Kony Reference Architecture SDK

API Programmers' Guide

Release V8 SP4

Document Relevance and Accuracy

This document is considered relevant to the Release stated on this title page and the document version stated on the Revision History page. Remember to always view and download the latest document version relevant to the software release you are using.

October, 2019

Revision History

Date	Document Version	Description of Releases and Updates
12/18/2017	1.1	Updated for release with Kony Visualizer V8 SP1.
09/21/2017	1.0	Updated for release with Kony Visualizer V8.

Table of Contents

1. Kony Reference Architecture API Programmers' Guide 6

2. Overviews 7

2.1 Kony Reference Architecture: Decoded 8

2.2 Advantages of Using Kony Reference Architecture 11

2.3 A Deeper Look at Kony Reference Architecture 13

2.3.1 Views 15

2.3.2 Controllers 16

2.3.3 Models 17

2.3.4 Views and Controllers 18

2.3.5 Models and Controllers 24

2.4 Kony Reference Architecture Features 24

2.4.1 Models, Views, and Controllers in Action 25

2.4.2 Components and Kony Reference Architecture 26

2.4.3 Form Navigation 26

2.4.4 Dynamic Module Loading 30

2.4.5 Define Namespaces in Apps 31

2.4.6 Access Kony Fabric Services through Kony Reference Architecture 32

2.4.7 Use Kony Reference Architecture for Kony Wearables Apps 33

2.5 Create an App with Kony Reference Architecture 33

2.5.1 Build Your Front-End Client App 34

2.5.2 Build Your App's Data Model 36

2.5.3 Import Kony Quantum Visualizer Apps into Kony Visualizer Enterprise 40

2.5.4 A Sample FormController 41

3. References 43

3.1 FormController Object 44

3.1.1 FormController Events 45

3.1.2 FormController Methods 50

3.1.3 FormController Properties 54

3.2 kony.model Namespace 55

3.2.1 kony.model Constants 56

3.2.2 kony.model Objects 57

3.3 kony.mvc Namespace 66

3.3.1 kony.mvc Functions 66

3.4 kony.mvc.registry Namespace 67

3.4.1 kony.mvc.registry Functions 67

3.5 Navigation Object 71

3.5.1 Navigation Methods 71

3.6 TemplateController Object 73

3.6.1 TemplateController Events 74

3.6.2 TemplateController Methods 77

3.6.3 TemplateController Properties 78

3.7 Deprecated 79

3.7.1 kony.sdk.mvvm Namespace 79

1. Kony Reference Architecture API Programmers' Guide

Kony Reference Architecture is an integrated set of development tools that enables you to build modularized apps and increase your code reuse. This architectural pattern lets designers, front-end app developers, and back-end service developers to work in parallel on the same app.

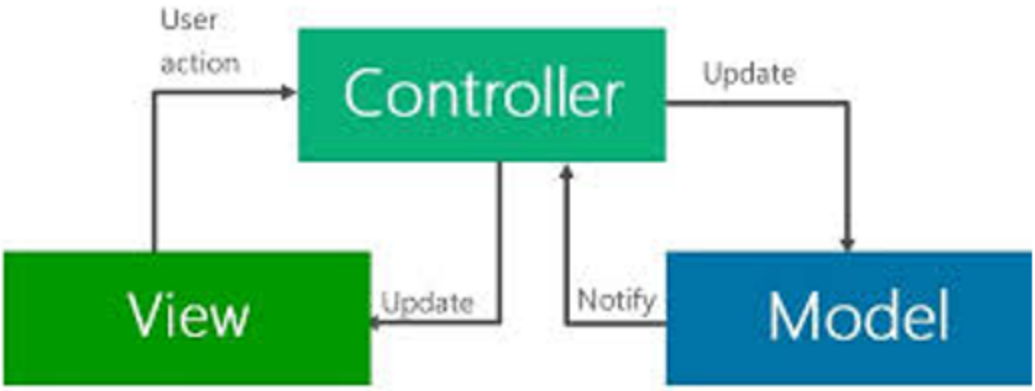
Kony Reference Architecture also enables you to create apps that you can deploy across many hardware platforms more rapidly than by using traditional JavaScript application-development techniques. Kony Reference Architecture provides a set of components and tools produced by Kony, Inc. that enables you to build apps in a highly modular fashion.

2. Overviews

Earlier, Kony apps were developed only with the Freeform JavaScript technique. JavaScript is a powerful language that provides developers with a lot of flexibility. It is an extremely accessible language that allows developers to start a project easily. However, all of these JavaScript features can create problems as a project grows in size and complexity. From Kony Visualizer 7.3 onwards, an MVC-based Reference Architecture has been integrated directly in to Kony Visualizer, which helps to improve the organization and consistency of the application code.

While developing applications by using the traditional Freeform JavaScript approach, developers had to heavily customize applications. This customization helped to overcome issues such as the usage of a large number of forms in the application code, the presence of global functions, and a lack of separation between the business logic and UI components. The Kony Reference Architecture mechanism takes these customized approaches to the next level by providing a standard in-built architecture to create apps.

Kony Reference Architecture allows you to create a separate Presentation layer. This Presentation layer enables a clear distinction between back-end objects, which model the perception of the real world, and presentation objects, which are the UI elements that appear on the screen. Furthermore, this separation helps you to avoid muddled dependencies and to keep a clear separation among app components.



While you develop apps by using Kony Visualizer and Kony Fabric, it is not mandatory to use Kony Reference Architecture. You can create apps by using Freeform JavaScript. You can, however, also use Kony Reference Architecture to develop apps, thereby leveraging the numerous advantages that this framework provides.

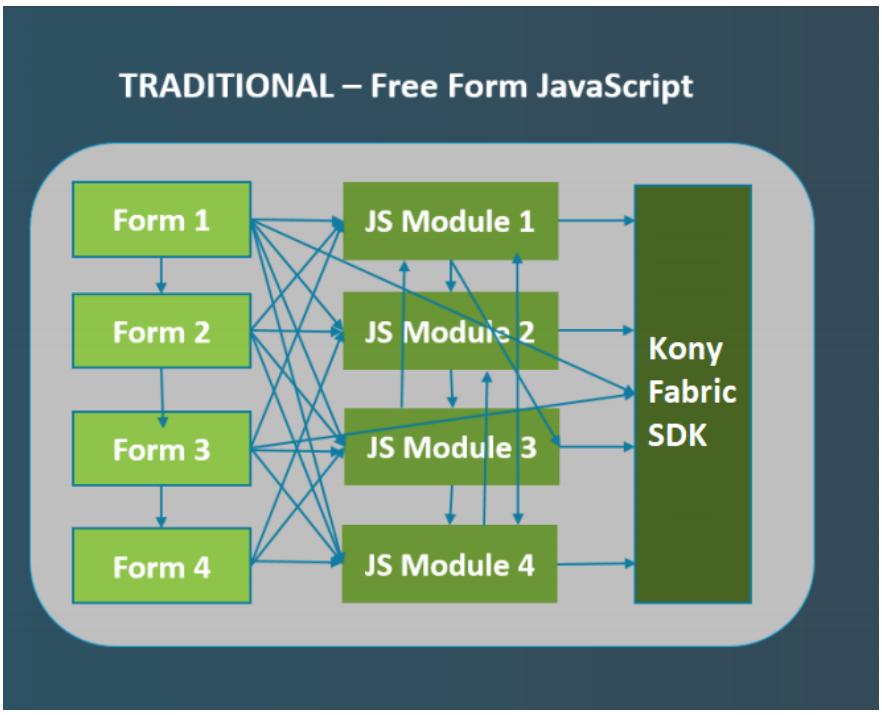
The following topics explain the overviews of Kony Reference Architecture:

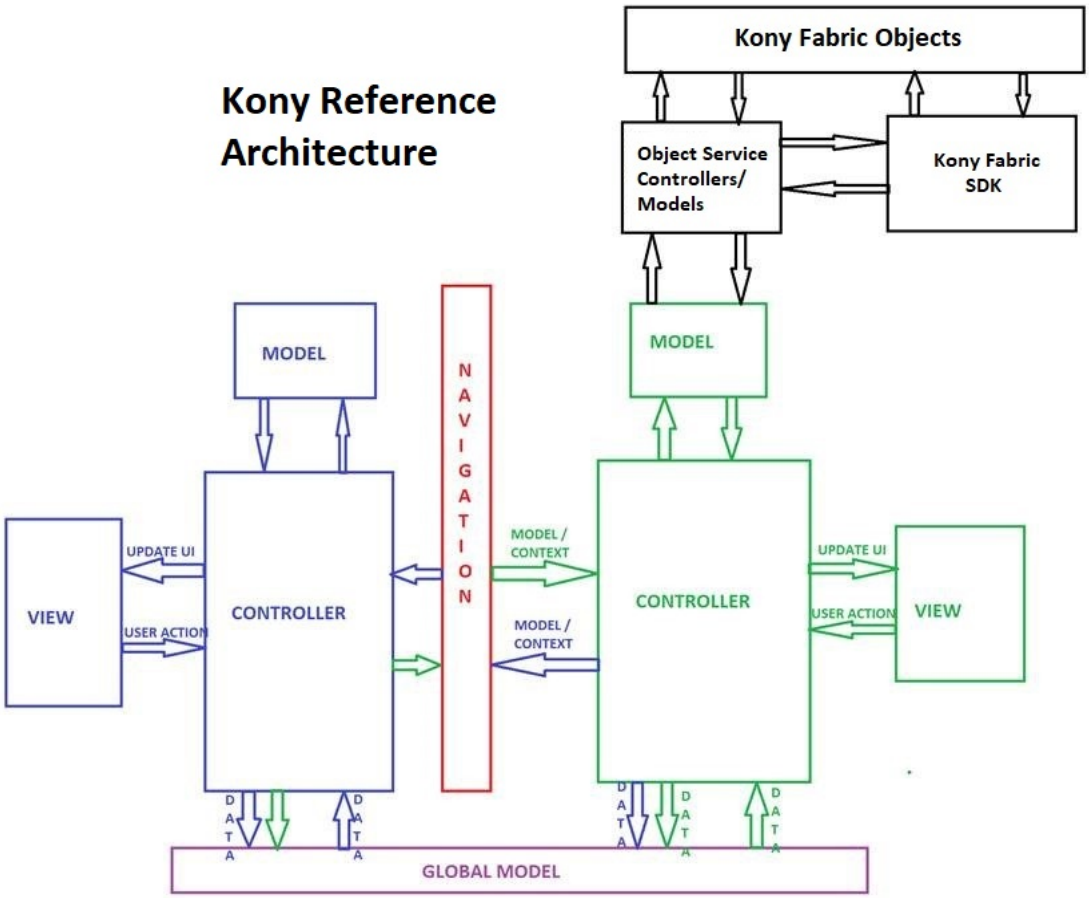
- Kony Reference Architecture: Decoded
- Advantages of Using Kony Reference Architecture
- A Deeper Look at Kony Reference Architecture
- Create an App with Kony Reference Architecture

2.1 Kony Reference Architecture: Decoded

Kony Reference Architecture allows you to develop highly modular and structured apps. Traditional JavaScript development results in the creation of muddled and unstructured apps, which introduce challenges as the apps grow in size and complexity. In a traditional JavaScript app, every element is global and can be accessed from anywhere in the program. Apps developed with Kony Reference Architecture, on the other hand, are highly structured even though they are still written in JavaScript. As a result, you can write highly reusable code modules that you can incorporate into many apps.

The following diagrams illustrate the differences between traditional Free Form JS app development and app development by using Kony Reference Architecture.





The diagrams show that because virtually everything is global in the traditional JavaScript model, any form could invoke any JavaScript module or any SDK function at any time. The result is that apps are nearly impossible to develop in a modular style. Code reuse is low and each new app often has to be rewritten from scratch, even if a previous app contained similar functionalities.

However, under Kony's implementation of the MVC architecture, Kony Reference Architecture structures app elements into three distinct components: Model, View, and Controller. This leads to benefits such as a structured separation of the code, parallel development of each app component, reduced complexity, and easier testing mechanism. For detailed information on the several advantages of using Kony Reference Architecture, click here.

