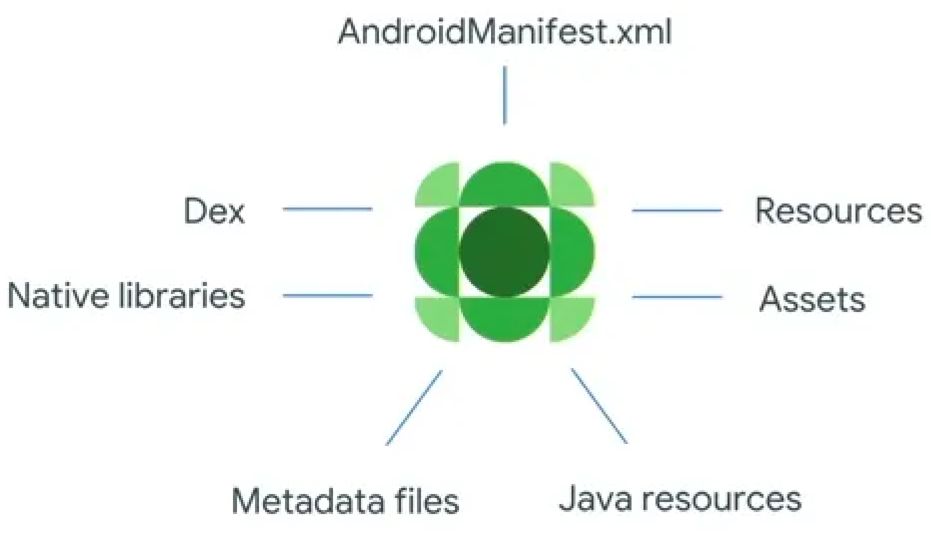
# Android App Bundles

Introducing AAB for MIT App Inventor

## Summary

The aim of this summer project is to bring to MIT App Inventor the option to build and export projects as Android App Bundles (.aab). This new format is an alternative to the Android Application Package (.apk) file format when distributing apps through Google Play Store. AAB files are intended for more professional apps, whose developers try to give their users a more optimized experience by distributing the app through the store. Moreover, by using Android App Bundles, users can enable App Signing by Play Store in their apps.



