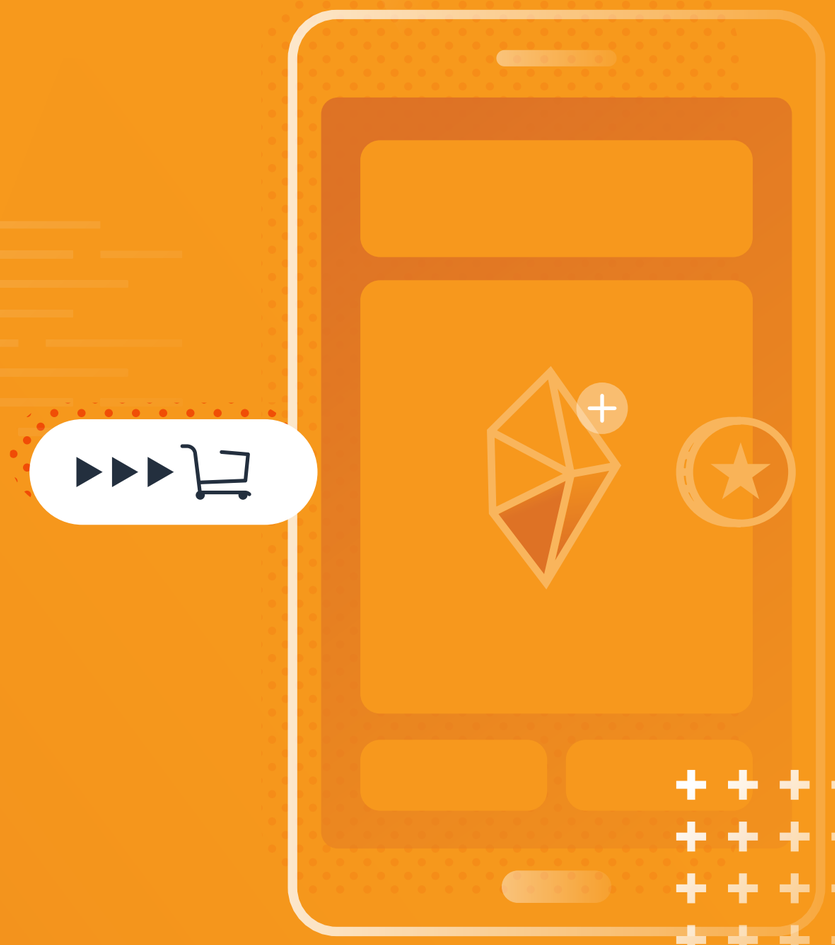
Amazon Developer CodeLabs

How to Add Amazon **In-App Purchases & Subscriptions** to Android Apps



amazon appstore

Mario Viviani
Technology Evangelist,
Amazon Appstore EU

Android developer since 2010

95+ apps published
12,000,000+ downloads

Google Developer Expert 2013-15

Startup Founder, Co-Worker

Speaker at: Droidcon, Casual Connect,
Big Android BBQ, Google I/O

Introduction

Since the early days of mobile development, monetization has always represented a key aspect for developer success.

The very first monetization model, still popular today among developers, was the paid app model. Also called “premium,” this model allows developers to monetize their content by charging for it up front, but it carries an inherent risk. This paywall often represents an obstacle to user acquisition.

Later came advertising, which also continues to be a viable app monetization model. Ads allow developers to give out their app for free, which can help propel download rates and contribute to an app’s popularity. Ads have their own associated risk, however, since they may spoil the user experience.

Mario Viviani
Technology Evangelist,
Amazon Appstore EU

Android developer since 2010

95+ apps published
12,000,000+ downloads

Google Developer Expert 2013-15

Startup Founder, Co-Worker

Speaker at: Droidcon, Casual Connect,
Big Android BBQ, Google I/O

Introduction

In an effort to offer the best of both worlds—removing upfront costs while still charging for some content—a new monetization model was introduced: in-app purchasing (IAP). In-app purchasing offers a low barrier to entry since there is no cost to download or install an IAP app. However, access to premium content and features is gated by a paywall inside the app itself. This combination of “free download” plus “premium content” is why IAP is also known as “freemium.” It is now one of the most popular monetization models for developers creating apps and games today, if not *the* most popular.

IAP has been especially successful for developers offering subscription content like news, magazines, and streaming media. Because the subscription model allows for recurring customer payments at regular intervals (weekly, monthly, quarterly, etc.), a key benefit is the long-term revenue stream it can help build for your app.

Mario Viviani
Technology Evangelist,
Amazon Appstore EU

Android developer since 2010

95+ apps published
12,000,000+ downloads

Google Developer Expert 2013-15

Startup Founder, Co-Worker

Speaker at: Droidcon, Casual Connect,
Big Android BBQ, Google I/O

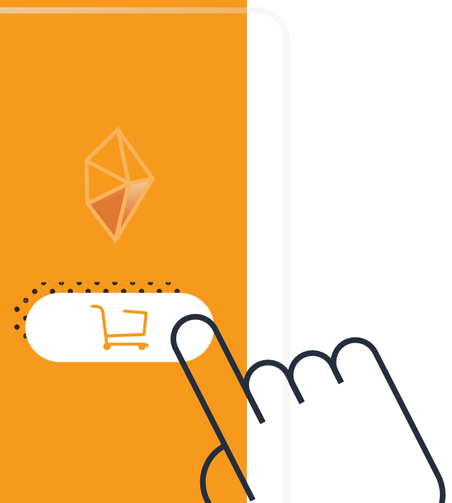
Introduction

In this workshop, we will guide you through the steps necessary to add IAP to your Android app. Together, we will implement, from scratch, an Android application that uses the Amazon In-App Purchase APIs to allow in-app purchase of a magazine subscription.

-Mario Viviani, Amazon (@mariuxtheone)

Duration: 1hr

Goal: Build an Android app that implements quarterly and monthly subscriptions to a magazine using the Amazon In-App Purchase SDK.



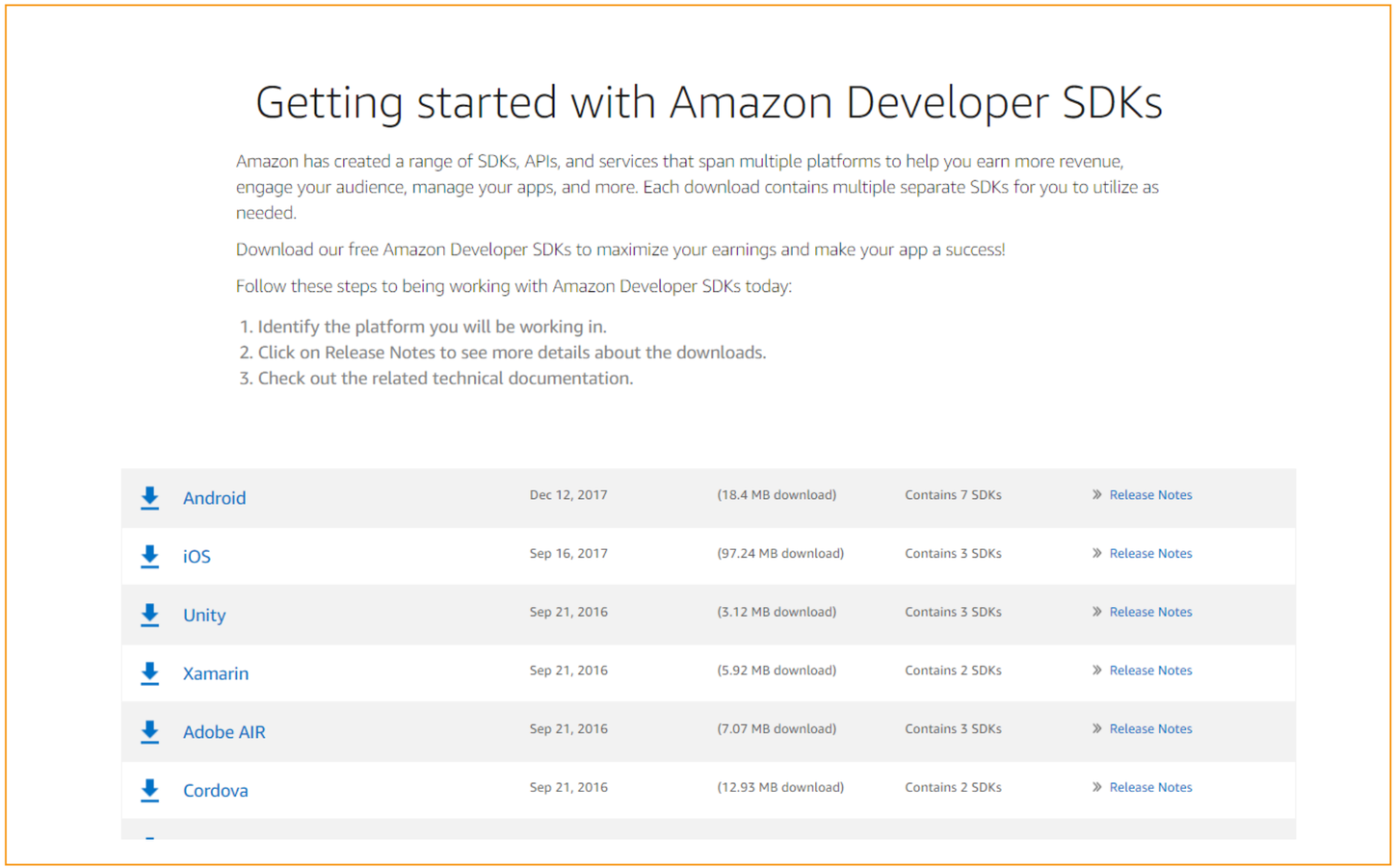
▷▷ Getting Started

Pre-requirements

- Beginner experience with Android/ Java development
- Laptop (Win/Mac/Linux)
- Install Android Studio (version 3.0.1+)
- Install Amazon Appstore on your device (if Android device)
- Download Amazon App Tester from the Amazon Appstore
- Register as an Amazon Appstore Developer

Devices required

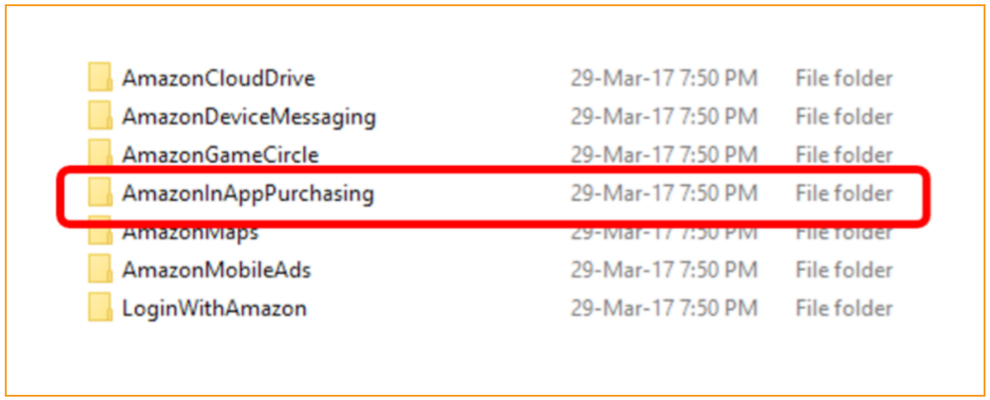
Amazon Fire Tablet (2015+) or an Android device (Android 5+ Lollipop required) with developer options turned on, and all the Android Debug Bridge tools installed, along with the device drivers.