The functionality of a motorbike is a real-world example for which the MVC architecture components can be explained. Every bike consists of the following three major units:

- View = User Interface (gears, suspension, seat, brake, clutch, exhaust nozzle)
- Model = Storage (fuel tank)
- Controller = Mechanism (engine)

2.2 Advantages of Using Kony Reference Architecture

- **Ease of use**: App developers have a shorter learning curve while using Kony Reference Architecture. This is because each developer needs to understand only the corresponding MVC component that he/she is developing. So, UI designers need to learn about only the View, the back-end developers have to know only about the Model, and the developers who create the app's business logic need to understand the Controller.
- **Get started easily**: Kony Reference Architecture provides code generation tools that help you to quickly get started with your app-development process. These tools automatically create Kony Reference Architecture classes that your app needs to access its services. You do not have to create these classes, so you can proceed directly to writing the business logic of your app.
- **Automatic generation of app components**: Kony Visualizer automatically generates most of the components of an app that is created under Kony Reference Architecture. The auto-generated objects provide straightforward and easily understandable interfaces. This results in the abstraction of most of the complexity of the app from both developers and customers.
- **Seamless integration with Kony Fabric**: If your app requires the use of back-end data services, Kony Reference Architecture provides a hassle-free integration with Kony Fabric. Your Kony Reference Architecture app can connect to the back-end data services available in Kony Fabric, with very little effort on your part.
- **Parallel app development**: As Kony Reference Architecture segregates all the elements of an app into three major units, it enables the development of both the front end and back end of the app in parallel. For instance, front-end developers do not have to wait until the back-end services of the app are implemented before they can develop the app. They can use mock objects services that simulate the app's back-end functionality while they develop the front end

of the app. Likewise, back-end developers can start development without needing any type of integration efforts with the app, until both the UI elements and the back-end services are in a stable state of development.

- **Faster app development**: The parallel app development feature of Kony Reference Architecture logically leads to the reduction in the time and effort required to develop an app. In addition, the use of Kony Reference Architecture speeds up your app development by avoiding to perform repetitive tasks such as writing code to fetch data or to set the value of widget properties. Instead, you can use declarative JSON data bindings to connect the fields in widgets to fields in data sources, even if those data sources are on remote servers. You do not have to write the code to update widget fields; it is generated automatically.

- **Code Separation and Reuse**: Kony Reference Architecture enables better code separation and reuse. Other development methods do not help you to encapsulate the JavaScript business logic of your apps. In other models, business logic, presentation logic, and navigation logic are often intermixed. This makes it difficult to reuse apps, in whole or in part, in other contexts.

  For example, suppose you develop banking services apps for banks. Using other architectures, the code for the business logic typically resides in the same code modules as the code for the navigation logic, presentation logic, or both. As a result, you will not be able to reuse the code from previous apps. Instead, you will probably need to start the app-development process from scratch.

  With Kony Reference Architecture, however, you can completely change the user interface and navigation logic when you write a new banking app, without having much impact on the business logic at all. Kony Reference Architecture separates all three types of program logic into different modules, which each have definite interfaces to encapsulate their internal functionality. This feature makes it easy to perform major changes to one part of the app, without breaking the rest of it. Presentation objects are completely separate from domain objects and business logic objects; so your app could potentially even support multiple presentations, possibly even simultaneously.

- **Designers, developers, and testers can work simultaneously**: Kony Reference Architecture lets designers and developers to easily work on their specific app components, without

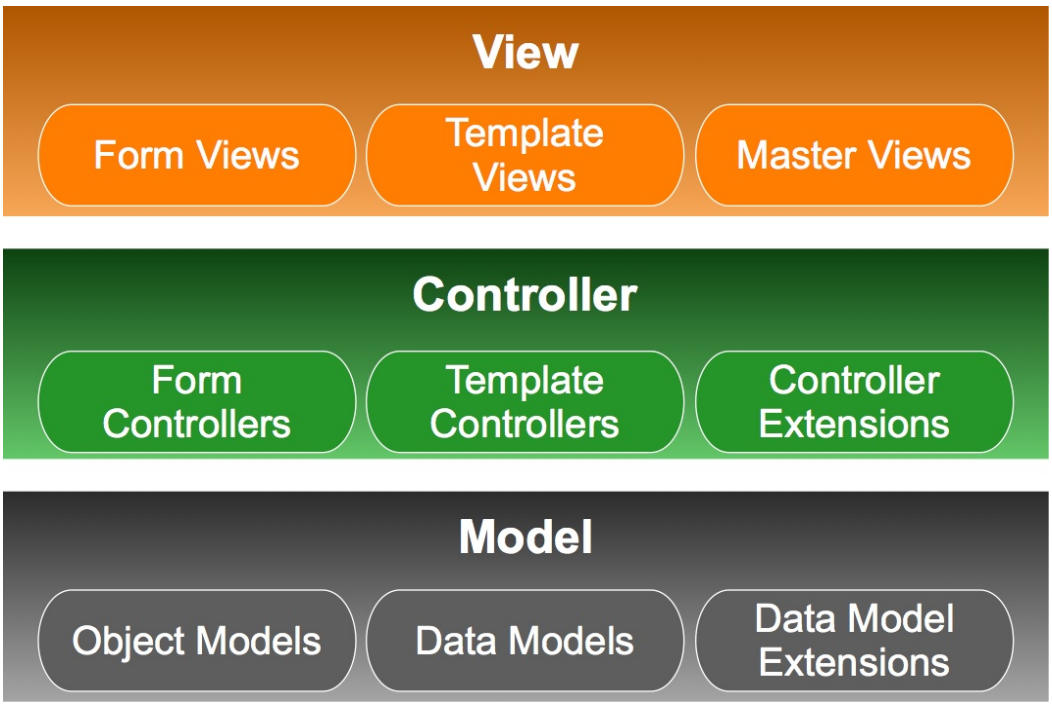
interfering with each other's work. Designers can create the user interface, iteratively improve the design, and perform all the testing they need to without impacting code developers on the project. Likewise, developers can write, revise, and test the app's business logic without having to worry about the presentation of the user interface. Furthermore, testers can test separate pieces of the app without waiting for the whole app to be complete. For instance, they can test the business logic even if the user interface has not been built. Or, they can test the user interface and navigation logic, regardless of whether or not the app's core business logic has been implemented.

- **ORM capabilities**: As many real-world apps generally use many remote data sources and services, object relational mapping (ORM) plays a critical role in app design and development. Object relational mapping (ORM) is a mechanism that makes it possible to address, access, and manipulate objects without having to consider how those objects relate to their data sources. Kony Reference Architecture simplifies ORM tasks by providing methods to discover ORM metadata. Your app can also use Kony Reference Architecture methods to auto-generate ORM queries.

2.3 A Deeper Look at Kony Reference Architecture

This section provides a more detailed examination of how Kony Reference Architecture works.

The following diagram shows a detailed presentation of the MVC architecture used by Kony Reference Architecture.



In Kony Reference Architecture , the actual implementation of the MVC architecture generally uses forms, with their widgets, as the View. The Controller and the Model are JavaScript code modules that implement their respective functionality.

Both the Controller and the Model are JavaScript modules. Kony Visualizer has a default naming scheme for your app's objects and files. So if you create a form in Kony Visualizer and set its name to `frmLogin`, then the Controller for that form is called `formLoginController` and it will be stored in a file called `frmLoginController.js`. Likewise, the file for the Model is named `frmLoginModel.js`. You can change these names in Kony Visualizer if you want to.

The default naming scheme is important to keep in mind when you're using the References section of this SDK's documentation. For instance, the **References** section contains documentation for the following objects.

- FormControllerObject
- TemplateController Object

You will not actually find objects with these names in your code. Instead, under the default naming scheme, you will find names such as those used above. That is, if you name your form `frmLogin`, then the FormController object for that form is called `frmLoginController`. And if you have a form called `frmMain`, then that form will have a FormController object called `frmMainController` that's stored in a file called `frmMainController.js`. All of your other FormController objects and TemplateController objects will be similarly named.

Note that there are some objects whose name is exactly what you see in the References section. These are as follows:

- kony.Model.Exception Object
- kony.Model.KonyApplicationContext Object
- Navigation Object

Your code accesses these objects by using appropriate names.

2.3.1 Views

Views in an app can be forms, templates, or masters. Apps under Kony Reference Architecture must have at least one form that functions as a View. More typically, apps have several forms, each one containing a variety of widgets for displaying information and for enabling user interaction. You create your app's forms in Kony Visualizer and add widgets as needed.

Templates enable you to provide your app with a uniform user interface. For instance, you can create a template for all of the buttons your app displays to make them all have the same colors, fonts, and shapes. If you make changes to the template, the changes propagate to all of the buttons that you have applied the template to.

Masters are a type of master form. In some ways they are similar to templates in that they provide a rapid way to add a standard user interface element to your app. However, masters are a forms. Therefore, you can encapsulate more into a master than you can encapsulate into a template. When building masters, you can add in forms, widgets, templates, code, and even other masters. This enables you to build highly complex standard components that you can just drop into as many projects as you want.

For example, you could create a master that provides all of the user interface elements and code needed to log into backend services that your company offers. Once this master is built and tested, you can easily add it to any app that you create, thus saving yourself large amounts of time.

Views are never global under Kony Reference Architecture . They can only be accessed by their Controllers. In fact, each View is stored in a member variable in the class of its Controller.

Kony Visualizer stores the forms for your Views in the Forms folder under the respective channels that you're developing your app for. So, for instance, forms for mobile devices are stored in a Forms folder under the Mobile channel.

2.3.2 Controllers

Every View requires an associated Controller. Therefore, your app's code can have form Controllers, master Controllers, and template Controllers in it. They are all implemented as JavaScript modules. Controllers contain the business logic of an app. They communicate with the data Model objects to retrieve, update, and process the app's data. Controller can communicate with as many Models as needed.

When Controllers operate on an app's data, they also send the data to the View to be displayed in the corresponding form, template, or master. In this way, it updates the user interface whenever there is a change in the displayed data from the Model.

In addition to form Controllers and template Controllers, Kony Reference Architecture also provides Controller extensions. You can write Controller extensions in JavaScript modules to provide specialized or enhanced functionality for components. For example, suppose that you create a master that encapsulates all of the functionality for logging onto your backend database. Imagine that you are creating a new app and you drop the login master into your new app. Now you want to add the ability to log in using Facebook. With a Controller extension, you can add the Facebook login functionality to your login master without changing the base login master itself. You just add in some new UI elements and add the new functionality for logging in with Facebook to a Controller extension that you write. That way, none of your new code impacts the standard login master that you've created and that you use in all of your apps. Each individual app can enhance the standard login master in any way you need without you having to modify the standard login master itself.

Controllers for Views are typically stored together with their forms, as the following figure shows.

However, shared Controllers are stored in the `Shared` folder, which appears after you create a shared Controller. When it is empty, the `Shared` folder is not shown.

2.3.3 Models

It's often the case that apps communicate with, retrieve data from, and update multiple data sources. Each data source is represented to the app as a Model. Models encapsulate data sources and make it possible for your app to access them in a standardized way. The data sources that Models encapsulate can be on the user's device or remotely accessible across the Internet.

Models are optional in your apps. Simple apps might not use them. For example, a calculator app would not need Models because the data it operates on is probably nothing more than a few variables containing some numbers.

Most enterprise-level apps use Models to interface to backend data sources. Typically, developers who create their apps with Kony Visualizer will also use Kony Fabric to create their server-side apps that provide access to their backend data sources. This is not required, it's just the easiest way to build your app. If you decide to use Kony Fabric for your backend app, you can get it to generate your Models for you. More specifically, you create your backend app by building object services with Kony Fabric . Utilizing the Kony Fabric console, you can then generate Models, called object Models, that provide your front-end Kony Visualizer app with access to your backend app's object services. After you generate your object Models for all of your backend data sources, Kony Visualizer downloads them into your front-end Kony Reference Architecture project that you are building in Kony Visualizer on your local development PC . The object Models provide your front-end app with code that enables the app to retrieve data from the backend object services, update, create, or delete the data, and save the changes to the backend object services.

One of the many advantages of using Models to represent your data sources is that designers and developers working on the front-end app don't have to wait until the backend Kony Fabric app is complete before they start their work. Developers on the front-end app can build objects that provide *mock services* to the app. That is, developers can create Models to use in the front-end app that simulate the interaction that the front-end Kony Reference Architecture app will have with the backend Kony Fabric app when the backend app is complete. Using these mock services, both the front-end app and the backend app can be under development at the same time.

Kony Reference Architecture also provides you with object Model extensions that you can put custom code into to enable your app to do data validation or process the data before it is displayed or saved. Kony Visualizer generates the object Model extension for you and includes them in your Kony Visualizer project.

Models are stored as a shared resource in your Kony Visualizer project.

2.3.4 Views and Controllers

Forms under Kony Reference Architecture work very similarly to the way they work in a free form JavaScript app built with Kony Visualizer. For example, whether you're building a Kony Reference Architecture app or a free form JavaScript app, you can drag and drop forms, widgets, and so forth onto any form using the WYSIWYG editor in Kony Visualizer. You can use forms across multiple channels. That is, you can use the same form for Android phones, iOS phones, and so on, Or, if you prefer, you can use specific forms for specific channels.

The main difference between forms in Kony Reference Architecture and forms in a free form JavaScript application is that forms in Kony Reference Architecture have Controllers associated with them. Kony Visualizer automatically generates form Controllers for each form you add to your UI. When you add actions to forms in Kony Reference Architecture , Kony Visualizer automatically generates action Controllers for them.