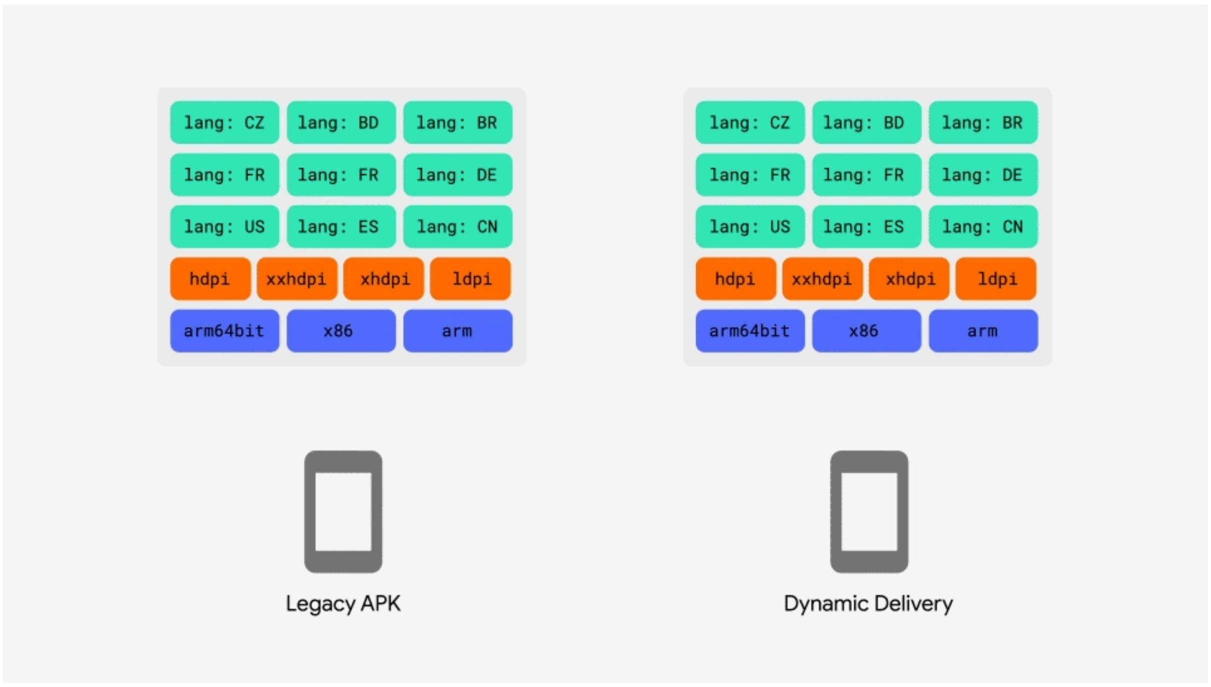
Android App Bundles

## Introduction to Android App Bundles

Android App Bundles are not a real alternative to APK files. They can only be used in Google Play Store, and they cannot be installed directly in the phone. They are a bundle (as stated by the name) which contain several files alternatives for each kind of device. For example, in Windows, you usually have x32 and x64 versions. The same applies to Android: there are specific source files that vary per device and, in consequence, some apps may not work in other devices (something like the Minimum SDK requirement).

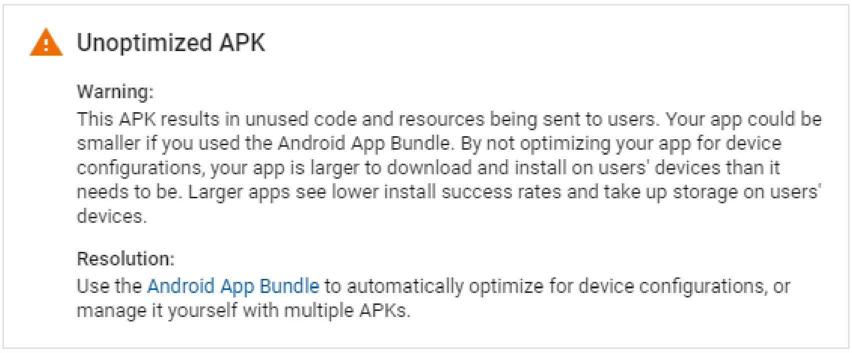
However, it is rare to see different APK files per architecture. This is because it is very common to generate an universal version that can be installed in any device. Thus, this universal version contains all specific runtime files that are needed in all architectures, screen sizes, etc. But this makes the APK to be unoptimized, having to include all files needed for all platforms, rather than the specific ones for the one that is going to be installed. Although tools like Android Studio can already export platform-specific versions, it is quite pesky having to build lots of APK files.

Android App Bundles are a solution to this problem. In contrast to APK files, they are ZIP files that contain all needed files for all platforms. From that bundle, any specific platform version APK can be generated. What Play Store does with that AAB file is generate all possible combinations, and deliver the specific one to the user when they install it through what they call “Dynamic Delivery”. This results in a much more optimized app for the user, saving bandwidth and resources in their phone, as they only get the specific source files for their phone.



How Android App Bundles work in Google Play Store

Also, by using Android App Bundles, App Inventor users can get rid of the annoying and confusing warnings that appear in the Play Store Console. These warnings can easily mislead users to understand that they cannot upload APK files, and they could be somehow forced to use the Android App Bundles.


Unoptimized APK

Warning:
This APK results in unused code and resources being sent to users. Your app could be smaller if you used the Android App Bundle. By not optimizing your app for device configurations, your app is larger to download and install on users' devices than it needs to be. Larger apps see lower install success rates and take up storage on users' devices.

Resolution:
Use the Android App Bundle to automatically optimize for device configurations, or manage it yourself with multiple APKs.

Warning that appears in Google Play Store

## Document Layout

The layout of this document will focus on two main targets that are affected by this change: app developers and core developers. App developers are understood as MIT App Inventor users who create apps with the platform, and core developers are understood as people who contribute and work in the development of the open-source platform itself. Both sections contain some open questions on which any kind of thought and suggestion are very welcome.

- Jump to app developers section
- Jump to core developers section

At the end, there is also a Miscellaneous section on which some points like Documentation and Testing are mentioned.

