MyHealth Web Component Development & Integration Guide (v1.3)

MyHealth will be built using Web Components (WC) to ensure consistent UX across all possible integrations.

Those components must adhere to a set of inputs & outputs and a granularity that allows them to be placed in various configurations depending on the integration.

Granularity

A given functionality is typically composed of multiple views. A simple example is a list view and a details view showing the details of an item selected in the list.

In order for those list and details views to be displayed on mobile or in various layouts on the web, we need the functionality to be broken down into multiple individual web components.

In our simple example, we would end up with 2 components: list and details

The way to break down a given functionality into 1 or more web components is left up to the development team.

The general rule however is to break things down when the pieces of the functionality are expected to be organized in different layouts depending on the platform or device they will be deployed (mobile or various web apps with different layout requirements).
In the list & details example above, we know that:

- on mobile we'll show the list on one screen and the details on another screen
- on one web app we might want to show the list on the left of the page and the details on the right
- on another web app we might want to put the list above the details
- etc.

In order to achieve this we'll want to break down the overall functionality into 2 components: list and details.

Family of Web Components

As described above, web components are fairly granular. This means that a set of features would likely be broken down into multiple web components.

This set of web components belong to what we will call a **family**.

For instance, Vidis can be a family (composed of multiple components for prescriptions, medication schemes, etc.).

There are a few considerations linked with this concept of family:

- all components of a given family share the same (in-memory) cache (see inputs below)
- all components of a given family share the same offline store area (in SQL we would say 1 family = 1 database table)

Styling

Web components must provide support for styling but the actual styling definition will come from integrators (hosts).

Styling is based on Material Design version 3.

Web components and integrators using the Angular framework are strongly invited to leverage Angular Materials v19+.

The MyHealth Design Kit is also available (location TBD) as a reference.

Web Component input/output specification

See the "`@smals-belgium/myhealth-wc-integration`" repository for the definition of the types used for the inputs and outputs above.

Inputs

Each Web Component accepts the following inputs:

Input	Type	Description
version	string	Indicate the version of the spec the Hosts uses. Component can compare this value with the version of the spec they use and ensure both speak the same language.
`language`	string	Language used to display information to the end user. One of: EN, NL, FR, DE
`configName`	string	Name of the configuration the application and the components are being deployed to. One of: DEV, DEMO, INT, ACC, PROD
`services`	`ComponentServices`	Set of services to be consumed by the component. See below.

version

The "`componentSpecVersion`" defined in the "`@smals-belgium/myhealth-wc-integration`" library provides the current spec version and should be used by both Hosts and components.

Versions have the format `<major>.<minor>`.

Major version changes indicate breaking changes, typically the spec types have been changed and are not backward compatible.
Minor version changes indicate non breaking changes. Components should accept versions from hosts that have the same major version but not necessarily the same minor value.

configName

The objective of this input is to indicate to a web component the kind of environment it's been deployed into, which can imply contacting specific backend servers or having different internal implementations based on the provided config.

The possible configuration values are:

- DEV: development
- DEMO: demo mode, see the WC development & integration sections below for more information about this
- INT: integration
- ACC: acceptance
- PROD: production

`services`

The services input holds an object with multiple services to be consumed by the component:

```
{

       cache: ComponentCache,
       offlineStore?: ComponentOfflineStore,

       getAccessToken: GetAccessToken,
       registerRefreshCallback: RegisterRefreshCallback

}
```

cache (required)

The cache service provides methods to store, retrieve and remove data in and out of the cache.

This cache is an in-memory store that is specific to each component family and is never persisted.
It is used to share any kind of information between components of the same family.

This cache will be destroyed on each page reload and components should never create expectations on the presence of data in the cache.
Hosts are free to clear the cache at any given moment.
Hosts however are required to always provide a valid cache to components.

The cache is not to be confused with the offline store, whose objective is to persist data between runs of the Host.

