Module 4 - Tag management systems

This toolkit is designed for Professional Developer Exam Aspirants. There are **six** Modules. Study Each module per week to stick to schedule. Technical Parts of applications are depicted in Videos, you can learn more about them from experience League. You can visit Get prep page to understand the contents and anticipate the learning journey.
This is Professional Exam, Developer toolkit Module 4. This module contains four sections.

4.1 Tags Overview

Tags in Adobe Experience Platform are the next generation of tag management capabilities from Adobe. Tags give customers a simple way to deploy and manage all of the analytics, marketing, and advertising tags necessary to power relevant customer experiences.

Tags empower anyone to build and maintain their own integrations, called *extensions*. These extensions are available to Adobe Experience Cloud customers in an app-store experience so they can quickly install, configure, and deploy their tags.

Tags are offered to Adobe Experience Cloud customers as an included value-add feature.

Key benefits

- Faster time to value.
- Trustworthy data through centralized collection, organization, and delivery using data elements.
- Compelling experiences through the integration of data and marketing technology using rule builder.

Key features

Extensions

An extension is a package of code (JavaScript, HTML, and CSS) that extends the tags functionality. Build, manage, and update your integrations using a virtually self-service interface. You can think of extensions as apps you use to achieve your tasks.

Extension catalog

Browse, configure, and deploy marketing/advertising tools built and maintained by independent software vendors.

Rule builder

Create robust rules that combine multiple events, sequenced in the way that you determine using if/then logic with conditions and exceptions. Rules provide options for:

- Events
- Conditions
- Exceptions
- Actions

The rule builder includes real-time error checking and syntax highlighting for your custom code.

When the criteria outlined in your rules are met and conditions are satisfied, the actions you define are executed in order.

Data elements

Collect, organize, and deliver data across web-based marketing and advertising technology.

Enterprise publishing

The publishing process enables teams to publish code to pages. Different people can create an implementation, approve it, and publish it on your pages.

- Changes to your code are encapsulated within the libraries you define.
- You specify where and when you want your code deployed.
- Multiple libraries can be built in parallel by different teams.
- Unlimited development environments.
- A deliberate, permission-based process for merging libraries together.

Open APIs

Automate implementations of individual technologies or a group of technologies.

- Tags interact with the Reactor API.
- Deployments can be automated through APIs.
- Integrate the APIs with your own internal systems.
- You can build your own user interface if desired.

Light, modular container tag

The content of your container is minified, including your custom code. Everything is modular. If you don't need an item, it is not included in your library. The result is an implementation that is fast and compact. See Minification.

Other highlights

Tags provide several improvements over similar systems, including:

- No use of `document.write ()` where Chrome doesn't allow it.
- The Page Top and Page Bottom rules are bundled into the main library to minimize unnecessary HTTP calls.

- Custom action scripts within a rule can be loaded in parallel, but are executed sequentially.
- If you avoid Page Top and Page Bottom rules, the code is mostly asynchronous, with a path to getting fully async.

4.2 Tag Management

Adobe Experience Platform Launch makes it easy to manage tags, and it provides innovative tools for collecting and distributing data across digital marketing systems.

Tag, you’re it.

Tags are at the heart of any analytics practice. They make it possible for you to collect data, which become the insights you need. The challenge is deploying and managing those tags efficiently.

Built by the same engineers that built Dynamic Tag Management (DTM) back in 2013, Launch is our next-generation tag management system that unifies our entire marketing technology ecosystem. With Launch, third-party developers can build, maintain, and continuously update their integrations with Adobe Experience Cloud, meaning you can deploy both Adobe and third-party apps with ease — and capture and use customer data as you please.

Extensive Extensions catalog

Browse, configure, and deploy marketing technology built and maintained directly by independent software vendors.

Redesigned rule builder

Integrate the data and functionality of marketing and ad technologies to unify different products.

Open APIs

Automate baseline implementations for one tool or several.

Component-based publishing

Publish only what you intend by bundling the rules, data elements, and extensions that make up a library.