Views are only available from within the form's Controller. So only the form's Controller can update the form's data. Your app uses the kony.mvc.Navigation function to create a Navigation object. It can then call the `Navigation` object's navigate function to move from form to form. Because access to a form only happens through the form's Controller, your app cannot call a form's show or destroy methods. Only a form's Controller can display the form on the screen. And if your app needs to destroy a form it calls kony.application.destroyForm, which destroys the form, its Controller, all widgets it contains, and its children.

Add Actions

Kony Visualizer enables you to add actions to your app's widgets. In fact, this is the way to add actions to your app's form Controllers. When you add actions to a widget, the `this` keyword inside the widget's callbacks refers to the form Controller. To add a function in a Controller as the event callback handler for a widget's event, your app uses code similar to the following.

```
btntest.onClick = Controller.AS_Button_OnClickEvent;
```

In the code snippet shown here, `btntest` is the name of a `Button` widget. This snippet sets the `Button` widget.'s `onClick` event. The event callback handler is the `AS_Button_OnClickEvent` function, which is a member of the `Controller` object. The `Controller` object is an object that Kony Visualizer generates for your form. The `AS_Button_OnClickEvent` function is written by you.

The following code sample demonstrates how an application might add an event callback handler to a button.

```
define('frmLogin', function ()
{
    return function (Controller)
    {
        function addWidgetsfrmLogin()
        {
            this.setDefaultUnit(kony.flex.DP);
            var btnSetIPAddress = new kony.ui.Button(
            {
                "height": "55dp",
                "id": "btnSetIPAddress",
                "onClick": Controller.AS_Button_
6c7c9d022bcc4a61a603aa3c89110efe,
                "skin": "buttonOnfrmLoginSkin",
                "text": "Set IPAddress",
```

```
                "width": "25%",
                "zIndex": 1
            },
            {
                "contentAlignment": constants.CONTENT_ALIGN_CENTER,
                "displayText": true,
                "padding": [0, 0, 0, 0],
                "paddingInPixel": false
            },
            {});
            this.add(btnSetIPAddress);
        };
        return [
            {"addWidgets": addWidgetsfrmLogin,  "id": "frmLogin",
"layoutType": kony.flex.FLOW_VERTICAL},
            {"displayOrientation": constants.FORM_DISPLAY_
ORIENTATION_PORTRAIT,},
            {"retainScrollPosition": false, "titleBar": false}]
    };
});
```

The example above adds a `Button` widget called `btnSetIPAddress` to a form called `frmLogin`, which is a form that is used to display a login screen. For the onClick event, the example sets a function called `AS_Button_6c7c9d022bcc4a61a603aa3c89110efe` as the event callback handler.

Share Controllers Between Forms

Typically, each form has its own Form Controller. However, you can assign a Controller to multiple forms if you choose to do so. If the forms that share the Controller are specific to a particular channel, such as iOS, Kony Visualizer automatically stores the shared Form Controller in a folder under that specific channel.

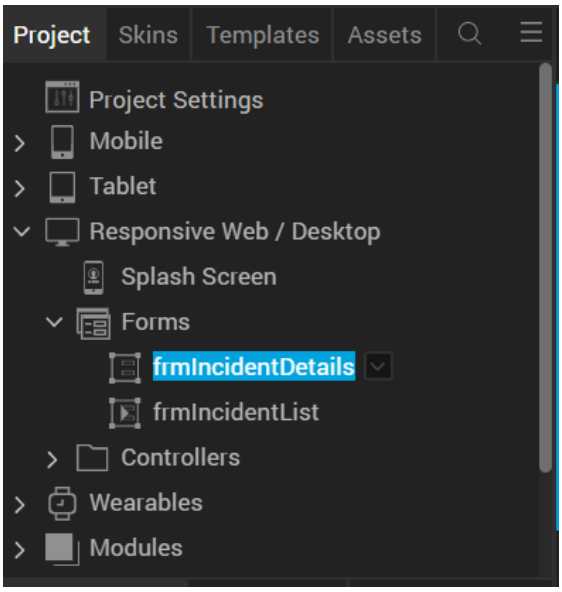
It is also possible for forms that are used across channels to share a single Form Controller. Let us suppose that your app has a set of three forms that are used on both the iPhone and Android phones. Furthermore, consider that all three of those forms share the same Controller. In such a scenario, the shared Form Controller can be found in a folder outside of the iOS and Android channels that is specifically for shared Controllers.

> ***Note:*** It is not possible to share the **ControllerActions** JavaScript file between multiple forms.

While developing your app, you can specialize existing forms for particular channels. This process is called *forking* the form because Kony Visualizer actually creates a new version of the form for the specific channel. If you fork the form, it automatically forks its Controller. Forked forms cannot be shared.

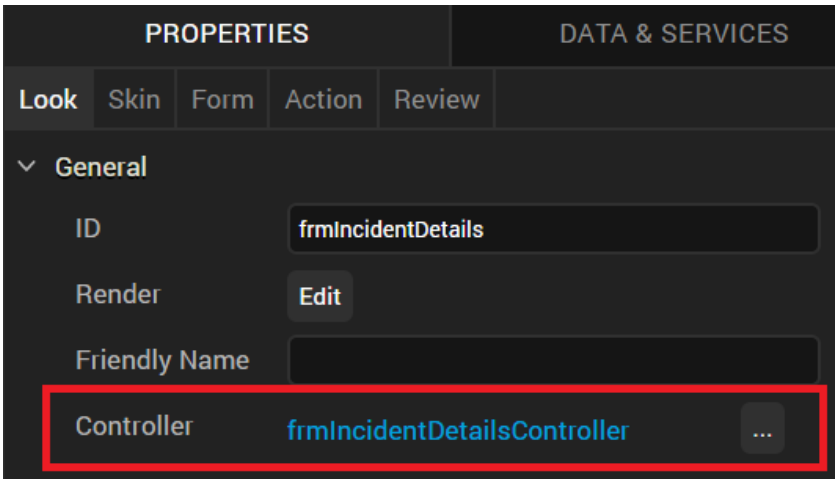
To share a Controller between forms, follow these steps:

1. In your Kony Reference Architecture project, click the form with which you want to share a Controller. Here, **frmIncidentDetails** is the selected form in the Responsive Web channel.

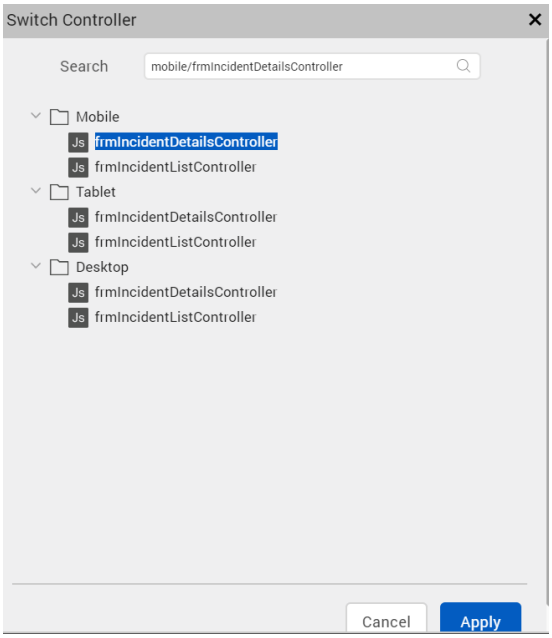


2. Go to the **Properties** panel > **Look** tab.

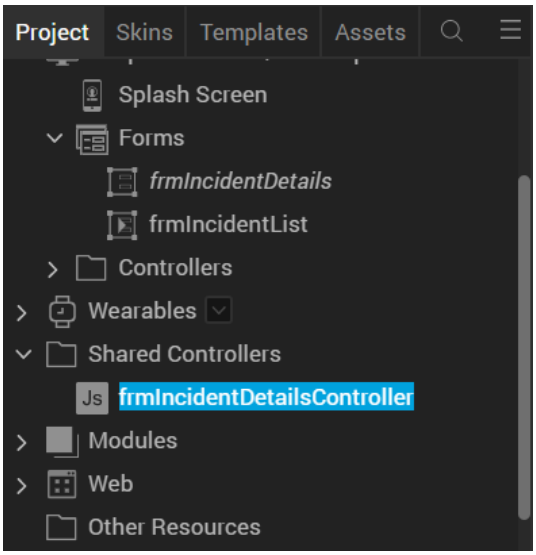
3. For the **Controller** field and beside the Form Controller name, click the Ellipsis Menu icon ...
Here, **frmIncidentDetailsController** is the Form Controller of the **frmIncidentDetails** form. The **Switch Controller** window appears, with the list of available Controllers in different channels and **frmIncidentDetailsController** in the Desktop (Responsive Web) channel selected by default.



4. Click the Controller that you want to share with the form. Here, we have selected **frmIncidentDetailsController** of the **frmIncidentDetails** Mobile form.



5. Click **Apply**. The **frmIncidentDetailsController** of the **frmIncidentDetails** Mobile form is shared with the **frmIncidentDetails** Responsive Web form. A new folder called **Shared Controllers** is also created in the Project Explorer, with **frmIncidentDetailsController** placed under it. When you write any code in the **frmIncidentDetailsController** JavaScript file, the code is shared with all the forms that this Controller is shared with.



2.3.5 Models and Controllers

Models encapsulate data storage locations and provide a standardized interface for creating data on those data storage locations, reading it into the app, updating it, and deleting it. The data storage locations can be on the user's device or remotely connected across a local network or the Internet. Wherever the data resides, the app uses Models as a standard way of accessing it.

In Kony Reference Architecture , Controllers contain the app's business logic. Therefore, an app's Controllers use Models to perform operations on data storage locations, which are often referred to as *data sources*.

2.4 Kony Reference Architecture Features

Kony Reference Architecture supports the use of Kony forms and widgets. You can use these elements to build your app's user interface just as you normally do when developing apps with Kony Visualizer. Under Kony Reference Architecture, you cannot use deprecated box-style widgets such as popups, VerticalBox forms, HorizontalBox forms, and box-based templates. You must build your app with FlexForm-based widgets.

To enable the modularization of your app's JavaScript source code, Kony Reference Architecture mandates the use of RequireJS and the Asynchronous Module Definition (AMD) API for loading JavaScript files and modules. Therefore, any code modules you add to your app must follow the RequireJS and AMD conventions.

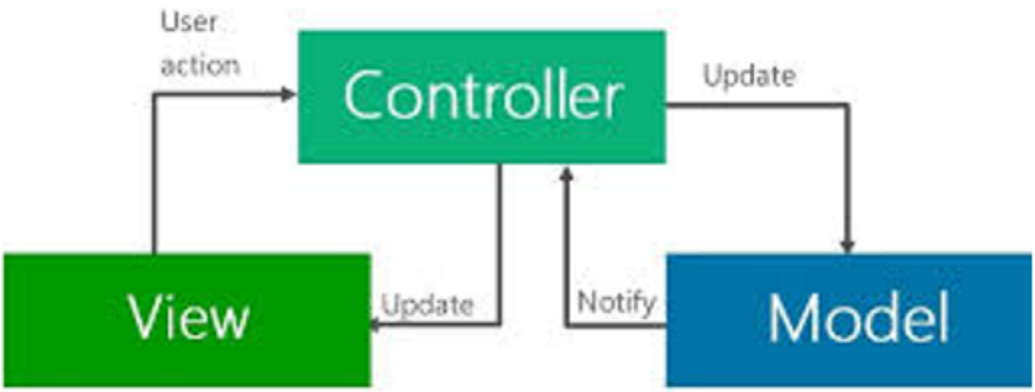
This section contains the following topics:

- Models, Views, and Controllers in Action
- Components and Kony Reference Architecture
- Form Navigation
- Dynamic Module Loading
- Define Namespaces in Apps

- Access Kony Fabric Services through Kony Reference Architecture
- Use Kony Reference Architecture for Kony Wearables App

2.4.1 Models, Views, and Controllers in Action

Models, Views, and Controllers work together to provide an app's functionality. The following diagram illustrates how Controllers interact with Views and Models.



The Controller responds to user actions that it receives from its associated View. As stated previously, each Controller is associated with exactly one View. However, Controllers may communicate with any number of Models.

All Controllers have a member variable named `View` that contains the View for that specific Controller. Views are only accessible from within their corresponding Controllers by using the statement `this.View`.

Each form, template, or master in an app has an associated Controller and only the individual Controllers can directly access their own Views. However, when needed, Controllers can invoke their parent Controller's methods by calling the `executeOnParent` function. This provides both a clean separation of the layers in the hierarchy of Views and a solid encapsulation of each View's functionality.

> ***Important:*** It is possible to define a master without a contract. The complete View hierarchy of a master without a Controller is accessible from both its own Controller and that of its parents. For more information, see Masters.

