



Contents

Quick Start.....	3
Introducing Cells	3
Cells ecosystem.....	4
Start Playing	5
Codelabs.....	17
Advance Guides.....	18
Cells Architecture.....	18
Composition	18
Application Start & End.....	21
Data Flow in Cells Applications	33
Cells Bridge.....	36
JSON page definition.....	36
Static page definition / Routable components	49
Connect components.....	62
Navigation	80
Customization service	85
Hooks	86
Event Channels (Hooks)	90
Manual integration	95
ES6 Support.....	96
Components.....	98
Templates.....	98
Data Components	99
Cross components.....	105
User Interface components	109
Components catalog	114
Styling User Interface components.....	116
Internationalization.....	125
Theme	125
Application level dependencies	128
Best Practices	130
Style Web Components	130

:host selector	130
SASS.....	131
Performance	131
Other "know-hows"	132
Using icons	134
Iconsets	134
Icons in components	135
Adding new icons to `coronita-icons`	136
Guides for designers	137
a11y and a11y for mobile	137
Handy resources	137
Follow Golden Standards	137
Minimum accessibility checklist.....	137
Implement a11y	138
WCAG applied to mobile (summary)	141
Related with the FIRST principle: Perceivable	141
Related with the SECOND principle: Operable	142
Related with the THIRD principle: Understandable.....	142
Related with the FOURTH principle: Robust.....	143
How to create a responsive element	143
We will use	144
Testing components.....	145
Run tests with Cells Cli.....	145
Run tests interactively	146
What do I need test for a component	146
End-to-End Testing.....	150
Dependencies.....	150
How to run it	150
About the tests.....	151
More information	155
Native	155
Hybrid App	155
Android App	155

Quick Start

Introducing Cells

Cells is an architecture built over the idea that instead of building applications is better to build a set of reusable components (business modules) that are assembled and communicated to create applications.

Cells was designed to enhance reusability of components and dynamism of applications.

If your application logic involves a lot of data communications among components and requires different configurations for different user experiences, it would be the case for which you can take the most advantage of the Cells architecture.

At a first glance a Cells application is a set of **pages** where each page is defined as a **layout** and a **list of components**. For each component there is a definition of its properties, its communications with other components and the layout zone where the component will be injected.

Since each page is **defined** (in a json format), you can define dynamically (using a backend service) your application, having the possibility to create different user experiences for different users. Cells applications are assembled at runtime.

You can configure complex applications according to different parameters such as device, user profile, environmental properties (such as backend workload) among others.

Cells architecture is valid for different technologies, it was initially designed for HTML5 applications but nowadays supports also native (ios & android) developments. In fact, you can build hybrid applications mixing html5 & native components.

There are different components that you can build to create an application:

1. [User Interface Components](#): It is a component with an associated UI & UX. An example would be a list, a slider, a button, a header, a chart... Depending on the granularity of the component we can differentiate atoms (very small like a button), molecules (an aggregation of atoms like a textbox plus a button) or an organism (an aggregation of molecules and atoms like a login form). Visual

components should never call directly to an api or should change the location of the page. Instead they must have properties, methods or events that can externally be mapped with a navigation or with the resulting payload of an api call.

2. [Data Components](#): These components are responsible for calling an api and exposing the resulting payload using properties or events. If the component just exposes a single API we call it a data provider, but if the component exposes many different apis and logic we call it a Data Manager. Imagine calling a login api, that is encapsulated with a data provider, but if you need to call a login api and then a user profile then you should create a data manager.
3. [template](#): Templates are layout components. They are divided in zones where other components will be injected. Each page requires a template. The template can be the same for all the pages or it can be different. Having different templates for different devices you can have a responsive application. Each component needs to be responsive too in order to have a fully informational responsive application.
4. [\(c.\) Cross Components](#): These components are injected in a zone that is present in all the pages of the application. They can be controllers, data mappers... Also they can be data or visual components. If you need for example a session controller, popups, spinners or whatever that you are going to use in all the application, tag them as cross components.
5. Native components: When creating a mobile app, sometimes it makes sense to create a native landing page to have an extra level of performance and user experience. Cells allows to combine native pages (developed with native components) with html5 pages in a single reactive architecture. There is detailed documentation for android ([UI](#) & [Data](#)). IOS documentation is underway.
6. Widgets: You can pack a UI component with a Data component in order to build a widget component. Widgets are just a high level component that should work just like a plug & play appliance. All the logic is encapsulated in the component. The widget should offer properties to customize the behaviour. Think for example in a stock chart component, with an associated data component calling to the yahoo stock data api, all encapsulated in a single stock chart widget. Widget family coming soon.

Cells ecosystem

There are several items that you need to know in the cells ecosystem.

1. Bridge

[Cells Polymer Bridge](#) will read each [page's configuration](#) and automatically create the page parting from the given configuration. You can think of an application as a set of pages. In each page's configuration you can define the [template](#) where you want to place the [visual components](#), the [connections](#) between the visual and the [data components](#) and the [navigations](#). Or have [global parameters](#) set to use across pages.

2. Catalog

In the [Cells Components Catalog](#), you'll find a great collection of the current certified Cells **Production Ready** and **Beta** components, categorized by families. Mostly you will find [User Interface Components](#), which are the visual custom elements, that paint the data coming from the [Data Components](#). Also, you can relay on the [\(c.\) Cross Components](#) to have the common functionality across the application.

3. Composer (Coming soon)

Composer is the Cells visual tool, with drag & drop capabilities and a backend that will store configurations for your pages. Composer allows you to create different visual experiences for your application depending on different variables as explained before. Read more in the [customization service guide](#).

4. CLI

The [Cells CLI](#) is a development environment to help you out build applications and components with Cells faster. It also will help you with the testing and packing of your app. Currently works only in Mac & Linux.

Start Playing

Install Cells and its environment

Before starting to develop,

1. Prepare the environment following the [Installation guide](#)
2. Then install the [Cells' command line tool](#)

```
3. $ npm install -g cells-cli
```

4. Setup Cells environment. This will help you to configure ssh and artifactory

```
5. $ cells environment:setup
```

Logging in previously to globaldevtools.bbva.com, in order to generate an **API key from artifactory** for instance.

6. To download Cells components and projects for reference:

```
7. $ cells workspace:create
```

Create a Cells application welcomeToCells

Just execute cells in the command line tool and select [app:create](#)

```
$ cells
```

```
> app:create [install & execute] (Context creation flow)
```

It will prompt you with two options, for now go with the following:

```
? Choose app scaffold (Use arrow keys)
```

```
> Blank cells app
```

```
? What is the project name? welcomeToCells
```

Include some components in your application

From [Cells Catalog](#), pick for example `<cells-switch>` and `<cells-radio-button>` to install them in your application. You will also need a template, take for example `<cells-template-paper-header-panel>`:

```
$ bower install --save cells-template-paper-header-panel#^1.1.0
```

```
$ bower install --save cells-switch
```

```
$ bower install --save cells-radio-button
```

Since the components use icons, install also an iconset as [an application dependency](#). For instance, the `coronita-icons`:

```
$ bower install --save coronita-icons
```

JSON page definition

As explained above, in a Cells application you need to define your pages, and each page will define the components and the communications with other components.

The initial route and page name of your application is configured in the `app/scripts/app.js` file. By default an initial 'login' page is defined in the route '/', so you need to configure the `login.json` with the layout and the components that you want to use.

The `login.json` is in the `app/composerMocksTpl` directory

Lets modify the `login.json` with the following:

```
{
```

```
"template": {  
  "familyPath": "cells-template-paper-header-panel",  
  "tag": "cells-template-paper-header-panel",  
  "properties": {  
    "mode": "seamed",  
    "zones": [  
      "app__header",  
      "app__main",  
      "app__footer"  
    ]  
  }  
},  
"components": [  
  {  
    "zone": "app__main",  
    "tag": "cells-switch",  
    "properties": {  
      "withIcons": true,  
      "iconChecked": "coronita:nfcconnect",  
      "iconUnchecked": "coronita:nfcdisconnect",  
      "iconToggle": "coronita:on"  
    }  
  },  
  {  
    "zone": "app__main",  
    "tag": "cells-radio-button",
```



```

    "properties": {
      "icon": "coronita:creditcard",
      "iconCheck": "coronita:checkmark"
    }
  }
]
}

```

This configuration will load the selected template from the [Cells Catalog Template family](#) and will inject the components in the selected zones. It will also configure the component's properties with the configuration provided in the properties section of each component.

Serve your Cells application

To see your app working you have to install the components and serve it to a browser. It will then refresh with every change you make. You will see the resulting page with the *two isolated components*. Meaning that toggling one doesn't affect the other one (for now!)

```

welcomeToCells$ bower install

welcomeToCells$ cells

> app:serve-app (Serve the distribution of a Cells
application)

? Choose file compress type(s) for the distribution

> for developing

? Choose the configuration for the distribution

> local

```

Connect the components

Now lets try to connect the switch with the radio button. For this we are going to use `cellsConnections`. The structure of a `cellsConnection` is represented in the following diagram.

What we are going to do is to **listen** in cells-radio-button the event **emitted** by cells switch. For that we are going to create a channel where this communication will happen.

If you check the `<cells-switch>` documentation, in the Events section there is a `cells-switch-changed` event that is triggered when checked value is changed, with current status as payload. So we are going to publish (`"out"`) that payload every time there is a change in the check value. As stated before, to publish or to subscribe to information you need a channel. To create a channel just pick the name that you like, for instance `"toggle-channel"` and include it in the `cellsConnections` section of a component. So the resulting configuration would be something like this

```
{
  "zone": "app__main",
  "tag": "cells-switch",
  "properties": {
    [.....]
  },
  "cellsConnections": {
    "out": {
      "toggle-channel": {
        "bind": "cells-switch-changed"
      }
    }
  }
}
```

At this point we are emitting in "toggle-channel" the state of the cells-switch. Now we are going to listen to that value in the `<cells-radio-button>` component. For that in `<cells-radio-button>` we are going to use the `"in"` connections to receive that value. The resulting json would be:

```
[.....]
```

```
{
```

```
  "zone": "app__main",
```

```
  "tag": "cells-switch",
```

```
  "properties": {
```

```
    "withIcons": true,
```

```
    "iconChecked": "coronita:nfcconnect",
```

```
    "iconUnchecked": "coronita:nfcdisconnect",
```

```
    "iconToggle": "coronita:on",
```

```
    "cellsConnections": {
```

```
      "out": {
```

```
        "toggle-channel": {
```

```
          "bind": "cells-switch-changed"
```

```
        }
```

```
      }
```

```
    }
```

```
  }
```

```
},
```

```
{
```

```
  "zone": "app__main",
```

```
  "tag": "cells-radio-button",
```

```
  "properties": {
```

```
    "icon": "coronita:creditcard",
```

```
    "iconCheck": "coronita:checkmark",
```

```
    "cellsConnections": {
```

```
      "in": {
```

```

      "toggle-channel": {
        "bind": "checked"
      }
    }
  },
  [.....]

```

Within the channel, you can bind an event, a method or a property, but learn more on different bindings and communicating components in the [connect components guide](#)

Add a new page to your app

Lets create a 'home' page in our application. For this we need to add the route in the app.js file and to create the home.json file in the composerMocksTpl directory.

Lets modify first the app.js file to add the home page.

```

var bridge = new window.CellsPolymerBridge({
  [.....]
  routes: {
    'login': '/',
    'home': '/home'
  }
}

```

Now lets create a new home.json with only the cells switch component

```

{
  "template": {
    "familyPath": "cells-template-paper-header-panel",
    "tag": "cells-template-paper-header-panel",

```

```

    "properties": {
      "mode": "seamed",
      "zones": [
        "app__header",
        "app__main",
        "app__footer"
      ]
    },
    "components": [
      {
        "zone": "app__main",
        "tag": "cells-switch",
        "properties": {
          "withIcons": true,
          "iconChecked": "coronita:nfconnect",
          "iconUnchecked": "coronita:nfcdisconnect",
          "iconToggle": "coronita:on"
        }
      }
    ]
  }
}

```

If you load the following url you should see your new page:
<http://yourlocalhosturl/index.html#!/home>

Configure a navigation

Now lets configure a navigation from login to home. For that you just need to stablish an out action with a link propertie. Following with our example lets create an out channel in the cells-radio-button of the login.json page like this:

```
{
  "zone": "app__main",
  "tag": "cells-radio-button",
  "properties": {
    "icon": "coronita:creditcard",
    "iconCheck": "coronita:checkmark",
    "cellsConnections": {
      "in": {
        "toggle-channel": {
          "bind": "checked"
        }
      },
      "out": {
        "foobar-channel": {
          "bind": "cells-radio-button-checked",
          "link" : {
            "page" : "home"
          }
        }
      }
    }
  }
}
```

```
}
```

We have binded the cells-radio-button-checked event (check its documentation) with a navigation to the home page. We have used a foobar-channel but you can use the name that you like. It's that simple. There is a [Navigation guide](#) where you can find info about how to send info from one page to the other.

Create a Cells component <cells-drink>

Just execute and select to create component:

```
$ cells
```

```
> component:create [install & execute] (Context creation flow)
```

It will prompt you with some options, for now go with this basic selection:

```
? Select the type of the component
```

```
Behavior
```

```
> Component
```

```
Data Manager
```

```
Theme
```

```
? Component's name... cells-drink
```

```
? Component's description (Your component description)
```

```
? It's an hybrid component? (y/N)
```

```
> No
```

```
? Is your component going to be multilanguage? (use i18n) (y/N)
```

```
> Yes
```

```
? Would you like to use a theme or/and icons? (y/N)
```

```
> Yes
```

```
? Would you use a theme? (Y/n)
```

```
> Yes
```

```
? Choose a theme for your component's demo
```

```
> ☒ cells-coronita-theme
```

```
? Would you use an iconset? (Y/n)
```

```
> Yes
```

```
? Choose the iconsets for your component's demo
```

```
> ☒ coronita-icons
```

Now you can serve the component:

```
cells-drink/$ cells
```

```
> component:serve (Serve your components)
```

It will open a brand new Cells Component in your browser, enjoy!

Start your project

To have a Web Components oriented application, take in account that is very important [the composition](#) of your visuals and functionality.

Use the [Application Start guide](#) to have a notion on how to serve, configure and build an application (or follow the PRPL in your Cells application).

Check also the [demo app](#) that we have created in the Cells Team.

Also take a look at the advanced guides to learn howto pack your app, test it, configure the continuous integration and more.

Basic Continuous integration for your project

As explained in the [CI how to start guide](#) you just need to include a jenkinsfile in your project at root level. The jenkinsfile for example to build and deploy in S3 your project would be something like this:

```
globalrepoBranch="master"

stage('Checkout Global Library') {
  def cells_ci, cells_node_label
  cells_node_label = 'ios';
  node (cells_node_label){
```



```

    globalrepo = "${env.globalrepo}"

globalrepo_credentials_id="${env.globalrepo_credentials_id}"
}
def repoParams = [
    "cellsNative": true,
    "piscoOpts" : [
        "type": ["vulcanize"],
        "config": "env",
        "platforms" : [ "webapp" ]
    ]
];

fileLoader.withGit(globalrepo, globalrepoBranch,
globalrepo_credentials_id, cells_node_label) {
    cells_ci = fileLoader.load('src/cells/ci/ci');
    cells_ci.cells_ci_flow(repoParams)
}
}

```

The repoParams propertie contains the configuration for your project, so you should focus only in this part of the configuration:

```

def repoParams = [
    "cellsNative": true,
    "piscoOpts" : [
        "type": ["vulcanize"],
        "config": "env",
        "platforms" : [ "webapp" ]
    ]
];

```

- config is the config environment file (As explained in the [app:create](#) guide) of your app that you want to use to build your application. You can have for example a play.json file or a work.json file in your config directory. By default cells apps have an env.json and a local.json. But you can change them to whatever you want.
- platforms are the different artifacts that you want to build (webapp,ios or android)
- type can be vulcanize or novulcanize. Apps are vulcanized to optimize the deployment (files are compressed and minified).
- cellsNative is just to let jenkins know that a Mac agent must be used. Dont change this value.

More possible configurations can be found in the CI guides

Codelabs

Cells Codelabs provide a guided, tutorial, hands-on coding experience. Most codelabs will step you through the process of setup up your Cells development environment, building components, testing components and applications, deploy your components and everything related with Cells Architecture!

<https://bbva-files.s3.amazonaws.com/cells/codelabs/index.html>

Advance Guides

Cells Architecture

Composition

Cells recommends using composition to **reuse code** between components. In fact, the key feature of Cells is composition pattern to design components and applications.

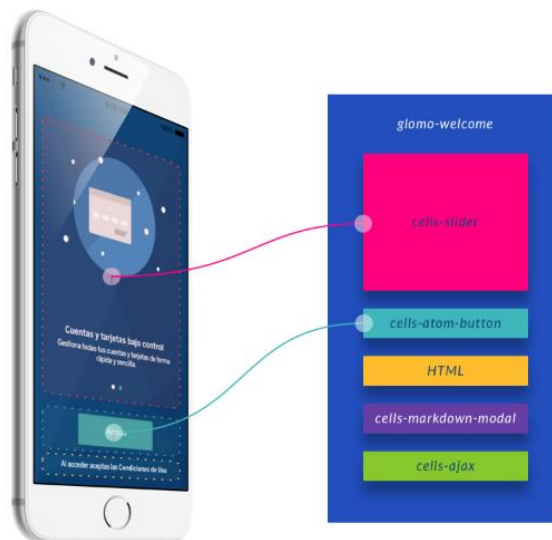
Components are often described as "just functions" but in our view they need to be more than that to be useful.

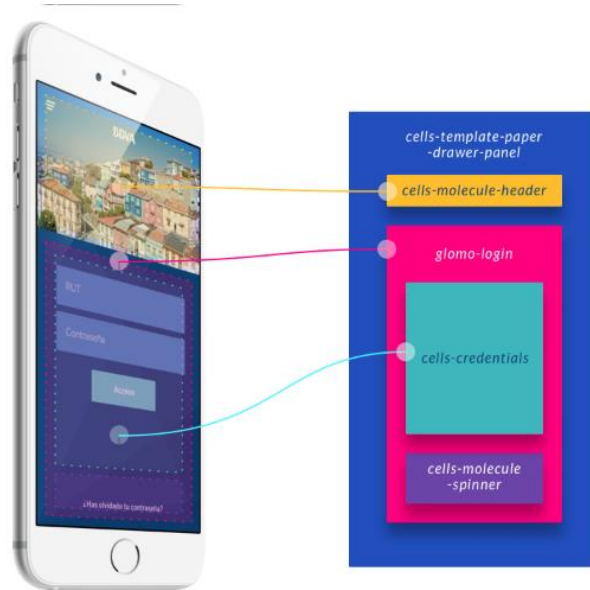
In Cells, components describe any composable behavior, and this includes **rendered UI and data**.

1. Examples of decomposing a visual

Decompose the view into components, following [atomic design](#) in its fundamentals:

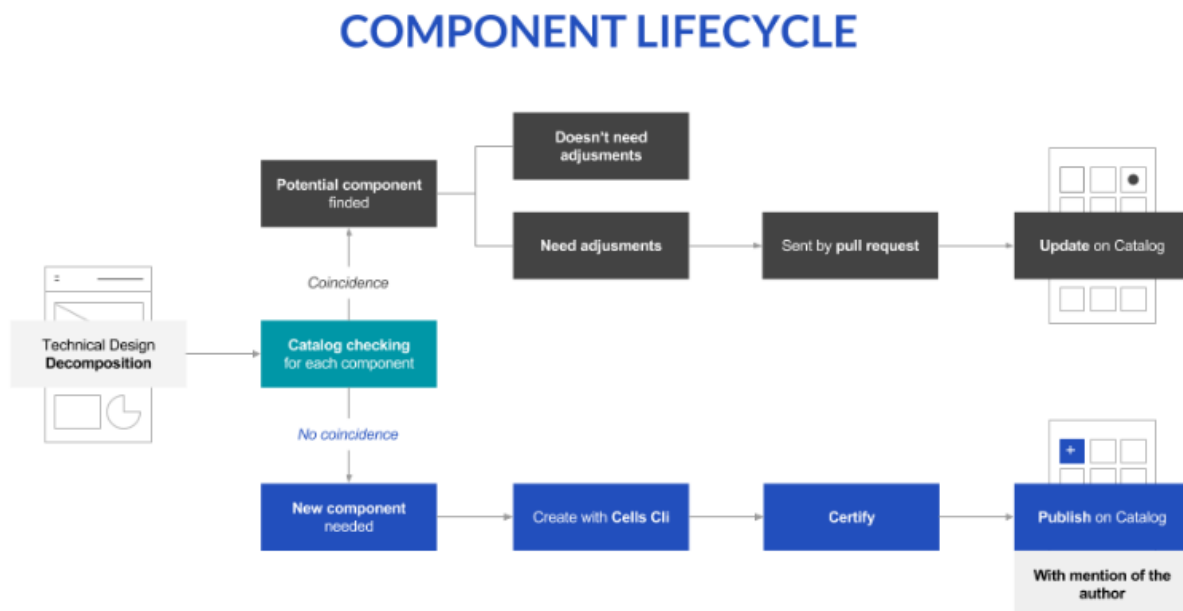
"...interfaces are made up of smaller components. This means we can break entire interfaces down into fundamental building blocks and work up from there."





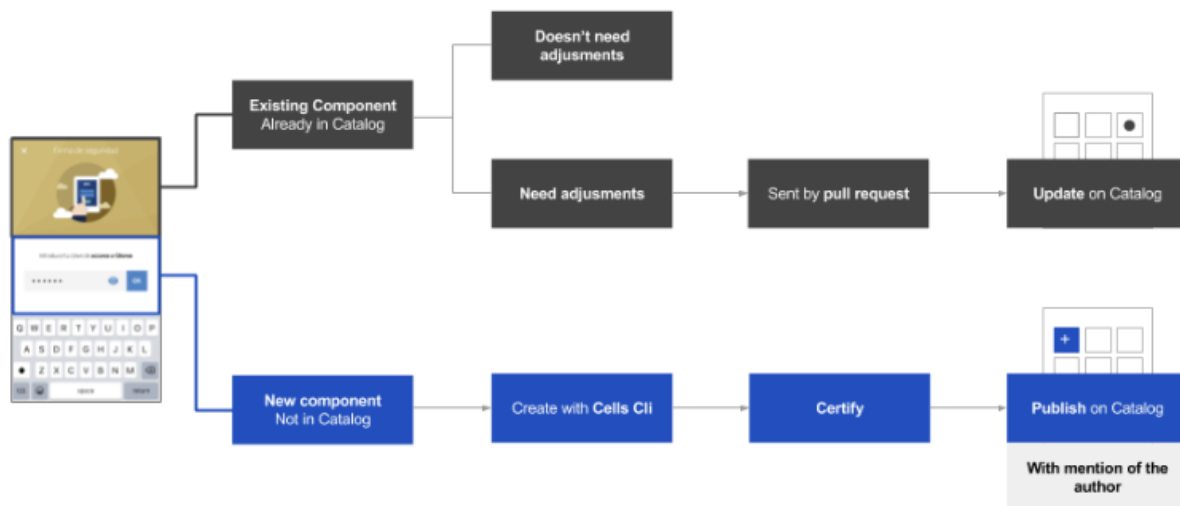
2. Component lifecycle

After the decomposition, the components' natural flow should be the following:



If you find an eligible one (with maybe some minor changes) in [Cells Components Catalog](#), then have a look to the [contributing guide](#)

COMPONENT LIFECYCLE *EXAMPLE*



At Cells we try to make the components so they are not tied to the application context, making them more reusable.

For example, avoiding relating properties' or component's name too much to the application data. We also try to avoid making them **too abstract** or "too reusable" because it would make the configuration too tedious. We see as something positive to have maybe two similar components but with different approaches, since it would not be very useful nor maintainable to have it all in one component.

Application Start & End

- [Create](#)
- [Serve](#)
- [Build](#)
- [The PRPL pattern](#)
- [Split bundles](#)
- [Pre-render](#)
- [Elements](#)
- [Logout](#)

Create

You can create a Cells application with the CLI generator as follow:

```
$ cells app:create
```

```
? Choose app scaffold (Use arrow keys)
```

```
> Blank cells app
```

```
Full cells app
```

```
? What is the project name? welcomeToCells
```

This command will create a folder with all the subfolders and files needed by a Cells application.