Step 1:

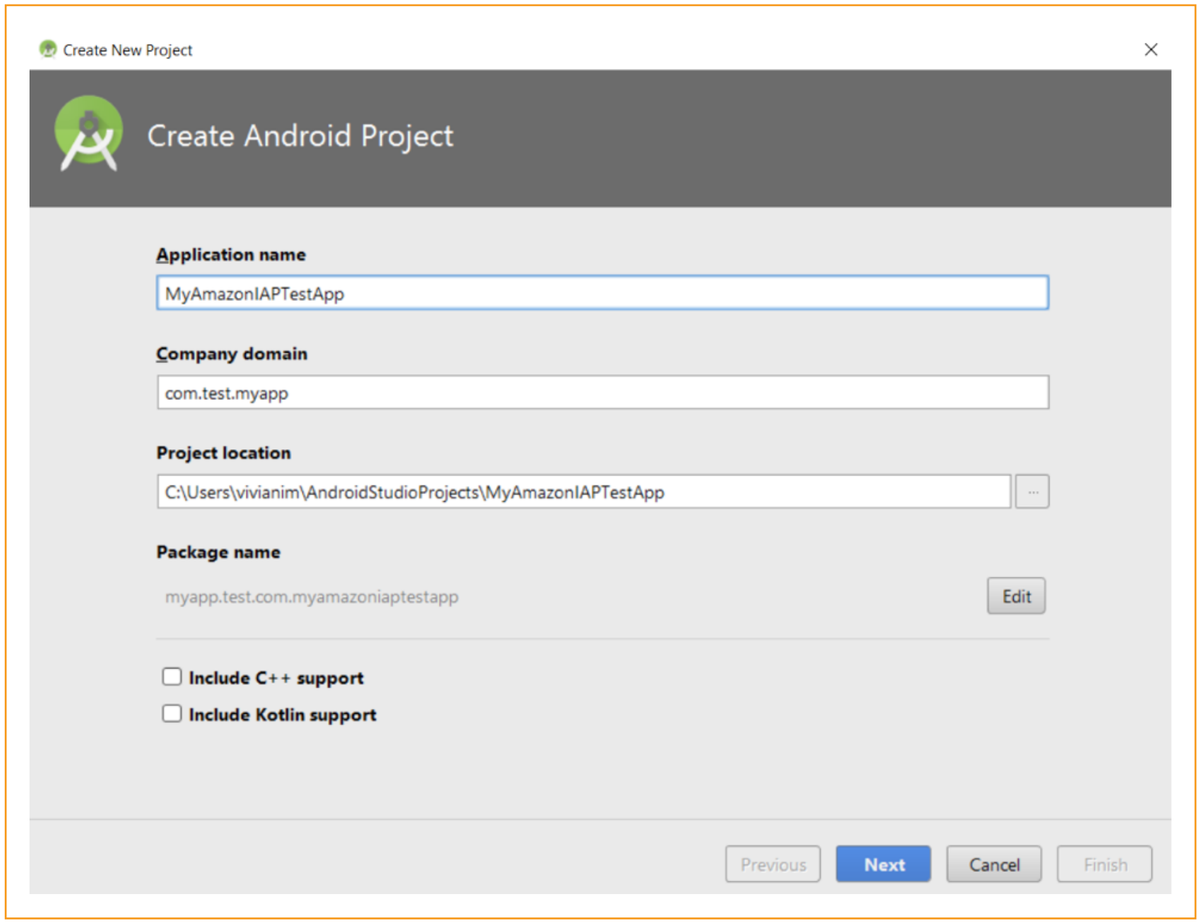
Download the Android SDK from the Amazon SDK download page

Download it from here:
developer.amazon.com/sdk-download



Unzip the SDK and open InAppPurchasing folder

This folder contains the IAP SDK, IAP Coding Samples and a test JSON file that we are going to use later on.



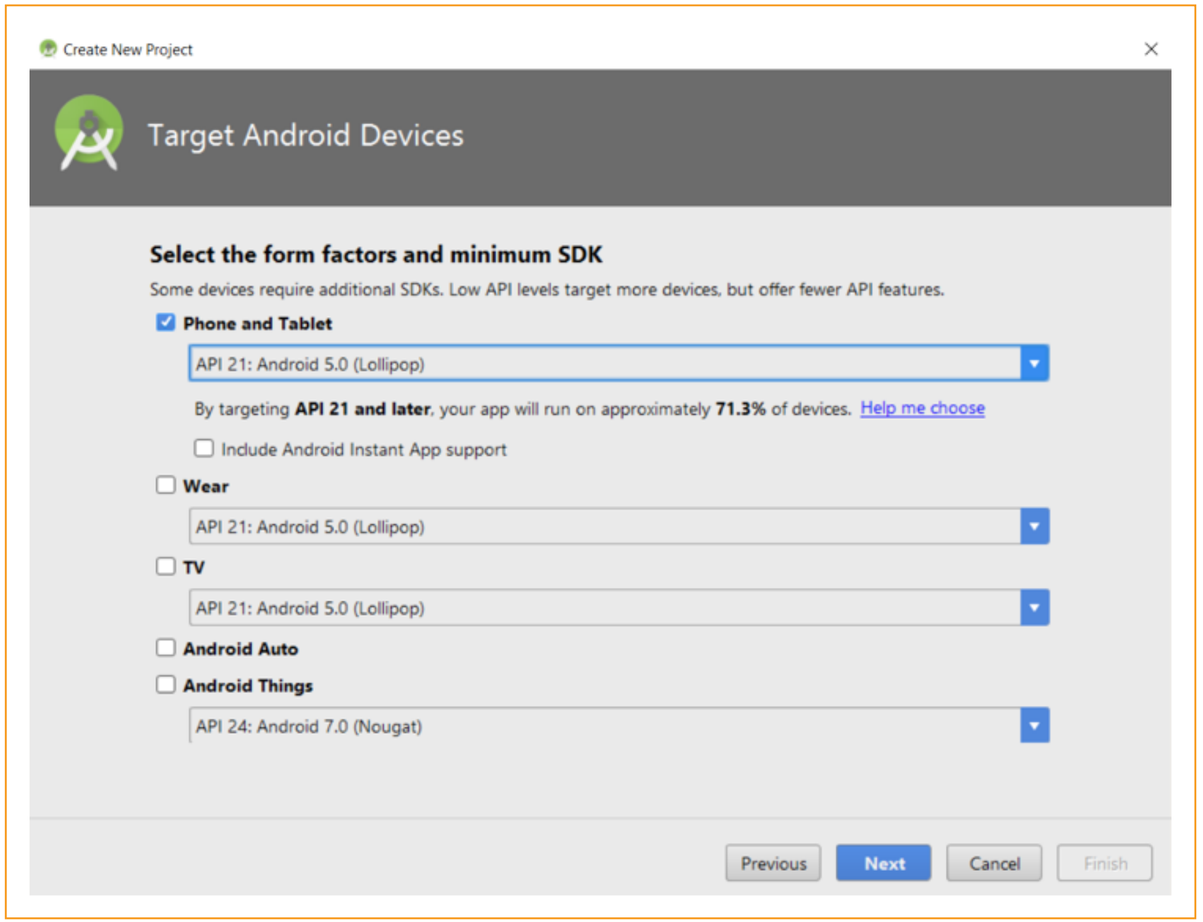
Step 2:

Create a new Android app with Android Studio

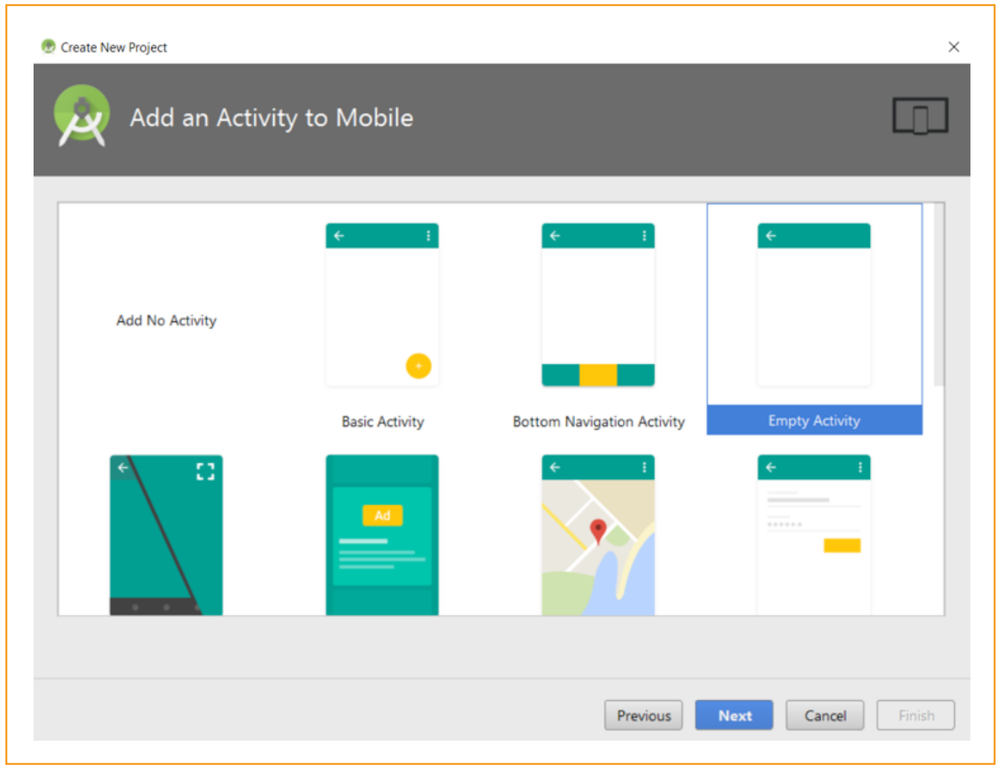
Create a new Android Studio Project with these details:

- **Application name:** MyAmazonIAPTestApp
- **Company domain:** com.test.myapp
- **Package name (should be created automatically):** myapp.test.com.myamazoniaptestapp

Note: Make sure you create a Java project and not Kotlin.



Select **API 21 (Lollipop)** as minimum SDK. Amazon devices are running on API 21 (Android 5.0 Lollipop)



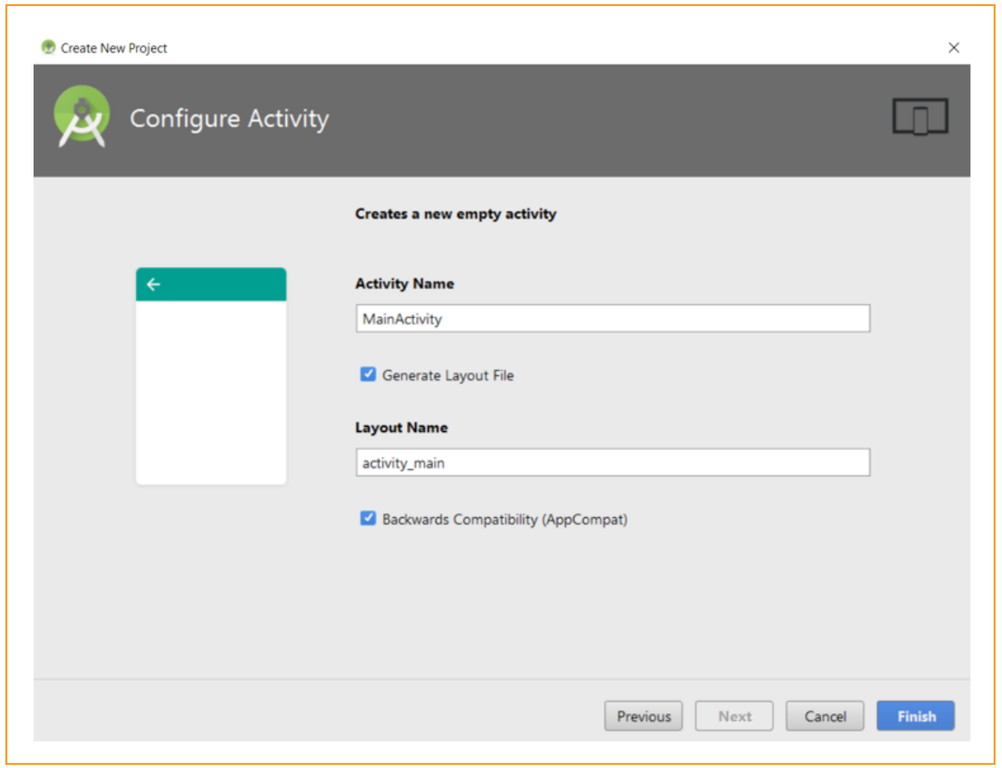
Create a new Empty Activity called **MainActivity**. This will create the Java class,

