Storefront Reference Architecture (SFRA)

rohan

Rohan Mallick

Overview

- Storefront Reference Architecture (SFRA) represents a new approach to building and customizing your storefront site
- The name change from ‘SiteGenesis’ to ‘Storefront Reference Architecture’
- Leverage the existing JavaScript controllers development model along with a new architecture and customization model intended to make it easier for development teams to follow a model-view-controller (MVC) approach.
- Leverage UX best practices, data driven design, and use of the popular **Bootstrap** mobile UI framework to optimize the shopping experience across mobile devices.

Setting Up Workspace

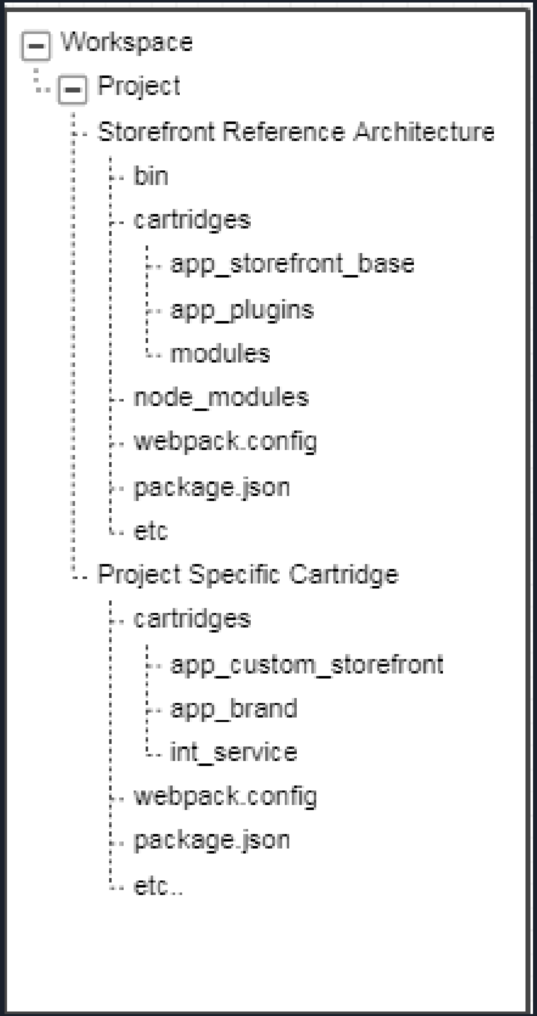
Github Setup (https://github.com)

- Should have Salesforce Commerce cloud GitHub repository access.
- Create an account in GitHub. (Verify your Email as well if not done).
- Under Github settings, got to security and enable two factor authentication. You need to have **Google Authenticator** app in your Mobile.
- Under Github settings, go to developer settings and create **Personal Access Token**.
- Open https://cc-community-authmgr.herokuapp.com/ url and link and request for SFCC repository access. You will receive an email to join the SFCC github project.
- Download the latest Git from https://git-scm.com/

Persona 01

Workspace Setup

- Create a workspace folder.
- Under workspace folder create new folder for project.
- Under project folder clone the SFRA repository.
- Create one more folder to store your project specific cartridge.

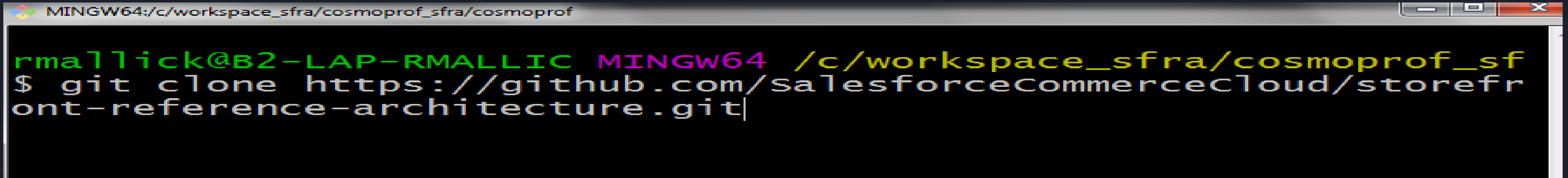


```
Workspace
  Project
    Storefront Reference Architecture
      bin
      cartridges
        app_storefront_base
        app_plugins
        modules
      node_modules
      webpack.config
      package.json
      etc
    Project Specific Cartridge
      cartridges
        app_custom_storefront
        app_brand
        int_service
      webpack.config
      package.json
      etc..
```

Persona 01

Clone SFRA Cartridges

- Open https://github.com/SalesforceCommerceCloud/storefront-reference-architecture and clone the repository.
- Go to your project directory and open the git bash command line.
- Type `git clone` followed by SFRA repo URI. If it ask for user name password then enter your username and Personal Access Token, we created earlier.



```
MINGW64:/c/workspace_sfra/cosmoprof_sfra/cosmoprof

rmallick@B2-LAP-RMALLIC MINGW64 /c/workspace_sfra/cosmoprof_sf
$ git clone https://github.com/SalesforceCommerceCloud/storefr
ont-reference-architecture.git
```

Persona 01

Build SFRA Cartridges

- Install sgmf-scripts node package manager `npm install sgmf-scripts -g`
- Navigate top level folder containing cartridges and use below commands.
- To compile both CSS and JavaScript -

```
npm run compile:js && npm run compile:scss && npm run compile:fonts
```

- To compile only JavaScript: `npm run compile:js`
- To compile only CSS: - `npm run compile:scss`
- Watch file changes - `npm run watch`

All above commands, we can find under the scripts of package.json of top level folder.

