FLOWX.AI

PLATFORM DEEP DIVE / Core components / Renderer SDKs

Contents

PLATFORM DEEP DIVE / Core components / Renderer SDKs / Using the Angular Renderer

FlowxProcessRenderer is a low code library designed to render UI configured via the Flowx Process Editor.

Angular project requirements

Your app MUST be created using the NG app from the @angular/cli~15 package. It also MUST use SCSS for styling.

```
npm install -g @angular/cli@15.0
ng new my-flowx-app
```

> **INFO**

To install the npm libraries provided by FLOWX you will need to obtain access to the private FLOWX Nexus registry. Please consult with your project DevOps.

CAUTION

The library uses Angular version **@angular~15**, **npm v8.1.2** and **node v16.13.2**.

CAUTION

If you are using an older version of Angular (for example, v14), please consult the following link for update instructions:

Update Angular from v14.0 to v15.0

Installing the library

Use the following command to install the **renderer** library and its required dependencies:

```
npm install @flowx/ui-sdk@3.21.0
@flowx/ui-toolkit@3.21.0
@flowx/ui-theme@3.21.0
paperflow-web-components
vanillajs-datepicker@1.3.1
moment@^2.27.0
@angular/flex-layout@15.0.0-beta.42
@angular/material@15.2.0
```

```
@angular/material-moment-adapter@15.2.0
@angular/cdk@15.2.0
ng2-pdfjs-viewer@15.0.0
event-source-polyfill@1.0.31
```

Also, in order to successfully link the pdf viewer, add the following declaration in the assets property of you project's angular.json:

```
{
  "glob": "**/*",
  "input": "node_modules/ng2-pdfjs-viewer/pdfjs",
  "output": "/assets/pdfjs"
}
```

Using the library

Once installed, FlxProcessModule will be imported in the `AppModule` `FlxProcessModule.forRoot({})`.

You MUST also import the dependencies of `FlxProcessModule: HttpClientModule` from `@angular/common/http` and **IconModule** from `@flowxai/ui-toolkit`.

Using Paperflow web components

Add path to component styles to stylePreprocessesOptions object in **angular.json file**

```
"stylePreprocessorOptions": {
  "includePaths": [
  "./node_modules/paperflow-web-components/src/assets/scss",
  "./node_modules/flowx-process-
renderer/src/assets/scss/style.scss",
  "src/styles"]
}
```

> **INFO**
>
> Because the datepicker module is build on top of angular material datepicker module, using it requires importing one predefined material theme in you **angular.json** configuration.

```
 "styles": ["...,
"./node_modules/@angular/material/prebuilt-themes/indigo-
pink.css"],
```

Theming

Component theming is done through two json files (`theme_tokens.json`, `theme_components.json`) that need to be added in the assets folder of your project The file paths need to be passed to the `FlxProcessModule.forRoot()` method through the `themePaths` object.

```
themePaths: {
    components: 'assets/theme/theme_components.json',
    tokens: 'assets/theme/theme_tokens.json',
  },
```

The **assets/theme/theme_tokens.json** - should hold the design tokens (e.g. colors, fonts) used in the theme. An example can be found here.

The **assets/theme/theme_components.json** - holds metadata used to describe component styles. An example can be found here.

For **Task Management** theming is done through the ppf-theme mixin that accepts as an argument a list of colors grouped under **primary**, **status** and **background**

```
@use 'ppf-theme';

@include ppf-theme.ppf-theme((
  'primary': (
    'color1': vars.$primary,
    'color2': vars.$secondary,
    'color3': vars.$text-color,
  ),
  'status': (
    'success': vars.$success,
    'warning': vars.$warning,
    'error': vars.$error,
  ),
  'background': (
    'background1': vars.$background1,
    'background2': vars.$background2,
    'background3': vars.$background3,
  ),
));
```

Authorization

INFO

Every request from the **FLOWX** renderer SDK will be made using the **HttpClientModule** of the client app, which means those requests will go through every interceptor you define here. This is most important to know when building the auth method as it will be the job of the client app to intercept and decorate the requests with the necessary auth info (eg. `Authorziation: Bearer ...`).