`MainActivity`

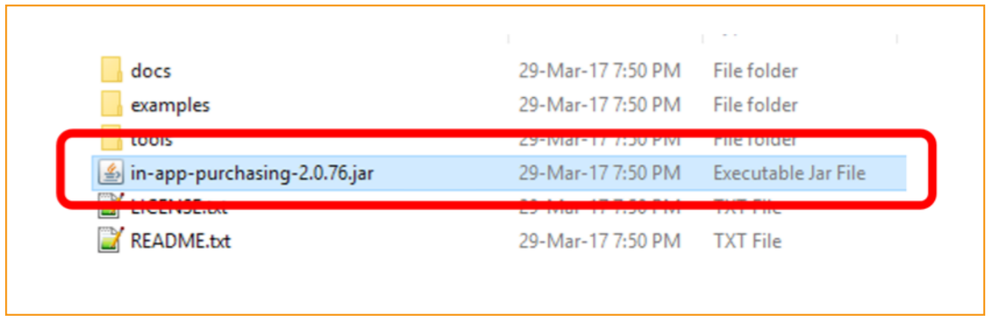
and the XML layout,

`activity_main.xml`

that we will edit afterwards.



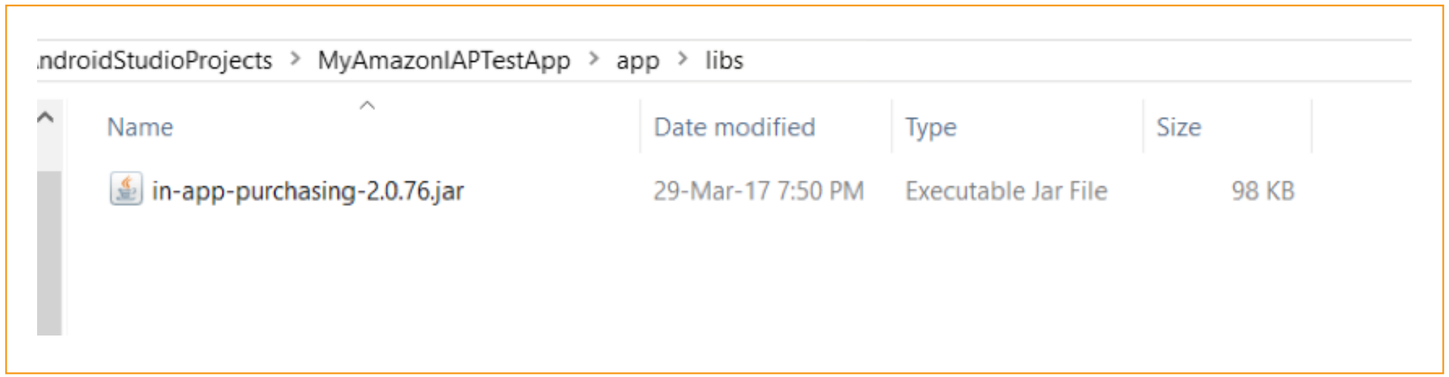
Click **Finish** and let Gradle compile the project (might take a few minutes).



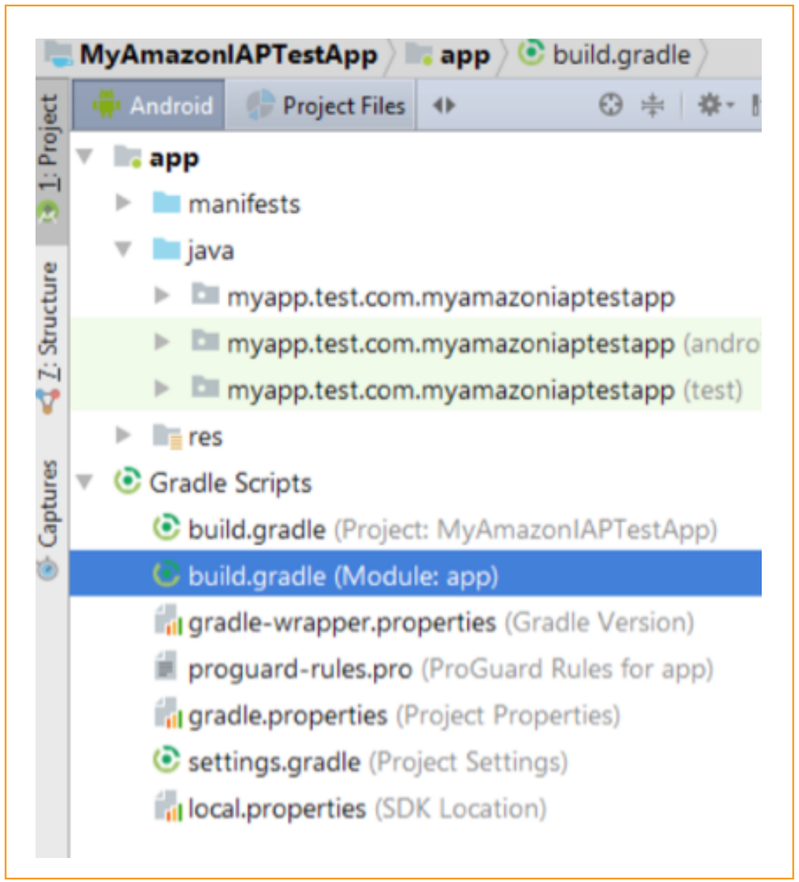
Step 3:

Add the IAP SDK library to your project

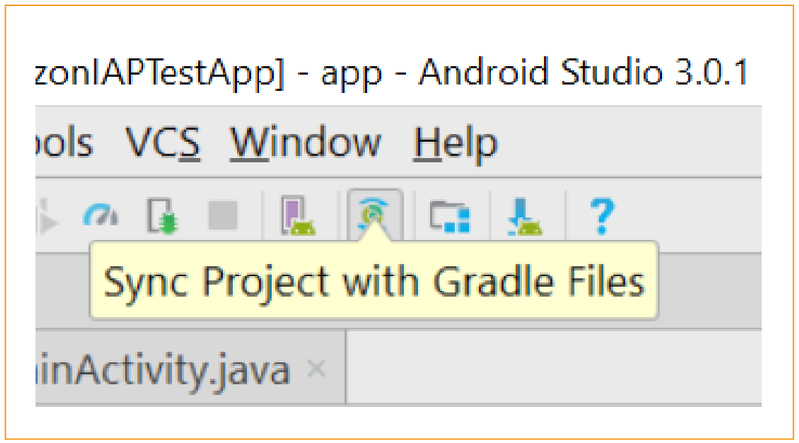
- Open the folder where you unzipped the Amazon Android SDKs.
- Open the AmazonInAppPurchasing folder.
- Copy the **in-app-purchasing-2.0.76.jar** JAR file. That's the IAP library we'll later add to our Android Project.



Go to your newly created Android Project folder and find the **app/libs** folder (should be *MyAmazonIAPTestApp/app/libs*) and paste the **in-app-purchasing-2.0.76.jar** so your libs folder looks like this.



In Android Studio, now we need to add the .jar file to your project dependency tree. To do this, open **build.gradle** for the module app.



To add the .jar dependency, under dependencies change:

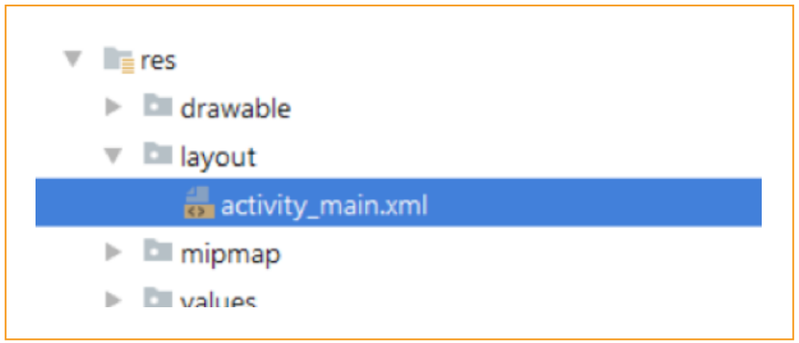
```
implementation fileTree(dir:
'libs', include: ['*.jar'])
```

to,

```
compile fileTree(dir: 'libs',
include: ['*.jar'])
```

Save and sync the project with Gradle files by **clicking on the sync button** on the top left.

Once Gradle finishes compiling, the Amazon IAP SDK will be added to your library and we are ready to create our IAP-enabled app.



Step 4:

Edit the layout of the app

The goal of the app is to allow customers to subscribe to a magazine. Therefore, now we need to edit the layout of the app to include a button that will allow the user to subscribe to the service.

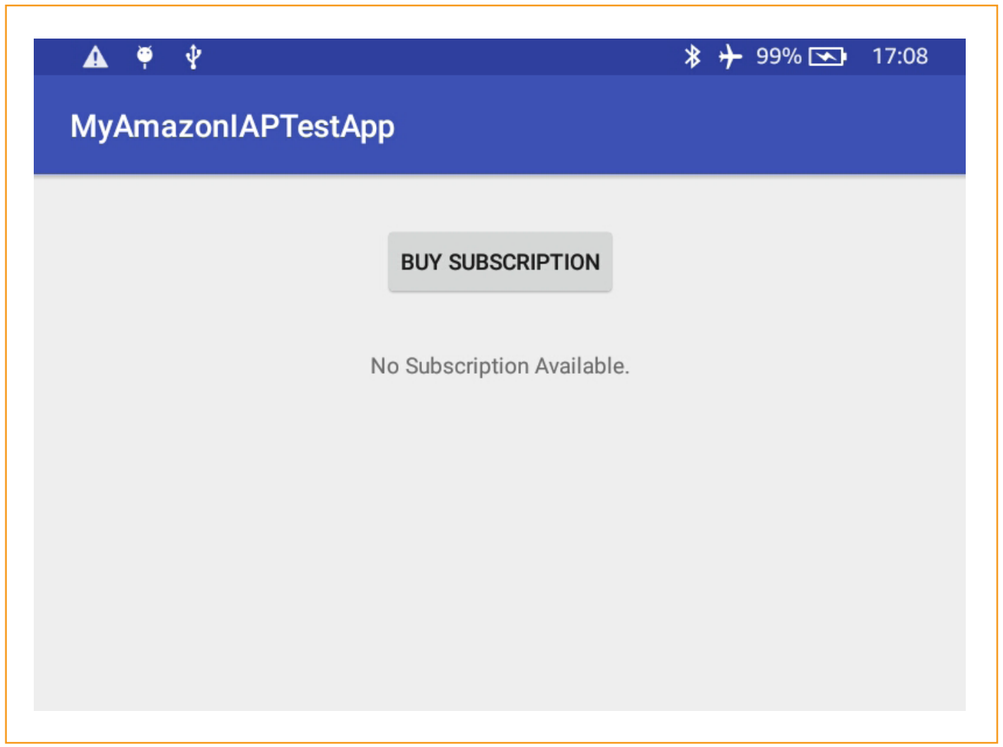
We will need to edit the **activity_main.xml** layout Android Studio has created for our MainActivity.

