SVILUPPIAMO UN'APPLICAZIONE MOBILE DI FILE HOSTING CON FLUTTER, AMPLIFY E AWS

Amazon S3 | Android Studio | AWS Amplify | AWS Identity and Access Management (IAM) | AWS Lambda | CI/CD | File hosting | Flutter | How-to | Mobile | Serverless

beSharp | 4 Settembre 2020

Al giorno d'oggi **Flutter** sta ottenendo sempre più riconoscimento come soluzione per lo sviluppo di applicazioni **mobile cross-platform**. Inoltre, **AWS Amplify** sta rapidamente guadagnando l'attenzione delle community di sviluppatori grazie all'incredibile semplicità con cui permette di configurare le applicazioni **senza preoccuparsi di gestire l'infrastruttura di backend**, gestendo in proprio l'intero processo.

Di recente i due framework hanno unito le loro forze ed è stata messa online la prima release di **Amplify per Flutter** a disposizione degli sviluppatori per i primi test con supporto a **Amazon Cognito**, **AWS S3** e **Pinpoint** per il logging.

Essendo ansiosi di metterci le mani sopra, abbiamo creato una Proof-of-Concept per verificare quanto bene queste librerie possano interagire tra di loro. Di seguito presentiamo quindi un tutorial su come sviluppare un'applicazione per **dispositivi mobili** simile a Dropbox, **supportata da AWS**, con la possibilità di gestire **l'autenticazione**.

Come bonus, proporremo anche una semplice configurazione per gestire la pipeline di CD/CI con **Travis CI**.

Il progetto può essere creato seguendo passo passo la nostra guida o scaricando direttamente la soluzione completa dal nostro **repository Github**.

Andiamo dunque a cominciare!

Configurare l’ambiente di progetto

Per cominciare a sviluppare applicazioni in Flutter, è necessario attuare alcuni step preliminari. Di seguito presenteremo le istruzioni per configurare l’ambiente di lavoro con tutti i tool necessari.

Aws Cli con account valido per utilizzare Amplify

Prima di poter effettuare altre operazioni, bisogna essere sicuri di aver creato un account AWS valido. Amplify necessita l’accesso alle seguenti risorse in cloud:

- AppSync
- API Gateway
- CloudFormation
- CloudFront
- Cognito
- DynamoDB
- IAM
- Lambda
- S3
- Amplify.

Per questo motivo ecco una semplice policy di AWS con un set di regole necessarie al funzionamento di Amplify:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "appsync:*",
                "apigateway:POST",
                "apigateway:DELETE",
                "apigateway:PATCH",
                "apigateway:PUT",
                "cloudformation:CreateStack",
                "cloudformation:CreateStackSet",
                "cloudformation:DeleteStack",
                "cloudformation:DeleteStackSet",
                "cloudformation:DescribeStackEvents",
                "cloudformation:DescribeStackResource",
                "cloudformation:DescribeStackResources",
                "cloudformation:DescribeStackSet",
                "cloudformation:DescribeStackSetOperation",
                "cloudformation:DescribeStacks",
```