Among the created folders you will find:

- `app/composerMocksTpl`, to put your page definition files.
- `app/config`, to configure each environment.
- `app/scripts`, in this folder it's the `app.js` file. which it has the code to boot your application.
- `app/tpls` contains templates needed to build and serve the application

Let's see the parts that we may need to modify:

The environment files

In these files you set the configuration for the application, for example `app/config/env.js`:

```
{
```

```
  "deployEndpoint": "",
```

```

    "i18nPath": "locales/",
    "componentsPath": "./components/",
    "composerEndpoint": "./composerMocks/",
    "debug": true,
    "mocks": true,
    "coreCache": false,
    "routerLog": false,
    "initialBundle": ["login.json"],
    "prplLevel": 1
  }

```

- `deployEndpoint`: This is the base location (URL) for your application, components and configurations. All the other routes are appended to this base location. If no location is provided all the routes will be considered local. This is useful if you want to deploy your `index.html` in a different location than the application bundle (i.e: a cordova app deployed in a CDN or a webapp deployed behind a reverse proxy).
- `componentsPath`: This is the path or URL where the components are located (in relation to the `deployEndpoint`)
- `composerEndpoint`: This is the path or URL where the json definition files are located.
- `i18nPath`: [i18nPath](#) is the path where the merged localization file, resulting of the merge of all components localization files, will be deployed
- `initialBundle`: See below
- `prplLevel`: Not implemented yet. Coming Soon.
- `debug`: If true it will display logs produced by the cells bridge
- `mocks`: If true it will launch a stubby server that can be used to mock your data components. This mocks need to be stored in a mock directory at root level of your project. An example can be found [here](#)
- `coreCache`: If true it will cache the json definition files.
- `routerLog`: If true, it will display logs produced by the cells router bridge.

script/app.js

This file (`script/app.js`) has the logic to start the application. Here you can do special adjustments, for example, set properties for **Cells Bridge**, change the way the views are rendered, set the initial page, etc.

Serve

[Cells-CLI](#) lets you serve the application locally so that you can test it while you continue working on it. The application can be served in three different ways:

- `no dist` the application is served directly from the components folder.
- `dist no vulcanized` the application is prepared to be served as in production mode. The files are minified but kept separated. It is the best option for servers that use HTTP/2.
- `dist vulcanized` the application is prepared for production, all the files are concatenated and minified in one file. This is the recommended for servers that don't support HTTP/2.

Build

The Cells tools also build the application for distribution. Again, it can be `dist no vulcanized` **OR** `dist vulcanized`.

The PRPL pattern

As we've seen before, the vulcanized option makes only one file with the whole application, which is the one that it's loaded when the application starts:

`app.js`

```
...  
  
function loadElements() {  
  
    var bundle = document.createElement('link');  
  
    bundle.rel = 'import';  
  
    bundle.href = window.AppConfig.deployEndpoint +  
window.AppConfig.componentsPath + 'components.html';  
  
    bundle.onload = finishLazyLoading;  
  
    document.head.appendChild(bundle);  
  
}  
  
...  
  
function finishLazyLoading() {
```



```
setShadow();
```

```
setI18n();
```

```
startCore();
```

```
fireComponentsLoadEvent();
```

```
}
```

```
...
```

```
function startCore() {
```

```
  var sections = {
```

```
    'home': '/',
```

```
    'events': '/events',
```

```
    'events-create': '/events/create',
```

```
    'info': '/events/:itemId/info'
```

```
  };
```

```
window.$core = new window.CellsPolymerBridge({
```

```
  debug: false,
```

```
  cache: false,
```

```
  componentsPath: window.AppConfig.componentsPath,
```

```
  templatesPath: window.AppConfig.composerEndpoint,
```

```
  routes: sections,
```

```
  onRender: function onRender(template) {
```

```
    if (!template.parentNode) {
```

```
      document.getElementById('app__content').appendChild(template);
```

```
    }
```

```
  }
```

```
});
```

```
loadElements();
```

```
}
```

With this approach, the loading of the application could take too long on slow connections because the initial request retrieves a very big file and the user has to wait until the request is fully served. This is a bad user experience. ☹️

The PRPL pattern solves this problem. **PRPL** is a pattern created to give the best user experience.

- Push
- Render
- Precache
- Lazy loading

See this [link](#) to know more about the PRPL pattern.

Since the version 0.6.0 of Cells the `R`, and `L` in `PRPL` are ready to use.

To enable this feature you will need Cells Polymer Bridge 0.6.0 and also update the Cells CLI tools.

Split bundles

Split bundles is a step further in the PRPL direction. With this approach there's an initial file (`initial-components.html`). This file has the essential content to do the initial render of the application and every page will have its content bundled in a separate file.

To use this mode, include these two lines in the config file of the environment:

`app/config/env.js`

```
{
```

```
...
```

```
"initialBundle": ["login.json"],
```

```
"prplLevel": 1
```

```
...
```

```
}
```

`initialBundle` is an array with the pages that must be bundled together and that are going to be rendered when the application starts. It must have, at least, the first page of the application.

The `app.js` has to change too:

```
...

// Load initial-components.html file

function loadElements() {

    var bundle = document.createElement('link');

    bundle.rel = 'import';

    /// BEGIN CHANGE

    /// it is initial-components.html instead of components.html

    bundle.href = window.AppConfig.deployEndpoint +
window.AppConfig.componentsPath + 'initial-components.html';

    /// END CHANGE

    bundle.onload = finishLazyLoading;

    document.head.appendChild(bundle);

}

...

/// BEGIN CHANGE

/// new function that will load the rest of components
/// that are in the file app-components.html

function loadAppElements() {

    document.removeEventListener('componentsInTemplateLoaded',
loadAppElements);

    var nextBundle = document.createElement('link');
```

```
nextBundle.rel = 'import';
```

```
nextBundle.href = window.AppConfig.deployEndpoint +
```

```
window.AppConfig.componentsPath + 'app-components.html';
```

```
nextBundle.setAttribute('async', '');
```

```
document.body.appendChild(nextBundle);
```

```
}
```

```
function shouldLoadAppElements() {
```

```
return window.AppConfig.initialBundle;
```

```
}
```

```
if (shouldLoadAppElements()) {
```

```
/// we listen to the event that will tell us
```

```
/// that the first components has been loaded
```

```
document.addEventListener('componentsInTemplateLoaded',
```

```
loadAppElements);
```

```
}
```

```
/// END CHANGE
```

```
function finishLazyLoading() {
```

```
removeSplashScreen();
```

```
fireComponentsLoadEvent();
```

```
}
```

```
function fireComponentsLoadEvent() {
```

```
var eventComponentsLoaded = document.createEvent('Event');
```

```
eventComponentsLoaded.initEvent('componentsLoaded', true, true);
```

```
document.body.dispatchEvent(eventComponentsLoaded);
```

```
}
```

```
...
```

```
function startCore() {
```

```
    window.removeEventListener('componentsLoaded', startCore);
```

```
    new window.CellsPolymerBridge({
```

```
        mainNode: 'app__content',
```

```
        debug: window.AppConfig.debug,
```

```
        cache: window.AppConfig.coreCache,
```

```
        binding: 'currentview',
```

```
        /// BEGIN CHANGE
```

```
        /// we set the bridge with PRPL
```

```
        prplLevel: window.AppConfig.prplLevel,
```

```
        /// END CHANGE
```

```
        componentsPath: window.AppConfig.componentsPath,
```

```
        templatesPath: window.AppConfig.composerEndpoint,
```

```
        routes: {
```

```
            'login': '/',
```

```
            'dashboard': '/dashboard',
```

```
            'account': '/account/:accountId'
```

```
        },
```

```
        onRender: function onrender(template) {
```

```
            if (!template.parentNode) {
```

```
                document.getElementById(this.mainNode).appendChild(template);
```

```
                /// BEGIN CHANGE
```

```

    /// Once the initial render has be done

    /// here we trigger the event to load the rest

    /// of the components

    if (shouldLoadAppElements()) {

        var eventComponentsLoaded = document.createEvent('Event');

        eventComponentsLoaded.initEvent('componentsInTemplateLoaded',

true, true);

        document.body.dispatchEvent(eventComponentsLoaded);

    }

    /// END CHANGE

}



}



});

});

window.addEventListener('componentsLoaded', startCore);

```

In the template file `apps/tpls/initial-component-import.tpl` put every component that is not explicitly required in any json file. **Don't remove the commented lines!**  

```

<link rel="import" href="polymer/polymer.html">

<link rel="import" href="cells-polymer-bridge/cells-polymer-bridge.html">

<!-- will be replaced with imports -->

<!-- will be replaced with dependencies -->

```

Pre-render

Since version 1.0.0 of Cells, it's possible to pre-render pages. This means that pages can be loaded and appended to the DOM in advance and kept hidden, so when you need to render the page it will be ready to be shown. The pre-render optimizes the time to show a page.

To use the pre-render a page definition (JSON file) must have the list of pages that can be accessed from that page and the bridge must have this properties set to `true: preCache` and `preRender`.

In `app.js`:

```
var bridge = new window.CellsPolymerBridge({  
  preCache: true,  
  preRender: true,  
  ...  
})
```

In the JSON page definition: `wellcome.json`

```
{  
  "pages": {  
    "login": {  
      "url": "/login"  
    },  
    "dashboard": {  
      "url": "/dashboard"  
    }  
  }  
}
```

In the example above, we have a page `wellcome` that can navigate to the login page or to the dashboard page so it declares these pages. When the Bridge is loading the `wellcome` page it will also load and pre-render the login and dashboard pages.

Elements

During the development phase, you can write your components under the `app/elements` folder and they will be served and vulcanized just like any other component. We provide this functionality to make the development task easier and faster. But remember, **it's just for the development phase**.

To enable this feature you will need to update the Cells CLI tools.

Logout

Every application should do some cleaning tasks before leaving, and it should generally happen when the user logs out.

Being this so common, Cells Bridge gives you a function that takes care of this task: it is the `logout` function.

`logout` does the following things:

- Removes every template from the DOM.
- Removes every template from the Cells' cache.
- Resets every channel (including the private channels).
- Resets every cross component.
- Navigates to the **another page** (usually the login page).

To use this function, you must access to the Bridge object which is in the `$score[0]` object and simply call it like this:

```
$score[0].logout();
```

The logout navigates to the **another page**, generally it's the initial page, so you must configure this in the Cells Bridge with the `initialTemplate` property:

```
initialTemplate = 'home';
```

For example, in the `app.js` it looks like this:

```
window.addEventListener('componentsLoaded', function() {  
  new window.CellsPolymerBridge({  
    initialTemplate = 'home',  
    ...  
  });  
  ...  
});
```

If this property is not explicitly set then the default value for `initialTemplate` will be `'login'`.

Data Flow in Cells Applications

or Cells architecture

1. Applicability
2. Implementation
3. Component API communication
4. Uncoupled navigation

Applicability

Cells library (Cells Bridge) implements the architectural *publish/subscribe pattern* with **unidirectional data flow**.

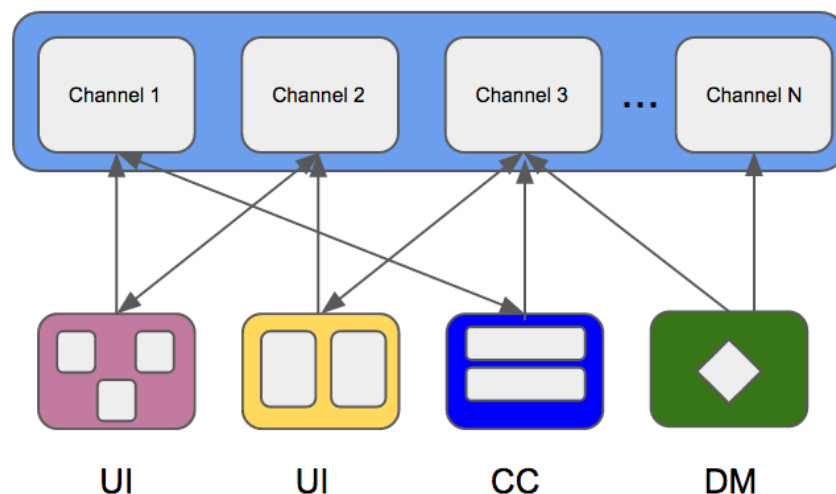
If your application logic involves a lot of data communications among components, it would be the case for which you can take the most advantage of this architecture.

Even though the pattern can be implemented just using built-in Polymer concepts, such as custom events and data binding, the Cells Bridge library provides a singleton module to manage all communications between components and helps to structure and simplify the code of the application.

Implementation

Cells Bridge provides the *publish/subscribe module* based on **channels** built on top of [RxJS](#). This pattern is one of the variations of the Observer designer pattern.

We can consider that Cells Bridge is a singleton application element on each project that will manage all communications of application components. Also, each element that needs to communicate with another one, directly or indirectly, sends events (emit actions) to one channel, and the other components are listening to those data changes.

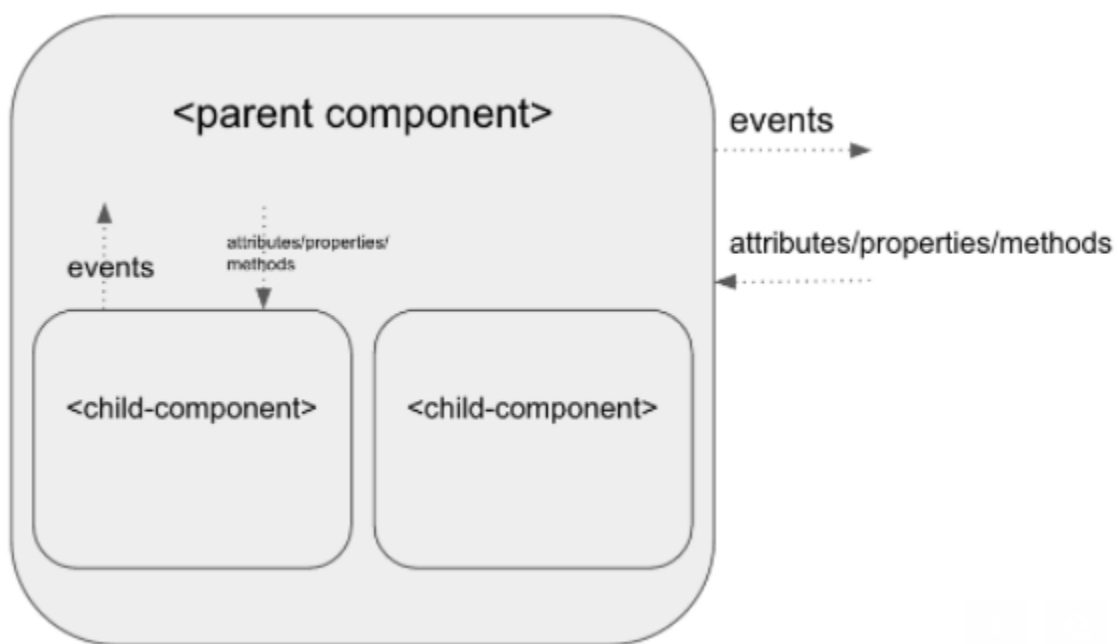


Component API communication

The communication among components, it is always offered from the components of "first level" (those that are instantiated in the page's definition JSON).

The children components of those "first level" components will communicate through events, as well as the "first level" components. These events will only have sense in the "first level" component's scope.

And so, the "first level" components are responsible to expose the communication API through events (actions "out") and properties, attributes or methods (actions "in").



Uncoupled navigation

It is not recommended that the components expose events with navigation context. For example: `go-to-other-page`. The recommendation is to associate any *action "out"* of a "first level" component (triggered events) to a navigation action in the page's definition. This way, a component offers an uncoupled and configurable functionality according to the application's context.

```
...
"out": {
  "go-dashboard": {
```

```
"bind": "name-of-event",  
"link": {  
  "page": "dashboard"  
}  
}  
}  
}  
...
```

Cells Bridge

JSON page definition

Local page definition files

When you create a Cells application, it has their page definition files (.js or .json) in a local folder, usually in the folder `composerMocksTpl`. During the build phase these files are transformed to json files (if they're js files) and moved to the folder `composerMocks` which is the folder from where they will be served.

The configuration file must have this entry:

```
"templatesPath": "../composerMocks/"
```

So when a page is required, Cells Bridge will know that the local pages are located in the `composerMocks` folder.

Remote pages

Cells applications may also have remote pages, that is, page definitions files that are served by Composer. To use remote pages you have to tell Cells Bridge which pages are remote and the url of Composer server used to request the definition files.

In the configuration file you will have:

```
"composerEndpoint":
```

```
"http://composer:20000/engine/apps/{appId}/platforms/{platform}/pages/{page  
}/page?uid=cells.admin"
```

```
"remoteTemplates": ["composerLogin", "userProfile"],
```

```
"appId": "composerHello"
```

- `composerEndpoint`: it has the composer's url with placeholders for `appId`, `platform` and `page`. Cells Bridge will inject the values for each one.
- `remoteTemplates`: it is an array with the names of the pages that will be served by Composer.
- `appId`: this is id that use Composer for the application

With this setting applications can use some local pages and get other pages from a Composer server.

Note. This feature is ready since Cells Bridge v3.3.0. In previous versions the only way to specify the location of page definition files was using **composerEndpoint** variable, so that it was not possible to have both remote and local pages at the same time.

Layout page configuration

You can connect your app to a customization service (like Cells Composer) or test it locally using mock configuration files (app/composerMocks). For now, we are going to use mocks for this example. So, create a new JSON file in the folder `app/ComposerMocksTpl`, for example `dashboard.json`.

The template is a layout component. Have a look to the Cells Components Catalog's [family of templates](#).

```
{
  "template": {
    "familyPath": "../components/cells-template-paper-header-panel",
    "tag": "cells-template-paper-header-panel",
    "properties": {
      "mode": "seamed",
      "zones": [
        "app__header",
        "app__main",
        "app__footer"
      ]
    }
  },
  "components": [
    {
      "zone": "app__main",
      "familyPath": "../elements/my-greeting",
      "tag": "my-greeting"
    }
  ]
}
```

```
}
]
}
```

The `familyPath` is not necessary to define it if the component is in `components/` folder. Since it is where the components install by default if they are as dependencies in the `bower.json`.

Place your components

In your project you'll have an `index.tpl` file which is used to build the `index.html` file of the app. In this template you must have a zone where the pages will be rendered within your SPA. More often this will be like this:

```
<body>

  <div id="app__content"></div>

  <script src="scripts/app-config.js"></script>
  <script src="scripts/app.js" async></script>

</body>
```

In the example above, the `div` with the id `app__content` is the zone where every template will be rendered. This zone is the main node of your application and it is defined in your `app.js` giving the id of the element:

```
window.addEventListener('componentsLoaded', function() {

  new window.CellsPolymerBridge({

    mainNode: 'app__content',

    ...

  });

});
```

The components that you have in the template are therefore placed inside the main node. The template of the main node typically has three different zones: header, content and footer. You place your components in these three zones using the JSON file of the template. Look at this definition:

```
"template": {
```

```
  "tag": "cells-template-paper-header-panel",
```

```
  "properties": {
```

```
    "mode": "seamed",
```

```
    "zones": [
```

```
      "app__header",
```

```
      "app__main",
```

```
      "app__footer"
```

```
    ]
```

```
  }
```

```
},
```

```
"components": [
```

```
{
```

```
  "zone": "app__header",
```

```
  "type": "UI",
```

```
  ...
```

```
},
```

```
{
```

```
  "zone": "app__main",
```

```
  "type": "UI",
```

```
  ...
```

```
},
```

```
...
```

```
],
```

```
...
```


In the example we say that we're using a template with 3 zones: `app__header`, `app__main` and `app__footer`. Then we define the components in those zones. The first one goes to the header zone, and the second one goes to the main zone.

Animated transitions and fixed zones

When you navigate from page to page, Cells animates the transition between pages in the main node, so every zone in the page will be animated. In some cases, you don't want this transition in a zone (the most common case is with footers) so Cells has a way to fix zones.

The fixed zone is a zone outside the main node. To fix a zone follow these steps:

1. In your `index.tpl` define an element to hold your fixed zone and give it an id:

```
<body class="fullbleed layout vertical loading">  
  <div id="fixed__header"></div>  
  <div id="app__content"></div>  
  <div id="fixed__footer"></div>  
  <script src="scripts/app-config.js"></script>  
  <script src="scripts/app.js" async></script>  
</body>
```

2. For every page configuration that has components in the fixed zone you have to define them, by using the id of the fixed zone for `zone` property and setting to true the `fixed` property:

```
"components": [  
  {  
    "zone": "fixed__header",  
    "type": "UI",  
    "fixed": true,  
    ...  
  },  
  {
```

```

    "zone": "fixed__footer",
    "type": "UI",
    "fixed": true,
    ...
  }
  ...

```

Note: You can only fix components that are `UI` type.

3. In your `app.js` when defining the `onRender` function you have to render the fixed components in the fixed zone. For most cases will be the same way:

```

onRender: function onrender(template, fixedComponents) {

  if (!template.parentNode) {

    document.getElementById(this.mainNode).appendChild(template);

  }

  // Get the fixed zones

  var fixedHeaderZone = document.getElementById('fixed__header');
  var fixedFooterZone = document.getElementById('fixed__footer');

  // Clean the content that could be there from previous page

  fixedHeaderZone.innerHTML = '';
  fixedFooterZone.innerHTML = '';

  // If this template has fixed components place them in the fixed zones

  if (fixedComponents) {

    fixedComponents.forEach(function(component) {

      document.getElementById(component.zone).appendChild(component.node);

    });

  }

}

```

```
}
```

4. Adjust the styles. As you are adding new zones in the page layout, you will probably have to adjust the zone for the main node. Use the css variables to adjust the template's zones.

Invalid routes management

When a user tries to access an invalid route, we can define the desired behavior in the `app.js` of the application.

Either redirect to another page, or show a custom 404 (not found) page. In the latest case, we use the reserved route '404'.

1. [Redirect to an existing page](#)
2. [Custom 404 page](#)

Redirect to an existing page

Assuming that we have the following pages defined in the `app.js`:

```
window.addEventListener('componentsLoaded', function() {  
  new window.CellsPolymerBridge({  
    ...  
    routes: {  
      'login': '/',  
      'dashboard': '/dashboard',  
      'account': '/account/:accountId'  
    },  
    ...  
  });  
  ...  
});
```

We could define a redirection to 'dashboard' page when the user tries to access an invalid route:

```

window.addEventListener('componentsLoaded', function() {
  new window.CellsPolymerBridge({
    ...
    routes: {
      'login': '/',
      'dashboard': '/dashboard',
      'account': '/account/:accountId',
      '404': '/dashboard'
    },
    ...
  });
  ...
});

```

Custom 404 page

Assuming that we have the following pages defined in the `app.js`:

```

window.addEventListener('componentsLoaded', function() {
  new window.CellsPolymerBridge({
    ...
    routes: {
      'login': '/',
      'dashboard': '/dashboard',
      'account': '/account/:accountId'
    },
    ...
  });

```

```
...  
});
```

We could define a custom NOT FOUND page when the user tries to access an invalid route:

```
window.addEventListener('componentsLoaded', function() {  
  new window.CellsPolymerBridge({  
    ...  
    routes: {  
      'login': '/',  
      'dashboard': '/dashboard',  
      'account': '/account/:accountId',  
      '404': '/not-found'  
    },  
    ...  
  });  
  ...  
});
```

The 'not-found' page is defined along with the other pages (app/composerMocksTpl/404.js) as follows:

```
module.exports = CONFIG => {  
  const PAGES = require('./_pages');  
  
  let _404JSON = {  
    pages: PAGES,  
    template: {  
      tag: 'cells-template-zones',
```

```
    properties: {
      disableEdgeSwipe: true,
      scaleContent: true,
      drawerWidth: '100%'
    }
  },
  components: [
    {
      zone: 'app__header',
      type: 'UI',
      tag: 'cells-component-app-header',
      properties: {
        title: 'PAGE NOT FOUND'
      }
    },
    {
      zone: 'app__main',
      type: 'UI',
      tag: 'cells-icon-message',
      properties: {
        type: 'error',
        icon: 'coronita:error',
        'icon-label': null,
        'icon-size': 26,
        message: 'The page you are looking for can\'t be found'
      }
    }
  ]
}
```

```

    },
    {
      zone: 'app__main',
      type: 'UI',
      tag: 'cells-go-dashboard-button',
      familyPath: '../elements/cells-go-dashboard-button',
      properties: {
        cellsConnections: {
          out: {
            _navigate_to_dashboard: {
              bind: 'go-dashboard',
              link: {
                page: 'dashboard'
              }
            }
          }
        }
      }
    }
  ];

  return _404JSON;
};

```

Management of not found page definition files

There may be situations in which the page definition files are not accessible temporarily. In this case, Cells Bridge provides you with a callback mechanism (`onPageDefinitionNotFound`) to handle this situation. You can find it on `app.js`:

```
window.addEventListener('componentsLoaded', function() {  
  ...  
  new window.CellsPolymerBridge({  
    ...  
    routes: {  
      'home': '/',  
      'profile': '/profile'  
    },  
    onRender: function onrender(template) {  
      ...  
    },  
    onPageDefinitionNotFound: function onPageDefinitionNotFound(page) {  
      window.location.href = '/#!';  
    },  
    ...  
  });  
});
```

Type of components

With Cells-Bridge you can manage different types of components. These types are:

- UI (default value): for [user interface components](#)
- DM: for [data managers](#)
- CC: for [cross components](#)

The type of a component is set using the `type` attribute in the definition file:


```
"components": [
```

```
{
```

```
  "zone": "app__main",
```

```
  "familyPath": "../elements/an-ui-component",
```

```
  "tag": "an-ui-component"
```

```
},
```

```
{
```

```
  "zone": "app__main",
```

```
  "type": "UI",
```

```
  "familyPath": "../elements/another-ui-component-with-explicit-type",
```

```
  "tag": "another-ui-component-with-explicit-type"
```

```
},
```

```
{
```

```
  "zone": "app__main",
```

```
  "type": "DM",
```

```
  "familyPath": "../elements/my-data-manager",
```

```
  "tag": "my-data-manager"
```

```
},
```

```
{
```

```
  "zone": "app__main",
```

```
  "type": "CC",
```

```
  "familyPath": "../elements/my-cross-component",
```

```
  "tag": "my-cross-component"
```

```
}
```

```
]
```

Static page definition / Routable components

Since version 3.3.0 of Cells Bridge and 2.1.0 of Cells Cli, you can build your applications through the paradigm of 'routable components' or 'static pages': the route will be rendered using the respective matching component definition (with their own scope, life cycle, data binding - everything that is provided by default with Polymer).