Open the **activity_main.xml** from the drop-down menu in **res/layout**.

Copy all the text and substitute it with the following:

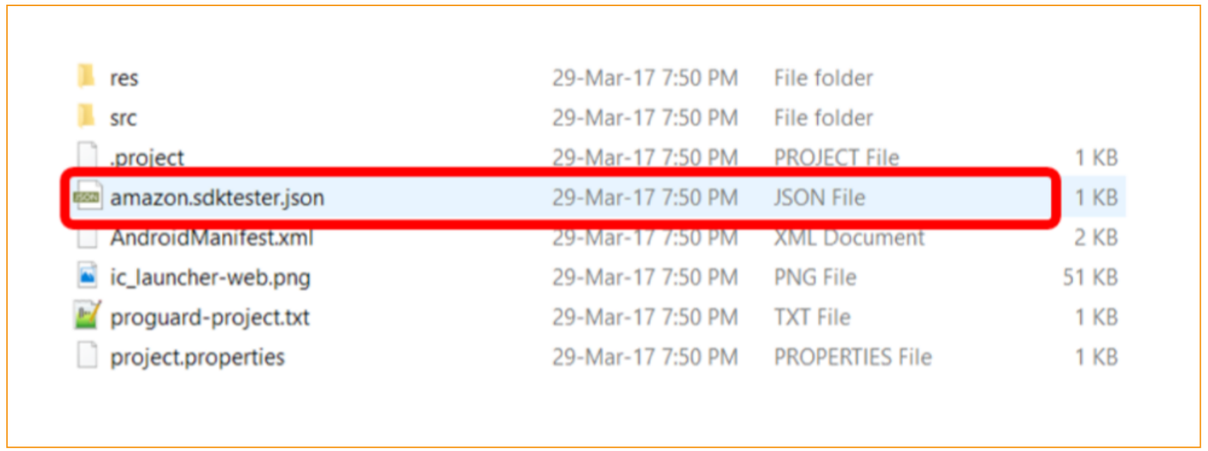
```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="myapp.test.com.myamazoniaptestapp.MainActivity">

    <Button
        android:id="@+id/subscriptionButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Buy Subscription"
        android:layout_marginTop="32dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        android:layout_marginStart="8dp"
        android:layout_marginEnd="8dp" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="32dp"
        android:text="No Subscription Available."
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/subscriptionButton" />

</android.support.constraint.ConstraintLayout>
```

This layout contains a button called **subscriptionButton** (to subscribe to the content) that will allow the customer to initiate the request to subscribe to the content and a TextView called **textView** that will display the status of the subscription.



Save and compile. When you launch, the app should look like this.

Now we have the interface we're going to use to connect to and query the Amazon IAP SDK so the user can buy a subscription.

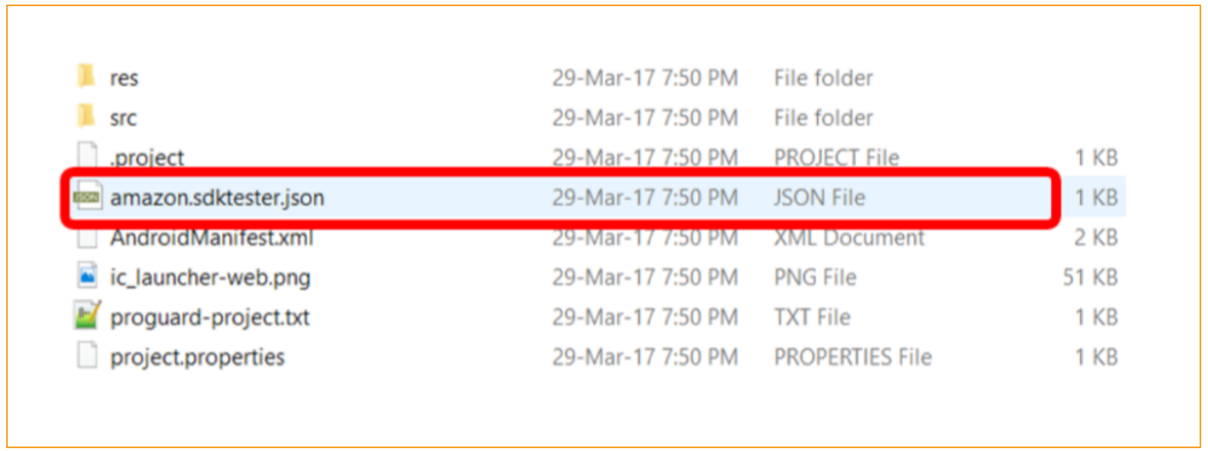


Step 5:

Add the SDK Tester JSON file

In order to test our app's IAP functionality, we will need a special JSON file containing information about all the in-app items that our app offers. We then **copy that file** to our Amazon Fire Tablet/Android device local storage.

In a real-world scenario, you would add your in-app items on the Amazon Developer Portal (see here for details), then create the JSON file for testing from the descriptions you provided for those in-app items.



For the sake of this workshop, we'll **use the JSON provided** in the IAP SDK subscription example.

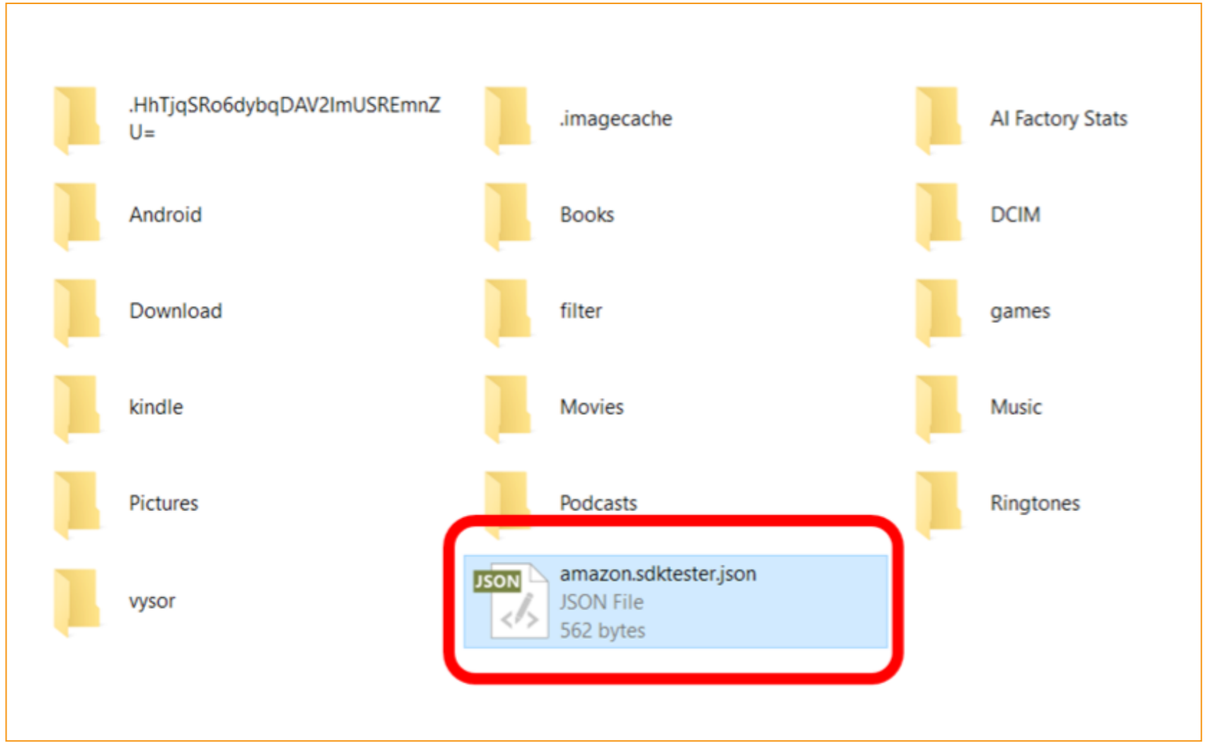
Copy the SDK Tester JSON file on your Android device.

1. Open the folder

Amazon-Android-SDKs/AmazonInAppPurchasing/examples/SampleIAPSubscriptionsApp

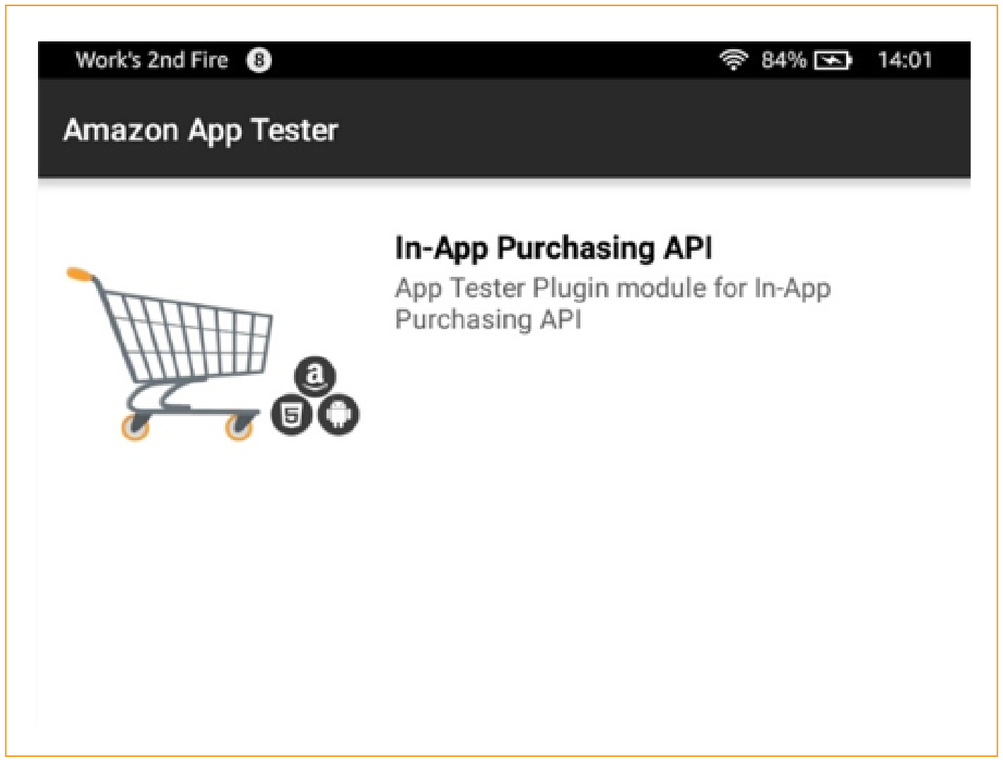
2. Copy the file

`amazon.sdktester.json`



Paste it in your local device (Fire Tablet/Android device) storage main folder.

At this point, your device is almost ready for local testing of the Amazon IAP SDK. The only missing part is the Amazon App Tester App.