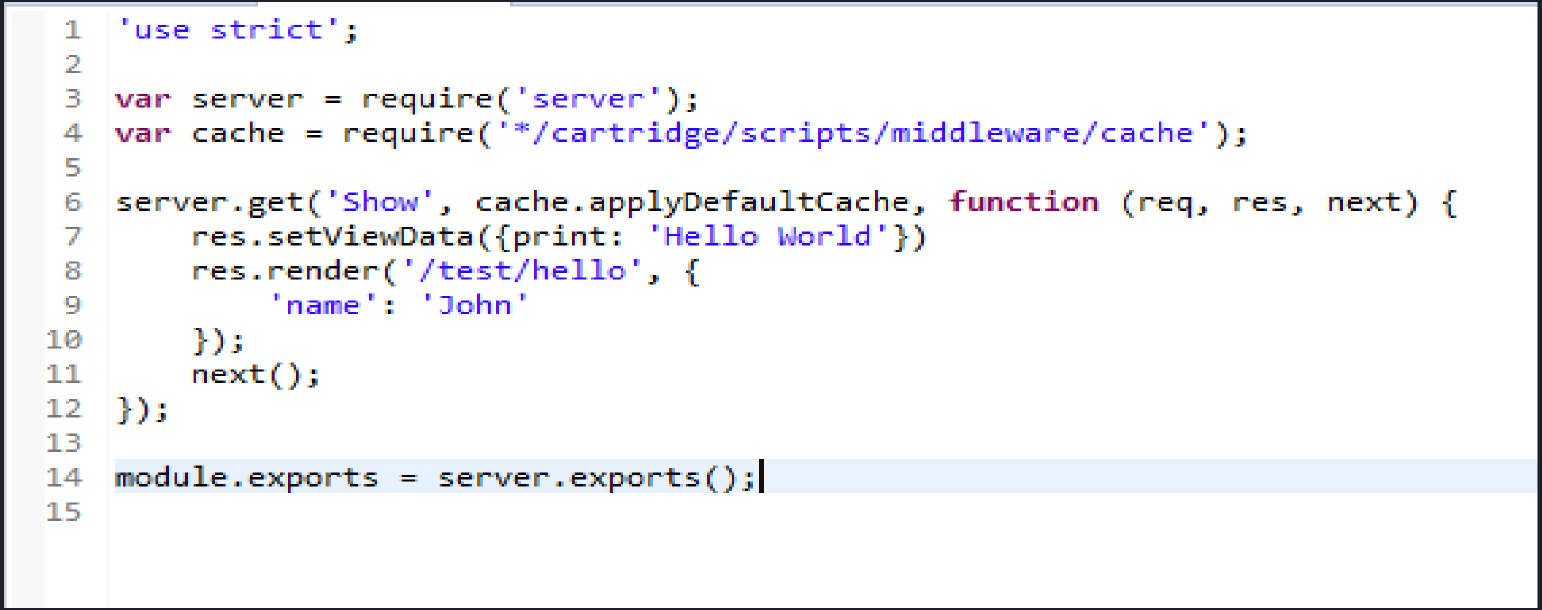
Persona 01

Configuring SFRA

- Import RefArch site from Site Import/Export in business manager.
- After importing RefArch, under manage site, select RefArch, configure cartridges under settings.
- Rebuild search index.
- Invalidated page cache and disable it for development.
- Upload cartridges.
- Open storefront. Boom 🙄

Persona 01

Let’s explore Hello World



```javascript
'use strict';

var server = require('server');
var cache = require('*/cartridge/scripts/middleware/cache');

server.get('Show', cache.applyDefaultCache, function (req, res, next) {
    res.setViewData({print: 'Hello World'})
    res.render('/test/hello', {
        'name': 'John'
    });
    next();
});

module.exports = server.exports();
```

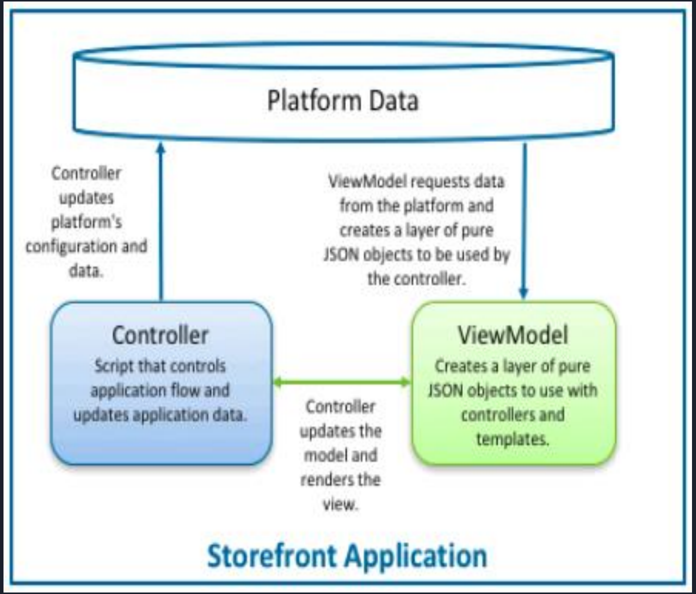
Let’s explore Hello World (Cont..)