The `ComponentCache` provides the following functions:

- **get**: `(key:string) => Promise<any>`
  Return the data associated with the given `key` from the cache.

Return null if no such data exist in the cache.

- **set**: `(key:string, value:any) => Promise<void>`
  Store the given `value` in the cache and associate it with the given `key`.

- **remove**: `(key:string) => Promise<void>`
  Entirely delete from the cache the data associated with the given `key`.
  Does nothing if the given `key` does not exist in the cache.

offlineStore (optional)

The offline store service provides methods to store, retrieve and remove data from the store.
It is a key-value store where the data is persisted on the user's device.
This store also provides a way to encrypt the data before it is being stored.

The service is optional and may not be injected in components.

Note that each family of web components is supposed to receive their own area in the offline store.

The `ComponentOfflineStore` provides the following functions:

- **get**: `(key:string) => Promise<any>`
  Return the data associated with the given `key` from the store.
  Return null if no such data exist in the store.

- **set**: `(key:string, value:any, encrypt:boolean=false) => Promise<void>`
  Store the given `value` in the store and associate it with the given `key`.
  First encrypt the given `value` when the `encrypt` parameter is true.

- **remove**: `(key:string) => Promise<void>`
  Entirely delete from the store the data associated with the given `key`.
  Does nothing if the given `key` does not exist in the store.

The sample web components show an example on how to work with an optional offline store.

getAccessToken (required)

Function signature: `GetAccessToken = (audience:string) => Promise<string|null>`

This method performs a token exchange given the `audience` and return the exchanged token

The token exchange mechanism must thus be implemented by the Host.

registerRefreshCallback (required)

Function signature:
```
  RegisterRefreshCallback = (callback: RefreshCallback) => void
  RefreshCallback = (done:()=>void) => void
```

This function can be invoked by components to register a callback function that will be called when the Host wants to trigger a refresh of the component.

Note that the callback method takes a single argument "`done`", which is a function that the component must call when the refresh is complete:

```
        callback( done:()=>void )
```

Outputs

Web Components also provide the following outputs:

Output	Type	Description
`onError`	`({title:string, text:string}) => void`	Fired to report an error to the Host.

Web Components may also provide their own specific outputs, which will need to be properly documented (see below about the section on Documentation of web components)

Web Component Development Guide

Developers are free to use any tech stack to build their web components (https://www.webcomponents.org).

The only constraint is to properly implement the inputs and outputs as described above, and to provide the documentation as described below.

Sample web components can be found at: https://github.com/smals-belgium/shared-myhealth-wc-integration-samples

Inputs

version

Components are expected to validate the version value provided by the Host.

Differences in the minor value are acceptable but differences in the major value should be rejected (breaking changes, spec seen by the Host is not compatible with the one seen by the component)

configName

When injected with the `"demo"` value it is expected that only static/mock data will be rendered.

This mode has been designed to have an integration free of real authentication and backend services. It is meant to run the application without dependencies on real data, for demo purposes.

As a web component developer, you are expected to embed mock data required by your component and use that data when in demo mode.

services

The `services` object contains services to be consumed by each component, see the description of Inputs above.

Outputs

Components must expose outputs that they need and properly document them so that integrations can respond to them appropriately.

All components must also expose the `onError` output.

OnError

This function can be used by components to let the Host display an error to the end user.

Documentation

Each web component must be properly documented.
This documentation must include the following information:

- Tag name associated with that component
- Description of what the component does
- Component family name (e.g. "Vidis Prescriptions")
- List of all outputs that the component exposes. Each output must indicate:
    - name of the event
    - optional parameters
    - description of that output, when it's being fired, etc.
- List of all the services that will be required by the component, such as Location or Calendar services.

The "`README-component.template.md`" file found in the "`@smals-belgium/myhealth-wc-integration`" repository gives a Markdown template to document your components.