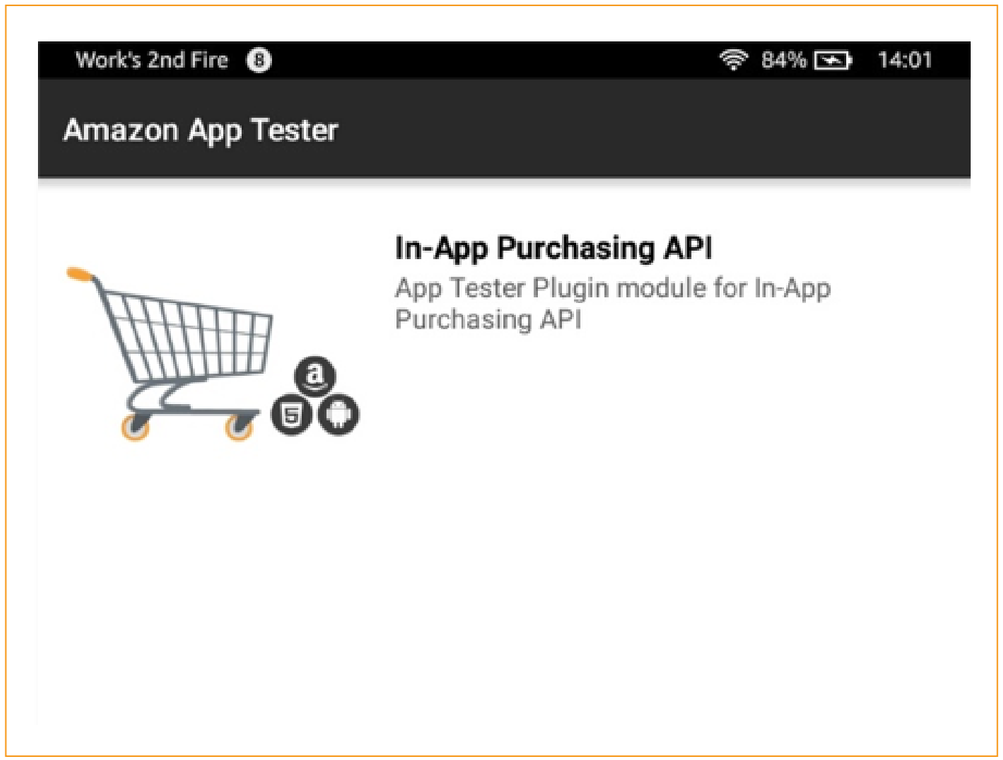


Step 6:

Install the Amazon App Tester App on your Android device

The next step is to download the Amazon App Tester app from the Amazon Appstore.

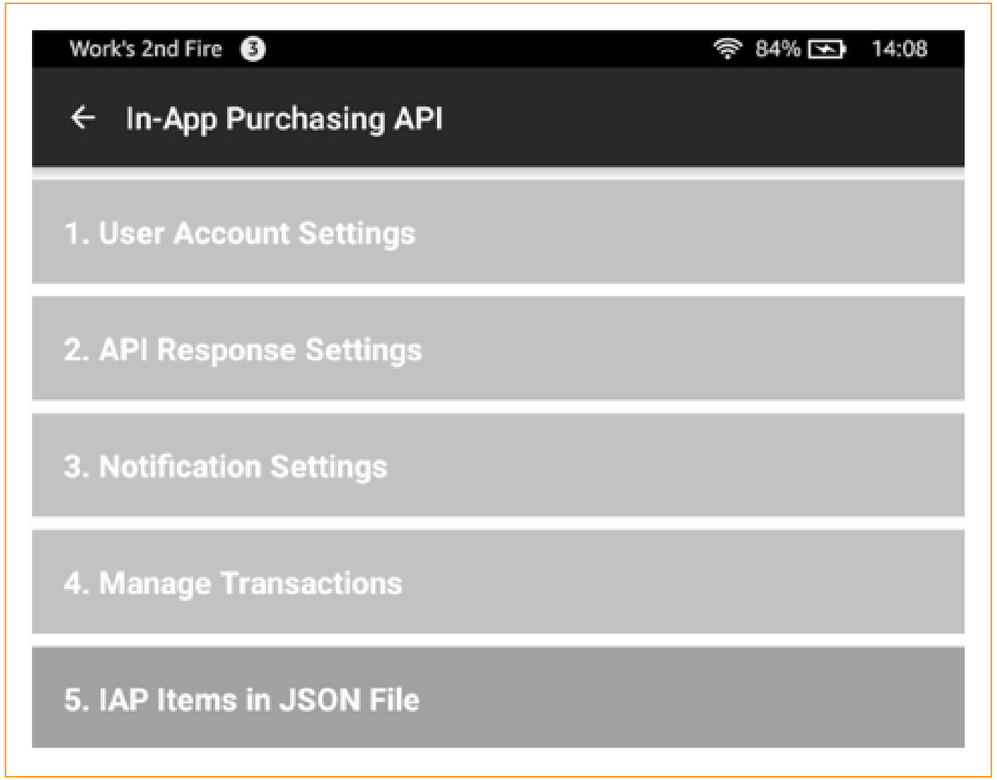
Using the Amazon App Tester, you can test your implementation of in-app shopping in a production-like environment. Testing in this sandbox environment allows you to **ensure that your app integrates** with the IAP API or Mobile Associates API, before submitting it to the Amazon Appstore for Android.



The Amazon App Tester lets you **test your implementation** without having to submit the app to the Appstore or access any cloud-based API.

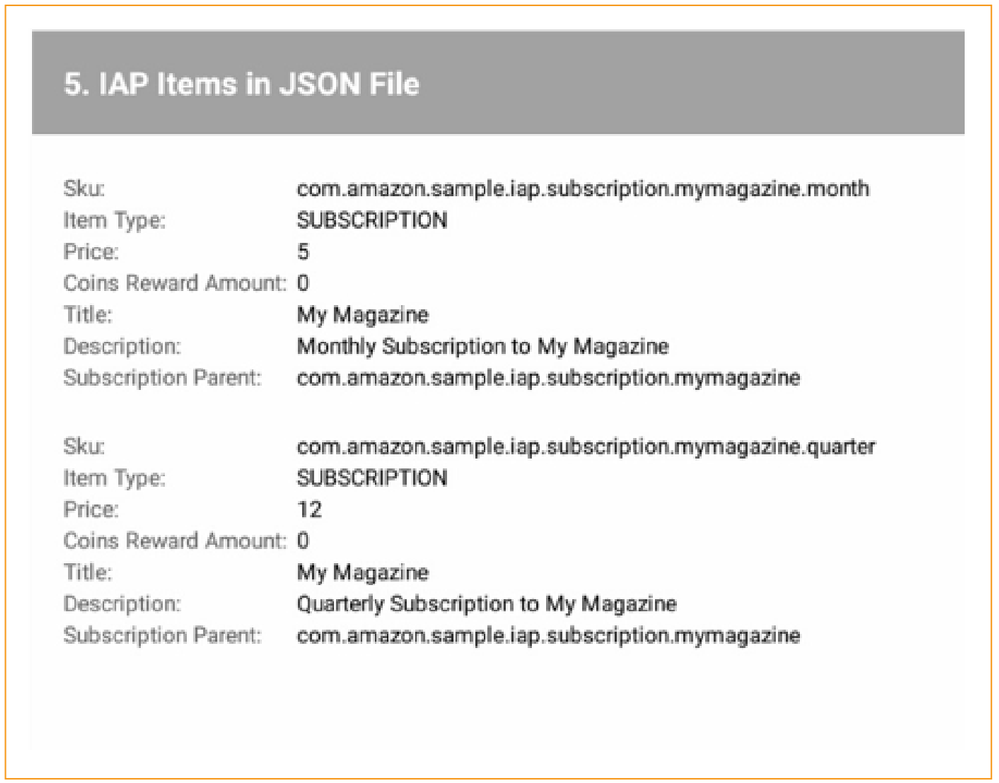
Once you have installed the App Tester on your Fire tablet or Android device, launch the app.

It will look like this.



The App Tester automatically detects that there's an **SDK Tester JSON file** (the previously added amazon.sdktester.json file) in the local storage of the device, and that the file contains in-app items to be used with the in-app purchasing API.

If we click on "**In-App Purchasing API**," a menu containing multiple features opens.



If we click on "**5. IAP Items in JSON File**," we see that the app shows the in-app items contained in the SDK Tester JSON.

In this case, it contains two subscriptions:

- A monthly subscription to "MyMagazine" for $5
- A quarterly subscription to "MyMagazine" for $5

5. IAP Items in JSON File

Sku:	com.amazon.sample.iap.subscription.mymagazine.month
Item Type:	SUBSCRIPTION
Price:	5
Coins Reward Amount:	0
Title:	My Magazine
Description:	Monthly Subscription to My Magazine
Subscription Parent:	com.amazon.sample.iap.subscription.mymagazine
Sku:	com.amazon.sample.iap.subscription.mymagazine.quarter
Item Type:	SUBSCRIPTION
Price:	12
Coins Reward Amount:	0
Title:	My Magazine
Description:	Quarterly Subscription to My Magazine
Subscription Parent:	com.amazon.sample.iap.subscription.mymagazine

The JSON also carries additional important information:

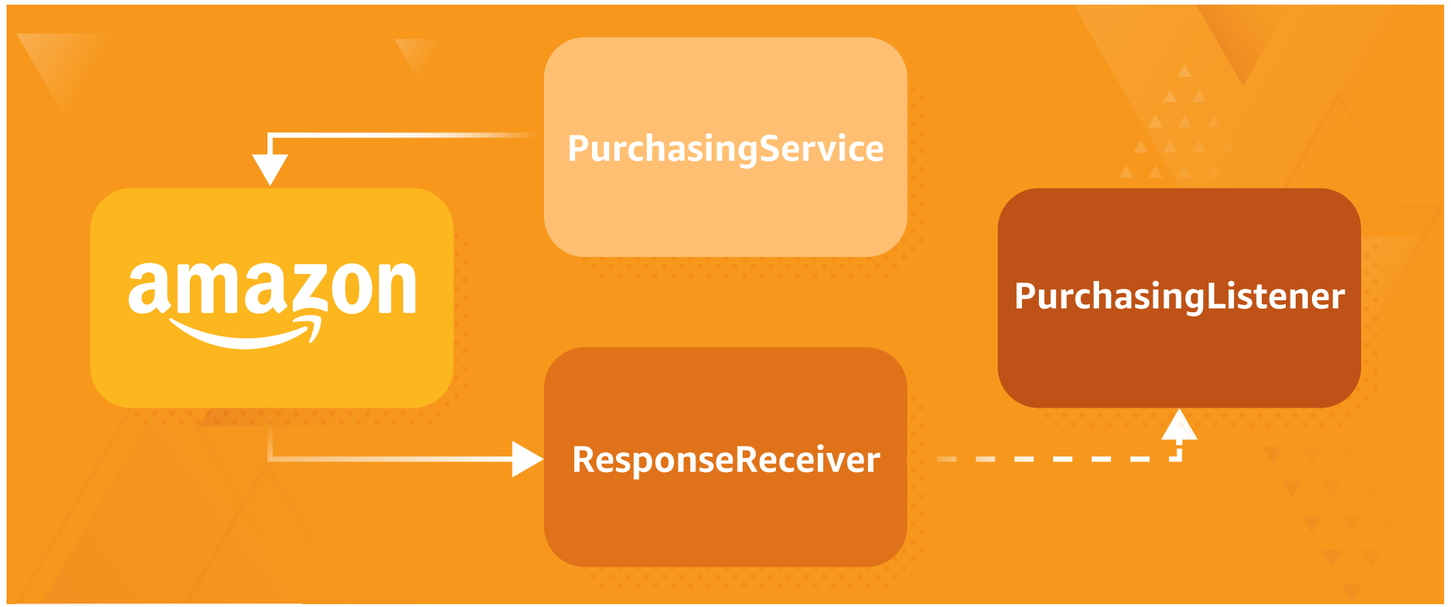
- **Sku:** This is the so-called "Child SKU." The SKU is a "unique product ID" that identifies a specific in-app item.
- **Item type:** This defines what type of in-app item it is. Can be CONSUMABLE, ENTITLEMENT or SUBSCRIPTION.
- **Price:** The price of the in-app item.
- **Title:** The name of the in-app item that will be displayed on the UI of the in-app purchase dialog.
- **Description:** A brief description that will be displayed on the UI of the in-app purchase dialog.
- **Subscription parent:** This is also called "Parent SKU" and uniquely identifies a specific subscription. This relates to the fact that we could set up different periods of subscriptions for the same in-app item. This is the SKU we'll be using to show the in-app purchase dialog in our Android app.

