salesforce

Fast Path to B2C Commerce Developer Certification

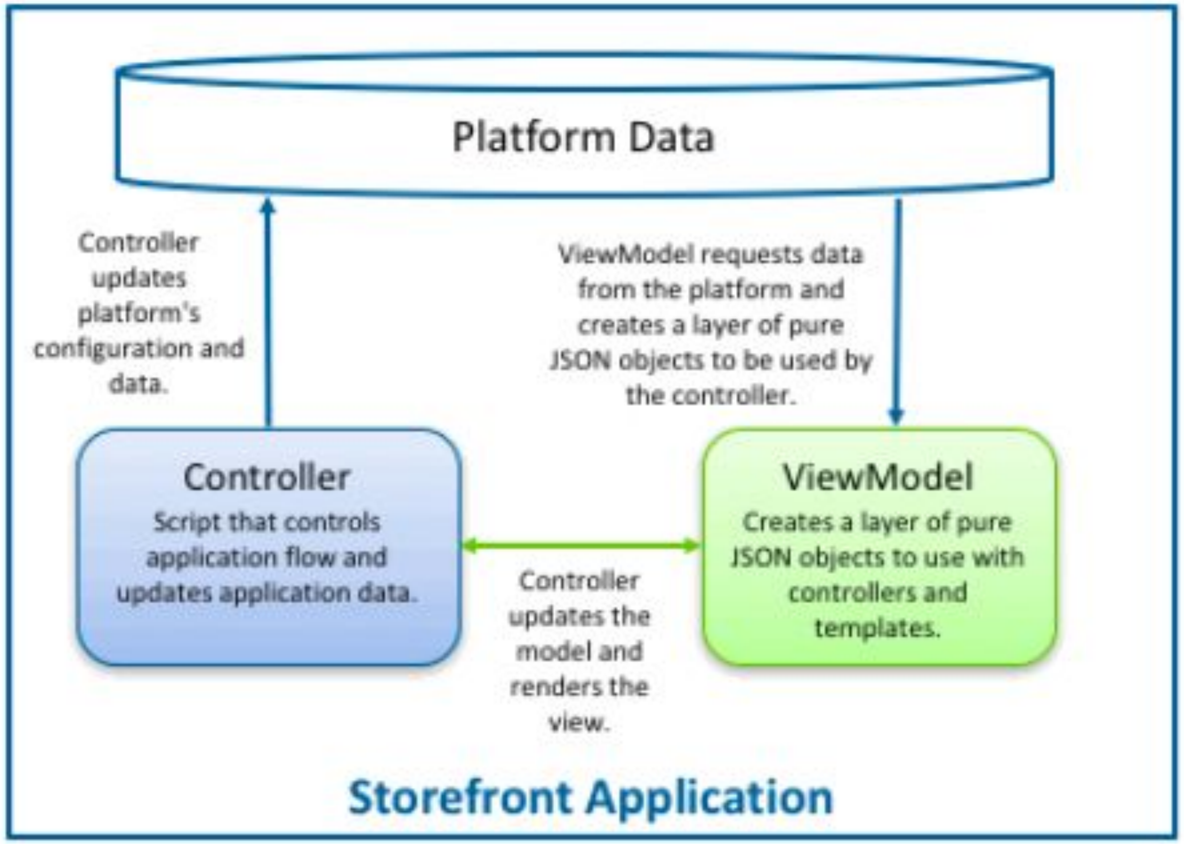
Module 3
Models, ISML & Working with Client Side Javascript

Module 3.1: Models, Decorators, Extending Models (30 mins)

- Objective
  - By the end of this module you will be able to define ***what*** a model is, ***what*** a decorator is and ***how*** it is used to extend a model.
- This module will cover:
  - What a model is?
  - What is a decorator?
  - How to extend a decorator
- What is this functionality and why does it matter for implementations?
  - SFRA data is managed by models
  - Separating business logic from data is a best practice

Module 3.1: What is a model?

- A model is the representation of the data in an MVC architecture
- SFRA models are serializable JSON objects that represent B2C Commerce system objects
- SFRA uses a variant of Model-View-Controller architecture:
  - Controllers handle information from the user, create ViewModels, and render pages
  - A ViewModel provides the data to render pages in the application and often combines data from multiple B2C Commerce script objects.
  - ViewModels are also referred to simply as models.

