# AEM ContextHub

The implementation of a website can require to leverage on some data kept in the browser and used to inject content in a page.
The simplest example is a page from a personal are of a logged user, such as a profile page.
Trying to optimize the pages caching on a web server, it would require to render profile page initially without any user data (so the same renderer html for all users) and then inject user data retrieving it from a client context.

In AEM a client context can be managed with an OOTB feature called *ContextHub*.

More, the ContextHub is one of the options to leverage on for a targeting contents implementation, that is provide a personalized user experience based on some information regarding the user itself.

## What is the ContextHub

The ContextHub is a JavaScript framework provided out-of-the-box by AEM to manage the client context, that is a set of data stored in the browser.

This framework is configurable in the AEM author instance as needed and is pretty extensible.

APIs are provided to access, manipulate and persistence data in the client context.

Data in client context have a structure based on stores.

### Stores

A *store* logically includes:

- a complex object persisted in the client context containing all data related to a specific logical entity (e.g., user profile, e-commerce cart, browser and device used, …);
- APIs to manage that object;
- *storeType*: the store ID;
- initial configuration in json format.

A store is implemented by a clientlib, whose *categories* property contains a value having this convention: *contexthub.store.<storeType>*

E.g., *contexthub.store.granite.profile*, *contexthub.store.contexthub.geolocation*.

A *store candidate* is a clientlib available in AEM to be registered in a *ContextHub configuration* using its *storeType*.
So, a *ContextHub configuration* only contains a subset of all store candidates, only those explicitly associated with it.

A ContextHub configuration can be binded to a whole site or to a specific page.

When a page site is rendered, the ContextHub (and its configuration) is loaded basically including only a JS, that populates all available stores.

Using ContextHub APIs, the website application code can access to a store data and use them.

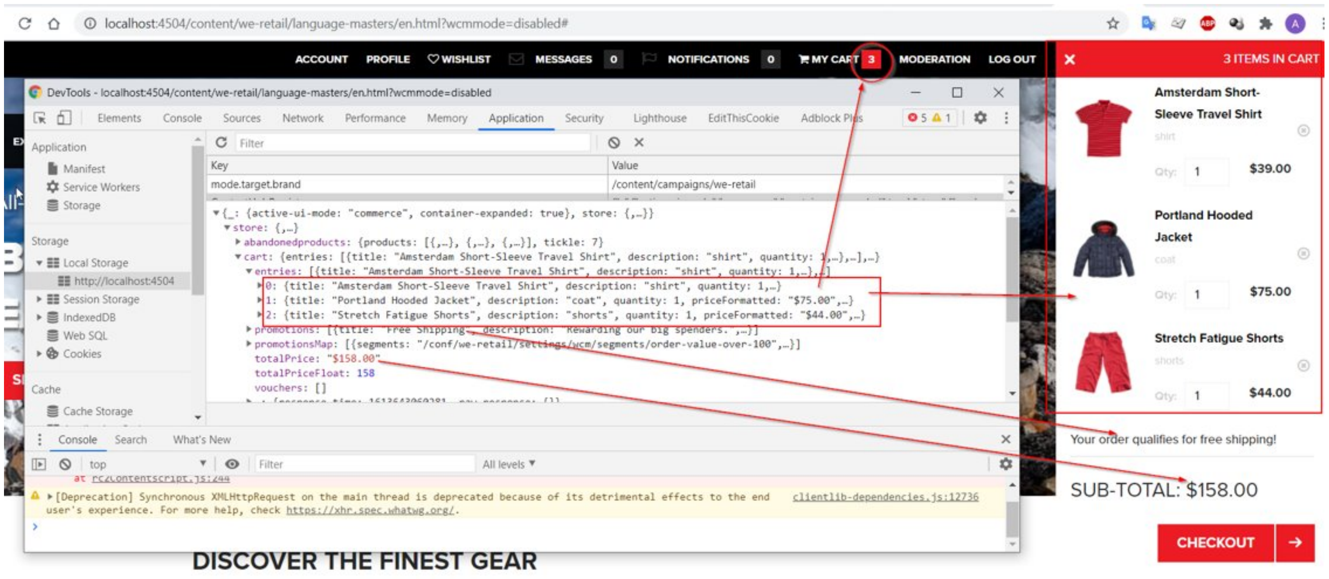
The global object *ContextHub* is available in the DOM.
Given a store name, the store object can be retrieved using *ContextHub.getStore("<store name>")* function.
E.g., var *cartStore = ContextHub.getStore("cart")*

Given a store object, data within the store can be retrieved using *storeObject.getItem("<data path>")* function.
E.g., *cartStore.getItem("entries/0/title")*

Given a store object, data within the store can be manipulate using *storeObject.setItem("<data path>", <data value>)* function.
E.g., *cartStore.setItem("totalPrice", "123€")*

More, each store can define event handler that are triggered when a specific event occurs.
*E.g,ContextHub.eventing.on(ContextHub.Constants.EVENT_STORE_UPDATED + ":cart", this.refreshCart)*

In the example below, the website application code accesses to *cart* store data to provide values to cart items number in the header and details cart composition and subtotal in the right panel:



```
▼{_: {active-ui-mode: "commerce", container-expanded: true}, store: {,…}}
  ▼store: {,…}
    ▶abandonedproducts: {products: [{,…}, {,…}, {,…}], tickle: 7}
    ▼cart: {entries: [{title: "Amsterdam Short-Sleeve Travel Shirt", description: "shirt", quantity: 1,…},…],…}
      ▼entries: [{title: "Amsterdam Short-Sleeve Travel Shirt", description: "shirt", quantity: 1,…},…]
        ▶0: {title: "Amsterdam Short-Sleeve Travel Shirt", description: "shirt", quantity: 1,…}
        ▶1: {title: "Portland Hooded Jacket", description: "coat", quantity: 1, priceFormatted: "$75.00",…}
        ▶2: {title: "Stretch Fatigue Shorts", description: "shorts", quantity: 1, priceFormatted: "$44.00",…}
      ▶promotions: [{title: "Free Shipping", description: "Rewarding our big spenders.",…}]
      ▶promotionsMap: [{segments: "/conf/we-retail/settings/wcm/segments/order-value-over-100",…}]
       totalPrice: "$158.00"
       totalPriceFloat: 158
       vouchers: []
```

```
⚠ ▶[Deprecation] Synchronous XMLHttpRequest on the main thread is deprecated because of its detrimental effects to the end   clientlib-dependencies.js:12736
  user's experience. For more help, check https://xhr.spec.whatwg.org/.
```

When a product is added in or removed from the cart, the cart status is automatically updated and manager by event handlers registered to the *cart* store.

## *Existing store candidates*

Out-of-the-box store candidates are:

- *contexthub.geolocation*: stores the user position coordinates and address using Google maps;
- *contexthub.surferinfo*: stores used browser and device;
- *granite.emulators*: stores data about emulators used on author side in preview mode;
- *granite.profile*: stores AEM user profile information (those contained at */home/users/we-retail/<user>/profile* path);
- *contexthub.datetime*: stores date, time and season calculated using user device system clock and geolocation;

- *contexthub.tagcloud*: stores “tagcloud” provided in the input config or retrieved from meta tags with the page;
- *contexthub.generic-jsonp*: stores data retrieved by a JSONP service;
- *campaign.metadata*: stores metadata related to a campaign;
- *campaign.seeddata*: stores seed related to a campaign;
- *aem.pagedata*: stores page data retrieved calling the current path with *.pagedata.json*;
- *aem.segmentation*: stores only resolved segments. This store is strongly dependent on which other stores are enabled and on the nature of available segments. E.g., the winter/summer segments from *WeRetail* demo website are resolved only if the *geolocation* store is also enabled;
- *aem.analyticsdata*: actually, useless without a proper (unknown) configuration
- *commerce.smartlists*: store user wishlist retrieved calling the current path with *commerce.smartlists.json*;
- *commerce.relatedproducts*: store products related to the current one (probably used only in product pages) calling the current path with *commerce.relatedproducts.json*;
- *commerce.abandonedproducts*: store products which have special change of state (e.g., a visited product that is added to cart);
- *commerce.recentlyviewed*: stores products whose pages have been visited;
- *commerce.cart:* stores all information related to ecommerce cart (items, total, promotions, vouchers) and allowing to manipulate it when user interactions occur (add/remove to cart, promotions or vouchers applied);
- *commerce.orderhistory*: store user previous orders calling the current path with *commerce. orderhistory.json*;

According to the specific implementation of each store, related information can be persisted to:

- localStorage: default option, the object key is *ContextHubPersistence*;
- sessionStorage;
- cookie;
- *window.name*;
- JavaScript objects.

### *Custom store candidates*

If out-of-the-box store candidates don’t provide a solution for a specific requirement, a custom one can be created.

Simply define a new clientlib with a category named *contexthub.store.<store type>*.
Within the js script, define a function that represents the store candidate, inheriting from one of these base stores:

- *ContextHub.Store.PersistedStore*
- *ContextHub.Store.SessionStore*
- *ContextHub.Store.JSONPStore*

- *ContextHub.Store.PersistedJSONPStore*

All of these anyway extends *ContextHub.Store.Core* store.
Eventually add or redefine some functions, manage store configuration extending the default one, add event handlers.

Then register the store using its store type, assigning a priority in case multiple store candidates are registered with the same store type.