NOTE

It's the responsibility of the client app to implement the authorization flow (using the **OpenID Connect** standard). The renderer SDK will expect to find the **JWT** saved in the browser **localStorage** object at the key named access_token.

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import { HttpClientModule, HTTP_INTERCEPTORS } from
'@angular/common/http';
import {FlxProcessModule} from 'flowx-process-renderer';
import { IconModule } from 'paperflow-web-components';

import {AppRoutingModule} from './app-routing.module';
import {AppComponent} from './app.component';

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
```

```
    BrowserModule,
    AppRoutingModule,
    // will be used by the renderer SDK to make requests
    HttpClientModule,
    // needed by the renderer SDK
    IconModule.forRoot(),
    FlxProcessModule.forRoot({
      components: {},
      services: {},
      themePaths: {
        components: 'assets/theme/theme_components.json',
        tokens: 'assets/theme/theme_tokens.json',
      },
    }),
  ],
  // this interceptor with decorate the requests with the
Authorization header
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor,
multi: true },
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

The `forRoot()` call is required in the application module where the process will be rendered. The `forRoot()` method accepts a config argument where you can pass extra config info, register a **custom component**, **service**, or **custom validators**.

Custom components will be referenced by name when creating the template config for a user task.

Custom validators will be referenced by name (`currentOrLastYear`) in the template config panel in the validators section of each generated form field.

```
// example
FlxProcessModule.forRoot({
  components: {
    YourCustomComponentIdenfier: CustomComponentInstance,
  },
  services: {
    NomenclatorService,
    LocalDataStoreService,
  },
  validators: {currentOrLastYear },
})
```

```
# example with custom component and custom validator
FlxProcessModule.forRoot({
  components: {
    YourCustomComponentIdenfier: CustomComponentInstance,
  },
  services: {
    NomenclatorService,
    LocalDataStoreService,
  },
  validators: {currentOrLastYear },
})

  // example of a custom validator that restricts data
selection to
  // the current or the previous year

  currentOrLastYear: function currentOrLastYear(AC:
```

```
AbstractControl): { [key: string]: any } {
    if (!AC) {
      return null;
    }

    const yearDate = moment(AC.value, YEAR_FORMAT, true);
    const currentDateYear = moment(new
Date()).startOf('year');
    const lastYear = moment(new Date()).subtract(1,
'year').startOf('year');

    if (!yearDate.isSame(currentDateYear) &&
!yearDate.isSame(lastYear)) {
      return { currentOrLastYear: true };
    }

    return null;
  }
```

> **CAUTION**
>
> The error that the validator returns **MUST** match the validator name.

The component is the main container of the UI, which will build and render the components configured via the **FlowX Designer**. It accepts the following inputs:

```
<flx-process-renderer
  [apiUrl]="baseApiUrl"
  [processApiPath]="processApiPath"
  [processName]="processName"
  [processStartData]="processStartData"
  [debugLogs]="debugLogs"
```

```
  [keepState]="keepState"
  [language]="language"
></flx-process-renderer>
```

Parameters:

Name	Description	Type	Mandatory	Default value	
baseApiUrl	Your base url	string	true	-	https
processApiPath	Engine API prefix	string	true	-	/onb
processName	Identifies a process	string	true	-	clien
processStartData	Data required to start the process	json	true	-	{ "firs "last
debugLogs	When set to true this will print WS messages in the console	boolean	false	false	-

Name	Description	Type	Mandatory	Default value	
language	Language used to localize the application.	string	false	ro-RO	-
keepState	By default all process data is reset when the process renderer component gets destroyed. Setting this to true will keep process data even if the viewport gets destroyed	boolean	false	false	-

Name	Description	Type	Mandatory	Default value	
isDraft	When true allows starting a process in draft state. *Note that isDraft = true requires that processName be the **id** (number) of the process and NOT the name.	boolean	false	false	-

Data and actions

Custom components will be hydrated with data through the $data input observable which must be defined in the custom component class.

```
@Component({
  selector: 'my-custom-component',
  templateUrl: './custom-component.component.html',
  styleUrls: ['./custom-component.component.scss'],
})
```

```
export class CustomComponentComponent  {
  @Input() data$: Observable<any>;
  }
```

Component actions are always found under `data` -> `actionsFn` key.

