Solvvy Android SDK Documentation (V1.0.3)

Getting Started

In your project `root` folder locate `build.gradle` add below lines.

```
allprojects {
repositories {
    maven{
          url "https://solvvy.mycloudrepo.io/public/repositories/mobile-sdk-android"
     }
  }
}
```

```
  implementation "com.solvvy.sdk:solvvy:1.0.3"
```

Add below permissions to AndroidManifest. If you already have those permissions in the manifest, you can ignore those. `READ_EXTERNAL_STORAGE` and `CALL_PHONE` are not mandatory permissions.

```
<!-- Permiss required to make api calls -->
<uses-permission android:name="android.permission.INTERNET" />
<!--Required if you need Attachment option -->
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<!-- Required if you need phone support option. -->
<uses-permission android:name="android.permission.CALL_PHONE" />
<!-- permission required to check internet connectivity -->
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

Add `SolvvyUiActivity` to the AndroidManifest file.

```
<activity
android:theme="@style/SolvvySdkTheme"
android:name="com.solvvy.sdk.presentation.ui.activity.SolvvyUiActivity"/>
```

Api Key, Connector Id, Org Id

To integrate the Solvvy Mobile SDK you will need an Api Key, Connector id and Org id. These can be obtained from the Solvvy Dashboard which can be found at https://dashboard.solvvy.com/. You will have to login. After login, go to `Profile Settings` which can be found at the right top under your username. You should find all this information under `Profile Settings`.

If you have multiple orgs associated with you (typically when you have multiple Personas), the same information needs to be obtained for each org.

Keep this information handy. You will using it during the integration.

The simplest integration

Create an instance of `Persona` and set your api key, org id and connector id. Create an instance of `SolvvySDK`, pass in the `Persona` object to it like below and call `startSolvvy` with `Activity context`.

That's it! You are ready to test `Solvvy` SDK integration in your app.

```
SolvvySdk solvvySdkInstance = SolvvySdk.getInstance();
```

```
Persona.Builder persona = new Persona.Builder();
persona.apiKey("YOUR API KEY")
.connectorIdForTicketCreation("CRM CONNECTOR ID")
.orgId("YOUR ORG ID");

solvvySdkInstance.init(persona.build());
solvvySdkInstance.startSolvvy(context);
```

Advanced Scenarios

Color/Font Customization

All the colors used by the Solvvy SDK are documented in `solvvy theme doc.pdf` which ships along with this document. You can set any of the colors used by theme to a value of your choice. You should have a `colors.xml` file in the values sub folder of the res folder. You can set the modified values of colors here.

For example to modify the `tab_layout_background_color` which is used in the `Toolbar`, include this line in the colors.xml file

```
<color name="tab_layout_background_color">#454242</color>
```

Multiple organizations

If you have multiple organizations, you have to create different personas. In the following example there are `Student` and `Instructor` personas.

```
List<Persona> personas = new ArrayList<>(2);

Persona.Builder studentPersona = new Persona.Builder();
studentPersona.apiKey("API KEY FOR THIS ORG")
.connectorIdForTicketCreation("CRM CONNECTOR ID FOR THIS ORG")
.orgId("ORG ID FOR THIS PERSONA")
.buttonText("Student");

Persona.Builder instructorPersona = new Persona.Builder();
instructorPersona.apiKey(
"API KEY FOR THIS ORG")
.connectorIdForTicketCreation("CRM CONNECTOR ID FOR THIS ORG")
.orgId("ORG ID FOR THIS PERSONA")
.buttonText("Instructor");

personas.add(studentPersona.build());
personas.add(instructorPersona.build());
```

Pass the personas collections to the solvvySdk. Rest of the code remains the same i.e.

