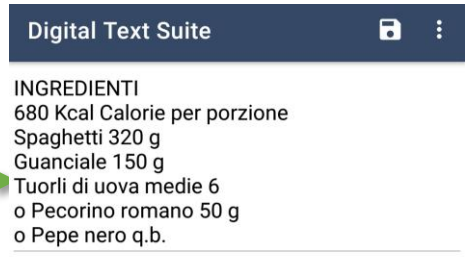
Digital Text Suite

Gruppo 30Bit+

Digitalizzazione di una foto: presentazione caso d'uso

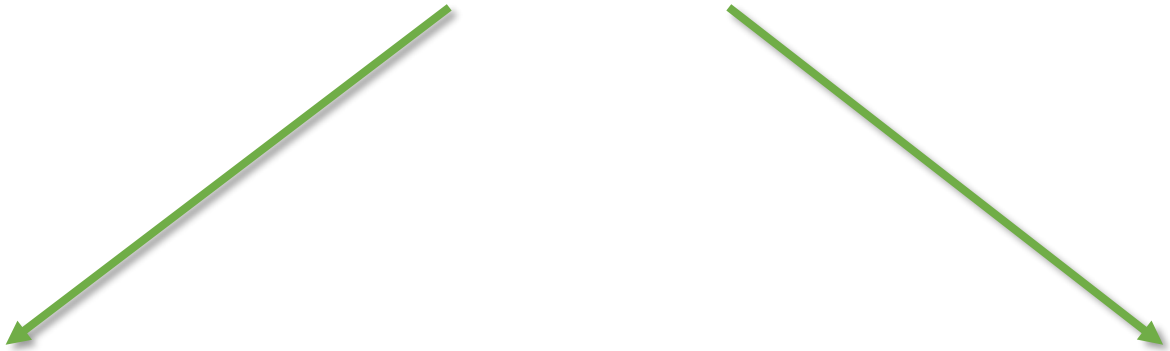
Da un'immagine

Al testo in essa contenuto



Setup

-avvio: disattivazione della modalità notte, gestione sottocartelle di archiviazione per vari casi d'uso, le modalità d'avvio



Normale
(toccando l'icona dell'app)

Condividendo un'immagine (o più) con l'app

Condividendo le immagini con l'app - 1

```
val imageUri = intent.getParcelableExtra<Parcelable>(Intent.EXTRA_STREAM) as? Uri
if (imageUri != null) {
    val intent = Intent(this, RealMainActivity::class.java)
    intent.putExtra("image", imageUri)
    startActivity(intent)
```

Dalla MainActivity: Se condividiamo una sola immagine

Condividendo le immagini con l'app - 2

```
else {
    val listMultiple =
intent.getParcelableArrayListExtra<Parcelable>(Intent.EXTRA_STREA
M)
    if(listMultiple!=null) {
        val listUri = listMultiple.filterIsInstance<Uri>()
        val arrayList = ArrayList<Uri>()
        for(uri in listUri){
            arrayList.add(uri)
        }
        val intent = Intent(this,RealMainActivity::class.java)
        intent.putParcelableArrayListExtra("images",arrayList)
        startActivity(intent)
    }
```

Se ne condividiamo più di una
In tutti e due i casi: intent a RealMainActivity
(vedi di seguito)

Condividendo le immagini con l'app - 3

```
private val REQUIRED_PERMISSIONS = arrayOf(Manifest.permission.CAMERA,
Manifest.permission.READ_EXTERNAL_STORAGE,
Manifest.permission.WRITE_EXTERNAL_STORAGE,
Manifest.permission.INTERNET)
.
.
if(allPermissionsGranted()){
    init()
}else{
    ActivityCompat.requestPermissions(this, REQUIRED_PERMISSIONS,
REQUEST_CODE_PERMISSIONS)
}
```

Nella onCreate: controllo su permessi (per altri casi d'uso)

Condividendo le immagini con l'app - 4

```
val fileUri = intent.getParcelableExtra<Uri>("image")
if(fileUri!=null){
    RecognizerUtil(this).recognize(fileUri,false)
}
else{
    val listUri = intent.getParcelableArrayListExtra<Uri>("images")
    if(listUri!=null){
        RecognizerUtil(this).recognizeAll(listUri,false)
    }
}
```