Action names are configurable via the process editor.

```
# data object example
data: {
   actionsFn: {
      action_one: () => void;
      action_two: () => void; }
   }
```

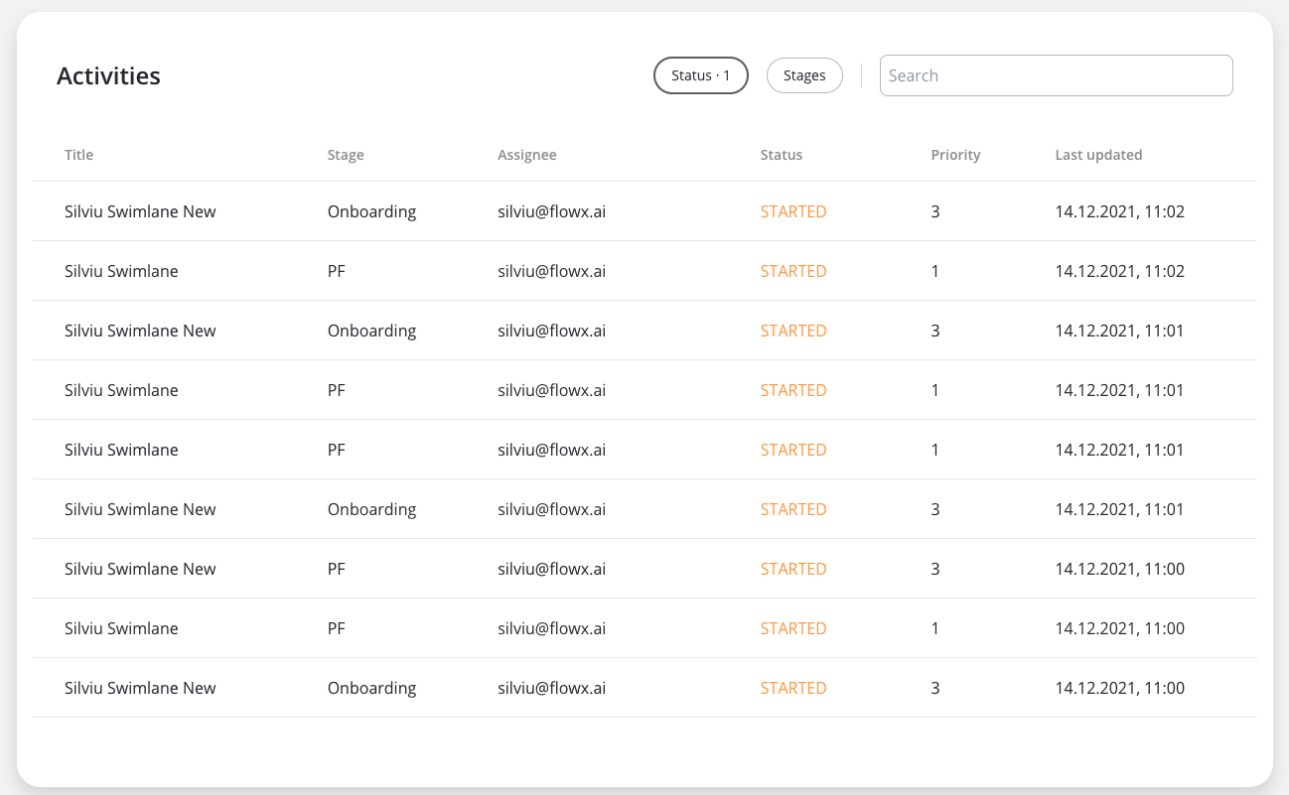
Interacting with the process

Data from the process is communicated via **SSE** protocol under the following keys:

Name	Description	Example	
Data	data updates for process model bound to default/custom components		
ProcessMetadata	updates about process metadata, ex: progress update, data about how to render components		

Name	Description	Example	
RunAction	instructs the UI to perform the given action		

Task management component



The flx-task-management component is found in the FlxTaskManagementModule. In order to have access to it, import the module where needed:

```
import {FlxTaskManagementModule} from 'flowx-process-
renderer';
@NgModule({
  declarations: [
```

```
    ...,
  ],
  imports: [
    ...,
    FlxTaskManagementModule
  ],

})
export class MyModule {
}
```

Then in the template:

```
<flx-task-management [baseUrl]="baseUrl" [title]="'Tasks'">
</flx-task-management>
```

Parameters:

Name	Description	Type	Default	Mandatory	
apiUrl	Endpoint where the tasks are available	string	-	true	https://yo
title	Table header value	string	Activities	false	Tasks

Name	Description	Type	Default	Mandatory	
pollingInterval	Interval for polling task updates	number	5000 ms	false	10000

Development

When modifying the library source code and testing it inside the designer app use the following command which rebuilds the flx-process-renderer library, recreates the link between the library and the designer app and recompiles the designer app:

```
npm run build && cd dist/flowx-process-renderer/ && npm link
&& cd ../../ && npm link flowx-process-renderer && npm run
start:designer
```

or alternatively run

`./start_with_build_lib.sh`

If you want to start the designer app and the flx-process-renderer library in development mode (no need to recompile the lib for every change) run the following command:

`npm run start:designer-dev`

> **CAUTION**

> Remember to test the final version of the code by building and bundling the renderer library to check that everything works e2e

Trying to use this lib with npm link from another app will most probably fail. If (when) that happens, there are two alternatives that you can use:

1. Use the build-and-sync.sh script, that builds the lib, removes the current build from the client app **node_modules** and copies the newly build lib to the node_modules dir of the client app:

```
./build-and-sync.sh ${path to the client app root}

# example (the client app is demo-web):
./build-and-sync.sh ../../demo-web
```

NOTE: This method uses under the hood the build-and-sync.sh script from the first version and the chokidar-cli library to detect file changes.

2. Use the build-and-sync:watch npm script, that builds the library and copies it to the client app's **node_module** directory every time a file changes:

```
npm run build-and-sync:watch --target-path=${path to the
client app root}

# example (the client app is demo-web):
npm run build-and-sync:watch --target-path=../../demo-web
```

Running the tests

```
ng test
```

Coding style tests

Always follow the Angular official coding styles.

Below you will find a Storybook which will demonstrate how components behave under different states, props, and conditions, it allows you to preview and interact with individual UI components in isolation, without the need for a full-fledged application:

» Storybook

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Renderer SDKs / Using the iOS Renderer

iOS Project Requirements

The minimum requirements are:

- iOS 14
- Swift 5.0

Installing the library

The iOS Renderer is available through Cocoapods and Swift Package Manager.

Swift Package Manager

In Xcode, click `File` → `Add Packages...`, enter FlowX repo's URL `https://github.com/flowx-ai/flowx-ios-sdk`. Set the dependency rule to `Up To Next Major` and add package.