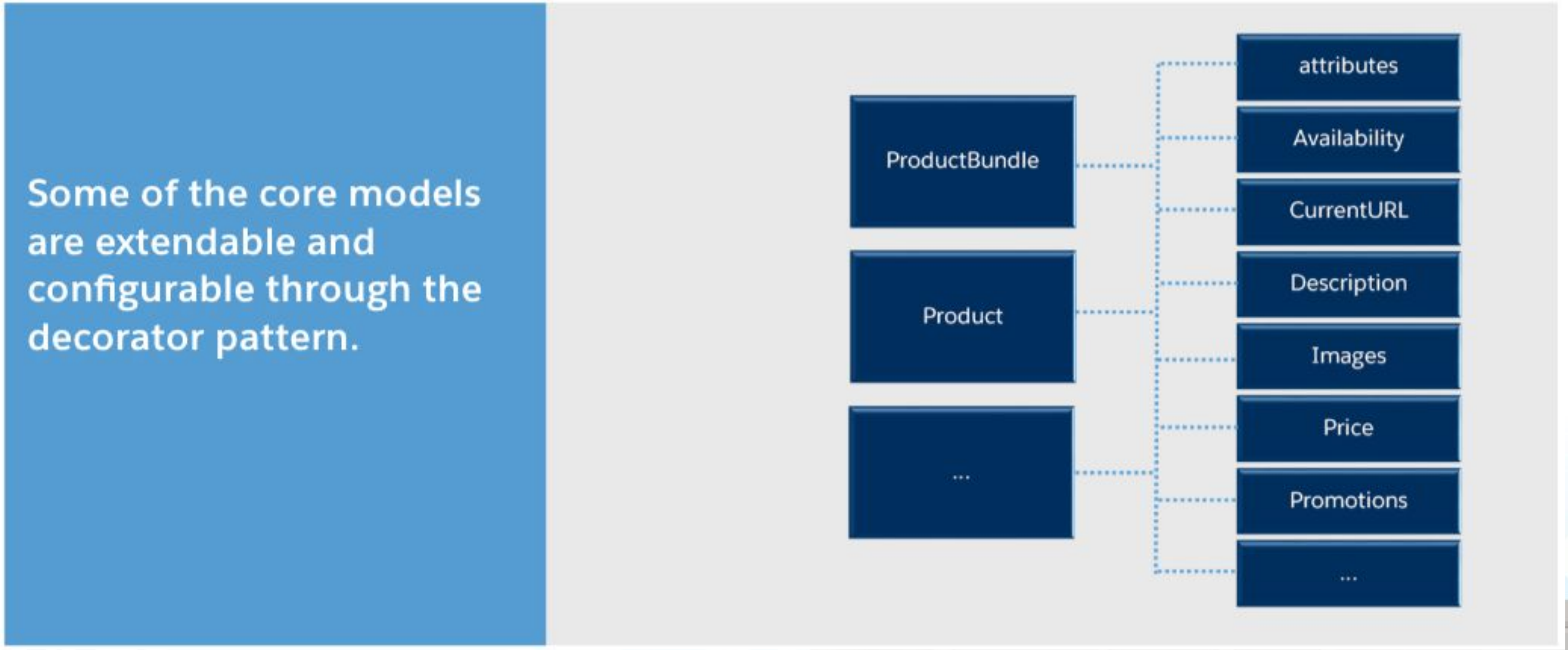

Platform Data
Controller updates platform's configuration and data.
ViewModel requests data from the platform and creates a layer of pure JSON objects to be used by the controller.
Controller
Script that controls application flow and updates application data.
ViewModel
Creates a layer of pure JSON objects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

Source: SFRA Overview - ViewModels and Contollers

Module 3.1: What is a model decorator?

Use decorators to extend models for reusability

The Decorator pattern extends (decorates) an object's behavior dynamically. The goal is to divide large numbers of data for a model into more functionally specific sub-models (aka decorators). Multiple decorators can add or override functionality to the original object.

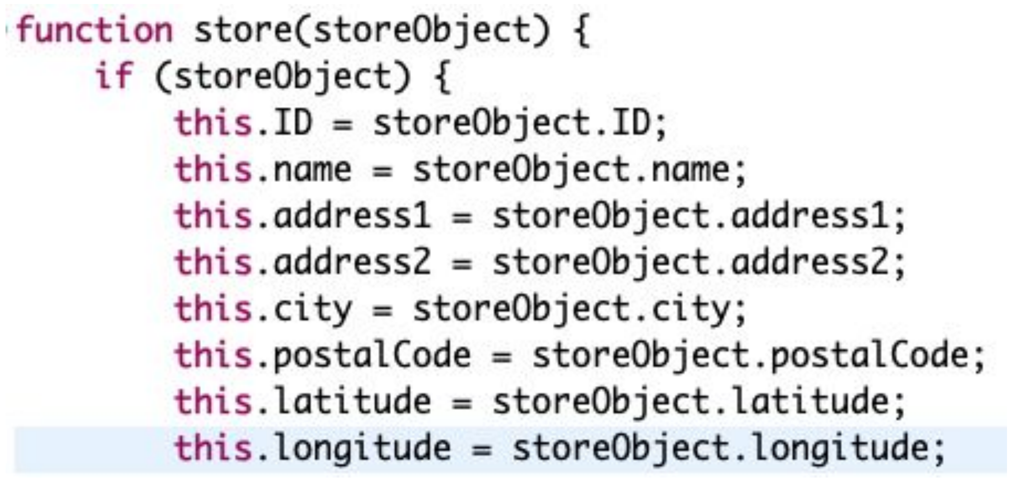


Source: SFRA Overview - ViewModels and Controllers

Module 3.1: What is a decorator?

Compare a simple model vs. a complex model

store.js is the model that represents store data, which is fairly simple:



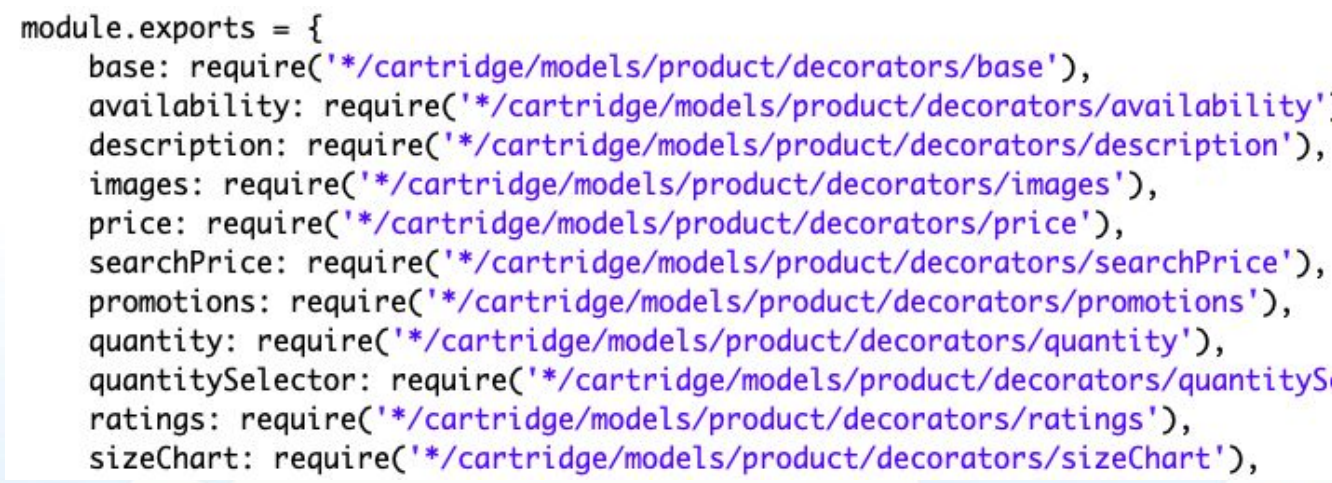
```
function store(storeObject) {
    if (storeObject) {
        this.ID = storeObject.ID;
        this.name = storeObject.name;
        this.address1 = storeObject.address1;
        this.address2 = storeObject.address2;
        this.city = storeObject.city;
        this.postalCode = storeObject.postalCode;
        this.latitude = storeObject.latitude;
        this.longitude = storeObject.longitude;
```