This architecture paradigm will fit in applications where you don't need to have a complex/multi environment configuration, simple applications, etc...

This approach will allow to boost your developing speed - and every feature from 'Cells' architecture will be available for you! (private channels, cells connections, etc...)

Moreover, we have built a generic Polymer component called 'cells-page' that allows you to easily build pages that interact with Cells architecture through a friendly API.

Appendix

1. [Configuration & Convention](#)
2. [JSON & Static page definition coexistence](#)
3. [Mapping route parameters to page component properties](#)
4. [API](#)
5. [HOOKS](#)

1. Configuration & Convention

Initially, static pages must follow a few rules:

Let's suppose we have the following routes defined in our application bootstrap (app.js):

```
routes: {  
  'login': '/login',  
  'dashboard': '/dashboard',  
  'report': '/report',  
  'customer': '/customer',  
  'movement-detail': '/movement-detail'  
}
```

You must consider:

1. You must mark which routes are going to be rendered as `routable` components or `static` pages on environment configuration files, inside `pages` property.

`app/config/local.json`

```
{  
  ...,  
  "pages": [ "login", "dashboard", "report", "customer", "movement-detail"  
],  
  ...  
}
```

2. Each route name should match a component definition name with the following convention: `<name>-page`.

```
'login'          -> 'login-page'  
'dashboard'      -> 'dashboard-page'  
'report'         -> 'report-page'  
'customer'       -> 'customer-page'  
'movement-detail' -> 'movement-detail-page'
```

3. Each component definition file must be placed inside `app/pages`, following the convention:

```
/app  
|  
|__ /pages  
|   |____ /login-page  
|       |_....  
|       |_....  
|       |__ login-page.html  
|  
|   |____ /dashboard-page  
|       |_....  
|       |_....  
|       |__ dashboard-page.html  
|  
|   |____ /report-page  
|       |_....  
|       |_....
```

```

|      |__report-page.html
|
|_____/customer-page
|      |_....
|      |_....
|      |__customer-page.html
|
|_____/movement-detail-page
|      |_....
|      |_....
|      |__movement-detail-page.html

```

NOTE: Default page component folder will be 'app/pages', but it can be overridden through defining "pagesPath" property inside configuration file.

app/config/local.json

```

{
  ...,
  "pages": [ "login", "dashboard", "report", "customer", "movement-
detail" ],
  "pagesPath": "../my-pages-component-folder/",
  ...
}

```

Resulting in:

```

/app
|
|__../my-pages-component-folder/
|   |_____/login-page
|   |      |_....
|   |      |_....
|   |      |__login-page.html
|
|   |_____/dashboard-page
|   |      |_....
|   |      |_....
|   |      |__dashboard-page.html
|

```

```

|_____ /report-page
|         |_....
|         |_....
|         |__report-page.html
|
|_____ /customer-page
|         |_....
|         |_....
|         |__customer-page.html
|
|_____ /movement-detail-page
|         |_....
|         |_....
|         |__movement-detail-page.html

```

IMPORTANT: Component definition template must include as a first level component, one of the cells-templates availables on Cells Component Catalog. It's a constrain (at the moment), and is required by Cells render & animation engine. You must use defined 'slots' in each template for place components inside them (as normal JSON page definition files).

app/pages/my-page/my-page.html

```
<dom-module id="my-page">
```

```
<template>
```

```
<cells-template-paper-header-panel mode="seamed"> <!-- required -->
```

```
<div slot="app__header">
```

```
...
```

```
</div>
```

```
<div slot="app__main">
```

```
...
```

```
</div>
```

```
</cells-template-paper-header-panel>
```

```
</template>
```

```
<script src="my-page.js"></script>
```

```
</dom-module>
```

2. JSON & Static page definition coexistence

You can mix dynamic & static page definition routes on the same application.

Routes included in 'pages' property on environment configuration files have more weight than normal ones, so if you have same page defined as JSON & static page, it will be gathered and rendered as a static page / routable component.

3. Mapping route parameters to page component properties

When you navigate to a route that has parameters defined in its URI, you can retrieve them defining a property called **'params'** on the target component.

```
params: {
```

```
  type: Object,
```

```
  value: {}
```

```
}
```

This property will be filled with all params defined in the URI, and will be available for use before private channels are fired as a normal component property, so you can do things like computed properties for example.

Assuming we have the following configuration & route definition:

app/scripts/app.js

```
routes: {
```

```
  ...,
```

```
  'my-page': '/my-page/:customer-name',
```

```
  ...
```

```
}
```

app/config/local.json

```

{
  ...,
  "pages": [ "my-page" ],
  "pagesPath": "./pages/",
  ...
}

```

app/pages/my-page/my-page.html

```

<dom-module id="my-page">
  <template>
    <cells-template-paper-header-panel mode="seamed">
      <div slot="app__header">
        <cells-component-app-header
          text=[[headerTitle]]>
        </cells-component-app-header>
      </div>

      <div slot="app__main">
        ...
      </div>
    </cells-template-paper-header-panel>
  </template>

  <script src="my-page.js"></script>
</dom-module>

```

app/pages/my-page/my-page.js

```
{
```

```
const CellsPage = customElements.get('cells-page');
```

```
class MyPage extends CellsPage {
```

```
static get is() {
```

```
return 'my-page';
```

```
}
```

```
static get properties() {
```

```
return {
```

```
// NOT NEEDED IF YOU INHERIT FROM CellsPage
```

```
params: {
```

```
type: Object,
```

```
value: {}
```

```
},
```

```
headerTitle: {
```

```
type: String,
```

```
computed: '_computeHeaderTitle(params)'
```

```
}
```

```
};
```

```
}
```

```
_computeHeaderTitle(params) {
```

```
const customerName = params['customer-name'] || '';
```

```
return decodeURI(customerName).replace(/-/g, ' ');
```

```
}
```



```
}
```

```
window.customElements.define(MyPage.is, MyPage);
```

```
}
```

IMPORTANT: If you extend from CellsPage component class, you don't need to define params property as it's already defined for you.

4. API

The interaction between your pages (components), and Cells Bridge, will be available through the inheritance of generic component called [CellsPage](#), and allows you to:

- subscribe(channelName, callback)

Subscribe to given bridge channel and executes given callback when it reacts to state change.

```
{
```

```
const CellsPage = customElements.get('cells-page');
```

```
class MyPage extends CellsPage {
```

```
static get is() {
```

```
return 'my-page';
```

```
}
```

```
static get properties() {
```

```
return { };
```

```
}
```

```
ready() {
```

```
super.ready();
```

```
this._handleConnections();
```

```
}
```

```
_handleConnections() {
```

```
// subscribe to customer channel and executes
```

```
_updateCustomerData with given channel value when channel state  
mutates.
```

```
this.subscribe('customer', customer =>
```

```
this._updateCustomerData(customer));
```

```
}
```

```
_updateCustomerData(customer) {
```

```
const items = this._normalizeCustomerData(customer);
```

```
this.set('items', items);
```

```
}
```

```
window.customElements.define(MyPage.is, MyPage);
```

```
}
```

- `publishOn(channelName, htmlElement, eventName)`

Automatically publish on given channelName the payload from event of type eventName dispatched by htmlElement .

```
{
```

```
const CellsPage = customElements.get('cells-page');
```

```
class MyPage extends CellsPage {
```

```
  static get is() {
```

```
    return 'my-page';
```

```
  }
```

```
  static get properties() {
```

```
    return { };
```

```
  }
```

```
  ready() {
```

```
    super.ready();
```

```
    this._handleConnections();
```

```
  }
```

```
  _handleConnections() {
```

```
    // customerDM will be a component defined on local DOM that will  
    dispatch a custom event called 'customer-loaded'.
```

```
    // 'customer-loaded' event payload will be stored in 'customer'  
    bridge channel.
```

```
    this.publishOn('customer', this.$.customerDM, 'customer-loaded');
```

```
  }
```

```
}
```

```
window.customElements.define(MyPage.is, MyPage);
```

```
}
```

- `publish(channelName, value)`

Publish a value to given channel.

- `navigate(page, params)`

Navigation to another route. Params are optional and are given as an object with key-value format.

NOTE: Remember to inherit from `CellsPage` component in all your page definition components to get previous features out of the box:

```
{
```

```
const CellsPage = customElements.get('cells-page');
```

```
class MyPage extends CellsPage {
```

```
  static get is() {
```

```
    return 'my-page';
```

```
  }
```

```
  static get properties() {
```

```
    return { };
```

```
  }
```

```
}
```

```
window.customElements.define(MyPage.is, MyPage);
```

```
}
```

5. HOOKS

`CellsPage` component provides 2 hooks that facilitates the implementation of custom logic that is a common requirement on pages/routes. If any route rendered by a routable

component has any of the following implementations, they will be automatically executed in its respective step.

- **onPageEnter**

Executed each time before transitioning to the route.

```
{  
  
  const CellsPage = customElements.get('cells-page');  
  
  class MyPage extends CellsPage {  
  
    static get is() {  
  
      return 'my-page';  
  
    }  
  
    static get properties() {  
  
      return { };  
  
    }  
  
    onPageEnter() {  
  
      // my custom logic to be executed each time before transition  
      to the route.  
  
    }  
  
  }  
  
  window.customElements.define(MyPage.is, MyPage);  
  
}
```

- **onPageLeave**

Executed each time after the route has been left.

```
{
```

```
const CellsPage = customElements.get('cells-page');
```

```
class MyPage extends CellsPage {
```

```
  static get is() {
```

```
    return 'my-page';
```

```
  }
```

```
  static get properties() {
```

```
    return { };
```

```
  }
```

```
  onPageLeave() {
```

```
    // my custom logic to be executed each time after the route has  
    been leave.
```

```
  }
```

```
}
```

```
window.customElements.define(MyPage.is, MyPage);
```

```
}
```

Connect components

Binding between components

1. [One way data binding](#)
2. [Two way data binding](#)
3. [Properties with methods](#)
4. [Properties with events](#)
5. [Events with methods](#)
6. [Components in different pages](#)
7. [To a non declared property](#)
8. [Private channels](#)
9. [Application Context channel](#)

Cells Bridge binding type

When Cells loads a template it also loads its components, in that moment it registers the components in the channel.

Depending on the moment you want this registration to happen, Cells Bridge offers the **'binding'** type to define that moment. For example, you may want to register the components as soon as they are loading (even when the page is not fully loaded yet) or in other cases you may need to wait until the page is ready to interact, to register the components -the page is ready when it has all its components loaded and is visible.

The available types of binding are:

- `always`: Register all components of all views. Never unregister them.
- `delayed`: Like 'always' but waits for all components to be loaded.
- `ui`: Register only ui and cross components of all views. Never unregister them. Datamanagers are only connected when the current page is ready.
- `currentview`: Register all components of the current view when the page is ready.

The default value is `always` but you can change the binding type in the `app.js` file, when you're creating the Cells bridge instance.

```
window.addEventListener('componentsLoaded', function () {  
  new window.CellsPolymerBridge({  
    binding : 'currentview',  
    ...  
  });  
});
```

```
});
```

One way data binding (from property to property)

As stated before Cells allows you to configure the communication between components. We are taking as example the home page, defined in `app/composerMocksTpl/home.json`. We are going to include another `my-greeting` component and bind the first with the second, that is, when you change the first it will cause a change in the second.

In the first component we define a *cells connection out*, using the channel `"component1-channel"` and bind the action with the event `greeting-changed` fired because `greeting` property is notifiable (`notify: true`). You only need set in bind property of action out the name of the property with suffixed `'-changed'`. In the second component we define a *cells connection in* with the same channel (`"component1-channel"`) and we bind it with the property `greeting`.

```
"components": [  
  {  
    "zone": "app__main",  
    "familyPath": "../elements/my-greeting",  
    "tag": "my-greeting",  
    "properties": {  
      "cellsConnections": {  
        "out": {  
          "component1_channel": {  
            "bind": "greeting-changed"  
          }  
        }  
      }  
    },  
    {  
      "zone": "app__main",
```



```

    "familyPath": "../elements/my-greeting",
    "tag": "my-greeting",
    "properties": {
      "cellsConnections": {
        "in": {
          "component1_channel": {
            "bind": "greeting"
          }
        }
      }
    }
  }
}
]

```

Two way data binding (from property to property)

If you want to set two way data binding, you can create a second connection doing the same but in the opposite way. We use different channels only to show the data flow, but you can set the same channel and it won't provoke an infinite loop.

```

"components": [
  {
    "zone": "app__main",
    "familyPath": "../elements/my-greeting",
    "tag": "my-greeting",
    "properties": {
      "cellsConnections": {
        "in": {
          "component2_channel": {

```

```
      "bind": "greeting"
    }
  },
  "out": {
    "component1_channel": {
      "bind": "greeting-changed"
    }
  }
}
},
{
  "zone": "app__main",
  "familyPath": "../elements/my-greeting",
  "tag": "my-greeting",
  "properties": {
    "cellsConnections": {
      "in": {
        "component1_channel": {
          "bind": "greeting"
        }
      },
      "out": {
        "component2_channel": {
          "bind": "greeting-changed"
        }
      }
    }
  }
}
```

```

    }
  }
}
}
]

```

From property to method

We are going to create a method to listen a greet from outside.

```

listenGreet: function (greet) {
  this.set('greeting', 'Thanks for say: ' + greet + '');
}

```

Now we only have to bind the action in with this method like this:

```

"components": [
  {
    "zone": "app__main",
    "familyPath": "../elements/my-greeting",
    "tag": "my-greeting",
    "properties": {
      "cellsConnections": {
        "in": {
          "component2_channel": {
            "bind": "listenGreet"
          }
        }
      },
      "out": {
        "component1_channel": {

```

```
    "bind": "greeting-changed"
  }
}
}
}
},
{
  "zone": "app__main",
  "familyPath": "../elements/my-greeting",
  "tag": "my-greeting",
  "properties": {
    "cellsConnections": {
      "in": {
        "component1_channel": {
          "bind": "listenGreet"
        }
      },
      "out": {
        "component2_channel": {
          "bind": "greeting-changed"
        }
      }
    }
  }
}
```

From event to property

To make it possible we have to change a bit `my-greeting` component to add the event we want to listen. Open the element `app/elements/my-greeting/my-greeting.html` and include a button that fires the event.

In the HTML code we add the button

```
<button id="greet">Greet</button>
```

And we set the listener and callback method in the JavaScript code

```
listeners: {
```

```
  'greet.tap': 'greet'
```

```
},
```

```
greet: function() {
```

```
  this.fire('greet', this.greeting);
```

```
}
```

Once you have made the changes, go back to the configuration file and change the connections between components to reflect the change only when the event is fired instead of when the property change.

```
"components": [
```

```
{
```

```
  "zone": "app__main",
```

```
  "familyPath": "../elements/my-greeting",
```

```
  "tag": "my-greeting",
```

```
  "properties": {
```

```
    "cellsConnections": {
```

```
      "in": {
```

```
        "component2_channel": {
```

```
        "bind": "greeting"
    }
},
    "out": {
        "component1_channel": {
            "bind": "greet"
        }
    }
}
},
{
    "zone": "app__main",
    "familyPath": "../elements/my-greeting",
    "tag": "my-greeting",
    "properties": {
        "cellsConnections": {
            "in": {
                "component1_channel": {
                    "bind": "greeting"
                }
            },
            "out": {
                "component2_channel": {
                    "bind": "greet"
                }
            }
        }
    }
}
```

```

    }
  }
}
]

```

From event to method

Now we are going to connect an event with the method of other component. Again, we have to change the component `my-greeting` to include the method that will receive the event callback.

```

listenGreet: function(payload) {
  this.set('greeting', payload);
}

```

After that, we have to define the new connection in the page layout

```

"components": [
  {
    "zone": "app__main",
    "familyPath": "../elements/my-greeting",
    "tag": "my-greeting",
    "properties": {
      "cellsConnections": {
        "in": {
          "component2_channel": {
            "bind": "listenGreet"
          }
        },
        "out": {

```

```
    "component1_channel": {  
      "bind": "greet"  
    }  
  }  
}  
},  
{  
  "zone": "app__main",  
  "familyPath": "../elements/my-greeting",  
  "tag": "my-greeting",  
  "properties": {  
    "cellsConnections": {  
      "in": {  
        "component1_channel": {  
          "bind": "listenGreet"  
        }  
      },  
      "out": {  
        "component2_channel": {  
          "bind": "greet"  
        }  
      }  
    }  
  }  
}
```



```
]
```

Cells Bridge provides at this moment a callback to be executed when the template is rendered.

Other examples:

```
// Binding an event with an Array as payload to the updateMovements method
```

```
updateMovements: function(data) {
```

```
  this.set('movementsList', data);
```

```
  this.set('moreMovements', data);
```

```
}
```

And another example assuming that the payload of the event is the following object:

```
{
```

```
  "isNew": true,
```

```
  "title": "a new title",
```

```
  "message": "Some message"
```

```
}
```

```
// Binding to writeInfo method
```

```
writeInfo: function(data) {
```

```
  if (data.isNew) {
```

```
    this.title = "NEW " + data.title;
```

```
  } else {
```

```
    this.title = data.title;
```

```
  }
```

```
  this.message = data.message;
```

```
}
```

Avoiding repetition of method invocation

Suppose you have a component with an in-connection binded to a method. Someone puts a value `x` in the channel causing the invocation of the method. Then the page goes away so the component is detached from the channel. When the page enters again, the component is re-attached to the channel and as the channel still has the value `x` it causes the invocation of the method one more time. In some cases you won't want this invocation to happen again. This can be avoided by declaring that the binding don't accept previous state that has been handled before. It's done by setting the property `previousState` to `false`.

```
"components": [  
  {  
    "zone": "app__main",  
    "tag": "hello-greeting",  
    "properties": {  
      "cellsConnections": {  
        "in": {  
          "channel": {  
            "bind": "hello-changed"  
            "previousState": false  
          }  
        }  
      }  
    }  
  },  
]
```

Components in different pages

Channels can be used to connect components that are in different pages. Just define the connections the same way you do with components in the same page.

Suppose you have component `hello-greeting` in page `wellcome` and you want the value in `hello-greeting` to go to another component, `bye-greeting` in page `goodbye`.

`wellcome.json`

```
"components": [  
  {  
    "zone": "app__main",  
    "tag": "hello-greeting",  
    "properties": {  
      "cellsConnections": {  
        "out": {  
          "interpage-channel": {  
            "bind": "hello-changed"  
          }  
        }  
      }  
    }  
  },  
]
```

goodbye.json

```
"components": [  
  {  
    "zone": "app__main",  
    "tag": "bye-greeting",  
    "properties": {  
      "cellsConnections": {  
        "in": {  
          "interpage-channel": {  
            "bind": "bye"  
          }  
        }  
      }  
    }  
  }  
]
```

```

    }
  }
}
},

```

Connect to a non declared property

Cells Bridge detects the kind of action in binding including the behaviors, but if you set a bind that is not defined in the web component, this will be set to the node element.

```

"components": [
  {
    "zone": "app__main",
    "familyPath": "../elements/my-greeting",
    "tag": "my-greeting",
    "properties": {
      "id": "myGreeting1",
      "cellsConnections": {
        "in": {
          "channel": {
            "bind": "noDeclaredProperty"
          }
        }
      }
    }
  },
  {
    "zone": "app__main",
    "familyPath": "../elements/my-greeting",

```

```

    "tag": "my-greeting",
    "properties": {
      "id": "myGreeting2",
      "cellsConnections": {
        "out": {
          "channel": {
            "bind": "greet"
          }
        }
      }
    }
  }
}
]

```

In this case the first element will have a new property in the HTML element called `noDeclaredProperty` and you can get it with:

```

var myGreeting1 = document.getElementById('myGreeting1');
var myGreeting2 = document.getElementById('myGreeting2');

console.log(myGreeting1.noDeclaredProperty); // undefined

myGreeting2.greet('Hello');

console.log(myGreeting1.noDeclaredProperty); // Hello

```

Private channels

Whenever a page is activated, Cells creates a private channel for it and it names it following this pattern: `__bridge_page_` + `pageName`. For example, given a page named `home`, Cells will create a private channel named `__bridge_page_home`. Thus you should not define a channel with a name that starts with `__bridge_page_`.

The private channel of a page is created the first time the page is loaded and it remains active the entire lifecycle of the application. Also, private channels are **read-only**, so you can only subscribe to them with *actions "in"*.

A private channel it's used to indicate if the page is active or not. The components of the application can know if a page is active just by reading the private channel of the page using a *Cells connection in*.

```
"components": [  
  {  
    "zone": "app__main",  
    "familyPath": "../elements/my-greeting",  
    "tag": "my-greeting",  
    "properties": {  
      "id": "myGreeting1",  
      "cellsConnections": {  
        "in": {  
          "__bridge_page_home": {  
            "bind": "homeStatus"  
          }  
        }  
      },  
      ...  
    }  
  ]
```

Application Context channel

When a Cells application starts, Cells creates a special channel with the name `__bridge_app`. This is a dedicated channel to keep track of the state of the application. The state is an object with the following information:

- *currentPage* the name of the page that is active.
- *fromPage* the name of the previous active page.

The application context channel remains active during the entire lifecycle of the application. Also, private channels are **read-only**, so you can only subscribe to them with *actions "in"*. Finally, you shouldn't create a channel that starts with `__bridge_app`.

With every navigation, the state is updated and published in the application context channel. Any component that needs to be notified about these changes can subscribe to the channel `__bridge_app`.

```

"components": [
  {
    "zone": "app__main",
    "familyPath": "../elements/my-greeting",
    "tag": "my-greeting",
    "properties": {
      "id": "myGreeting1",
      "cellsConnections": {
        "in": {
          "__bridge_app": {
            "bind": "listenAppContextChannel"
          }
        }
      }
    }
  },
  ...
]

<dom-module id="example-cross-component">

```

```
<template>
```

```
<style>
```

```
:host {
```

```
display: block;
```

```
}
```

```
</style>
```

```
<p>Current page: [[currentPage]] </p>
```

```
<p>Previous page: [[previousPage]] </p>
```

```
</template>
```

```
<script>
```

```
Polymer({
```

```
is: 'example-cross-component',
```

```
properties: {
```

```
previousPage : { type: String },
```

```
currentPage : { type: String }
```

```
},
```

```
listenAppContextChannel: function(data) {
```

```
this.previousPage = data.value.fromPage;
```

```
this.currentPage = data.value.currentPage;
```

```
}
```

```
});
```

```
</script>
```

```
</dom-module>
```


Navigation

You can navigate between pages via establishing an `out` action inside the configuration. This action can accept parameters in the url.