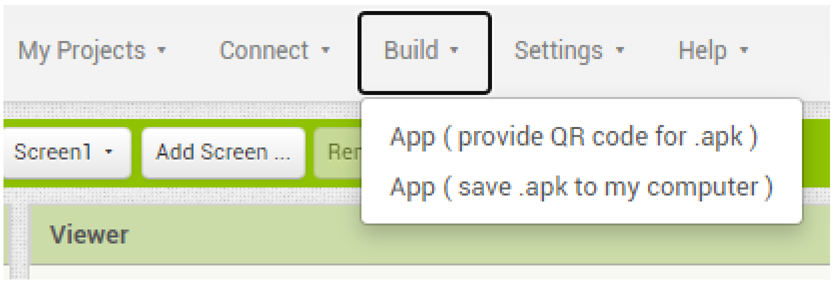
Also, keep in mind this document is a proposal. All ideas are welcome to be taken into account to get into a final one.

## For app developers

App Inventor app developers will not notice a significant change. This is because the work is mainly done in the backend, without affecting the interface. They would, however, get the option to export the apps in a new file format.

### Unifying both export options

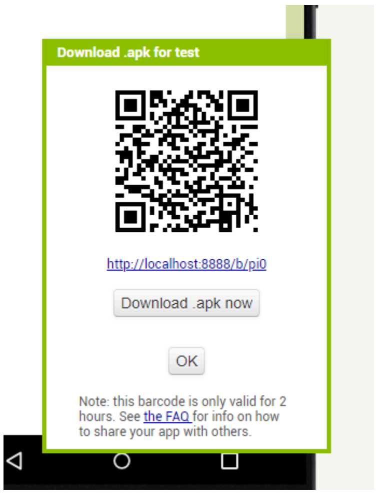
As it is required to add a new export choice in the dropdown, it is highly recommended to unify both export as APK options into a single button, making it simpler.


My Projects
Connect
Build
Settings
Help
Screen1
Add Screen ...
Viewer
App ( provide QR code for .apk )
App ( save .apk to my computer )

Classic “Build” dropdown

Both export options were actually doing the same: building the project as APK and sending it back to App Engine. The only user-perceivable difference between each of them is that the “save .apk to my computer” choice will directly download the APK whereas the “provide QR code for .apk” was providing a nonce link (`/b/abcd`) in the form of a QR code, which expires after 2 hours. This second option is the most similar to the workflow we intend to achieve, so it will prevail over the direct download export to apply changes.

This behaviour could be improved by changing the “QR code” dialog, adding a “Download Now” button, so the user would get both export options into the same build process. This is how the dialog might look like:


Download .apk for test

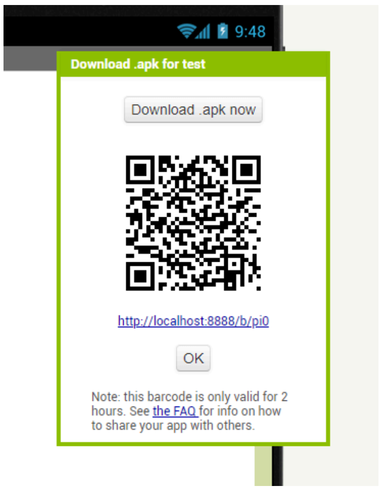
http://localhost:8888/b/pi0

Download .apk now

OK

Note: this barcode is only valid for 2 hours. See the FAQ for info on how to share your app with others.