fullProduct.js is the model that represents a product, which is quite complex:

- It uses an index of decorators:

```
require('*/cartridge/models/product/decorators/index');
```

- Which in turn includes many decorators:



```
module.exports = {
    base: require('*/cartridge/models/product/decorators/base'),
    availability: require('*/cartridge/models/product/decorators/availability'
    description: require('*/cartridge/models/product/decorators/description'),
    images: require('*/cartridge/models/product/decorators/images'),
    price: require('*/cartridge/models/product/decorators/price'),
    searchPrice: require('*/cartridge/models/product/decorators/searchPrice'),
    promotions: require('*/cartridge/models/product/decorators/promotions'),
    quantity: require('*/cartridge/models/product/decorators/quantity'),
    quantitySelector: require('*/cartridge/models/product/decorators/quantityS
    ratings: require('*/cartridge/models/product/decorators/ratings'),
    sizeChart: require('*/cartridge/models/product/decorators/sizeChart'),
```

Source: [SFRA Overview - ViewModels and Contollers]

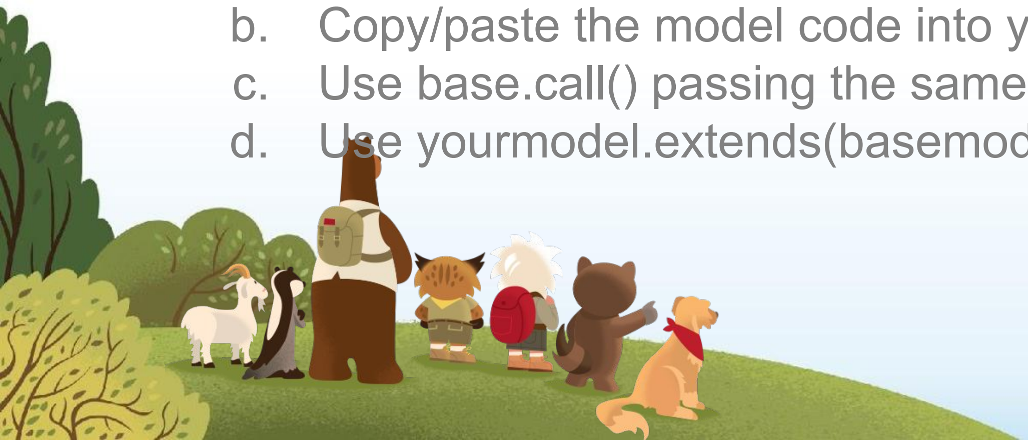
Module 3.1: Demo: Extend the Product Model to Include ATS (available to sell) in availability decorator

1. We will walk thru modifying the availability decorator to demonstrate extending a product model.
2. This should test your knowledge
   a. What models are located
   b. What a decorator is and how to extend it
   c. How to verify the JSON data returned to the browser contains the updated model
3. Later in the webinar we tackle updating and compiling the client-side JavaScript to surface this model update.

Time to Test your Knowledge!

Module 3.1 - All Questions at the end

1. What is a model? (choose 2 out of 4)
   a. It is a function that only the controller calls
   b. It is the object that represents the data the controller sends to the view
   c. It is a serializable JSON object
   d. It is a model in the modules folder
2. What is a decorator? (choose 1)
   a. It is an object that decorates the ISML page
   b. It is a subset of the model that makes it easier to extend the model
   c. It is a hardcoded JSON file that decorates the model
3. How do you extend a model? (choose 2)
   a. Use module.superModule to identify the model to extend
   b. Copy/paste the model code into your cartridge
   c. Use base.call() passing the same parameters that the base model needs
   d. Use yourmodel.extends(basemodel)

