YOUR VOICE HAS NO LIMIT

Getting Started with VSDK (Android Edition)

Getting Started with VSDK (Android Edition)

The Vivoka SDK is meant to greatly facilitate the creation of voice-enabled applications, regardless of who's providing the underlying technologies!

Installation

Simply import the .aar file into your project with Android Studio and add a gradle dependency to use it.
For example, for the `vsdk-csdk` library:

```
implementation project(path: 'vsdk')
implementation project(path: ':vsdk-csdk')
```

Configuration

All VSDK engines are to be initialized with a JSON configuration file. This file contains a version, then each engine will declare its own configuration block:

```
{
  "version": "2.0",
  "csdk": {
    ...
  },
  ...
}
```

We **strongly** recommend you put this file under a `config` directory because most engines will generate additional configuration files or use a cache (configurable).

Library versions

You can access the version of VSDK like so:

```
String vsdk_version = Vsdk.getVersion();
```

Error Handling

The SDK will throw exceptions on each function call. A custom exception class has been created to help track the origin of the error.

```
try {
  Vsdk.init(context, "configPath", callback);
} catch (com.vivoka.vsdk.Exception e) {
  // print the stacktrace (recommanded)
  e.printFormattedMessage();

  // get the stacktrace in a string
  String error = e.getFormattedMessage();

  // as com.vivoka.vsdk.Exception inherits from the java.lang.Exception class, you can also do this
  e.printStackTrace();
}
```

Audio Management

VSDK being a SDK around voice technologies, audio is a central component. Starting from version 4, **audio pipelining** has been added for greater power and simplicity.

Pipeline

An audio pipeline is composed of 3 types of audio modules: one **Producer**, zero or more **Modifiers** and zero or more **Consumers**. Simply put: a producer sends audio (either synchronously or asynchronously) into modifiers (if any), and the resulting audio is finally given to consumers (if any).

```
// Create a pipeline
Pipeline pipeline = new Pipeline();

// Create an audio recorder which inherit from the ProducerModule
AudioRecorder audioRecorder = new AudioRecorder();
try {
  pipeline.setProducer(audioRecorder);
} catch (Exception e) {
  e.printFormattedMessage();
}

// Create a recognizer which is inherited from the ConsumerModule
Recognizer recognizer;
try {
  pipeline.pushBackConsumer(recognizer);
} catch (Exception e) {
  e.printFormattedMessage();
}
```

If you would like to create your own Audio Module, inherit from `ProducerModule`, `ModifierModule` or `ConsumerModule`.

ASR

VDK 3 features one ASR library: **CSDK**.

Basics

You will need to manipulate 2 concepts: Recognizers & Models. Both need to be configured but first let's explain who's who.
Models are fed to the Recognizer and describe the range of words and utterances that can be recognized. They will either be pre-compiled by the provider (like “free speech” models), or compiled from a grammar that you've written beforehand in the VDK Studio.
There are 3 types of models:

Type	Description
static	Static models embed all possible vocabulary inside a single file or folder.
dynamic	Dynamic models have “holes” where you can plug new vocabulary at runtime. These need to be prepared and compiled at runtime before installing it on a recognizer.
free-speech	Free-Speech models are very large vocabulary static models. They often require additional files and are not supported by all engines.

Configuration

Each engine has its own configuration quirks and tweaks, but here is a common (though incomplete) pattern using VSDK-CSDK, which supports all 3 types of models:

```
{
  "version": "2.0",
  "csdk": {
    "paths": {
      "data_root": "../data"
    },
    "asr": {
      "recognizers": {
        "rec": { ... }
      },
      "models": {
        "static_example": {
          "type": "static",
          "file": "<model_name>.fcf"
        },
        "dynamic_example": {
          "type": "dynamic",
          "file": "<base_model_name>.fcf",
          "slots": {
            "firstname": { ... },
            "lastname": { ... }
          },
          ...
        },
        "free-speech_example": {
          "type": "free-speech",
          "file": "<base_model_name>.fcf",
          "extra_models": { ... }
        }
      }
    }
  }
}
```

Starting the engine

```
com.vivoka.vsdk.Vsdk.init(mContext, "config/main.json", vsdkSuccess -> {
  if (vsdkSuccess) {
    com.vivoka.csdk.asr.Engine.getInstance().init(mContext, engineSuccess -> {
      if (engineSuccess) {
        // at this point the AsrEngine has been correctly initialized
      }
    }
  }
}
```

You can't create two separate instances of the same engine! Attempting to create a second one will get you another pointer to the existing engine.

