)`: Doesn't render anything, but sets the output object. This can be helpful if you want to add multiple objects to the `pdict` of the template, which contains the information for rendering that is passed to the template.
- `setViewData`: Merges all the data that you passed into a single object, so you can call it at every step of the middleware chain.

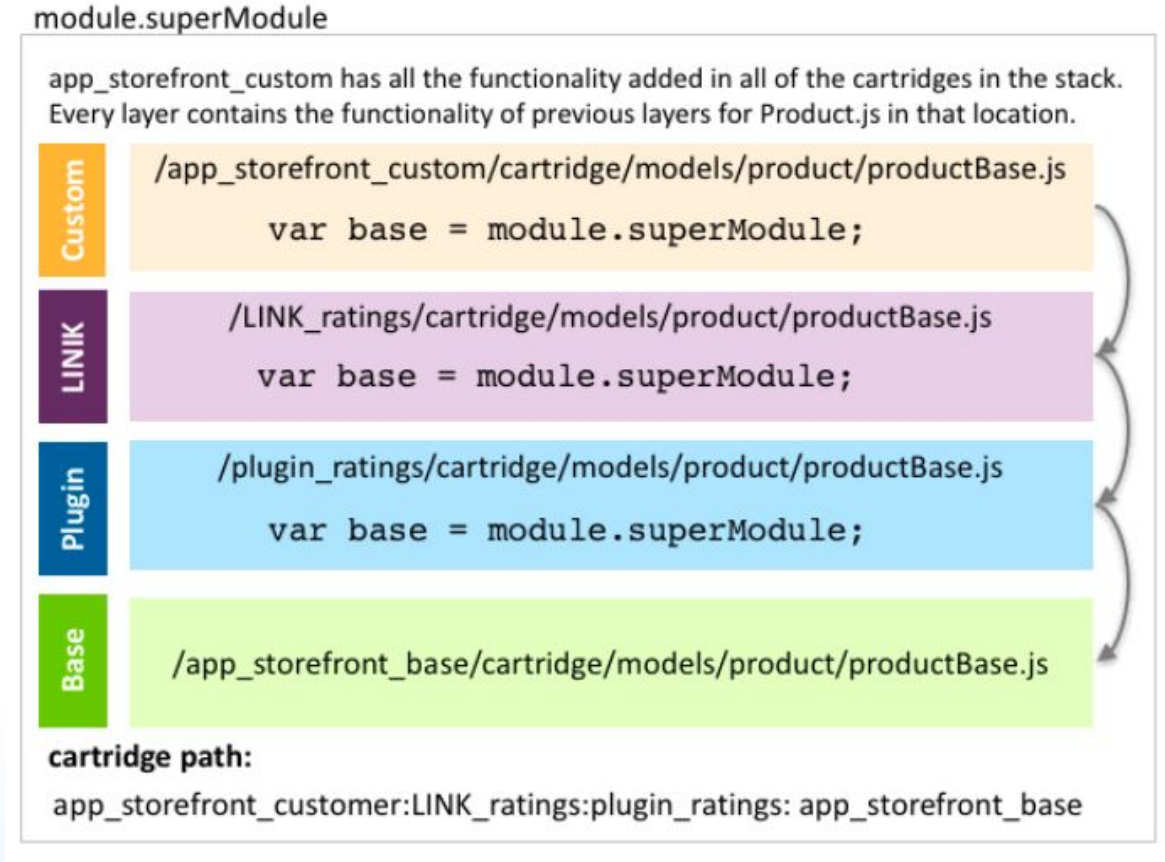
The `next` function notifies the server that you are done with a middleware step so that it can execute the next step in the chain.

Module 2.2: Extending Controllers (and Models)

Modules can extend other modules in the cartridge path: not just controllers

Use of **server.extend(module.superModule)**

- The module.superModule global property provides access to the most recent module on the cartridge path with the same path/name as the current module
- A controller can extend or override another controller with the same path/name without having to copy code from one to the other: this results in reusability that SiteGenesis controllers did not have
- There are 3 main methods used to extend or overwrite a route:
  - append: extends functionality by executing AFTER the superModule route
  - prepend: extends functionality by executing BEFORE the superModule route
  - replace: overrides the superModule route



Module 2.2: Examples of Extending a Controller

Based on your current cartridge path, find some examples of extending

plugin_cartridge_merge:**plugin_instorepickup**:**plugin_wishlists**:plugin_giftregistry:lib_productlist:plugin_productcompare:plugin_site map:plugin_applepay:plugin_datadownload:**app_storefront_base**

Example A:

- plugin_instorepickup/**Cart.js** extends app_storefront_base/**Cart.js**
- server.**replace**('AddProduct'...) executes INSTEAD of the base AddProduct() function

Example B:

- plugin_wishlists/**Account.js** extends app_storefront_base/**Account.js**
- server.**prepend**('Login'...) executes BEFORE the base Login() function
- server.**append**('Login'...) executes AFTER the base Login() function

Module 2.2: Demo of Append and Prepend in Routes

Time to Test your Knowledge!

Module 2.2: Controllers

1. Which of the following statements is not correct?
   a. Controllers are the main entry point into the storefront application
   b. Controllers gather the data from the model, and pass the data to the ISML template
   c. A controller can invoke another controller
   d. Controllers are commonJS modules
2. Which is not a method for extending a specific controller route (i.e. Home-Show):
   a. append
   b. prepend
   c. extend
   d. replace
3. If you extend a controller route, can you prepend as well as append to the same route? T/F
4. If you remove `next();` on a route, what is the effect?
   a. The next route in the chain is not executed
   b. It does not break anything
   c. It is the same as using a replace on the route

Time to Test your Knowledge!

Module 2.2: These questions will require some extra work on your part

1. The Home-Show route uses this middleware chain:

   ```
   server.get('Show', consentTracking.consent, cache.applyDefaultCache, function (req, res, next)
   {...});
   ```

   and another cartridge extends this route without a middleware chain:

   ```
   server.append('Show', function (req, res, next) {...});
   ```

   Assuming the code is correct on both functions, does this work? T/F

2. Where are the methods of the response listed (i.e. res.render in controller code)?
   a. Under dw.system.Response documentation
   b. Under SFRA / Server-side JS / Class: Response documentation
   c. Part of the commonJS documentation
   d. Part of the server middleware functions

3. In the Home-Show controller, why is the a middleware function in blue executed after the route?

   ```
   server.get('Show', consentTracking.consent, cache.applyDefaultCache, function (req, res, next)
   {
       ...
   }, pageMetaData.computedPageMetaData );
   ```