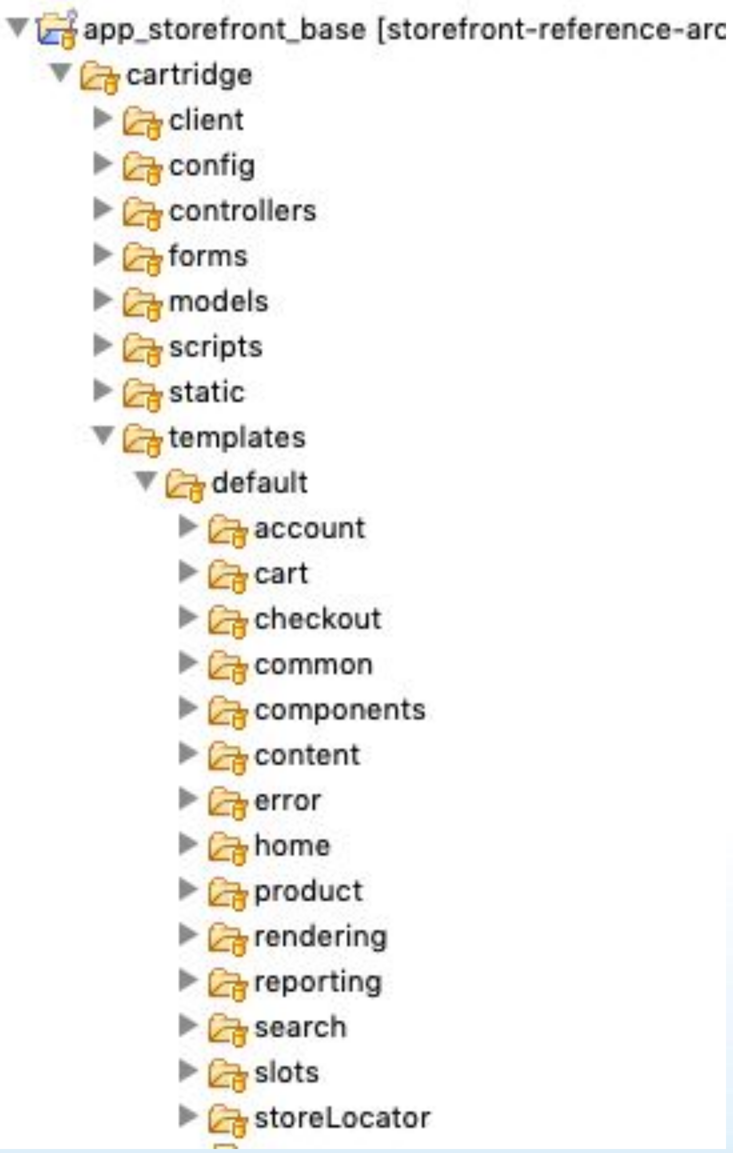


Module 3.2: ISML (40 mins)

- Objective
  - By the end of this module you will be able to understand the best practices around ISML usage in SFRA
- This module will cover:
  - What is ISML and templates?
  - Best Practices on ISML usage in SFRA
  - Demo ISML templates and tags
- What is this functionality and why does it matter for implementations?
  - Easier loop management
  - ISML template to render a page
  - Reusability of templates

Module 3.2: What is ISML and templates?

- ISML stands for Internet Store Markup Language
- It is a set of proprietary **ISML tags** that are used inside ISML templates, aka **templates**
- Templates render the view in the SFRA MVC architecture
- Templates:
  - Are rendered by controllers
  - Receive JSON data from the model
  - Process this data by using a mixture of HTML and ISML tags
  - Can use javascript and css by requiring other files



```
▼ app_storefront_base [storefront-reference-arc
  ▼ cartridge
    ▶ client
    ▶ config
    ▶ controllers
    ▶ forms
    ▶ models
    ▶ scripts
    ▶ static
    ▼ templates
      ▼ default
        ▶ account
        ▶ cart
        ▶ checkout
        ▶ common
        ▶ components
        ▶ content
        ▶ error
        ▶ home
        ▶ product
        ▶ rendering
        ▶ reporting
        ▶ search
        ▶ slots
        ▶ storeLocator
```

Module 3.2: Best Practices on ISML usage in SFRA

There is an internal ban on certain ISML tags. You can still use them, but here are the caveats:

- **iscache**: use instead the cache.js middleware methods in controllers:

```
server.get('Show', cache.applyPromotionSensitiveCache, …)
```

- **isset:** use it only to set a variable used by an isinclude tag, not for complex expressions.

```
<isset name="store" value="${pdict.store}" scope="page"/>
<isinclude template="storeLocator/storeDetails" />
```

- **isscript:** use it only to add client-side JS and CSS to an ISML template:

```
<isscript>
    var assets = require('*/cartridge/scripts/assets.js');
    assets.addJs('/js/productTile.js');
    assets.addCss('/css/homePage.css');
</isscript>
```

- **ismodule:** use the single custom tag defined in modules.isml to render content assets. Any other ismodule has become a script that you invoke.

Module 3.2: Best Practices on ISML usage in SFRA (cont.)

- **iscontent:** use controller code like

  ```
  resetPasswordEmail.setContent(content, 'text/html', 'UTF-8’);
  ```

- **iscookie**: use the function getCookie() in cookie.js. There is no equivalet setCookie() method. Use:

  ```
  window.localStorage.setItem('previousSid', previousSessionID);
  ```

- **isredirect:** use code in controller like:

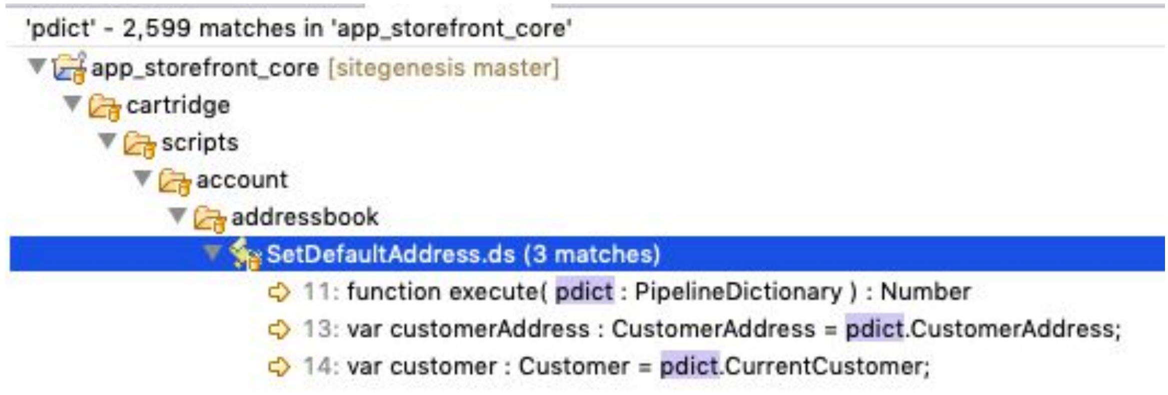
  ```
  window.location.href = err.responseJSON.redirectUrl;
  ```