- Server module is used for providing all routing functionality in SFRA.
- In Commerce Cloud, **modules folder** is globally available folder.
- Server module create global namespace for request, response, session and request middleware.
- Server module create a Route object on call of `server.exports();`
- server.exports() expose the route to the server to serve the http request by calling `route.public = true`
- SFRA is based on the concept of middleware chain execution.
- Server modules also provides a way to extend or override any controller.
- `res.setViewData({})` and `res.getViewData()` is used as a setter or getter to the pidct variable.
- res object have function like render, json, xml to render the response to Http request.
- `res.redirect()` function is used for redirect to another URL

Persona 01

SFRA MVC Model

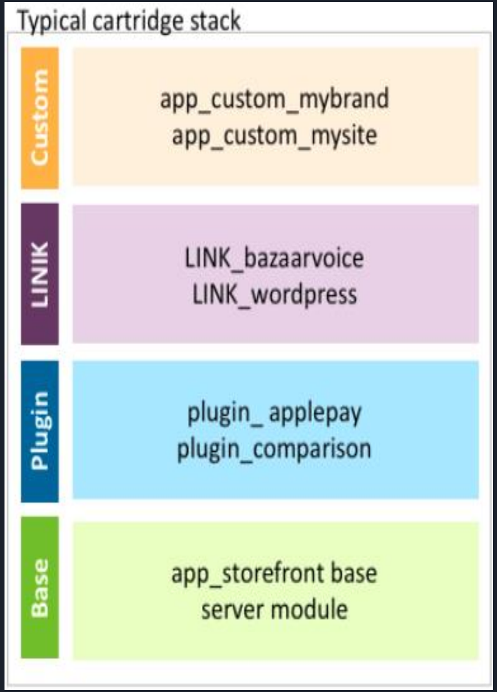
- Controller handles the user information, create viewModel and reder the pages.
- Controller requests data from B2C Commerce and passes the returned objects to a ViewModel to be converted into a serializable JavaScript object.
- The viewModels request data from B2C Commerce, convert B2C Commerce script API objects into pure JSON objects, and apply business logic.
- The viewModel provides the information needed to render the pages.


Platform Data
Controller updates platform's configuration and data.
ViewModel requests data from the platform and creates a layer of pure JSON objects to be used by the controller.
Controller
Script that controls application flow and updates application data.
ViewModel
Creates a layer of pure JSON objects to use with controllers and templates.
Controller updates the model and renders the view.
Storefront Application

Persona 01

SFRA Cartridge Stack

- Base Cartridge provides the core functionality provided by Commerce Cloud team through GitHub.
- It is highly recommend that we should not modify app_storefront_base. Instead, create our own cartridge and add it as an overlay in the Business Manager cartridge path.
- Plugins enhance the base feature of the SFRA as a plug and play architecture.
- LINK Cartridges adds third-party functionality to the site.
- Adds specific customizations for the brand and organization.
- The cartridge path is always searched left to right, and the first controller or pipeline found with a particular name is used.

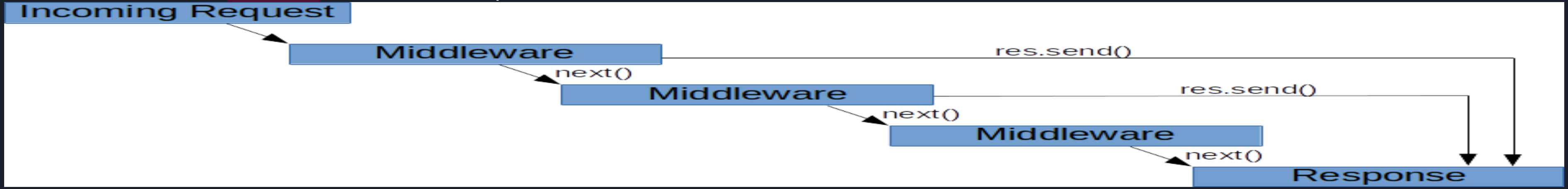


```
app_custom_mybrand:app_custom_mysite:LINK_bazaarvoice:LINK_wordpress:plugin_applepa
y:plugin_comparison:app_storefront_base
```

Persona 01

Middleware

- SFRA is based on the concept of middleware chain execution.
- Middlewares are functions executed in the middle after the incoming request then produces an output which could be the final output passed or could be used by the next middleware until the request/response cycle is completed.
- We can have more than one middleware and they will execute in the order they are declared.
- Each middleware function take three parameter `req, res, and next`.
- The next middleware function is commonly denoted by a variable named `next`
- In Hello World example, `cache.applyDefaultCache` and `function (req, res, next)` is an example of middleware.