Simple navigation

An example about how to navigate between pages in a simple way:

```
{
  "template": {
    "familyPath": "../../components/cells-template-empty",
    "tag": "cells-template-empty"
  },
  "components": [
    {
      "zone": "app__main",
      "familyPath": "../elements/my-greeting",
      "tag": "my-greeting",
      "properties": {
        "cellsConnections": {
          "in": {},
          "out": {
            "foobar-channel": {
              "bind": "foobar-event",
              "link" : {
                "page" : "another"
              }
            }
          }
        }
      }
    }
  ]
}
```

```

    }
  }
}
]
}

this.fire('foobar-event');

```

In the example above, every time the component `my-greeting` fires the event `foobar-event`, the application will go to the page named `another`.

Navigate with params

An example about how to navigate between pages, passing a parameter to the url:

```

{
  "template": {
    "familyPath": "../components/cells-template-empty",
    "tag": "cells-template-empty"
  },
  "components": [
    {
      "zone": "app__main",
      "familyPath": "../elements/my-greeting",
      "tag": "my-greeting",
      "properties": {
        "cellsConnections": {
          "in": {},
          "out": {

```

```

        "foobar-channel": {
          "bind": "foobar-event",
          "link" : {
            "page" : "another",
            "params": {
              "eventId": "itemId"
            }
          }
        }
      }
    }
  }
}

this.fire('foobar-event', { 'eventId': 1234 });

```

In the example above, every time the component `my-greeting` fires the event `foobar-event`, the application will go to a page named `another`, passing a parameter named `itemId` to the url: `#!/another?itemId=1234`

Get params from the URL

If we want to get a param from the url we have to define it in the *cells connection params* property, with the pattern component's property name as key and url param name as value. Setting the following cells connection param in the first component of the `home.json` we can get the greeting from the url like: <http://localhost/#!/?urlGreeting=Hola>

```

"cellsConnections": {
  "params": {
    "greeting": "urlGreeting"
  }
}

```

```
}
```

```
}
```

Binding properties from events to navigate a specific page

You can bind values from properties of events in out connections to match dynamically navigations to pages.

An example about how to navigate with binding properties from events:

```
this.dispatchEvent(new CustomEvent('event-from-component', {  
  bubbles: true,  
  composed: true,  
  detail: {  
    'foo': 'homePage'  
  }  
}));  
  
"components": [  
  {  
    "zone": "app__main",  
    "familyPath": "../elements/my-component",  
    "tag": "my-component",  
    "properties": {  
      "cellsConnections": {  
        "in": {},  
        "out": {  
          "channel-example": {  
            "bind": "event-from-component",  
            "link" : {  
              "page" : {
```

```
    "bind": "foo"
  }
}
}
}
}
}
}
}
}
```

In this example, my-component dispatch a event with one property in the payload ('foo') that have the name of one page ('homePage') of your application. In your JSON definition file, you can bind the value of the property from event to indicate the name of the page to navigate.

Customization service

Remember that files inside the folder `app/composerMocks` are only temporal and they are a mock of the real files generated by the Customization Service (Composer).

Callbacks

Cells Bridge provide, at this moment, a callback (`onRender`) to be executed when the template has been rendered.

Example

```
var options = {  
  cache: false,  
  componentsPath: '../elements/',  
  generateRequestUrl: function(page) {  
    return '../mocks/section-' + page + '.json'  
  },  
  routes: {  
    'home': '/'  
  },  
  onRender: function(template) {  
    // area to customize  
  }  
};  
  
new CellsPolymerBridge(options);
```

Cells in two minutes

Here you have a short video about how cells works with the customization service.

<http://youtu.be/XFvyj0tEbP8>

Hooks

*** DEPRECATED ***

Hooks **as** events are deprecated. You can **get** the same result **using** the **new** **event** channels.

Cells Bridge provide some hooks to run your own code in these points. These hooks are available as events that you can listen.

These events are attached to the main node chosen for your app.

These events are:

- page-ready
- parse-route
- after-publish
- nav-request
- before-set-attr-to-node
- after-set-attr-to-node
- before-create-node
- after-create-node
- before-import
- after-import
- page-request
- page-response
- data-load
- template-transition-end
- template-registered

Usage

1. The first thing you must indicate in the Cells Bridge configuration the main node where the application will be rendered. You can find the initialization of Cells Bridge in the `app.js` file

```
2. new window.CellsPolymerBridge({  
3.   mainNode: 'app__content',  
4.   ...  
5. });
```

6. The main node refers to the html container that you must define in the `index.html`

```
7. <div id="app__content"></div>
```

8. You can listen then the events defined as hooks on that node. For example `'page-ready'`

```
9. document.getElementById('app__content').addEventListener('page-  
ready', function(e) {
```

```
10. // Do something
```

```
11. });
```

page-ready

When a page is ready to use.

parse-route

When a route is parsed.

The hook sends the route object.

after-publish

After register an event.

The hook sends the event object.

nav-request

When there is a navigation request.

The hook will send:

```
{
```

```
  event: event,
```

```
  detail: connection.link
```

```
}
```


before-set-attr-to-node

Before set an attribute to a component.

The hook sends the tagname of the component.

after-set-attr-to-node

After set an attribute to a component.

The hook sends the component object.

before-create-node

Before create a component.

The hook sends the tagname of the component.

after-create-node

After create a component.

The hook sends the component object.

before-import

Before import a component.

after-import

After import a component.

page-request

Before request a page to the connector.

The hook sends the options of the request.

page-response

After the request of the page.

The hook sends the response object.

data-load

When the data have been loaded.

The hook sends the sanitized object.

template-transition-end

When a transition ends.

The hook sends the template object.

template-registered

When all components in the template have been connected to channels.

Event Channels (Hooks)

Cells Bridge provide some hooks to run your own code in these points. Hooks are available through event channels. Whenever an internal event is fired, Cells Bridge publish the event in its own channel.

These events are:

- `page-ready`
- `parse-route`
- `after-publish`
- `nav-request`
- `before-set-attr-to-node`
- `after-set-attr-to-node`
- `before-create-node`
- `after-create-node`
- `before-import`
- `after-import`
- `page-request`
- `page-response`
- `data-load`
- `template-transition-end`
- `teplate-registered`

Every event has its own channel:

- `__bridge_evt_page-ready`
- `__bridge_evt_parse-route`
- `__bridge_evt_after-publish`
- `__bridge_evt_nav-request`
- `__bridge_evt_before-set-attr-to-node`
- `__bridge_evt_after-set-attr-to-node`
- `__bridge_evt_before-create-node`
- `__bridge_evt_after-create-node`
- `__bridge_evt_before-import`
- `__bridge_evt_after-import`
- `__bridge_evt_page-request`
- `__bridge_evt_page-response`
- `__bridge_evt_data-load`
- `__bridge_evt_template-transition-end`
- `__bridge_evt_teplate-registered`

Usage

There are three ways to subscribe to event channels:

1. By configuration in the Cells Bridge.

Cells Bridge has the `eventSubscriptions` property which accepts an array with objects that must have the properties: `event` and `callback`.

event must be the name of one of the special events listed above.

`callback` must be function that will be invoked when the event is published in the channel.

In the `app.js` file:

```
new window.CellsPolymerBridge({
  mainNode: 'app__content',
  eventSubscriptions: [
    {event : 'page-ready', callback : function(ev) { console.log('page
ready ', ev);} },
    {event : 'nav-request', callback : function(ev) { console.log('nav
request ', ev);} }
  ],

```

2. Programatically.

The Cells Bridge object: `$score[0]` exposes the method `subscribeToEvent(eventName, callback)`

`eventName` is the name of the event to subscribe.

`callback` is the function to call when the event channel is activated with a new value.

```
$score[0].subscribeToEvent('page-ready', function(ev) {  
    console.log('page ready ', ev);  
})
```

3. Declarative subscription.

Subscribe to the channel as you normally do with other channels, in the definition json file of your page.

```
"components": [  
  {  
    "zone": "app__main",  
    "tag": "my-greeting",  
    "properties": {  
      "cellsConnections": {  
        "in": {  
          "__bridge_evt_page-ready": {  
            "bind": "listenGreet"  
          }  
        }  
      }  
    }  
  }  
]
```

page-ready

When a page is ready to use.

parse-route

When a route is parsed.

The hook sends the route object.

after-publish

After register an event.

The hook sends the event object.

nav-request

When there is a navigation request.

The hook will send:

```
{  
  event: event,  
  detail: connection.link  
}
```

before-set-attr-to-node

Before set an attribute to a component.

The hook sends the tagname of the component.

after-set-attr-to-node

After set an attribute to a component.

The hook sends the component object.

before-create-node

Before create a component.

The hook sends the tagname of the component.

after-create-node

After create a component.

The hook sends the component object.

before-import

Before import a component.

after-import

After import a component.

page-request

Before request a page to the connector.

The hook sends the options of the request.

page-response

After the request of the page.

The hook sends the response object.

data-load

When the data have been loaded.

The hook sends the sanitized object.

template-transition-end

When a transition ends.

The hook sends the template object.

template-registered

When all components in the template have been connected to channels.

Manual integration

Use and config Cells Bridge manually

The first is to import Cells-Bridge library in your project. Currently there is only implementation of Polymer (cells-polymer-bridge.html).

If you'd like to use bower, it's as easy as:

```
$ bower install ssh://globaldevtools.bbva.com:7999/cel/cells-polymer-bridge.git
```

Import library in your project:

```
<!-- Import cells-polymer-bridge -->  
  
<link rel="import" href="cells-polymer-bridge/cells-polymer-bridge.html">
```

Config and init Cells-Bridge:

```
window.addEventListener('componentsLoaded', function() {  
  new window.cells.bridge.Bridge({  
    mainNode: 'app__content',  
    componentsPath: window.AppConfig.componentsPath,  
    templatesPath: window.AppConfig.composerEndpoint,  
    routes: {  
      'home': '/'  
    },  
    onRender: function onrender(template) {  
      if (!template.parentNode) {  
        document.getElementById('app__content').appendChild(template);  
      }  
    }  
  })  
})
```



```
});
```

```
});
```

ES6 Support

Enable ES6 support

You can easily enable ES6 support, by enabling an step in your application building process that transpiles all your elements' scripts from ES6 to ES5. Modify the JSON file config file (located in `app/config`), and add these two lines at the end of JSON:

```
{
```

```
...
```

```
"transpile": true,
```

```
"transpileExclude": [ "moment" ]
```

```
}
```

The `transpileExclude` option is recommended to use in order to speed up the transpile process. You must add here all libraries being used in your project.

Modify your elements' scripts

Open one of your elements' scripts and try it. Here's an example:

```
_onClickButton: function(evt) {
```

```
  let itemId = `item-${evt.data.id}`
```

```
  let array = evt.data.array.map( data => `data-${data}`)
```

```
  this.fire('button-clicked', { itemId, array });
```

```
}
```

And build/serve it. Here's the result:

```
_onClickButton: function _onClickButton(evt) {
```

```
var itemId = 'item-' + evt.data.id;

var array = evt.data.array.map(function (data) {

  return 'data-' + data;

});

this.fire('button-clicked', { itemId: itemId, array: array });

}
```

This build process will split Polymer components in two files: .html and .js

To master ES6 try this <http://es6katas.org/>

Components

Templates

The [templates](#) are the components that distribute other components in a page using **zones**.

Zones

- there can be 'n' amount of zones in a template
- not all the templates have to have the same number of zones
- nor the zones have to be named the same than other templates (although is much handier that they are if they are similar to other templates' zones)
- the order of the components in the [JSON page definition](#) is reflected in the rendered page
- there can be zones for hidden layers

Transitions

- Templates are encharged of in- and out- transitions between pages
- The transition is done with a common behavior: [cells-template-animation-behavior](#)
- The possible transitions of the behavior are: horizontal, vertical and static. Static meaning that the one that is static, it stays in its place regardless of other pages transitioning on top of it.

Personalization

You can customize the transitions using CSS custom properties of the [behavior](#)

```
:root {  
  --cells-template-animation-duration: 600ms;  
  --cells-template-animation-timing-function: cubic-bezier(0.66, -0.01,  
0.52, 0.96);  
}
```

States

The states of the template can be: 'cached', 'inactive' and 'active'.

And when the template of the current page becomes active, an event `template-active` is triggered. Except **when the state is 'cached' that the template doesn't trigger any event.**

All this is also included in the [behavior](#)

Catalog

There is a family of templates available in the [Cells Components Catalog](#)

Two components of them based in Polymer components: [paper-header-panel](#) and [paper-drawer-panel](#), use them as reference to build your own

Data Components

- [Data Managers and Data Providers](#)
- [Use of private channels](#)
- [Create a Data Manager](#)
- [Nomenclature](#)
- [Family of data components](#)

Data Managers and Data Providers

A **Data Manager (DM)** is a non-visual component which is responsible for getting data from external services. Generally this is done using **Data Providers (DP)**. The Data Providers encapsulate the communication just as the Data Managers encapsulate the use of Data Providers.

In most cases, you will get data providers generated automatically from its API (more precisely, from the [RAML](#) file).

In case you require a data provider which is not in [Family of data components](#), [ask support](#) for its development and the new data provider will be published ASAP.

In case the automatic generation is not possible, is suggested to use the [cells-generic-dp](#) in order to perform AJAX requests.

```
<cells-generic-dp host="http://bbvalabs.com:8000" native ...></cells-  
generic-dp>
```

Data managers expose a method to request the data that your application may need. Usually the retrieval of the information is done asynchronously and the component will be notifying the state of that request through events (when it starts, when it finishes and if there's an error).

Use of private channels

It's a very common practice to request data as soon as the page is active and when the data has arrived, feed the UI components with that data. Private channels are the way to go with this scenario:

When the page is active it will put a value `true` in its private channel and once it is binded to a method of the DM, the DM can control either the start of a data request or the cleanup of data once the page goes inactive.

JSON file:

```
{
  ...
  "components": [
    {
      "zone": "app__main",
      "type": "DM",
      "tag": "my-data-dm",
      "properties": {
        "host": "http://bbvalabs.com:8000",
        "cellsConnections": {
          "in": {
            "__bridge_page_dashboard": {
              "bind": "willBeActive"
            }
          }
        }
      }
    }
  ]
}
```

Component file:

```
function willBeActive(active) {
```

```

    if (active) {
        this.requestMyData();
    } else {
        this.reset();
    }
}

```

Don't forget!

Once the data request has been made it may take sometime to complete. So meanwhile your page is idle, waiting for the data to arrive, it is recommended that the user would have a visual clue that the app is working.

For this, you can use the events fired by the DM to show that a request is still in progress. For example, when the request starts you show a spinner and when the request ends successfully, you hide the spinner. If there was an error you should also hide the spinner and show an error message.

For example:

```

{
  ...
  "components": [
    {
      "zone": "app__main",
      "type": "DM",
      "tag": "my-data-dm",
      "properties": {
        "host": "http://bbvalabs.com:8000",
        "cellsConnections": {
          "out": {
            "request_start": {

```

```
        "bind": "my-data-request-start"
    },
    "request_ok": {
        "bind": "my-data-request-success"
    },
    "request_error": {
        "bind": "my-data-request-failure"
    }
}

}

}

}

},
{
    "zone": "app__main",
    "type": "UI",
    "tag": "cells-molecule-spinner-veil",
    "properties": {
        "cellsConnections": {
            "in": {
                "request_start": {
                    "bind": "show"
                },
                "request_ok": {
                    "bind": "hide"
                },
                "request_error": {
```

```

        "bind": "hide"
    }
}
}
}
}
}
}
]
}

```

Create a Data Manager

When creating a data manager, have in mind that a DM should:

- have a data provider
- expose at least one method to request data
- fire an event when it starts the request
- fire an event when the request is completed successfully
- fire an event when there is an error
- have a method that receives a Boolean value (which if true, indicates that the DM is active and false, inactive)

```

// expose at least one method to request data

getMyData: function() {

    // have a data provider

    var myDP = new MyDataProvider.get({ host: this.host });

    // fire an event when it starts the request

    this.fire('my-data-request-started');

    myDP.generateRequest()

    .then(function(response) {

        this.data = response.data;

        // fire an event when the request is completed successfully
    })
}

```



```

        this.fire('my-data-request-success');
    }

    }.bind(this))

    .catch(function(error) {

        this.data = null;

        // fire an event when the is an error

        this.fire('my-data-request-failure', error);

    }.bind(this));

},

```

```

// have a method that receives a Boolean value

// (which if true, indicates that the DM is active and false, inactive)

function willBeActive(active) {

    if (active) {

        this.getMyData();

    } else {

        this.reset();

    }

}

}

```

Nomenclature

The nomenclature for the Cells Data Components is that they contain at least the characters `dm` or `dp` accordingly.

For instance: `glomo-accounts-dm`, `dm-global-apis-cards` **or** `cells-dp-enpp-sessions`.

Recommended: `<cells-dm-enax-enpp-login>` **or** `<cells-composer-login-dm>`

Family of data components

The Cells Components Catalog has a [family of data components](#)

Cross components

A Cross component provides a common functionality throughout the application lifecycle and all the pages.

Cells Bridge instantiate Cross components as singletons (a unique instance throughout the whole application). And so, they are available from any page and can communicate with any other component.

Typical needed functionality include:

- Authentication
- Authorization
- Cache storage
- Error handling
- Etc.

They have to be included in every [JSON page definition](#) of the application, indicating the type "type": "CC":

```
{  
  "tag": "cells-offline-notification",  
  "type": "CC"  
}
```

However, each page may define the cellsConnections property according to its needs.

Examples with an offline notification component and a firebase authentication DM:

```
{  
  "tag": "cells-offline-notification",  
  "type": "CC",  
  "properties": {  
    "text1": "SIN CONEXIÓN",  
    "text2": "Conectate a una red de datos",  
    "iconSrc": "images/no-wifi.svg",  
    "image": "images/events-symbol.svg",  
  }  
}
```

```

    "cellsConnections": {
      "in": {
        "connectivity_off_channel": {
          "bind": "show"
        },
        "connectivity_on_channel": {
          "bind": "hide"
        }
      }
    },
    {
      "tag": "google-firebase-signin-dm",
      "type": "CC",
      "properties": {
        "clientIdWeb": "... ..",
        "clientIdCordova" : "... ..",
        "clientSecret" : "... ..",
        "hostedDomain": "bbva.com",
        "authDomain": "##firebaseAuth##",
        "databaseUrl": "##firebaseDb##",
        "apiKey": "##firebaseApiKey##",
        "scopes": "profile https://www.googleapis.com/auth/userinfo.email
https://www.googleapis.com/auth/forms",
        "native": true,

```

```
"cellsConnections": {  
  "in": {  
    "active-signin-request-channel": {  
      "bind": "activeSignIn"  
    },  
    "passive-signin-request-channel": {  
      "bind": "passiveSignIn"  
    },  
    "logout-channel": {  
      "bind": "signOut"  
    }  
  },  
  "out": {  
    "firebase-login-error-channel": {  
      "bind": "firebase-login-error"  
    },  
    "firebase-changed-channel": {  
      "bind": "app-changed"  
    },  
    "firebase-user-changed-channel": {  
      "bind": "computeduser-changed"  
    },  
    "signin-in-progress-channel": {  
      "bind": "signin-in-progress"  
    },  
    "active-signin-success-channel": {
```

```
"bind": "active-signin-success"
```

```
},
```

```
"passive-signin-success-channel": {
```

```
"bind": "passive-signin-success",
```

```
"link" : {
```

```
"page": "events"
```

```
}
```

```
},
```

```
"connectivity_off_channel":{
```

```
"bind":"firebase-disconnected"
```

```
},
```

```
"connectivity_on_channel":{
```

```
"bind":"firebase-connected"
```

```
},
```

```
"google-profile-error-channel":{
```

```
"bind": "google-profile-error"
```

```
}
```

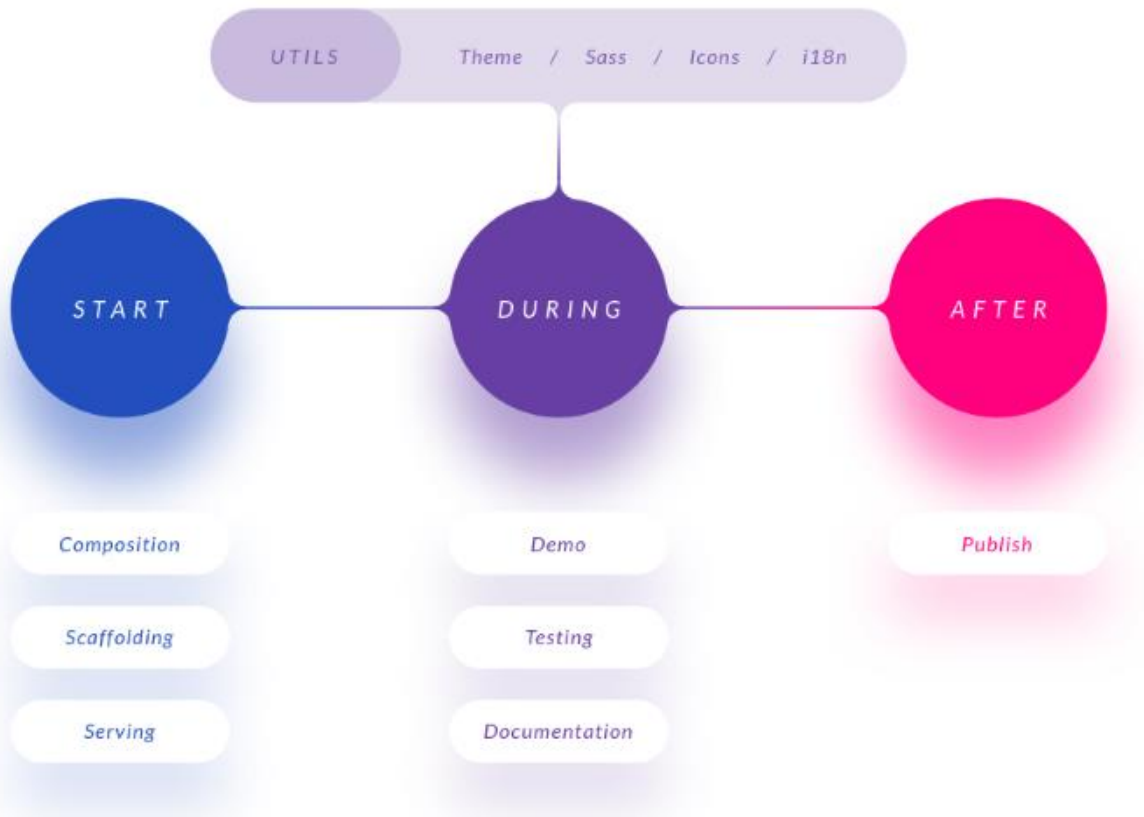
```
}
```

```
}
```

```
}
```

```
},
```

User Interface components



Warming up

A Cells component is a custom element created using [Polymer](#).

Check the [composition guide](#) to understand what level of abstraction and functionality a Cells component should have.

Cells encourages you to send the component to be part of [Cells Components Catalog](#). Read more in the [How to contribute guide](#)

Consider

Is there any component in the [Cells Components Catalog](#) that could fulfill your needs with maybe a minor change? Take a look to the [Contribution guide](#).

Tip If you identify a component that may be electable for use in terms of functionality but not really because of the structure, disposition of elements or that it may have more functionality than needed (for example labels). Take in account that all Cells Components have their styles totally open. So, using the available mixins and variables in [the theme](#) of the project you can evaluate whether is worth making the changes that way rather than making a new component

Start

1. [Scaffolding](#)
2. [Serve a component](#)

During

1. [Demo](#)
2. [Unit testing](#)
3. [Documentation](#)
4. [Utils \(theme, cells-sass, cells-icons, i18n\)](#)

After

1. [Publish](#)

Scaffolding

Name the component and choose one of the scaffolds provided:

```
$ cells
? Write a name for your component cells-component-name
? Which type is your component? (Use arrow keys)
> UI (purely graphical)
  UI (empty)
  DM (data manager)
```

Cells components have domShadow, useNativeCSSProperties and lazyRegister activated.