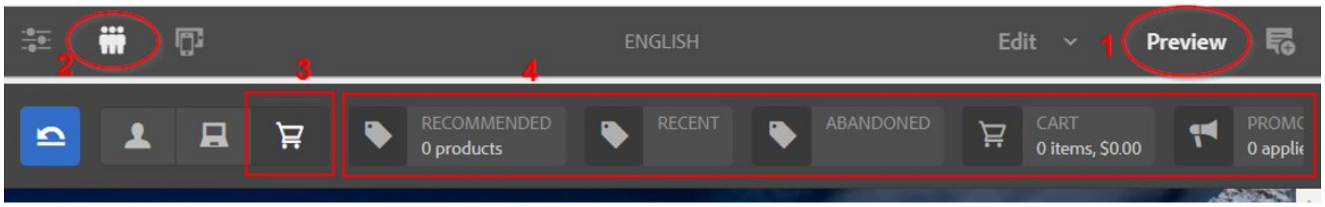
Below is reported a pseudocode for a store candidate definition:

```
(function($, window) {
   var defaultConfig = { ... };

   myStoreCandidate = function(name, config){
      this.config = $.extend(true, {}, defaultConfig, config);
      this.init(name, this.config);
       ...
   };

  ContextHub.Utils.inheritance.inherit(myStoreCandidate,ContextHub.Store.PersistedStore);
  ContextHub.Utils.storeCandidates.registerStoreCandidate(myStoreCandidate, '<store type>', 0);
}(ContextHubJQ, this));
```

## ContextHub toolbar

AEM provides a toolbar for editors to allow data stores display and manipulation.
This is useful to preview a content based on specific client context.
The toolbar is fully customizable and configurable.

The image below shows a toolbar example:



1. Changing view mode in *Preview*;
2. an icon appears in the upper bar; clicking on it the ContextHub toolbar is showed. It includes a set of *UI Modes* and *UI Modules*;
3. each *UI mode* is identified by an icon; clicking on it, the *UI modules* included in the current *UI mode* are showed;
4. Each UI module displays data related to a ContextHub store. Some of them allow also to manipulate the data to test a specific behavior of the website.

The toolbar is totally rendered in JS.

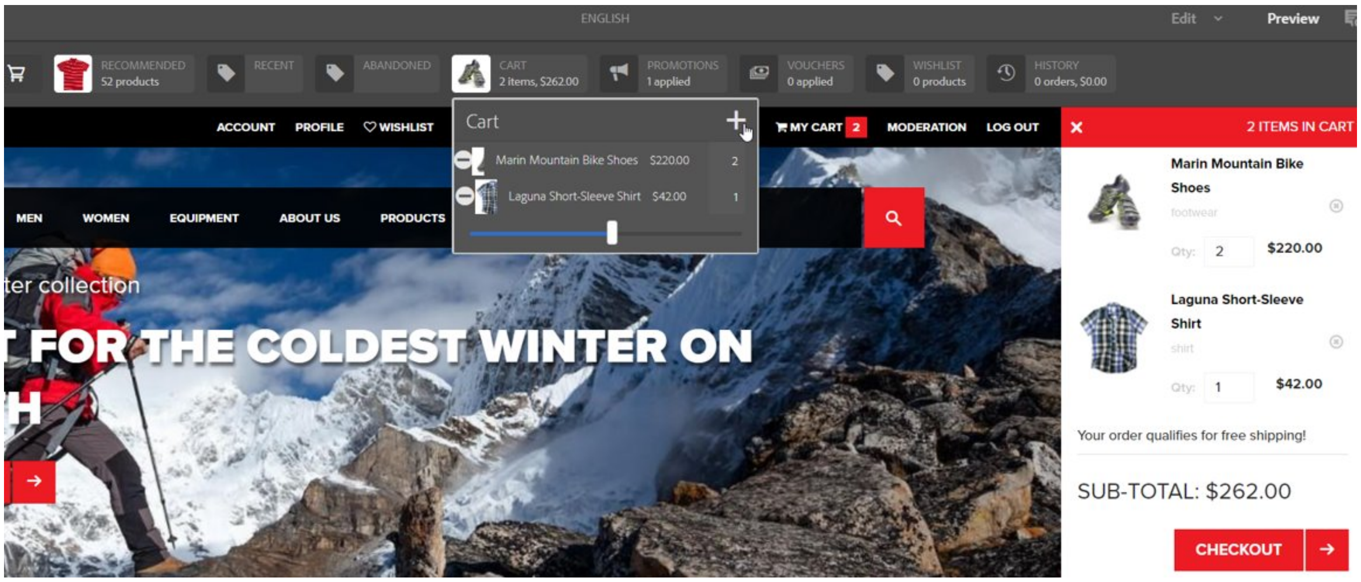
A UI module is implemented by a clientlib, whose *categories* property contains a value having this convention: *contexthub.module.<moduleType>*

E.g., *contexthub.module.granite.profile*, *contexthub.module.contexthub.location*.

Such as stores, only modules registered in the *ContextHub configuration* related to a site page are available in that page.

Modules used ContextHub APIs to access data contained into one or more ContextHub stores to populate the widget in the toolbar and to persist manipulated data into the ContextHub stores again.

In the example below is showed the *cart* UI module. It allows users to add products to cart using a picker, rather than using the specific website UX, and to persist the cart data in the related ContextHub store.
More, the cart in the right panel is updated automatically, because of changing on *cart* ContextHub store triggers some event handlers.



## *Existing UI Modules*
Out-of-the-box *UI Modules* are:

- *commerce.recentlyviewed* and *commerce.abandonedproducts* (same clientlib): displays existing and allows to add products (using a product picker) respectively in the *recentlyviewed* and in *abandonedproducts* ContextHub stores;
- *contexthub.tagcloud*: doesn't work in AEM 6.5 (throws *Uncaught ReferenceError: Class is not defined*);
- *contexthub.device*: should displays the current emulator, but seems not to work well;
- *contexthub.location*: if the browser geolocation is enabled, displays user latitude, longitude, city, country, country code, postal code, region, street number and the google map. It allows also to set a fake geolocation choosing among a predefined list of locations;
- *contexthub.datetime*: doesn't work in AEM 6.5 (throws *Uncaught ReferenceError: Class is not defined*);

- *contexthub.browserinfo*: display browser and operating system version;
- *contexthub.season*: doesn't work in AEM 6.5 (throws *Uncaught ReferenceError: Class is not defined*);
- *campaign.seeddata*: displays campaign seed;
- *commerce.smartlists*: displays existing wishlists and all items contained in each of them;
- *commerce.relatedproducts*: displays recommended products related to the current one (probably used only in product pages);
- *commerce.cart*: displays items currently added to cart and allows to add new one using product picker;
- *commerce.promotions*: displays current applied promotions and allows to add new one using a campaign offer picker;
- *commerce.vouchers*: displays current applied vouchers and allows to add new one using a textfield but seems not to work very well;
- *commerce.orderhistory*: displays user previous orders;
- *granite.resolvedsegments*: displays only available campaign segments for the current page that satisfy all their rules agains the current client context;
- *granite.profile*: displays all user information contained in the profile store and allows to impersonate others users;
- *granite.demographic*: doesn't work in AEM 6.5 (throws *Uncaught ReferenceError: Class is not defined*);
- *contexthub.screen-orientation*: displays device screen orientation (landscape or portrait);
- *contexthub.base*: generic module that needs proper init configuration, can be used to displays specific data retrieved by specific store.

However, all not working UI Modules are neither integrated into We.Retail demo website.

### *Custom UI Modules*

If out-of-the-box UI Modules don't provide a solution for a specific requirement, a custom one can be created.

Simply define a new clientlib with a category named *contexthub.module.<module type>*.
Within the js script, define a function that represents the module renderer, inheriting from *ContextHub.UI.BaseModuleRenderer*.

Eventually add or redefine some functions, manage module configuration extending the default one.

Then register the module using its module type.

Below is reported a pseudocode for a module definition:

```
(function($, window) {
   myModuleRenderer = function(){};

  ContextHub.Utils.inheritance.inherit(myModuleRenderer, ContextHub.UI.BaseModuleRenderer);

        myModuleRenderer.prototype.defaultConfig = { … };
  myModuleRenderer.prototype. onFullscreenClicked = function(module) {
    var config = $.extend({}, this.defaultConfig, module.config);
     …
   };
  myModuleRenderer.prototype.onListItemClicked = function(module, position, data) {…};
  myModuleRenderer.prototype. getPopoverContent = function(module, popoverVariant) {…};

  ContextHub.UI.ModuleRenderer('<module type>', new myModuleRenderer ());
}(ContextHubJQ, this));
```

## How to setup the ContextHub

There are some steps to follow if you want to add the ContextHub feature to your project.
First of all, we can distinguish between developments and configurations.

