
salesforce
Fast Path to B2C Commerce Developer Certification
Module 4
Forms, Transactions and Middleware Events

Module 4.1: Forms

Objective - By the end of this module you will be able to **define** a form, **render** a form, and **use** client-side JavaScript to **submit** the form and **handle** responses.

This module will cover:

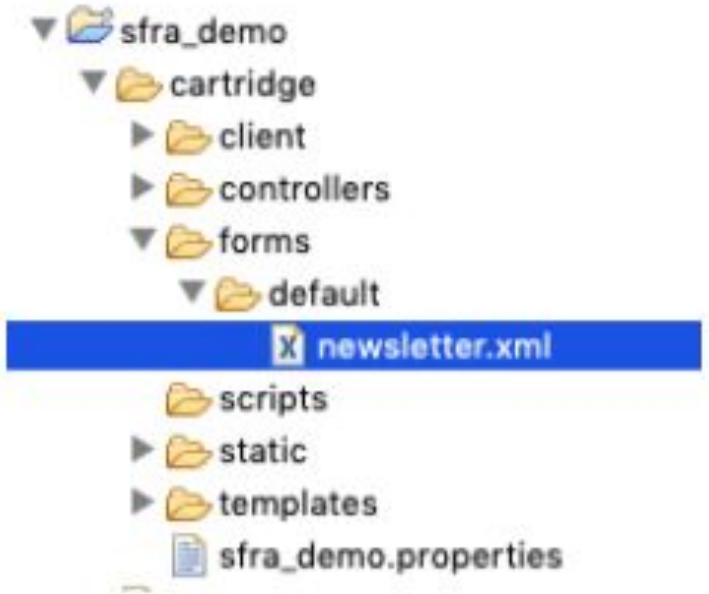
- Form definition
- ISML template
- Controller route to render the from
- Controller route to handle form submission
- Client-side JavaScript that use Ajax to submit the form, and handle responses

- What is this functionality and why does it matter for implementations?
  - Forms are an integral part of every commerce site: cart, account, checkout
  - Understanding the inner workings of forms is essential B2C Commerce knowledge

Module 4.1: Form Metadata

Form Definition

- XML file located in the cartridge/forms folder
- Defines the fields to store data:
  - Submitted during execution, or
  - Initialized from a system object (i.e. account profile)
- During execution, the form becomes a session object that is accessible via server.forms.getForm('formfilename');
- Common field form elements:
  - mandatory: makes the field required
  - max-length: filed length allowed
  - type: string, integer, number, boolean and date
- More information on documentation:
  - SFRA Forms
  - Form Definition Elements
- Be aware that SFRA does not use all the field attributes that SG does: it relies heavily on browser-based validation.



Module 4.1: Controller route that renders form ISML

Contains a route to render an ISML template containing an HTML form

The route loads and clears the form.xml defined before:

```
var actionUrl = dw.web.URLUtils.url('Newsletter-Handler');
var newsletterForm=server.forms.getForm('newsletter');
newsletterForm.clear();
```

The route then renders the form ISML template:

```
res.render('/newsletter/newslettersignup',{
    actionURL: actionURL,
    newsletterForm: newsletterForm
});
```

The actionURL must be one of the following:

- Another route in the same controller: Newsletter-Handler
- A route on another controller: Login-Show calls Account-Login

Module 4.1: Form ISML template

ISML template to show a form

Adds any specific js and css files for this form to handle client-side validation and submission via Ajax:

```
assets.addJs('/js/newsletter.js');
```

Contains standard HTML form definition with specific parameters to support client-side validation provided by the browser:

Form definition with actionUrl passed from controller:

```
<form action="${pdict.actionUrl}" method="POST" class="newsletter-form"
<isprint value="${pdict.newsletterForm.attributes}" encoding="off" />>
```

Form fields with specific form attributes (use inspect to see attributes):

```
<input type=”input” class=”form-control” id=”newsletter-form-fname”
<isprint value=”${pdict.newsletterForm.fname.attributes}”
encoding=”off” />>
```

Labels use the pdict.form.field.label defined in form metadata:

```
<isprint value=”${pdict.newsletterForm.fname.label}”
encoding=”htmlcontent” />
```