```
solvvySdkInstance.init(personas);
solvvySdkInstance.startSolvvy(context);
```

Collect more information using Forms

The SDK allows for additional information to be collected from the user along with their question. This information is collected in a "form" shown to the user. What forms need to be displayed and what fields are in the forms should

already have been configured for you by Solvvy in the backend. By default such information is collected in the Review tab along with the customer question.

But you may also control when such additional information is collected.

Pre Question

It maybe collected in the beginning (in the "Ask" tab) before the user has even entered her question. This is called "pre-question". To do so

```
FormSettings.Builder formSettingsBuilder = new FormSettings.Builder();

PreQuestionForm preQuestionForm = new PreQuestionForm();
preQuestionForm.setShow(true);

formSettingsBuilder.preQuestionForm(preQuestionForm);

solvvySdkInstance.setFormSettings(formSettingsBuilder.build());

solvvySdkInstance.startSolvvy(context);
```

Pre Contact

Alternatively, additional information maybe collected in the "Review" tab but before contact options are presented. This is called "pre-contact".

Multiple Ticket Forms

You may have multiple different ticket forms to collect additional information from a user. Different strategies maybe followed to select which form should be shown to the user.

Default Form

If one of the forms is marked as default it will be shown.

Custom Ticket Form

Alternatively, in the mobile SDK you can control which form is shown to the user. Obtain the `form Id` for the corresponding form from your CRM and set it using the following code.

```
FormSettings.Builder formSettingsBuilder = new FormSettings.Builder();

formSettingsBuilder.customTicketFormId(your-ticket-form-id);
```

If a `customTicketFormId` is set it takes precedence. Even if another form is marked as default, the form corresponding to `customTicketFormId` is shown.

User Selects Form

You can also give the choice to the user to select the form. This usually makes sense when the form names correspond to different categories of questions.

```
FormSettings.Builder formSettingsBuilder = new FormSettings.Builder();

formSettingsBuilder.userSelectsForm(true);
```

Controlling contact options shown

You may want to control what contact options are shown based on information collected from the user or any other context available to the integration code. For that override `SolvvySdk.getSupportOption()` in `SolvvySdk.SolvvySdkCallBack`.

```
SolvvySdk.SolvvySdkCallBack solvvySdkCallBack = new SolvvySdk.SolvvySdkCallBack() {
@Override
public List<SupportOption> getSupportOption(Map<String, Object> solvvyState) {
List<SupportOption> supportOptions = new ArrayList<>(4);
supportOptions.add(new ChatSupportOption());
supportOptions.add(new EmailSupportOption());
CommunitySupportOption communitySupportOption = new CommunitySupportOption();
communitySupportOption.setCommunityLink("https://community.upwork.com/");
supportOptions.add(communitySupportOption);
PhoneSupportOption phoneSupportOption = new PhoneSupportOption();
phoneSupportOption.setPhoneNo("ACTUAL PHONE NUMBER TO CALL");
supportOptions.add(phoneSupportOption);
return supportOptions;
}
};
solvvySdkInstance.setSolvvySdkCallback(solvvySdkCallBack);
```

Four kinds of contact options are currently supported by the SDK. They are "Submit a Ticket" (`EmailSupportOption`), "Chat" (`ChatSupportOption`), "Redirect to a web page" (`CommunitySupportOption`) and "Phone call" (`phoneSupportOption`). The logic in `getSupportOption` may return one or more of the contact options. It can also configure the options. For example, the phone number for premium customers maybe different from the others.

Hide the properties from the UI.

Pass the collection of propertyId to `SolvvySdk.FormSettings.hidePropertyList`, those ID's will be ignored in the UI and passed in the create ticket call.

```
SolvvySdk.FormSettings.Builder commonOptionBuilder =
                      new SolvvySdk.FormSettings.Builder();

     List<String> hideList = new ArrayList<>(2);
               hideList.add("custom_303132");
               hideList.add("custom_23028966");

               commonOptionBuilder..hidePropertyList(hideList)
               solvvySdkInstance.setFormSettings(commonOptionBuilder.build());
```

Integrating chat

One of the contact options is `Chat` , which allows the app to direct the user to chat with an agent. Solvvy Mobile SDK has an open architecture. Any chat SDK can be invoked from the chat contact option.

To show a support option to chat, `ChatSupportOption` should have been returned by `getSupportOptions`.

Next, in `SolvvySdkCallBack` override the `handleChatOption()` method. Initialize the respective chat SDK and start it.

For example to integrate Kustomer Chat the following code snippet maybe used

```
@Override
public void handleChatOption(SupportOption supportOption, FragmentActivity context,
Map<String, Object> solvvyState) {
Kustomer.init(getApplicationContext(), "Kustomer API Key");
Intent intent = new Intent(getApplicationContext(), KUSSessionsActivity.class);
startActivity(intent);
}
```

To integrate another SDK such as Zendesk chat or Intercom or any other chat provider, similar code will have to be written inside the `handleChatOption` callback.

Custom Contact Options

Instead of the contact options discussed so far, you may want to show a different contact option such as a custom UI. This can be achieved by extending the `SupportOption` and override `getType()` method, then return `ContactType.CUSTOM`