### Developments

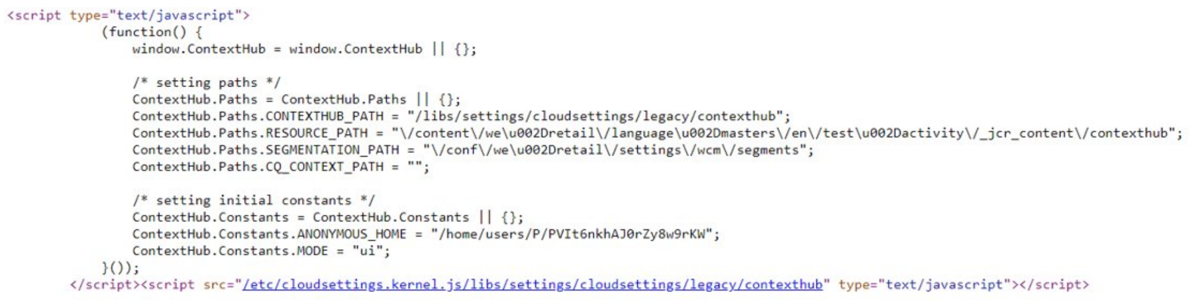
According to your project requirements, could be needed to implement custom store candidates and UI Modules.
This implies to create specific clientlibs whose categories must have a proper naming convention with a fixed prefix and a different suffix for each store and module (see related sections above).
You can include the clientlibs in your project such as any other clientlib.

Then you must include js framework in you rendering scripts so that the inclusion occurs in html *head* tag. In HTL scripts, the following can be used:

*<sly data-sly-resource="${'contexthub' @ resourceType='granite/contexthub/components/contexthub'}"/>*

If you have to enable the ContextHub in the whole site and for example you're inheriting page component from *Core component* one, a good point of inclusion is the *customheaderlibs.html*.

A rendered output sample is showed in the image below



```
<script type="text/javascript">
            (function() {
                window.ContextHub = window.ContextHub || {};

                /* setting paths */
                ContextHub.Paths = ContextHub.Paths || {};
                ContextHub.Paths.CONTEXTHUB_PATH = "/libs/settings/cloudsettings/legacy/contexthub";
                ContextHub.Paths.RESOURCE_PATH = "\/content\/we\u002Dretail\/language\u002Dmasters\/en\/test\u002Dactivity\/_jcr_content\/contexthub";
                ContextHub.Paths.SEGMENTATION_PATH = "\/conf\/we\u002Dretail\/settings\/wcm\/segments";
                ContextHub.Paths.CQ_CONTEXT_PATH = "";

                /* setting initial constants */
                ContextHub.Constants = ContextHub.Constants || {};
                ContextHub.Constants.ANONYMOUS_HOME = "/home/users/P/PVIt6nkhAJ0rZy8w9rKW";
                ContextHub.Constants.MODE = "ui";
            }());
        </script><script src="/etc/cloudsettings.kernel.js/libs/settings/cloudsettings/legacy/contexthub" type="text/javascript"></script>
```

Basically, it contains only some configurations and the contextHub framework inclusion.

No other development steps are required for a basic integration.

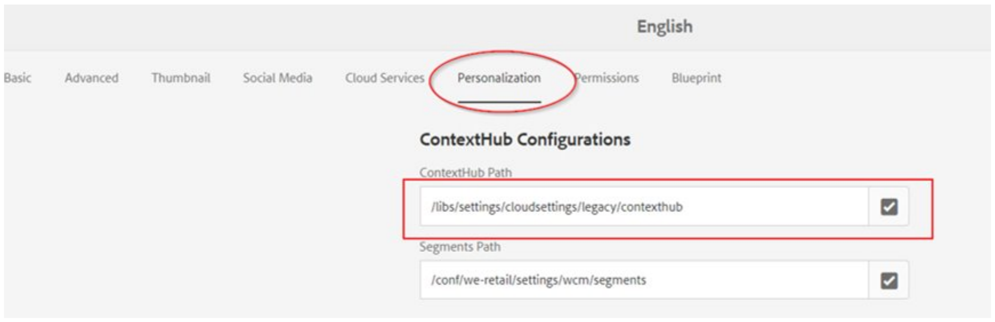
## Configurations

Next step is to create a *ContextHub Configuration*, including stores, UI Modes, UI Modules.

ContextHub configurations are located in the repository at:

- */libs/settings/cloudsettings/legacy*: OOTB configuration ready to use, including the most OOTB stores and UI Modules (grouped into 3 UI Modes: *persona*, *device*, *commerce*);
- */conf/global/settings/cloudsettings*: location used to share configuration among different projects;
- */conf/<tenant>/settings/cloudsettings*: configuration related to a tenant.

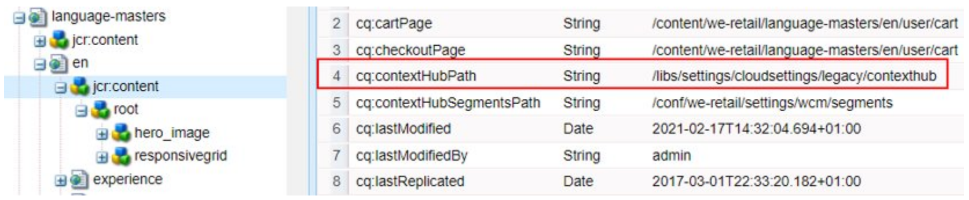
Anyway, the configuration defined in a tenant conf can be used by any page of any website (with proper ACL read privileges settings).
This is because the binding between the ContextHub configuration and the page is made using page properties. Below images show how the binding works:



ContextHub Configurations

ContextHub Path: /libs/settings/cloudsettings/legacy/contexthub

Segments Path: /conf/we-retail/settings/wcm/segments

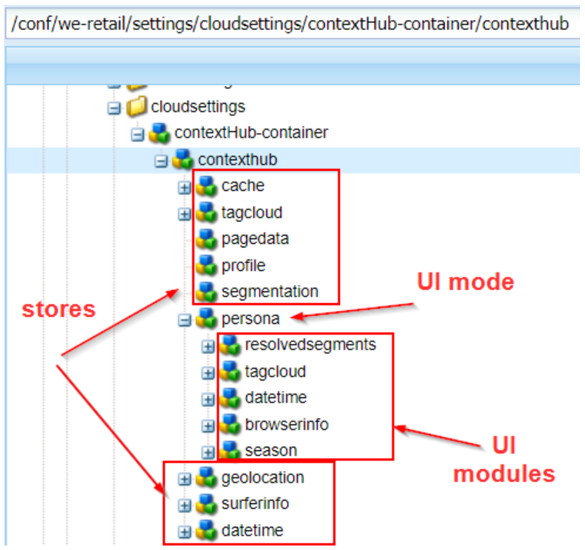


| | | | |
|---|---|---|---|
| 2 | cq:cartPage | String | /content/we-retail/language-masters/en/user/cart |
| 3 | cq:checkoutPage | String | /content/we-retail/language-masters/en/user/cart |
| 4 | cq:contextHubPath | String | /libs/settings/cloudsettings/legacy/contexthub |
| 5 | cq:contextHubSegmentsPath | String | /conf/we-retail/settings/wcm/segments |
| 6 | cq:lastModified | Date | 2021-02-17T14:32:04.694+01:00 |
| 7 | cq:lastModifiedBy | String | admin |
| 8 | cq:lastReplicated | Date | 2017-03-01T22:33:20.182+01:00 |

More, the binding it automatically inherited from upper-level pages if not specifically set in the current page, so it’s enough to set it in the homepage to be valid for the whole website.

The contextHub configuration has the structure showed in the image below:

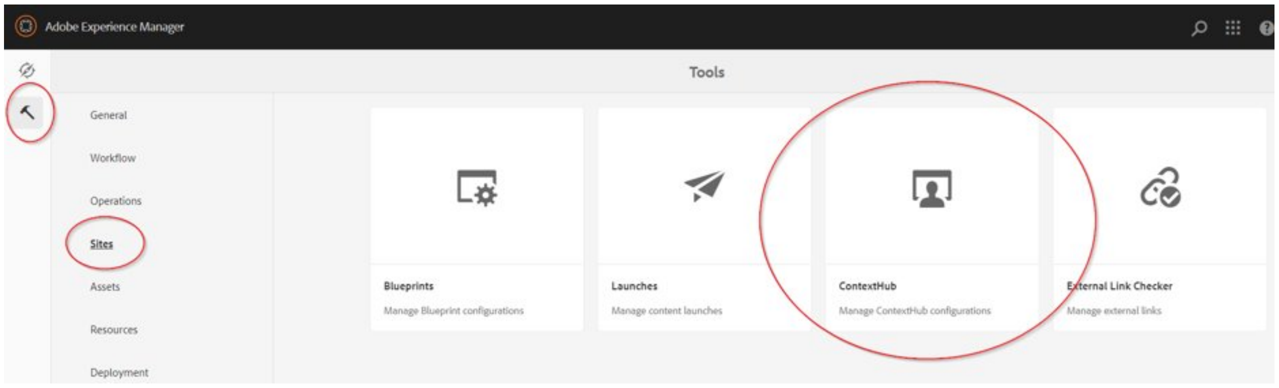
1. First level: stores or UI modes;
2. Second level: UI modules (only with UI modes as parent)



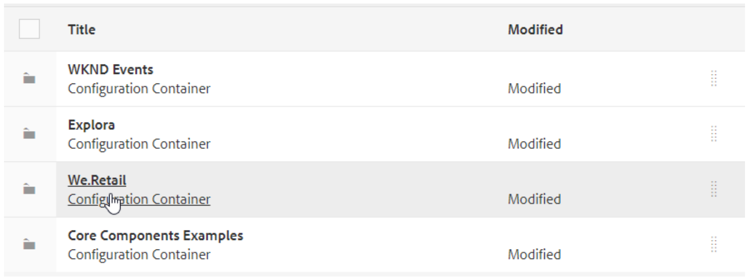
Configuration under a specific tenant can be made editorially in the author instance.