4.3 Packet analyzers

Packet analyzers let you view the data sent by your implementation to Adobe data collection servers.

Similar to the Adobe Experience Cloud debugger, a packet monitor shows what data parameters are being passed in an image request; however, packet monitors provide added functionality:

- View custom link tracking image requests
- View image requests using implementation methods other than JavaScript, such as hard-coded image requests or Appmeasurement

To view Analytics requests, filter outgoing requests using "b/ss".

In very rare cases, the debugger will report an image request although no request makes it to Adobe's Analytics processing servers. Using a packet monitor is a great way to be 100% sure that a specific image request is being fired successfully.

While Adobe does not provide an official packet monitor, there are a wide range of them on the internet. The following are some packet monitors others have found useful.

TIP

These lists are not meant to be comprehensive, but rather information on frequently used monitors.

Firefox	Internet Explorer	Chrome	
Observe Point (tag viewer)	HttpWatch	Observe Point (tag viewer)	Cha
HttpFox		Chrome Developer Tools	Fid
Tamper Data		Firebug Lite	Wir
HttpWatch			
Firebug			

NOTE

Adobe does NOT support or troubleshoot any issues you experience with these packet monitors. Consult the packet monitor's originating site for assistance.

Typical HTTP response status codes

When AppMeasurement sends data to Adobe data collection servers, servers respond with a response status code.

- **200 OK**: The most common response from data collection servers. The image request was successfully received and a transparent image was returned.
- **302 FOUND**: There are a couple possible reasons to receive this response:

    - The first image request of a visitor: A redirect occurs if a user visits your site for the first time. This redirect is to obtain a visitor cookie. It does not affect data collection.
    - Integration between Comscore and Adobe: If your organization uses a Comscore/Analytics integration, each image request always results in a 302 response.
- **404 NOT FOUND**: This response means that the image request was not found, and data is not sent to Adobe data collection servers. This response is also possible when hardcoded image requests are not formatted correctly. Work with the individual or team who implemented Analytics to resolve this issue.

NS_BINDING_ABORTED in response codes

This message occurs because the link tracking image request is designed to let the browser proceed to the next page before waiting for a response from the Adobe data collection servers.

Adobe's response to the image request is simply a blank 1x1 transparent image, which is not relevant to the content of the page. If you see a line item in your packet monitor from Adobe, either with a **200 OK** response or an **NS_BINDING_ABORTED** response, the data has reached Adobe's servers. There is no need to have the page wait any longer.

Packet monitors integrated as a plug-in rarely see the full response. They tend to see the request as aborted because the full response was not received. These monitors also rarely make a distinction between whether it was the request or response that was aborted. A stand alone packet monitor typically has more detailed messages and reports the status more accurately. For example, a user may get a message in *Charles* saying "Client closed connection before receiving entire response." This means the data did reach our servers, just the browser moved on to the next page before the 1x1 pixel was received.

If an external packet monitor reports that the data collection request is aborted, rather than the response, this is a cause for concern. Adobe Customer Care can provide help in troubleshooting.

More help on this feature

- Data collection query parameters
- Hash collisions
- Legacy Adobe Experience Cloud Debugger

4.4 Satellite object reference

This document serves as a reference for the client-side `_satellite` object and the various functions you can perform with it.

`track`

Code

```
_satellite.track(identifier: string [, detail: *] )
```

Copy
Toggle Text Wrapping

Example

```
_satellite.track('contact_submit', { name: 'John Doe' });
```

Copy
Toggle Text Wrapping

`track` fires all rules using the Direct Call event type that has been configured with the given identifier from the Core tag extension. The above example triggers all rules using a Direct Call event type where the configured identifier is `contact_submit`. An optional object containing related information is also passed. The detail object can be accessed by entering `%event.detail%` within a text field in a condition or action or `event.detail` inside the code editor in a Custom Code condition or action.