Error Reporting

When developing web components it is important to ensure that any error generated by the component automatically propagate to the host/integrator.

This is needed so that the integrator can deal with errors in a uniform manner across web components.

The way to achieve this will differ based on the tech stack being used.

Web components built using Angular for instance would add the following code to their `main.ts` file:

```
import { createApplication } from '@angular/platform-browser';
import { ErrorHandler } from '@angular/core';

class WebComponentErrorHandler implements ErrorHandler {
 handleError(error:any) {
   throw new Error(error)
 }
}

createApplication({
 providers: [
   { provide: ErrorHandler, useClass: WebComponentErrorHandler },
 ]
})
.then((app) => {
  // TODO
})
.catch((err) => console.error(err));
```

Web Component Integrating Guide

Each Host (the "integrator") is free to manage the layout of components and to define the navigation and routing that works for them.
For instance, in response to an event fired by a web component, the Host could decide to navigate to a new route or simply update another view on the current web page.

Hosts are also responsible for the user authentication and token exchanges (see `services` below).

Sample host implementations for the web and mobile can be found at: https://github.com/smals-belgium/shared-myhealth-wc-integration-samples

Inputs

configName

An integrator can set the configName input with the `"demo"` value to indicate it is running in demo mode.

When in this mode, all data (including the one from all web components) is expected to be static and not coming from real servers.

This mode has been designed to have an integration free of real authentication and backend services. It is meant to run the application without dependencies on real data, for demo purposes.

services

See the service input description above.
All fields are required except for the `offlineStore` one.

The services property must always be injected into components.

cache

For security reasons, Hosts are expected to provide a separate cache for each family of components.
A global cache shared amongst all components can create a **security risk** by giving any component access to cached data they should not see.

As already explained before, the cache **cannot be persisted**, ever! The offline store is there to provide persistence between executions.
The cache should be cleared at least when the user exits or enters the application.

The Host is also free to clear those caches anytime it desires, components cannot create expectations on the availability of the data they place in it.

offlineStore

This service is optional and only provided by Hosts that support persistence and offline storage.

If provided and as with the cache, offline stores must be implemented per component family.
A store global to all components can create security risks and should never be implemented that way.

The offline store is different than the "in-memory" cache described above and pursue different objectives:

- the offline store stores data permanently on the user's machine, possibly in an encrypted way
- the cache can be seen as a plain JavaScript object given to components to share data between themselves.
  The cache starts empty every time the host application is (re)started.

The sample host for mobile provides an example of an offline store with the proper encryption mechanism and a way to securely store the encryption key.

getAccessToken

Hosts must be able to provide a valid authentication or exchanged token at any time through this method.
They must manage token refresh of authentication tokens and the token exchanges based on the given audience.

Hosts can cache those tokens but as long as those remain valid

registerRefreshCallback

This function can be called by components to register a callback that can be called by the Host when a refresh of the component is desired.

Hosts are thus supposed to store references to the callbacks provided through this function and call them when a refresh of the component is needed.
Note that it is allowed for a single component to register multiple callbacks.

Callbacks take a single "`done`" argument, which is a function that components must call once done with the refresh. This can give a chance for the Host to terminate the refresh process (e.g. hide a spinner)

Outputs

Integrators should respond to events sent by components, which should be part of the component's documentation.

The following `onError` output is always exposed.

OnError

Components will call this method to let the Host display an error on the screen.

Document History

Version	Date	Who	Changes
1.0	08-jan-2025	Laurent Brucher	Initial version, reviewed by: Dylan Cabal, Alphonse Van Assche, Laurent Lamouline
1.1	20-jan-2025	Laurent Brucher	Reviewed with Sopra Steria
1.2	03-feb-2025	Laurent Brucher	Refactored input `services.accessToken` into `services.getAccessToken()`
1.3	12-mar-2025	Laurent Brucher	Added new "`demo`" `configName` value Added Error Reporting in WC Development section Styling section updates