## *Configuration using Touch-UI*

For Touch-UI navigation menu, go to Tools -> Sites -> ContextHub, such as showed below:

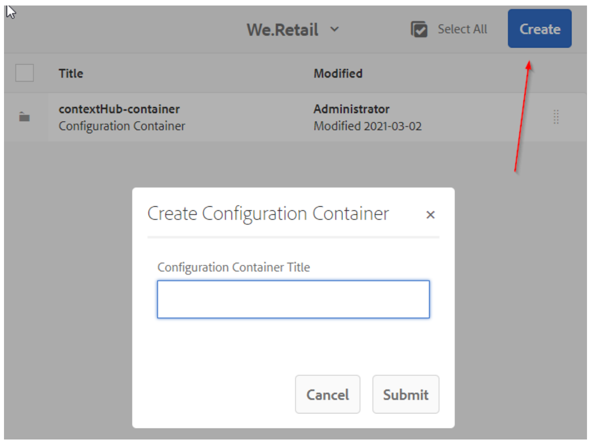


Click on a tenant (if you want to create a new tenant go to Tools -> General -> Browser configuration)

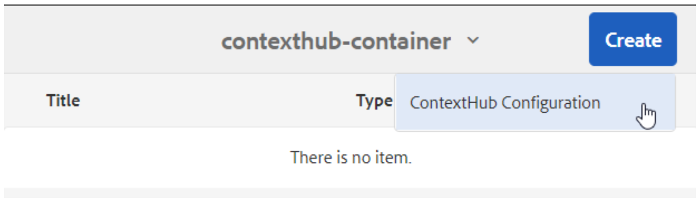


| Title | Modified |
|---|---|
| WKND Events<br>Configuration Container | Modified |
| Explora<br>Configuration Container | Modified |
| We.Retail<br>Configuration Container | Modified |
| Core Components Examples<br>Configuration Container | Modified |

Create a configuration container, <u>using only lowercase letters, numbers and minus characters otherwise net step could not work</u>:



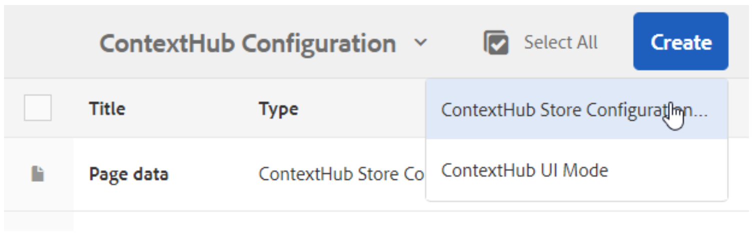
Then click on the configuration container just created and create a *ContextHub configuration*.



Possible settings are:

- *Debug*: if checked the contextHub js framework will not be minified;
- *Disable ContextHub*: if checked this configuration will not be avaiable to be binded in the page properties.

Clicking on the contextHub configuration just created is possible to create ContextHub Store configurations and UI Modes.

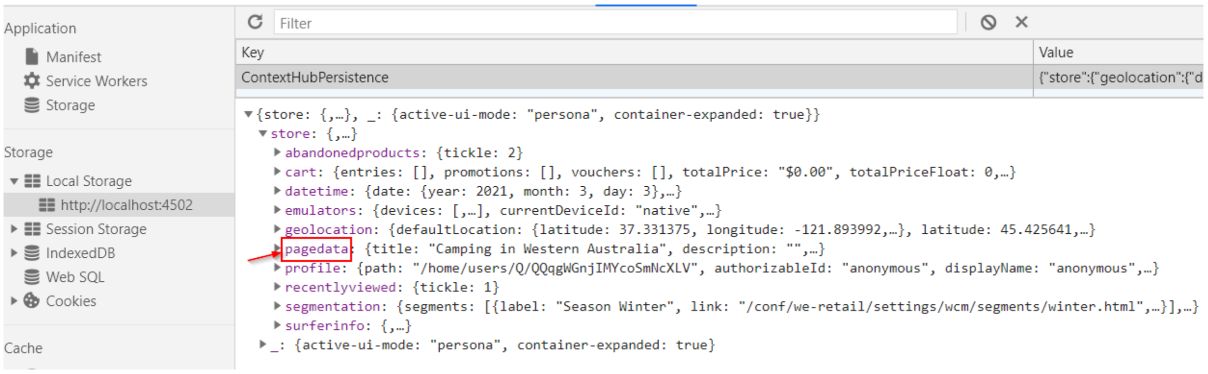


## ContextHub Store configuration

The wizard for ContextHub Store configuration creation contains:

- *Title*: simple label

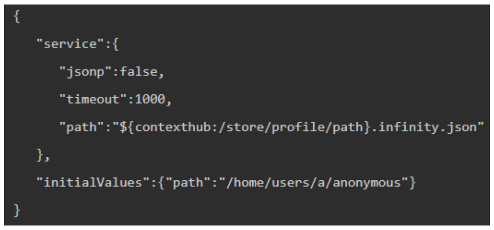
- *Name*: used as store key in the persistence object. E.g., the *pagedata* key in the image below



| Key | Value |
|---|---|
| ContextHubPersistence | {"store":{"geolocation":{"d |

```
▼{store: {,…}, _: {active-ui-mode: "persona", container-expanded: true}}
  ▼store: {,…}
    ▶abandonedproducts: {tickle: 2}
    ▶cart: {entries: [], promotions: [], vouchers: [], totalPrice: "$0.00", totalPriceFloat: 0,…}
    ▶datetime: {date: {year: 2021, month: 3, day: 3},…}
    ▶emulators: {devices: [,…], currentDeviceId: "native",…}
    ▶geolocation: {defaultLocation: {latitude: 37.331375, longitude: -121.893992,…}, latitude: 45.425641,…}
    ▶pagedata: {title: "Camping in Western Australia", description: "",…}
    ▶profile: {path: "/home/users/Q/QQqgWGnjIMYcoSmNcXLV", authorizableId: "anonymous", displayName: "anonymous",…}
    ▶recentlyviewed: {tickle: 1}
    ▶segmentation: {segments: [{label: "Season Winter", link: "/conf/we-retail/settings/wcm/segments/winter.html",…}],…}
    ▶surferinfo: {,…}
  ▶_: {active-ui-mode: "persona", container-expanded: true}
```

- *Store type*: used in for matching with clientlibs *categories* property values (see naming conventions for stores clientlibs). The store is registered using this value in the ContextHub store candidates register;
- *Required*: if checked and no matching clientlib are found for the defined store type field, an error is printed in the browser console but the ContextHub works anyway (excluding this store);
- *Enabled*: if checked, this store configuration is considered in the ContextHub configuration javascript code creation;
- *Detail Configuration (JSON)*: a json configuration used to initialize the store object. Each store candidate has a specific configuration structure due to his nature. E.g., the *granite.profile* store candidate has a configuration such as:



```
{
   "service":{
      "jsonp":false,
      "timeout":1000,
      "path":"${contexthub:/store/profile/path}.infinity.json"
   },
   "initialValues":{"path":"/home/users/a/anonymous"}
}
```

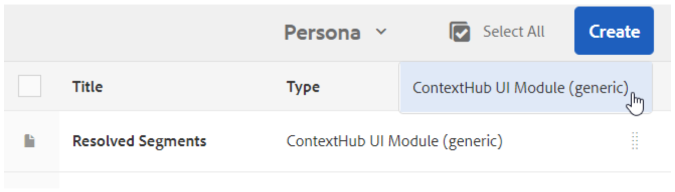
## UI Mode configuration

The wizard for UI Mode configuration creation contains:

- *Title*: simple label;
- *Name*: name of the node (unused);
- Mode Icon: coral css class identifiying a coral icon to show in the toolbar. Must have pattern *coral-Icon--<name>* (e.g., *coral-Icon--user*);
- *Enabled*: if checked, the icon is showed in the ContextHub toolbar.

## UI Modules configuration

Clicking on a UI mode configuration is possible to create a UI Modules configuration



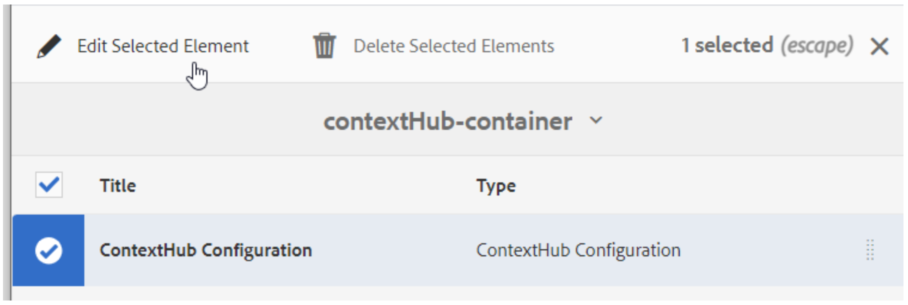
| Title | Type |
|---|---|
| Resolved Segments | ContextHub UI Module (generic) |

The wizard for UI Module configuration creation contains:

- *Title*: simple label;
- *Name*: node of the node (unused);
- *Module type*: used in for matching with clientlibs *categories* property values (see naming conventions for UI modules clientlibs). The module is registered using this value in the ContextHub modules register;
- *Enabled*: if checked, the widget related to this module is showed in the ContextHub toolbar (when the containing UI Mode is enabled).