Added “Download .apk now” button in between the link and OK button


9:48

Download .apk for test

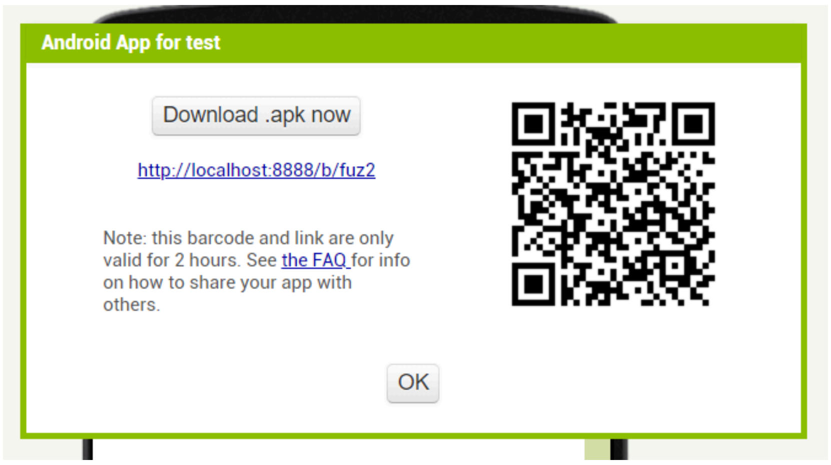
Download .apk now

http://localhost:8888/b/pi0

OK

Note: this barcode is only valid for 2 hours. See the FAQ for info on how to share your app with others.

Added “Download .apk now” button above the QR code


Android App for test

Download .apk now

http://localhost:8888/b/fuz2

Note: this barcode and link are only valid for 2 hours. See the FAQ for info on how to share your app with others.

OK

Refactored dialog using a horizontal layout

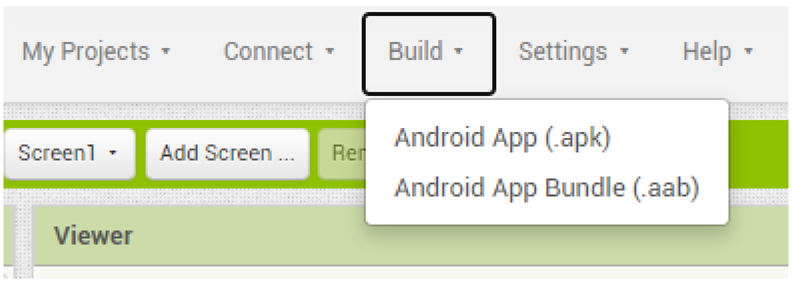
All options seem to be correct. The logic behind each of them is:

- **Between link and OK button:** The “Download .apk now” button actually goes to the link and the link automatically initiates the download. So, it is like the link points to both QR code and “download now” button, making both connected to the link as being next to each other.

- **Above the QR code:** The second choice is to actually place it above all. Before, there was an specific button to request its download, without invoking the dialog. So, by placing it above all it seems to be given more importance against the rest of the data in the dialog (which was there previously).

- **Horizontal layout (thanks to Evan and Beka for the suggestion):** To avoid banner-blindness, this horizontal layout places in one side all click-related elements, and on the right side the camera/interactive ones. The button gains more visibility and importance, and the link is kept for the case of sharing a screenshot and wanting to manually type the URL. The label gets modified to mention that both link and barcode expire.

And finally, the “Build” dropdown. Here is a mockup of how it might look like:



Example of the new “Build” dropdown

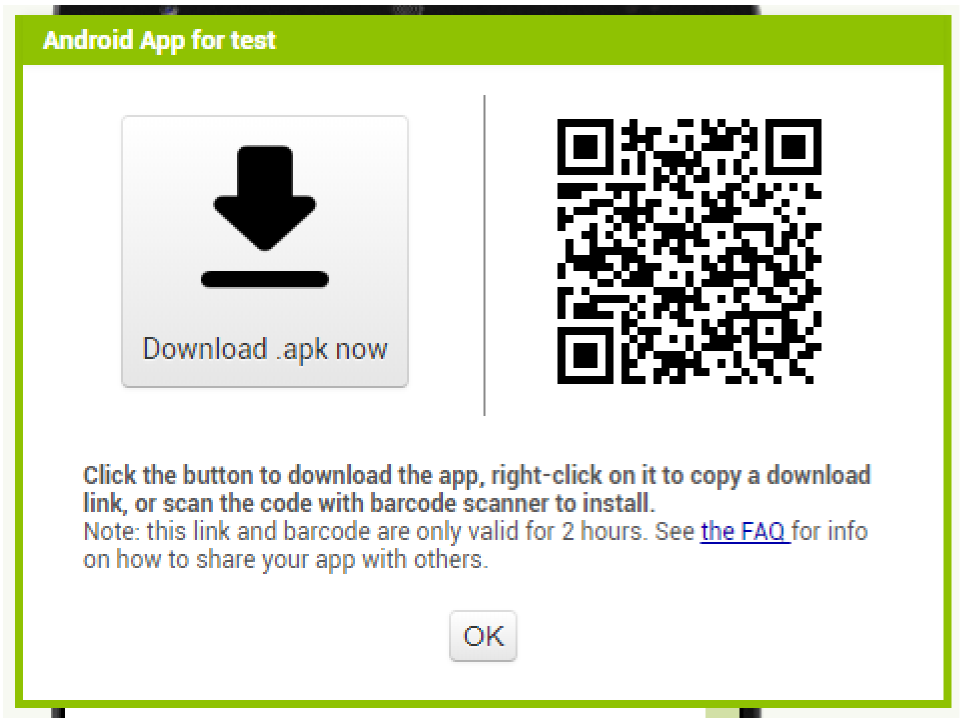
As it can be seen, there is just one “export as .apk” option, that will invoke the dialog that was shown above. Also, the “App” label has been renamed to be a bit more generic: “Android App”. This was motivated by the upcoming ability to export MIT App Inventor apps for iOS, so having “App” everyone might confuse users.