Successivamente, controlliamo se
effettivamente sono state passate immagini:
Supponiamo di si->La RecognizerUtil analizzerà le immagini

Condividendo le immagini con l'app - 5

```
fun recognize . .
.
val stream =
context.contentResolver.openInputStream(fileUri)
val bitmap = BitmapFactory.decodeStream(stream)
stream!!.close()
val image = InputImage.fromBitmap(bitmap,0)
val recognizer =
TextRecognition.getClient(TextRecognizerOptions.
DEFAULT_OPTIONS)
```

Se riceviamo una sola immagine:
Dall'Uri ricaviamo prima la bitmap dell'immagine,
poi l'oggetto InputImage su cui usare il recognizer
(uso della libreria MLKit)

Condividendo le immagini con l'app - 6

```
recognizer.process(image).addOnSuccessListener { text ->
            if(text.text.isNotEmpty()) {
                val intent = Intent(context, TextResultActivity::class.java)
                val language = text.textBlocks[0].recognizedLanguage
                val note = Note(text.text,"","",
                language,System.currentTimeMillis(),false)
                intent.putExtra("result",note)
                intent.putExtra("type", TextResultType.NOT_SAVED.ordinal)
                if(temp)
                    fileUri.toFile().delete()
                    context.startActivity(intent)
            }
```

Intent all' Activity che mostrerà il testo contenuto nell'immagine

Creazione oggetto Nota: servirà poi per un' eventuale inserimento in DB

Se l'immagine è temporanea allora la cancelliamo (dipende dal caso d'uso)

Aggiunta la nota e il suo tipo all'intent

Funzionamento a blocchi di testo

Condividendo le immagini con l'app - 7

```
else{
    CoroutineScope(Dispatchers.Main).launch{
        Toast.makeText(context,
context.getString(R.string.empty_text),
Toast.LENGTH_LONG).show()
        if(temp)
            fileUri.toFile().delete()
    }
}
```

In caso di testo vuoto oppure di errore nell' elaborazione, lo notifichiamo all'utente senza fare altro

```
.addOnFailureListener{
    CoroutineScope(Dispatchers.Main).launch{
        Toast.makeText(context,context.getString
        (R.string.error_recognition),
        Toast.LENGTH_LONG).show()
        if(temp)
            fileUri.toFile().delete()
    }
}
```

Condividendo le immagini con l'app - 8

```
private fun recognizeRecursive(listUri: List<Uri>, temp: Boolean, count: Int){
        .
        .
        recognizer.process(image).addOnSuccessListener {
            text ->
            if(text.text.isNotEmpty()){
                result.append(text.text)
                result.append("\n")
                if(count == listUri.size-1){
                    .
                    .
                    context.startActivity(intent)
                }
                else{
                    if(temp)
                        listUri[count].toFile().delete()
                    recognizeRecursive(listUri,temp,count+1)
                }
            }
        }.addOnFailureListener{
            .
            .
            }
        }
    }
}
```

Se riceviamo più immagini: quasi stessa funzione, ma ricorsiva, aggiungendo man mano le note ricavate da ogni singola immagine

Avviando regolarmente - 1

```
else {
    val sharedPref = getPreferences(Context.MODE_PRIVATE)
    val isFirst = sharedPref.getBoolean("isFirst",true)
    if(isFirst){
        val intent = Intent(this,TutorialActivity::class.java)
        sharedPref.edit().putBoolean("isFirst",false).apply()
        startActivity(intent)
    }
    else {
        Handler(Looper.getMainLooper()).postDelayed({
            val intent = Intent(this, RealMainActivity::class.java)
            startActivity(intent)
            finish()
        }, SPLASH_SCREEN_DURATION)
    }
}
```

Se è la prima volta che avviamo l'app: tutorial

Altrimenti: splash screen di 1,5 secondi

Se avviamo regolarmente, ovviamente verrà eseguito l'altro ramo nella Main activity

Avviando regolarmente - 2

```
override fun onPageSelected(position: Int) {
    super.onPageSelected(position)
    requestedOrientation = if(position==2 || position==3){
        ActivityInfo.SCREEN_ORIENTATION_PORTRAIT
    } else{
        ActivityInfo.SCREEN_ORIENTATION_FULL_SENSOR
    }
}
```

Dettagli sul tab layout: controllo pagina selezionata. Problema grafico con la fotocamera.