```
public class CustomSupportOption extends SupportOption {

    @Override
    public ContactType getType() {
        return ContactType.CUSTOM;
    }
}
```

First, return the `CustomSupportOption` in the list of options returned by the `getSupportOptions` method. You can control the icon shown on the custom contact option using the `titleImageResource` field.

Second, implement the `handleCustomOption` method. Similar to `handleChatOption` you can add any code here to bring up whatever UI you would want to show the user.

if `titleImageResource` is not set in `CustomSupportOption`, Sdk uses `R.drawable.ic_custom_support` as the default place holder.

```
SolvvySdk.SolvvySdkCallBack solvvySdkCallBack = new SolvvySdk.SolvvySdkCallBack() {
@Override
public List<SupportOption> getSupportOption(Map<String, Object> solvvyState) {
List<SupportOption> supportOptions = new ArrayList<>(4);
supportOptions.add(new ChatSupportOption());
//Initialize the custom support
CustomSupportOption customSupportOption = new CustomSupportOption();
customSupportOption.setTitle("Custom");
customSupportOption.setDescription("Custom description");
customSupportOption.setButtonDescription("Custom button text");
customSupportOption.setTitleImageResource(R.drawable.ic_custom_support);
supportOptions.add(customSupportOption);
return supportOptions;
}
...
@Override
public void handleCustomOption(SupportOption supportOption, FragmentActivity context,
Map<String, Object> solvvyState) {
```

```
// Bring up whatever UI you want to present to the user
}

};
solvvySdkInstance.setSolvvySdkCallback(solvvySdkCallBack);
```

Skipping Solvvy dialog for some questions

`SolvvySdk.SolvvySdkCallBack` also allows for the "Review" tab to be completely skipped (i.e. Solvvy will not try to provide any answers) and go directly to the "Complete" tab. For example, in a form field the user has indicated his question is about refunds and you know from your analytics that such questions are hard to answer automatically. In this case implement the `showQuestionSearch()` method and return `false`.

```
SolvvySdk.SolvvySdkCallBack solvvySdkCallBack = new SolvvySdk.SolvvySdkCallBack() {
@Override
public boolean showQuestionSearch(final Map<String, Object> solvvyStates) {
return false;
}
};
solvvySdkInstance.setSolvvySdkCallback(solvvySdkCallBack);
```

Solvvy Magic text

If `__solvvy_magic_text_ignore_ticket` is included somewhere in the ticket description, the ticket will not be submitted to your CRM. The Solvvy backend will simply drop the ticket. This is useful while testing.

Handle Analytics Environment

Default all the events fired from `Solvvy` tagged with `prod`. If you wish to enable the `dev` environment for testing, pass `true` in the `setEnvironment(boolean isDevEnvironment)` method.

```
 solvvySdkInstance.setEnvironment(true);
```

Reference

`SolvvySdk` class has `persona`, collections of `personas`, `formSettings` and `solvvySdkCallback`

- `persona` represent single configuration.
- `personas` is a collection of `persona`
- If there are multiple users in the client application, `personas` contain multiple `persona` items. Each persona item has its own organization ID (`orgId`), Api Key (`apikey`), connectorId (`connectorIdForTicketCreation`) and user title (`buttonText`).

```

class SolvvySdk {
private Persona persona;
private List<Persona> personas;
private FormSettings formSettings;
private SolvvySdkCallBack solvvySdkCallback;
}
```

```
class Persona {
private String connectorIdForTicketCreation;
private String apiKey;
```

```
private String buttonText;
private String orgId;
}
```

- `FormSettings` class contains following items.

```
public static class FormSettings {
private PreQuestionForm preQuestionForm;
private PreContactForm preContactForm;
private boolean userSelectsForm = true;
private boolean allowAttachments = true;
private boolean requireCaptcha = true;
private boolean showQuestionSearch;
private Map<String, Object> solvvyState;
}
```

1. `preQuestionForm` is a class of type `PreQuestionForm`. It contains `instructionString` of type `String` (header text), `show` boolean variable (indicates whether to show preQuestionForm or not), `fieldIdWhitelist` list of `String` which accepts the whitelisted fieldIds. Any required properties which are not in the `fieldIdWhiteList` will not be shown in the `preQuestionForm`.