Controllers can also retrieve information from Models, display it in Views, and enable the user interact with it. Based on the user's input, the Controller can send notifications to the Model, which saves the changes onto the data source.

2.4.2 Components and Kony Reference Architecture

When you create a component in either a Free Form Java Script or in a Kony Reference Architecture project, Kony Visualizer automatically creates one Controller.js and one ControllerActions.js file. Consequently, any component that is created contains Kony Reference Architecture modules by default.

For more information about components, refer the Creating Applications With Components section in the Kony Visualizer User Guide.

2.4.3 Form Navigation

Kony Reference Architecture dynamically loads forms at runtime. When a Kony Reference Architecture app creates a form, it also assigns the form a "friendly" name that is more readable to humans than the form's ID. A form's friendly name must be unique and it should make sense to the programer or programmers maintaining the app's source code.

> ***Note:*** Your app can also assign friendly names to templates. But templates are not involved in navigation.

Kony Reference Architecture maps the friendly names to the forms in your app. To navigate between forms, an app must create a Navigation object by calling the kony.mvc.Navigation function. When invoking the `kony.mvc.Navigation` function, your app passes it the friendly name of the target form. Once the `Navigation` object is created, the Controller for the currently-displayed form can

switch to the target form by calling the `Navigation` object's navigate method . This activates the Controller for the destination form. When the Controller for the destination form is active, it can then display its View, get data from one or more Models, and so forth. The following code sample illustrates how this is done.

```
var params = {"title" : "My Title", "description" : "My
description"};
var x = new kony.mvc.Navigation("FormFriendlyName");
x.navigate(params);
```

The example code here navigates to a new form whose friendly name is `FormFriendlyName`. In the call to the `Navigation` object's `navigate` method, it passes parameters from the current form Controller to the destination form Controller through the `params` argument. The `params` argument is a JavaScript object that is passed to the Controller of the target form. It can contain a small amount of context information for the target form's Controller.

Calling the `Navigation` object's `navigate` method creates the target form and its Controller, and then activates the target form's Controller. Your app does not need to call the form's destroy method on the form being navigated away from. In fact, under Kony Reference Architecture , it can't invoke the `destroy` method for any form. Instead, your app calls the kony.application.destroyForm method to dispose of forms, their Controllers, and all of their child widgets.

Your app also cannot call the `show` method on any form and does not need to. Under Kony Reference Architecture , the form is the implementation of the View. It can only be directly accessed by the form's Controller through the Controller's `View` property. Therefore, the Controller can get access to its View with the `this.View` statement.

Customize Form Navigation

Your app can customize the navigation process by implementing callback handler functions for the target form's Controller events. These events are triggered during navigation and before the target form is visible. Providing callback handler functions for them enables you to customize what happens when a form is navigated to.

For example, if you want to customize the context information the target form receives, you can provide a callback handler function for the onNavigate Event. This is shown in the sample below,

```
onNavigate : function(context, isBackNavigation)
{
    this.context = context;
}
```

> ***Note:*** The object that is sent as part of the onNavigate Event is accessible for all form lifecycle events.

The context that is passed with the onNavigate Event of the kony.mvc.Navigation Object is available in the navigationContext key of FormController instance.

The following code snippets demonstrate how to access the context from the lifecycle events of forms:

1. Navigate from source form to destination form.

```
var nav = new kony.mvc.Navigation("DestinationForm");
nav.navigate({"key1":"value1"});
```

2. Link preShow, postShow, and onMapping Events of the destination form with the appropriate events function defined here.

```
function preShow()
{
 kony.print(this.navigationContext);
}
function onMapping()
{
kony.print(this.navigationContext);
}
function postShow()
{
kony.print(this.navigationContext);
```

```
}
//Here, this.navigationContext contains the context that was passed
in navigate Method during the navigation from the source form to
the destination form.
```

3. In the `onNavigate` method, your app may need to pause the navigation so that it can load data, or do whatever else it needs to do, by invoking the pauseNavigation and resumeNavigation methods.

4. You may also want to specify a custom Model for the target form. To do so, provide callback handler functions for the getModel and setModel functions, as illustrated in the following sample code.

```
getModel : function()
{
    this.Model = new CustomFormModel();
    return this.Model;
}

setModel : function(newModel)
{
    this.Model = newModel;
}
```

Control Flow of `navigate` Function

The exact control flow for the `navigate` function is as follows:

1. Get the Controller if it exists already. If not, create it.

2. Update the Model with the `Navigation` object's Model.

3. If it is defined, invoke the target Controller's `onNavigate` callback handler function.

4. The target Controller shows the form.

2.4.4 Dynamic Module Loading

Kony Reference Architecture apps can define distinct modules that contain discreet functionality and load them dynamically on demand. In fact, Kony Reference Architecture does this with its own code modules. For instance, under older programming Models, apps loaded all of their JavaScript modules at startup. However, Kony Reference Architecture loads them on demand. This both saves memory and decreases startup time.

Using Kony Visualizer, you can create your JavaScript modules consisting of a form and a form Controller The file containing the form has the name:

```
<formID>.js
```

where `<formID>` is the unique ID of the form your app is loading. Similarly, the form Controller is contained in a file called:

```
<formID>Controller.js
```

where `<formID>` is the unique ID of the form your app is loading.

These two files follow the format defined by the RequireJS standard. In addition, Kony Reference Architecture adds a method called `addWidgets` to the form. This method has the following signature.

```
addWidgets(formref);
```

where `formref` is a reference to the widget to add.

Kony Reference Architecture uses an AMD stack for loading JavaScript modules, so the functionality in your modules must use the AMD conventions.

When loading a module, your app must follow the standard RequireJS notation. So when your app specifies the file name it must not include an extension suffix. This is illustrated in the following sample code.

```
ControllerConfig = require("accountModule");
```

As the example shows, an app can load a file called `accountModule.js` by invoking the `require` function and passing it the name of the file without the `.js` extension. The file name must match the name given in `define` notation in your app. All of the define notation uses that are mentioned in the RequireJS documentation are supported in Kony Reference Architecture except for `require.config`. Paths are always relative to the root JavaScript folder.

Kony Reference Architecture also supports module dependencies. So if your app loads a module that is dependent on another module, it is loaded as well.

2.4.5 Define Namespaces in Apps

In addition, Kony Reference Architecture lets you define namspaces in your apps for the masters that you create. Each fragment inside the namspace's name is a folder name. For example, suppose you create the namespace mycompany.ui in your app. Further imagine that the mycompany.ui namespace contains a file called `ChartControll.js`. The path to the file would then be `mycompany\ui\ChartControl.js`. The name for this file in RequireJS notation would be `"mycompany/ui/chartcontrol"`. To load this file, your app would need code similar to the following example.

```
require(
    ["mycompany/ui/chartcontrol"],
    function(retValue)
    {
        //use retValue
    });
```

> ***Important:*** You can only define namespaces for your masters, not for forms.

If your app needs to load a module in the context of a worker thread, it can do so by adding the worker thread before the file name, as shown in the following code.

```
ControllerConfig = require("workerthread\accountModule2");
```

2.4.6 Access Kony Fabric Services through Kony Reference Architecture

In addition to modularizing and encapsulating an app's internal components for increased re-use, the Kony Reference Architecture SDK also modularizes and encapsulates the app's access to backend services. In particular, the Kony Reference Architecture SDK interfaces directly to Kony Fabric services to a seamless, end-to-end development environment for your apps.

The easiest way access backend data sources is to interface your front-end client app with a backend Kony Fabric app. In this way, you can easily access a wide range of backend data source through the uniform and standardized interface that Kony Fabric provides. Backend data sources are accessed through object services. Object services, in turn, are represented in your app by object Models, which are often just called Models. So the Kony Reference Architecture SDK uses object Models to provide front-end client apps with a uniform way to exchange data with backend data sources. In fact, the Kony Reference Architecture SDK generates object Models for you that provide you with code to create, read, update, and delete records in backend data sources.

Using Kony Kony Fabric, your Kony Reference Architecture SDK app can quickly send multiple requests to backend services that can then be executed concurrently. For example, if you were writing a banking app, your app can use the Kony Reference Architecture SDK and Kony Fabric to rapidly send requests for account information and customer personal information and also request map information from a commercial map server, such as Mapquest. All of these requests are executed on their respective concurrently because the successive requests are sent out before any of them return information. When they do respond, the information appears to come back to your app "automatically" because the Kony Reference Architecture SDK and Kony Fabric handle most of the work.

Of course, you can add custom logic to your app to do whatever data processing is necessary. For instance, in the preceding banking app, your app can request a map of the area in which the user is standing. It can also send out a request to the bank's corporate servers asking where the branch offices are in that locality. When the two pieces of information come back to the user's device, the app can use custom logic that you write to combine the branch office locations with the map so that the user can see where the nearest branches are.

When you develop an app, you build your object services in Kony Fabric to provide your front-end client app with access to backend data sources. You then use Kony Visualizer to create your front-end client app. With the Kony Reference Architecture SDK and Kony Fabric, you can provide end-to-end solutions for your customers and at the same time focus on the specific logic for the task at hand rather than user interface tasks, backend connection tasks, and so forth. The Kony Reference Architecture SDK and Kony Fabric provide you with a powerful toolset that enables you to automate most of the job of app production.

The Kony Microservices Framework Server Tools provide server-side objects that connect with one or more Kony Fabric services. These services can range from Identity services to Messaging and Sync services. You can also interface your app with SAP, SOAP, REST, and RDBMS services through Kony Fabric. With this development Model. you have full access to the Kony backend services that any other app built on Kony technologies would have. And most of the objects, for both the client and the server sides of the app, can be generated automatically so you don't have to code them yourself.

2.4.7 Use Kony Reference Architecture for Kony Wearables Apps

It is important to note that you can create a Kony Wearables app under Kony Reference Architecture . For example, Kony Wearables enables you to develop apps for the Apple Watch. When you create an Apple Watch app, you can use Kony Visualizer to create the app's forms. However, Kony Visualizer does not create Controllers for the forms in an Apple Watch app because the Apple Watch app has its own specific architecture.

In addition, you can add Apple App Extensions to your Kony Reference Architecture project so that it can use Apple App Extensions on iOS and OS X. Kony Visualizer does not generate any Kony Reference Architecture for Apple App Extensions. So adding App Extensions does not result, for example, in additional Controllers in your project.

2.5 Create an App with Kony Reference Architecture

When you create an app with Kony Reference Architecture, you can start by building the app's data model in Kony Fabric Console. You can add various back-end services and operations that your front-end client app requires. You can then build your front-end client app with Kony Visualizer. Kony Visualizer provides you with a way to interface your front-end client app with your back-end Kony Fabric app, as described on Kony Visualizer User Guide and in Kony Fabric User Guide.

2.5.1 Build Your Front-End Client App

After you have created your Object services by using Kony Fabric, you can build your front-end client app with Kony Visualizer.

Using the Kony Fabric channel in the Kony Visualizer Enterprise Edition **Project** pane, you can connect your front-end client app to your back-end Kony Fabric app and the services it offers, and then generate the object model. The Kony Fabric node is not available on Kony Visualizer Starter Edition.

You can select the channels for which you want to build your app, such as Desktop, Mobile, Android Wear, or Tablet. The Reference Architecture Extensions feature is not available for the Apple Watch channel. You can then design the user interface of your app by using various widgets available on Kony Visualizer. For more information on channels, widgets, and API functions that are available on Kony Visualizer, refer Kony Visualizer User Guide, Kony Visualizer Widget Programmer's Guide, and Kony Visualizer API Programmer's Guide.

Create a Kony Reference Architecture Project

You must follow these steps to create a Kony Reference Architecture project on Kony Visualizer:

1. On Kony Visualizer, click **File**, and then click **New Project**. Kony Visualizer displays the **New Project** dialog box with the types of apps that you can create.
2. Select the **Create Custom App** option, and click **Choose**. Kony Visualizer again displays the **New Project** dialog box with the available project types
3. Select the **Kony Reference Architecture** project type.
4. Type the name of your project in the **Project Name** field. You must follow these guidelines while specifying the name of your project:
    - The name must always start with an alphabet.
    - The name should contain only alphabets and digits.

- Special characters and reserved words are not allowed.
- The name must contain more than three characters.

5. Click **Create**. Kony Visualizer creates the project.

Build the App's User Interface

Your client app's user interface displays one or more screens, also called views. Views can be forms, templates, or masters. Every view must have at least one of these. More typically, a view requires multiple forms, templates, or masters. The process of creating views is described in the Kony Visualizer User Guide.

After creating at least one screen for your app using forms, you can add widgets to the forms. Widgets provide your app with the user interface elements that it needs. These include buttons, menus, text labels, calendars, and more. They also give your app access to the functionality of the user's device through the use of a camera widget, a phone, widget, and so forth. The process of populating your app's forms with widgets is presented in the Kony Visualizer User Guide.

