Google Summer of Code

GSoC 2020 Proposal for

Visible Component Extensions

Proposed by Pavitra Golchha (@pavi2410)

Under the category: prototypes and proofs of concept

Organisation: MIT App Inventor 

Table of Contents

Interest in App Inventor

I am interested in App Inventor because it allowed me to create an Android app without any prior knowledge of Computer Science. It helped me learn the fundamentals of computer programming. I was amazed how I can develop apps easily that I can run on my phone!

Interest in introductory programming

Though I have taught little basics of App Inventor to my siblings and friends, I have no other significant experience in teaching introductory programming in App Inventor. If not teaching, then I have helped many people through their problems faced with using App Inventor.

Proposed summer project

Visible Component Extensions[1]

Brief explanation: Work has been done during the last few GSoC sessions in developing a Component Developer's Kit to App Inventor. This facility provides for loading externally developed components into App Inventor. However this work is currently limited in scope. In particular only components that have no visible UI elements can be added. We would like to extend the CDK capabilities to include extensions with UI elements. Extensions are currently only available in English. We would like an enhancement to that extension writers can provide strings in other languages as well, as is the case with builtin components.

Expected results: The ability to build and add extensions with text in non-English languages as well as access to UI elements.

Knowledge Prerequisite: Java and familiarity with GWT and the Android SDK. Familiarity with systems integration, release management, and systems architecture is a plus.

Difficulty and estimated time: Hard; this would be a whole summer project.

I chose this project because it was something many extension developers have requested in the past [2] [3] and was one key missing feature of the CDK. As a result of

[1] https://github.com/mit-cml/appinventor-sources/wiki/Google-Summer-of-Code-2020#visible-component-extensions

[2] https://groups.google.com/d/msg/app-inventor-open-source-dev/9Uzm2d7x54A/JKqlbxTxEAAJ

[3] https://groups.google.com/d/msg/app-inventor-open-source-dev/mjkaDdrTwEE/ktkhjtFtAgAJ

that, extension developers have worked around this by making a function which takes a component (as parent container) which makes using those extensions unintuitive [4] [5].

Abstract + Implementation

Currently, extensions are limited to just non-visible components. Due to this, extensions can't be dragged into the mock form designer. To work around this, extension developers have to get a reference to a visible component as parent under which they create UI views. This, however, defeats the idea of App Inventor, which allows the users to create their UI of their apps using the concept of WYSIWYG.

My proposal to solve this problem is to have a generic `*MockVisibleComponentExtension*` which can be dragged into the designer. The extension will be just a visible component as opposed to non-visible component. This allows the extension to be placed under a `*ComponentContainer*` under which the extension can create its views. This allows the users to view a clear hierarchy of their app's UI when using "visible" extension. Then, there will also be a need for `MockComponentContainerExtension` into which any visible component/extension can be placed.

There will be default mock provided for such extensions which do not have specialized mock (yet to be discussed). **The default mock will only contain the extension icon**. This is good because the mechanism to generate the mock has not been thought upon, and we are yet to discuss the possibility. Extension developers can choose to not provide a specialized mock if their extension is simple enough.

On the Android side, there is no need to do anything. The instantiation of components is well-handled by the runtime.scm#L51 file. Same goes for the buildserver.

---

[4] [Free] ActionBar Extension - #ThunkableClassicExtensions

[5] [Free] CardView Extension - #ThunkableClassicExtensions

For **extension generation** also, there is nothing special to do, other than generating mock.

As part of my research, I've successfully created a branch [6] where I am able to build a visible component extension [7], drop it to the designer, and succeeded in running the aforementioned extension in the companion app as well as a standalone app.

Thoughts on generating mock components

A Mock component is used to show a preview of the component in the designer. This component reacts to the changes in the component's properties and adapts accordingly. This works great for built-in components, because we already have all the information of every built-in component. But this is something of a concern for achieving the goal of this proposal. This was not a problem with non-visible extensions because it's not visible in the designer as the name suggests.

We could create an interface or an abstract class, which will then be extended by the extension developer (optional, as the extension dev may choose not to create the mock component, so we provide a default implementation which just shows the extension's icon). Due to the limitations of GWT, we cannot use Java Reflection nor load JS files dynamically using `eval()` due to security concerns. After learning more about GWT, I found about "Deferred Binding". But, according to the docs, "*Deferred binding is a feature of the GWT compiler that works by generating many versions of code at compile time, only one of which needs to be loaded by a particular client during bootstrapping at runtime*. …", this may not work for us because we want to be able to load many different visible component extensions.