```
public class PreQuestionForm {
private boolean isShow;
private List<String> fieldIdWhitelist;
private String instructionString;
}
```

2. `preContactForm` is a class of type `PreContactForm`. It contains `instructionString` of type `String` (header text), `show` boolean variable (indicates whether to show preContactForm or not), `fieldIdWhitelist` list of `String` which accepts the whitelisted fieldIds. Any required properties which are not in the `fieldIdWhiteList` will not be shown in the `preContactForm`.

```
public class PreContactForm {
private boolean isShow;
private List<String> fieldIdWhitelist;
private String instructionString;
}
```

3. `userSelectsForm` is a boolean. If set `true`, then the default form won't be pre-populated and user has to select the form from the dropdown, default value is set to `true`
4. `allowAttachments` is a boolean which indicates whether to support attachments in the ticket creation, default value is set to `true`.
5. `requireCaptcha` a boolean which indicates whether the app requires captcha support, default value is set to `true`.
6. `solvvyState` is a Map of type `Map<String,Object>` which may contain the default value for the properties like `email`. Key will be the propertyId and value will be respective values for the propertyId which can be any object. This will be pre-filled in the UI.
7. `hidePropertyList` is collection that takes list of properties, when it is set, these properties will not show up in the UI, but will be send to create ticket api call.

Configuring captcha

Set `requireCaptcha` to true in `FormSettings`. In addition reach out to the `Solvvy` team since certain settings need to be enabled on the backend.

Example for `formSettings` configuration with fieldIdWhiteList

```
FormSettings.Builder formSettings = new FormSettings.Builder();

String[] preContactFieldWhiteList = {
"custom_44456188",
"custom_44519307",
"custom_44523707",
"custom_44525407",
"custom_44457808",
"custom_44459308",
"custom_44460968",
"custom_44463528",
"custom_44464668",
"custom_44534007"
};
PreContactForm preContactForm =new PreContactForm();
preContactForm.setShow(true);
preContactForm.setFieldIdWhitelist(Arrays.asList(preContactFieldWhiteList));

Map<String, Object> solvvyState = new HashMap<>();
solvvyState.put("email", "test@gmail.com");

formSettings.preContactForm(preContactForm)
.solvvyState(solvvyState)
.allowAttachments(true)
.showQuestionSearch(true)
.requireCaptcha(true)
.userSelectsForm(true);
```

`SolvvySdkCallBack` options

```
SolvvySdkCallBack solvvySdkCallBack = new SolvvySdkCallBack() {
@Override
public List<SupportOption> getSupportOption(Map<String, Object> solvvyState) {
List<SupportOption> supportOptions = new ArrayList<>();
supportOptions.add(new ChatSupportOption());
supportOptions.add(new EmailSupportOption());
CommunitySupportOption communitySupportOption = new CommunitySupportOption();
communitySupportOption.setCommunityLink("https://community.upwork.com/");
supportOptions.add(communitySupportOption);
PhoneSupportOption phoneSupportOption = new PhoneSupportOption();
phoneSupportOption.setPhoneNo("+917353980930");
supportOptions.add(phoneSupportOption);
return supportOptions;
}

@Override
public void handleChatOption(final SupportOption supportOption,
final FragmentActivity context, Map<String, Object> solvvyState) {
//Implement
}
```

```
@Override
public void handleCallOption(SupportOption supportOption, FragmentActivity context,
Map<String, Object> solvvyState) {
super.handleCallOption(supportOption, context);
// discard the default implementation by not calling super.handleCallOption
// (supportOption, context);
}

@Override
public void handleCommunityOption(SupportOption supportOption, FragmentActivity
context, Map<String, Object> solvvyState) {
super.handleCommunityOption(supportOption, context);
// discard the default implementation by not calling super.handleCommunityOption
// (supportOption, context);
}

@Override
public boolean showQuestionSearch(final Map<String, Object> solvvyStates) {
return true;
}
};
solvvySdkInstance.setSolvvySdkCallback(solvvySdkCallBack);
```

If you have only one support option which in not the instance of `PhoneSupportOption` then `solvvySdk` will auto fire the support options callbacks.

If it is `PhoneSupportOption` user has to manually click the support option to initiate the call.

Calling from React-Native

Create Java module

`ActivityStarter` is just a Java class that implements a React Native Java interface called NativeModule. The heavy lifting of this interface is already done by BaseJavaModule, so one normally extends either that one or ReactContextBaseJavaModule