## *ContextHub configuration*

Once some stores, UI Modes and UI Modules have been defined, it's possible to check how these settings are used to create two JSON configurations used by ContextHub JavaScript framework. These JSON are visible in the ContextHub Configuration properties accessible clicking on *Edit Selected Element* button:

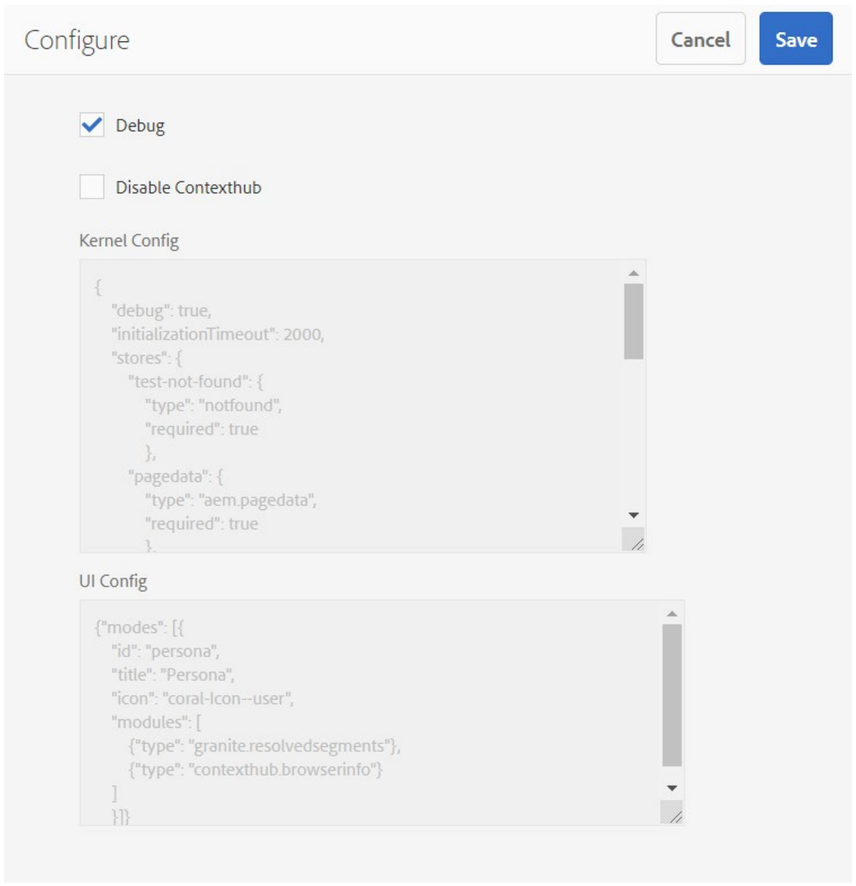


| Title | Type |
|---|---|
| ContextHub Configuration | ContextHub Configuration |

Configurations are:

- *Kernel Config*: contains stores configurations;
- *UI Config*: contains UI Modes and UI Modules configurations.

They are not modifiable because are autogenerated.



[✓ Debug]

Disable Contexthub

Kernel Config

```
{
   "debug": true,
   "initializationTimeout": 2000,
   "stores": {
      "test-not-found": {
         "type": "notfound",
         "required": true
         },
      "pagedata": {
         "type": "aem.pagedata",
         "required": true
```

UI Config

```
{"modes": [{
   "id": "persona",
   "title": "Persona",
   "icon": "coral-Icon--user",
   "modules": [
      {"type": "granite.resolvedsegments"},
      {"type": "contexthub.browserinfo"}
   ]
   }]}
```

## Targeted Content

ContextHub can be used as engine to provide a user personalized experience, such as show different contents within a component.

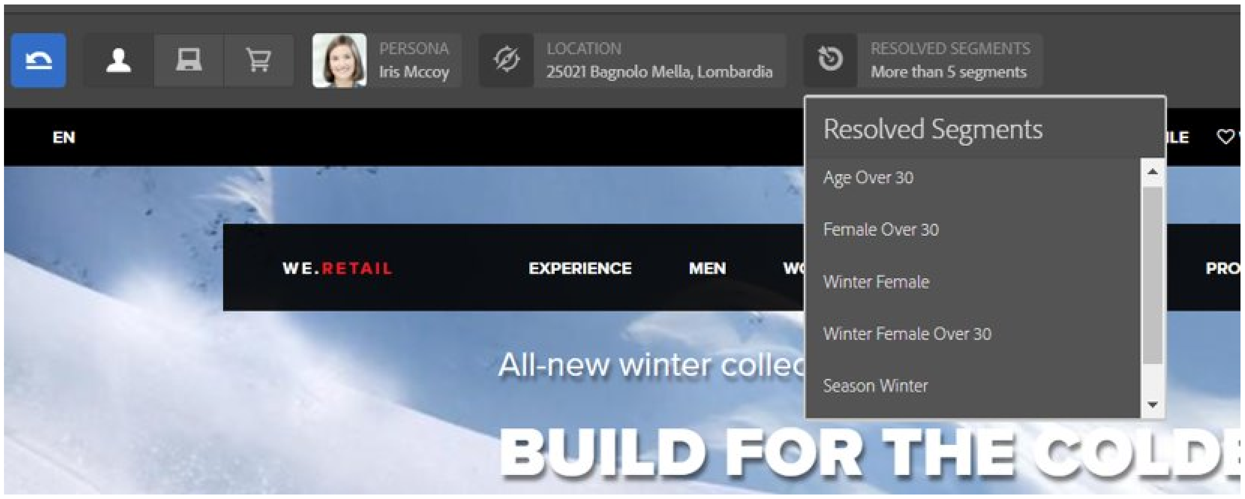
Users are divided into categories called **segments**.
A user belongs to a segment if its client context satisfies all **traits** included in the segment definition.
Criteria are can be collected in several ways: user personal information and preference, user history, navigation, a survey, ...
A user can belong to zero, one or more segments, called **resolved segments**.
The process regarding creation of segments is called **segmentation**.

An **activity** is a marketing initiative regarding a subset of segments.
A component to be targeted must be associated to an activity.
Then a specific content can be binded to a specific segment.

When a user belonging to a segment navigates to a page with a targeted component, the content associated to the user segment is showed.

It's required to add segmentation ContextHub store candidate to the ContextHub configuration in order to make content targeting works.

It's also useful to add segmentation UI Module to provide the ContextHub toolbar a specific widget that shows resolved segments, as showed below



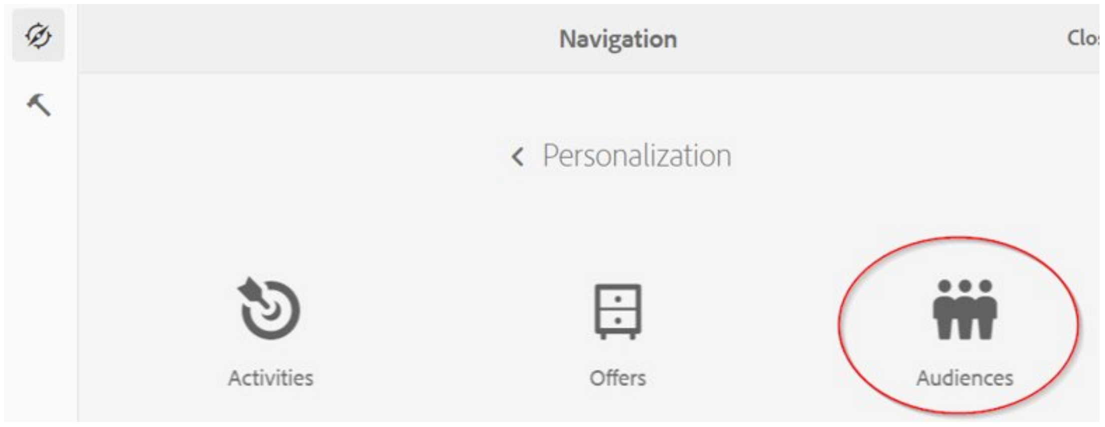
## Use case

An ecommerce website has in its catalogue the following products taxonomy: male, female, unisex.
The website users are segmented in male, female and anonymous.
The homepage hero slider can contain references to several products, usually most wanted products (independently from the taxonomy).
Marketing people wants to create an activity associated with the hero slider and involving male and female segments so that:

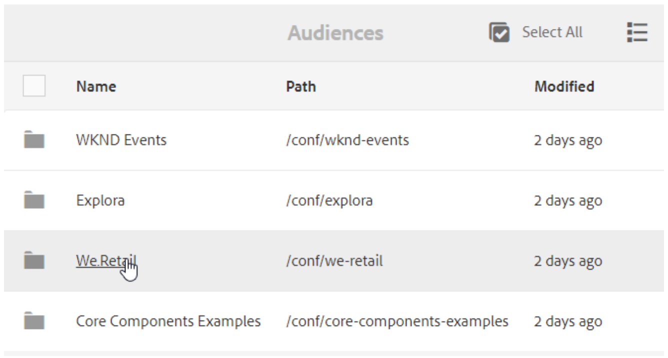
- for male segment the hero slider only contains references to male and unisex products;
- for female segment the hero slider only contains references to female and unisex products;
- otherwise default content is showed.