Add Functionality to Your App

Each time you add forms to your app, Kony Visualizer automatically adds a controller for each form and creates a folder in your project to put it in. You'll find the controllers for your forms in the project tree under the channel that you're developing the app for. So if you add a form called frmMain to your project and you're developing the app for Android and iOS, you'll find folders for the frmMainController in the Android and iOS branches of the project tree. Whenever you change the names of your forms and templates, Kony Visualizer automatically renames the controllers associated with them.

Likewise, when you add templates to your apps, Kony Visualizer adds the corresponding controller for each template. Renaming your template automatically renames its controller.

To add functionality to your app, you add your custom JavaScript code to the controllers in your app. The controller for a form or template is the only object that has access to the form or template. Only the controller can performs actions on it.

Your app may also contain models, one for each backend data source. The data sources can be local on the device or remote servers that are accessed across the network. If you need to, you can add custom code to your app's models to enhance or customize the model's functionality.

In addition, you can add actions to your forms just as you would with any other Kony Visualizer app. When you do, Kony Visualizer automatically creates an action controller for your actions. However, this is an autogenerated file and you should not make any changes to it. If you do, they will be overwritten the next time the file is regenerated.

2.5.2 Build Your App's Data Model

The steps required to build the data model of your app are as follows:

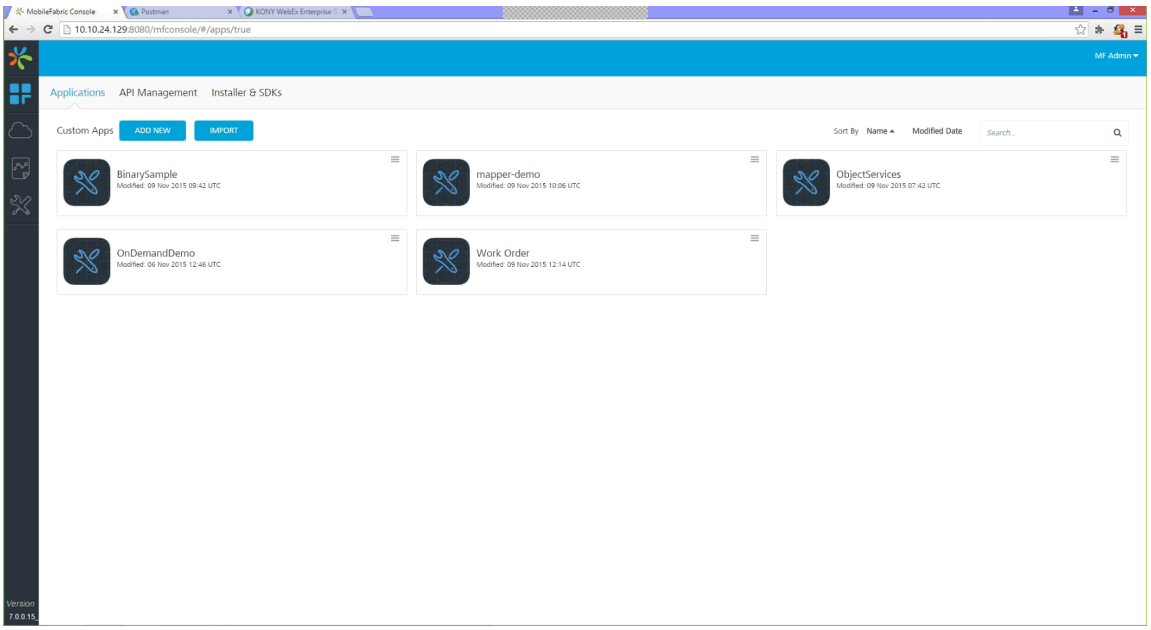
- Build a Kony Fabric app
- Configure Identity Services
- Create an Object Service
- Configure the Data Model

Build a Kony Fabric App

To integrate your front-end client app with the back-end services that you want the app to access through Kony Fabric, you must first create a Kony Fabric app by using Kony Fabric Console. For more details on how to do so, refer Kony Fabric documentation.

In this walkthrough, we will create a simple service that integrates with SAP data in the back end. Although your data may reside in a different backend storage system, the basic workflow for building your app's data model will be very similar to what's shown here. In this walkthrough, we will assume that you have already created your Kony Fabric app.

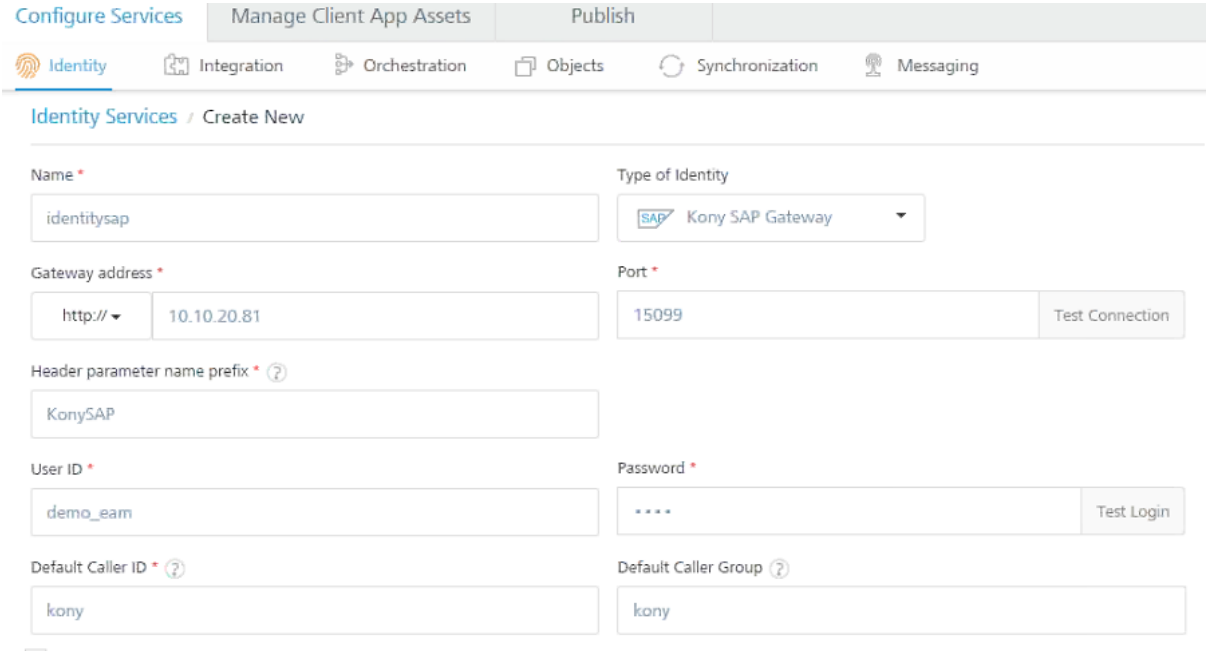
For our example, the Kony Fabric app is called Work Order. The Work Order Kony Fabric app gets its data from SAP. In the Kony Fabric console, the results will resemble the following illustration.



Configure Identity Services

1. Select the app you just created. In this example, it's the Work Order app.
2. Click the **Configure Services** tab.
3. Choose **Identity Services**.
4. Click the **Configure New** button.
5. Set the identity's name.
6. Select the **Type of Identity**. In this example, it will be Kony SAP Gateway.
7. Set the address and port of the gateway server.

8. Supply the remaining information such as the login credentials and so forth. You screen will resemble the following.

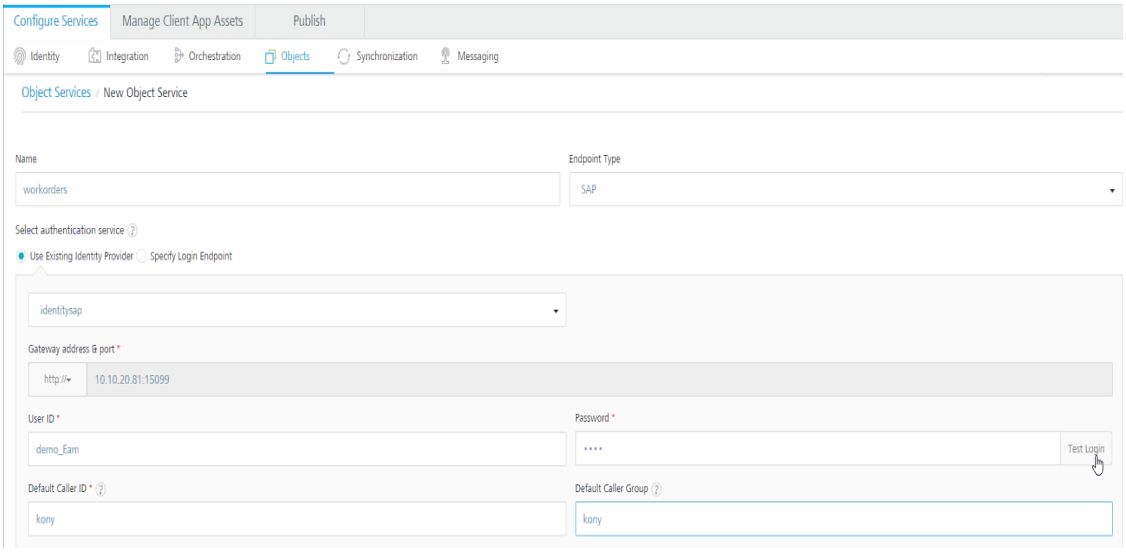


Create an Object Service

Next, you create an object service that will provide your front-end client app with access to the data in the data store. In this example, the client app on the device or desktop will access the work order data in the SAP database.

1. In the **Configure Services** tab in the Kony Fabric console, click **Objects**.
2. Select the **Configure New** button.
3. Set the name and endpoint type. In this example, the endpoint type is SAP.
4. Select **Existing Identity Provider** and enter the name of the identity service you create in Step 2. This example uses the name identitysap.

5. Fill in the other information such as User ID, Password, and so forth.



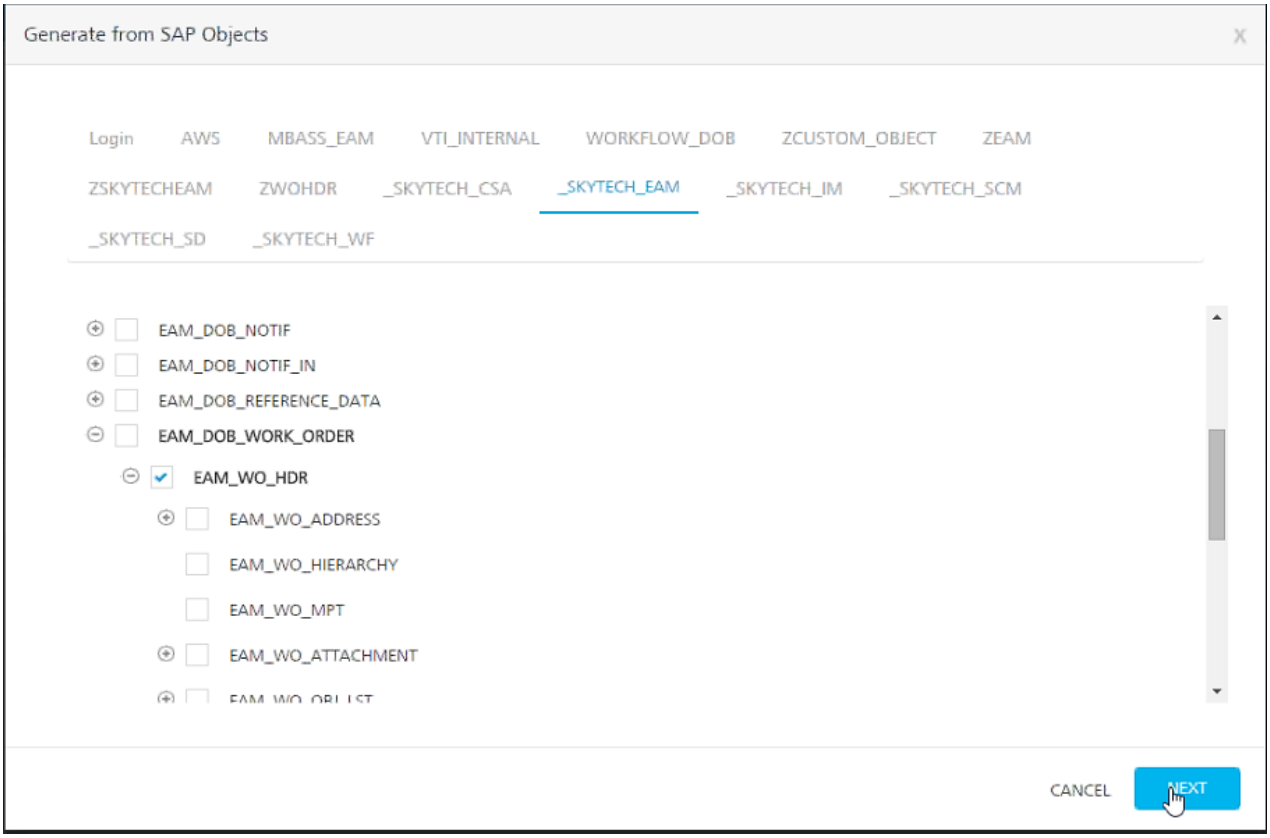
6. Click the **Save & Configure** button.