Persona 01

Let’s see Middleware function

Below middleware function will filter the Post request.

```
function post(req, res, next) {

    if (req.httpMethod === 'POST') {

        next();

    } else {

        next(new Error('Params do not match route'));

    }

}
```

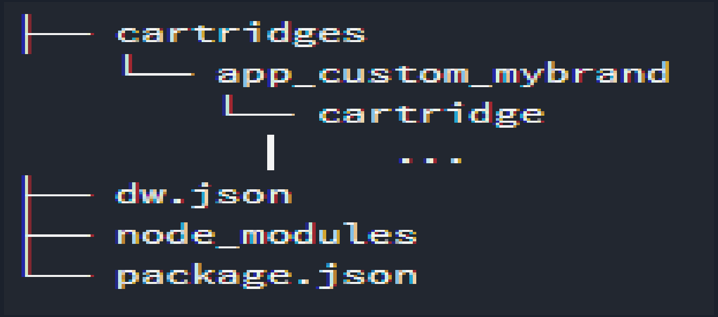
Persona 01

Adding Custom Cartridges

- Implementing a site requires at least one custom cartridge.
- Navigate to your project specific directory and use sgmf-scripts commands we installed earlier.

```
sgmf-scripts --createCartridge app_custom_storefront
```

- Above command will generate all directories and files in your folders.



```
├── cartridges
    └── app_custom_mybrand
        └── cartridge
          |      ...
├── dw.json
├── node_modules
└── package.json
```

Install all dependencies required by the new cartridge.

```
npm install
```

Persona 01

Update package.json

- As we can’t modify the base cartridge directly, SFRA provides a mechanism to selectively override CSS styles and client-side JavaScript by updating “path” property of package.json file under project folder.
- The paths property lists every cartridge with CSS and client-side JavaScript functionality customized in our site.

```
"paths": {

    "base": "../storefront-reference-architecture/cartridges/app_storefront_base/",

    "ratings": "../plugin_ratings/cartridges/plugin_ratings/",

    "applepay": "../plugin-applepay/cartridges/plugin_applepay/"

  }

}
```

Persona 01

Customizing Templates

- In order to override the template from the base cartridge in custom cartridge, include a template with the same name as the template you want to override.
- The location of your template in your cartridge must same the location of the original template in the base cartridge.
- There are only two decorator template in SFRA. page.isml and checkout.isml.
- Checkout.isml doesn't have navigation information improve the percentage of cart abandonment. 🙄
- All pt_template that existed in SiteGenesis has been removed in SFRA.
- Each template will now have separate Client-Side scripts and CSS using assets.addJs and assets.addCss function in ISML under script tag.

```
<isscript>var assets = require('~/cartridge/scripts/assets.js');assets.addCss('/css/cart.css');</isscript>
```

Persona 01

Best Practices with ISML

Any business logic shouldn’t be included in ISML. It should be used only for rendering pages.

Iscache:	Set by default to 24 hours. We recommend changing this value in the controller.
isset:	Only used to set a variable used by isinclude. Don't use complex expressions in the isset.
isscript:	Only used to add client-side JavaScript/CSS to the page using the asset module.
ismodule:	Only used for content assets.
iscontent:	Change this value in the controller.
Iscookie:	Change this value in the controller.
isredirect:	Change this value in the controller.
isstatus:	Change this value in the controller.
isobject:	Used to wrap products for analytics and the Storefront Toolkit.

Persona 01

Customize SFRA Controllers

- We can extend and override the controller from the Base SFRA cartridge.
- But the decision of extending and overriding Controllers should be wise enough because it can significantly impact functionality and performance.
- If you are okay with the existing logic and you just need to extend/update some more logic/data, you can extend the existing controller by adding extra middleware function using `server.append(), and server.prepend()`.
- Sometimes you might want to reuse the route's name, but do not want any of the existing functionality. In those cases, you can use `server.replace()` command to completely remove and re-add a new route.

Persona 01

Let’s extend Controllers

Demo.js (In Base Cartridge)

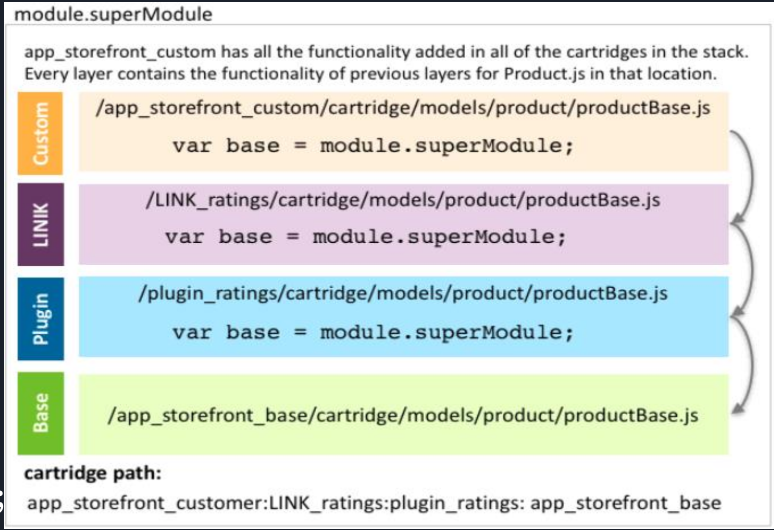
```
var server = require('server');

server.get('Show', function(req, res, next) {

    res.render('test/demo', { value: 'Hello World' });

    next();

});

module.exports = server.exports();
```

Persona 01

Let’s extend Controllers

Demo.js (In custom Cartridge)

```
var server = require('server');

var page = module.superModule;

server.extend(page);

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

    res.setViewData({ value: 'Hello Commerce Cloud' });

    next();

});

module.exports = server.exports();
```


module.superModule

app_storefront_custom has all the functionality added in all of the cartridges in the stack. Every layer contains the functionality of previous layers for Product.js in that location.