The companion app, on the other hand, can load extension classes dynamically by loading the extension .dex files in the *DexClassLoader*.This works because the environment the companion app runs on is Android, which has a Dalvik/ART VM equivalent of JVM. So, it is guaranteed by the JVM to be able to load classes

[6] https://github.com/pavi2410/appinventor-sources/tree/visible-extensions
[7] https://gist.github.com/pavi2410/211607ae22e263ca8f3c89eb909766fb

dynamically. Whereas, in the GWT side, it is transpiled into JS files and depends on the emulation of the JRE, due to which stuff like reflection and class loading is simply not possible. To tackle such limitations, GWT has introduced “Deferred Binding” and “Generators”, which have their own set of limitations.

However, in an ideal condition, I have thought of a dummy implementation plan:

- There will be an abstract class (say MockVisibleComponentExtension)

  ```
  public class MockVisibleComponentExtension {}
  ```

- Extension devs would then implement the above abstract class to create the Mock for their visible component extension.

  NOTE: The API surface has to be thought upon in order to access the properties set in the properties panel for this extension.

- The implemented class will be transpiled by the GWT compiler to JS files which will be added to the .aix file.
- When a visible component extension is imported into the builder. The builder will register the extension’s Mock component like this:

  MockComponentRegistry.register(String extensionFqcn,
  Class<MockVisibleComponentExtension> mockClass);

- When this extension is dragged and dropped into the MockForm designer, the `mockClass` corresponding to the `extensionFqcn` will be instantiated and added into the parent MockComponent by calling:

  ```
  MockVisibleComponentExtension mvce =
  mockClass.newInstance();

  parentMockConatinerComponent.add(mvce); // hypothetical
  method for demonstration
  ```

Deliverables

The proposal aims to come up with the following:

- Ability to create a visible component extension.
- Ability for extension devs to create a customized mock for their visible component extension.

Experience with the development tools

- **Java**

  I have around 4 years of progressive experience in Java. I have a good understanding of OOP, SOLID principles and clean code architecture. I know Ant and Gradle build tool systems.

  Although I have now switched to over Kotlin, I have a handful of projects and achievements to showcase:

  - https://repl.it/@pavi2410/Markdown-Generator
  - My commits to App Inventor's open source repo, notably PR #1417
  - Passed LinkedIn Assessment for Java

- **JavaScript**

  I have 2.5 years of experience in JavaScript. Until now, I have made many websites and web apps to learn new concepts. I have also worked with Node.js.

  - https://github.com/pavi2410/REPLisp - A programming language
  - https://github.com/pavi2410/Random-Quote-Machine
  - https://github.com/pavi2410/URL-Inspector
  - https://glitch.com/~dsc-survey

- **Android development with the Java SDK**

  I have been learning and developing Android apps for 2 years now. Daily, I learn new things and try to put that into practice.

  - https://github.com/pavi2410/GooglePlayCloneKt
  - https://github.com/pavi2410/VRCompatibilityChecker
  - https://github.com/pavi2410/DSC-ToDo-List

- **Git/Github**

  I am proficient in using GitHub and prefer it over other Git servers. My GitHub profile can be found here.

  I can also use git to stage files, make commits, create and checkout branch, pull/push changes from/to GitHub, all from the command line. I mostly use GitKraken though.

Experience with teams, online developer communities and large code bases

I am active in online developer communities through the platforms like StackOverflow, Reddit, Twitter. On StackOverflow, I have gained over 700+ reputation. On Reddit and Twitter, I mostly follow Android and Web devs, and actively participate in their discussions.

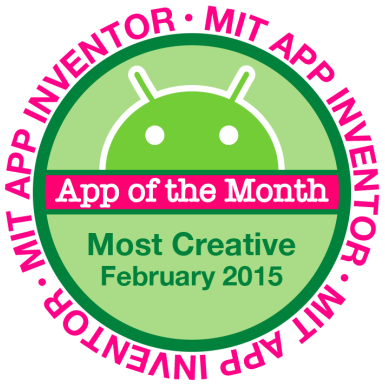
In the App Inventor world, I am part of the App Inventor Open Source Development community, I was a part of the Thunkable community and now I am active in the Kodular community, helping people through their various problems.