5. IAP Items in JSON File

Sku:	com.amazon.sample.iap.subscription.mymagazine.month
Item Type:	SUBSCRIPTION
Price:	5
Coins Reward Amount:	0
Title:	My Magazine
Description:	Monthly Subscription to My Magazine
Subscription Parent:	com.amazon.sample.iap.subscription.mymagazine

Sku:	com.amazon.sample.iap.subscription.mymagazine.quarter
Item Type:	SUBSCRIPTION
Price:	12
Coins Reward Amount:	0
Title:	My Magazine
Description:	Quarterly Subscription to My Magazine
Subscription Parent:	com.amazon.sample.iap.subscription.mymagazine

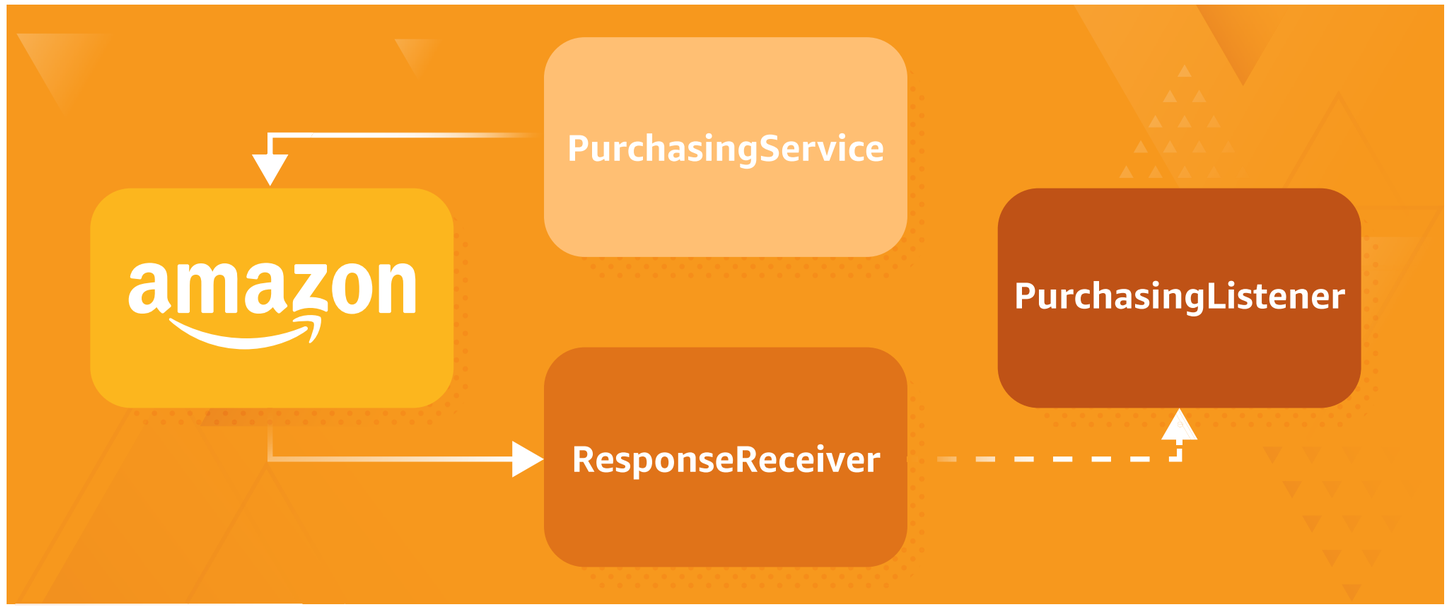
Our test environment is now fully set. We now need to **implement the In-App Purchase API** in our Android app.



Step 7:

Understanding the In-App Purchase API components

There are multiple components we need to add to our Android app to implement the In-App Purchasing API and to allow our user to buy a subscription (or a Consumable or an Entitlement) from the Amazon Appstore.

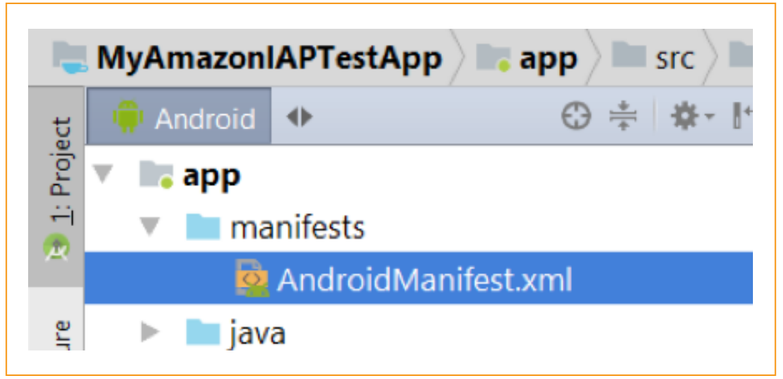


These are the main components:

- `PurchasingService`
  Class that **initiates requests** through the Amazon Appstore.

- `ResponseReceiver`
  Class that **receives broadcast intents** from the Amazon Appstore.

- `PurchasingListener`
  Interface that **receives asynchronous responses** to the requests initiated by the PurchasingService.



Now let's see how to implement these components in our app's main activity.

Find your app **AndroidManifest.xml** file and open it.

Add the ResponseReceiver to your app Android manifest

```
<receiver android:name="com.amazon.device.iap.ResponseReceiver">
    <intent-filter>
        <action
            android:name="com.amazon.inapp.purchasing.NOTIFY"
            android:permission="com.amazon.inapp.purchasing.Permission.NOTIFY" />
    </intent-filter>
</receiver>
```

After the <activity> tag, add the following elements.

Add the ResponseReceiver to your app Android manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="myapp.test.com.myamazoniaptestapp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <receiver android:name="com.amazon.device.iap.ResponseReceiver">
            <intent-filter>
                <action
                    android:name="com.amazon.inapp.purchasing.NOTIFY"
                    android:permission="com.amazon.inapp.purchasing.Permission.NOTIFY"
/>
            </intent-filter>
        </receiver>

    </application>

</manifest>
```

So your Android manifest should look like this.

At this point, our app, through the **ResponseReceiver**, is instrumented to listen to responses from the Amazon Appstore.

Now, let's **add the components** to make requests to the Appstore and to handle the responses in our app.

Querying the Amazon Appstore for the current status of in-app items in onResume()

```
public class MainActivity extends AppCompatActivity {

    String parentSKU = "com.amazon.sample.iap.subscription.mymagazine";
    Handler handler;
```

The first thing we need to do is to add the SKU of our subscription. Since we only have one magazine to manage, we add a string **containing the parent SKU** in our main activity right after we create the **MainActivity** class. We also add a handler, which will allow us to update the interface in the main UI.

(Tip: if you need to update your imports, press Alt+Enter to quickly import the missing classes.)

Querying the Amazon Appstore for the current status of in-app items in onResume()

```
@Override
protected void onResume() {
    super.onResume();

    //getUserData() will query the Appstore for the Users information
    PurchasingService.getUserData();

    //getPurchaseUpdates() will query the Appstore for any previous purchase
    PurchasingService.getPurchaseUpdates(true);

    //getProductData will validate the SKUs with Amazon Appstore
    final Set<String> productSkus =  new HashSet<String>();
    productSkus.add(parentSKU);
    PurchasingService.getProductData(productSkus);

    Log.v("Validating SKUs",  "Validating SKUs with Amazon" );


}
```

Now, we need to **query the Amazon Appstore** in order to make sure we have all the information to 1) populate the interface, and 2) execute the right transactions (previous purchases, user information, SKU availability).

We need to do this operation in the **onResume()** method of our activity.

Querying the Amazon Appstore for the current status of in-app items in onResume()

```
@Override
protected void onResume() {
    super.onResume();

    //getUserData() will query the Appstore for the Users information
    PurchasingService.getUserData();

    //getPurchaseUpdates() will query the Appstore for any previous purchase
    PurchasingService.getPurchaseUpdates(true);

    //getProductData will validate the SKUs with Amazon Appstore
    final Set<String> productSkus =  new HashSet<String>();
    productSkus.add(parentSKU);
    PurchasingService.getProductData(productSkus);

    Log.v("Validating SKUs",  "Validating SKUs with Amazon" );


}
```

These methods from the

`PurchasingService`

will **query the Amazon Appstore** for all the necessary information.

Now, **we need to react** when the Amazon Appstore replies with this information. Our activity needs a

`PurchasingListener`.

Implementing the PurchasingListener

```
PurchasingListener purchasingListener = new PurchasingListener() {

    @Override
    public void onUserDataResponse(UserDataResponse userDataResponse) {

    }

    @Override
    public void onProductDataResponse(ProductDataResponse productDataResponse) {

    }

    @Override
    public void onPurchaseResponse(PurchaseResponse purchaseResponse) {

    }

    @Override
    public void onPurchaseUpdatesResponse(PurchaseUpdatesResponse purchaseUpdatesRe-
sponse) {

    }
};
```

Let's add a `PurchasingListener` to our activity.

The `PurchasingListener` has four main callback methods that we must implement:

1. `onUserDataResponse`: Invoked after a call to getUserData(). Determines the UserId and marketplace of the currently logged on user.

2. `onPurchaseUpdatesResponse`: Invoked after a call to `getPurchaseUpdates` (boolean reset). Retrieves the purchase history. Amazon recommends that you persist the returned `PurchaseUpdatesResponse` data and query the system only for updates.

Implementing the PurchasingListener

```
PurchasingListener purchasingListener = new PurchasingListener() {

    @Override
    public void onUserDataResponse(UserDataResponse userDataResponse) {

    }

    @Override
    public void onProductDataResponse(ProductDataResponse productDataResponse) {

    }

    @Override
    public void onPurchaseResponse(PurchaseResponse purchaseResponse) {

    }

    @Override
    public void onPurchaseUpdatesResponse(PurchaseUpdatesResponse purchaseUpdatesRe-
sponse) {

    }
};
```

Let's add a `PurchasingListener` to our activity.

The `PurchasingListener` has four main callback methods that we must implement:

3. `onProductDataResponse (ProductDataResponse productDataResponse):` Invoked after a call to `getProductDataRequest` (java.util.Set skus). Retrieves information about SKUs you would like to sell from your app. Use the valid SKUs in `onPurchaseResponse().`

4. `onPurchaseUpdatesResponse`: Invoked after a call to purchase (String sku). Used to determine the status of a purchase.

User data response

```
//Define UserId and MarketPlace
private String currentUserId;
private String currentMarketplace;
```

The first callback we implement is `onUserdataResponse()`. Let's define two variables outside of the `PurchasingListener` that we'll call **currentUserID** and **currentMarketplace**.

User data response

```
@Override
public void onUserDataResponse(UserDataResponse response) {

    final UserDataResponse.RequestStatus status = response.getRequestStatus();

    switch (status) {
        case SUCCESSFUL:
            currentUserId = response.getUserData().getUserId();
            currentMarketplace = response.getUserData().getMarketplace();
            break;

        case FAILED:
        case NOT_SUPPORTED:
            // Fail gracefully.
            break;
    }
}
```

Now let's override `onUserDataResponse`.

This method **recovers the user ID** and the reference marketplace of the user. In case we can't retrieve this information, or the callback can't be executed, our code should handle those conditions and fail gracefully (we won't cover the implementation details in this workshop, however).

Product data response

```
@Override
public void onProductDataResponse(ProductDataResponse productDataResponse) {
    switch (productDataResponse.getRequestStatus()) {
        case SUCCESSFUL:


            //get informations for all IAP Items (parent SKUs)
            final Map<String,Product> products = productDataResponse.getProductData();
            for ( String key : products.keySet()) {
                Product product = products.get(key);
                Log.v("Product:", String.format( "Product: %s\n Type: %s\n SKU: %s\n
Price: %s\n Description: %s\n" , product.getTitle(), product.getProductType(),
product.getSku(), product.getPrice(), product.getDescription()));
            }

            //get all unavailable SKUs
            for ( String s : productDataResponse.getUnavailableSkus()) {
                Log.v("Unavailable SKU:"+s,  "Unavailable SKU:" + s);
            }

            break;


        case FAILED:
            Log.v("FAILED",  "FAILED" );
            break ;
    }
}
```

onProductDataResponse allows us to get information and details about all IAP items in the catalog and all unavailable SKUs. This comes in handy if we want to strike through specific IAP Items in our app UI.

Making a purchase

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    PurchasingService.registerListener(this, purchasingListener);
```

The first thing we have to do is to **register the PurchasingListener**. Let's do this in our MainActivity onCreate() method.