```
                "cloudformation:UpdateStack",
                "cloudformation:UpdateStackSet",
                "cloudfront:CreateCloudFrontOriginAccessIdentity",
                "cloudfront:CreateDistribution",
                "cloudfront:DeleteCloudFrontOriginAccessIdentity",
                "cloudfront:DeleteDistribution",
                "cloudfront:GetCloudFrontOriginAccessIdentity",
                "cloudfront:GetCloudFrontOriginAccessIdentityConfig",
                "cloudfront:GetDistribution",
                "cloudfront:GetDistributionConfig",
                "cloudfront:TagResource",
                "cloudfront:UntagResource",
                "cloudfront:UpdateCloudFrontOriginAccessIdentity",
                "cloudfront:UpdateDistribution",
                "cognito-identity:CreateIdentityPool",
                "cognito-identity:DeleteIdentityPool",
                "cognito-identity:DescribeIdentity",
                "cognito-identity:DescribeIdentityPool",
                "cognito-identity:SetIdentityPoolRoles",
                "cognito-identity:UpdateIdentityPool",
                "cognito-idp:CreateUserPool",
                "cognito-idp:CreateUserPoolClient",
                "cognito-idp:DeleteUserPool",
                "cognito-idp:DeleteUserPoolClient",
                "cognito-idp:DescribeUserPool",
                "cognito-idp:UpdateUserPool",
                "cognito-idp:UpdateUserPoolClient",
                "dynamodb:CreateTable",
                "dynamodb:DeleteItem",
                "dynamodb:DeleteTable",
                "dynamodb:DescribeTable",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:UpdateTable",
                "iam:CreateRole",
                "iam:DeleteRole",
                "iam:DeleteRolePolicy",
                "iam:GetRole",
                "iam:GetUser",
                "iam:PassRole",
                "iam:PutRolePolicy",
                "iam:UpdateRole",
                "lambda:AddPermission",
                "lambda:CreateFunction",
                "lambda:DeleteFunction",
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration",
                "lambda:InvokeAsync",
                "lambda:InvokeFunction",
                "lambda:RemovePermission",
                "lambda:UpdateFunctionCode",
                "lambda:UpdateFunctionConfiguration",
                "s3:*",
                "amplify:*"
            ],
            "Resource": "*"
        }
    ]
}
```

La policy va agganciata ad un utente con **Accesso Programmatico**, prenderemo poi nota di **access e secret key**. Se non vi sentite sufficientemente sicuri a creare una policy AWS Policy da soli, potete seguire questa guida, altrimenti consigliamo di seguire una delle tante guide disponibili online.

Nota a margine: è altamente consigliato rimuove l'utente una volta che Amplify ha completato la creazione di tutte le risorse, per evitare di lasciare potenziali brecce di sicurezza.

Il prossimo step consiste nel configurare la AWS CLI sul proprio computer. Scarichiamola selezionando l'installer appropriato per il proprio Sistema Operativo. Una volta completata l'installazione accediamo ad un terminale a piacere e digitando: **aws configure**.

Usare l'**Access e la Secret key** salvate in precedenza quando richiesto dal prompt del terminale e lasciare il resto delle opzioni come consigliato.

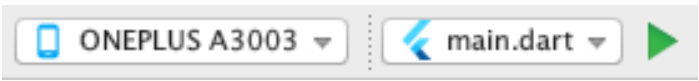
Una volta inserite tutte le informazioni richieste, procediamo a installare Flutter e Amplify correttamente.

Installare Flutter e configurare il primo progetto

Utilizzando questa guida siamo riusciti a installare ed eseguire Flutter senza particolari difficoltà. Una volta scelto il proprio Sistema Operativo (nel nostro caso MacOS) e seguite le istruzioni, possiamo configurare il tutto per funzionare con Android Studio, ma è possibile utilizzare anche IntelliJ. Dopo aver installato il framework, possiamo procedere a connettere un cellulare di test ed eseguire flutter devices in un terminale per verificare che quest'ultimo venga riconosciuto correttamente.

Creiamo un nuovo **Flutter project** dal menu di Android Studio e quindi **Flutter Application**. Una volta fornite le opzioni richieste l'IDE procederà a creare la struttura di progetto per noi.

Selezioniamo il nostro cellulare e il file **main.dart** quindi premiamo play come in figura:



Una volta eseguita la compilazione vedremo la demo funzionare sul dispositivo. Ora è possibile procedere configurando **AWS Amplify**.

Configurare Amplify per Flutter

Amplify ha da poco rilasciato una specifica versione della sua libreria per funzionare direttamente con Flutter, scritta in **Dart** e modulare, che permette quindi di scaricare ed eseguire soltanto i moduli necessari.

Abbiamo inizialmente seguito la guida ufficiale che Amplify ha predisposto per Flutter.

Questa guida aiuta a configurare Amplify e a completare tutti gli step preliminari (installare la libreria, configurare l'utente, ecc.) al fine di poter eseguire un ***amplify push*** nel terminale ed ottenere un file amplifyconfiguration.dart**.** Questo file è fondamentale in quanto contiene **tutte le**

informazioni sensibili e le configurazioni necessarie alla propria soluzione Amplify oltre a tutti i dati necessari a far funzionare i plugin di Amplify per la soluzione di Flutter. E' dunque fondamentale tenerlo sempre al sicuro!

Un paio di note circa il modulo Amplify Auth

Qualora decidessimo di installare Amplify con l'intenzione di utilizzare i metodi Cognito SignIn e SignUp raccomandiamo di prestare particolare attenzione ai seguenti dettagli per evitare situazioni non previste:

- Eseguire **amplify init** e **amplify configure** all'interno della **cartella del progetto flutter**. Se tutto risulterà eseguito correttamente, il file amplifyconfiguration.dart verrà creato all'interno della cartella **lib**.
- Assicurarsi di configurare sia il modulo **analytics** che **auth** con amplify add analytics e amplify add auth. Saranno necessari entrambi per garantire il corretto funzionamento della soluzione.
- Usare **flutter clean** nel terminale e installare sul proprio dispositivo un .apk pulito per garantire la lettura di tutti gli aggiornamenti fatti finora al progetto.

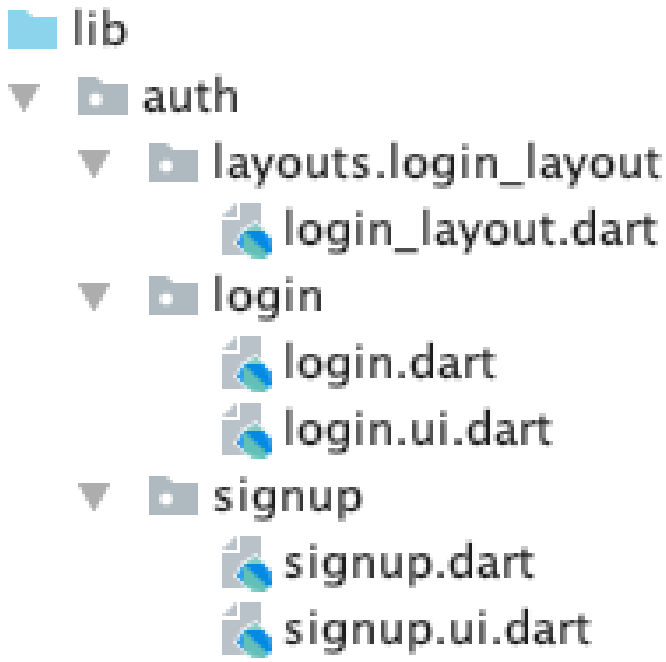
Arrivati a questo punto abbiamo svolto un check preliminare della demo di Amplify unita alla soluzione Flutter di default per essere sicuri che tutto funzionasse come dovuto. Dopo aver verificato, siamo potuti passare alla Proof-of-Concept vera e propria!

Creiamo il nostro clone mobile di dropbox!

Prima di ogni altra cosa: appena provati Flutter e Dart, ci siamo resi conto di come vi siano delle similarità con ReactNative, in particolare riguardo a come strutturare le applicazioni, il che significa praticamente quasi nessuna struttura 😅 .

Portiamo ordine in Flutter

Al fine di mantenere il progetto il più strutturato possibile, abbiamo deciso di scrivere il progetto con **Angular** in mente come riferimento: abbiamo optato per dividere il codice per funzionalità seguendo le raccomandazioni del **Domain Driven Design**. Abbiamo rimosso la parte di UI dai file dart dei widget principali, spostandola in classi specializzate per mantenere il codice più **DRY** e **Pulito.** A tal proposito vediamo un esempio semplificato:



Come in Angular, abbiamo definito un package per **Dominio**, in questo caso **Auth.** All'interno abbiamo creato un altro package per **un file di layout** e uno per ogni "**componente**" (login e signup). Per ogni componente, sono stati definiti 2 file: uno per la logica e uno, con suffisso **.ui,** contenente la grafica.

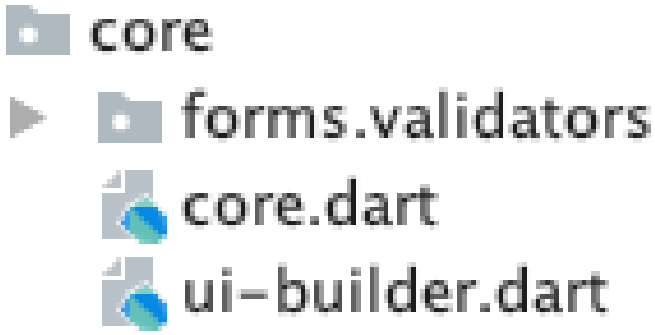
Diamo un'occhiata al codice; all'interno di **login.dart** troviamo:

```
// Call this when drawing component graphics
@override
Widget build(BuildContext context) {
 return loginUi(this);
}
```

Si può notare come non vi sia della grafica, ma al suo posto, viene chiamata la **classe loginUi** passando la classe **LoginState** di tipo **State<Login>** mediante la chiave **this**. Ora vediamo **login.ui.dart**:

```
UiBuilder loginUi = (state) => Container(
 child: Stack(
     children: [
        ...
     ]
 )
);
```

Il file adibito alla "grafica" è dichiarato come **UiBuilder<LoginState> loginUi = (state) => Container(child: ...** dove **UiBuilder** è un "helper" definito ad hoc (che vedremo a breve) e con tipo "forzato" a LoginState che passa la variabile **state** al metodo. Mediante questo approccio è possibile definire la grafica separatamente dalla logica, garantendo inoltre accesso a **tutte le variabili** dichiarate in **login.dart** all'interno della classe **LoginState**!
A tal proposito abbiamo definito anche un package **core** che contiene l'implementazione di UiBuilder:



Ora vediamo **ui-builder.dart**:

```
import 'package:flutter/material.dart';
typedef UiBuilder = Widget Function(T context);
```

Nulla più che la definizione di un **nuovo 'type'** (UiBuilder) facente riferimento ad una funzione che ritorna un oggetto **Widget** (per la grafica) con un **context** che dipende dal tipo generico **T**, in questo modo abbiamo un metodo pulito per separare grafica e logica!
Per completezza diamo uno sguardo al file **login_layout.dart**:

```
import 'package:flutter/material.dart';

class LoginLayout extends StatefulWidget {

 final Widget child;

 LoginLayout({ this.child, });

 }

class LoginLayoutState extends State {
 @override
 Widget build(BuildContext context) {
   return Container(
…
        appBar: null,
       body:
...
                    this.widget.child
...
   );
 }
}
```

In questo file i 2 elementi più importanti sono il **costruttore** LoginLayout({ **this.child**, }); dove si richiede una variabile **final** Widget **child**; (la classe **Login** in questi esempi), è quindi possibile iniettare la variabile nella proprietà **body** del layout grazie a **this.widget.child** per includere la grafica del widget. Infine, per poter mostrare questo componente all'interno di questo layout usiamo LoginLayout(child: Login()).

Struttura di Progetto

Abbiamo deciso di sviluppare una semplice applicazione in cui un utente si può registrare per ottenere accesso ad uno spazio personale su S3 per caricare e scaricare file. Per questo motivo abbiamo definito **2 domini**: Autenticazione e Gestione di S3.

Il primo contiene i componenti di **login** e **signup**, mentre il secondo dominio è adibito a gestire le azioni di **upload**, **download**, **delete** e **list** sul bucket dell'applicazione.

Creare i widget di Login e Signup widgets e connetterli con Cognito

Come primo passo, abbiamo deciso di sviluppare le feature di login e signup per permettere agli utenti di registrarsi e avere dunque accesso all'uso dell'applicazione. Come prerequisito, abbiamo seguito la guida di Amplify per analytics prima, e quella per l'autorizzazione dopo, come già descritto precedentemente.

Incominciamo con il lanciare il seguente comando nella cartella principale di progetto:

```
amplify add analytics
```

Utilizziamo i parametri di default. Una volta completato il comando, passiamo ad abilitare le funzionalità di login e signup con:

```
amplify add auth
```

Essendo il progetto una semplice POC, abbiamo deciso di lasciare i parametri di default come indicato di seguito:

```
? Do you want to use the default authentication and security configuration?
    `Default configuration`
? How do you want users to be able to sign in?
    `Username`
? Do you want to configure advanced settings?
    `No, I am done.`
```

Quindi abbiamo salvato la configurazione online utilizzando l'utente creato durante la fase preliminare con:

```
amplify push
```

In questa fase sarebbe buona cosa lanciare un **flutter clean** e **ricaricare** l'applicazione sul cellulare per essere sicuri che la configurazione venga aggiornata correttamente sul dispositivo.

Avere la configurazione caricata online significa aver creato un **bucket S3 con l'applicazione Amplify**, una **user pool,** e una **identity pool** su AWS Cognito. Tutte queste risorse possono essere

lasciate così come sono.

Ora concentrandoci sulla parte di codice, mostreremo le parti più interessanti, poichè l'intero progetto è disponibile sul nostro repository Github.

Login

Innanzitutto utilizziamo questa riga di codice per identificare il form di login:

```
final formKey = GlobalKey();
```

Questo passo è necessario per permettere la validazione dei campi del form con un'altra semplice riga di codice:

```
// Validate the form with this line...
if (formKey.currentState.validate()) {
```

Per accedere ai valori dei campi del form si usano oggetti di tipo **TextEditingController** come i seguenti:

```
final usernameController = TextEditingController();
final passwordController = TextEditingController();
```

Per effettuare il login dell'utente abbiamo invocato i metodi base di Amplify come da documentazione:

```
SignInResult res = await Amplify.Auth.signIn(
    username: usernameController.text.trim(),
    password: passwordController.text.trim(),
);
```

E per passare alle schermate di **signup** e di **s3** si può usare:

```
Navigator.push(context, MaterialPageRoute(builder: (context) => LoginLayout(child: Si
gnup())));
Navigator.push(context, MaterialPageRoute(builder: (context) => S3ViewerLayout(child:
S3Viewer())));
```

Come già descritto in precedenza abbiamo passato il **layout** corretto al **MaterialPageRoute** specificando il **componente** desiderato.
Inoltre per **notificare** l'**interfaccia** che lo stato dell'applicazione è cambiato, abbiamo utilizzato il metodo di Flutter **setState:**

```
setState(() {
    loggingIn = false;
});
```

Per la parte di UI, diamo un'occhiata al file **login.ui.dart**; la variabile **formKey** viene assegnata al form per identificarlo in modo univoco:

```
child: Form(
    key: state.formKey,
```

Per standardizzare la grafica all'interno dell'applicazione sono stati creati due componenti condivisi: **roundedTextFormField** e **roundedRectButton;** entrambi definiti all'interno del package **shared**. Giusto per curiosità diamo uno sguardo anche a questi:

```
import 'package:flutter/material.dart';

Widget roundedTextFormField(TextEditingController controller, String hintText, Color m
ainColor, Color backColor, Function validation, obscured) {
 return Padding(
     padding: EdgeInsets.only(bottom: 10, left: 50, right: 50),
     child: TextFormField(
         obscureText: obscured,
         controller: controller,
         validator: (value) => validation(value),
         style: TextStyle(color: mainColor),
         decoration: new InputDecoration(
           border: new OutlineInputBorder(borderRadius: BorderRadius.circular(100.0
),),
           filled: true,
           hintStyle: new TextStyle(color: mainColor.withOpacity(0.5)),
           hintText: hintText,
           fillColor: backColor
         ),
     )
 );
}
```

In **Dart** si ha un comportamento simile a Javascript nel senso che è possibile **definire un widget** e importarlo nel file dove sia necessario, proprio come nel caso presentato con **roundedTextFormField**.

Signup

Per quanto riguarda il processo di Signup, sono stati utilizzati i parametri di default di Amplify durante il wizard di configurazione: **email**, **username** (che verranno sfruttati per identificare lo spazio personale nel bucket S3) e **una password valida**. Dopo il processo di registrazione, avviene un **processo di conferma**. Una mail viene inviata all'utente contenente un codice di conferma da inserire nel programma per completare la registrazione.

Diamo un'occhiata anche al file **signup.dart.** Ci sono due metodi principali: **signupNewUser** e **confirmNewUser**; l'approccio è simile a quello visto per il login:

```
if (formKey.currentState.validate()) {
```

Abbiamo definito una variabile **formKey**, collegata poi al form della pagina, quindi si è usato Amplify per la registrazione:

```
// Create a map attributes dictionary for holding extra information for the user
Map userAttributes = {
    'email': emailController.text.trim(),
    // additional attributes as needed: we set email because is a common way
    // to define a unique value to use for S3 folders
};

// Signup using Amplify with Cognito
SignUpResult res = await Amplify.Auth.signUp(
    username: usernameController.text.trim(),
    password: passwordController.text.trim(),
    options: CognitoSignUpOptions(
        userAttributes: userAttributes
    )
);
```

Ancora una volta si può cambiare lo stato dell'applicazione con **setState**, alterando così la visibilità di diversi componenti all'interno del file **signup.ui.dart** mediante il **Visibility Widget**:

```
Visibility(
   visible: state.registering,
…
```

Per mantenere il codice più snello abbiamo usato **setState** anche in **signup.dart** per definire quale Widget rendere visibile: **signup** o **confirm**.

```
@override
Widget build(BuildContext context) {
 return this.isSignUpComplete ? confirmUi(this) : signupUi(this);
}
```

In pratica la variabile **isSignUpComplete** viene utilizzata per decidere quale widget mostrare (entrambi codificati nel file **signup.ui.dart**).

Per completare la registrazione utilizziamo:

```
SignUpResult res = await Amplify.Auth.confirmSignUp(
   username: usernameController.text.trim(),
   confirmationCode: confirmController.text.trim()
);
```

Infine l'utente viene inoltrato di nuovo alla pagina di login.

Creare il widget di gestione di S3 e connetterlo con il Bucket

Il manager di S3 è un widget con quattro funzioni principali:

- Listare il file online
- Caricare un nuovo file
- Scaricare un file online
- Cancellare un file online

Di seguito vedremo come è stato sviluppato il tutto.

```
// Storage Item list
List items = [];

// Check if the app is uploading something
bool isUploading = false;

// Check if a file is being removed
bool isRemoving = false;

// Check if a file is being downloaded
bool isDownloading = false;

// Check if the app is retrieving the list of files
bool isListing = false;
```

Cominciamo col definire un array contenente gli elementi da mostrare, e quattro variabili booleane per gestire i cambi di stato dell'applicazione relativi alle suddette quattro azioni: in questo modo è semplice utilizzare **setState** per bloccare eventuali interazioni dell'utente sui pulsanti durante l'esecuzione di task lunghi.
Durante l'avvio del componente viene lanciato il metodo **listFiles()**, che necessita di conoscere il nostro utente autorizzato:

```
AuthUser user = await Core.getUser();
```

E per listare tutti i file usiamo il metodo standard di Amplify senza opzioni:

```
ListResult res = await Amplify.Storage.list();
```

Prima di elencare i file, questi vengono filtrati per utente di modo da evitare di mostrare file di altri:

```
items = res.items.where((e) => e.key.split('/').first.contains(user.username)).toList
();
```

Per caricare un file, questo va recuperato dal cellulare e il modo più semplice per farlo è usare la libreria **File Picker** che gestisce da sola anche i permessi Android:

```
import 'package:file_picker/file_picker.dart';
```

File picker permette di recuperare un file e, grazie all'utente autorizzato, è possibile creare una chiave univoca per l'upload:

```
// We put this outside of try to avoid logging user cancel
File file = await FilePicker.getFile();
AuthUser user = await Core.getUser();

try {
 if(file.existsSync()) {

   setState(() {
     isUploading = true;
   });

   final key = user.username + '/' + file.path.split('/').last;

   // Upload the file
   UploadFileResult result = await Amplify.Storage.uploadFile(
     key: key,
     local: file
   );

...
```

Per rimuovere un file utilizziamo il metodo di Amplify passando la chiave identificativa:

```
RemoveResult res = await Amplify.Storage.remove(
   key: item.key,
);
```

Infine per effettuare il download di un file sono necessari due elementi; verificare di nuovo i permessi dell'utente perché le nuove versioni di android lo richiedono a runtime:

```
import 'package:permission_handler/permission_handler.dart';

Future checkPermission() async {
 final status = await Permission.storage.status;
 if (status != PermissionStatus.granted) {
   final result = await Permission.storage.request();
   if (result == PermissionStatus.granted) {
     return true;
   }
 } else {
   return true;
 }
 return false;
}
```

E chiaramente scaricare il file:

```
var dir = await DownloadsPathProvider.downloadsDirectory;
var url = await Amplify.Storage.getUrl(key: item.key, options: GetUrlOptions(expires:
3600));

await checkPermission();

final taskId = await FlutterDownloader.enqueue(
   url: url.url,
   fileName: item.key.split('/').last,
   savedDir: dir.path,
   showNotification: true, // show download progress in status bar (for Android)
   openFileFromNotification: true, // click on notification to open downloaded file (f
or Android)
);
```

Una nota a parte: abbiamo riscontrato alcune difficoltà a scaricare i file con il metodo apposito di download di Amplify. Siamo riusciti invece a completare l'operazione con una combinazione dei metodi **getUrl** di Amplify e di **FlutterDownloader.enqueue**.

Per la parte di UI si può vedere il file **s3viewer.ui.dart**; nulla di nuovo.

Come costruire il file di Core contenente le utility

Il componente di core contiene alcuni metodi **statici** che servono da utility per tutta l'applicazione ma anche i metodi di validazione per i form. Diamo un'occhiata ai seguenti esempi:

```
// Static method to get the current logged user
static Future getUser() async {
    return Amplify.Auth.getCurrentUser();
}
```

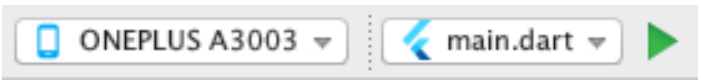
Quello sopra per ottenere un utente di Amplify e di seguito un semplice validatore per le email:

```
static emailValidator(value) {
 if (value.isEmpty) {
   return 'Please fill the field';
 }
 if (!RegExp(r"^[a-zA-Z0-9.a-zA-Z0-9.!#$%&'*+-/=?^_`{|}~]+@[a-zA-Z0-9]+\.[a-zA-Z]+").h
asMatch(value)) {
   return 'Please insert a valid email';
 }
 return null;
}
```

Quest'ultimo richiede una stringa e ritorna un messaggio di errore oppure **null** se l'input è valido; quest'ultima può essere passata ad un **TextFormField** come proprietà **validator** (si può fare riferimento alla soluzione su github per completezza).

Testare l'applicazione su un dispositivo reale

Per sviluppare l'applicazione su un dispositivo reale, Flutter permette di compilare e testare il codice su un cellulare con una funzionalità di **hot reload**. Ci sono due modi per fare questo:
1) Premere **play** nella parte in alto a destra di Android Studio come in figura:



2) Scrivere **flutter devices** nel terminale e prendere nota del **device id**. Quindi, sempre nel terminale, scrivere **flutter run -d <DEVICE_ID> -t ./lib/main.dart**, assicurandosi di essere nella cartella di progetto.

In modalità di test un **flag di debug apparirà automaticamente nella parte a destra della appBar**, e premendo **r** nel terminale è possibile forzare l'hot reload.

Per completare i preliminari necessari alla build bisogna eseguiamo questi due step:

1) Nella directory **android** accediamo a **app/src/build.gradle** e verifichiamo di avere questi valori per l' SDK:

```
minSdkVersion 23
targetSdkVersion 29
compiledSdkVersion 29
```

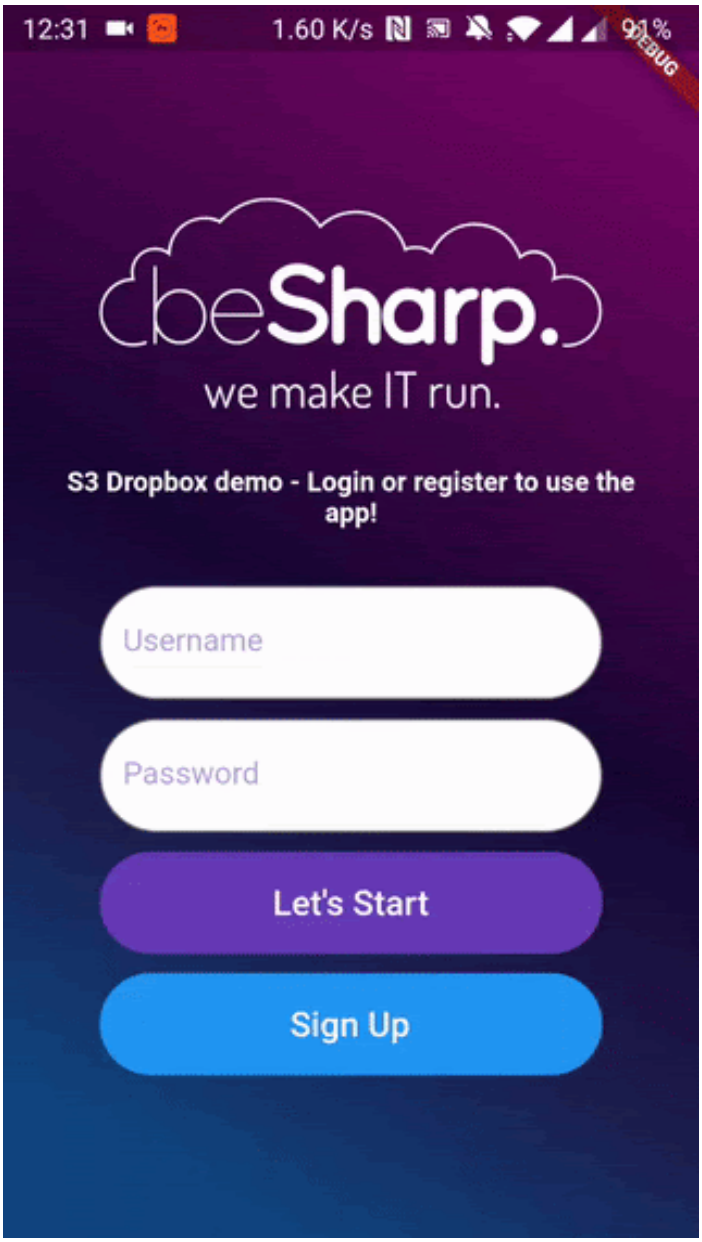
Questo passaggio serve ad evitare problemi di build con le librerie utilizzate per la demo.

2) Sempre nella directory **android** apriamo **app/src/main/AndroidManifest.xml** e aggiungiamo le seguenti permission:

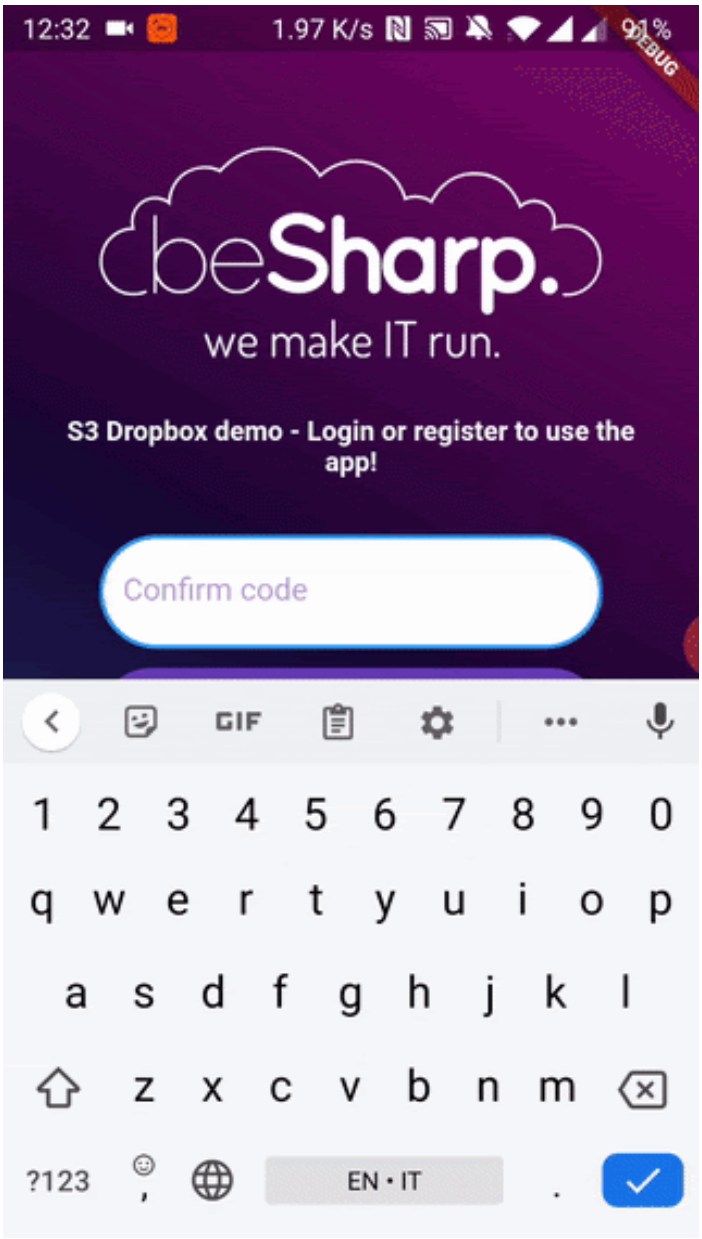
```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Direttamente all'interno del **tag manifest** e **android:requestLegacyExternalStorage="true"** come attributo del tag **application**.

Infine ecco l'applicazione in funzione:



Registrazione di un nuovo utente nell'applicazione



Login e inizio caricamento delle immagini

Caricamento e scaricamento dei file

Come usare Travis CI per la CI/CD

Travis CI è un servizio di CI/CD che permette di testare, generare e rilasciare un'applicazione, e può essere facilmente integrato con il proprio repository GitHub, in nel nostro caso uno pubblico. Per rimanere in tema con l'articolo, ci concentreremo su come generare un file APK per Android a partire dal codice rilasciato sul nostro repository GitHub mediante Travis CI.

Una volta effettuato l'accesso a Travis CI, nel nostro caso mediante account GitHub, dobbiamo attivare il nostro repository, per permettere ai **Git push events** di attivare la pipeline, che rispetterà le condizioni specifiche definite nel file di configurazione di Travis CI.

Dalla console di Travis CI, andiamo alla sezione Settings presente nel menù a tendina che compare cliccando sul proprio avatar. Ora, dal menu laterale di sinistra dobbiamo selezionare **l'organization** che contiene il repository che vogliamo integrare con Travis CI. Dalla lista di repository associati con

l'organization selezionata, cliccare sul bottone "toggle" che attiverà il repository desiderato per l'integrazione.

Una volta attivato, ci concentreremo sul file di configurazione di Travis CI, quello che descrive tutto il processo di CI/CD, nel nostro caso come costruire un file Android APK.

Creiamo nella cartella principale di progetto un file e nominiamolo **.travis.yml**.