`getVar`

Code

```
_satellite.getVar(name: string) => *
```

Copy
Toggle Text Wrapping

Example

```
var product = _satellite.getVar('product');
```

Copy
Toggle Text Wrapping

In the example provided, if a data element exists with a matching name, the data element’s value will be returned. If no matching data element exists, it will then check to see if a custom variable with a matching name has previously been set using `_satellite.setVar()`. If a matching custom variable is found, its value will be returned.

NOTE

You can use percent (`%`) syntax to reference variables for many form fields in your tag implementation, reducing the need to call `_satellite.getVar()`. For example, using `%product%` will access the value of the product data element or custom variable.

When an event triggers a rule, you can pass the rule's corresponding `event` object into `_satellite.getVar()` like so:

```
// event refers to the calling rule's event
var rule = _satellite.getVar('return event rule', event);
```
Copy
Toggle Text Wrapping

setVar

Code

```
_satellite.setVar(name: string, value: *)
```
Copy
Toggle Text Wrapping

Example

```
_satellite.setVar('product', 'Circuit Pro');
```
Copy
Toggle Text Wrapping

`setVar()` sets a custom variable with a given name and value. The value of the variable can later be accessed using `_satellite.getVar()`.

You may optionally set multiple variables at once by passing an object where the keys are variable names and the values are the respective variable values.

```
_satellite.setVar({ 'product': 'Circuit Pro', 'category': 'hobby' });
```
Copy
Toggle Text Wrapping

getVisitorId

Code

```
_satellite.getVisitorId() => Object
```
Copy
Toggle Text Wrapping

Example

```
var visitorIdInstance = _satellite.getVisitorId();
```

Copy
Toggle Text Wrapping

If the Adobe Experience Cloud ID extension is installed on the property, this method returns the Visitor ID instance. See the [Experience Cloud ID Service documentation]() for more information.

`logger`

Code

```
_satellite.logger.log(message: string)
```

Copy
Toggle Text Wrapping

```
_satellite.logger.info(message: string)
```

Copy
Toggle Text Wrapping

```
_satellite.logger.warn(message: string)
```

Copy
Toggle Text Wrapping

```
_satellite.logger.error(message: string)
```

Copy
Toggle Text Wrapping

Example

```
_satellite.logger.error('No product ID found.');
```

Copy
Toggle Text Wrapping

The `logger` object allows for a message to be logged to the browser console. The message will only be displayed if tag debugging is enabled by the user (by calling `_satellite.setDebug(true)` or using an appropriate browser extension).

Logging Deprecation Warnings

```
_satellite.logger.deprecation(message: string)
```

Copy
Toggle Text Wrapping

Example

```
_satellite.logger.deprecation('This method is no longer supported, please use
[new example] instead.');
```

Copy

Toggle Text Wrapping

This logs a warning to the browser console. The message is displayed whether or not tag debugging is enabled by the user.

cookie

`_satellite.cookie` contains functions for reading and writing cookies. It is an exposed copy of the third-party library js-cookie. For details on more advanced usage of this library, please review the js-cookie documentation.

Set a cookie

To set a cookie, use `_satellite.cookie.set()`.

Code

```
_satellite.cookie.set(name: string, value: string[, attributes: Object])
```

Copy
Toggle Text Wrapping
NOTE

In the old `setCookie` method of setting cookies, the third (optional) argument to this function call was an integer that indicated the cookie's expiration time in days. In this new method, an "attributes" object is accepted as a third argument instead. In order to set an expiration for a cookie using the new method, you must provide an `expires` property in the attributes object and set it to the desired value. This is demonstrated in the example below.

Example

The following function call writes a cookie that expires in one week.

```
_satellite.cookie.set('product', 'Circuit Pro', { expires: 7 });
```

Copy
Toggle Text Wrapping

Retrieve a cookie

To retrieve a cookie, use `_satellite.cookie.get()`.

Code

```
_satellite.cookie.get(name: string) => string
```

Copy
Toggle Text Wrapping

Example

The following function call reads a previously set cookie.

```
var product = _satellite.cookie.get('product');
```
Copy
Toggle Text Wrapping

Remove a cookie

To remove a cookie, use `_satellite.cookie.remove()`.