The other noticeable change is the “Android App Bundle” choice. This option will just download the AAB file, rather than displaying a dialog, because AAB files cannot be installed directly into phones, thus does not make sense to have a special way to send them to smartphones directly, they are intended to be uploaded to Play Store. The reason of choosing “Android App Bundle” as label was to avoid confusion for users: by setting “Android App” as the APK export option, it seems like that should be the primary choice, and “Android

App Bundles” will be picked by people who already know what they are and for what they can use it.

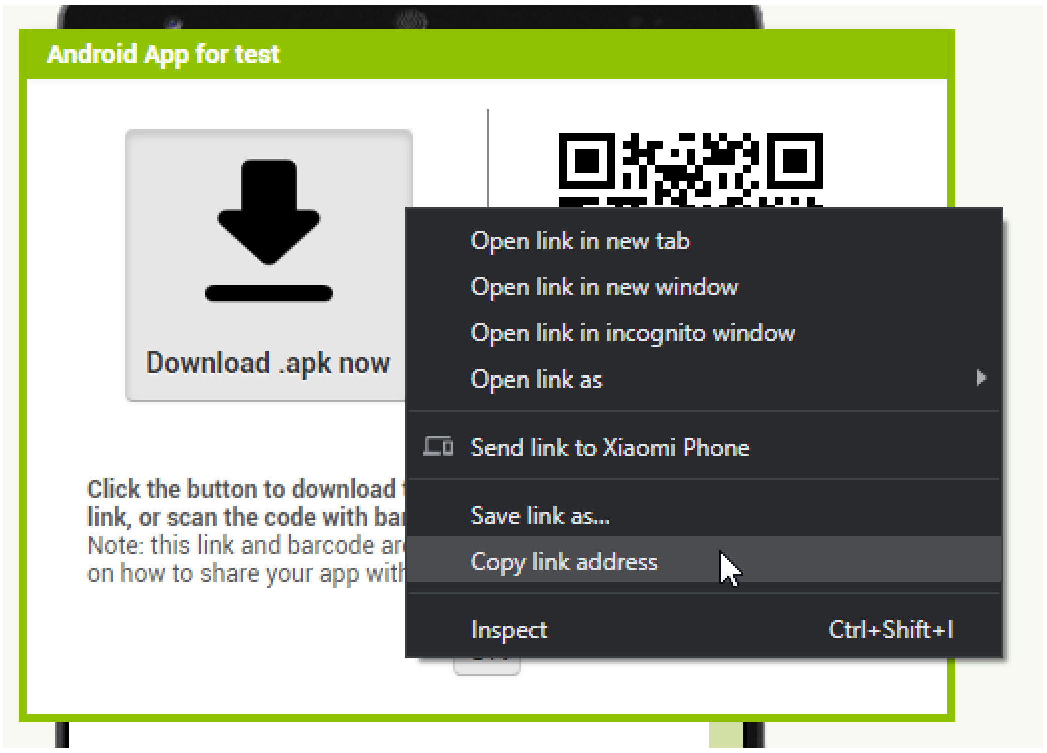
## Final approach

After all those variants, a final decision had to be made, so this ends up after taking into account all the suggestions given by mentors, students and community as well. The final layout chosen for the export app is the following one:


Android App for test

Click the button to download the app, right-click on it to copy a download link, or scan the code with barcode scanner to install.
Note: this link and barcode are only valid for 2 hours. See the FAQ for info on how to share your app with others.