Module 4.1: Controller route that handles form submission

Controller to handle the form submission

There are two ways the form controller can be written:

- Without client-side JS: the form posts to a route directly, like SiteGenesis
- With client-side Java: the form uses Ajax to post form data to a route

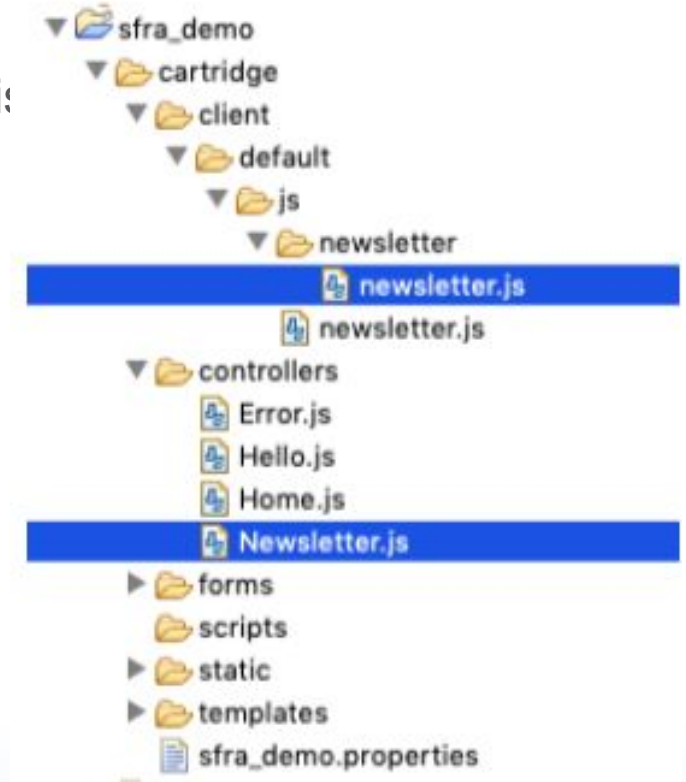
We will start with the first option. Basically, the controller:

- Retrieves form data from the session:

```
var newsletterForm = server.forms.getForm('newsletter');
```

- Renders a page:

```
res.render('/newsletter/newslettersuccess', {
    continueUrl: continueUrl,
    newsletterForm: newsletterForm
});
```



Module 4.1: Form handling without client-side JS (version 1)

1. We will walk thru NewsletterV1 controller example
2. We will look at:
   a. NewsletterV1-Show route: render signup page
   b. Form metadata: newsletter.xml
   c. Form ISML: newslettersignupV1.isml
   d. NewsletterV1-Handler route: renders success or error page
3. Questions:
   a. Is there an action on the form metadata?
   b. How does the form ISML know where to post to?
   c. Where is the client-side validation implemented? For example, the email regular expression.
   d. Why is the handler code validating the form?

Module 4.1: Controller handler that interacts with client-side JS

This controller handler sends JSON objects to the client-side

The handler route uses Response.json() to send a JSON object containing status and a message to the client side:

```
var newsletterForm = server.forms.getForm('newsletter');

// Perform any server-side validation before this point, and invalidate form accordingly.
if (newsletterForm.valid) {
    // Show the success page
    res.json({
        success: true,
        redirectUrl: URLUtils.url('NewsletterV2-Success').toString()
    });
} else {
    // Handle server-side validation errors here: this is just an example
    res.setStatusCode(500);
    res.json({
        success: false,
        error: [Resource.msg('error.crossfieldvalidation', 'newsletter', null)]
    });
}
```

Module 4.1: Form submission using client-side JS

SFRA uses client-side Javascript extensively to handle forms

Must use **base** defined in package.json to refer to app_storefront_base code:

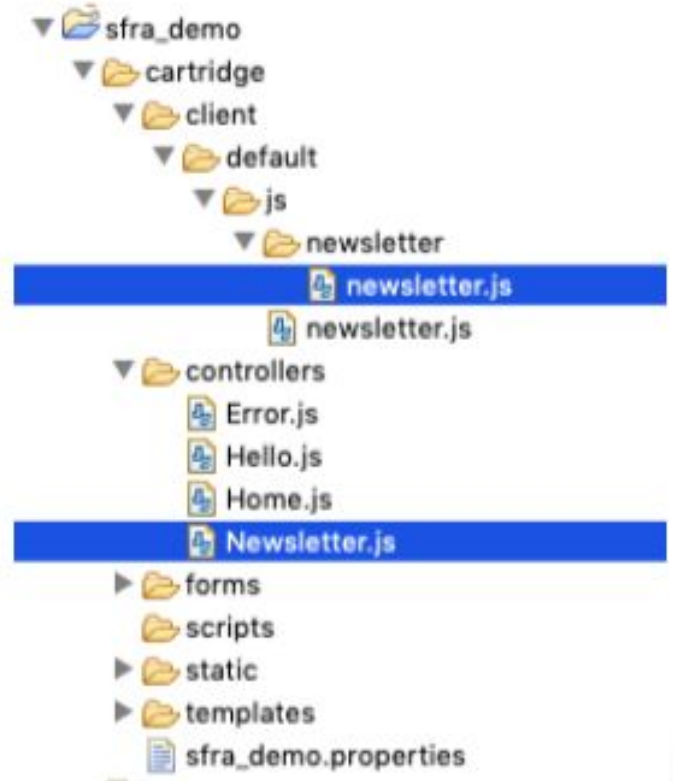
```
var formValidation = require('base/components/formValidation');
```

Then use Ajax to post the form data to the controller handler route:

```
$.ajax({
    url: url,
    type:'post',
    data Type:'json',
    data: $form.serialize()
```

Finally define handler functions that receive the json object sent back by the controller handler route:

```
success: function (data) {
    $form.spinner().stop();
    if (!data.success) {
        formValidation($form, data);
    } else {
        location.href=data.redirectUrl;
    }
}
```



Module 4.1: Cross-site request forgery (CSRF) protection

Implementing CSRF on SFRA forms

Require the csrf.js script:

```
var csrfProtection = require('*/cartridge/script/middleware/csrf');
```

Add a CSRF middleware method to generate tokens on the Show route:

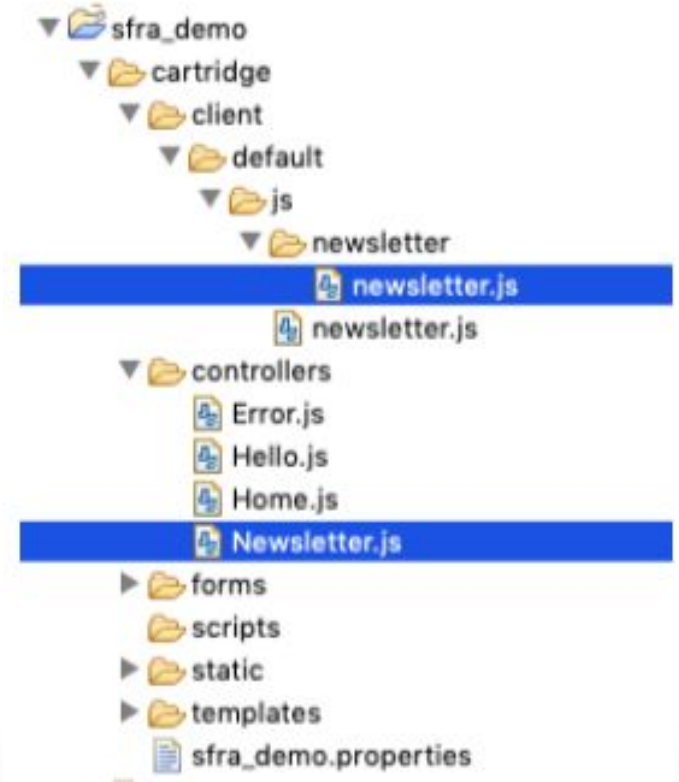
```
server.get(
    'Show',
    server.middleware.https,
    csrfProtection.generateToken,
```

Add a hidden field on the ISML form template:

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

Add a CSRF middleware method to validate the ajax request (only when using client-side submission):

```
server.post(
    'Handler',
    csrfProtection.validateAjaxRequest,
    server.middleware.https,
```