Custom
/app_storefront_custom/cartridge/models/product/productBase.js
var base = module.superModule;
LINIK
/LINK_ratings/cartridge/models/product/productBase.js
var base = module.superModule;
Plugin
/plugin_ratings/cartridge/models/product/productBase.js
var base = module.superModule;
Base
/app_storefront_base/cartridge/models/product/productBase.js

cartridge path:
app_storefront_customer:LINK_ratings:plugin_ratings: app_storefront_base

Persona 01

Let’s replace Controllers

Demo.js (In custom Cartridge)

```
var server = require('server');

var page = module.superModule;

server.extend(page);

server.replace('Show', server.middleware.get, function(req, res, next){

    res.render('test/newDemo');

    next();

});

module.exports = server.exports();
```

Persona 01

Summary Controllers

- **module.superModule**: Imports functionality from the first controller with the same name and location found to the right of the current cartridge on the cartridge path.
- **server.extend:** Inherits the existing server object and extends it with a list of new routes from the super module.
- **server.append:** Modifies the existing route by appending middleware that adds properties to the viewData object for rendering. Using server.append causes a route to execute both the original middleware chain and any additional steps. If you're interacting with a web service or third-party system, don’t use server.append.
- **res.getViewData:** Gets the current viewData object from the response object.
- **res.setViewData**: Updates the viewData object used for rendering the template.

Persona 01

Forms

- If we are creating a simple form that doesn't store data, we can use standard HTML form that uses AJAX for validation and error rendering.
- We can also create a complex form that stores data, requires server-side validation using B2C Commerce form definition.
- In SFRA, we need to create a JSON object of the form which contains all form data.
- We can use `server.forms.getForm(‘profile’)` function to create JSON object from form definition.
- We can use `form.clear()` function to clear the data from form.
- We can access form object in template using `pdict` but we need to pass this object to the template by the controller.
- Note - In form definition, don’t use secure attribute as a true. This is deprecated now.

Form Definition

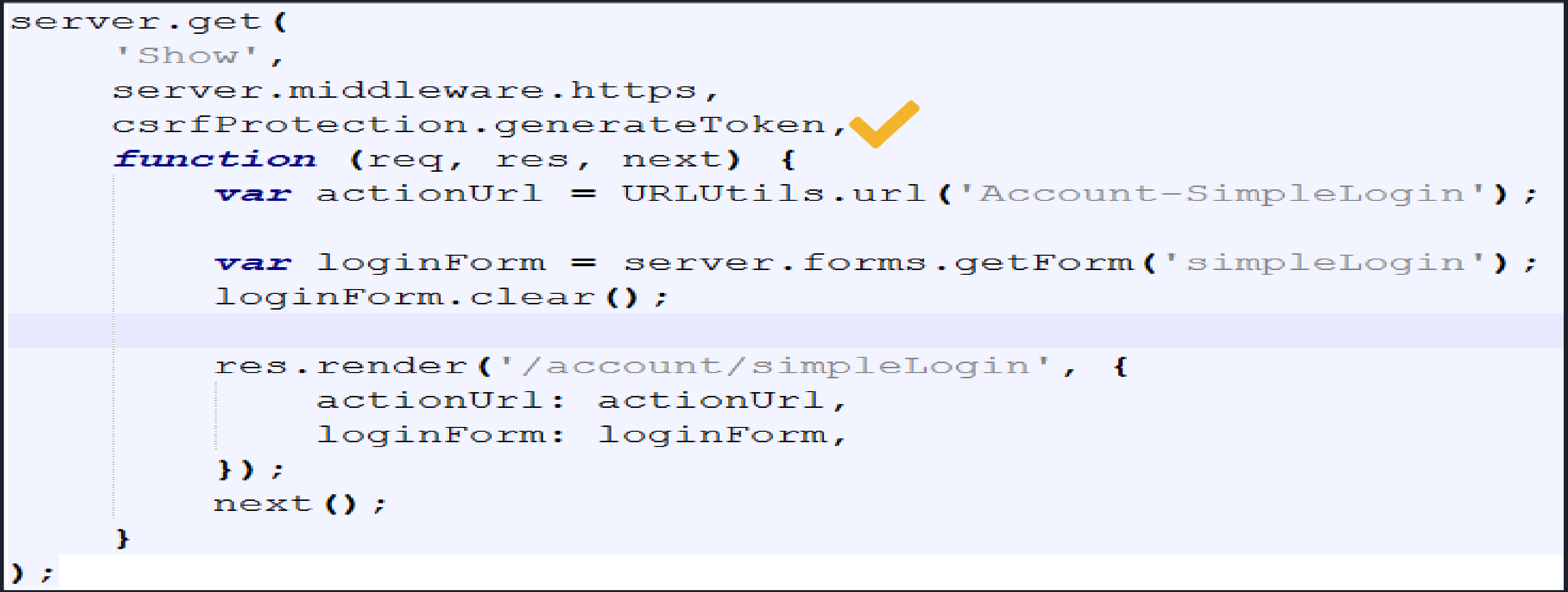
simpleLogin.xml

```
<?xml version="1.0"?>

<form xmlns="http://www.demandware.com/xml/form/2008-04-19" secure="false">

     <field formid="username" label="label.loginId" type="string" mandatory="true"/>


     <field formid="password" label="label.password" type="string" mandatory="true"/>

</form>
```