All'inizio del file .travis.yml andremo a descrivere il sistema operativo sul quale la pipeline dovrà girare, il linguaggio supportato e la JDK necessaria alle operazioni di build.

```
language: android
os: linux
dist: xenial
jdk: oraclejdk8
```

Specificando **android** come linguaggio supportato dall'ambiente di CI/CD, quest'ultimo andrà ad installare implicitamente la Android SDK, necessario per completare il processo di build. **Xenial** è una Distro di Ubuntu che corrisponde alla versione 16.04. **oraclejdk8** va utilizzata come SDK per poter usare **sdkmanager**, un tool da riga di comando utilizzato per installare la piattaforma Android e la suite di tool di build. Dopo questo blocco di codice iniziale dobbiamo definire tutti i moduli principali di Android, necessari per le fasi successive del processo di build.

```
android:
 components:
   - tools
   - platform-tools
   - build-tools-28.0.3
   - android-29
```

Il terzo blocco principale riguarda l'installazione della piattaforma Android e degli strumenti di build, per finire anche l'installazione di Flutter all'interno dell'ambiente di CI/CD.

```
install:
 - yes | sdkmanager "platform-tools" "platforms;android-29" "build-tools;28.0.3"
 - git clone https://github.com/flutter/flutter.git -b flutter-1.22-candidate.9
```

Nota: l'opzione -b permette di specificare la versione esatta di Flutter che andremo a clonare nell'ambiente di CI/CD. Per avere una panoramica di tutti i branch di Flutter, possiamo visitare https://github.com/flutter/flutter/branches/all. Il branch Flutter-1.22-candidate.9 fa riferimento alla versione 1.22.0-9.0.pre.

Focalizziamoci ora sul blocco di script del file .travis.yml. Qui andremo a definire tutti i comandi necessari a generare la nostra APK Android.

```
script:
 - ./flutter/bin/flutter build apk --debug
```

Come possiamo notare, abbiamo utilizzato il flag **–debug**. Come descritto qui, in debug mode, l'applicazione viene generata per il debugging su dispositivi fisici, emulatori, o simulatori.

Una volta pronta, la nostra APK Android può essere rilasciata mediante una **GitHub release**. Per automatizzare il processo di rilascio, dobbiamo aggiungere questo blocco di configurazione.

```
deploy:
 provider: releases
 api_key: $API_KEY
 file: build/app/outputs/apk/debug/app-debug.apk
 skip_cleanup: true
 name: $TRAVIS_TAG
 on:
   tags: true
```

Questi comandi permettono a Travis CI di creare una nuova release solo se è presente un tag nell'ultimo commit eseguito. Usando la variabile di ambiente **TRAVIS_TAG**, siamo in grado di indicare il nome da utilizzare per la release.

Per aggiungere un file, dobbiamo specificare il percorso dopo la sua chiave.

L'ultimo elemento importante da definire è l'**api_key**, ad esempio il nostro **GitHub personal access token**, necessario all'autenticazione.

Alla fine del processo saremo in grado di vedere la nostra nuova release nel repository di GitHub.

Questo è il file .travis.yml completo:

```
language: android
os: linux
dist: xenial
jdk: oraclejdk8
android:
 components:
   - tools
   - platform-tools
   - build-tools-28.0.3
   - android-29
install:
 - yes | sdkmanager "platform-tools" "platforms;android-29" "build-tools;28.0.3"
 - git clone https://github.com/flutter/flutter.git -b flutter-1.22-candidate.9
script:
 - ./flutter/bin/flutter build apk --debug
deploy:
 provider: releases
 api_key: $API_KEY
 file: build/app/outputs/apk/debug/app-debug.apk
 skip_cleanup: true
 name: $TRAVIS_TAG
 on:
   tags: true
```

Referenze

Di seguito presentiamo tutte le referenze utili a poter creare questo progetto:

- https://flutter.dev/docs/get-started/install/macos
- https://docs.amplify.aws/start/getting-started/setup/q/integration/flutter
- https://docs.amplify.aws/lib/auth/getting-started/q/platform/flutter#configure-auth-category
- https://aws.amazon.com/cli/
- https://dart.dev/docs
- https://github.com/git-touch/file-icon
- https://pub.dev/packages/file_picker
- https://pub.dev/packages/fluttertoast
- https://github.com/fluttercommunity/flutter_downloader
- https://pub.dev/packages/permission_handler

E di seguito il nostro repository per scaricare e testare l'applicazione di demo:

- https://github.com/besharpsrl/flutter-amplify-demo

Nota: al momento della scrittura di questo articolo Amplify per Flutter non compila in release mode, di conseguenza l'applicazione può essere provata solo in debug mode.

Conclusioni

In questo articolo abbiamo visto come sia semplice creare un'applicazione mobile con Flutter utilizzando Amplify per gestire tutta la complessità di creare un'infrastruttura AWS e mantenere i dati sincronizzati tra l'applicazione e il Cloud. Al momento dell'articolo solo 3 servizi sono stati resi compatibili con Flutter: Cognito, S3 e Pinpoint, ma vedendo come questi funzionino già molto bene, la speranza è di poter vedere ancora molti update a questa libreria.

Come nota di chiosa, il lettore si senta libero di fare riferimento alla nostra semplice demo per espandere ed esplorare tutte le possibilità di questa soluzione. Nel caso si abbiano domande o spunti, incoraggiamo a contattarci per discuterne insieme.

Fino al prossimo articolo, grazie per il tempo dedicato alla lettura e alla prossima! 😉

#Proud2beCloud

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it