Output

- cells-component-name.html, it also has the documentation in JsDoc format.
- cells-component-name.js
- cells-component-name.scss, it is compiled at serving time and **generates:** cells-component-name-styles.html.
- bower.json
- README.md
- index.html, it uses [iron-component-page](#) to visualize the component documentation and demo. It doesn't need to be modified.
- .editorconfig

- .gitignore
- /demo/
 - index.html
 - /js/demo.js
 - /css/demo-styles.html
- /test/
- /locales/: (IF i18n) Contains the translations in JSON files for at least english(en.json) and spanish(es.json).

Serving

cells component:serve is a command that allows you to serve the component locally:

```
/your-component/$ cells
```

```
/your-component/$ ➤ component:serve (Serve your components)
```

The component is served running the command within the component folder. It opens the browser pointing to the demo of

component: `http://localhost:3000/components/your-component/demo/index.html`

Demo

The demo should show all the use cases and reflect the status changes (events, properties activations, etc).

The catalog has a [family of demo components](#) available also to ease building it up.

Events

For a common showcase of the events, Cells components are using [cells-demo-event-toaster](#) and so we recommend its use or showcase them similarly.

Unit testing Cells Components (WCT)

Check [Web Component Tester](#)

To launch the tests serve the component with `cells-cli` and go to this

url: `http://localhost:3000/components/cells-component-name/test/`

Documentation

Please follow the [Polymer style guide](#). Cells components need to be thoroughly documented to be used by other developers.

The documentation of the component is present in the following files:

- `bower.json`, in the "keywords" and "description" fields.
- `README.md`
- `.html`, with the main *description*, *declarative examples* and *styling* documentation.
- `.js`, it is where the [properties](#), [methods](#), the [events](#) and [behaviors](#) used in the component will be analyzed if they've been documented appropriately in markdown format.

To see the JsDocs result: `http://localhost:3000/components/cells-component-name/index.html`

If your component has available open variables and mixins, make sure they are documented in the table of # Styling, in the `.html`.

```
# Styling
```

Custom property	Description	Default
-----------------	-------------	---------

:-----	:-----	:-----
--------	--------	--------

--cells-component-name-[node-scope]	[the node to which affects]
-------------------------------------	-----------------------------

[default value]

--cells-component-name	empty mixin	{}
------------------------	-------------	----

Also, leave a **declarative example** of the minimal configuration to use the component.

```
`cells-component-name` description and examples of usage
```

```
# Examples
```

Does **this** one thing:

```
```
```

```
<cells-component-name title="Mocked title"></cells-component-name>
```

```
```
```

We recommend you also leaving an example of the rest of use cases you might have developed, to make easier the reusability.

Utils

Theme and styling

Check the extended guide for these topics on [Styling Cells Components](#).

Sass

Cells components have as a development dependency, [cells-sass](#). It is a brief set of functions and mixins which help in the development of the component's SCSS.

Icons

Check an extended guide for this topic, [Cells Icons guide](#)

Internationalization

To have your application in more than one language, Cells offers you `cells-i18n-behavior`, which is available in [Cells Catalog](#).

You can have i18n in your component 2 ways, but first check the documentation of the behavior.

1. Traditional way: The string that you would use as parameter for the translation method is directly the msgid, which will have its corresponding translations in the `locales/` folder of the component. Using that msgid, you can also [override the default translations of any component from the application](#).

```
2. <p class="component-title">[[t('component-title')]]</p>
```

3. Leaning on the capacity of [overriding translations from the application](#): The string that you would use as parameter for the translation method is a property. So if the text that you pass to the component, is a msgid present in `locales-app/`, it will translate normally.

```
4. properties: {
```

```
5.   heading: String
```

```
6. }
```

```
7. <p class="component-heading">[[t(heading)]]</p>
```

Cells internationalization works using a [Polymer behavior](#).

Publish

Validation and certification

With [cells-cli](#) tool version and validate your component and send it for [certification](#).

Components catalog

In the [Cells Components catalog](#), all the production ready Cells components are published. They are categorized according to their functionality, "families".

- [User interface components](#)
 - Families: Cells Controls, Cells Inputs, Cells Navigation, Cells Distributors, Cells Widgets, Cells Media, Cells Forms, Cells Charts, Formats
- [Data components](#) that are generated from the BBVA Global API
 - Families: BBVA APIs
- [Templates](#)
 - Families: Templates
- Behavioral components that encapsulate a shared functionality
 - Families: Behaviors
- Components to help you build your component's demos
 - Families: Cells Demos
- [Themes](#) and icon libraries
 - Families: Themes
- Third parties wrappers
 - Families: Wrappers
- Components that connect with Cordova plugins
 - Families: Cells Native

The catalog is supposed to grow and so, it is possible to [contribute](#) by sending new components or fixing published ones.

To highlight some components, some of them are tagged

- **NEW:** they are components that have been published in the current version of the catalog
- **TO BE DEPRECATED:** they are dismissed components typically because of the existence of another similar but more "optimized" component

Families

Cells Components are categorized under the following families:

Cells Controls

They provide ways to interact with the page

e.g. buttons, toolbars, toggle elements

Cells Inputs

They allow users to enter/choose/input data

e.g. inputs, radio buttons, dropdowns

Cells Navigation

They are aimed to help users navigate through an app

e.g. menus, pagination bullets, header blocks

Cells Distributors

They provide specific order and disposition for content

e.g. lists, grids, columns

Cells Widgets

They are full featured dynamisms

e.g. accordions, tabs, modals, sliders, swipers

Cells Media

They include and manage media elements

e.g. video, audio, image formats & effects

Cells Forms

They are composed of multiple elements for specific steps or functionalities

e.g. login pages, vote pages

Cells Charts

They are data presentation components

e.g. bar charts, diagrams, plots, tree maps

Cells Native

Elements are aimed to import & connect Cordova plugins for native apps

Cells Demos

They help to improve components showcasing

e.g. dynamic mobile frames, toast to indicate event firing

BBVA APIs

BBVA Data Managers and Data Providers from BBVA Global API

Templates

Define available page regions and main structure

e.g. template defining header, main, sidebar and footer regions

Themes

App look&feel, and visual resources like icons

Behaviors

Shared code components aimed to be used from other components.

e.g. i18n functionalities, ajax helpers

Formats

Components to give different presentations of raw strings or numbers.

e.g. currencies, dates

Wrappers

Components for wrap and encapsulate third-party libraries

e.g. Swiper and cells-swiper-import

Styling User Interface components

- Overview
- [Open styles](#)
 1. [CSS Variables](#)
 2. [Mixins \(Polymer\)](#)
 3. [:root styles](#)
 4. [Shared styles](#)
- [Using themes](#)
- [Further reading and resources](#)

Overview

Cells User Interface components follow web components standards, and are styled taking in account the [Shadow DOM specifications](#).

Which basically means, that **component's styles are encapsulated**. Each component has its full set of styles to work and show correctly. The component's classes and IDs can be used inside its local DOM regardless of whether they are being used outside the component. Local DOM prevents users from styling other components by accident. Read more, [Getting started with Shadow DOM](#)

And although it's not possible to style directly the component's local DOM from outside, as a component's author, you should expose ways for users to style specific parts of the component (in other words, leave [component's styles open](#)).

Cells components are Polymer components, so it's possible to use Polymer tools and recommendations for it. Check [Polymer Local DOM](#) definitions in order to familiarise yourself with Local DOM, Shadow DOM and Shady DOM.

Checkout all [Polymer Styling](#) possibilities for further reading.

Full styling?

Cells components aim to be as reusable as possible.

The component should always have a full set of styles, so it's functional and visually complete. This way, the component will be fully usable in any context and 'standalone'. In order to adapt a component's styles to an application's look&feel, authors should use [the application theme](#) and shared styles.

Switching between Shadow/Shady DOM, and native CSS custom properties

By default, Polymer uses a Shadow DOM polyfill called Shady DOM. As useful as it is, Cells components should always be styled attending to both Shady & Shadow DOM.

You can check [Polymer Global Settings](#) to learn how to switch between Shady/Shadow DOM, and how to switch on native Custom CSS properties (if the browser supports them, of course).

Open styles

Cells components have **open styles**, using **CSS variables** and Polymer **mixins** to allow redefining styles from outside.

Also, **shared styles** can be used, mainly from themes, to customize the styles of the component.

1. CSS variables

They allow to define values which can be reused through multiple selectors and properties.

With them, it's easy to for example define a standard font from [your theme](#), so it applies to different components. The values are usually defined in the host of the scope in which they are going to apply.

You can work with CSS variables following the standard specification (check [Using CSS Variables](#)), but keep in mind the following recommendations:

Naming

These variables will be used from outside your component in order to modify its styles. So it's important to apply semantic naming to them in order to keep them maintainable and easy to read.

Each variable should be named as **your component + the property** it affects.

For example:

```
--my-custom-element-color
```

```
--my-custom-element-bg-color
```

```
--my-custom-element-box-shadow
```

Fallback values

CSS variables are applied using the `var()` function. Be sure to always provide a **fallback value** ([a second value in the var\(\) declaration](#)). That value will be applied when the variable is not defined; for example, when your theme does not need to provide a different value for it. This is the preferred way to style an element while maintaining it open for others to modify/override it.

```
property: var(--my-custom-element-property, defaultValue);
```

```
:host {
```

```
  background-color: var(--my-custom-element-background-color, #fff);
```

```
}
```

One step more would be to have a themed fallback value. Meaning, using a second variable for the fallback value. This second variable would be an already defined variable (e.g. `--bbva-white`) with its fallback value as the default value.

```
:host {  
  background-color: var(--my-custom-element-background-color, var(--bbva-white, #fff));  
}
```

Redefining CSS variables from parent components

Keep in mind that defining them can prevent further definitions.

For example, if you define a CSS variable value for a `<child-element>` of your element (`<your-element>`) under `:host`, and also define it with a different value on the `:host` of an `<upper-element>`, the definition from the `<upper-element>` will be overridden by the one in your element.

```
<upper-element>  
  <your-element>  
    <child-element></child-element>  
  </your-element>  
</upper-element>
```

```
<dom-module id="child-element-styles">  
  :host {  
    color: var(--child-element-color, white);  
  }  
</dom-module>
```

```
/* upper-element styles */  
:host {
```



```
--child-element-color: blue;
```

```
}
```

```
/* your-element styles */
```

```
:host {
```

```
--child-element-color: red;
```

```
}
```

If you think the definition of a custom property value could be overridden by an upper element, you can also leave the definition open.

```
/* upper-element styles */
```

```
:host {
```

```
--your-element-child-element-color: blue;
```

```
}
```

```
/* your-element styles */
```

```
:host {
```

```
--child-element-color: var(--your-element-child-element-color, red);
```

```
}
```

In `.scss`/`SASS` files, mixins need to be declared using the `unquote` function, in order to be correctly parsed:

```
--my-mixin: unquote("{
```

```
background: red;
```

```
font-size: 18px;
```

```
display: flex;
```

```
}");
```

Limitations

Cross-platform support for custom properties is provided in Polymer by a JavaScript library that approximates the capabilities. For performance reasons, Polymer does not attempt to replicate all aspects of native custom properties.

Styling distributed elements is not supported, which is important, for example, for selectors like [:host ::content selector](#)

There are also limitations to inheritance.

Please make sure you check Polymer [Custom properties shim limitations](#)

2. Mixins (Polymer)

As described in [Polymer Styling documentation](#), mixins allow to define an entire set of properties as a single mixin or declaration. This is specially useful to add non-previously-existing properties to a component selector, or override property values.

You can open CSS mixins in your component with **@apply(--mixin-name)**, and define them as exposed in Polymer documentation, but keep in mind the following recommendations:

Naming

The mixin name should combine the component name and the selector target it is applying to. For example, a correct name for the :host

```
:host {  
  @apply(--my-custom-element);  
}
```

while a correct mixin name for the active state of a link inside the element could be

```
:host .link:active {  
  @apply(--my-custom-element-link-active);  
}
```

Apply at the end of the selector

In the CSS selector the `@apply` declaration gets replaced by the properties the user set when defining the mixin.

This means that any property set **AFTER** the `@apply` in the selector **won't be overridable**.

So, it's important to put the `@apply` at the end of the selector, after all other properties.

```
.title {  
  color: #000;  
  @apply(--my-custom-element-title);  
}
```

Otherwise, redefining the color using the mixin will not work:

```
--my-custom-element-title: {  
  color: red  
};  
  
...  
  
.title {  
  @apply(--my-custom-element-title);  
  color: #000;  
}  
  
...  
  
/* Resulting in the following CSS: */  
  
.title {  
  color: red;  
  color: #000;  
}
```

3. :root styles

CSS variables and mixins define parts of the component that can be styled from the outside. You can assign values to those variables and mixins from your [theme](#) using the :root selector. With this selector, the values you define will apply to all component instances in your app, so it's the best option to establish the general "look & feel" of your app components.

:root selector is used inside a [custom-style](#).

```
<style is="custom-style">
```

```
:root {
```

```
--my-custom-element-element-text-color: red;
```

```
--my-custom-element-title: {
```

```
  color: red;
```

```
  background-color: blue;
```

```
};
```

```
}
```

```
</style>
```

4. Shared styles

Sometimes custom properties and mixins are not enough. Maybe you need to style specific parts not covered with a CSS variable or an open mixin, extend your component's styling in a very specific way, or make sure some component CSS variable receives certain value just when that component is a child of another specific one.

As a last option, [shared styles](#) provide a way to include an additional set of selectors and properties in the component styles. Shared styles consist on defining elements `<dom-module>` just containing styles (we'll call them style modules), which can then be included in one or more components.

That style module is usually defined in your theme (shared-styles.html). You can define new selectors, properties, redefine custom properties...

```
<!-- theme-base-shared-styles.html -->
```

```
<dom-module id="my-custom-element-shared-styles">
```

```
  <template>
```

```

<style>

  /* new value of custom property for specific element */

  :host {

    --my-second-custom-element-warning-color: red;

    color: blue;

  }

  :host(.info) {

    color: red;

  }

</style>

</template>

</dom-module>

```

Any component with an include for that component in its style tag will apply that style module. All Cells components should have an include for a shared styles module named after them in their 'style' tag.

```

<!-- include the standard styles and the shared styles -->

<style include="my-custom-element-style my-custom-element-shared-
styles"></style>

```

Using themes

Check the [themes guide](#) to provide to the application, an homogeneous "look & feel".

In brief Cells components do not have direct dependencies to themes (they are just devDependencies for demo purposes); they're expected to be imported a single time from the `bower.json` of an application. Therefore, themes should be only imported within the `demo/` of the component.

Further reading & resources

- [Browser rendering optimization](#)
- [Shadow DOM v1](#)
- [Lea Verou - Variables: var\(--subtitle\)](#)

- [Native CSS mixins](#)
- [CSS Containment](#)

Internationalization

Cells User Interface Components use [cells-i18n-behavior](#) for translations.

This is why, the Cells App at serve time, concatenates all the `locales/` from `components/` to one single `.json` file per language. Output located in `app/locales/`.

Also, some configuration is made in `app/scripts/app.js` regarding the global Object `I18nMsg`.

Overriding locales

To override the default locales of components, add the folder `locales-app/` inside `app/` to your application where you have the respective language `.json` files and in there define the ones you want to override.

Serving the app with `cells-cli`, a task `buildI18n` will concatenate that folder (`locales-app/`) overriding the locales you defined in `app/`.

Theme

The theme gives to the application a "look&feel". Themes are a place to:

- Have custom variables and mixins

The so called **project themes** mean that are doing one or more of the following:

- Override other components' styles
- Extend another theme
- Shared styles

Usage

Themes are [application level dependencies](#).

Have custom variables and mixins

Themes define custom property values at document level, usually setting main and secondary colors, fonts, button styles...

Use the theme to have your project's style guide (e.g. color palette) as custom properties and use it across your components.

This will make the look&feel of the application very easily to change.

Example

Check `cells-coronita-theme`, through its catalog `cells-coronita-theme-catalog`

Override other component's styles

In order to change the look&feel of a component without actually making changes to it, use the theme for that.

The theme should be used to give the new values to the *open mixins* and *variables* that are available from the components.

Save yourself from doing Pull Requests, as long as your desired changes are graphical and can be achieved only with CSS

Example

Check the `example-project-theme`

How

Redefine the variable globally or scoped ([Polymer Styling](#)):

```
<style is="custom-style">

  /* Globally */

  :root{

    --cells-molecule-toggle-button: {

      background-color: darkblue;

      height: 20px;

      width: 48px;

    };

  }

  /* Scoped */
```

```
cells-basic-login {
```

```
--cells-molecule-toggle-button: {
```

```
background-color: darkblue;
```

```
height: 20px;
```

```
width: 48px;
```

```
};
```

```
}
```

```
</style>
```

Extend another theme

To customize a theme's variables and mixins, you can either import it totally or just the parts that will be useful for your project. For example, fonts may differ:

In `example-project-theme.html`:

Import the whole theme you want to reuse,

```
<link rel="import" href="../cells-coronita-theme/cells-coronita-  
theme.html">
```

Or just some parts,

```
<!-- vars -->
```

```
<link rel="import" href="cells-coronita-theme-color.html">
```

```
<link rel="import" href="cells-coronita-theme-layout.html">
```

```
<!-- mixins -->
```

```
<link rel="import" href="cells-coronita-theme-utils.html">
```

```
<link rel="import" href="cells-coronita-theme-buttons.html">
```

```
<link rel="import" href="cells-coronita-theme-boxes.html">
```

```
<link rel="import" href="cells-coronita-theme-components.html">
```


Shared styles

Themes can also contain shared style modules for each of our components. That way, it's possible to modify/override the CSS of the components without modifying the component itself. Read how and when to use the *shared-styles* in the [styling cells components guide](#)

Also, available the [Polymer Shared styles documentation](#)

Application level dependencies

They are basically:

- [Themes](#)
- [Iconsets](#)

The application dependencies are components that have to be imported a single time in the application. And so, they should be **development dependencies** of *your components* and **production dependencies** of *your application*.

Component

bower.json:

```
...  
"devDependencies": {  
  ...  
  "coronita-icons": "^1.0.0",  
  "cells-coronita-theme": "^1.0.0"  
}
```

And so, imported only in the `demo/` of the component, `demo/index.html`:

```
<head>  
...  
<link rel="import" href='../..coronita-icons/coronita-icons.html'>
```

```

<link rel="import" href='../cells-coronita-theme/cells-coronita-
theme.html'>
...
</head>

```

Application

bower.json:

```

...
"dependencies": {
...
"coronita-icons": "^1.0.0",
"cells-coronita-theme": "^1.0.0"
}
...

```

Also in the application they are imported in the `app/tpls/initial-components-imports.tpl` template:

```

<link rel="import" href="cells-polymer-bridge/cells-polymer-
bridge.html">
<link rel="import" href="polymer/polymer.html">
<link rel="import" href="coronita-icons/coronita-icons.html">
<link rel="import" href="cells-coronita-theme/cells-coronita-
theme.html">

```

Best Practices

Style Web Components

- [:host selector](#)
- [Sass](#)
- [Performance](#)
- [Other know-hows](#)

:host selector

:host display and :host hidden

Custom elements behave as "display: inline" elements by default. So, it's usual to set a `:host { display: block; }` selector in the component CSS, to make it behave as a block, for example. However, Polymer uses attribute styles like `[hidden] {display: none; }` in order to hide components with the hidden attribute. Under Shadow DOM, the `display: block` property assigned to `:host` can override the hidden property. So, it is a good practice to include this selector in your components:

```
:host([hidden]) {  
  
  display: none;  
  
}
```

Parenthesis for classes and attributes

Always use parenthesis for classes, attributes... when applying them to the `:host` selector.

```
:host([hidden]) { ... }  
  
:host(.large) { ... }
```

Do NOT use parenthesis when referring to pseudoelements like `:before` and `:after`.

```
:host:before { ... }  
  
:host:after { ... }
```

::content usage

When using ::content to select distributed children, always leave a white space between its parent and ::content, so `content` is treated as a DOM element.

```
:host::content .reverse { ... } /* Not ok */
```

```
:host ::content .reverse { ... } /* Ok */
```

SASS

Cells uses SASS for styling. This allows to use a set of common resources and functions to help with the component styling process.

For more information on cells-sass, please check out [its repository](#)

Usage

In the SASS file of your component, `my-component.scss`, include an import to cells-sass:

```
@import "bower_components/cells-sass/main";
```

this provides a brief set of functions and mixins focused in typography, reset and accessibility, to help in the development of the components' SCSS.

Alternatively, you could use individual imports

```
@import "bower_components/cells-sass/utils.scss"
@import "bower_components/cells-sass/mixins.scss"
@import "bower_components/cells-sass/breakpoint.scss"
```

Performance

Some practices can help to improve components & apps performance, specially for animations.

will-change

This property can be used to improve animations performance on the page, telling the browser which properties and elements are just going to change, but it should be used carefully so it doesn't actually harm performance. [CSS-Tricks on will-change](#)

Viewport units instead of percentage units

Animations using [viewport units](#) (for example, opening a sidebar panel which should move 40vw of the screen width) can perform better than animations using percentage units.

backface-visibility & perspective 1000

Setting `backface-visibility: hidden` and `perspective: 1000` to an element can help to trigger GPU acceleration on the element rendering while preventing flickering effects in Chrome and Safari, as described [here](#).

Polymer.RenderStatus.afterNextRender

Polymer provides a callback that's called just after an element has finished rendering. This can be useful to establish and optimize rendering defers and sequences, as described [here](#).

CSS containment

The [contain](#) property allows to limit the scope to which certain effects and styles apply.

Browser rendering optimization

By knowing which properties will trigger reflow (calculating styles and layout), it's possible to help the browser avoid performance problems.

- [Rendering performance](#)
- [CSS triggers](#)
- [What forces layout/reflow](#)

Other "know-hows"

Web components, Polymer and Cells components are continuously evolving, and so, it's possible to still find little problems or issues.

Animation names & Polymer parser

Animation names can cause problems when Polymer parses them for Shady DOM. For example, given the following:

```
@keyframes from-top {  
  0% { opacity: 0; }
```

```
100% { opacity: 1; }
```

```
}
```

```
@keyframes from-top-to-bottom {
```

```
0% { background-color: red; }
```

```
100% { background-color: yellow; }
```

```
}
```

```
.elem1 {
```

```
animation: from-top 2s infinite;
```

```
}
```

```
.elem2 {
```

```
animation: from-top-to-bottom 2s infinite;
```

```
}
```

Polymer will correctly parse the keyframes names for Shady DOM, but not the animation calls, actually producing a mismatch:

```
@keyframes from-top-my-component-0 {
```

```
0% { opacity: 0; }
```

```
100% { opacity: 1; }
```

```
}
```

```
@keyframes from-top-to-bottom-my-component-0 {
```

```
0% { background-color: red; }
```

```
100% { background-color: yellow; }
```

```
}
```

```
.elem1 {
```

```
  animation: from-top-my-component-0 2s infinite;
```

```
}
```

```
.elem2 {
```

```
  animation: from-top-my-component-0-to-bottom 2s infinite;
```

```
}
```

To prevent this, one animation name should not "contain" other animation names.

Tap events & Click events in Cordova

Although Polymer recommends using tap events over click events, this has caused problems before when the component is used in a Cordova application. So, it's recommended to use click event handlers instead of tap.

CSS Variables & calculations

Without native CSS custom properties, Polymer replaces custom property declarations in the DOM with their final values. For example, `--my-component-margin-top: 20px; margin-top: var(--my-component-margin-top)` will get replaced with `margin-top: 20px`. This allows to apply operations directly on the CSS custom properties. For example, `--my-component-margin-top: 20px; margin-top: -(var(--my-component-margin-top))` will get replaced with `margin-top: -20px`, which is also a valid value.

However, using native CSS custom properties, that replacement is not applied, as the browser directly manages the custom property value. This means that browsers like Chrome do not correctly understand operations applied directly to the custom property.

To avoid problems, always use `calc()` function to apply operations on custom properties. For example: `margin-top: calc(-1 * var(--my-component-margin-top))`.

Using icons

- [Iconsets](#)
- [Icons in components](#)
- [Adding new icons to coronita-icons](#)
- [Guides for designers](#)

Iconsets

A Polymer Iconset is a component using the `iron-iconset-svg` element, that includes a collection of SVG definitions for icons grouped in a single SVG in a similar manner that a [SVG sprite](#).