Persona 01

Forms in Controller

SimpleLogin.js

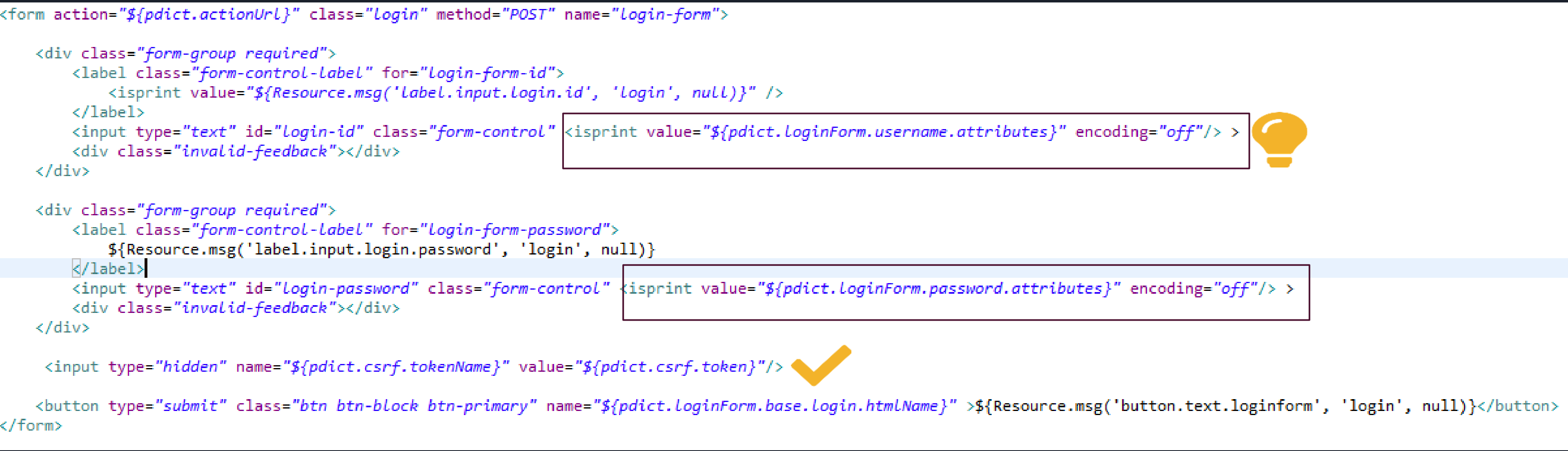


```
server.get(
    'Show',
    server.middleware.https,
    csrfProtection.generateToken,✓
    function (req, res, next) {
        var actionUrl = URLUtils.url('Account-SimpleLogin');

        var loginForm = server.forms.getForm('simpleLogin');
        loginForm.clear();

        res.render('/account/simpleLogin', {
            actionUrl: actionUrl,
            loginForm: loginForm,
        });
        next();
    }
);
```