Each engine has its own configuration document, check it out for further details, as well as the ASR samples to get started with actual, production-ready code.

Creating a Recognizer

```
// First you have to create a recognizer listener to subscribe to the recognizer events
IRecognizerListener recognizerListener = new IRecognizerListener() {
  @Override
  public void onEvent(RecognizerEventCode eventCode, int timeMarker, String message) {}

  @Override
  public void onResult(String result, RecognizerResultType resultType, boolean isFinal) {}

  @Override
  public void onError(RecognizerErrorCode error, String message) {}

  @Override
  public void onWarning(RecognizerErrorCode error, String message) {}
};

// Then you can create a recognizer
recognizer = Engine.getInstance().makeRecognizer("rec", recognizerListener);
```

And finally, apply a model to actually recognize vocabulary:

```
recognizer.setModel("static_example"); // same call whether the model is static, dynamic or free-speech!
```

Also, don't forget to insert it in the pipeline or nothing's going to happen by itself:

```
pipeline.pushBackConsumer(recognizer);
```

Dynamic Models

Only dynamic models need to be manipulated explicitely to add the missing data at runtime:

```
DynamicModel model = com.vivoka.csdk.asr.Engine.getInstance().prepareDynamicModel("dynamic_example");
model.addData("firstname", "André");
model.addData("lastname", "Lemoine");
model.compile();
// We can now apply it to a recognizer!
recognizer.setModel("dynamic_example");
```

TTS

VDK 3 features two TTS libraries: **CSDK** and **Baratinoo**.

Configuration

TTS engines must be configured before the program starts. Here is a complete setup with 2 channels, each one using a different language (using the CSDK engine):

```
{
  "version": "2.0",
  "csdk": {
    "tts": {
      "channels": {
        "channelFrf": {"voices": ["frf,aurelie,embedded-compact", "enu,ava,embedded-compact"]},
        "channelEnu": {"voices": ["enu,ava,embedded-compact"]}
      }
    }
  }
}
```

An empty channel list will trigger an error, as well as an empty voice list!

Voice format

Each engine has its own voice format, described in the following table:

Engine	Format	Example
vsdk-csdk	`<language>,<name>,<quality>`	`enu,evan,embedded-pro`
vsdk-baratinoo	`<name>`	`Arnaud_neutre`

Starting the engine

```
com.vivoka.vsdk.Vsdk.init(mContext, "config/main.json", vsdkSuccess -> {
  if (vsdkSuccess) {
    com.vivoka.csdk.tts.Engine.getInstance().init(mContext, engineSuccess -> {
      if (engineSuccess) {
        // at this point the TtsEngine has been correctly initialized
      }
    }
  }
}
```

Creating a channel

Remember, channel must be configured beforehand!

```
Channel channelFrf = com.vivoka.csdk.tts.Engine.getInstance().makeChannel("channelFrf",
"frf,aurelie,embedded-compact");
```

Speech Synthesis

Speech Synthesis is **asynchronous**! That means the call will not block the thread during the synthesis.

```
channelFrf.synthesisFromText("Bonjour ! Je suis une voix synthétique", () -> {
  // channelFrf.synthesisResult contains the audioData to play
}

// Also works with SSML input
final String ssml = "<speak version=\"1.0\" xmlns=\"http://www.w3.org/2001/10/synthesis\"
xml:lang=\"fr-FR\">Bonjour Vivoka</speak>";
channelFrf.synthesisFromSSML(ssml, () -> {
  // channelFrf.synthesisResult contains the audioData to play
});
```

Playing the result

VSDK provides an audio player. Playing the result is very easy:

```
AudioPlayer.play(channel.synthesisResult.getAudioData(), channel.synthesisResult.getSampleRate(), new
AudioTrack.OnPlaybackPositionUpdateListener() {
  @Override
  public void onMarkerReached(AudioTrack track) {}

  @Override
  public void onPeriodicNotification(AudioTrack track) { }
});
```

The audio data is a 16bit signed Little-Endian PCM buffer. Channel count is always 1 and sample rate varies depending on the engine:

Engine	Sample Rate (kHz)
csdk	`22050`
baratinoo	`24000`

Storing the result on disk

```
channel.synthesisResult.saveToFile("directory", "filename", new ICreateAudioFileListener(){});
```

Only PCM extension is available, which means the file has no audio header of any sort.