## Audiences

Segments are located at path */conf/<tenant>/settings/wcm/segments* and they can be configured editorially from Navigation -> Personalization -> Audiences:

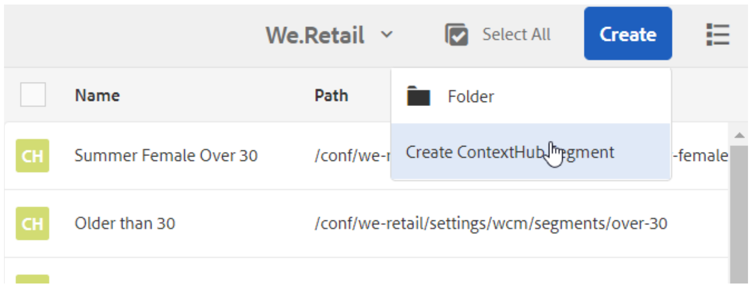


Clicking on a tenant



| Name | Path | Modified |
|---|---|---|
| WKND Events | /conf/wknd-events | 2 days ago |
| Explora | /conf/explora | 2 days ago |
| We.Retail | /conf/we-retail | 2 days ago |
| Core Components Examples | /conf/core-components-examples | 2 days ago |

is then possible to create a *ContextHub Segment*



| Name | Path |
|---|---|
| Summer Female Over 30 | /conf/we-r -female |
| Older than 30 | /conf/we-retail/settings/wcm/segments/over-30 |

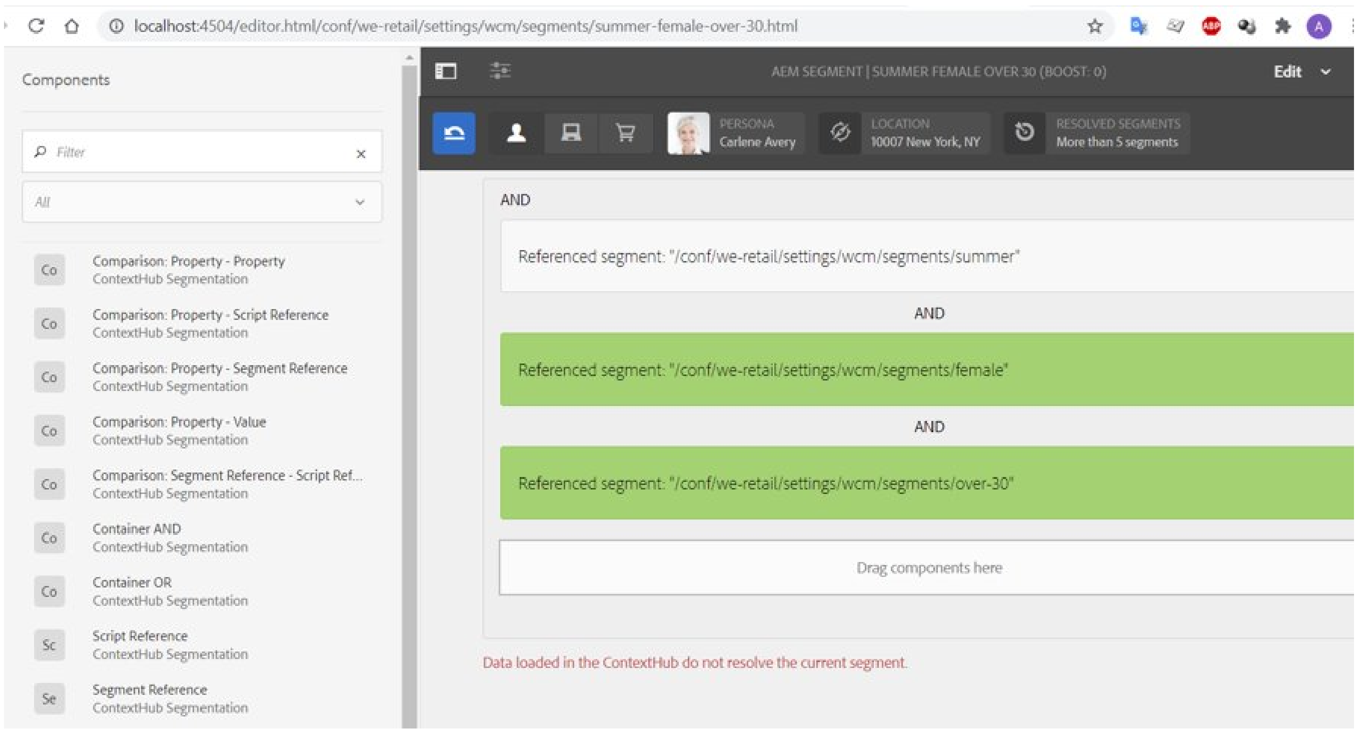
The wizard has only two fields:

- *Title*: simple label;
- *Boost*: numeric value to establish a ranking between segments so that a user that resolves more segments is associated with the segment with higher boost value.

Accessing to segment just created is possible to define a complex rule combining constraint with OR and AND operators.
A constraint can be:

- Comparison between a store value and a primitive value;
- Comparison between a store value and another store value;
- A script reference;
- A segment reference, that is including a rule defined in another segment;
- Comparison between a store value and a script reference;
- Comparison between a store value and a segment reference;
- Comparison between a script reference and a segment reference.

Custom constraints can be developed as well.



localhost:4504/editor.html/conf/we-retail/settings/wcm/segments/summer-female-over-30.html

Components

AEM SEGMENT | SUMMER FEMALE OVER 30 (BOOST: 0)

PERSONA Carlene Avery · LOCATION 10007 New York, NY · RESOLVED SEGMENTS More than 5 segments

| Component | Group |
|---|---|
| Comparison: Property - Property | ContextHub Segmentation |
| Comparison: Property - Script Reference | ContextHub Segmentation |
| Comparison: Property - Segment Reference | ContextHub Segmentation |
| Comparison: Property - Value | ContextHub Segmentation |
| Comparison: Segment Reference - Script Ref... | ContextHub Segmentation |
| Container AND | ContextHub Segmentation |
| Container OR | ContextHub Segmentation |
| Script Reference | ContextHub Segmentation |
| Segment Reference | ContextHub Segmentation |

AND

Referenced segment: "/conf/we-retail/settings/wcm/segments/summer"

AND

Referenced segment: "/conf/we-retail/settings/wcm/segments/female"

AND

Referenced segment: "/conf/we-retail/settings/wcm/segments/over-30"

Drag components here

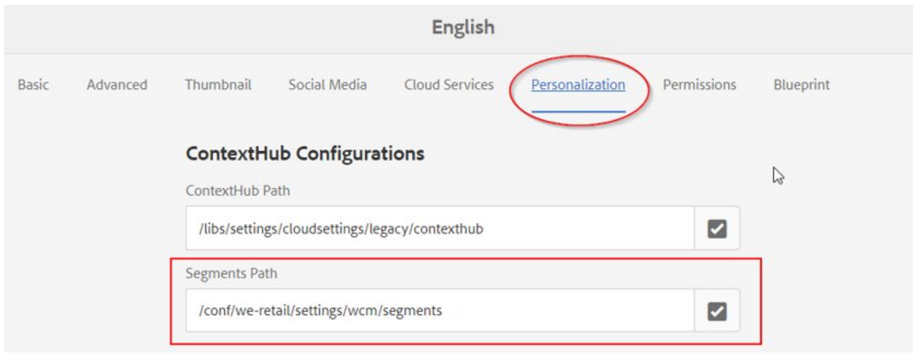
Data loaded in the ContextHub do not resolve the current segment.

Image above shows segment Summer-Female-over-30 defined as combination via AND of following referenced segments:

- Summer: uses geolocation store and season store;
- Female: uses gender from profile store;
- Over-30: uses age from profile store

The green rules are those satisfied by current client context, so the current segment is resolved by current client context only if all rules are green.