Persona 01

Forms in ISML

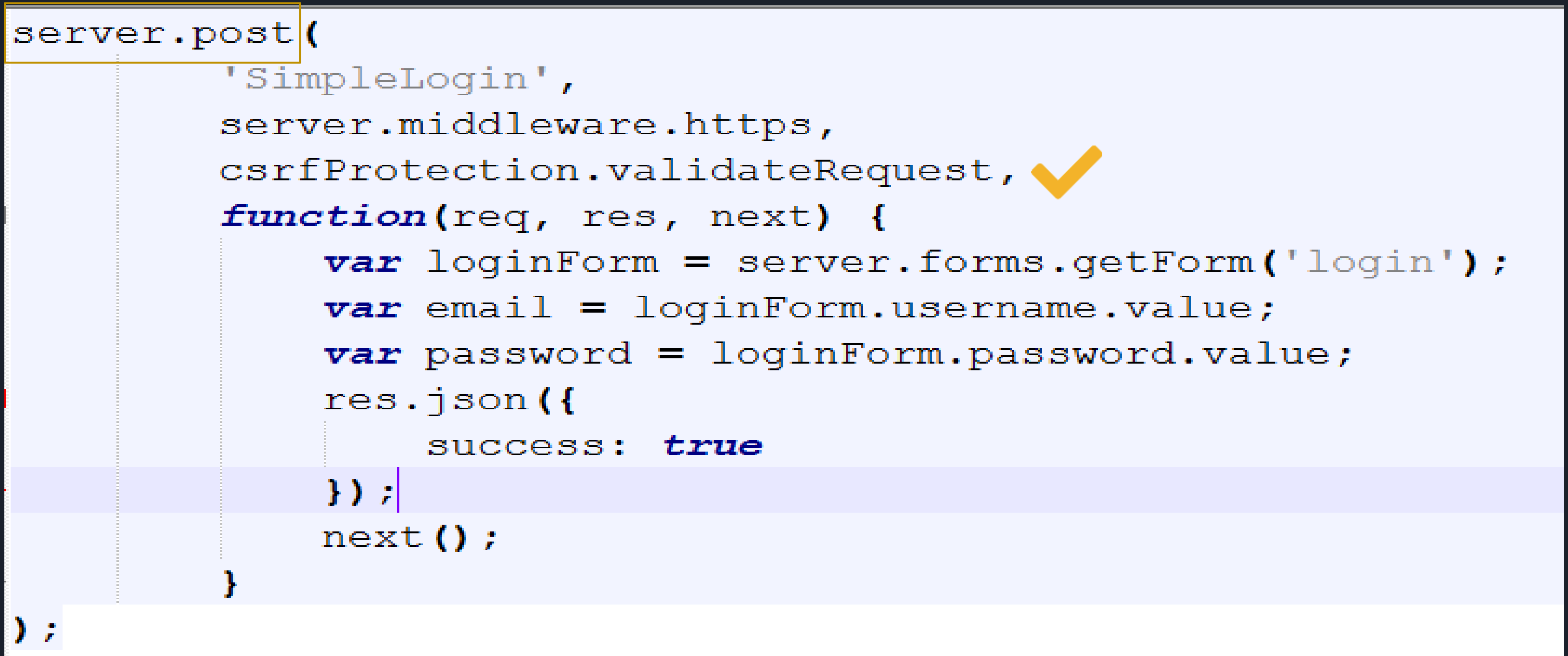
SimpleLogin.isml



```
<form action="${pdict.actionUrl}" class="login" method="POST" name="login-form">

    <div class="form-group required">
        <label class="form-control-label" for="login-form-id">
            <isprint value="${Resource.msg('label.input.login.id', 'login', null)}" />
        </label>
        <input type="text" id="login-id" class="form-control" <isprint value="${pdict.loginForm.username.attributes}" encoding="off"/> >
        <div class="invalid-feedback"></div>
    </div>

    <div class="form-group required">
        <label class="form-control-label" for="login-form-password">
            ${Resource.msg('label.input.login.password', 'login', null)}
        </label>
        <input type="text" id="login-password" class="form-control" <isprint value="${pdict.loginForm.password.attributes}" encoding="off"/> >
        <div class="invalid-feedback"></div>
    </div>

     <input type="hidden" name="${pdict.csrf.tokenName}" value="${pdict.csrf.token}"/>

    <button type="submit" class="btn btn-block btn-primary" name="${pdict.loginForm.base.login.htmlName}" >${Resource.msg('button.text.loginform', 'login', null)}</button>
</form>
```

Handle Form Submit

Account- SimpleLogin



```
server.post(
          'SimpleLogin',
          server.middleware.https,
          csrfProtection.validateRequest, ✓
          function(req, res, next) {
               var loginForm = server.forms.getForm('login');
               var email = loginForm.username.value;
               var password = loginForm.password.value;
               res.json({
                    success: true
               });
               next();
          }
);
```

Event Emitters

- SFRA is based on asynchronous event-driven architecture in which server module emits events at every step of execution.
- We can subscribe and unsubscribe events from a given route.
- We can add the event listener to the emitted events.
- We can add listener using **`this.on()`** and remove listener using **`this.off()`** to the events.
- List of events.
  - **`route:Start:`** - Emitted as a first thing before middleware chain execution.
  - **`route:Redirect:`** - Emitted right before res.redirect execution.
  - **`route:Step:`** - Emitted before execution of every step in the middleware chain.
  - **`route:Complete:`** - Emitted after every step in the chain finishes execution. Currently subscribed to by the server to render ISML or JSON back to the client.
  - **`route:BeforeComplete`**: - Emitted before the route:Complete event but after all middleware functions. Used to store user submitted data to the database; most commonly in forms.

