Kinetise Source Code

1. Getting Started

This document describes the overview of the architecture of the application source code generated by Kinetise. It explains most essential terms and concepts and provides example scenarios for customization. The scenarios assume that you keep working in Kinetise Editor, using Custom Widgets or custom JavaScript actions, which do not have any effect from start, but for which you can provide your own implementation to provide missing functionality or framework integration.

The last part of the document provides hands-on tutorials for specific scenarios for both Android and iOS codebase.

2. Application Architecture

Overview

Kinetise produces cross-platform applications working on iOS and Android operating systems. Kinetise defines its own application abstraction layer that is natively implemented on both platforms. Uniform application description is processed at runtime and translated to native code that achieves same behavior on all different mobile devices.

Application package

Application package contains all the embedded assets (mainly images) application uses and most importantly *project.xml* file that describes entire application structure. *project.xml* is parsed at runtime to instantiate user interface views and configure application behavior.
When application is edited in Kinetise Editor a new application package is created. It is enough to replace the package in the project structure to apply changes into the project.

Descriptors

Descriptors are data objects representing contents of *project.xml* file (so all the settings originating from Kinetise Editor). They represent and describe all possible views managed by the app: screens, overlays and controls. Descriptors tell how to layout them, render them, and what are the data contents and reaction to events. In general, descriptors form a runtime model of an application.

Based on the descriptors application creates actual screens and view structure as needed.

Screens

Application consists of any number of screens. At most one screen is active and displayed at a time. Screen renders over entire available phone display. Screen is divided into three sections: header (optional, located at screen top), body (taking whole available space between other two sections, vertically scrollable) and navigation panel (optional, located at screen bottom). Every section may contain any number of child views, so-called *controls*.
Kinetise includes a set of standard navigation operations allowing to change currently active screen (including transition animations). Screen state (as a descriptor object) is saved in the history stack every time a screen is changed and is used when app returns back to it.
Every screen has its unique ID. There are three screens with special handling.

- *Splash screen* – the first screen of the application, by default shown for a short period of time, after which *Main screen* is loaded.
- *Login screen* – if defined, it is first screen of the application (replaces *Splash screen*) and the only one where user logged-in operations are permitted. Also, app will return to *Login screen* after user is logged out from the app.

- *Main screen* – the screen that will be loaded after *Splash screen* timeout or (by default) after user login on *Login screen*. Application with Login screen will start on *Main screen* if logged-in user session exists.

Screen may be entered “with context” meaning that a specific data item is a context for the screen. This allows controls on the screen to load data from the context data item. This is used to achieve *Detail screen* behavior.

Overlays

Application may define a list of overlays that act as menus, popups, etc.. At most one overlay can be displayed on top of the currently active screen. It can appear from any side of the screen (or be centered) and it can slide over the screen or move it alongside. Overlay may be of any size and may contain any child controls structure. Same overlay may be displayed over any screen.

Layout system

Kinetise-based applications use own measure/layout system to ensure views look the same on any platform. The main assumptions of the layout system are:

- View hierarchy is built out of *controls*
- A *control* may be a *container*. Containers may contain any number of child controls. Non-container controls, in turn, cannot have any children.
- *Containers* are either *horizontal, vertical* or *thumbnail*. Horizontal and vertical containers lay out their children linearly, from left-to-right (horizontal) or from top-to-bottom (vertical). Additionally, controls may define horizontal and vertical alignment that adjust their positioning inside a container. Thumbnail container lays out it children from left to right until there is space, and then wraps to the next line and proceeds like that until all children are laid out.
- Dimensions are expressed in the following units:
    - MIN – special value indicating that dimension should as small as possible to fit view contents
    - MAX – special value indicating that dimension should as big as remaining space in its parent (or equal to parent dimension in case of scrolled parents)
    - KPX – unit indicating 1/1000 part of the size of the shorter edge of the available display
    - % - unit indicating percent of parent’s corresponding dimension
- Measure and layout calculations are conducted first on the descriptors layer, meaning that after calculations pass descriptors hold actual size and position information (expressed in on-screen pixels) for all the views. Then, the controls take the calculated values and use them to render actual views on the screen.

Controls

Controls are special Kinetise views that represent Widgets from Kinetise Editor. Every control share some common properties like: width, height, paddings, margins, size and color of optional border, optional rounded corners, background color or image, optional on-click action. Content of the control is specific to the control itself and may be any combination of native views. Every control has its dedicated descriptor that holds all the common parameters as well as all control-specific parameters coming from *project.xml* and the current control data state.
There are two specific control types, i.e. data feed controls and form controls, described below.