Configure the Data Model

At this point, you need to configure the data model your service will use.

1. Continuing from #6 in Step 3, click the **Generate** button.
2. In dialog box that appears, select the object service you want to use. The Kony Fabric console then displays a list of objects offered by the selected object service.

3. Choose the object or objects that you want your client app to have access to. In this example, we will select only one object, as shown in the following figure.



4. Click **Next**.

5. In the dialog box that appears, click **Generate**. Your data model is now generated automatically by the system.

6. Click the **Publish** button to publish your Kony Fabric data service app.

2.5.3 Import Kony Quantum Visualizer Apps into Kony Visualizer Enterprise

If you create your app on Kony Visualizer Starter Edition and you decide to import it into Kony Visualizer Enterprise Edition so that you can integrate your app with Kony Fabric backend services, you will need to generate ObjectModel and ObjectModelExtension classes for your app. To do so, use the following steps.

1. From the Kony Visualizer main menu, choose **File** and then **Import**.
2. In the **Import Kony Application** dialog box, ensure that S**elect project root** is selected.
3. Click the **Browse** button, navigate to your Kony Visualizer Starter Edition project, select it, and click **OK**.
4. After the Kony Visualizer Starter Edition project loads, point your mouse cursor at the **Kony Fabric** channel in the Kony Visualizer Enterprise Edition **Project** pane.
5. Click the down arrow that appears and choose **Generate Object Model** from the context menu.
6. If prompted to do so, specify the name of your Kony Fabric app, as well as the object services you want to use in your front-end Kony Visualizer app.

2.5.4 A Sample FormController

The following sample code shows the partial implementation of a `FormController` object. Note that the implementation is in RequireJS format, which is mandatory for Kony Reference Architecture applications.

```
define(
{
    onIPRecievedFromIPControl: function (masterController1, newtext)
    {
        if (null != newtext)
        {
            alert(newtext);
        }
    },

    AS_Button_6c7c9d022bcc4a61a603aa3c89110efe: function
(eventobject)
    {
        this.view.defaultAnimationEnabled = false;
```

```
        this.view.master1.onIPAddressSet =
this.onIPRecievedFromIPControl;
        this.view.master1.IPAddress = "212.212.100.110";
    }
});
```

> ***Note:*** In an MVC project, a top-level FlexContainer is added by default when you create a new template.

3. References

This section provides detailed documentation about the objects and other API elements that the Kony Reference Architecture SDK provides.

Model	Controller	Other
kony.model Namespace	FormController Object	kony.mvc Namespace
	TemplateController Object	kony.mvc.registry Namespace
		Navigation Object

Note that there are no View objects provided in the SDK because, under the Kony Reference Architecture, forms, templates, and masters function as views. You create forms, templates, and masters in Kony Visualizer.

When you're building your Kony Reference Architecture app in Kony Visualizer, Kony Visualizer generates some of your app's objects for you and creates files to store them in. Kony Visualizer uses a default naming scheme for the objects and files it generates. The default naming scheme is important to keep in mind when you're using the **References** section of this SDK's documentation. For instance, the **References** section contains documentation for the following objects.

- FormControllerObject
- TemplateController Object

You will not actually find objects with these names in your code. Instead, under the default naming scheme, you will find names based on the form names you use in Kony Visualizer. That is, if you create a form in Kony Visualizer and name it `frmLogin`, then the FormController object for that form is called `frmLoginController` and it is stored in a file named `frmLoginController.js`. Likewise, if you have a form called `frmMain`, then that form will have a FormController object called `frmMainController` that's stored in a file called `frmMainController.js`. All of your other FormController objects, FormControllerExtension objects, and so on, are similarly named.

There are some objects whose name is exactly what you see in the **References** section. These are as follows.

- Navigation Object
- TemplateController Object

Your code accesses these objects using the exact names you see here.

3.1 FormController Object

The code for the FormController object is created by the code generation tool for you. It communicates with both the models for the data sources and the viewmodels for the forms.

You should not modify the source code for the FormController object. Instead, your app calls the methods that the FormController object provides. However, most apps will need custom business logic. You add that to the FormControllerExtension object rather than the FormController object itself.

The FormController object offers the following.

Methods

getCurrentForm Method

getCurrentFormFriendlyName Method

getPreviousForm Method

getPreviousFormFriendlyName Method

Properties

view

> ***Note:*** If you change the default template of the controller for dependency injection, the methods from the controller will not be displayed as part of intellisense to invoke functions in the Action Editor.

3.1.1 FormController Events

The `FormController` object provides the following events.

> ***Note:*** While using the `this` keyword (for example, `this.view`) in a FormController event in order to point to the current controller, you must ensure that the function is not a **fat arrow** function. Because in fat arrow types of function declarations, the `this` keyword is taken from the parent scope and might not point to the current FormController. For more information on this limitation, click here.

getModel Event

Invoked when the Navigation object retrieves the model for the current FormController object.

Syntax

```
getModel();
```

Parameters

None.

Return Values

Returns the model object that is required for the form.

Remarks

Your app does not directly access the `FormController` object for a form. If your app needs the model associated with the `FormController` object, it can access the model by retrieving it through an instance of the `Navigation` object. This event handler retrieves the model that you want it to use for the form.

Example

```
getModel : function ()
{
```

```
    var model = new CustomFormModel();
    return model;
}
```

onCreateView Event

Called when the controller is ready to create the view.

Syntax

```
onCreateView();
```

Parameters

None.

Return Values

Returns either the file name of the form to use as the view or an instance of the form.

Remarks

Use this method to dynamically select which view to use for the controller when your app has more than one view for a controller. For more information, see Sharing Controllers Between Forms.

Example 1

```
onCreateView : function ()
{
    return "ViewFileName.js");
}
```

Example 2

```
onCreateView : function ()
{
    // Create an instance of the view to return or
    // retrieve the instance from somewhere in your
```

```
    // code where you have stored it. In this example,
    // it's saved in a variable called newInstance.
    return (viewInstance);
}
```

onDestroy Event

Triggered just before a form is destroyed.

Syntax

```
onDestroy();
```

Parameters

None.

Return Values

None

Remarks

Use this event callback handler function to perform cleanup tasks when a form is about to be destroyed.

Example

```
onDestroy : function ()
{
    this.context = null;
    this.model = null;
}
```

onNavigate Event

This event is invoked when you navigate from one form to another. This is a Form Controller event and is used only in Kony Reference Architecture-based projects.

Syntax

```
onNavigate(
    context,
    isBackNavigation)
```

Parameters

context [Object]

A JavaScript object that contains the data that the destination form requires after navigation.

isBackNavigation [Boolean]

This parameter determines whether you have clicked the back button or not. It has the value as *true* when you click the back button and *false* when you do not click the back button.

Read/Write

Read + Write

Remarks

To navigate from one form to another, you must create a Navigation Object. This object navigates to the destination form's controller. The form's controller in turn displays the view of the form.

This event is useful in the following scenarios:

- To prepare data that the destination form requires after the navigation.
- To pause the navigation if any asynchronous calls are in progress.

> ***Note:*** The object that is sent as part of the onNavigate Event is accessible for all form lifecycle events.

Example

```
define({
```

```
    onNavigate: function(context, isBackNavigation) {
        this.context = context;
        this.pauseNavigation();
        kony.net.invokeServiceAsync(url, this.callback1);
    },

    callback1: function(result) {
        this.resumeNavigation();
    }

});
```

Platform Availability

Available on all platforms

---

setModel Event

Invoked while navigating to a new form the model to set the form's updated model object.

Syntax

```
setModel(
    model);
```

Parameters

model

The model object for the new form.

Return Values

None.

Remarks

Use this event callback handler to set a model for the form being navigated to.

Example

```
setModel : function (model)
{
    this.model = model;
}
```

3.1.2 FormController Methods

The FormController object contains the following methods.

getCurrentForm Method

Retrieves the name of the current form.

Syntax

```
getCurrentForm();
```

Parameters

None.

Return Values

Returns a string containing the name of the current form.

Example

```
ver currentForm = this.getCurrentForm();
```

---

getCurrentFormFriendlyName Method

Retrieves the friendly name of the current form.

Syntax

```
getCurrentFormFriendlyName();
```

Parameters

None.

Return Values

Returns a string containing the friendly name of the current form.

Example

```
ver currentFormFriendlyName= this.getCurrentFormFriendlyName();
```

getPreviousForm Method

Retrieves the name of the previous visible form.

Syntax

```
getPreviousForm();
```

Parameters

None.

Return Values

Returns a string containing the name of the previous visible form, or `null` if there is no previous visible form.

Example

```
ver previousForm = this.getPreviousForm();
```

getPreviousFormFriendlyName Method

Retrieves the friendly name of the previous visible form.

Syntax

```
getPreviousFormFriendlyName();
```

Parameters

None.

Return Values

Returns a string containing the friendly name of the previous visible form, or `null` if there is no previous visible form.

Example

```
ver previousFormFriendlyName = this.getPreviousFormFriendlyName();
```

---

pauseNavigation Method

Pauses when navigating from one form to another.

Syntax

```
pauseNavigation();
```

Parameters

None.

Return Values

None.

Remarks

Your app calls this method to pause when navigating from form to form and wait for tasks that need to be completed before the new form is shown. The only time your app can call this function is in the onNavigate event callback handler function, which you must provide. If your app calls it anywhere else, it does nothing.

To resume navigation, your app must call the resumeNavigation method.

Example

```
onNavigate : function(context, isBackNavigation)
{
    this.context = context;
    this.pauseNavigation();
    kony.net.invokeServiceAsync(url, this.callback1);
}

callback1: function(result)
{
    this.resumeNavigation();
}
```

resumeNavigation Method

Resumes the process of navigating from form to form.

Syntax

```
resumeNavigation();
```

Parameters

None.

Return Values

None.

Remarks

When your app is navigating from form to form, it can pause the process of navigation by calling the pauseNavigation method. After navigation has been paused, your app must call the `resumeNavigation` method to continue the navigation process and display the target form. If `pauseNavigation` has not been called, this method does nothing.

> ***Important:*** Failing to call `resumeNavigation` after your app has called `pauseNavigation` may result in your app locking up.

Example

```
onNavigate : function(context, isBackNavigation)
{
    this.context = context;
    this.pauseNavigation();
    kony.net.invokeServiceAsync(url, this.callback1);
}

callback1: function(result)
{
    this.resumeNavigation();
}
```

3.1.3 FormController Properties

The FormController object contains the following properties.

view Property

Contains a reference to the FormController object's view.

Syntax

```
view
```

Type

Object

Read / Write

Read-only

Remarks

Your app can access the view using the syntax `this.view`.

Example

```
var view = this.view;
```

3.2 kony.model Namespace

The kony.model namespace contains the following API elements.

Constants

- kony.model.ExceptionCode Constants
- kony.model.ValidationType Constants

Objects

- kony.model.Exception Object
  - Properties
    - code
    - message
    - name
- kony.model.KonyApplicationContext Object
  - Methods
    - createModel Method
    - login Method
    - logout Method

save

getByPrimaryKey

update

partialUpdate

remove

removeByID

getAll

customVerb

getByCriteria

3.2.1 kony.model Constants

The kony.model namespace provides the following constants.

kony.model.ExceptionCode Constants

Specifies the error code that occurred for the exception.

Constant	Description
kony.model.ExceptionCode.CD_ERROR_CREATE	An error occurred while performing the create operation.
kony.model.ExceptionCode.CD_ERROR_CUSTOMVERB	An error occurred while performing the operation specified by a custom verb.
kony.model.ExceptionCode.CD_ERROR_DELETE	An error occurred while performing the delete operation.

Constant	Description
kony.model.ExceptionCode.CD_ERROR_DELETE_BY_PRIMARY_KEY	An error occurred while performing the delete by primary key operation.
kony.model.ExceptionCode.CD_ERROR_FETCH	An error occurred while performing the fetch operation.
kony.model.ExceptionCode.CD_ERROR_FETCHING_DATA_FOR_COLUMNS	An error occurred while fetching the data for the specified columns.
kony.model.ExceptionCode.CD_ERROR_LOGIN_FAILURE	An error occurred while trying to log in.
kony.model.ExceptionCode.CD_ERROR_UPDATE	An error occurred while performing the update operation.
kony.model.ExceptionCode.CD_ERROR_VALIDATION_CREATE	An error occurred while performing the validation create operation.
kony.model.ExceptionCode.CD_ERROR_VALIDATION_UPDATE	An error occurred while performing the validation update operation.

kony.model.ValidationType Constants

Specifies the type of validation to be performed.