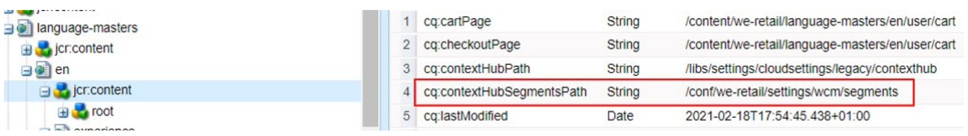
Finally, segments defined in a tenant must be binded to a site or single page (such as already showed with ContextHub Configuration) in the following way:



ContextHub Configurations

ContextHub Path: /libs/settings/cloudsettings/legacy/contexthub ✓

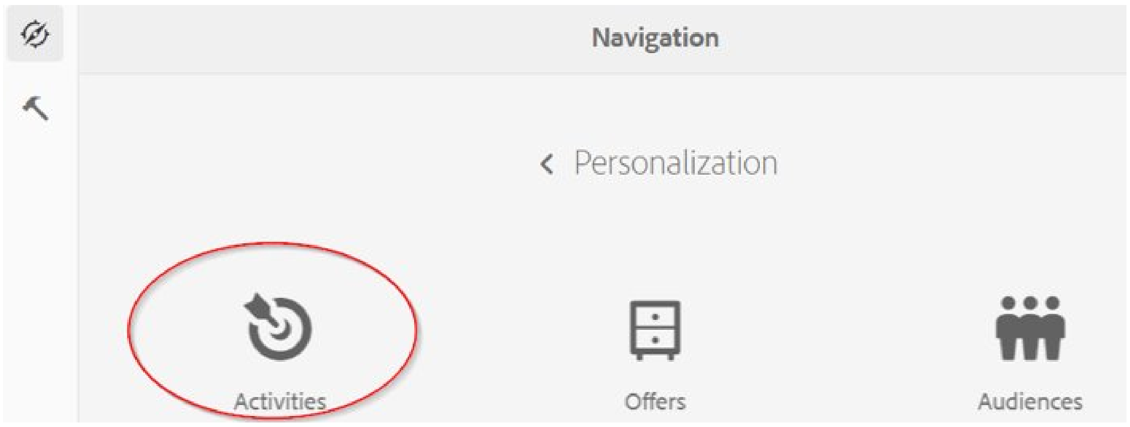
Segments Path: /conf/we-retail/settings/wcm/segments ✓



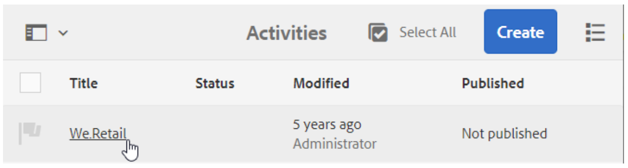
| | | |
|---|---|---|
| cq:cartPage | String | /content/we-retail/language-masters/en/user/cart |
| cq:checkoutPage | String | /content/we-retail/language-masters/en/user/cart |
| cq:contextHubPath | String | /libs/settings/cloudsettings/legacy/contexthub |
| cq:contextHubSegmentsPath | String | /conf/we-retail/settings/wcm/segments |
| cq:lastModified | Date | 2021-02-18T17:54:45.438+01:00 |

## Activities

Activities are located at path */content/campaigns/<tenant>/master* and they can be configured editorially from Navigation -> Personalization -> Activities:

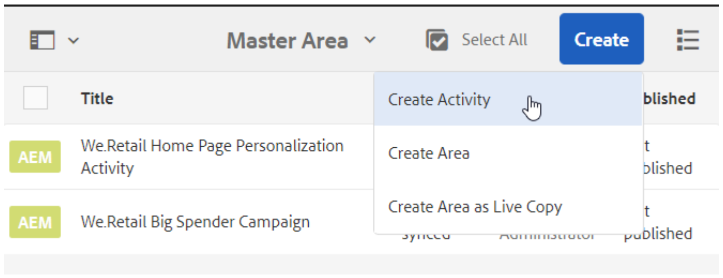


Click on a brand



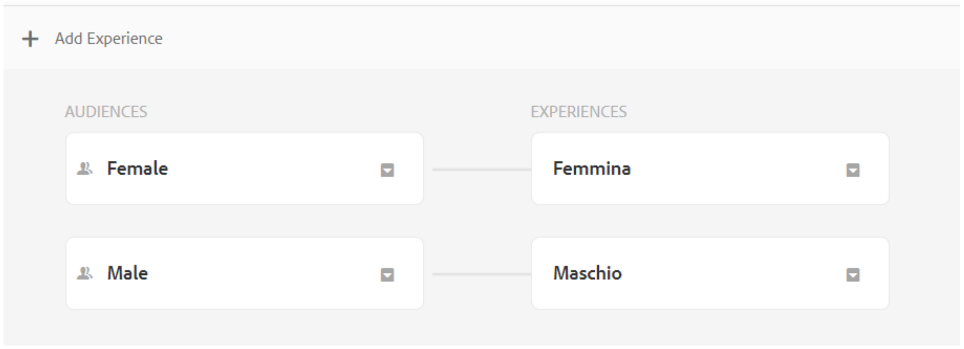
| Title | Status | Modified | Published |
|---|---|---|---|
| We.Retail | | 5 years ago Administrator | Not published |

Then is possible to create an activity



The wizard for activity creation contains:

- *Title*: simple label;
- *Name*: node name (unused);
- *Targeting Engine*: choose ContextHub (AEM);
- *Objective*: description for the activity;
- Adding one or more *Experience*, that is kind of renaming of segments defined in Audiencies section.

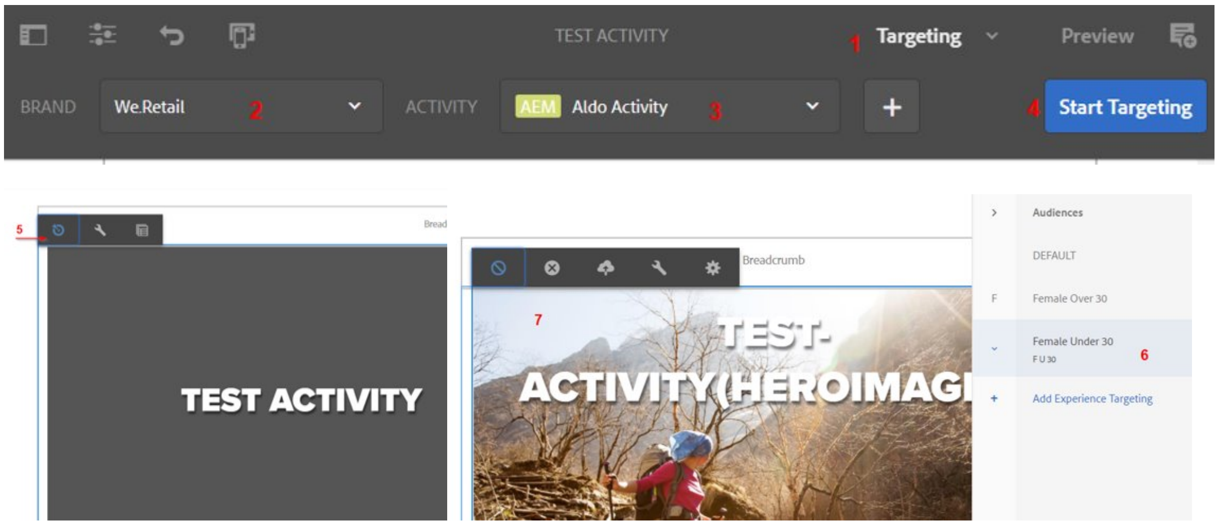


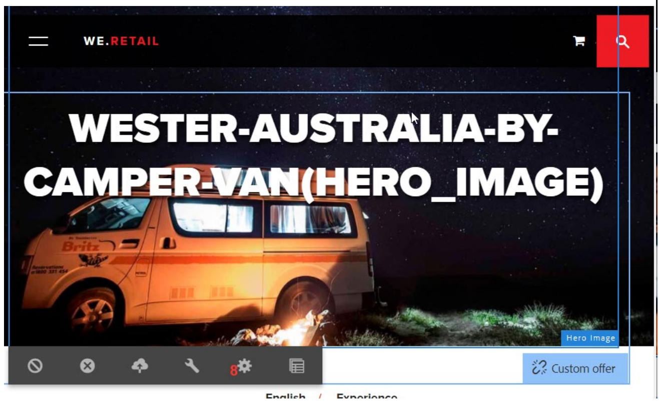
Be aware that when you add a new Experience, the showed list contains segments from all tenant and legacy configurations;

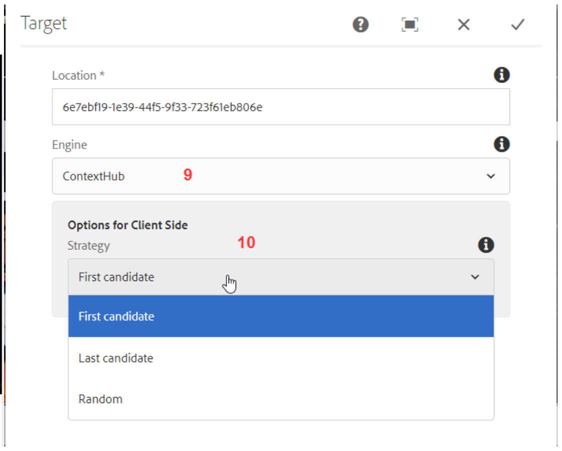
- *Duration – start*: specifies since when the activity is valid.
    - When activated: on publish, as far as the targeted content page is activated;
    - Specified Date and Time: starting from a timestamp chosen with a date picker.
- *Duration – end*: specifies to when the activity is valid.
    - When deactivated: on publish, as far as the targeted content page is deactivated;
    - Specified Date and Time: until a timestamp chosen with a date picker.
- *Priority*: if more activities are applied to the same components, only the activity with higher priority is considered.

## Targeting configuration

Once an activity with some segments is created, a component can be targeted.







Location *
6e7ebf19-1e39-44f5-9f33-723f61eb806e

Engine
ContextHub

Options for Client Side
Strategy
First candidate

First candidate
Last candidate
Random

1. Select *Targeting* view mode within the page editor;
2. Select the *Brand*;
3. Select the *Activity*;
4. Click on *Start Targeting* button;
5. Choose the component to be targeted and click on *Target* button;
6. Chose a segment in the *Audiences* right panel;
7. Once clicked on a segment, the component is ready to be personalized for the selected segment. Insert new contents in the component (e.g., if the component is an hero image, set another image);
8. Click on *Target component setting* button;
9. Select *ContextHub* in the *Engine* field;
10. Select strategy to apply when multiple segments are resolved:
    - First candidate: upper experience in the list defined in the activity;
    - Last candidate: lower experience in the list defined in the activity;
    - Random

Repeat steps 6 and 7 for each segment in the activity.