- Iconsets are [application level dependencies](#), so they should not be included as a direct dependency of a component (in dependencies) or imported directly in the component. **They are imported as a devDependency and included only in the component's demo.** A component may use different iconsets in different contexts (applications).
- Cells has [some existing iconsets available](#).
- There is a [tool to generate iconsets](#) from a folder of SVG files.

Icons in components

With any iconset, now you can use either the `iron-icon` or the `cells-atom-icon` to display the icons. `cells-atom-icon` provides some CSS classes to set the size of the icon (only even numbers), while using `iron-icon` you can set the icon size by binding the corresponding size property to an inline `style` attribute.

When your component uses icons,

1. Make the icon's ID and size configurable:

.js

```
properties: {  
  closeIcon: {  
    type: String,  
    value: 'coronita:close'  
  },  
  closeIconSize: {  
    type: Number,  
    value: 24  
  }  
}
```

.html


```
<iron-icon icon="[[closeIcon]]" style$="width: [[closeIconSize]]px; height: [[closeIconSize]]px;"></iron-icon>
```

or

```
<cells-atom-icon icon="[[closeIcon]]" class$="icon-size-[[closeIconSize]]"></cells-atom-icon>
```

2. Also document that your component needs an iconset for its minimal configuration, in the `.html` and `README.md` (this paragraph is automatically included by the component generator if you choose to use icons):

3. **## Icons**

4.

5. Since this component uses icons, it will need an

[`iconset`] (<https://globaldevtools.bbva.com/bitbucket/projects/CS/repos/cellsjs-guides-resources/browse/docs/best-practices/cells-icons.md>)

6. in your project as an [`application level`

`dependency`] (<https://globaldevtools.bbva.com/bitbucket/projects/CS/repos/cellsjs-guides-resources/browse/docs/advanced-guides/advanced-guides/application-level-dependencies.md>) .

7. In fact, this component uses an iconset in its demo.

Adding new icons to `coronita-icons`

To prevent the creation of duplicated icons with different names or very similar icons, **there is a single team dedicated to create and maintain the Coronita iconset**. If you need an icon that is not available in `coronita-icons`, you have to follow the following steps in order to include it in future releases of the component.

1. Check if your icon already appears in [BBVA-Coronita-Icons-Master.pdf](#). If the icon exists in the PDF, it is also available as SVG in [Google Drive](#).
2. If your icon already exists, simply make a **Pull Request** with the new icon(s) downloaded from [BBVA icons svg in Google Drive](#) to the `coronita-icons-svg` repository. This repo is a **dependency of the coronita-icons** component. Most of the times, coronita-

icons will be updated with the new icons by the same person who approved your PR to `coronita-icons-svg`. If that is not the case, you can make a Pull Request to `coronita-icons` with the updated iconset. To update the `coronita-icons` component you need to use the `iconset-generator`. Check the [installation and usage instructions](#) in its README.

3. If your icon does not exist, you can ask for it via the [request excel](#).

Note: you may need to request access to some files in Google Drive.

Guides for designers

There are also some guides for designers and/or UX:

1. [Create icons SVGs](#)
2. [BBVA SVG Guide](#)

a11y and a11y for mobile

Handy resources

- [Accessibility in the Platform - by Rob Dobson](#)
- [A11ycasts with Rob Dobson](#)
- [Accessibility is My Favorite Part of the Platform - Google I/O 2016](#)
- [A11y with Polymer - The Polymer Summit 2015](#)
- [Google Accessibility course](#)

Follow Golden Standards

<https://github.com/webcomponents/gold-standard/wiki> >> Minimum for universal

accessibility with mark

Minimum accessibility checklist

- An input should have an associated label. If the design doesn't provide one, it should be visually hidden.
 - Case of CELLS: Use the hidden class (which is actually a visually-hidden class, you can check the `cells-sass` devDependency), and have in mind that if you want to hide the element for all users, you have to use `display:none` or the 'hidden' attribute.
- An element that has an associated event (click or tap) should be focusable. If the element is not naturally focusable (a, button, input, etc), use `tabindex="0"` and add the event `keydown` or `keyBindings` Polymer provides (giving `tabindex="0"` is not enough to make an element clickable).
- An element that appears in screen as a consequence of some user action should announce its appearance.

- You can use `role="alert"` (acts as a `aria-live="assertive"`)
- If that element has a clickable element inside, it should be focused automatically to the element and when the dialog or alert is closed, give back the focus to the element where the user was previously.

Implement a11y

Handy components from Polymer to implement a11y

- [iron-a11y-announcer](#) → "iron-a11y-announcer is a singleton element that is intended to add a11y to features that require on-demand announcement from screen readers. In order to make use of the announcer, it is best to request its availability in the announcing element."
- [iron-a11y-keys](#) → "iron-a11y-keys provides a normalized interface for processing keyboard commands that pertain to [WAI-ARIA best practices](#). The element takes care of browser differences addressed to Keyboard events and uses an expressive syntax to filter key presses."
- [iron-a11y-keys-behavior](#) → "Polymer.IronA11yKeysBehavior provides a normalized interface for processing keyboard commands that pertain to [WAI-ARIA best practices](#). The element takes care of browser differences addressed to Keyboard events and uses an expressive syntax to filter key presses."

Common attributes needed

1. → `role` on custom-elements, only on the ones which are not extending native tags and that they have a functionality equal to the one of a native tag. In other words, make sure your custom-element fulfills these 2 golden standards:
 - "Declared Semantics: Does the component expose its semantics by wrapping/extending a native element, or using ARIA roles, states, and properties?"
 - "Labels — Are the component's significant elements labeled so that a user relying on an assistive device can understand what those elements are there for?"

```
Polymer({
  is: 'my-button',
  hostAttributes: {
    role: 'button'
  }
});
```

- List of roles with examples: [Open Ajax Accessibility](#) list of roles.
- ARIA roles available at [w3cRoles](#)

1. → `tabindex` only for elements that are not naturally navigable through tabulation

```
Polymer({  
  is: 'my-row',  
  hostAttributes: {  
    'tabindex': 0  
  }  
});
```

1. → `aria-hidden="true"` e.g. icons are not relevant for screen readers (aria states and properties [WAI-ARIA](#)).

```
Polymer({  
  is: 'my-icon',  
  hostAttributes: {  
    'aria-hidden': true  
  }  
});
```

1. → `aria-readonly` which should be navigable only in screen reader mode.

e.g. when explicative labels for columns. Note: this won't hide the elements.

```
Polymer({  
  is: 'my-table-th',  
  hostAttributes: {  
    'aria-readonly': true  
  }  
});
```

As shown in the examples, these common attributes can be defined at the custom-element registration. Also, **remember to update aria-states in your JS:**

Example:

```
Polymer({  
  hostAttributes: {  
    role: 'checkbox',  
    'aria-checked': false,  
  },  
  properties: {  
    checked: {  
      type: Boolean,  
      observer: '_checkedChanged'  
    }  
  },  
  _checkedChanged: function() {  
    this.setAttribute('aria-checked',  
      this.checked ? 'true' : 'false');  
  }  
})
```

[polymerSampleSlide](#)

Behaviors backing component

[iron-a11y-keys-behavior](#)

Example:

```
Polymer({  
  is: 'key-aware',  
  behaviors: [Polymer.IronA11yKeysBehavior],  
})
```

```
keyBindings: {  
  'left up': '_prevItem'  
  'right down': '_nextItem'  
}  
});
```

[polymerSampleSlide](#)

WCAG applied to mobile (summary)

[Mobile Accessibility: How WCAG 2.0 and Other W3C/WAI Guidelines Apply to Mobile](#)

Related with the FIRST principle: Perceivable

Small Screen Size

- Providing a **reasonable default size for content and touch controls** (see "B.2 Touch Target Size and Spacing") to minimize the need to zoom in and out for users with low vision.
- Positioning form fields below, rather than beside, their labels (in portrait layout).

Zoom / magnification

- Text to be resizable without assistive technology up to 200 percent. To meet this requirement **content must not prevent text magnification by the user**.
- Ensure that the browser pinch zoom is **not blocked by the page's viewport meta element** so that it can be used to zoom the page to 200%. Restrictive values for user-scalable and maximum-scale attributes of this meta element should be avoided. Note: Relying on full viewport zooming (e.g. not blocking the browser's pinch zoom feature) requires the user to pan horizontally as well as vertically. While this technique meets the success criteria it is less usable than supporting text resizing features that reflow content to the user's chosen viewport size. It is best practice to use techniques that support text resizing without requiring horizontal panning.

Contrast

Mobile devices are more likely than desktop/laptop devices to be used in varied environments including outdoors, where glare from the sun or other strong lighting sources are more likely. This scenario heightens the importance of the use of a good contrast for all users and may compound the challenges that users with low vision have accessing content with poor contrast on mobile devices.

Related with the SECOND principle: Operable

Keyboard control

Keyboard accessibility remains as important as ever, and most major mobile operating systems do include keyboard interfaces, allowing mobile devices to be operated by external physical keyboards (e.g. keyboards connected via Bluetooth, USB On-The-Go) or alternative on-screen keyboards (e.g. scanning on-screen keyboards)

Touch Target Size and Spacing

Elements must be big enough and have enough distance from each other so that users can safely target them by touch.

Best practices for touch target size include the following:

- Ensuring that touch targets are at least 9 mm high by 9 mm wide.
- Ensuring that touch targets close to the minimum size are surrounded by a small amount of inactive space.

Touchscreen gestures

- Gestures in apps should be as easy as possible to carry out.
- **Using the mouseup or touchend event to trigger actions helps prevent unintentional actions during touch** and mouse interaction. Mouse users clicking on actionable elements (links, buttons, submit inputs) should have the opportunity to move the cursor outside the element to prevent the event from being triggered. This allows users to change their minds without being forced to commit to an action.
- **Another issue with touchscreen gestures is that they might lack onscreen indicators that remind people how and when to use them.** For example, a swipe in from the left side of the screen gesture to open a menu is not discoverable without an indicator or advisement of the gesture.

Placing buttons and links where they are easy to access

Mobile sites and applications should position interactive elements (buttons, links) where they can be **easily reached** when the device is held in different positions.

Related with the THIRD principle: Understandable

Changing Screen Orientation (portrait/landscape)

Some mobile applications automatically set the screen to a particular display orientation (landscape or portrait) and expect that users will respond by rotating the mobile device

to match. However, some users have their mobile devices mounted in a fixed orientation (e.g. on the arm of a power wheelchair).

Mobile application developers should try to support both orientations.

Changes in orientation must be programmatically exposed to ensure detection by assistive technology such as screen readers. For example, if a screen reader user is unaware that the orientation has changed the user might perform incorrect navigation commands.

Grouping operable elements that perform the same action

When multiple elements perform the same action or go to the same destination (e.g. link icon with link text), these should be contained within the same actionable element. This increases the touch target size for all users and benefits people with dexterity impairments. It also reduces the number of redundant focus targets, which benefits people using screen readers and keyboard/switch control.

Provide instructions for custom touchscreen and device manipulation gestures

For many people, custom gestures can be a challenge to discover, perform and remember.

Therefore, instructions (e.g. overlays, tooltips, tutorials, etc.) should be provided to explain what gestures can be used to control a given interface and whether there are alternatives. To be effective, the instructions should, themselves, be easily discoverable and accessible. The instructions should also be available anytime the user needs them, not just on first use.

Related with the FOURTH principle: Robust

Set the virtual keyboard to the type of data entry required

Some mobile devices also provide different virtual keyboards depending on the type of data entry.

For example, using the different HTML5 form field controls (see Method Editor API) on a website will show different keyboards automatically when users are entering information into that field. Setting the type of keyboard helps prevent errors and ensures formats are correct but can be confusing for people who are using a screen reader when there are subtle changes in the keyboard.

How to create a responsive element

So we've got to constantly ask ourselves, how can I make this flexible for others to use?

The first part in this process is figuring out when we want to change the appearance of our element.

Every site is going to have different amounts of content and layout, and in particular at different screen sizes, it's going to need to know when to update its appearance.

So how can I create a responsive element that doesn't bake in all of its break points? One option is to allow the developer to actually pass in a breakpoint, to actually configure my element from the outside and tell me when they want it to collapse.

We will use

iron-media-query for our level changes screen size

```
<!-- Responsive handlers -->
```

```
<iron-media-query id="mq-phone" full query="(min-width:320px) and (max-width:768px)"
```

```
  query-matches="{{isPhoneSize}}"></iron-media-query>
```

```
<iron-media-query id="mq-tablet" full query="(min-width:769px) and (max-width:959px)"
```

```
  query-matches="{{isTabletSize}}"></iron-media-query>
```

```
<iron-media-query id="mq-desktop" query="(min-width:960px)"
```

```
  query-matches="{{isDesktopSize}}"></iron-media-query>
```

The CellsBehaviors.DeviceBehavior for our level changes device

```
<template is="dom-if" if="{{isIOS}}">IOS</template>
```

```
<template is="dom-if" if="{{isAndroid}}">Android</template>
```

```
<template is="dom-if" if="{{isCordova}}">Cordova</template>
```

```
<template is="dom-if" if="{{isWindows}}">Windows</template>
```

```
<script>
```

```
  Polymer({
```

```
    is: 'my-element',
```

```
    behaviors: [CellsBehaviors.DeviceBehavior],
```

```

listeners: {
  'orientation-changed': '_logNewOrientation'
},

_logNewOrientation: function(e) {
  console.log("changed orientation-", this.orientation);
}
});
</script>

```

Testing components

This guide shows you the basics to run unit tests, and how to accomplish various tasks and scenarios using the Web Component Tester library (the underlying library that powers Polymer CLI's testing tools).

The underlying library that powers Polymer CLI's unit testing tools is called Web Component Tester.

Web Component Tester is an end-to-end testing environment built by the Polymer team. It enables you to test your elements locally, against all of your installed browsers, or remotely, via Sauce Labs. It is built on top of popular third-party tools, including:

- Mocha for a test framework, complete with support for BDD and TDD. <https://mochajs.org/>
- Chai for more assertion types that can be used with your Mocha tests. <http://chaijs.com/>
- Sinon for spies, stubs, and mocks. <http://sinonjs.org/>
- Selenium for running tests against multiple browsers.
- Accessibility Developer Tools for accessibility audits.

Run tests with Cells Cli

When you run in component folder:

```
$ cells component:validate
```

Cells Cli automatically searches for a test directory and runs any tests it finds in there.

Run tests interactively

You can also run your tests in the browser. This allows you to use the browser's DevTools to inspect or debug your unit tests.

Run in component folder (For example: my-element):

```
$ cells component:serve
```

And then, to run the basic unit test in the browser, you would open a web browser and go to the following URL:

```
http://localhost:8080/components/my-element/test/index.html
```

What do I need test for a component

We should mainly verify that the implementation and behavior of our component is correct. In this way we must test:

- Properties that the component has. (Default values, types, etc ...)
- Functions that the component has. It is necessary to check both public and private functions, as well as the different scenarios depending on the different values they can receive
- DOM of the component. It is necessary to verify that the components are found and behave as expected in the different use cases that the component can have depending on its properties. Update DOM components, translations, component condition elements.
- Events. It is necessary to verify that the component emits the expected events for its communication with the outside.

Testing properties

Here's an example of how we can make sure we've successfully put a property in our component:

```
test('should have my-element have property equal to "propA"', function() {
```

```
    var element = fixture('ExampleTestFixture');
```

```
    expect(element).to.have.property('propA');
```

Checking the type of a component property:

```
test('should have property "propA" as a boolean', function() {

  var element = fixture('ExampleTestFixture');

  expect(element.propA).to.be.a('boolean');

});
```

Testing events

The following is an example of how we can check events launched by a component:

```
test('test if component dispatch a event', (done) => {

  var element = fixture('BasicTestFixture');

  sinon.spy(element, 'dispatchEvent');

  element.functionWithDispatch();

  expect(element.dispatchEvent).called;

  expect(element.dispatchEvent.args[0][0].type).to.equal('my-event');

  element.dispatchEvent.restore();

  done();

});
```

Backwards compatibility. The fire instance method in the legacy API. If you test Polymer 1.X components, ensure that yours events emits with fire instance.

Testing DOM

Here's how to select Shadow DOM elements from the component

```
test('setting a property on the element works', function () {

  var element = fixture('ChangedPropertyTestFixture');

  assert.equal(element.prop1, 'new-prop1');

  var elementShadowRoot = element.shadowRoot;

  var elementHeader = elementShadowRoot.querySelector('h2');

  assert.equal(elementHeader.innerHTML, 'new-prop1!');
```

```
});
```

Example of how to test dom-repeat or dom-if blocks:

```
<div> List element: </div>
```

```
<template is="dom-repeat" items="{{items}}" id="elementList">
```

```
  <div>Name: <span>{{item.name}}</span></div>
```

```
  <div>Amount: <span>{{item.amount}}</span></div>
```

```
</template>
```

```
test("check when items in a list change", function(done) {
```

```
  var element = document.querySelector('#elementList');
```

```
  element.set("elements",
```

```
[
```

```
  {name: 'Account A', description: 'Test1'
```

```
},
```

```
  {name: 'Account B', description: 'Test2'
```

```
}]
```

```
]);
```

```
  flush(function () {
```

```
    assert.equal(element.length, 2);
```

```
    done();
```

```
  }
```

```
});
```

Asynchronous tests

Whereas the traditional way of testing asynchronous code in Mocha is to have your test function take a done callback function:

```

test('testing a async context with done', (done) => {
  setTimeout(function () {
    assert.equal(true, true);
    done();
  }, 200);
});

```

With built-in promise support - you no longer need the done callback. All you have to do is return a promise from the test function:

```

test('testing a promise without done', () => {
  let promiseTest = new Promise((resolve, reject) => {
    setTimeout(function () {
      resolve('Finish!');
    }, 300);
  });
});

```

```

return promiseTest.then((success, error) => {
  assert(true, true);
});
});

```

Spies

A test spy is a function that records arguments, return value, the value of this and exception thrown (if any) for all its calls. There are two types of spies: Some are anonymous functions, while others wrap methods that already exist in the system under test

```

test('test with spy', (done) => {
  sinon.spy(element, 'methodTest');
});

```

```
element.methodTest();
```

```
expect(element.dispatchEvent).called;
```

```
element.methodTest.restore();
```

```
});
```

You can see more examples in: <http://sinonjs.org/>

End-to-End Testing

Example implementing a couple of basic scenarios for end to end testing a Cells application.

You can checkout a e2e example project in <https://globaldevtools.bbva.com/bitbucket/projects/BBVACELLSAPP/repos/cellsstarterkit.gb.e2e/browse/README.md?at=develop>

Dependencies

- node 6
- [cells-cli](#)

How to run it

You simply go to the *cellsstarterkit.gb* folder and type

```
cells app:validate
```

This command will run the complete validation flow for the app building it and executing the functional tests on it.

If you just want to execute the e2e project on the app, run

```
cells app::functional-testing
```

For this command to execute correctly, you must have previously built the application to test (by `cells app:build`).

For BOTH commands to execute correctly you have to provide the right *baseUrl* parameter. You can modify this parameter in the file *.piscosour/piscosour.json* in the app, or pass to the tool it via the command line like

```
cells app:validate --functionalTesting.webapp.configParams.baseUrl
```

```
http://yourbaseurlhere
```

```
cells app::functional-testing --
```

```
functionalTesting.webapp.configParams.baseUrl http://yourbaseurlhere
```

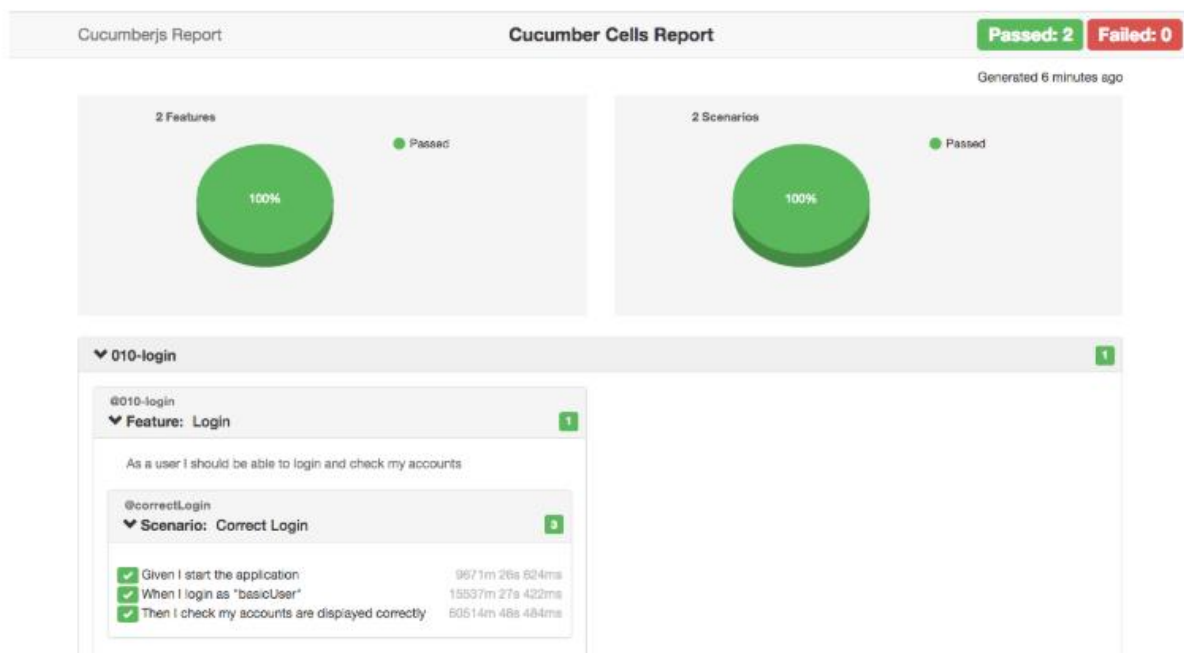
For example, if you have just built the application using the *env* configuration, and the *vulcanizetype*, you should execute the functional tests as follows:

```
cells app::functional-testing --
```

```
functionalTesting.webapp.configParams.baseUrl
```

```
http://localhost:8001/env/vulcanize
```

At the end of the execution (if everything was OK), you should see a report like this:



About the tests

Scenarios

Tests are written using [Gherkin](#) syntax, allowing you to write them in a human readable way. Sample scenarios in this project are found in files ending in `.feature`, all contained in `features/`.

This is the declaration of the test 010-login:

```
@010-login
Feature: Login
  As a user I should be able to login and check my accounts

  @correctLogin
  Scenario: Correct Login
    Given I start the application
    When I login as "basicUser"
    Then I check my accounts are displayed correctly
```

This test starts the `cellsstarterkit.gb` application in the browser, login using a "basicUser" profile and then checks the accounts are shown for the user as expected.

cells-pepino

These scenarios are run by the [cells-pepino](#) test runner, which integrates a bunch of testing technologies and offers them to the developers. Some of these technologies (the ones we'll be using here) are:

- [cucumber-js](#)
- [webdriver.io](#)
- [selenium webdriver](#)

Scenario Steps

[Cucumber-js](#) is a BDD framework used to execute test scenarios like the ones in this project. It does it by mapping each of the steps in any given scenario to a Javascript function which will then execute the actions needed for the test to run.

The code for the steps can be found in `lib/step-definitions/*.js`. There you can see that each of them is mapped to a function by using a regular expression. This is really powerful, as it allows you to pass parameters for each step while maintaining its mapping.

For example, this is the mapping for the step `When I login as "basicUser"`

```
this.When(/^I login as "([^"]*)"$/, loginAsRole);
```

and the function `loginAsRole` associated to the step

```
const loginAsRole = function(role) {
```

```
const roleData = this.getRole(role);
```

```
return Promise.resolve()
```

```
.then(_ => loginUsingIdAndPassword(roleData.id, roleData.password))
```

```
.then(_ => {
```

```
    this.setCurrentRole(role);
```

```
});
```

```
};
```

Above you can see the regular expression used to map the step, the captured parameter ("basicUser" in this case) and the function receiving it which will then execute the actions needed to login using a particular role.

Page objects

As we have seen, the code in the step definitions executes the actions we are describing in the Gherkin scenarios. In e2e application testing these actions are used to simulate the behavior of any user of the web application. To do this we use [Selenium webdriver](#), a browser automation framework which offers us an API for controlling browsers. In the Cells project we also integrate [webdriver.io](#), a set of bindings designed to make the use of the Selenium API from [node.js](#) easier.