Data feed controls

Data feed controls represent controls that can retrieve data from external source, process and display that data. Data feed control defines URI of the data source, optional request parameters, type of expected data (JSON/XML) and hints on how data should be read from the raw data source response.
Data response is parsed into list of items, which are then rendered by the control. Most typical data feed control is List control that creates dynamically separate item control (container) for each item received from data source.

Form controls

Form controls represent controls that can take input from the user like text inputs, checkboxes, dropdown selectors etc. Logically, form controls are always grouped inside a container that also contains a *Send* button. Action attached to the *Send* button collects all the values from the form controls and sends them to the data target defined in the action itself (most typically, to REST API endpoint). Data entry for every control is identified by control's *form ID* that must be unique in the scope of the form container.
Notable features of form controls are value validation and binding of a initial value.

Variables

Some of the descriptor properties are expressed as *variables* - objects that can be resolved at runtime to return dynamic value. Kinetise applications use variables to bind dynamic data to a specific control at a given time.
Internally variables are expressed as textual scripts (either KinetiseScript, or JavaScript, coming from *project.xml*) that are evaluated at runtime, translated into internal implementations and executed to retrieve value. Most Scripts are created by Kinetise Editor behind the scenes, but it is possible to inject user-created scripts as well.
Typical usage is a variable that retrieves field value from a data feed item.

Actions

Actions are objects attached to Kinetise events and executed when the event happens. Most typical usage is an on-click action of every Kinetise control.

Internally actions are expressed as textual scripts (either KinetiseScript, or JavaScript, coming from *project.xml)* that are evaluated at runtime, translated into internal implementations and executed. Scripts are created by Kinetise Editor behind the scenes, but it is possible to inject user-created scripts as well.

KinetiseScript

KinetiseScript is a simple script language used to encode dynamic actions and variables to be resolved at runtime. It can be recognized by the following scheme: [d]…[/d]. KinetiseScript is used internally by Kinetise Editor and is not advised to be used directly or modified manually in any way. For user-supplied scripts JavaScript interface is recommended.

Localization support

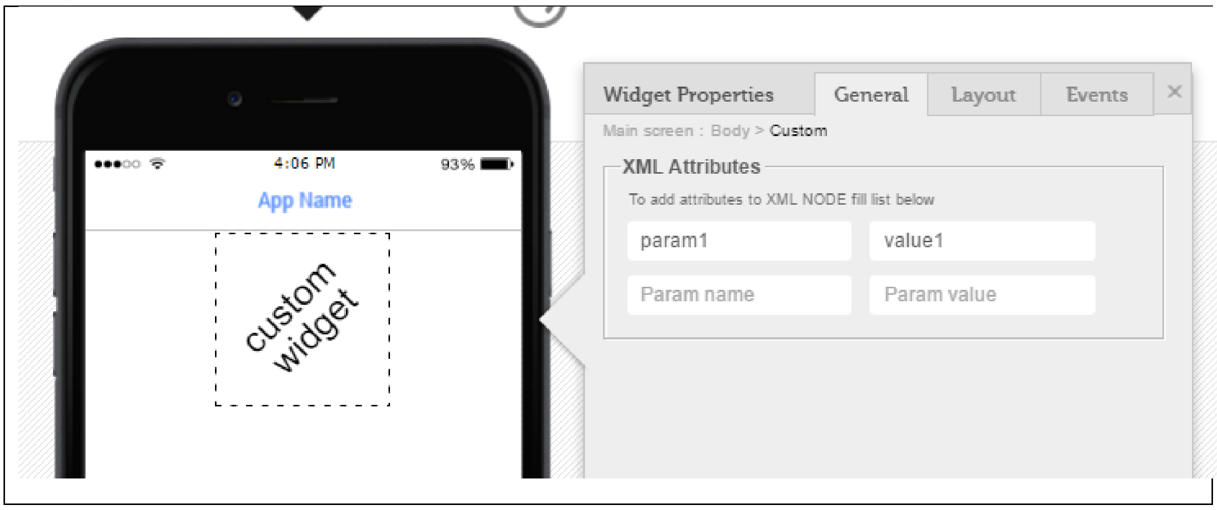
Kinetise allows applications to switch active language at runtime, at any time, without restarting of the application. Application package contains a translations dictionary for every supported language as separate *strings.json* file for each language.