Targeted contents are located at path
*/content/campaigns/<tenant>/master/<activity>/<segment>/<component>*

## Targeting rendering

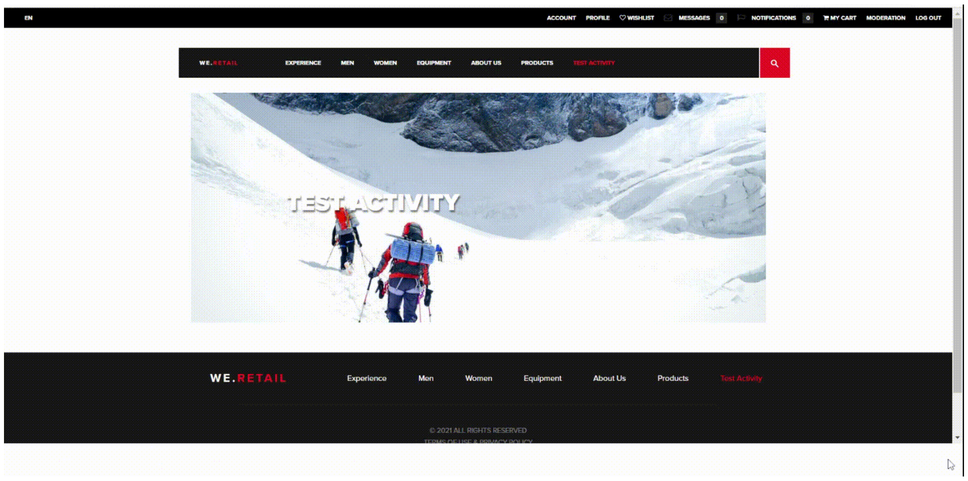
The following are the steps for a targeting rendering:

1. When a page containing a targeted component is rendered, the targeted component is initially hided with *visibility:hidden* css rule, that is a blank block is showed, although the default content is loaded in the source code;
2. The Segment Engine retrieves all resolved segments but only the one with higher boost is considered;
3. For a targeted component with multiple activities, only the one with higher priority is considered;

4. If there is a matching between the considered user segment and a segment related to the considered activity, the content related to that activity segment is retrieved with an ajax call and replaces the default one;
5. The component is showed,

A side effect occurs during this process that is called *flickering*, caused because the targeted content is injected client side after the server side rendering.

The image below shows this effect



## ContextHub js framework anatomy

The JavaScript framework is actually are dynamically generated scripts, because it includes *Kernel Config* and *UI Config* (see *ContextHub Configuration* section) that are generated automatically based on editorial settings.

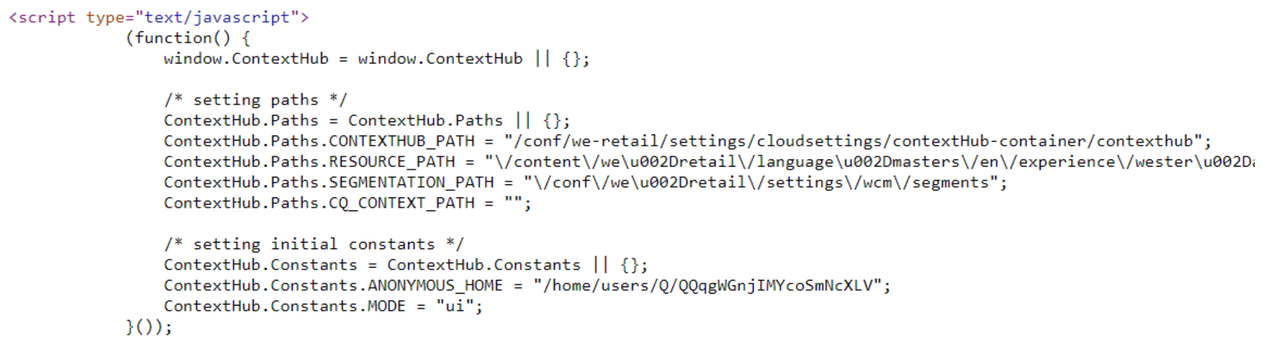
The bundle *Adobe Granite ContextHub Commons (com.adobe.granite.contexthub.commons)* manages the scripts creations.

Actually 2 scripts are included

- One to manage ContextHub stores;
- On to manage ContextHub toolbar (only with wcmmode.Disabled).

### ContextHub store js (author and publish)

First of all, a script such as the one below is included as inline js:



```
<script type="text/javascript">
            (function() {
                window.ContextHub = window.ContextHub || {};

                /* setting paths */
                ContextHub.Paths = ContextHub.Paths || {};
                ContextHub.Paths.CONTEXTHUB_PATH = "/conf/we-retail/settings/cloudsettings/contextHub-container/contexthub";
                ContextHub.Paths.RESOURCE_PATH = "\/content\/we\u002Dretail\/language\u002Dmasters\/en\/experience\/wester\u002D
                ContextHub.Paths.SEGMENTATION_PATH = "\/conf\/we\u002Dretail\/settings\/wcm\/segments";
                ContextHub.Paths.CQ_CONTEXT_PATH = "";

                /* setting initial constants */
                ContextHub.Constants = ContextHub.Constants || {};
                ContextHub.Constants.ANONYMOUS_HOME = "/home/users/Q/QQqgWGnjIMYcoSmNcXLV";
                ContextHub.Constants.MODE = "ui";
            }());
```

It includes configuration settings defined in the page properties (such as ContextHub configuration path and audiences configuration path).

Then a js script is included, having URL with pattern */etc/cloudsettings.kernel.html/<contextHub configuration path>*
E.g., */etc/cloudsettings.kernel.htm /conf/we-retail/settings/cloudsettings/contextHub-container/contexthub*

The JS is structured in this way:

- *Kernel Config* containing stores configurations, retrievable at *<contextHub configuration path>.config.kernel.js* (e.g., */conf/we-retail/settings/cloudsettings/contextHub-container/contexthub.config.kernel.js*)
- *contexthub.utils* clientlib (*/libs/granite/contexthub/code/kernel/utils*) and its dependencies:
    - *contexthub.jquery* (*/libs/granite/contexthub/dependencies/jquery*)
    - *contexthub.shared* (*/libs/granite/contexthub/code/kernel/shared*) and its dependencies
        - *contexthub.polyfills* (*/libs/granite/contexthub/code/kernel/polyfills*)
        - *contexthub.constants* (*/libs/granite/contexthub/code/kernel/constants*)
- the core *contexthub.kernel* clientlib (*/libs/granite/contexthub/code/kernel/core/initialization*) and its dependency:
    - *contexthub.config.override* (*/libs/granite/contexthub/code/kernel/overrides/kernel-config*)
- *contexthub.generic-stores* clientlib (*/libs/granite/contexthub/code/kernel/generic-stores*): contains base stores to be extended (see *Custom store candidates* section);
- *contexthub.segment-engine* clientlib (*/libs/cq/contexthub/code/kernel/segment-engine*)
- *contexthub.segment-engine.operators* clientlib  (*/libs/cq/contexthub/code/kernel/segment-engine/comparison-operators*): contains OOTB segment operators constraints definition
- *contexthub.segment-engine.operators.custom* clientlib: placeholder for custom segment operators constraints definition inclusion;
- *contexthub.segment-engine.scripts* clientlib (*/libs/cq/contexthub/code/kernel/segment-engine/comparison-scripts*): contains OOTB segment script constraints definition;
- *contexthub.segment-engine.scripts.custom* clientlib: placeholder for custom segment script constraints definition;
- *contexthub.segment-engine.page-interaction* clientlib (*/libs/cq/contexthub/code/kernel/segment-engine/page-interaction*)

- For each enabled store, inclusion of clientlib containing the implementation (e.g., for *granite.profile* store the clientlib at */libs/granite/contexthub/components/stores/profile*)

## ContextHub UI Modes and UI Modules js (only author)

Again, an inline script is included at first, containing some application code to inject the ContextHub toolbar.

Then a js script is included, having URL with pattern */etc/cloudsettings.ui.html/<contextHub configuration path>*
E.g., */etc/cloudsettings.ui.html/conf/we-retail/settings/cloudsettings/contextHub-container/contexthub*

The JS is structured in this way:

- *UI Config* containing stores configurations, retrievable at *<contextHub configuration path>.config.ui.js* (e.g., */conf/we-retail/settings/cloudsettings/contextHub-container/contexthub.config.ui.js*)
- *the core Contexthub.ui* clientlib (*/libs/granite/contexthub/code/ui/container*) and its dependencies:
    - *contexthub.jquery* (*/libs/granite/contexthub/dependencies/jquery*)
    - *contexthub.handlebars4* (*/libs/granite/contexthub/dependencies/handlebars*)
- For each enabled UI Module, inclusion of clientlib containing the implementation (e.g., for *granite.profile* store the clientlib at */libs/granite/contexthub/components/modules/profile*)

## Authors

Aldo Caruso

## Bibliography

Adobe documentation about developing ContextHub
Adobe documentation about authoring ContextHub
Tutorial about enabling ContextHub in a project
Tutorial about audiences and activities management
Tutorial about segment engine