As it can be seen, it has a horizontal layout on which both button and barcode have the same size, giving both the same importance. As it can be seen, the title is “Android App for *$app_name*”, as it does not only involve a download app action. The warning label has been modified to include more information. A vertical thin line has been added between both elements, as a separator. And the “button” is not actually a `<button>`, it is an anchor `<a>` tag faking the same style as a button, which allows to copy the download link to the clipboard.


Android App for test
Download .apk now
Click the button to download
link, or scan the code with ba
Note: this link and barcode ar
on how to share your app wit
Open link in new tab
Open link in new window
Open link in incognito window
Open link as
Send link to Xiaomi Phone
Save link as...
Copy link address
Inspect Ctrl+Shift+I

This allows also to make a long-click in mobile devices to get the download link and sent it somewhere.

## Uploading to Google Play Store

Android App Bundles require App Signing by Google Play. This means app developers will have to provide their private signing keys to Google so they can sign the auto-generated APK files.

It is important to mention that once opting-in for App Signing in any app, it cannot be disabled. So, once an app gets an update using an Android App Bundle, it will require all future updates to be in AAB files, because APK files do not currently support App Signing.

## Open questions

1. **Which QR Code dialog seems a better option?**
   The question here is what is more important: noticing the linking between the link and the items, or to primarily show the “download now” button. Are their labels fine?
   Also, any other mockup is very welcome.

2. **“Build” dropdown labels**

   Are those labels fine? Which other ones could be picked to avoid confusion in long-term users who don’t check documentation as usual as recently-joined users? Maybe adding some mention to “publish” or “store” could help?

3. **What about adding a button to create both APK and AAB in the same process?**

   It may be useful for some people to get both APK and AAB from the same build request, in a ZIP file. However, there is a tool called `bundletool` (and that will be needed to integrate in MIT App Inventor) that already does this task.

4. **Does the link in the dialog still make sense?**

   The user is actually getting a button and a QR code, that both point to the link. Is the anchor element still needed?

5. **Anything else important I may have missed?**

# For core developers

The big changes reside in this part. Building an AAB file is going to be part of the MIT App Inventor Compiler. The main work will reside in the `Compiler.java` file, the process which transforms an App Inventor project into an installable file. However, `BuildServer.java` will also need to be modified, as it has to handle a new parameter that indicates whether to build an APK or AAB.

## Handle AAB export requests

There are two main changes needed to handle Android App Bundle export requests: modifying in the `appengine/` folder the click handler of the dropdown item, to trigger the appropriate request to the buildserver, and adding a new parameter in the `buildserver/` to detect whether the request is for APK or AAB, to consequently call `Compiler.java` to export the app. An implementation option is to create a new parameter at the buildserver which is set to true when the user requests an AAB file.

### Changes in App Engine

Going to `TopToolbar.java`, it can be found the `BarcodeAction` class, which is triggered with the dropdown item click. Then, a tree of invokes comes to action:
`BuildCommand.execute() > ProjectServiceImpl.build() > YoungAndroidProjectService.build()`
In the last function, the buildserver URL constructor is called (`L677:682`), so at that point the URL should change between the normal APK and the new AAB file. It could be something like:

```
buildServerUrl = new URL(getBuildServerUrlStr(
          user.getUserEmail(),
          userId,
          projectId,
          secondBuildserver,
          outputFileDir,
          isAab));
```

Where isAab is a boolean that has been inherited from the parent call at `TopToolbar.java`. By using just one boolean, all the current code can be reused, without needing to write anything new. Then, in the `getBuildServerUrlStr`, it could be specified as a new parameter for the URL like “aab=1”, which when isAab is true inserts this parameter in the returned URL.