- **isstatus:** use code in controller like:

  ```
  res.setStatusCode(500);
  res.setStatus(404);
  ```

Module 3.2: Use of pdict in SFRA

- pdict stands for pipeline dictionary (legacy demandware terminology
- It is a scope that a pipeline or controller can use to store data for the currently executing request
- Templates use pdict to display data to the page
- It is still used extensively in the SiteGenesis Reference Application



```
'pdict' - 2,599 matches in 'app_storefront_core'
▼ app_storefront_core [sitegenesis master]
  ▼ cartridge
    ▼ scripts
      ▼ account
        ▼ addressbook
          ▼ SetDefaultAddress.ds (3 matches)
              11: function execute( pdict : PipelineDictionary ) : Number
              13: var customerAddress : CustomerAddress = pdict.CustomerAddress;
              14: var customer : Customer = pdict.CurrentCustomer;
```

- SFRA controllers uses the [Response JS class] to add data to the pdict:
  - Controller adds data to the view by passing a JSON object:

    res.setViewData({ param1: 'Data that you want on the template' });
  - ISML template finds the data in the pdict: ${pdict.param1}
  - Usually, full models are passed to the view instead of using ViewData:

    var template = new Template('storeLocator/storeLocatorResults');
    return template.render(context).text;
  - In the template, the context that was passed becomes the pdict:
    <isloop items="${pdict.stores.stores}" var="store" status="loopState">

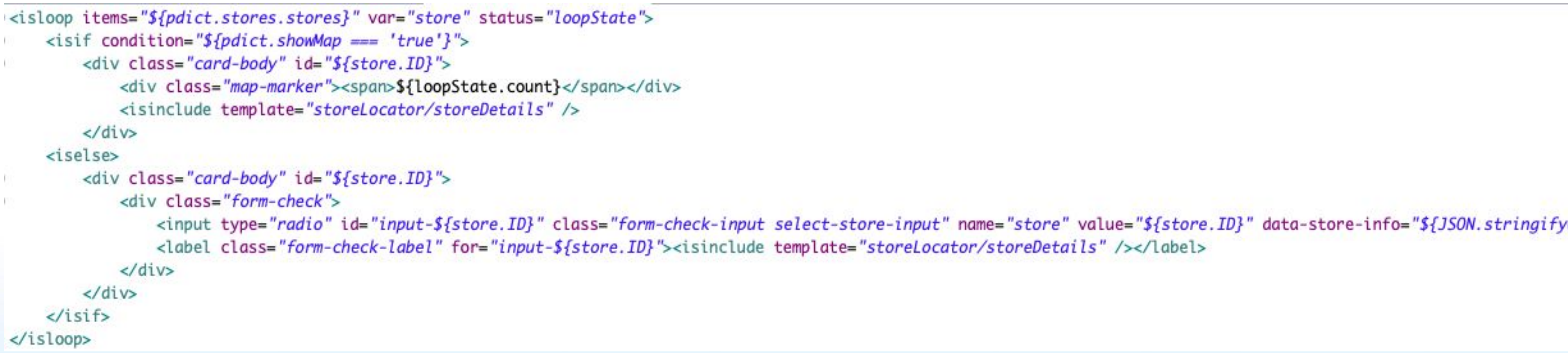
Module 3.2: ISML tags used in SFRA

The following are still in active use, and usually used together:

- **isactivedatahead, isactivedatacontext and isobject:** use them to collect active data. See documentation.

- **isloop, isnext, isbreak, iscontinue:** create and manage loops.

- **isif, iselseif, iselse:** manage conditional logic. Example containing a loop and conditional logic:



```
<isloop items="${pdict.stores.stores}" var="store" status="loopState">
    <isif condition="${pdict.showMap === 'true'}">
        <div class="card-body" id="${store.ID}">
            <div class="map-marker"><span>${loopState.count}</span></div>
            <isinclude template="storeLocator/storeDetails" />
        </div>
    <iselse>
        <div class="card-body" id="${store.ID}">
            <div class="form-check">
                <input type="radio" id="input-${store.ID}" class="form-check-input select-store-input" name="store" value="${store.ID}" data-store-info="${JSON.stringify
                <label class="form-check-label" for="input-${store.ID}"><isinclude template="storeLocator/storeDetails" /></label>
            </div>
        </div>
    </isif>
</isloop>
```

Module 3.2: ISML tags used in SFRA (cont.)

- **isslot:** allows the insertion of a content slot in an template. Content slots are:
  - Areas of a template that the merchant customizes via Business Manager
  - Requires a rendering template that you create
  - To learn how to use content slots, refer to the Learning Path > Grow your Expertise > Fast Path to certification

  Module 6 - Content Slots

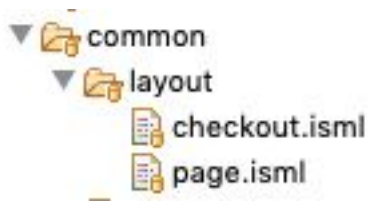
  ```
  <isslot id="home-main-m" description="Main home page slot." context="global" />
  ```

- **isprint**: use it to print strings, formatted date and times. There is extensive documentation on this tag:

  ```
  <td class="quantity"><isprint value="${tier.quantity}" style="INTEGER" /></td>
  ```

- **isselect:** not used in SFRA, but it is an enhancements to the HTML<select> tag

Module 3.2: ISML decorators (not the same as model decorator!