Constant	Description
kony.model.constants.ValidationType.CREATE	The operation creates a record in the backend data source.
kony.model.constants.ValidationType.UPDATE	The operation updates a record in the backend data source.

3.2.2 kony.model Objects

The kony.model provides the following objects.

kony.model.Exception Object

Properties

code

message

name

kony.model.KonyApplicationContext Object

Methods

createModel Method

login Method

logout Method

kony.model.Exception Object

The kony.model.Exception object simplifies exception handling for your app.

Properties

code

message

name

kony.model.Exception Properties

The kony.model.Exception object provides the following properties.

code Property

Specifies the error code.

Syntax

```
code
```

Type

Number

Read / Write

Read only

Remarks

This property can only be set to one of the values in the kony.model.ExceptionCode constants.

---

message Property

Contains a description of the error message.

Syntax

```
message
```

Type

String

Read / Write

Read only

---

name Property

Contains the name of the exception

Syntax

```
name
```

Type

String

Read / Write

Read only

---

kony.model.KonyApplicationContext Object

The kony.model.KonyApplicationContext class contains the following.

Methods

createModel Method

login Method

logout Method

kony.model.KonyApplicationContext Methods

The KonyApplicationContext provides the following methods.

kony.model.ApplicationContext.createModel Method

Creates a model using the specified inputs.

Syntax

```
kony.model.ApplicationContext.createModel(
    entityName,
    serviceName,
    options,
    metadataOptions,
    successCallback,
    errorCallback)
```

Parameters

entityName

A string that specifies the name of the model.

serviceName

A string that contains the name of the object service that the model specified in the *entityName* parameter belongs to.

options

A JavaScript object containing the access options for the service that the app is logging into. This object contains one key, named `access`. The values for this key can be either "online" or "offline".

metadataOptions

An object that contains parameters that the app passes to the Kony Reference Architecture framework while fetching Kony Fabric metadata. The only parameter currently supported is "getFromServer" which can be set to `true` or `false`. A value of `true` forces the model to fetch the metadata from the server rather than retrieve it from the cache. A value of `false` allows the metadata to be fetched from the cache. If "getFromServer" is set to true, then the metadata is refreshed and a new instance is created.

successCallback

A JavaScript function, which you provide, that is automatically invoked when the model object is created. The signature of this function is as follows.

```
successCallback(modelObject);
```

The *modelObject* parameter to this callback function contains the model object that was created.

errorCallback

A JavaScript function, which you provide, that is automatically invoked when the model object is not created. The signature of this function is as follows.

```
loginErrorCallback(error);
```

The *error* parameter to this callback function holds a kony.model.Exception object.

Return Values

Returns the model object.

---

kony.model.ApplicationContext.login

Performs a login operation.

Syntax

```
kony.model.ApplicationContext.login(
    params,
    loginSucCallback,
    loginErrCallback)
```

Parameters

params

A JavaScript object that holds key-value pairs specifying the login authorization information. The keys in this object are as follows.

Key	Value
authParams	A JavaScript object that holds the authorization parameters for logging into the service. For more details, see **Remarks** below.
options	A JavaScript object containing the access options for the service that the app is logging into. This object contains one key, named `access`. The values for this key can be either "online" or "offline".
identityServiceName	A string that specifies the name of the identity service that performs the authentication.

loginSucCallback

A JavaScript function, which you provide, that is automatically invoked when the login is successful. The signature of this function is as follows.

```
loginSuccessCallback();
```

loginErrCallback

A JavaScript function, which you provide, that is automatically invoked when the login is not successful. The signature of this function is as follows.

```
loginErrorCallback(err);
```

The *err* parameter to this callback function contains the error value and error message string for the error that occurred.

Return Values

None.

Remarks

The *params* parameter contains key-value pairs that hold information needed to log into a server. The `authParams` key in the *params* parameter is an object that also contains key-value pairs. The keys it contains are given in the following table.

Key	Value
userid	A string containing the User ID for the account or service that the app is logging into.
password	A string containing the password for the account or service that the app is logging into.

The `options` key in the *params* object is a JavaScript object that specifies the type of access. The key name for selecting the type of access is "access". A value of "online" indicates that the app is logging into a remote service that is not on the device, but rather on the network. The value "offline" means that the service is on the device.

Example

```
var params = {
    "authParams" : {
        "userid" : "MyUserID",
        "password" : "MyPassword"
    },
    options :{"access" :"online"},
    "identityServiceName" : "TheIdentityServiceName"
};

function loginSuccessCallback()
{
    // Your code goes here.
}

function loginErrorCallback(err)
{
    // Your code goes here.
}

kony.model.ApplicationContext.login
(params,loginSuccessCallback,loginErrorCallback);
```

kony.model.KonyApplicationContext.logout Method

Performs a logout operation.

Syntax

```
logout(
    successCallback,
    errorCallback);
```

Parameters

successCallback

A JavaScript function, which you provide, that is automatically invoked when the logout is successful. The signature of this function is as follows.

```
loginSuccessCallback();
```

errorCallback

A JavaScript function, which you provide, that is automatically invoked when the logout is not successful. The signature of this function is as follows.

```
loginErrorCallback(err);
```

The *err* parameter to this callback function contains the error value and error message string for the error that occurred.

Return Values

None

Remarks

This function clears all form controllers, models, and so forth from the `KonyApplicationContext` object's application context. It then logs the app out of Kony Fabric services that it is logged into.

Example

```
var appContext = kony.model.KonyApplicationContext.getAppInstance();
appContext.logout();
```

3.3 kony.mvc Namespace

The `kony.mvc` namespace provides the following API elements.

- Kony mvc namespace enables your app to create a Navigation object, which it uses to navigate from form controller to form controller.

Functions

- Navigation

3.3.1 kony.mvc Functions

The `kony.mvc namespace` contains the following function.

kony.mvc.Navigation Function

Creates an instance of the Navigation object.

Syntax

```
kony.mvc.Navigation(
    friendlyName);
```

Parameters

friendlyName

The friendly name of the form that the Navigation object is to be created for.

Return Values

Returns a Navigation object on success, or `null` on failure.

Remarks

A form can have multiple `Navigation` objects, so it is possible for an app to call this function multiple times on a form.

Example

```
var Navigation = new kony.mvc.Navigation("FormFriendlyName");
```

3.4 kony.mvc.registry Namespace

The `kony.mvc.registry` namespace provides the following API elements

Functions

- add Function
- getViewName Function
- getControllerName Function
- remove Function

3.4.1 kony.mvc.registry Functions

The kony.mvc.registry namespace contains the following functions.

kony.mvc.registry.add Function

Enables you to add a new form name, along with its controller, extension controller, and friendly name, to the registry.

Syntax 1

```
kony.mvc.registry.add("friendlyName", "formId");
kony.mvc.registry.add("friendlyName", "formId", "formController");
kony.mvc.registry.add("friendlyName", "formId", {"controllerName" :
"formController" , "controllerType" : <controllerType>});
kony.mvc.registry.add("friendlyName", "formId", "formController",
"formExtController");
```

Syntax 2

```
kony.mvc.registry.add( "friendlyName", "formId", {"controllerName" : "",
"controllerExtName" : "", "controllerType" : ""});
```

Parameters

friendlyName [string] [Mandatory]

You can assign a "friendly" name to the form, which will be easier for you to remember than the actual formId. The friendlyName string maps the navigation path to the formId and its corresponding controller.

formId [string] [Mandatory]

The name of the form. Given formId as "f1," the Framework automatically searches for the availability of " f1.js" and "f1Controller.js" for initializations.

The following parameters are considered in the third parameter if it is a dictionary (Refer **Syntax 2** and **Example** for more information):

formController [string] [Optional]

The name of the file that contains the form controller.

formExtController [string] [Optional]

The name of the file that contains the form extension controller. You can use form extension controllers to extend the functionality of the form.

controllerExtName [string] [Optional]

The name of the file that contains the extension controller.

controllerType [string] [Optional]

For data-driven forms, this parameter is **kony.mvc.ModelFormController**. You can inherit your own controller from **kony.mvc.FormController** and provide the name here.

Return Values

Returns `true` if the form name is successfully added to the registry, otherwise it returns `false`.

Returns false if the same friendly name has already been registered.

Remarks

- If the *friendlyName* or the *formName* parameter (or both) is an empty string, `null`, or undefined, this function does nothing.
- If the *formController* parameter is `null`, undefined, not provided, or is an empty string, the string in the *formId* parameter is suffixed with the string "Controller." For example, if *formId*contains the string "form1" and the *formController* parameter is not provided, then "form1Controller" will used as the name of the form controller file.

Example

```
kony.mvc.registry.add(
"friendlyName",
"formId",
{"controllerName" : "", "controllerExtName" : "", "controllerType" : ""});
```

---

kony.mvc.registry.getViewName

Retrieves the form or template name from the registered friendly name.

Syntax

```
kony.mvc.registry.getViewName(
    friendlyName);
```

Parameters

friendlyName

The friendly name of the form to retrieve the name from.

Return Values

Returns a string containing the form name if the friendly name is found in the registry, or `null` if it is not found.

Example

```
formName = kony.mvc.registry.getViewName("Form1");
```

kony.mvc.registry.getControllerName

Retrieves the controller name from the registered friendly name.

Syntax

```
kony.mvc.registry.getControllerName(
    friendlyName);
```

Parameters

friendlyName

The friendly name of the form to retrieve the name from.

Return Values

Returns a string containing the controller name if the friendly name is registered and the controller name is found. Returns a string containing "<viewName>.Controller" if the friendly name is registered and the controller name is not found. Returns `null` if the friendly name is not registered.

Example

```
kony.mvc.registry.getControllerName("FriendlyName");
```

kony.mvc.registry.remove

Removes the name of a form controller from the registry.

Syntax

```
kony.mvc.registry.remove(
    friendlyName);
```

Parameters

friendlyName

The friendly name of the form whose controller is to be removed.

Return Values

None.

Example

```
kony.mvc.registry.remove(FriendlyName");
```

3.5 Navigation Object

The `Navigation` object provides your app with the ability to navigate from form to form. It does this by navigating to a target form controller, which then displays the form's view. To create a Navigation object, your app must call the kony.mvc.Navigation function.

Methods

navigate Method

3.5.1 Navigation Methods

The Navigation object provides the following methods.

getModel

Retrieves the model for the form.

Syntax

```
getModel();
```

Parameters

None.

Return Values

Returns a JavaScript object that contains the model for the form. The model is either the model that the app previously set or the model that is retrieved from the `FormController`. This method triggers the FormController.getModel event.

Remarks

This method retrieves the form's model.

Example

```
var formModel = navObject.getModel();
```

---

navigate Method

Performs a form navigation.

Syntax

```
navigate(
    params);
```

Parameters

params

A JavaScript object containing key/value pairs that are passed to the target form from the current form.

Return Values

None.

Remarks

The *params* parameter is passed to all of the lifecycle events, such as preShow, postShow, and init, on the target form.

Example

```
var x = new kony.mvc.Navigation("friendlyName/formName", model);
x.navigate(params);
```

---

setModel

Sets the model for the form being navigated to.

Syntax

```
setModel(
    newModel
```

Parameters

newModel

A JavaScript object that holds the model for the target form.

Return Values

None.

Remarks

This method sets the model of the target form, which is the form being navigated to. It triggers the FormController.setModel event.

3.6 TemplateController Object

The code for the TemplateController object is created by the code generation tool for you. It communicates with both the models for the data sources and the viewmodels for the forms.

You should not modify the source code for the TemplateController object. Instead, your app calls the methods that the TemplateController object provides.

When your app passes a template as a string to a widget, the widget creates the corresponding TemplateController object when it needs the template's view. It automatically searches for a TemplateController name that is mapped in the registry for that template. If it doesn't find a mapping, it searches for a template controller whose file name is of the form <templateName>Controller.js, where <templateName> is the name of the template. It then creates the TemplateController object for that template.

The TemplateController object offers the following.

Methods

executeOnParent Method

Properties

view Property

3.6.1 TemplateController Events

The TemplateController object supports the following events.

onCreateView Event

Called when the controller is ready to create the view.

Syntax

```
onCreateView();
```

Parameters

None.

Return Values

Returns either the file name of the template to use as the view or an instance of the template.

Remarks

Use this method to dynamically select which view to use for the controller when your app has more than one view for a controller. For more information, see Sharing Controllers Between Forms.

Example 1

```
onCreateView : function ()
{
    return "ViewFileName.js");
}
```

Example 2

```
onCreateView : function ()
{
    // Create an instance of the view to return or
    // retrieve the instance from somewhere in your
    // code where you have stored it. In this example,
    // it's saved in a variable called newInstance.
    return (viewInstance);
}
```

onDestroy Event

Triggered just before a template is destroyed.