### Changes in BuildServer

The next step is handling the AAB request in the `BuildServer.java` file. To do so, the three build path functions should add a new `@QueryParam("aab") boolean isAab`, so it can be detected whether the compiler should build the APK or an AAB. In addition, as there is no “default value” for parameters in Java (like in Python it could be like `isAab=False`), it may be useful to use the method overloading to specify that by default the boolean shall be false (it should build APK by default). Next, the boolean should be propagated to `ProjectBuilder.java`, so it reaches the `Compiler.java` file to process the build request.

Also, it would be a good idea to modify `Main.java` as well, allowing to build AAB files from ZIP from command line. This could be useful to generate a companion bundle for Play Store. It should be only required to handle an `@Option(name = "--aab") boolean isAab = false;`, and then pass that boolean to `ProjectBuilder.java`, like stated above. This may improve the build speed of the system by reusing compilation work between the PlayApp, Emulator, and PlayAppExtras build targets.

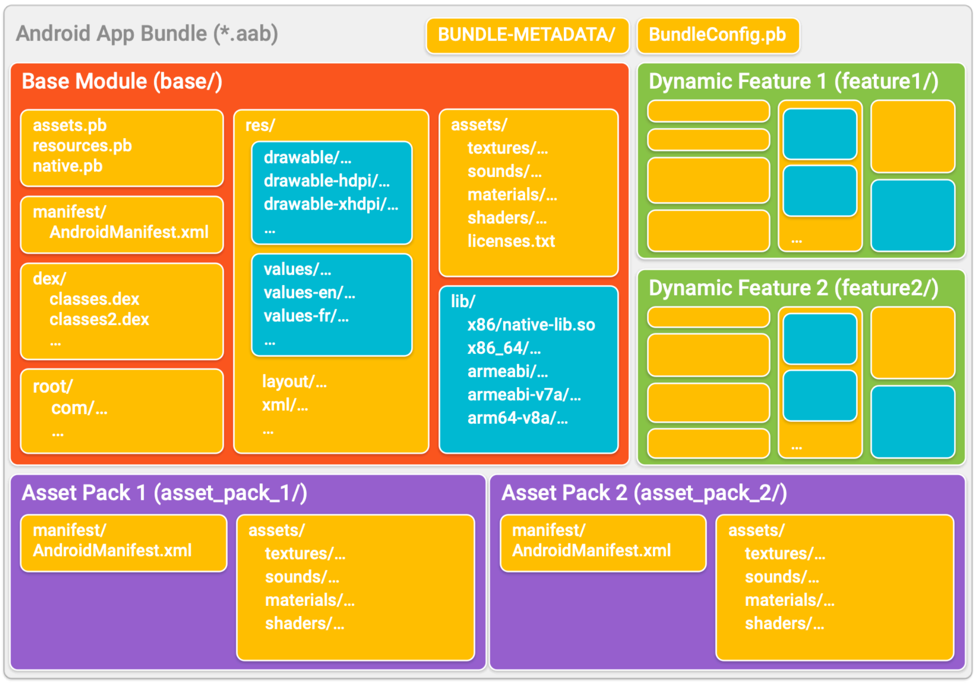
## Implementing AAB exports in buildserver

In order to build an AAB file from the existing building procedure in MIT App Inventor, it has to be understood how an Android App Bundle is different compared to an Android APK file. Although they are quite similar, it is not as easy as “replace .apk to .aab”.

This link may be useful to get started with the process: Build your app from the command line | Android Developers.

### Going inside Android App Bundles

The inside of an Android App Bundle looks like this:



Structure of an Android App Bundle

In the image, there can be seen something called “Dynamic Feature”, and those are not needed for the project right now [1]. The same applies “Asset Pack” [2]. Adding both of those features would imply a bump in the Minimum SDK in App Inventor, and adding more complex blocks, confusing users.