3. Customization

Specific architecture of Kinetise-based applications require specific approach for its customization. We strongly recommend keeping the existing architecture in place and using the extension capabilities that are supported by Kinetise. You may attempt whatever customization and application extension as you feel appropriate, although we suggest starting with the supported scenarios. They include:

Providing custom control

Create custom control class and implement custom look and behavior. Custom controls can be added to *project.xml* from Kinetise Editor (by dragging Custom Widget to the screen) to ease up the integration. You need to give such Widget a *name* and use the same name in code, as presented below, in sections for specific platforms. Additionally, Kinetise Editor allows to define a set of key-value pairs for every Custom Widget. They will be accessible by a custom control class as well.

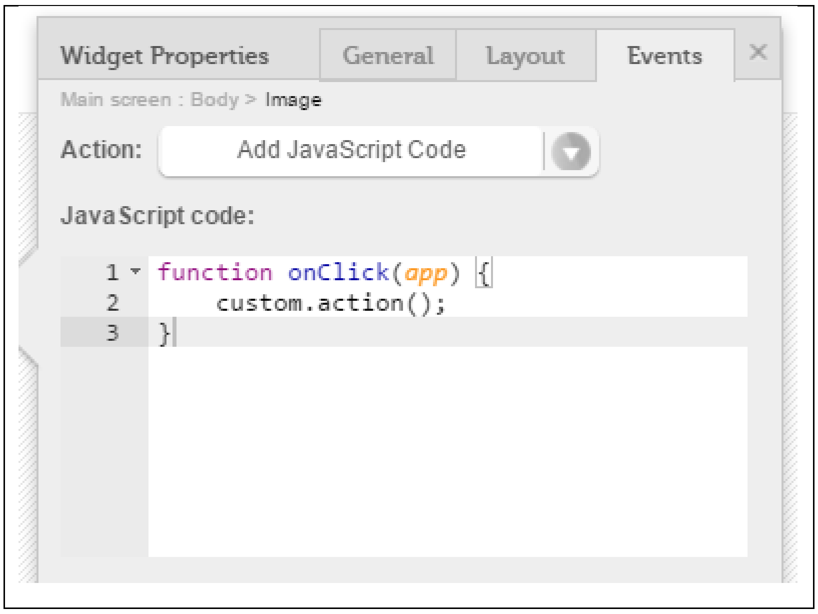


Providing custom screen

This scenario is almost the same as the one above, with the only difference that you should use a single Custom Widget on the entire screen and set its dimensions to MAX/MAX. Such Widget will fill the screen and the corresponding custom control may contain children view structure is necessary to represent custom screen.

Providing custom JavaScript actions

Provide implementation for custom JavaScript action that can be invoked from script defined for any action using Kinetise Editor. When action will be executed by the application, the corresponding script will be evaluated and your custom action executed. All custom actions should be invoked via to `custom` JavaScript object (acting as namespace). Example of invoking custom action as defined in Kinetise Editor:



Action: Add JavaScript Code

JavaScript code:

```
function onClick(app) {
    custom.action();
}
```

Use Kinetise API of actions

Most of the crucial operations Kinetise components may do are implemented as an internal Kinetise API. Custom code may invoke this API as needed to access most of the Kinetise features. API is access via *Action Manager* singleton class.

4. Customization Tutorial – Android

This section describes how to start with each of the basic customization scenarios Kinetise supports. The below content is specific for Android version of the project.

General notes

Android project is a usual Gradle-based project that can be opened with Android Studio. Project contains *kinetise* module, which constitutes core Kinetise codebase. We suggest to not modify that code if not necessary, as it will have global effect on the application. There is also *app* module, where all your specific customizations should be placed.

Providing custom control

Description

Given you added a Custom Widget with *name* to your application using Kinetise Editor, the minimum of two steps are needed to have the custom control working:

1. Create a class for your control extending *AGCustomControlView*
2. Register your class in *ViewFactoryManager* providing Custom Widget *name* (as provided in Kinetise Editor, but prefixed with `controlcustom` value)

Location

It is not strictly enforced, although we suggest to:

1. Add custom control classes to *com.kinetise.app.views* package of *app* module
2. Register view in *com.kinetise.app.App onCreate()* method.