The functions called from the step definitions code are actions grouped in `lib/page-objects/*.js`. Each of the modules in this folder is a [Page Object](#). Page objects are modules abstracting the possible actions a user can perform on any given page of the application, and hiding all the HTML related details from the users. This way we can create a DSL which then we can use to write our tests, and make them independent from the page's implementation.

For example, this is the code for the `Dashboard Page Object`

```
const { expectResultLengthToBe } = require('../support/utils');
```

```
const itemList = 'cells-product-item-list';
```

```
const item = 'cells-product-item';
```

```
const selectNthItem = (n) => {
```

```
  return (result) => driver.elementIdClick(result.value[n].ELEMENT);
```

```
};
```

```
const loaded = () => {
```

```
  return driver.waitForVisible(itemList);
```

```
};
```

```
const selectNthAccount = (n) => {
```

```
  return driver
```

```
    .element(itemList)
```

```
    .elements(item)
```

```
    .then(selectNthItem(n));
```

```
};
```

```
const expectNumberOfAccountsToBe = (number) => {
```

```
  return driver
```

```
    .elements(item)
```

```
    .then(expectResultLengthToBe(number));
```

```
};
```

```
module.exports = {
```

```
  loaded,
```

```
  selectNthAccount,
```

```
  expectNumberOfAccountsToBe
```

```
};
```

We can see only particular actions are being exported in the module, and we are hiding the HTML implementation details from the public API consumers.

The `driver` object is an instance of the *webdriver.io* driver that is injected in all the page objects at execution time. It is the object we will use to perform all the actions we need on the different page elements (click, read text value, select a given HTML tag and so).

In the example we can see that the action `selectNthAccount` needs to know the tag name of the list element containing the different account items. It is also necessary to know the tag name for the item elements, then select the one in the 'n'th position and click it. In a future, these tag names may change, or maybe the account list is implemented by using a `<select>` tag. By hiding all these details from the API consumer and exposing only the abstract action (select the account in the nth position), we provide more reusability in our tests and decrease the coupling between our step definitions and the implementation of the application pages.

More information

You can find lots of more information on the frameworks and technologies used in this project in the following links:

- [cells-cli](#)
- [cells-pepino](#)
- [cucumber-js](#)
- [webdriver.io](#)
- [selenium webdriver](#)
- [chai](#)

Native

Hybrid App

Android App

Navigation

Component UI

How to develop screens within an hybrid architecture

Cells Hybrid

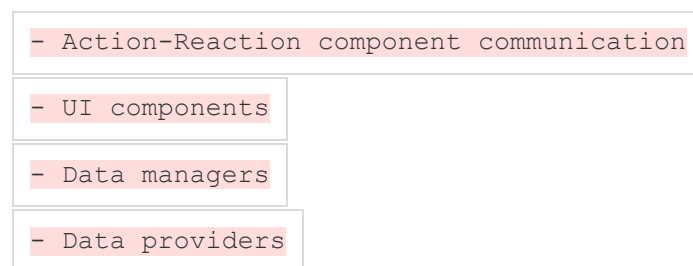
Cells Hybrid is an architecture proposal that allows developing applications by componentization, this approach means native components reusability in screens creation and faster application development.

This document is meant to walk you through a step by step Cells Hybrid project. It will be explained in detail how to configure and Android project in order to use this development approach. Once the project is configured it will be shown how to begin with hybrid screen creation and components communication.

This proposal provides a layer communication component in order to allow message-passing within web layer components and native layer components as well as managing navigations through different screens no matter the layer they belong.

It also provides component aspect and behavior customization.

Cells Hybrid is made of several modules:



Configuration

Project Configuration

Hint: In case you are already using the 'cells-cli' in order to scaffold a new project, you may skip this step and focus directly on 'First Page' step.

Artifactory Setup

First of all, you should set artifactory's configuration [Artifactory's Configuration](#)

Now, add in project/build.gradle:

```
allprojects {
    repositories {
        // ...
        maven {
            url
'https://globaldevtools.bbva.com/artifactory-api/cells-native'
            credentials {
                username artifactory_user
                password artifactory_password
            }
        }
    }
}
```

And then in app\build.gradle add:

```
compile(group: 'com.bbva.cells', name: 'cells-core',  
version: <CURRENT_VERSION>)
```

[Further gradle configuration](#)

Add to app\build.gradle add the following dependencies:

```
implementation 'com.android.support.design:26.0.1'  
api 'android.arch.lifecycle:runtime:1.0.0-alpha9'  
api 'android.arch.lifecycle:extensions:1.0.0-alpha9'  
annotationProcessor 'android.arch.lifecycle:compiler:1.0.0-alpha9'
```

Provide Java 8 support inside android{} node

```
compileOptions {  
    sourceCompatibility JavaVersion.VERSION_1_8  
    targetCompatibility JavaVersion.VERSION_1_8  
}
```

[App configuration](#)

Next step is configuring the Application class.

A RouterFactory must be created, it should be provided with all the routes linked to the activities so it will be able to manage launching them from web components. In this way, a component will perform the opening a screen operation using this route as reference.

In this scenario we will be developing an application that will contain a LoginActivity and a DashboardActivity.

```
public class App extends Application {
```

```
    @Override
```

```
    public void onCreate() {
```

```
        super.onCreate();
```

```
        // Web Routing - needs application router factory definition
```

```
        RouterFactory routerFactory = new RouterFactory(new
```

```
        HashMap<>());
```

```
routerFactory.add("login", new Route(LoginActivity.class));
```

```
routerFactory.add("dashboard", new
```

```
Route(DashboardActivity.class));
```

```
//Native Routing - needs Router class implementation
```

```
ComponentManager.add(Router.ID, new Router());
```

```
}
```

```
}
```

WARNING: Application name should be set inside Manifest.xml file so everything works as expected.

Once the router is correctly configured, AppLifecycle must be initialized:

```
public class App extends Application {
```

```
@Override
```

```
public void onCreate() {
```

```
// ...
```

```
new AppLifecycle.Builder()
```

```
.app(this)
```

```
.routerFactory(new RouterFactory(new HashMap<>()))
```

```
.reactionManager(new PageReactionManager())
```

```
.create();
```

```
}
```

```
}
```

Further App configuration

AppLifecycle Builder allows several configurations using different settings:

Operation	Description
<code>.webContentsDebuggingEnable(boolean enable)</code>	Allows the webView inside the application to be debugged using external tools
<code>.htmlIndex(String indexName)</code>	Allows to load another web page rather than the index.html default given
<code>.flavor(String country)</code>	Specifies what folder inside assets/routes will be selected to work with
<code>.plugins(List plugins)</code>	Allows the javascript code call a list the native plugins
<code>.addPlugin(Plugin plugin)</code>	Allows the javascript code call a specified native plugin
<code>.enableLog(boolean enable)</code>	Enable the library logs
<code>.nativeWebContainer(Route route)</code>	Replace the default WebViewActivity when a web is open
<code>.addTracker(Tracker tracker)</code>	Add a new tracker

```

new AppLifecycle.Builder()
    .app(this)
    .routerFactory(routerFactory)
    .addPlugin(new EmailComposerPlugin())
    .reactionManager(pageReactionManager)
    .flavor(country)
    .webContentsDebuggingEnable(true)
    .htmlIndex(countryList.contains(country) ?
        String.format(URL_PRO, country)
        :
        String.format(URL_PRO,
            DEFAULT_COUNTRY))
    .create();

```


First Page

In order to create our first screen made out from components we must declare an Activity and register it into the Manifest.xml file using an unique identifier.

```
<activity android:name="LoginActivity">  
    <meta-data android:name="cells-tag" android:value="my-unique-id"/>  
</activity>
```

The linked Activity is supposed to own this same ID.

```
public class LoginActivity extend AppCompatActivity {
```

```
    public static final String ID = "my-unique-id";
```

```
    // ...
```

```
}
```

Each component event-driven behavior definition will be defined inside a **PageReaction** class:

```
public class LoginReaction extend PageReaction {
```

```
}
```

PageReactions must be added to our Application class making reference to the Activity in which they are executed.

```
public class App extends Application {
```

```
    @Override
```

```
    public void onCreate() {
```

```
        // ...
```

```
        PageReactionManager reactionManager = new
```

```
PageReactionManager();
```

```

        reactionManager.add(LoginActivity.ID, () -> new
LoginReaction());

```

```

        new AppLifecycle.Builder()
            // ...
            .reactionManager(reactionManager)
            // ...
    }
}

```

Actions-Reactions

Each component **emmits a collection of events** that could be observed and trigger reactions in other components from them. To do so, following expresions are available for component reacting:

[when - Communication](#)

The following example shows how a component triggers an event and its reaction in another component using when command:

```

public class LoginReaction extend PageReaction {

    @Override
    protected void configure() {

        when(R.id.credentials)

            .doAction(LoginCredentialsViewModel.LOGIN_REQUEST)

            .then()

            .in(UserStore.ID, UserStore.class)

            .doReaction((userStore, params) ->
userStore.login(params));

    }
}

```

```
}
```

To be more specific, in this code, two different components are interacting. LoginCredentialsViewModel component is triggering a LOGIN_REQUEST event and as a consequence UserStore will react the way it is defined, in this case performing a login reaction (userStore.login(params):

when - Navigation

The following example shows how a component triggers a navigation event and its reaction using when command:

```
public class LoginReaction extend PageReaction {
```

```
@Override
```

```
protected void configure() {
```

```
when(R.id.welcome)
```

```
.doAction(LoginWelcomeViewModel.ACCESS_TAPPED)
```

```
.then()
```

```
.in(Router.ID, Router.class)
```

```
.doNavigation((activity, router, params) ->
```

```
router.showDashboard(params));
```

```
}
```

```
}
```

when - Web Navigation

In order to perform native to web screen navigations, when() function should be defined the following way:

```
when(R.id.welcome)
```

```
.doAction(LoginWelcomeViewModel.ACCESS_TAPPED)
```

```
.then()
```

```
.doWebNavigation("dashboard")
```

```
// Or with animation
```

```
.doWebNavigation("dashboard", R.anim.slide_left_in,  
R.anim.slide_left_out);
```

when - Multiple Reactions

Multiple Reactions may be defined to be triggered from a single action using and() operator.

```
public class LoginReaction extend PageReaction {
```

```
@Override
```

```
protected void configure() {
```

```
when(R.id.credentials)
```

```
.doAction(LoginCredentialsViewModel.LOGIN_REQUEST)
```

```
.then()
```

```
.in(UserStore.ID, UserStore.class)
```

```
.doReaction((userStore, params) -> userStore.login(params))
```

```
.and()
```

```
.in(Logger.ID, Logger.class)
```

```
.doReaction((logger, params) -> logger.logDebug(params));
```

```
}
```

```
}
```

when - Permissions

In order to execute a reaction with any permission, witPermission(...) operator should be used.

```
when(R.id.gallery)
```

```
.doAction(GalleryViewModel.TAKE_PICTURE)
```

```

        .then()
        .withPermissions(Manifest.permission.CAMERA,
            Manifest.permission.WRITE_EXTERNAL_STORAGE)
        .in(TakePictureActivity.ID, TakePictureActivity.class)
        .doNavigation((activity, component, params) -> {
            // take picture
        });

```

when - Delay

In order to execute a delayed reaction with any permission, delay(milliseconds) operator should be used.

```

when(R.id.gallery)
    .doAction(GalleryViewModel.TAKE_PICTURE)
    .then()
    .delay(3000)
    .in(R.id.loader, LoaderViewModel.class)
    .doReaction((loader, params) -> {
        loader.hide();
    });

```

Filter - onlyIf

The onlyIf operator filters a reaction by only allowing actions through that pass a test that you specify in the form of a predicate function.

```

when(R.id.gallery)
    .doAction(GalleryViewModel.TAKE_PICTURE)
    .then()
    .onlyIf(params -> params != null)
    .in(R.id.loader, LoaderViewModel.class)

```

```

        .doReaction((loader, params) -> { // That reaction only is executed
when params is not null
        loader.hide();
    });

```

trace - Trace

Cells Native Core allows tracking component's events.

You could trace a specified event with the trace() operation.

```

when(R.id.gallery)
    .doAction(GalleryViewModel.TAKE_PICTURE)
    .then()
    .in(R.id.loader, LoaderViewModel.class)
    .trace()
    .doReaction((loader, params) -> {
        loader.hide();
    });

```

Create a new Tracker. In that tracker you will receive all the events using a trace enabled.

```

public class EventTracker implements Tracker {
    @Override
    public void onEvent(TrackingEvent event) {
        // Send to tracker service
    }
}

```

trace - Custom Trace

In order to add custom properties to the tracker event, a custom TrackingEvent could be created.

```
when(R.id.gallery)
```

```
.doAction(GalleryViewModel.TAKE_PICTURE)
```

```
.then()
```

```
.in(R.id.loader, LoaderViewModel.class)
```

```
.trace((actionName, params) -> new CustomEvent("take_picture",
```

```
System.currentTimeMillis()))
```

```
.doReaction((loader, params) -> {
```

```
    loader.hide();
```

```
});
```

```
public static class CustomEvent extends TrackingEvent {
```

```
    private long timestamp;
```

```
    public CustomEvent(String actionName, long timestamp) {
```

```
        super(actionName);
```

```
        this.timestamp = timestamp;
```

```
    }
```

```
@Override
```

```
public Map<String, Object> toMap() {
```

```
    Map<String, Object> result = super.toMap();
```

```
    result.put("timestamp", timestamp);
```

```
    return result;
```

```
}
```

```
@Override
```

```
public String toString() {
```

```
    return null;
```

```

    }
}

```

writeOn

In order to use component emitted data in another component from another page, writeOn() operator should be used.

So, communicating different components in different pages involves a communication channel creation to store the data we want to communicate. For example, if we want to communicate a native to a web component we'll use the WebChannel. For further information about channels, please visit the 'channels' section in this document.

```

Channel<Session> sessionChannel = new Channel<Session>("session-
channel");

public class LoginReaction extend PageReaction {

    @Override
    protected void configure() {
        writeOn(sessionChannel)
            .when(SessionStore.ID)
            .doAction(SessionStore.SESSION_CHANGED);
    }
}

```

In this case, the operator waterfall describes a situation in which by the time the SessionStore emits the 'SESSION CHANGED', this value should be written inside the sessionChannel.

reactTo

In order for a page to read data from an specific channel, reactTo operator must be used.

```

public class LoginReaction extend PageReaction {

    @Override

```



```

protected void configure() {
    reactTo(sessionChanel)
        .then()
        .in(UserStore.ID, UserStore.class)
        .doReaction((userStore, session) ->
            userStore.getCustomerData(session));
}
}

```

Using this operator, adds a reaction in a component so this reaction is executed when the channel (this component is bound to) content changes. In this case, when the data inside sessionChannel changes.

`when(,)`

Bidirectional communication between two components (this means that actions in both components will trigger reactions in the other one) `when(,)` operator must be considered.

I.E, it is expected a behavior in which the `LoginCredentialsView` fires an event to perform a login and then, the `UserStore` component will perform that login operation. Once the `UserStore` component is finished with that task, it should send the results back to the `LoginCredentialsView`.

This operation could be done using two different `when` operators.

```

when(R.id.credentials)
    .doAction(LoginViewModel.LOGIN_REQUEST)
    .then()
    .in(UserStore.ID, UserStore.class)
    .doReaction((userStore, params) -> userStore.login(params));

when(UserStore.ID)
    .doAction(UserStore.ID.LOGIN_SUCCESS)

```

```

        .then()

        .in(R.id.credentials, LoginViewModel.class)

        .doReaction((loginViewModel, params) ->
loginViewModel.hideLoading());

```

Or do it, using the when(.) operator the following way.

```

        when(R.id.credentials, UserStore.ID)

        .doActions(LoginViewModel.LOGIN_REQUEST, UserStore.LOGIN_SUCCESS)

        .then()

        .doForwardReaction(UserStore.class, (userStore, params) ->
userStore.login(params))

        .doBackwardReaction(LoginViewModel.class, (loginViewModel, params)
-> loginViewModel.hideLoading());

```

Using when(.) operator, needs both components involved in the communication to be defined. The doActions() operator will take all actions to be listened to from each component. The doForwardReaction() operator will trigger the first reaction related to the first action specified inside the doActions() operator, [LoginViewModel.LOGIN_REQUEST] the doBackwardReaction() will trigger a second reaction related to the second action specified [UserStore.LOGIN_SUCCESS].

The sample above shows how the LoginViewModel component performs de LOGIN_REQUEST action and then executes the forward operation, and once the UserStore component emits a LOGIN_SUCCESS, the backward reaction is executed.

Channels

Channels are the communication engine among channels, each action performed by a component will be communicated (carrying data as a payload when neccessary) using channels.

If two components belong to the same page, there is no need to declare a channel. Communication could be implemented just using the when() operators that actually, will internally create a channel. In case those components belong to different pages, channels must be declared.

A channel storing primitive, Bundle or Parcelable values, could be defined the following way:

```
Channel<String> usernameChannel = new Channel<String>("username-  
channel");
```

In case, a different type is needed for a channel, it should be defined the following way (saving and restoring the channel content by the time the OS decides to kill the application).

```
Channel<Session> sessionChannel = new ChannelBuilder<Session>("session-  
channel")  
    .replay()  
    .onSave((outState, session) ->  
outState.putSerializable("channel-session-content", session))  
    .onRestore(inState -> (Session)inState.getSerializable("channel-  
session-content"));
```

To perform communications between web and native components, there are specific channels to make this happen, WebChannel and BundleWebChannel.

Using WebChannel needs an specification about how to save and retrieve the data as well as an adapter to handle the data sent from the web into the channel depending on its type.

```
Channel<Session> sessionChannel = new  
WebChannel.Builder<Session>("session-channel")  
    .toWebObject(session -> Collections.singletonMap("tsec",  
session.getTsec())) // Session to Map. From Native to Web  
    .toNativeObject(params -> params.getString("tsec")) // Bundle  
to Session. From Web to Native  
    .create();
```

This is a WebChannel, but Bundle typed, this kind of channel does not need an adapter implementation.

```
BundleWebChannel sessionChannel = new
```

```
BundleWebChannel("sessionChannel");
```

Clean channel

To use channels among pages, they should be declared as Replay, this is the difference between those declared automatically to communicate components in the same page. This means, everytime an Activity executes its onStart method, the defined Reaction will be executed again, in order to avoid this behavior repetition, the following operation could be considered:

```
reactTo(sessionChannel)
```

```
.then()
```

```
.in(UserStore.ID, UserStore.class)
```

```
.doReactionAndCleanChannel((userStore, session) ->
```

```
userStore.getCustomerData(session));
```

Clean All channels

Channels will hold information that is likely to be cleaned. To do so, there are two methods available:

- cleanAllChannels -> cleans all the channels
- cleanChannel(String channelName) -> cleans the specified channel

```
AppLifecycle appLifecycle = ...
```

```
appLifecycle.cleanAllChannels();
```

```
// or
```

```
appLifecycle.cleanChannel(name);
```

Fragment Support

To combine page reactions and fragments instead of use a meta-data you should add the tag in the replace method.

```
getSupportFragmentManager().beginTransaction()
```

```

        .replace(R.id.container, new WelcomeFragment(), /* <TAG> */
WelcomeFragment.ID /* </TAG> */)

        .commit();

```

Then, in the app class.

```

reactionManager.add(WelcomeFragment.ID, () -> new WelcomeReaction());

```

Hybrid zones

To combine web components with native components in a single page you should create a custom WebActivity and use the CellsWebView component.

```

public class DashboardActivity extends WebViewActivity {

```

```

    public static final String ID = "dashboard";

```

```

    @Override

```

```

    protected int getLayout() {

```

```

        return R.layout.activity_dashboard;

```

```

    }

```

```

}

```

```

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"

```

```

    xmlns:app="http://schemas.android.com/apk/res-auto"

```

```

    android:orientation="vertical"

```

```

    android:layout_width="match_parent"

```

```

    android:layout_height="match_parent">

```

```

    <com.bbva.cellscore.web.component.CellsWebView

```

```

        android:layout_width="match_parent"

```

```
android:layout_height="match_parent"/>
```

```
<!-- Native Components -->
```

```
</FrameLayout>
```

To create a PageReaction to the custom WebActivity you should extend the WebViewReaction class instead of PageReaction class.

```
public class CustomReaction extends WebViewReaction {
```

```
    @Override
```

```
    protected void configure() {
```

```
        super.configure();
```

```
        // ...
```

```
    }
```

```
}
```

And register de new PageReaction

```
PageReactionManager.add(DashboardActivity.ID, CustomReaction::new);
```

- Warning

If you don't need add custom reactions you should register the WebViewReaction class.

```
PageReactionManager.add(DashboardActivity.ID, WebViewReaction::new);
```

when - Custom Web Navigation

In order to perform native to web screen navigation with custom activity, when() function should be defined the following way:

```
when(R.id.welcome)
```

```
    .doAction(LoginWelcomeViewModel.ACCESS_TAPPED)
```

```
.then()
```

```
.doWebNavigation("dashboard", new Route(DashboardActivity.class));
```

```
// Or with animation
```

```
.doWebNavigation("dashboard", new Route(DashboardActivity.class,
```

```
R.anim.slide_left_in, R.anim.slide_left_out));
```

when - WebNavigation

There is another way of creating the navigation operations, using WebNavigation class.

```
Route nativeContainer = new Route(DashboardActivity.class);
```

```
final WebNavigation webNavigation = new WebNavigation.Builder()
```

```
.route(route)
```

```
.nativeContainer(nativeContainer)
```

```
.build();
```

```
when(R.id.welcome)
```

```
.doAction(LoginWelcomeViewModel.ACCESS_TAPPED)
```

```
.then()
```

```
.doWebNavigation(webNavigation);
```

when - Custom web loader

To avoid the webview's white screen when still loading, a splash view could be set. That view will be gone by the time the webview is full loaded.

```
Route route = new Route.Builder()
```

```
.activity(CustomWebFragmentActivity.class)
```

```
.animationIn(com.bbva.cellscore.R.anim.slide_up_in)
```

```
.loadingLayout(R.layout.web_view_loading)
```

```
.build();
```

```

when (R.id.welcome)

    .doAction(LoginWelcomeViewModel.ACCESS_TAPPED)

    .then()

    .doWebNavigation("dashboard", route);

```

when - Web navigation options

There is another way of creating the navigation options, using WebNavigation class.

```

NavigationOptions options = new NavigationOptions.Builder()

    ...

    build();

final WebNavigation webNavigation = new WebNavigation.Builder()

    .route(route)

    .nativeContainer(target)

    .navOptions(NavigationOptions)

    .build();

when (R.id.welcome)

    .doAction(LoginWelcomeViewModel.ACCESS_TAPPED)

    .then()

    .doWebNavigation(webNavigation);

```

Navigation options

Option

Description

Option	Description
skipHistory	Clears the history navigation stack

You must to include this option over the navigation to the destination web page, where you want to invalidate the back transition. For example, if you want to navigate from Login to Dashboard's page, you will include this option over Dashboard's navigation, not over the Login one.


```
NavigationOptions options = new NavigationOptions.Builder()
```

```
//...
```

```
.build();
```

```
// or
```

```
NavigationOptions options = NavigationOptions.skipHistory(true)
```

Set invalid navigations to back transitions.

Different back transitions might be established from the native side as invalid depending on the native business logic.

Parameters Description

routes	Array of SkipTransitionsDTO objects.
from	Web page identifier, which is the transition's origin that will be tagged as valid or not depending on the skipHistory's value.
to	Web page identifier, which is the transition's destination that will be tagged as valid or not depending on the skipHistory's value.
skipHistory	Bool to identify the transition as invalid (true) or valid (false), this value will be false by default.

```
WebController webController = ComponentManager.get(WebController.ID);
```

```
List<SkipTransitionDTO> navigations = new ArrayList<>();
```

```
navigations.add(new SkipTransitionDTO("login", "dashboard", true));
```

```
navigations.add(new SkipTransitionDTO("dashboard", "account", true));
```

```
webController.skipNavigations(navigations);
```

WebController

Use the webController component to trigger a web navigation to some route.