```
class ActivityStarterModule extends ReactContextBaseJavaModule {

    ActivityStarterModule(ReactApplicationContext reactContext) {
        super(reactContext);
    }

    @Override
    public String getName() {
        return "ActivityStarter";
    }

    @ReactMethod
    void startSolvvy() {
          ReactApplicationContext context = getReactApplicationContext();
          SolvvySdk solvvySdkInstance = SolvvySdk.getInstance();
          SolvvySdk.Persona.Builder persona = new SolvvySdk.Persona.Builder();
          persona.apiKey("YOUR API KEY")
                  .connectorIdForTicketCreation("CRM CONNECTOR ID")
                  .orgId("YOUR ORG ID");
          solvvySdkInstance.init(persona.build());
          solvvySdkInstance.startSolvvy(context.getCurrentActivity());
    }
```

```
}
```

The name of this class doesn't matter; the ActivityStarter module name exposed to JavaScript comes from the getName() method.

Each method annotated with a `@ReactMethod` attribute is accessible from JavaScript.

The default app generated by react-native init contains a MainApplication class that initializes React Native. Among other things it extends ReactNativeHost to override its getPackages method:

```
@Override
protected List<ReactPackage> getPackages() {
    return Arrays.<ReactPackage>asList(
            new MainReactPackage()
    );
}
```

This is the point where we hook our Java code to the React Native machinery. Create a class that implements ReactPackage and override createNativeModules:

```
class ActivityStarterReactPackage implements ReactPackage {

    @Override
    public List<NativeModule> createNativeModules(ReactApplicationContext
reactContext) {
        List<NativeModule> modules = new ArrayList<>();
        modules.add(new ActivityStarterModule(reactContext));
        return modules;
    }


    @Override
    public List<ViewManager> createViewManagers(ReactApplicationContext reactContext)
{
        return Collections.emptyList();
    }
}
```

Finally, update MainApplication to include our new package:

```
public class MainApplication extends Application implements ReactApplication {

    private final ReactNativeHost mReactNativeHost = new ReactNativeHost(this) {
        @Override
        public boolean getUseDeveloperSupport() {
            return BuildConfig.DEBUG;
        }

        @Override
        protected List<ReactPackage> getPackages() {
            return Arrays.<ReactPackage>asList(
                    new ActivityStarterReactPackage(), // This is it!
                    new MainReactPackage()
```

```
            );
        }
    };

    @Override
    public ReactNativeHost getReactNativeHost() {
        return mReactNativeHost;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        SoLoader.init(this, false);
    }
}
```

React Native side all you need to do is import `NativeModules` and call the java method like below

```
NativeModules.ActivityStarter.startSolvvy()
```

Sample React-Native implementation.

```
/**
 * Sample React Native App
 * https://github.com/facebook/react-native
 *
 * @format
 * @flow
 */

import React, {Component} from 'react';
import {Platform, StyleSheet, Text, View,Button,NativeModules} from 'react-native';


const instructions = Platform.select({
  ios: 'Press Cmd+R to reload,\n' + 'Cmd+D or shake for dev menu',
  android:
    'Double tap R on your keyboard to reload,\n' +
    'Shake or press menu button for dev menu',
});

type Props = {};
export default class App extends Component<Props> {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>Welcome to React Native!</Text>
        <Text style={styles.instructions}>To get started, edit App.js</Text>
        <Text style={styles.instructions}>{instructions}</Text>
        <Button
            onPress={() => NativeModules.ActivityStarter.startSolvvy()}
            title='Start example activity'
          />
      </View>
    );
  }
}
```

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  instructions: {
    textAlign: 'center',
    color: '#333333',
    marginBottom: 5,
  },
});
```

Debug section

- If the forms are not populating with content, please double check your `apiKey` / `orgId`.
- All the configuration and `SolvvySdkCallBack` need to be set before you call `solvvySdk.startSolvvy(context)`, if you won't follow this, SDK can behave strangely.
- Solvvy SDK throws `SolvvySdkException` when supplied configuration is invalid.

Note:

All the items in `formSettings` are optional.