Example

Custom control example presenting a control using Android AppCompat Button and setting its text to value provided in Kinetise Editor as additional property of Custom Widget under key *text*.

```
package com.kinetise.app.views;

import (...)

public class CustomCompatButton extends AGCustomControlView {
    private static final String TEXT_ATTRIBUTE = "text";

    private final View mButton;

    public CustomCompatButton(SystemDisplay display,
                              AGCustomControlDataDesc desc) {
        super(display, desc);
        mButton = createCompatButton();
        addView(mButton);
    }

    private View createCompatButton() {
        AppCompatButton button = new AppCompatButton(mDisplay.getActivity());
        button.setText(mDescriptor.getAttribute(TEXT_ATTRIBUTE));
        return button;
    }
}
```

Registering the above custom control class to match *compatbutton* name provided in Kinetise Editor (actual XML node name in *project.xml* is *controlcustomcompatbutton*).

```
public class App extends KinetiseApplication {

    @Override
    public void onCreate() {
        super.onCreate();

        ViewFactoryManager.registerCustomView("controlcustomcompatbutton",
                                              CustomCompatButton.class);
    }
}
```

Providing custom screen

It essentially the same as for *Providing custom control* scenario, assuming that control may have any child view hierarchy.

Providing custom JavaScript actions

Description

Kinetise used *DuckTape* library for JavaScript engine. To provide implementation for custom actions you need to extend *ICustomJSInterface* interface with the custom actions needed and provide their implementation in *CustomJSInterface* class. Follow *DuckTape* rules for defining the interface.

Location

com.kinetise.app.javascript.ICustomJSInterface
com.kinetise.app.javascript.CustomJSInterface

Example

ICustomJSInterface.java

```
package com.kinetise.app.javascript;

public interface ICustomJSInterface {
    void action();
    String action1();
    void action2(Integer x);

    //define any additional custom actions here:
}
```

CustomJSInterface.java

```
package com.kinetise.app.javascript;

public class CustomJSInterface implements ICustomJSInterface {

    @Override
    public void action() {
        //provide implementation
    }
    @Override
    public String action1() {
        //provide implementation
        return "";
    }

    @Override
    public void action2(Integer x) {
        //provide implementation
    }

    //provide implementation for any additional custom methods you added

}
```

Use Kinetise API of actions

Description

Kinetise API is accessed via *ActionManager.class*, which is a singleton. It implements all methods used by internal Kinetise actions.

Example

```
ActionManager.getInstance().goToScreen("screen12345", AGScreenTransition.FADE);
```

5. Customization Tutorial – iOS

This section describes how to start with each of the basic customization scenarios Kinetise supports. The below content is specific for iOS version of the project.

Providing custom control

Description

You can add custom controls to the application. Controls implementation is devided to data model known as control descriptor and control view. Control descriptor is created based on application xml file in XML category. Descriptor must inherit from *AGControlDesc* and view must inherit from *AGControl*. Every control must be registered in *AGParser*.

Location

Kinetise/Helpers/Parser.m
Kinetise/Application/Controls/Custom/AGCustomDesc.h
Kinetise/Application/Controls/Custom/ AGCustomDesc.m
Kinetise/Application/Controls/Custom/ AGCustomDesc+XML.h
Kinetise/Application/Controls/Custom/ AGCustomDesc+XML.m
Kinetise/Application/Controls/Custom/ AGCustom.h
Kinetise/Application/Controls/Custom/ AGCustom.m

Example

The following example presents all steps needed to create new custom control.

1. Register control in *AGParser.m* classWithName method. Use *name* as provided for Custom Widget in Kinetise Editor prefixed with `controlcustom`.

```
#import "AGCustomButtonDesc.h"
…

+ (Class)classWithName:(NSString *)name {
    NSMutableDictionary *classMapper = [[NSMutableDictionary alloc] init];

     …

    // Register custom button
    [classMapper setObject:NSStringFromClass([AGCustomButtonDesc class])
forKey:@"controlcustombutton"];

    …
}
```

2. Add control data model - descriptor. Remember to implement copy method.

AGCustomButtonDesc.h:

```
#import "AGControlDesc.h"

@interface AGCustomButtonDesc : AGControlDesc

@property(nonatomic, copy) NSString *text;

@end
```

AGCustomButtonDesc.m:

```
#import "AGCustomButtonDesc.h"

@implementation AGCustomButtonDesc

@synthesize text;

#pragma mark - Initialization

- (void)dealloc {
    [super dealloc];
    [text release];
}

#pragma mark - Copying

- (id)copyWithZone:(NSZone *)zone {
    AGCustomButtonDesc *obj = [super copyWithZone:zone];

    obj.text = text;

    return obj;
}

@end
```

3. Control descriptor is data model but also acts as bridge between view and XML. Add XML category to store some information based on control XML. This step is optional.