Talking about large code bases, I have worked with App Inventor's source code (and Kodular alike). Apart from that, I worked on the Bulma CSS framework.

Application challenges

Challenge 1: Creating a non-trivial app

The app I made is called “Speak Now!”. It is a recreation of my own app, which was featured as the “MIT App Inventor App of the Month - Most Creative” in February, 2015 [8].



> Speak Now! is an app that converts written text to speech. It was created by 14-year-old Pavitra Golcha from India so that people could learn how to correctly pronounce English words.

As instructed to use a local server, unfortunately, I cannot make the Yandex Translate component work [9]. Instead, I used a Web component to get the translations.

AIA: https://drive.google.com/open?id=1WSI1YGhhRltrnT5MYBJ1UBFDGlHDDZEE

APK: https://drive.google.com/open?id=1XGppb4QGkIZKNP0f5IW6NUVYFguiBniG

[8] App of the Month Winners 2015
[9] https://github.com/mit-cml/appinventor-sources/issues/2126

Challenge 2:Design challenge -- Enhancing the Camera operation

Deprecation of UseFront property in the Camera component

The UseFront property was deprecated because it was using an **undocumented, test** Intent Extra flag (*"android.intent.extras.CAMERA_FACING"*) defined in the AOSP Camera app source. Since this flag is undocumented, it is not surprising at all if it doesn't work at some point in time. The UseFront property didn't work because there was no way left to tell the camera app which camera to use through the means of Intent.

Capture images without user intervention

The Camera component uses an Intent (*"android.media.action.IMAGE_CAPTURE"*) to launch the device's camera app. This requires users to manually take a picture. But, in order to take a picture without user intervention, we must display a preview to the user, considering the user's privacy. As the existing Camera component is a *AndroidNonvisibleComponent,* I would create a new visible component named "CameraView" in the appinventor/components/src/com/google/appinventor/components/runtime directory.

The visible component can be placed in any container in the designer and can used to replicate a camera app. The component will add a SurfaceView in the layout in combination with the Camera API (not using Camera2 API as it's only available from API 21 and App inventor should be compatible with API 9 or newer; to provide compatibility with newer devices, we can instead use the CameraX Jetpack library.)

I would then define the following methods/functions:

- TakePicture

This will capture the image and save it to the file specified. Must call StartPreview before calling this method.

- RecordVideo

  The component could also record videos.

- StartPreview

  Starts the preview shown to the user.

- StopPreview

  Stops the preview shown to the user.

- IsPreviewRunning

  To check whether the preview is running.

Also, these properties will be added:

- FilePath(String)

  Used to set the path of the file to save the image/video.

- UseFrontCamera(Boolean)

  Setting this to true will set the preview to use the front camera.

- Rotate(Int)

  The user can specify the angle of the preview in degrees.

- TurnFlash(Boolean)

  This property tells whether to turn on the flash.

- … several camera specific properties.

For the implementation, the official Android docs gives us a clear view of how the integration will work.

> The general steps for creating a custom camera interface for your application are as follows:
>
> - **Detect and Access Camera** - Create code to check for the existence of cameras and request access.
> - **Create a Preview Class** - Create a camera preview class that extends SurfaceView and implements the SurfaceHolder interface. This class previews the live images from the camera.
> - **Build a Preview Layout** - Once you have the camera preview class, create a view layout that incorporates the preview and the user interface controls you want.
> - **Setup Listeners for Capture** - Connect listeners for your interface controls to start image or video capture in response to user actions, such as pressing a button.
> - **Capture and Save Files** - Setup the code for capturing pictures or videos and saving the output.
> - **Release the Camera** - After using the camera, your application must properly release it for use by other applications.

When the component is initialised, we will add a surface view to the container like this:

```
public CameraView(ComponentContainer container) {
 …
 surfaceView = new SurfaceView(this);
 container.$add(surfaceView);
}
```

and to take a picture:

```
@SimpleFunction
public void TakePicture() {
  camera.takePicture(null, null, this /*
Camera.PictureCallback jpeg */);
}
```

The Camera.PictureCallback interface will be implemented by the component class and this method will be overridden:

```
@Override
public void onPictureTaken(byte[] data, Camera camera) {
  // save the image to the file
}
```