If you are developing a framework and use FlowX as a dependency, add to your `Package.swift` file:

```
dependencies: [
    .package(url: "https://github.com/flowx-ai/flowx-ios-
sdk", .upToNextMajor(from: "0.96.0"))
]
```

Cocoapods

Prerequisites

- Cocoapods gem installed

Cocoapods private trunk setup

Add the private trunk repo to your local Cocoapods installation with the command:

```
pod repo add flowx-specs git@github.com:flowx-ai/flowx-ios-
specs.git
```

Adding the dependency

Add the source of the private repository in the Podfile

```
source 'git@github.com/flowx-ios-specs.git'
```

Add the pod and then run `pod install`

```
pod 'FlowX'
```

Library dependencies

The iOS Renderer library depends on the following libraries:

- Socket.IO-Client-Swift
- Alamofire
- SVProgressHUD
- SDWebImageSwiftUI

Configuring the library

The SDK has 2 configurations, available through shared instances.

It is recommended to call the configuration methods at app launch.

Otherwise, make sure you do it before the start of any FlowX process.

FXConfig

This config is used for general purpose properties.

Properties

Name	Description	Type	Requirement
baseURL	The base URL used for REST networking	String	Mandatory
imageBaseURL	The base URL used for media library images	String	Mandatory
language	The language used for retrieving enumerations and substitution tags	String	Mandatory. Defaults to "en"

Name	Description	Type	Requirement
stepViewType	The type of the custom step view class	FXStepViewProtocol.Type	Optional
logEnabled	Value indicating whether console logging is enabled. Default is false	Bool	Optional

Sample

```
FXConfig.sharedInstance.configure { (config) in
    config.baseURL = myBaseURL
    config.imageBaseURL = myImageBaseURL
    config.language = "en"
    config.logEnabled = true
    config.stepViewType = CustomStepView.self
}
```

FXSessionConfig

This config is used for providing networking or auth session-specific properties.

The library expects a session instance managed by the container app. Request adapting and retrying are handled by the container app.

Properties

Name	Description	Type
sessionManager	Alamofire session instance used for REST networking	Session
token	JWT authentication access token	String

Sample

```
FXSessionConfig.sharedInstance.configure { config in
    config.sessionManager = mySessionManager
    config.token = myAccessToken
}
```

Using the library

The library's public APIs are called using the shared instance of FlowX, `FlowX.sharedInstance`.

How to start and end FlowX session

After all the configurations are set, you can start a FlowX session by calling the `startSession()` method.

This is optional, as the session starts lazily when the first process is started.

`FlowX.sharedInstance.startSession()`

When you want to end a FlowX session, you can call the `endSession()` method. This also does a complete clean-up of the started processes.

You might want to use this method in a variety of scenarios, for instance when the user logs out.

`FlowX.sharedInstance.endSession()`

How to start a process

You can start a process by calling the method below.

The container app is responsible with presenting the navigation controller holding the process navigation.

```
public func startProcess(navigationController:
UINavigationController,
                         name: String,
                         params: [String: Any]?,
                         isModal: Bool = false,
                         showLoader: Bool = false)
```

`navigationController` - the instance of UINavigationController which will hold the process navigation stack

`name` - the name of the process

`params` - the start parameters, if any

`isModal` - a boolean indicating whether the process navigation is modally displayed. When the process navigation is displayed modally, a close bar button item is displayed on each screen displayed throughout the process navigation.

`showLoader` - a boolean indicating whether the loader should be displayed when starting the process.

Sample

```
FlowX.sharedInstance.startProcess(navigationController:
processNavigationController,
                                  name: processName,
                                  params: startParams,
                                  isModal: true
                                  showLoader: true)

self.present(processNavigationController, animated: true,
completion: nil)
```

How to resume a process

You can resume a process by calling the method below.

```
public func continueExistingProcess(uuid: String,
                                    name: String,
                                    navigationController:
UINavigationController,
                                    isModal: Bool = false) {
```

`uuid` - the UUID string of the process

`name` - the name of the process

`navigationController` - the instance of UINavigationController which will hold the process navigation stack

`isModal` - a boolean indicating whether the process navigation is modally displayed. When the process navigation is displayed modally, a close bar button item is displayed on each screen displayed throughout the process navigation.

How to end a process

You can manually end a process by calling the `stopProcess(name: String)` method.

This is useful when you want to explicitly ask the FlowX shared instance to clean up the instance of the process sent as parameter.