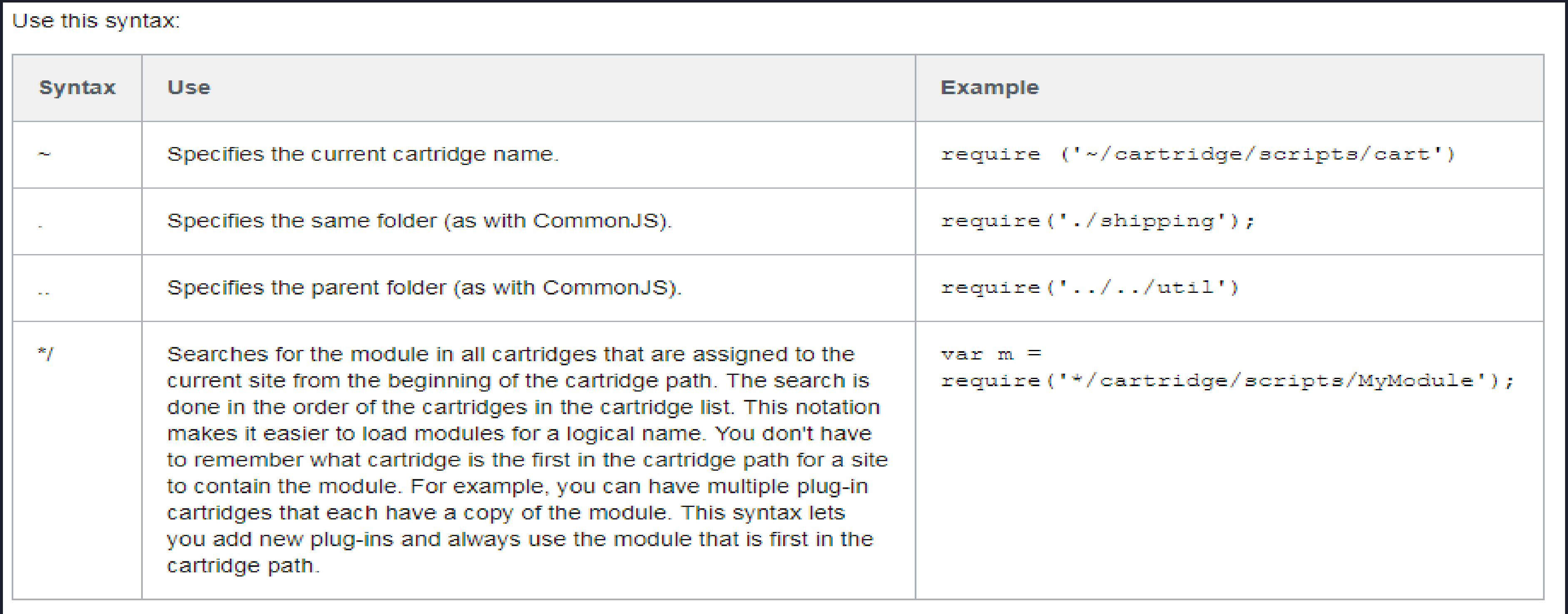
Persona 01

Access Modules

- Commerce Cloud supports **CommonJS** `require` syntax to access modules and relative paths.
- A module hides private data while exposing public objects and methods via a free variable named `exports`.
- Storefront Reference Architecture (SFRA) uses **`module.exports`**, instead of simply **`exports,`** because it accepts objects.
- **`TopLevel.global.require(path:String)`** method can be used to import the modules After the module is loaded, you can use any of its exported variables or methods.
- Currently, Commerce Cloud supports these extensions, listed in priority order:
  - .js - JavaScript file
  - .ds - B2C Commerce script file
  - .json - JSON file

Persona 01

Accessing Modules


Use this syntax:

Syntax	Use	Example
~	Specifies the current cartridge name.	`require ('~/cartridge/scripts/cart')`
.	Specifies the same folder (as with CommonJS).	`require('./shipping');`
..	Specifies the parent folder (as with CommonJS).	`require('../../util')`
*/	Searches for the module in all cartridges that are assigned to the current site from the beginning of the cartridge path. The search is done in the order of the cartridges in the cartridge list. This notation makes it easier to load modules for a logical name. You don't have to remember what cartridge is the first in the cartridge path for a site to contain the module. For example, you can have multiple plug-in cartridges that each have a copy of the module. This syntax lets you add new plug-ins and always use the module that is first in the cartridge path.	`var m = require('*/cartridge/scripts/MyModule');`

Persona 01

SuperModule Scenarios

Suppose the cartridge path as below and we have Page.js in cartridge_B

cartridge_A: cartridge_B: cartridge_C: cartridge_D: app_storefront_base.

If we use var page = module.superModule; in Page.js,

If	Returns
All cartridges contain a Page.js file in the same location as cartridge_B.	cartridge_C module Page.js module- because it's the next cartridge after cartridge_B in the path
cartridge_A and cartridge_D contain a Page.js file in the same location as cartridge_B.	cartridge_D module Page.js module - because it's after cartridge_B in the path
Only cartridge_A contains a Page.js file in the same location as cartridge_B.	null - because cartridge_A isn't after cartridge_B in the path
cartridge_C contains a Page.js file in a different location and app_storefront_base contains a Page.js file in the same location as	app-storefront_base Page.js module - because the files must match name and location.

Persona 01

JavaScript Bonus

Ways to create JS object.

Using Object Literal

```
let bike = {name: 'SuperSport', maker:'Ducati', engine:'937cc'};

bike.wheelType = 'Alloy';

bike.stop = function() {

    console.log('Applying Brake...');

}
```

Persona 01

JavaScript Bonus

Ways to create JS object.

Using Object with Constructor

```
function Vehicle(name, maker) {

   this.name = name;

   this.maker = maker;

   this.stop = function() {

    console.log('Applying Brake...');

}

}
```

- Use **this** keyword for adding properties and methods to the object.
- Use **new** keyword to create the Object

```
let car1 = new Vehicle('Fiesta', 'Ford');
let car2 = new Vehicle('Santa Fe', 'Hyundai')
```

JavaScript Bonus

Ways to create JS object.

Using Object.create() [Object.create(proto [, properties Object])]

- Object.create() will create a brand new object using an existing object and set existing object's prototype to itself.
- Object.create() allowed to create object with more attribute options like **configurable**, **enumerable**, **writable** and **value**

```
let car = Object.create(Object.prototype, {

                    Name:{ value: 'Fiesta', configurable: true, writable: true,
enumerable: false }

});
```