Code

```
_satellite.cookie.remove(name: string)
```
Copy
Toggle Text Wrapping

Example

The following function call removes a previously set cookie.

```
_satellite.cookie.remove('product');
```
Copy
Toggle Text Wrapping

`buildInfo`

Code

```
_satellite.buildInfo
```
Copy
Toggle Text Wrapping

This object contains information about the build of the current tag runtime library. The object contains the following properties:

`turbineVersion`

This provides the Turbine version used inside the current library.

`turbineBuildDate`

The ISO 8601 date when the version of Turbine used inside the container was built.

`buildDate`

The ISO 8601 date when the current library was built.

This example demonstrates the object values:

```
{
  turbineVersion: "14.0.0",
  turbineBuildDate: "2016-07-01T18:10:34Z",
  buildDate: "2016-03-30T16:27:10Z"
}
```

Copy
Toggle Text Wrapping

`environment`

This object contains information about the environment that the current tag runtime library is deployed on.

Code

```
_satellite.environment
```

Copy
Toggle Text Wrapping

The object contains the following properties:

```
{
  id: "ENbe322acb4fc64dfdb603254ffe98b5d3",
  stage: "development"
}
```

Copy
Toggle Text Wrapping

Property	Description
`id`	The id of the environment.
`stage`	The environment for which this library was built. The possible values are `development`, `stagi`

`notify`

NOTE

This method has been deprecated. Please use `_satellite.logger.log()` instead.

Code

```
_satellite.notify(message: string[, level: number])
```

Copy
Toggle Text Wrapping

Example

```
_satellite.notify('Hello world!');
```

Copy
Toggle Text Wrapping

`notify` logs a message to the browser console. The message will only be displayed if tag debugging is enabled by the user (by calling `_satellite.setDebug(true)` or using an appropriate browser extension).

An optional logging level can be passed which will affect the styling and filtering of the message being logged. Supported levels are as follows:

3 - Informational messages.

4 - Warning messages.

5 - Error messages.

If you do not provide a logging level or pass any other level value, the message will be logged as a regular message.

`setCookie`

IMPORTANT

This method has been deprecated. Please use `_satellite.cookie.set()` instead.

Code

```
_satellite.setCookie(name: string, value: string, days: number)
```

Copy
Toggle Text Wrapping

Example

```
_satellite.setCookie('product', 'Circuit Pro', 3);
```

Copy
Toggle Text Wrapping

This sets a cookie in the user's browser. The cookie will persist for the number of days specified.

readCookie

IMPORTANT

This method has been deprecated. Please use `_satellite.cookie.get()` instead.

Code

```
_satellite.readCookie(name: string) => string
```
Copy
Toggle Text Wrapping

Example

```
var product = _satellite.readCookie('product');
```
Copy
Toggle Text Wrapping

This reads a cookie from the user's browser.

removeCookie

NOTE

This method has been deprecated. Please use `_satellite.cookie.remove()` instead.

Code

```
_satellite.removeCookie(name: string)
```
Copy
Toggle Text Wrapping

Example

```
_satellite.removeCookie('product');
```
Copy
Toggle Text Wrapping

This removes a cookie from the user's browser.

Debugging Functions

The following functions should not be accessed from the production code. They are intended only for debugging purposes and will change over time as needed.

`container`

Code

```
_satellite._container
```

Copy

Toggle Text Wrapping

Example

IMPORTANT

This function should not be accessed from the production code. It is intended only for debugging purposes and will change over time as needed.

`monitor`

Code

```
_satellite._monitors
```

Copy

Toggle Text Wrapping

Example

IMPORTANT

This function should not be accessed from the production code. It is intended only for debugging purposes and will change over time as needed.