Making a purchase

```
Button button = (Button) findViewById(R.id.subscriptionButton);
    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            PurchasingService.purchase(parentSKU);

        }
    });
```

In order to allow the user to make a purchase, we need to **add a ClickListener** to our Button.

The PurchasingService.purchase() method will initialize the purchase for a specific SKU (in this case, our parentSKU for the magazine).

Making a purchase

```
//create a handler for the UI changes
handler = new Handler(Looper.getMainLooper()){
    @Override
    public void handleMessage(Message msg) {
        if (msg.obj.equals("Subscribed")){
            TextView textView = (TextView) findViewById(R.id.textView);
            textView.setText("SUBSCRIBED");
            textView.setTextColor(Color.RED);

        }
    }
};
```

Then, we need to add a component to show on-screen if we are effectively subscribed to the magazine. **Let's add this in our onCreate()**.

The handler will wait for the message "subscribed," which will indicate that we should update the **TextView** color and text to show that we're subscribed to the content.

Making a purchase

```
@Override
public void onPurchaseResponse(PurchaseResponse purchaseResponse) {

    switch (purchaseResponse.getRequestStatus()) {
        case SUCCESSFUL:

PurchasingService.notifyFulfillment(purchaseResponse.getReceipt().getReceiptId(),
                            FulfillmentResult.FULFILLED);
            break ;

        case FAILED:

            break ;
    }


}
```

Now, we need to handle the `onPurchaseResponse()` callback in our `PurchaseListener`. `onPurchaseResponse()` is triggered only when the user **makes a specific purchase** for the first time.

If the purchase is **successful**, then we can fulfill the item to the customer using `PurchasingService.notifyFulfillment()`.
This would trigger the `onPurchaseUpdateResponse()` callback, and that's **the last component** that we need to implement.

Fulfilling the item to the user

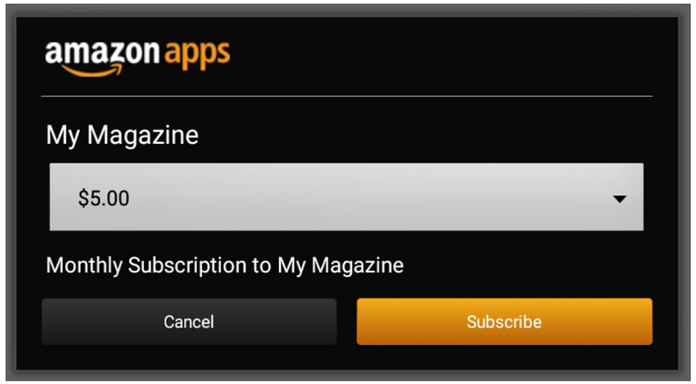
```
@Override
public void onPurchaseUpdatesResponse(PurchaseUpdatesResponse response) {


    // Process receipts
    switch (response.getRequestStatus()) {
        case SUCCESSFUL:
            for ( final Receipt receipt : response.getReceipts()) {
                // Process receipts
                if (!receipt.isCanceled()){
                    Message m= new Message();
                    m.obj="Subscribed";
                    handler.handleMessage(m);

                }
            }

            if (response.hasMore()) {
                PurchasingService.getPurchaseUpdates(true);
            }
            break ;
        case FAILED:
            Log.d("FAILED","FAILED");
            break ;
    }

}
```

In order to fulfill the item to the user, we need to do it in the `onPurchaseUpdateResponse()`. That's because we would fulfill the subscription any time the user logs in, if the receipt is still valid.

If the status of the Request is **successful**, it means this user has the right to access the item (in this case, the subscription).

We iterate through all the receipts for the user, and **if the receipt is not canceled**, we can send the message "Subscribed" to the handler, which will update the UI accordingly.

Our app is now ready to be tested!

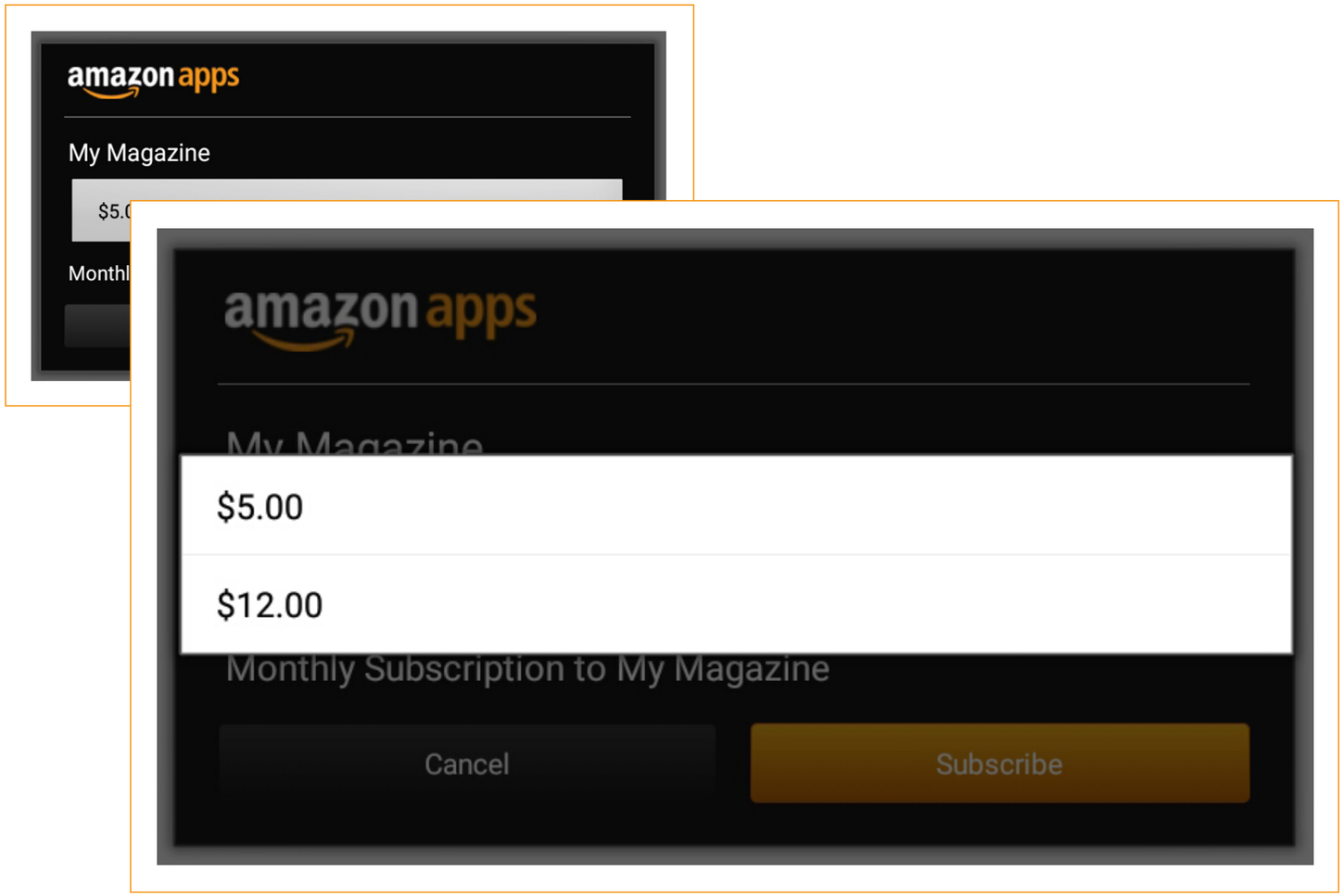


Step 8:

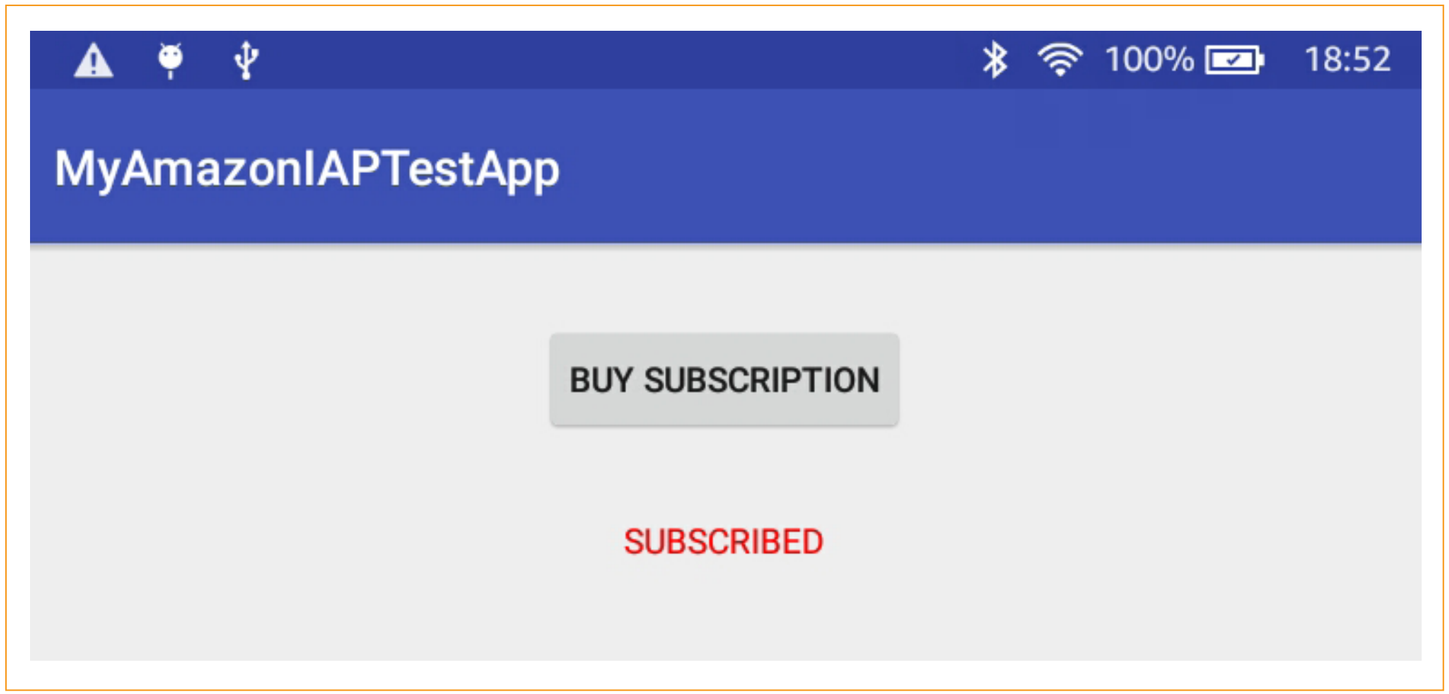
Testing IAP with the Amazon App Tester

Let's click on the "BUY SUBSCRIPTION" button. This UI will appear.