Voice Biometrics

VDK 3 features two Voice Biometrics libraries: **TSSV** and **IDVoice**.

Configuration

Voice biometrics engines must be configured before the program starts. Here is a complete setup for the TSSV provider:

```
{
    "version": "2.0",
    "tssv": {           // Contrary to other technologies,
        "biometrics": { // biometrics paths are relative to the program's working directory!
            "generated_models_path": "data/models",
            "background_model_TI": "data/text-independent-16kHz.ubm",
            "background_model_TD": "data/text-dependent-16kHz.ubm"
        }
    }
}
```

Starting the engine

```
com.vivoka.vsdk.Vsdk.init(mContext, "config/main.json", vsdkSuccess -> {
  if (vsdkSuccess) {
    com.vivoka.tssv.Engine.getInstance().init(mContext, engineSuccess -> {
      if (engineSuccess) {
        // at this point the BiometricEngine has been correctly initialized
      }
    }
  }
}
```

Creating a model

Models contain enrollment data that recognition operations need.

```
// To create a text independant model
Model model = com.vivoka.tssv.Engine.getInstance().makeModel("test_ti", ModelType.TEXT_INDEPENDANT);

// To create a text dependant model
Model model = com.vivoka.tssv.Engine.getInstance().makeModel("test_ti", ModelType.TEXT_DEPENDANT);
```

Checking users in the model

If a model was previously created with the same name it will be loaded. You can check enrolled users with:

```
if (model.getUsers().size() == 0) {
  // No enrolled users
}
```

Adding a user to the model

You can either add raw audio data directly. After adding all the data for a given user, compile the model to finalize the enrollment process:

```
//model.addRecord(String name, Buffer buffer);
model.addRecord("victorien", buffer);
```

The more data you give the model, the better the result will be.
Prefer to register the data in the condition of the use case of the model.

The format is preferred to be 16Khz mono-channel.

Performing authentication or identication

Both are covered in the same chapter as it is very similar:

```
// Identificator
Identificator identificator = com.vivoka.tssv.Engine.getInstance().makeIdentificator("ident", model,
new IStatusReporter<IdentificatorResultType, IdentificatorEventCode, IdentificatorErrorCode>() {
    @Override
        public void dispatchResult(IdentificatorResultType authenticatorResultType, String s,
JSONObject jsonObject, boolean b) {
        String user = jsonObject.optString("id");
    }

    @Override
        public void dispatchEvent(IdentificatorEventCode identificatorEventCode, String s, String s1,
int i) {}

        @Override
    public void dispatchError(ErrorType errorType, IdentificatorErrorCode identificatorErrorCode,
String s, String s1) {}
});

// Authenticator
Authenticator authenticator = com.vivoka.tssv.Engine.getInstance().makeAuthenticator("ident", model,
new IStatusReporter<AuthenticatorResultType, AuthenticatorEventCode, AuthenticatorErrorCode>() {
    @Override
        public void dispatchResult(AuthenticatorResultType authenticatorResultType, String s,
JSONObject jsonObject, boolean b) {
        String user = jsonObject.optString("id");
    }

    @Override
        public void dispatchEvent(AuthenticatorEventCode authenticatorEventCode, String s, String s1,
int i) {}

        @Override
        public void dispatchError(ErrorType errorType, AuthenticatorErrorCode authenticatorErrorCode,
String s, String s1) {}
});
authenticator.setUserToRecognize("victorien");
```

The only difference is that the authentication can only recognize user “victorien” and the identification can recognize every user enrolled in the model.
Different providers will give you different results, for example IDVoice reports varying results as it analyzes the audio, while TSSV only sends you result if the engine thinks it is acceptable (depending of the confidence level you set).
We recommend that you try it out the application in real situation to select your custom minimum score required to satisfy your need in false rejection and false acceptation. But by default you can just check if the score is above 0.