For example, it could be used for modally displayed processes that are dismissed by the user, in which case the `dismissRequested(forProcess process: String, navigationController: UINavigationController)` method of the FXDataSource will be called.

Sample

```
FlowX.sharedInstance.stopProcess(name: processName)
```

How to run an action from a custom component

The custom components which the container app provides will contain FlowX actions to be executed. In order to run an action you need to call the following method:

```
public func runAction(action: ProcessActionModel,
                      params: [String: Any]? = nil)
```

`action` - the `ProcessActionModel` action object

`params` - the parameters for the action

How to run an upload action from a custom component

```
public func runUploadAction(action: ProcessActionModel,
                            image: UIImage)
```

`action` - the `ProcessActionModel` action object

`image` - the image to upload

```
public func runUploadAction(action: ProcessActionModel,
                            fileURL: URL)
```

`action` - the `ProcessActionModel` action object

`fileURL` - the local URL of the image

Getting a substitution tag value by key

```
public func getTag(withKey key: String) -> String?
```

All substitution tags will be retrieved by the SDK before starting the first process and will be stored in memory.

Whenever the container app needs a substitution tag value for populating the UI of the custom components, it can request the substitution tag using the method above, providing the key.

Getting a media item url by key

```
public func getMediaItemURL(withKey key: String) -> String?
```

All media items will be retrieved by the SDK before starting the first process and will be stored in memory.

Whenever the container app needs a media item url for populating the UI of the custom components, it can request the url using the method above, providing the key.

```
public func getTag(withKey key: String) -> String?
```

All substitution tags will be retrieved by the SDK before starting the first process and will be stored in memory.

Whenever the container app needs a substitution tag value for populating the UI of the custom components, it can request the substitution tag using the method

above, providing the key.

Handling authorization token changes

When the access token of the auth session changes, you can update it in the renderer using the `func updateAuthorization(token: String)` method.

FXDataSource

The library offers a way of communication with the container app through the `FXDataSource` protocol.

The data source is a public property of FlowX shared instance.

`public weak var dataSource: FXDataSource?`

```
public protocol FXDataSource: AnyObject {
    func controllerFor(componentIdentifier: String) ->
FXController?

    func viewFor(componentIdentifier: String) -> FXView?

    func viewFor(componentIdentifier: String,
customComponentViewModel: FXCustomComponentViewModel) ->
AnyView?

    func navigationController() -> UINavigationController?

    func errorReceivedForAction(name: String?)

    func validate(validatorName: String, value: String) ->
Bool
```

```
    func dismissRequested(forProcess process: String,
navigationController: UINavigationController)

}
```

- `func controllerFor(componentIdentifier: String) -> FXController?`

This method is used for providing a custom component UIKit view controller, identified by the componentIdentifier argument.

- `func viewFor(componentIdentifier: String) -> FXView?`

This method is used for providing a custom UIKit view, identified by the componentIdentifier argument.

- `func viewFor(componentIdentifier: String, customComponentViewModel: FXCustomComponentViewModel) -> AnyView?`

This method is used for providing a custom SwiftUI view, identified by the componentIdentifier argument. A view model is provided as an ObservableObject to be added as @ObservedObject inside the SwiftUI view.

- `func navigationController() -> UINavigationController?`

This method is used for providing a navigation controller. It can be either a custom `UINavigationController` class, or just a regular `UINavigationController` instance themed by the container app.

- `func errorReceivedForAction(name: String?)`

This method is called when an error occurs after an action is executed.

- `func validate(validatorName: String, value: String) -> Bool`

This method is used for custom validators. It provides the name of the validator and the value to be validated. The method returns a boolean indicating whether the value is valid or not.

- `func dismissRequested(forProcess process: String, navigationController: UINavigationController)`

This method is called, on a modally displayed process navigation, when the user attempts to dismiss the modal navigation. Typically it is used when you want to present a confirmation pop-up.