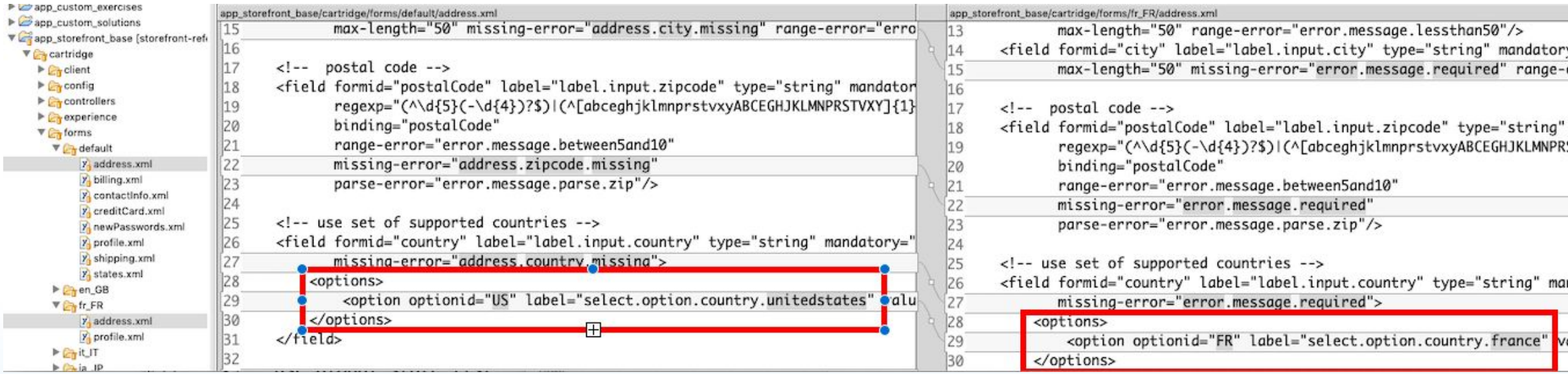
Module 4.1: Form handling using client-side JS (version 2)

1. We will walk thru NewsletterV2 controller example
2. We will look at:
   a. NewsletterV2-Show route: notice middleware CSRF token added
   b. Form ISML: newslettersignup.isml contains reference to client-side script
   c. Client-side script: newsletter.js and newsletter/newsletter.js
   d. NewsletterV2-Handler route:
      i. uses CSRF validation middleware
      ii. uses res.json() to send data back to client-side
   e. Compiling the client-side newsletter.js script
3. Questions:
   a. Why use client-side form submission and error handling?
   b. Where is the form posted from?
   c. Where is the JSON sent back from the handler received?
   d. Why use CSRF?

Module 4.1: Form localization (labels, lengths, fields, etc)

Form localization is handled using localized form metadata and ISML templates

The localization framework allows the system to pick the right metadata based on current locale. Using localized metadata, en_US states and fr_FR regions are shown properly for the corresponding locales:



Module 4.2: Custom Objects and Transactions

- Objective
  - By the end of this module you will be able to a **create** custom object to store a newsletter subscription, and **use** a transaction to commit the custom object to the database.

This module will cover:

- Custom Object creation
- Transaction methods
- Use of the Logger API

- Why does this functionality matter for implementations?
  - Usage of Custom Objects is required in many implementations
  - Transactions are required to persist data to the database
  - The Logger API is used extensively to help with support for the storefront

Module 4.2: Using Custom Objects (COs) in your application

Extend your application to store custom data

- Sometimes you won't find any System Object that can be used for your custom data. In that case, Custom Objects are the answer.
- For best practices, consult this XChange article.
- COs have quota restrictions that are enforced on every instance: build a strategy to clean them up, or face the consequences when the quota is exceeded!