Sample

On your web page running a tag library, add a snippet of code to your HTML. Typically, the code is inserted into the `<head>` element before the `<script>` element that loads the tag library. This allows the monitor to catch the earliest system events that occur in the tag library. For example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
  <script>
    window._satellite = window._satellite || {};
    window._satellite._monitors = window._satellite._monitors || [];
    window._satellite._monitors.push({
      ruleTriggered: function (event) {
        console.log(
          'rule triggered',
          event.rule
        );
      },
      ruleCompleted: function (event) {
        console.log(
          'rule completed',
          event.rule
        );
      },
      ruleConditionFailed: function (event) {
        console.log(
          'rule condition failed',
          event.rule,
          event.condition
        );
      }
    });
  </script>
  <script src="//assets.adobedtm.com/launch-
EN5bfa516febde4b22b3e7c6f96f6b439f.min.js"
          async></script>
</head>
<body>
  <h1>Click me!</h1>
</body>
</html>
```
Copy
Toggle Text Wrapping

In the first script element, because the tag library has not been loaded yet, the initial `_satellite` object is created and an array on `_satellite._monitors` is initialized. The script then adds a monitor object to that array. The monitor object can specify the following methods which will later be called by the tag library:

`ruleTriggered`

This function is called after an event triggers a rule but before the rule's conditions and actions have been processed. The event object passed to `ruleTriggered` contains information about the rule that was triggered.

`ruleCompleted`

This function is called after a rule has been fully processed. In other words, the event has occurred, all conditions have passed, and all actions have been executed. The event object passed to `ruleCompleted` contains information about the rule that was completed.

`ruleConditionFailed`

This function is called after a rule has been triggered and one of its conditions has failed. The event object passed to `ruleConditionFailed` contains information about the rule that was triggered and the condition that failed.

If `ruleTriggered` is called, either `ruleCompleted` or `ruleConditionFailed` will be called shortly thereafter.

NOTE

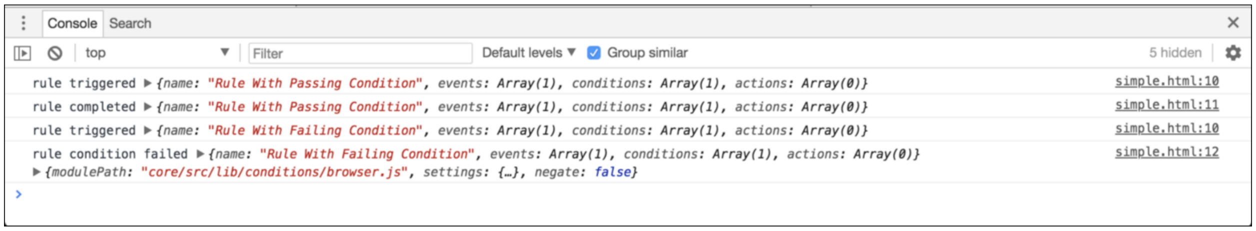
A monitor doesn't have to specify all three methods (`ruleTriggered`, `ruleCompleted`, and `ruleConditionFailed`). Tags in Adobe Experience Platform work with whatever supported methods have been provided by the monitor.

Testing the Monitor

The example above specifies all three methods in the monitor. When they're called, the monitor logs out relevant information. To test this, set up two rules in the tag library:

1. A rule that has a click event and a browser condition that passes only if the browser is Chrome.
2. A rule that has a click event and a browser condition that passes only if the browser is Firefox.

If you open the page in Chrome, open the browser console, and select the page, the following appears in the console:



```
rule triggered ▶{name: "Rule With Passing Condition", events: Array(1), conditions: Array(1), actions: Array(0)}      simple.html:10
rule completed ▶{name: "Rule With Passing Condition", events: Array(1), conditions: Array(1), actions: Array(0)}      simple.html:11
rule triggered ▶{name: "Rule With Failing Condition", events: Array(1), conditions: Array(1), actions: Array(0)}      simple.html:10
rule condition failed ▶{name: "Rule With Failing Condition", events: Array(1), conditions: Array(1), actions: Array(0)}      simple.html:12
▶{modulePath: "core/src/lib/conditions/browser.js", settings: {…}, negate: false}
>
```

Additional hooks or additional information might be added to these handlers as needed.

More help on this feature

- Data elements
- Asynchronous deployment
- Tags overview