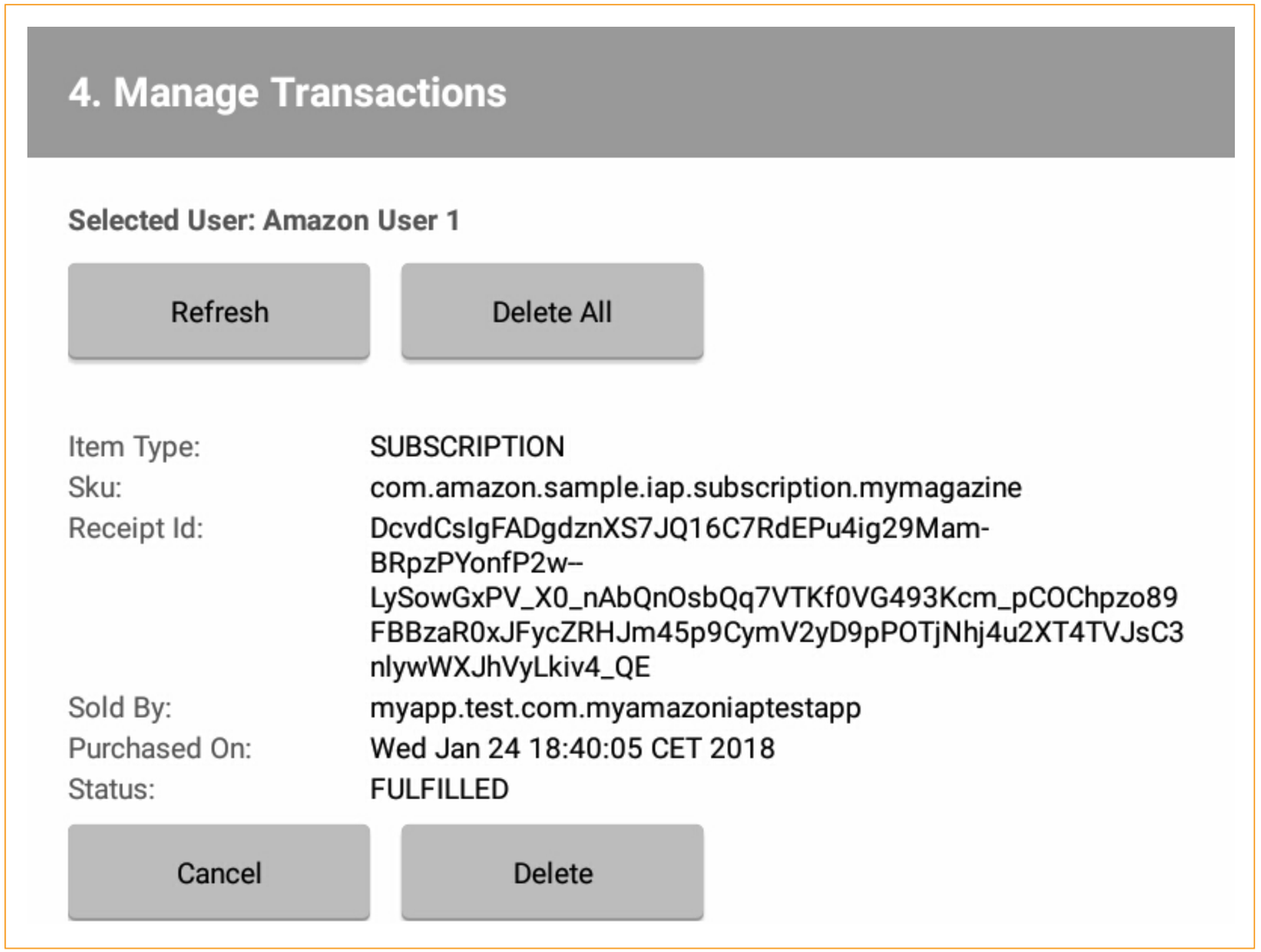
This is NOT the UI that the end-customer will see when the app is published on the Amazon Appstore. **This is a test UI** provided by the Amazon App Tester that allows us to test the transactions.



If we click on the dropdown menu, we can select from the **multiple subscription periods** we defined in the amazon.sdktester.json file. In this case, we have a $5 monthly subscription and a $12 quarterly subscription.



If we complete a purchase, you'll see that the **TextView will change** color and text to show we are subscribed to the magazine.



Now, if we exit our app and open the Amazon App Tester, click on "**In-App Purchasing API**" and "**4. Manage Transactions**," we can see the transaction details.

From here, we can even **cancel or delete** the transaction and this will be reflected in our app.

▷▷ Next Steps

Once you finish adding IAP to your Android app and complete testing, the next step would be to publish your app on the Amazon Appstore and start monetizing.

However, before launching the app in production, there are a few additional steps you might want to consider.

Test your app using Amazon Live App Testing

App testers experience the app and have access to the full suite of Amazon services—including In-App Purchasing—in the actual production environment, so you can ensure that your app is working as expected.

App testers are not charged for downloading your test app, and are not charged for in-app items. This tool allows you to improve quality, increase stability, and optimize the experience before you push your app live for all customers to download.

To learn more about Live App Testing, see here: https://developer.amazon.com/live-app-testing

▷▷ Next Steps

Once you finish adding IAP to your Android app and complete testing, the next step would be to publish your app on the Amazon Appstore and start monetizing.

However, before launching the app in production, there are a few additional steps you might want to consider.

Verify your customer receipts using Receipt Verification Service (RVS)

The Receipt Verification Service (RVS) enables validation of purchases made by your app's users from your own server.

This is very useful if you need to query Amazon about the current status of a specific receipt.

The following workflow describes the basic process for verifying purchase transactions:

1. A user completes an in-app purchase.
2. Get the receiptID returned to the onPurchaseResponse() callback.
3. Send the transaction's userId and receiptID from your app to your server.
4. Send a request from your server to RVS to verify the receipt.
5. Send the result of the RVS verification back to your app.
6. If the receipt is valid, fulfill the item. This means granting access to the in-app item for one-time use (consumable) or continued use (non-consumable). It may also require that you push the content to the user's device, such as for a subscription.

▷▷ Next Steps

Once you finish adding IAP to your Android app and complete testing, the next step would be to publish your app on the Amazon Appstore and start monetizing.

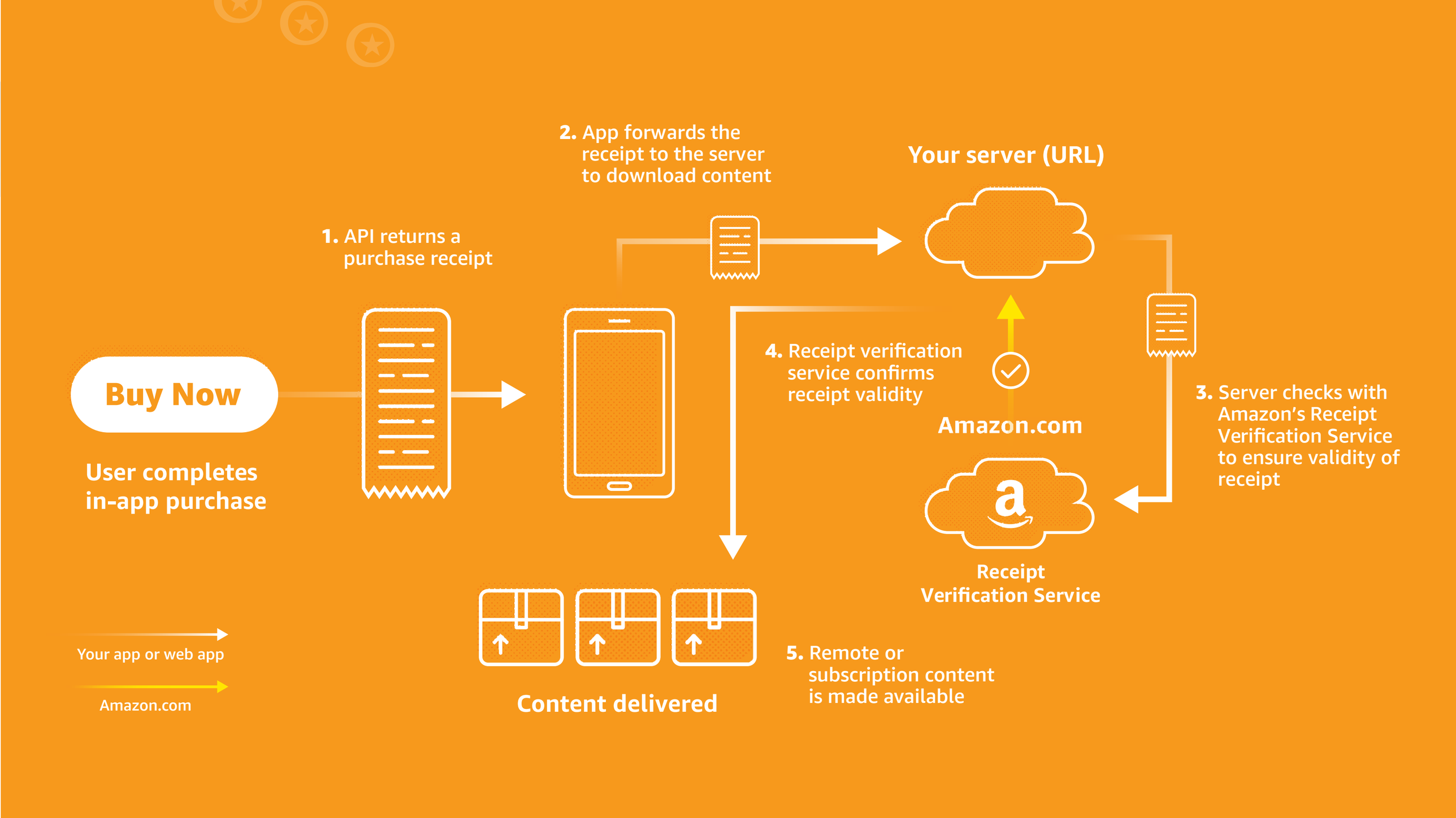
However, before launching the app in production, there are a few additional steps you might want to consider.

The following diagram illustrates the RVS workflow

You can also use RVS to **enable access** to a subscription purchased on another platform, like your website, as long as the purchase was made through Amazon. The following scenario describes this workflow:

1. A user purchases a subscription through Amazon via your company's website.
2. Your app receives a receipt for the purchased subscription.
3. To enable access, your app then sends information from the receipt to your server. Finally, your server validates this transaction by querying RVS.

To learn more about RVS, see the official complete documentation here: https://developer.amazon.com/docs/in-app-purchasing/iap-rvs-for-android-apps.html.


Buy Now
User completes in-app purchase
1. API returns a purchase receipt
2. App forwards the receipt to the server to download content
Your server (URL)
3. Server checks with Amazon's Receipt Verification Service to ensure validity of receipt
4. Receipt verification service confirms receipt validity
Amazon.com
Receipt Verification Service
5. Remote or subscription content is made available
Content delivered
Your app or web app
Amazon.com

Thank you

Thank you for following along and learning how to implement in-app purchasing to your Android app. I hope you found this how-to session helpful, and don't hesitate to reach out to me on Twitter with any questions.

Keep your eye out for more Amazon Appstore Developer CodeLabs.

— Mario Viviani, Technology Evangelist, Amazon (@mariuxtheone)

Free content to help intrepid app developers.

To acquire a free digital download of this book, visit the Developer Portal and search the title, “How to Add Amazon In-App Purchases and Subscriptions to Android Apps.”

Content by Amazon.com, Inc.

www.developer.amazon.com

Cover design, formatting, and editing by Graphiti.

The content of this book and the website links contained herein are simply for educational purposes, and are not intended to replace professional business advice or guarantee business outcomes. Every effort has been made to ensure that the content provided in this content is accurate and helpful for our readers at the time of publishing. However, this is not an exhaustive treatment of the subjects. No liability is assumed for losses or damages to the information provided. You are responsible for your own choices, actions, and results.

First edition
March 2018