Custom Objects : OK

Quota	Enforced	Used	Limit
Non-Replicable Custom Objects	Yes	35	400,000
Object Type Definitions	No	76	300
Replicable Custom Objects	Yes	0	400,000

- To use a CO, first create a custom object definition in BM:
  - Manually in Administration > Site Development > Custom Object Types, or
  - Import a CO definition metadata from a file: Administration > Site Development > Import & Export
- Second, create instances of the CO in your application:

```
var CustomObjectMgr = require('dw/object/CustomObjectMgr');
var CustomObject =
CustomObjectMgr.createCustomObject('NewsletterSubscription',newsletterForm.email.value);
```

Module 4.2: Using Transactions to persist data in your application

Custom Objects are persistent and extensible: check the documentation

Since the CO must be persisted to the database, you must use a Transaction. There are 2 flavors:

- Explicit transaction: you decide when to commit or rollback

```
var txn = require('dw/system/Transaction');
txn.begin();
try {
    //Create or modify any Persistent object using a Mgr class: CustomObjectMgr, OrderMgr, etc
    txn.commit();
} catch(e) {
    //Oops!
    txn.rollback();
}
```

- Implicit transaction: commits or rollbacks automatically

```
var txn = require('dw/system/Transaction');
txn.wrap(function(){
    //Create or modify any Persistent object
});
```

Module 4.2: Using Logger API

Not everything is calm on the Western Front

As a good developer, you should log informational messages, warnings and errors that could happen during normal execution of your code. Have a logging strategy that will guide support when your code runs in production.

Logs are available under https://<your_instance>/on/demandware.servlet/webdav/Sites/Logs

- Use dw.system.Logger

```
var Logger = require('dw/system/Logger');
```

- Log to a custom file and/or category if needed for complex integrations:

```
Logger.getLogger("adyen_checkout_log", "payment_auth_category").error(...);

Logger.getLogger("payment_auth_category").warn(...);
```

- Otherwise, messages get logged to the general info, warn, error and fatal logs:

warn-blade0-3.mon.demandware.net-0-appserver-20191029.log

error-blade0-3.mon.demandware.net-0-appserver-20191029.log

- Be aware that Business Manager supports enabling and disabling of log levels and categories
  - In production, only error and fatal should be logged to keep logs smaller
  - Turn on other levels as needed to evaluate a problem
  - On SBs, turn on other levels to evaluate the execution of your code (i.e. during system testing)

Module 4.2: Create a CO and use a transaction (version 3)

1. We will walk thru NewsletterV3 controller example
2. We will look at NewsletterV3-Handler route:
   a. Import a CO definition in BM
   b. Create a NewsletterSubscription CO instance in the code
   c. Populate the CO with data from the form
   d. Enclosed the CO creation logic inside a transaction
   e. Handle possible errors:
      i. Missing CO definition
      ii. Duplicate primary key (same email used twice)
3. Questions:
   a. Why does the CO creation cause an error without a transaction?
   b. Why do you need the keyword **custom** when initializing a CO field (i.e. CustomObject.custom.firstName = newsletterForm.fname.value)?
   c. If your use Transaction.wrap(), do you still need to do a Transaction.rollback()?
   d. What are possible reasons the transaction may fail?

Module 4.3: Middleware Events

- Objective
  - By the end of this module you will be able to a **enclose** a transaction inside a middleware event

This module will cover:

- Middleware Events in documentation
- Middleware Events implementation in SFRA

- What is this functionality and why does it matter for implementations?
  - Middleware events are necessary to control when in a middleware chain a specific business requirement (i.e. a transaction) needs to execute.

Module 4.3: SFRA Middleware Events

Do you know who is extending your route?

Consider this: your controller might be extended by another controller (really!) but you have a transaction in your controller that “should” be executed at the end of the route to account for any changes to the data.

To handle these situations, the **server module** supports middleware events your code can listen for.

To make sure your transaction is the last thing that gets executed by the route, use the **route:beforeComplete** middleware event, as follows:

```
this.on('route:BeforeComplete', function (req, res) {
    var Transaction = require('dw/system/Transaction');
    try {
        Transaction.wrap(function () {
            ... transactional code ...
        });
    } catch (e) {
        ... error handling code ...
    }
});  //end route:BeforeComplete
```

Module 4.3 Demo: Use a Middleware Event

1. We will walk thru the last Newsletter controller example (finally!)
2. In the Newsletter-Handler route, enclose the transaction and error handling inside a **route:BeforeComplete** middleware event.
3. Questions:
   a. Why do you need this middleware event?
   b. Where do you find the following code:

```
this.emit('route:BeforeComplete', req, res);
this.emit('route:Complete', req, res);
```