So, finally, the contents of the Android App Bundle generated with MIT App Inventor are as follows:

- `assets.pb`: It is a protocol-buffer file, that contains the table/list of all the assets that are in the app (under `assets/` folder).
- `resources.pb`: It is the same as the `assets.pb` file, but pointing to the `resources/` files. It is the equivalent to `resources.arsc` file in normal package files.
- `native.pb`: And it is the same as `resources.pb`, but pointing and linking the native files specific for each architecture, under the `lib/` folder.
- `manifest/AndroidManifest.xml`: The normal Android manifest file needed in every app. But instead of being in binary like in normal .apk files, it is in protocol buffer encoding, making it easier for parsers.
- `dex/`: Contains all the `classesY.dex` files needed to run the app.

[1] A Dynamic Feature is a feature that does not get installed when the app does. Instead, it is downloaded from Google Play Store when the user requests it, causing some “waiting time” for them.
[2] An Asset Pack is a set of assets that are not installed when the app does (or they do for really large files, like “downloading additional files…”). They only get downloaded when the user requests it.

- `res/`: Contains all the resources files, specifying the screen density of each specific one.
- `assets/`: Contains all the media assets needed for the app. They can have a structure like `assets/<name>#<key>_<value>/...`, where key can be any of *lang*, *tfc*, *opengl* or *vulkan*, specifying the type of asset.
- `lib/`: Contains all the specific architecture files, that make the app work on the given configuration.

With those files set properly, it is only needed to run a zip call to bundle all generated files into the AAB file, being ready for distribution.

## Changes needed in the process

In order to transform the files into AAB-compatible ones, some methods should be created in `Compiler.java` to make the needed modifications. These methods must be invoked in the `compile()` function when the `aab` boolean is set to true. The step where they should run is probably after dexing the sources and before the ~~zipalign~~ apkbuilder step, where all files are bundled into a zip, to then get signed and exported as APK. Moreover, if the AAB build is requested, signing gets delegated to Google Play Store (as they have to sign every single possible APK combination). Just the bundle has to be signed to authenticate its upload, meaning that App Signing will be required in Android App Bundles from MIT App Inventor.

The following components of an APK have to be converted to compatible ones in Android App Bundles:

- `assets.pb`: This file should be created and encoded as protocol-buffer specifying the resources at `assets/`.
- `resources.pb`: Either taking the generated resources.arsc file and converting it into a protocol-buffer one, or crafting a new one.
- `native.pb`: This file should be created and encoded as protocol-buffer specifying the resources at `lib/`.
- `manifest/AndroidManifest.xml`: It goes into a folder rather than in the root of the file.
- ~~`dex/`: No changes.~~
- ~~`res/`: No changes.~~
- `assets/`: Convert all files folders' structure to the appropriate one: `assets/<name>#<key>_<value>/...`. Rather than moving all files into the correct folder, it may be

worthy to consider placing them in the correct one in the `attachCompAssets()` function.

- ~~`lib/`: No changes.~~

## Open questions

1. **Is it fine to use a parameter, or should be a different endpoint?**
   The current document contemplates to use a parameter, so less code should be added to the `BuildServer.java` file.

2. **Should a new file be created rather than using the already existing compiler?**
   It is considered that the same `Compiler.java` file will be used, and some functions will be added to make the required changes in the compiled files.

3. **Should an option to create an APK from an AAB be added to the command line?**
   This would be useful for companion, so that when invoking a command line compiler, there could be a parameter to generate an universal APK from the exported AAB using `bundletool`.

4. **Anything I may have missed in the proposal?**
   Any other comments about the implementation method? Things that need clarification?

# Miscellaneous

Adding such a feature to export apps as Android App Bundles can mean an abrupt change to users who have been using MIT App Inventor for a while, so they may get confused when they see a new export choice. Thus, it should be clear enough what it implies for them, and which new possibilities they have with this export option.

## Documentation

Documentation is important in development, but not as much for this feature (there are no blocks or properties that shall need an explanation). Instead, maybe two articles in the reference website could clear more doubts: one that explains Android App Bundles and for what they can be used, and another one with a step-by-step guide on how to distribute AAB files through Google Play Store.

## Testing

Currently, the buildserver lacks individual testing of features. After getting Android App Bundles working in MIT App Inventor, it could be considered to split the large `Compiler.java` functions into different classes, allowing them to be tested individually. This would, however, mean a total refactor of the current App Inventor compiler, changing how it internally works.