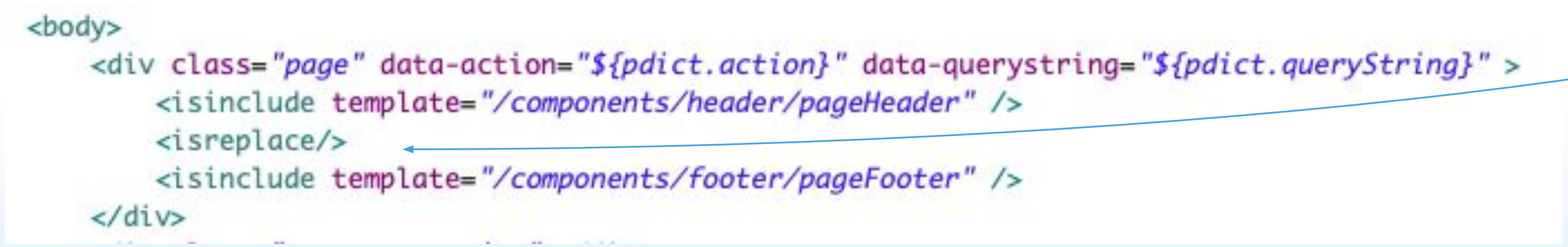


```
▼ common
  ▼ layout
      checkout.isml
      page.isml
```

- A decorator template is an “encapsulating” template reused by another template
- There are only 2 decorator templates used in SFRA (a lot less than in SG)
  - checkout.isml does not contain navigation
  - page.isml does
- **isdecorate** is used to specify a decorator to use. homePage.isml uses the *page* decorator, as follows:

```
<isdecorate template="common/layout/page">
        … the stuff you want to show inside the header and a footer ...
</isdecorate>
```

- **isreplace** is used inside the *page* decorator template to insert homePage content inside header and footer:



```
<body>
    <div class="page" data-action="${pdict.action}" data-querystring="${pdict.queryString}" >
        <isinclude template="/components/header/pageHeader" />
        <isreplace/>
        <isinclude template="/components/footer/pageFooter" />
    </div>
```

Module 3.2: ISML isinclude tags

- **isinclude** is another example of reusability in SFRA
- There are two usages for **isinclude:**
  - Include another template:

    `<isinclude template="/product/components/pricing/default" />`
  - Call another controller, and include its output on the current page:

    `<isinclude url="${URLUtils.url('Cart-MiniCart')}" />`
- This last isinclude is also called a r**emote include**, and it allows you to create pages that have different **caching** for different areas of the page. For example:
  - Home-Show controller renders homePage.isml with a 24 hours cache
  - homePage.isml uses decorator template page.isml
  - page template uses a remote include to Cart-MiniCart controller
  - This controller produces the mini cart, which is NOT cached since it is specific to the current user

Module 3.2: Localizing ISML templates

- You can localize templates as needed for specific locales:
  - **templates/default** contains the templates that are used by the default locale
  - **templates/fr_FR** contains just those templates that require rendering for French in France (as opposed to French in Canada: fr_CA). Locales are always defined by language_REGION.

- You should localize all strings used in templates using Resource Bundles:
  - **templates/resources** folder to locate the
  - Localize strings using files with a locale suffix: account_**fr_FR**.properties (again, language_REGION)
  - Use the dw.util.Resource api to render localized strings:

    ```
    ${Resource.msg('msg.no.saved.addresses','address',null)}
    ```
  - The msgf method allows you to pass parameters to the localized string:

    ```
    ${Resource.msgf('label.product.ratings', 'common', null, product.rating)}
    ```

Module 3.2: Demo
Passing data via setViewData and using the new model in ISML

In this demo, I'll showcase 3 things:

- Copying ISML productDetails template from base (no extending here)
- Updating to include new client-side JavaScript
- Walkthru of ISML code

In the next section we cover the compiling and uploading of the client-side file so as to complete the functionality of the availability model that we started in Module 3.1.

Time to Test your Knowledge!

1. What is the use of an decorator template? (choose 2)
    a. To decorate a model passed to the template
    b. To reuse another template that contains the header and footer
    c. To use the decorator pattern on ISML templates
2. Which 2 statements are true about the usage of pdict?
    a. Pipeline Dictionary, but no longer used on SFRA
    b. Use pdict on ISML templates to refer to the data passed in
    c. pdict is available in SFRA controllers, just like it was in SG controller
    d. Use ${pdict.variableName} in a template to access data passed from the controller
3. Which 2 tags allow you to reuse other templates?
    a. isinclude
    b. isdecorate
    c. isreuse
    d. iscache
4. How can a controller call another one? Choose 2.
    a. Controller1 renders an ISML template, which does a remote include to Controller2
    b. Controller1 sends a JSON object containing a redirectURL for Controller2 to the client-side
    c. Controller1.call(Controller2-Start);
    d. Not possible

Module 3.3: Building SCSS and client-side JS (20 mins)

- Objective
  - By the end of this module you will be able to **override** a client-side JavaScript file, **compile** it using npm scripts using package.json and webpack.config.js and **debug** the file on the browser
- This module will cover:
  - Modifying the a client side script to show variant availability
  - Using package.json to compile the client-side JavaScript
  - Using webpack.config.js to compile the client-side in non-minified mode to allow for browser debugging
- What is this functionality and why does it matter for implementations?
  - In SFRA, you use node to compile CSS and JavaScript for the client-side
  - Client-side behaviour that happens after page load is often changed

Module 3.3: Client-side JavaScript in SFRA

Dynamic behaviour on your storefront needs client-side code

SFRA uses client-side JavaScript code extensively.

The following technologies are used:

- jQuery
- Bootstrap
- Ajax
- SCSS
- Node.js
- NPM (Node Package Manager)
- Webpack

These technologies are not part of the certification test, but how they are used is important to understand how the SFRA client-side works.

Module 3.3: Node use in SFRA

Compiling the client-side code requires Node.js and NPM installed in your system

The storefront-reference-architecture repository (contains SFRA cartridges) requires node and npm to compile JavaScript, SCSS and upload cartridges.