The container app is responsible with dismissing the UI and calling the stop process APIs.

FXController

FXController is an open class, which helps the container app provide UIKit custom component screens to the renderer. It needs to be subclassed for each custom screen.

```
open class FXController: UIViewController {

    internal(set) public var data: [String: Any]?
    internal(set) public var actions: [ProcessActionModel]?
```

```
    open func titleForScreen() -> String? {
        return nil
    }

    open func populateUI(data: [String: Any]) {

    }

    open func updateUI(data: [String: Any]) {

    }

}
```

- `internal(set) public var data: [String: Any]?`

`data` is a dictionary property, containing the data model for the custom component.

- `internal(set) public var actions: [ProcessActionModel]?`

`actions` is the array of actions provided to the custom component.

- `func titleForScreen() -> String?`

This method is used for setting the screen title. It is called by the renderer when the view controller is displayed.

- `func populateUI(data: [String: Any])`

This method is called by the renderer, after the controller has been presented, when the data is available.

This will happen asynchronously. It is the container app's responsibility to make sure that the initial state of the view controller does not have default/residual values displayed.

- `func updateUI(data: [String: Any])`

This method is called by the renderer when an already displayed view controller needs to update the data shown.

FXView

FXView is a protocol that helps the container app provide custom UIKit subviews of a generated screen to the renderer. It needs to be implemented by `UIView` instances. Similar to `FXController` it has data and actions properties and a populate method.

```
public protocol FXView: UIView {
    var data: [String: Any]? { get set }
    var actions: [ProcessActionModel]? { get set }

    func populateUI(data: [String: Any]?)
}
```

- `var data: [String: Any]?`

`data` is a dictionary property containing the data model needed by the custom view.

- `var actions: [ProcessActionModel]?`

`actions` is the array of actions provided to the custom view.

- `func populateUI(data: [String: Any]?)`

This method is called by the renderer after the screen containing the view has been displayed.

It is the container app's responsibility to make sure that the initial state of the view does not have default/residual values displayed.

NOTE: It is mandatory for views implementing the FXView protocol to provide the intrinsic content size. Sample:

```
override var intrinsicContentSize: CGSize {
    return CGSize(width: UIScreen.main.bounds.width, height:
100)
}
```

FXCustomComponentViewModel

`FXCustomComponentViewModel` is a class implementing the `ObservableObject` protocol. It is used for managing the state of custom SwiftUI views. It has two published properties, for data and actions.

```
    @Published public var data: [String: Any] = [:]
    @Published public var actions: [ProcessActionModel] = []
```

Example

```
struct SampleView: View {

    @ObservedObject var viewModel:
```

```
FXCustomComponentViewModel

    var body: some View {
        Text("Lorem")
    }
}
```

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Renderer SDKs / Using the Android Renderer

Android project requirements

To use the Android Renderer library, ensure that your Android project meets the following minimum requirements:

- minSdk 26

Installing the library

1. Add the following code to your Android project's `settings.gradle` file::

```
dependencyResolutionManagement {
    ...
```

```
    repositories {
        ...
        maven {
            credentials {
                username "YOUR_USERNAME_HERE"
                password "YOUR_PASSWORD_HERE"
            }
            url 'https://nexus-
jx.dev.rd.flowx.ai/repository/flowx-maven-releases/'
        }
    }
}
```

2. Add the following code to your `app/build.gradle` file:

```
dependencies {
    ...
    implementation "ai.flowx.android:android-sdk:2.0.1"
    ...
}
```

Library dependencies

The Android Renderer library depends on the following libraries:

- Koin
- Retrofit
- Coil

Accessing the documentation

To access the Android Renderer library's documentation, follow these steps:

1. Download the **javadoc.jar** file from the same repository as the library.
2. Extract the **javadoc.jar** file.
3. Open the **index.html** file in your browser.
4. Navigate to `ai.flowx.android.sdk.FlowxSdkApi`.

Was this page helpful?