Syntax

```
onDestroy();
```

Parameters

None.

Return Values

None

Remarks

Use this event callback handler function to perform cleanup tasks when a template is about to be destroyed.

Example

```
onDestroy : function ()
{
    this.context = null;
    this.model = null;
}
```

onViewCreated

Triggered when the view is created.

Syntax

```
onViewCreated();
```

Parameters

None.

Return Values

None.

Remarks

This method is automatically invoked just after the onCreateView event has finished and the template's view has been created. Developers can use this method to configure the template.

Example

```
onViewCreated: function ()
{
    this.view.addGestureRecognizer(
```

```
        constants.GESTURE_TYPE_SWIPE,
        {fingers: 1},
        function(widgetRef, gestureInfo, context)
        {
            alert("Swipe Gesture");
        }
    );
}
```

3.6.2 TemplateController Methods

The TemplateController object provides the following method.

executeOnParent Method

Executes the specified method of the parent object.

Syntax

```
executeOnParent(
    methodName,
    methodParams);
```

Parameters

methodName

A string containing the name of the parent's method.

methodParams

An optional list of parameters to pass to the method specified by the *methodName* parameter.

Return Values

None.

Remarks

The parent of this object is always a FormController object. This method should only be called from sub-view controllers.

Example

```
this.executeOnParent("func1", "param1", "param2");
```

---

getCurrentView Method

Retrieves the current view for the template controller.

Syntax

```
getCurrentView();
```

Parameters

None.

Return Values

Returns the template controller's view.

Example

```
var currentView = tmpController.getCurrentView();
```

3.6.3 TemplateController Properties

The TemplateController object contains the following property.

view Property

Contains a reference to the TemplateController object's view.

Syntax

```
view
```

Type

Object

Read / Write

Read-only

Remarks

Your app can access the view using the syntax `this.view`.

Example

```
var view = this.view;
```

3.7 Deprecated

The API elements in this section are deprecated and should not be used in the development of new software. The documentation in this section is provided to help with the maintenance of legacy software.

3.7.1 kony.sdk.mvvm Namespace

The kony.sdk.mvvm namespace is now deprecated. New software should not use anything in this namespace. Instead, use the kony.model namespace.

Documentation on the kony.sdk.mvvm namespace is provided here to assist with maintaining legacy software. The kony.sdk.mvvm namespace contains the following API elements.

Constants

- kony.sdk.mvvm.OperationType Constants

Objects

- kony.sdk.mvvm.KonyApplicationContext Object
  - Methods
    - appServicesLogin Method
    - dismissLoadingScreen Method
    - getAllFormControllers Method
    - getAppInstance Method
    - getFactorySharedInstance Method
    - getFormController Method
    - getMetadataStore Method
    - getModel Method
    - getObjectService Method
    - init Method
    - logout Method
    - showLoadingScreen Method

kony.sdk.mvvm Constants

The kony.sdk.mvvm namespace provides the following constants.

kony.sdk.mvvm.OperationType Constants

Specifies the operation to be performed.

Constant	Description
kony.sdk.mvvm.OperationType.ADD	Add a data model object.
kony.sdk.mvvm.OperationType.FILTER_BY_PRIMARY_KEY	The operation is filtered by the data object's primary key.
kony.sdk.mvvm.OperationType.NO_FILTER	The operation is not filtered.

Remarks

Use these constants to specify data model operations when performing form navigation. For more information, see kony.sdk.mvvm.NavigationObject Object.

---

kony.sdk.mvvm Objects

The kony.sdk.mvvm provides the following objects.

Objects

- kony.sdk.mvvm.KonyApplicationContext Object
  - Methods
    - appServicesLogin Method
    - dismissLoadingScreen Method
    - getAllFormControllers Method
    - getAppInstance Method
    - getFactorySharedInstance Method
    - getFormController Method
    - getMetadataStore Method
    - getModel Method

- getObjectService Method
- init Method
- logout Method
- showLoadingScreen Method

kony.sdk.mvvm.KonyApplicationContext Object

The kony.sdk.mvvm.KonyApplicationContext class contains the following.

Methods

- appServicesLogin Method
- dismissLoadingScreen Method
- getAllFormControllers Method
- getAppInstance Method
- getFactorySharedInstance Method
- getFormController Method
- getMetadataStore Method
- getModel Method
- getObjectService Method
- init Method
- logout Method
- showLoadingScreen Method

kony.sdk.mvvm.KonyApplicationContext Methods

The KonyApplicationContext provides the following methods.

kony.sdk.mvvm.KonyApplicationContext.appServicesLogin Method

Performs initialization, registration, and login services for an app.

Syntax

```
kony.sdk.mvvm.KonyApplicationContext.appServicesLogin(
    params,
    loginSuccessCallback,
    loginErrorCallback);
```

Parameters

params

An object containing the authorization parameter and options, as well as the synchronization configuration information. This object uses the following format.

- authParams: An object containing a userID and a password.
- options: An object specifying the type of access that the app uses. The object contains one key, called "access", which can have a value of either "online" or "offline".
- syncOptions: An object containing synchronization configuration information.

loginSuccessCallback

An optional event handler function that is called upon success.

loginErrorCallback

An optional event handler function that is called if the `appServicesLogin` function fails.

Return Values

None.

Remarks

This method performs initialization, configuration, and login services. It calls the kony.sdk.mvvm.KonyApplicationContext.init method. If your app invokes `appServicesLogin`, it does not need to call `kony.sdk.mvvm.KonyApplicationContext.init`. The `appServicesLogin` method also registers and starts the AuthenticationServiceManage and MetadataServiceManager objects. Therefore, this app must have identity services configured prior to calling `appServicesLogin`.

In the case of an app that uses offline storage, this method also registers and starts the `SyncManager` object.

Your app calls this method directly by using its fully-qualified name.

Example

```
params = {
    "authParams" : {
        "userid" : "Aard",
        "password" : "Vark"
    },
    "options" :{
        {"access":"online"}
    },
    "syncOptions" : {
        "syncConfig":{
            "batchsize" : 10000000,
            // Other sync configuration params.
        }
    }
}

kony.sdk.mvvm.KonyApplicationContext.appServicesLogin(params);
```

kony.sdk.mvvm.KonyApplicationContext.dismissLoadingScreen Method

Dismisses a loading screen that was previously displayed using the showLoadingScreen method.

Syntax

```
dismissLoadingScreen();
```

Parameters

None.

Return Values

None.

Remarks

Typically, your app calls the showLoadingScreen method to display a screen that lets the user know that it is loading data and that the user must wait. After the data has been loaded, your app calls this method to dismiss the loading screen.

Example

```
var appContext = kony.sdk.mvvm.KonyApplicationContext.getAppInstance();
appContext.dismissLoadingScreen();
```

---

kony.sdk.mvvm.KonyApplicationContext.getAllFormControllers Method

---

Retrieves controler objects for every form in the current application context.

Syntax

```
getAllFormControllers();
```

Parameters

None.

Return Values

Returns an object containing all of the form controllers in the application context. The object contains a group of key-value pairs in which the form ID is the key and the value is the controller for the specified form.

Example

```
var appContext = kony.sdk.mvvm.KonyApplicationContext.getAppInstance();
var allControllers = appContext.getFormControllers();
```

kony.sdk.mvvm.KonyApplicationContext.getAppInstance Method

Retrieves in instance of a KonyApplicationContext object.

Syntax

```
kony.sdk.mvvm.KonyApplicationContext.getAppInstance();
```

Parameters

None.

Return Values

Returns a kony.sdk.mvvm.KonyApplicationContext object.

Remarks

Your app calls this function any time it needs an instance of the global KonyApplicationContext object.

Your app calls this method directly by using its fully-qualified name.

Example

```
var appContext = kony.sdk.mvvm.KonyApplicationContext.getAppInstance();
```

kony.sdk.mvvm.KonyApplicationContext.getFactorySharedInstance Method

Retrieves an instance of the AppFactory object.

Syntax

```
getFactorySharedInstance();
```

Parameters

None.

Return Values

Returns a kony.sdk.mvvm.AppFactory object.

Remarks

Apps use the AppFactory object to instantiate instances of classes in the kony.sdk.mvvm namespace.

Example

```
var appContext = kony.sdk.mvvm.KonyApplicationContext.getAppInstance();
var appFactoryInstance = appContext.getFactorySharedInstance();
```

---

kony.sdk.mvvm.KonyApplicationContext.getFormController Method

---

Retrieves the form controller for the specified form.

Syntax

```
getFormController(
    formId)
```

Parameters

formID

A string containing the ID of the form.

Return Values

Returns the controller associated with the specified form.

Example

```
var appContext = kony.sdk.mvvm.KonyApplicationContext.getAppInstance();
appContext.getFormController(formId);
```

---

kony.sdk.mvvm.KonyApplicationContext.getMetadataStore Method

Retrieves a kony.sdk.mvvm.MetadataStore object from the application's context.

Syntax

```
kony.sdk.mvvm.KonyApplicationContext.getAppInstance().getMetadataStore();
```

Parameters

None.

Return Values

Returns the MetadataStore object from the app's context.

Example

```
var appContext = kony.sdk.mvvm.KonyApplicationContext.getAppInstance();
var appMetadataStore = appContext.getMetadataStore();
```

kony.sdk.mvvm.KonyApplicationContext.getModel Method

Retrieves the specified model.

Syntax

```
getModel(
    entityName,
    serviceName,
    options);
```

Parameters

entityName

A string containing the name of the model.

serviceName

A string that contains the name of the object service that the model in the *entityName* parameter belongs to.

options

An object that defines the access options for the model. The object contains one key, called "access", which can have a value of either "online" or "offline".

Return Values

Returns the specified model.

Remarks

Apps based on theKony Reference Architecture SDK use models to abstract the access to data sources. Data sources can include both local data storage on the device and remote data services that your app accesses across the Internet. For each data source, there is a model that provides a standardized interface to the data source. This function returns the model associated with a data source.

Example

```
var modelName = "MyModel";
var serviceName = "MyKony FabricSerice";
var serviceOptions = {"access":"online"};
var appContext = kony.sdk.mvvm.KonyApplicationContext.getAppInstance();
var currentModel = appContext.getModel
(modelName,serviceName,serviceOptions);
```

---

kony.sdk.mvvm.KonyApplicationContext.getObjectService Method

Retrieves the specified object service.

Syntax

```
getObjectService(
    options,
    objectServiceName);
```

Parameters

options

A JavaScript object that specifies the access options for the service. The object contains one key, called "access", which can have a value of either "online" or "offline".

objectServiceName

The name of the object service to retrieve.

Return Values

Returns the specified object service.

Example

```
var serviceName = "MyKony FabricSerice";
var serviceOptions = {"access":"online"};
var appContext = kony.sdk.mvvm.KonyApplicationContext.getAppInstance();
var onlineObjSer = appContext.getObjectService(serviceOptions,serviceName);
```

---

kony.sdk.mvvm.KonyApplicationContext.init Method

Initializes an instance of a KonyApplicationContext object.

Syntax

```
kony.sdk.mvvm.KonyApplicationContext.init();
```

Parameters

None.

Return Values

None.

Remarks

You must call the `init` method before you can use any other method that this object provides. If you do not call this method first, all of the other methods of this class will return an error.

Your app calls this method directly by using its fully-qualified name.

Example

```
kony.sdk.mvvm.KonyApplicationContext.init();
```

kony.sdk.mvvm.KonyApplicationContext.logout Method

Performs a logout operation.

Syntax

```
logout(
    successCallback,
    errorCallback);
```

Parameters

successCallback

An event handler function that is called when the logout operation is successful.

errorCallback

An event handler function that is called when the logout operation results in an error.

Return Values

None

Remarks

This function clears all form controllers, models, and so forth from the KonyApplicationContext object's application context. It then logs the app out of Kony Fabric services that it is logged into.

Example

```
var appContext = kony.sdk.mvvm.KonyApplicationContext.getAppInstance();
appContext.logout();
```

kony.sdk.mvvm.KonyApplicationContext.showLoadingScreen Method

Displays a loading screen with the specified text.

Syntax

```
showLoadingScreen(
    text);
```

Parameters

text

A string containing the text to display

Return Values

None.

Remarks

You app calls this method when it needs to display a screen informing the user that data is loading. The typical use case for this method is when your app is getting data from a remote service across the Internet.

This method displays the loading screen with the message specified in the *text* parameter and then returns. When the data is loaded, call the dismissLoadingScreen method to dismiss the loading screen.

Example

```
var text = "Quite please, I'm thinking..."
var appContext = kony.sdk.mvvm.KonyApplicationContext.getAppInstance();
appContext.showLoadingScreen(text);
```