It is encouraged you use the latest version of node, but it may be difficult to upgrade if you have an earlier version.

- Node version: node -v → v8.9.4
- NPM version: npm -v → 6.11.3

Once you have node and npm installed, use npm to install node packages on the SFRA folder:

- Change to storefront-reference-architecture folder
- At command line: npm install
- The command generates a node_modules on the root folder

Correct any errors you might get by using stackoverflow to diagnose the problems.

Module 3.3: Using an NPM package.json file

All npm packages contain a package.json at the root folder

This file can contain a lot of metadata about your project. But mostly it will be used for two things:

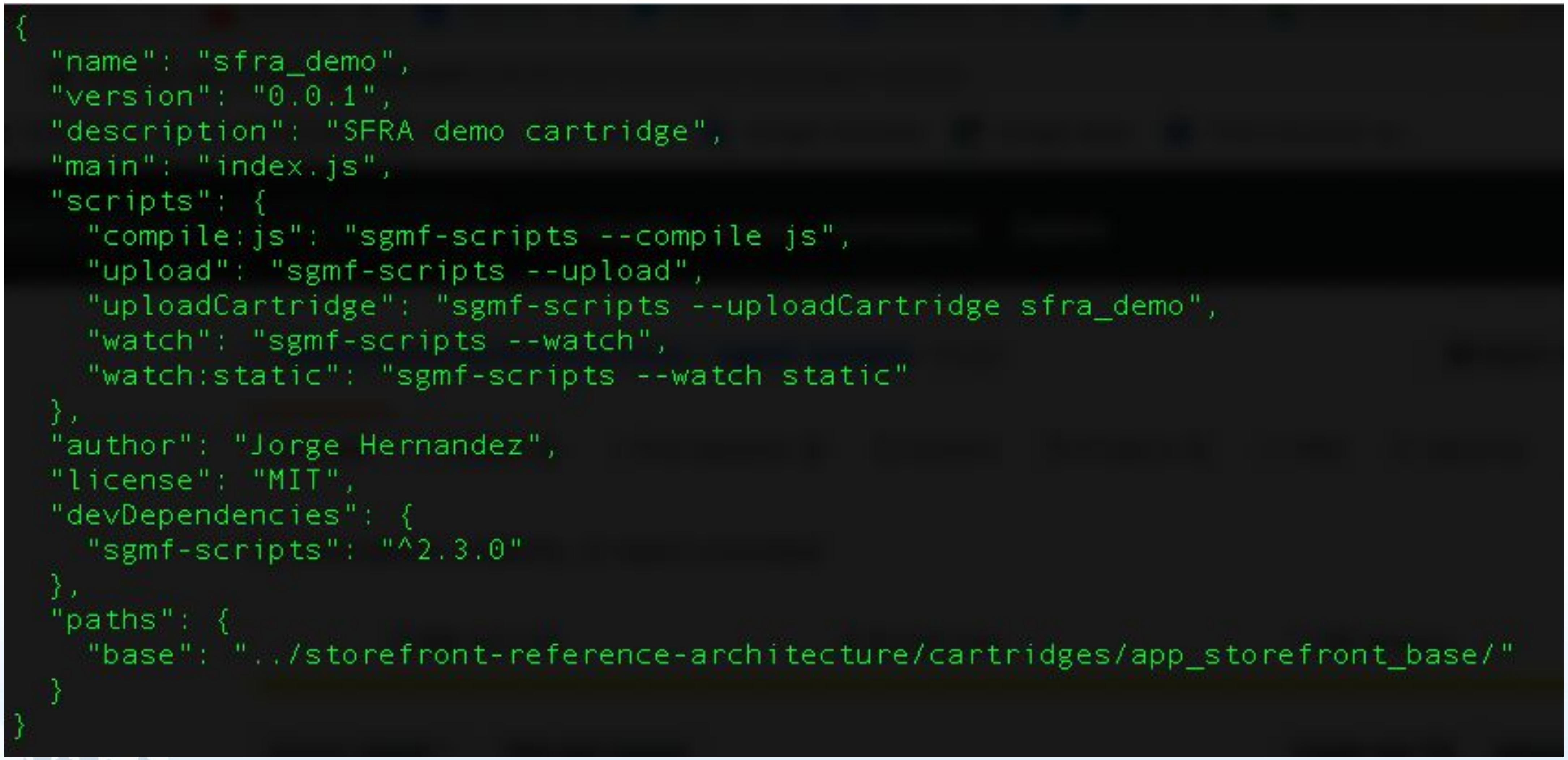
- Managing dependencies of your project
- Scripts, that helps in generating builds, running tests and other stuff in regards to your project
- Define a base path to located storefront-reference-architecture folder

SFRA contains a fairly comprehensive package.json, but you own project only needs a few scripts:

- compile:js → compile client-side JavaScript
- compile:scss → generate css files based on scss
- uploadCartridge → uploads cartridges defined in dw.json
- watch → watches the folder for changes, and automatically uploads them