NavigateTo

```
when (UserStore.ID)

    .doAction (UserStore.ID.LOGIN_SUCCESS)

    .then ()

    .in (WebController, WebController1.class)

    .doReaction ((webController, params) ->

webController.navigateTo (...));
```

Send message to web

Use the webController component to trigger a web message with some payload.

```
when (UserStore.ID)

    .doAction (UserStore.ID.LOGIN_SUCCESS)

    .then ()

    .in (WebController, WebController.class)

    .doReaction ((webController, params) ->

webController.sendMessageToWeb (channelName, payload));
```

Animations

To open a web screen or a hybrid screen you could use one of the default animations or you could create a custom animation.

- Default Animation
 - Fade in/out: R.anim.fade_in , R.anim.fade_out
 - Zoom in/out: R.anim.zoom_in , R.anim.zoom_out
 - Slide up: R.anim.slide_up_in , R.anim.slide_up_out
 - Slide down: R.anim.slide_down_in , R.anim.slide_down_out
 - Slide right: R.anim.slide_right_in , R.anim.slide_right_out
 - Slide left: R.anim.slide_left_in , R.anim.slide_left_out
 - No animation: R.anim.none

Animation code example (R.anim.slide_left_in):

```
<?xml version="1.0" encoding="utf-8"?>

<set xmlns:android="http://schemas.android.com/apk/res/android">
```

```

<translate
xmlns:android="http://schemas.android.com/apk/res/android"

    android:interpolator="@android:anim/decelerate_interpolator"

    android:fromXDelta="100%p" android:toXDelta="0"

    android:duration="@android:integer/config_mediumAnimTime"/>

</set>

```

Standard properties dynamically

How to configure components standard properties dynamically

One of the main Cells Component Kit features is the possibility to dinamically configure the standard properties of its components dynamically.

Commons

To configure the standard properties you should use one of the following statement when available.

Literal

```

{
    "type": "literal"
    "value": "#000000"
}

```

Resource

```

{
    "type": "resource",
    "value": "black" // R.color.black
}

```

View

Standart properties to: LinearLayouts, RelativeLayout, TextView, ImageView...

Enable

```
"view": {  
  "enable": {  
    "value": true  
  }  
}
```

Visibility

Values: [gone | invisible | visible]

```
"view": {  
  "visibility": {  
    "value": gone  
  }  
}
```

Background Color

```
"view": {  
  "background-color": {  
    "type": "literal",  
    "value": "#00ff00"  
  }  
}
```

Background Color State

```
"view": {  
  "background-color-state": {  
    "type": "resource",  
    "value": "background_state" // R.drawable.background_state  
  }  
}
```

```
}
```

TextView or EditText

Standart properties to TextView and EditText

Text

```
"textView": {  
  "text": {  
    "type": "resource"  
    "value": "message" // R.string.message  
  }  
}
```

Input type

Values: [text, numeric]

```
"textView": {  
  "input-type": {  
    "value": "numeric"  
  }  
}
```

Text Color

```
"textView": {  
  "text-color": {  
    "type": "resource"  
    "value": "blue" // R.color.blue  
  }  
}
```

Text Color State

```
"textView": {
```

```
    "text-color-state": {  
      "type": "resource"  
      "value": "selector_color_state" //  
      R.color.selector_color_state  
    }  
  }  
}
```

Text Font

```
    "textView": {  
      "text-font": {  
        "type": "resource"  
        "value": "my_font" // R.font.my_font  
      }  
    }  
  }  
}
```

Text hint

```
    "textView": {  
      "text-hint": {  
        "type": "literal"  
        "value": "User"  
      }  
    }  
  }  
}
```

Text hint color

```
    "textView": {  
      "text-hint-color": {  
        "type": "literal"  
        "value": "#00FF0F"  
      }  
    }  
  }  
}
```

Text hint color state

```
"textView": {  
    "text-hint-color-state": {  
        "type": "resource"  
        "value": hint_state" // R.color.hint_state  
    }  
}
```

Text Size

```
"textView": {  
    "text-size": {  
        "type": "resource"  
        "value": "text_size" // R.dimen.text_size  
    }  
}  
  
"textView": {  
    "text-size": {  
        "type": "literal",  
        "unit": "sp", // ["sp", "dp", "px"]  
        "value": 15  
    }  
}
```

Image View

Standart properties to imageView

Scale

Values: ["center", "center-crop", "fit-xy", "fit-center"]

```
"imageView": {
```

```
"scale": {
```

```
  "value": "center"
```

```
}
```

```
}
```

Source

```
"imageView": {
```

```
  "source": {
```

```
    "type": "resource"
```

```
    "value": "ic_open" // R.drawable.ic_open
```

```
  }
```

```
}
```

Component DM / DP

How to create a Data Manager component

Data Manager components are in charge of hooking UI components with the outside world for example performing operations such as network requests and so on.

Setup

First of all, you should configure artifactory's configuration [Artifactory's Configuration](#)

Now, in build.gradle file:

```
allprojects {
```

```
  repositories {
```

```
    // ...
```

```
    maven {
```

```
      url 'https://globaldevtools.bbva.com/artifactory-api/cells-
```

```
native'
```

```
      credentials {
```

```
        username artifactory_user
```



```

        password artifactory_password
    }
}
}
}
}

```

And in component\build.gradle

```

        provided 'com.bbva.cells:component-kit: <CURRENT_VERSION>'
    }
}

```

Data Manger Component Inner Architecture

The development flow to be followed when using Cross Components seeking the highest reusability is: *Store > UseCase > Repository

Each functionality will require an UseCase that communicates with data repositories through a Repository [Repository pattern](#).

UseCase

In order to implement a custom UseCase, UseCase class must be extended and call method should be overwritten in order to get the data source. UseCase class will be in charge of triggering the operation (call method) in a background thread and return the result in the UI thread transparently.

```

    public class LoginUseCase extends UseCase<LoginResult, LoginRequestDTO>
    {

        private final UserRepository mUserRepository;

        private final LoginResultMapper mMapper;

        public LoginUseCase(UserRepository userRepository,
        LoginResultMapper mapper) {

            mUserRepository = userRepository;

            mMapper = mapper;
        }
    }

```

```
}
```

```
@Override
```

```
protected LoginResult call(LoginRequestDTO params) throws Exception
```

```
{
```

```
    LoginDTO loginDTO = mUserRepository.login(params);
```

```
    return mMapper.toData(loginDTO);
```

```
}
```

```
}
```

Or with RxJava.

```
public class LoginUseCase extends UseCase<LoginResult, LoginRequestDTO>
```

```
{
```

```
    private final UserRepository mUserRepository;
```

```
    private final LoginResultMapper mMapper;
```

```
    public LoginUseCase(UserRepository userRepository,
```

```
        LoginResultMapper mapper) {
```

```
        mUserRepository = userRepository;
```

```
        mMapper = mapper;
```

```
}
```

```
@Override
```

```
protected Flowable<LoginResult> create(LoginRequestDTO params) {
```

```
    return Flowable.fromCallable(() ->
```

```
        mUserRepository.login(params))
```

```
.map(mMapper::toData);
```

```
}
```

```
}
```

Store

```
public class UserStore {
```

```
    public static final String ID = "component-data-manager-login-user-  
store";
```

```
    private final LoginUseCase mLoginUseCase;
```

```
    public UserStore(...) {
```

```
        mLoginUseCase = new LoginUseCase(userRepository, new  
LoginResultMapper());
```

```
}
```

```
    public void login(LoginParams params) {
```

```
        mLoginUseCase.execute(loginRequestDTO,
```

```
                                (result) -> onLoginSuccess(result),
```

```
                                (error) -> onLoginError(error));
```

```
}
```

```
}
```

```
public class UserStore {
```

```
    public static final ActionSpec<LoginResult> LOGIN_SUCCESS = new  
ActionSpec<>("login-success");
```

```

        public static final ActionSpec<Throwable> LOGIN_ERROR = new
ActionSpec<>("login-error");

//...

    private void onLoginSuccess(LoginResult result) {
        Reactor.in(ID)
            .doAction(LOGIN_SUCCESS)
            .with(result)
            .fire();
    }

    private void onLoginError(Throwable error) {
        Reactor.in(ID)
            .doAction(LOGIN_ERROR)
            .with(error)
            .fire();
    }
}

```

IOS APP

Navigation

How to develop iOS screens within an hybrid architecture

Cells Hybrid

Cells Hybrid is an architecture proposal that allows developing applications by componentization, this approach means native components reusability in screens creation and faster application development.

This document is meant to walk you through a step by step Cells Hybrid project for iOS platforms guide. It will be explained in detail how to configure an iOS project in order to use this development approach. Once the project is configured it will be shown how to begin with hybrid screen creation and components communication.

This proposal provides a layer communication component in order to allow message-passing within web layer components and native layer components as well as managing navigations through different screens no matter the layer they belong.

It also provides component aspect and behavior customization.

Cells Hybrid is made of several modules:

- Action-Reaction component communication

- UI components

- Data managers

[Configuration](#)

First of all, you should configure artifactory's configuration [Artifactory's Configuration](#)

[Project Configuration](#)

Hint: In case you are already using the 'cells-cli' in order to scaffold a new project, you may skip this step and focus directly on 'First Page' step.

[Artifactory Setup](#)

In order to add Cells Native Core to your Xcode project using CocoaPods, add the dependency in your Podfile:

```
plugin 'cocoapods-art', :sources => ['cells-native-cocoapods', 'master']
```

```
platform :ios, '8.0'
```

```
target 'TargetName' do
```

```
  pod 'CellsNativeCore', '~> 2.0'
```

```
end
```

App Configuration

Next step is configuring the Application class.

A RouterFactory must be created, it should be provided with all the routes linked to the view controllers so it will be able to manage launching them from web components. In this way, a component will perform the opening a screen operation using this route as reference.

In this scenario we will be developing an application that will contain a LoginViewController and a DashBoardViewController.

```
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?
    var routerFactory: RouterFactory?

    func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {

        self.routerFactory = RouterFactoryApp.init()

        return true
    }
}
```

Native Routing - needs RouterFactory class implementation

```
class RouterFactoryApp: RouterFactory {

    override init() {

        super.init()

        let toLoginRoute: Route = Route.init(viewController:
{ () -> UIViewController in
            return LoginViewController()
        }, animationToNext: { (originViewController,
destinationViewController) in
            //Here we could define the way to
        }, animationToPrevious: { (originViewController,
destinationViewController) in
```

```

        })

        let toDashboardRoute: Route =
Route.init(viewController: { () -> UIViewController in
            return DashboardViewController()
        }, animationToNext: { (originViewController,
destinationViewController) in

            }, animationToPrevious: { (originViewController,
destinationViewController) in

            })

        self.addRoute(route: LoginViewController.ROUTE,
routeObject: toLoginRoute)
        self.addRoute(route: DashboardViewController.ROUTE,
routeObject: toDashboardRoute)
    }

}

```

Once the router is correctly configured, AppLifecycle must be initialized:

```

class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?
    var appLifeCycle: AppLifecycle?
    var routerFactory: RouterFactory?

    func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // ...

        self.routerFactory = RouterFactoryApp.init()

        self.appLifeCycle = AppLifecycle.Builder { [weak
self] builder in
            builder.app = UIApplication.shared
            builder.routerFactory = self?.routerFactory
            builder.pageReactionManager =
PageReactionManager.init()
        }.create()
    }
}

```

```

        return true
    }
}

```

First Page

In order to create our first screen made out from components we must declare an view controller using an unique identifier as property of this view controller:

```

class LoginViewController: CellsBaseViewController {

    static let ID: String = "my-unique-id"

    //...
}

```

Each component event-driven behavior definition will be defined inside a **PageReaction** class:

```

class LoginViewControllerPageReaction: PageReaction {
}

```

PageReactions must be added to our Application class making reference to the View Controller in which they are executed.

```

class AppDelegate: UIResponder, UIApplicationDelegate {

    // ...

    var pageReactionManager: PageReactionManager =
PageReactionManager.init()

    func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {

        // ...

        self.appLifecycle = AppLifecycle.Builder { [weak
self] builder in
            // ...

```



```

        builder.pageReactionManager =
self?.pageReactionManager
        }.create()

        self.pageReactionManager.add(id:
LoginViewController.ID) { () -> PageReaction in
            LoginViewControllerPageReaction.init()
        }

        return true
    }
}

```

Actions-Reactions

Each component **emmits a collection of events** that could be observed and triggers reactions in other components from them. To do so, following expresions are available for component reacting:

when - Communication

The following example shows how a component triggers an event and its reaction in another component using when command:

```

class LoginViewControllerPageReaction: PageReaction{

    override func configure(){

        let _ = when(id: LoginHeaderView.ID, type:
Bool.self)

                .doAction(actionSpec:
LoginHeaderViewModel.ON_ANIM_FINISH)
                .then()
                .inside(componentID: LoginViewController.ID,
component: LoginViewController.self)
                .doReaction(reactionExecution: { (component,
value) in

                    component.enableTextField(value)

                })
    }
}

```

To be more specific, in this code, two different components are interacting. LoginHeaderViewModel component is triggering a ON_ANIM_FINISH event and as a consequence LoginViewController will react the way it is defined, in this case performing the textField enable to write into it (`controller.enableTextField(value)`)

when - Navigation

The following example shows how a component triggers a navigation event and its reaction using when command:

```
let _ = when(id: LoginViewController.ID, type: Any.self)
    .doAction(actionSpec:
LoginViewController.ACTION_OPEN_NEXT_NATIVE_PAGE)
    .then()
    .inside(componentID: LoginViewController.ID,
component: LoginViewController.self)
    .doNavigation { (viewController: UIViewController,
component: ViewController, value: Any) in

        viewController.navigationController?

.pushViewController(destinationViewController, animated:true)

        // Or with stored animation located in
RouterFactory

        let routeToNextViewController: Route =
((UIApplication.shared.delegate as! AppDelegate)
        .routerFactory?
        .getRoute(route:
DashboardViewController.ROUTE, defaultValue: ""))!

        let destinationViewController =
routeToNextViewController.getViewController()

        viewController
            .navigationController?

.pushViewController(destinationViewController, animated:false)

        routeToNextViewController
            .getAnimationToNext()!(viewController,
destinationViewController)
    }
```

when - Web Navigation

In order to perform native to web screen navigations, `when()` function should be defined the following way:

```
// With the default animation to show and hide the
WebViewController

let routeToWebViewController: Route =
((UIApplication.shared.delegate as!
AppDelegate).routerFactory?.getRoute(route:
WebViewController.ROUTE, defaultValue: ""))!

// Or with the expesified animation

let routeToWebViewController: Route =
Route.init(viewController: { () -> UIViewController in
    return ((UIApplication.shared.delegate as!
AppDelegate)
        .routerFactory?
        .getRoute(route: WebViewController.ROUTE,
defaultValue: ""))!
        .getViewController()
    }, animationToNext: { (originViewController,
destinationViewController) in
    //Animation from OriginViewController to
DestinationViewController is the next in the navigation flux
    }) { (originViewController, destinationViewController)
in
    //Animation from OriginViewController to
DestinationViewController where DestinationViewController is the
previous in the navigation flux
    }

let _ = when(id: LoginViewController.ID, type:
Void.self)
    .doAction(actionSpec:
ViewController.ACTION_OPEN_WEB_PAGE)
    .then()
    .doWebNavigation(webRoute: "dashboard",
nativeRoute: routeToWebViewController)
```

when - Multiple Reactions

Multiple Reactions may be defined to be triggered from a single action using `and()` operator.

```
class LoginViewControllerPageReaction: PageReaction{

    override fun configure(){

        let _ = when(id: LoginHeaderView.ID, type:
Bool.self)

                .doAction(actionSpec:
LoginHeaderViewModel.ON_ANIM_FINISH)
                .then()
                .inside(componentID: LoginViewController.ID,
component: LoginViewController.self)
                .doReaction { (component, value) in
                    controller.enableTextField(value)
                }
                .and()
                .inside(componentID: LoginFooterView.ID,
component: LoginFooterView.self)
                .doReaction { (component, value) in
                    controller.enablePromotionsButton(value)
                }
            }
    }
}
```

writeOn

In order to use component emitted data in another component from another page, `writeOn()` operator should be used.

So, communicating different components in different pages involves a communication channel creation to store the data we want to communicate. For example, if we want to communicate a native to a web component we'll use the `WebChannel`. For further information about channels, please visit the 'channels' section in this document.

```
let dataUserChannel: Channel = JsonChannel("data-user-
channel")
```

A `JsonChannel` is a specialization of `WebChannels` that contains elements of type `Any`.

```

class LoginViewControllerPageReaction: PageReaction{

    override func configure(){
        let _ = writeOn(channel: dataUserChannel)
            .when(componentID: LoginViewController.ID)
            .doAction(actionSpec:
LoginViewController.DATA_USER_CHANGED)
    }
}

```

In this case, the operator waterfall describes a situation in which by the time the LoginViewController emits the 'DATA USER CHANGED', this value should be written inside the dataUserChannel.

reactTo

In order for a page to read data from an specific channel, reactTo operator must be used.

```

class DashboardViewControllerPageReaction: PageReaction{

    override func configure(){
        let _ = reactTo(channel: dataUserChannel)?
            .then()
            .inside(componentID:
DashboardViewController.ID, component:
DashboardViewController.self)
            .doReaction(reactionExecution: { (component,
param) in
                component.updateDataUser(param)
            })
    }
}

```

Using this operator, adds a reaction in a component so this reaction is executed when the channel (this component is bound to) content changes. In this case, when de data inside dataUserChannel changes.

Channels

Channels are the communication engine among channels, each action performed by a component will be communicated (carrying data as a payload when necessary) using channels.

If two components belong to the same page, there is no need to declare a channel. Communication could be implemented just using the `when()` operators that actually, will internally create a channel. In case those components belong to different pages, channels must be declared.

A channel storing primitive could be defined the following way:

```
let dataUserChannel: Channel<String> =  
Channel<String>("username-channel")
```

To perform communications between web and native components, there are specific channels to make this happen, `WebChannel`.

Using `WebChannel` needs an specification about how to save and retrieve the data as well as an adapter to handle the data sent from the web into the channel depending on its type.

```
let userObjectChannel: Channel<UserObject> =  
WebChannel<UserObject>.init(name: "username-channel",  
bundleAdapter: { (serializableData) -> UserObject in  
    // Transform serializable data into internal model  
    data object  
    }, mapAdapter: { (userObjectInstance) -> Any in  
    // Transform internal model data object into  
    serializable data  
    })
```

This is a `WebChannel`, but `Json` typed, this kind of channel does not need an adapter implementation.

```
let dataUserChannel: Channel = JsonChannel("data-user-  
channel")
```

Clean channel

To use channels among pages, they should be declared as `Replay`, this is the difference between those declared automatically to communicate components in the same page. This means, everytime a View Controller executes its `viewWillAppear`

method, the defined Reaction will be executed again, in order to avoid this behavior repetition, the following operation could be considered:

```
let _ = reactTo(channel: dataUserChannel)?  
    .then()  
    .inside(componentID: DashboardViewController.ID,  
component: DashboardViewController.self)  
    .doReactionAndClearChannel(reactionExecution: {  
(component, param) in  
        component.updateDataUser(param)  
    })
```

Clean All channels

Channels will hold information that is likely to be cleaned. To do so, there are two methods available:

- `cleanAllChannels` -> cleans all the channels
- `cleanChannel(String channelName)` -> cleans the specified channel

```
let appLifecycle: AppLifecycle = ...  
  
appLifecycle.cleanAllChannels()  
  
// or  
  
appLifecycle.cleanChannel(name)
```

Hide CellsView if needed

There are situations where will be necessary hide the CellsView shown in the top of the window's subviews, from native side. For example if you develop the logout timer knowledge in the native side.

For this kind of situations, our CellsBaseViewController instance have a solution:

```
func hideCellsViewIfNeededForWebViewZones()
```

If you couldn't extend our CellsBaseViewController, you must implement content of this method in your BaseViewController class.

Hybrid zones

To combine web components with native components in a single page you should create a custom View Controller and use the CellsWebView component.

```

class DashboardViewController: CellsBaseViewController {

    static let ROUTE: String = "dashboard"
    static let ID: String = "my-dashboard-unique-id"
    //...

    override func viewDidLoad() {

        self.configure(appLifecycle:
(UIApplication.shared.delegate as! AppDelegate).appLifeCycle!,
id:HibrydZoneFullScreenViewController.ID)
        super.viewDidLoad()
        self.navigationController?.navigationBar.isHidden =
false

    }
}

```

And we have to add to our xib file this CellsWebView

```

<view customClass="CellsWebView" customModule="CellsNativeCore">

</view>

```

To create a PageReaction to the hybrid Zone View Controller you should extend the PageReaction class as well you guess for a normal page.

```

class DashboardViewControllerPageReaction: PageReaction {
    override func configure(){

    }

}

```

And register de new PageReaction

```

self.pageReactionManager.add(id: DashboardViewController.ID)
{ () -> PageReaction in
    DashboardViewControllerPageReaction.init()
}

```


WebController

Use the WebController component to trigger a web navigation to some route.

NavigateTo

```
let _ = when(id: LoginHeaderView.ID, type: Void.self)
    .doAction(actionSpec:
LoginHeaderViewModel.ON_ANIM_FINISH)
    .then()
    .inside(componentID: WebController.ID, component:
WebController.self)
    .doReaction { (component, value) in
        controller.navigateTo(route: "dashboard")
    }
```

Send message to web

Use the WebController component to trigger a web message with some payload.

```
let _ = when(id: DashboardViewController.ID, type:
Dictionary.self)
    .doAction(actionSpec:
DashboardViewControllerPageReaction.EVENT_CELL_LOGIN_SUCCESS)
    .then()
    .inside(componentID: WebController.ID, component:
WebController.self)
    .doReaction { (controller, param) in

        controller.sendMessageToWeb(channel: "update-
user-data-channel", payload: param)

    }
```

when - Web navigation options

In addition, you could use some navigation options when you send navigateTo messages to WebController object.

Navigation options

Option

Description

skipHistory	Clears the history navigation stack
-------------	-------------------------------------

You must to include this option over the navigation to the destination web page, where you want to invalidate the back transition. For example, if you want to navigate from

Login to Dashboard's page, you will include this option over Dashboard's navigation, not over the Login one.

```
controller.navigateTo(route: "dashboard", payload: "",  
[skipHistory: true])
```

Manage back actions from Hybrid Zone to Native

It's possible to get the control when in web layout a back action to native is tapped. The way is implementing the protocol `TransitionAnimation`:

```
extension DashboardViewController: TransitionAnimation{  
    func animateFinish() {  
        // If we have a Navigation Controller that manage our  
flow  
        self.navigationController?.popViewController(animated:  
true)  
    }  
}
```

If you want to invalidate a group of transitions, the next `WebController` method should be implemented.

Set invalid navigations to back transitions.

Different back transitions might be established from the native side as invalid depending on the native business logic.

Parameters Description

routes	Array of <code>SkipTransitionsDTO</code> objects.
from	Web page identifier, which is the transition's origin that will be tagged as valid or not depending on the <code>skipHistory</code> 's value.
to	Web page identifier, which is the transition's destination that will be tagged as valid or not depending on the <code>skipHistory</code> 's value.
skipHistory	Bool to identify the transition as invalid (true) or valid (false), this value will be false by default.

```
let controller: WebController = ComponentManager.get(id:  
WebController.ID)
```

```
controller.skipNavigations(routes: [SkipTransitionDTO.init(from:
"login", to: "dashboard", skipHistory: true),
SkipTransitionDTO.init(from: "dashboard", to: "account",
skipHistory: true), ...])
```

Alternative to extend CellsBaseViewController in all your View Controller

If your development requirements avoid your development extends in the UIViewControllers of your project any different class like our CellsBaseViewController we have another way for you. You'll have to implement in your BaseViewController's UIViewController life cycle protocol the next calls to AppDelegate:

```
open class YourBaseViewController: UIViewController {

    override open func viewDidLoad() {
        super.viewDidLoad()
        self.appLifecycle?.didLoad(viewController: self, id:
self.id)
    }

    override open func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        self.appLifecycle?.willAppear(viewController: self,
id: self.id)
    }

    override open func viewWillDisappear(_ animated: Bool) {
        super.viewWillDisappear(animated)
        self.appLifecycle?.willDisappear(viewController:
self, id: self.id)
    }

    override open func viewDidDisappear(_ animated: Bool) {
        super.viewDidDisappear(animated)
        self.appLifecycle?.didDisappear(viewController:
self, id: self.id)
    }

    deinit {
        self.appLifecycle?.onDestroy(viewController: self,
id: self.id)
    }

}
```