Avviando regolarmente - 3

Dettagli su view pager: caricamento dei fragment

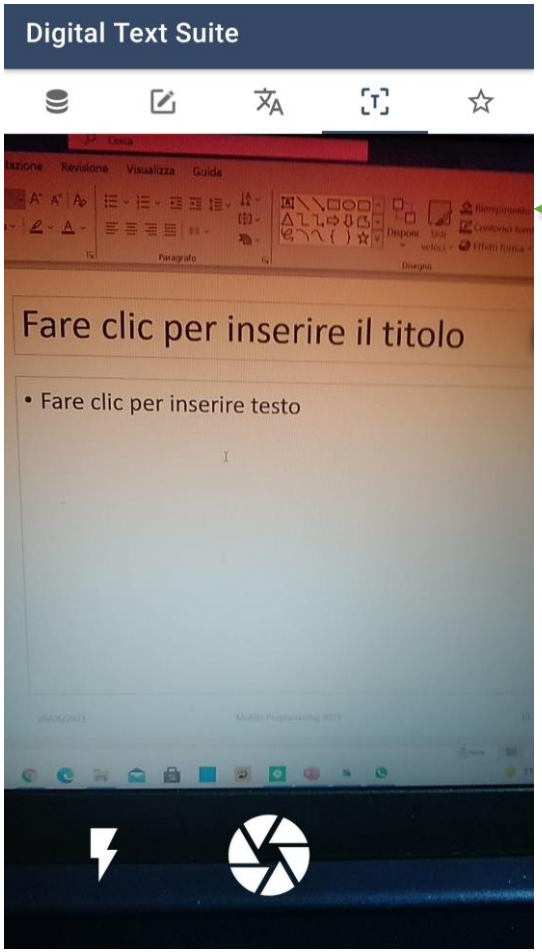
Cliccando su un tab: vengono inizializzati (onCreate, onViewCreated…) tutti i fragment dal primo a quello successivo rispetto al selezionato, poi: onResume su quello interessato

Nostro problema: uso di requireContext() che può generare un'eccezione (faceva crashare l'app)

Avviando regolarmente - 4

La RealMainActivity, in questo caso, non chiamerà ancora nessun metodo di RecognizerUtil.
Cosa possiamo fare?
Clicchiamo qui:

Avviando regolarmente - 5



Il fragment che ci si presenterà.
Consente di scattare una foto, da cui poi verrà estratto il testo.
Come funziona?
Fragment Astratto

Avviando regolarmente - 6

```
abstract class CameraFragment : Fragment(), CaptureHandler {
    protected var cameraExecutor: ExecutorService =
    Executors.newSingleThreadExecutor()
    protected var imageAnalysis: ImageAnalysis? = null
    protected var lensFacing = CameraSelector.DEFAULT_BACK_CAMERA
    protected lateinit var captureHandler: CaptureHandler
    protected var imageCapture : ImageCapture? = null
    protected lateinit var preview : Preview
    protected lateinit var cameraProvider: ProcessCameraProvider
    protected var camera : Camera? = null
```

Executor: oggetto diverso dalle coroutine che permette la programmazione concorrente, vive finché non viene ucciso da programma. Il funzionamento di CameraX prevede il loro utilizzo.

Le altre varie variabili: riguardano CameraX, per esempio la preview o l'oggetto che gestisce le impostazioni di cattura (es. flash)

CaptureHandler: interfaccia che le classi concrete devono implementare, riuso (vedi in seguito)

Avviando regolarmente - 7

```
override fun onAttach(context: Context) {
    if(context is CaptureHandler){
        captureHandler = context
    }
    else
        captureHandler = this
    super.onAttach(context)
}
```

Quando ci sarà il binding tra fragment e ciclo di vita.