Module 3.3: package.json for the sfra_demo project

Notice the base path: this is a relative path to the SFRA base directory



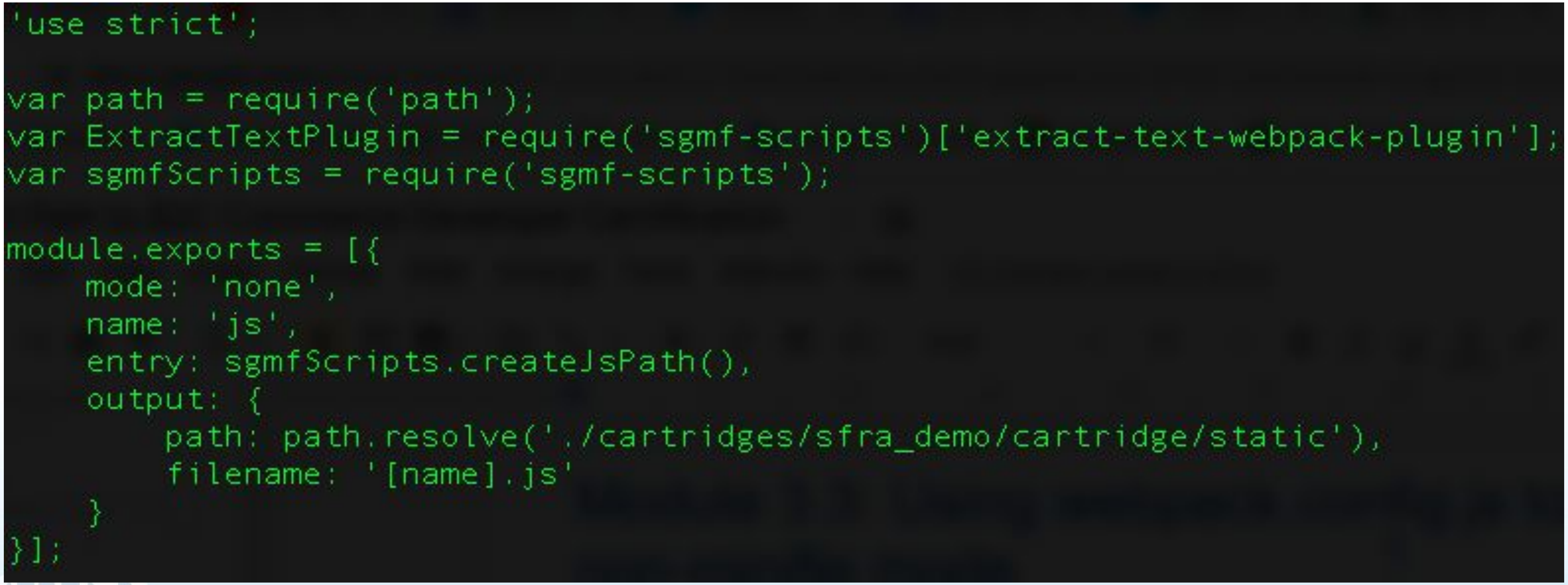
```
{
  "name": "sfra_demo",
  "version": "0.0.1",
  "description": "SFRA demo cartridge",
  "main": "index.js",
  "scripts": {
    "compile:js": "sgmf-scripts --compile js",
    "upload": "sgmf-scripts --upload",
    "uploadCartridge": "sgmf-scripts --uploadCartridge sfra_demo",
    "watch": "sgmf-scripts --watch",
    "watch:static": "sgmf-scripts --watch static"
  },
  "author": "Jorge Hernandez",
  "license": "MIT",
  "devDependencies": {
    "sgmf-scripts": "^2.3.0"
  },
  "paths": {
    "base": "../storefront-reference-architecture/cartridges/app_storefront_base/"
  }
}
```

Module 3.3: Using webpack.config.js to compile JavaScript in non-minified mode

Its main purpose is to bundle JavaScript files for usage in a browser.

mode: ‘none’ → compile for debugging using Developer Tools on the browser

mode: ‘production’ → compile in minified mode for production



```
'use strict';

var path = require('path');
var ExtractTextPlugin = require('sgmf-scripts')['extract-text-webpack-plugin'];
var sgmfScripts = require('sgmf-scripts');

module.exports = [{
    mode: 'none',
    name: 'js',
    entry: sgmfScripts.createJsPath(),
    output: {
        path: path.resolve('./cartridges/sfra_demo/cartridge/static'),
        filename: '[name].js'
    }
}];
```

Module 3.3: Overriding a base client-side script on your cartridge

Override client-side code from the base when you need new front-end functionality

When we extended the availability model, we were able to get the new data passed in as JSON.

However, the ISML page does not refresh when we changed variants, therefore the variant’s ATS is not showing.

We must override the base client-side code such that the the function that handles the availability message update uses the new ATS message passed in the model.

The analysis required to find the right method to override, and the client-side JavaScript code implementation is beyond the scope of this webinar. But stay tuned for a complementary webinar!

Your code can refer to the base SFRA code as follows:

```
var base = require('base/product/base');
```

First “base” refers to the base path (SFRA base cartridge), **second** “base” refers to the base.js script.

Then the new functionality can override the base as follows:

```
base.updateAvailability = updateAvailability;

module.exports = base;
```

Module 3.3: Building SCSS and client-side JS

In this demo we'll override the client-side so that the proper ATS message will show:

- Use a new file to override the updateAvailability method from the base
- Make sure webpack.config.js is set up for non-minified compilation
- On the command line:
  - npm install to build the node_modules
  - npm run compile:js to compile the JS
  - npm run uploadCartridge (only if your dw.json file is setup correctly)
- On the browser:
  - Refresh the SFRA page
  - Open the Developer Tools window
  - Put a breakpoint on the new updateAvailability method
  - Step thru the code to see if the right message is created
  - Show the generated page with the new ATS message

Time to Test your Knowledge!

Module 3.3 - All Questions at the end

1. Where is the client-side JavaScript code?
   a. In the cartridge/client folder
   b. In the cartridge/static folder
   c. It is outside the cartridge folder
2. Do you have to use UX Studio to upload code? T/F
3. To compile client-side JavaScript, what file/folder don't you need?
   a. package.json
   b. webpack.config.js
   c. node_modules folder
   d. dw.json
4. Can you extend the client-side JavaScript functions from the base? T/F
5. Where do you specify a base path for the client-side code?
   a. dw.json
   b. Use a require('app_storefront_base') on the client-side code
   c. package.json
   d. webpack.config.js