AGCustomButtonDesc+XML.h:

```
#import "AGCustomButtonDesc.h"

@interface AGCustomButtonDesc (XML)

@end
```

AGCustomButtonDesc+XML.m:

```
#import "AGCustomButtonDesc+XML.h"
#import "AGControlDesc+XML.h"

@implementation AGCustomButtonDesc (XML)

#pragma mark - Initialization

- (id)initWithXML:(GDataXMLNode *)node {
    self = [super initWithXML:node];

    // Initialize based on XML representation
    self.text = [node stringValueForXPath:@"@text"];

    return self;
}

@end
```

4. Finally add control view.

AGCustomButton.h

```
#import "AGControl.h"

@interface AGCustomButton : AGControl

@end
```

AGCustomButton.m

```
#import "AGCustomButton.h"
#import "AGCustomButtonDesc.h"

@interface AGCustomButton()
@property(nonatomic, retain) UIButton *button;
@end

@implementation AGCustomButton

@synthesize button;

#pragma mark - Initialization

- (void)dealloc {
    [super dealloc];
    [button release];
}

- (id)initWithDesc:(AGCustomButtonDesc *)descriptor_ {
    self = [super initWithDesc:descriptor_];

    // Initialize view based on descriptor
    // Add subviews to self.contentView or customize content view by overriding
contentClass method
    self.button = [UIButton buttonWithType:UIButtonTypeSystem];
    [button setTitle:descriptor_.text forState:UIControlStateNormal];
    [self.contentView addSubview:button];

    return self;
}

- (Class)contentClass {
    return [UIView class];
}

#pragma mark - Layout

- (void)layoutSubviews {
    [super layoutSubviews];

    // Layout subviews
    self.button.frame = self.contentView.bounds;
}

@end
```

Providing custom screen

It essentially the same as for *Providing custom control* scenario, assuming that control may have any child views hierarchy.

Providing custom JavaScript actions

Description

Kinetise custom JavaScript actions mechanics uses *JavaScriptCore* framework.
AGJSCustom protocol wraps all custom java script actions.
Add your custom properties or functions in *AGJSCustom* protocol following *JavaScriptCore* framework recommendations.
More about JavaScriptCore https://developer.apple.com/reference/javascriptcore

Location

Kinetise/Helpers/ActionManager/JavaScript/AGJSCustom.h
Kinetise/Helpers/ActionManager/JavaScript/AGJSCustom.m

Example

AGJSCustom.h

```
#import <JavaScriptCore/JavaScriptCore.h>

@protocol AGJSCustom <JSExport>
- (void)action;
@end

@interface AGJSCustom : NSObject <AGJSCustom>

@end
```

AGJSCustom.m

```
#import "AGJSCustom.h"

@implementation AGJSCustom

- (void)action {
    // Put your custom action implementation here
}

@end
```

Use Kinetise API of actions

Description

Access all application actions through AGApplication singleton. Actions are grouped in categories.

Location

Kinetise/Helpers/ActionManager/JavaScript/AGActionManager.h
Kinetise/Helpers/ActionManager/JavaScript/AGActionManager+Actions.h
Kinetise/Helpers/ActionManager/JavaScript/AGActionManager+Navigation.h
Kinetise/Helpers/ActionManager/JavaScript/AGActionManager+Forms.h
Kinetise/Helpers/ActionManager/JavaScript/AGActionManager+Authorization.h
Kinetise/Helpers/ActionManager/JavaScript/AGActionManager+Controls.h

Kinetise/Helpers/ActionManager/JavaScript/AGActionManager+Localization.h
Kinetise/Helpers/ActionManager/JavaScript/AGActionManager+External.h
Kinetise/Helpers/ActionManager/JavaScript/AGActionManager+Logic.h
Kinetise/Helpers/ActionManager/JavaScript/AGActionManager+Text.h

Example

```
#import "AGActionManager+Navigation.h"

…

[[AGActionManager sharedInstance] goToScreen:nil :nil :@"screen1"];
```