Distruzione dell'executor

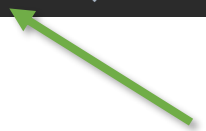
```
override fun onDestroy() {
    super.onDestroy()
    cameraExecutor.shutdown()
}
```

```
fun setFlash(mode: Int){
    imageCapture!!.flashMode=mode
}
```

Impostazione modalità flash

Avviando regolarmente - 8

```
fun takePhoto(save: Boolean){
    val outputDirectory = if(save)
    //store the image
        requireActivity().getExternalFilesDir
        (Environment.DIRECTORY_DCIM+File.separator+"effectscam")
    else
    //cache the image
        requireContext().cacheDir
    val filename = "Image_" + System.currentTimeMillis()
    val extension = ".jpg"
    val photoFile = if(save)
        File(outputDirectory, filename+extension)
    else
        File.createTempFile(filename,extension,outputDirectory)
```



L'immagine è da salvare in memoria o meno?

Avviando regolarmente - 9

```
val outputFileOptions: ImageCapture.OutputFileOptions
if(lensFacing== CameraSelector.DEFAULT_FRONT_CAMERA){
    val metadata = ImageCapture.Metadata()
    //Reverse image
    metadata.isReversedHorizontal=true
    outputFileOptions = ImageCapture.OutputFileOptions.
    Builder(photoFile).setMetadata(metadata).build()
}
else
    outputFileOptions = ImageCapture.OutputFileOptions.Builder(photoFile).build()
```

Impostazioni per la cattura dell'immagine: riuso dall'esonero

Avviando regolarmente - 10

```
imageCapture!!.takePicture(outputFileOptions, cameraExecutor,
  object : ImageCapture.OnImageSavedCallback {
    override fun onImageSaved(outputFileResults: ImageCapture.OutputFileResults) {
        captureHandler.saveImage(Uri.fromFile(photoFile),!save)
    }

    override fun onError(exception: ImageCaptureException) {
        captureHandler.errorHandler(exception)
    }
})
```

Metodo messo a disposizione delle API di cameraX.
Parametri e funzioni sovrascritte.

Avviando regolarmente - 11

Fragment concreto di questo caso d'uso: FragmentCameraDigitalize

```
private lateinit var mediaPlayer: MediaPlayer
private var flashMode = ImageCapture.FLASH_MODE_ON
```

Effetto audio

Modalità flash

Le funzioni dell'interfaccia CaptureHandler implementate: se si cattura l'immagine, verrà inviata all'oggetto RecognizerUtil

```
override fun saveImage(fileUri: Uri, temp: Boolean) {
    RecognizerUtil(requireContext()).recognize(fileUri,true)
}

override fun errorHandler(exception: ImageCaptureException) {
    CoroutineScope(Dispatchers.Main).launch{
        Toast.makeText(requireContext(),
        getString(R.string.error_camera),
        Toast.LENGTH_LONG).show()
    }
}
```

Avviando regolarmente - 12

```
. . .
    object : OrientationEventListener(requireContext()) {
        override fun onOrientationChanged(orientation: Int) {
            if (orientation == UNKNOWN_ORIENTATION) {
                return
            }

            val rotation = when (orientation) {
                in 45 until 135 -> Surface.ROTATION_270
                in 135 until 225 -> Surface.ROTATION_180
                in 225 until 315 -> Surface.ROTATION_90
                else -> Surface.ROTATION_0
            }
            if(imageCapture!=null)
                imageCapture!!.targetRotation = rotation
        }
    }
}
```

Gestione orientamento immagine per non avere problemi con l'analizzatore dopo aver catturato l'immagine: come funziona?

Avviando regolarmente - 13

```
mediaPlayer = MediaPlayer.create(requireContext(), R.raw.camera_shutter_click)
binding.imgBtnCamera.setOnClickListener{
    takePhoto(false)

    mediaPlayer.start()

    val zoomOut = AnimationUtils.loadAnimation(requireContext(),
      R.anim.zoom_out)
    zoomOut.setAnimationListener(ZoomAnimationListener
      (binding.imgBtnCamera,requireContext()))

    binding.imgBtnCamera.startAnimation(zoomOut)
}
```

Stessa cosa per l'animazione

L'effetto audio viene preso dai file in resources e viene fatto partire

Avviando regolarmente - 14

```
binding.imgBtnFlash.setOnClickListener{
    when(flashMode){
        ImageCapture.FLASH_MODE_ON -> {
            flashMode = ImageCapture.FLASH_MODE_AUTO
        }
        ImageCapture.FLASH_MODE_AUTO -> {
            flashMode = ImageCapture.FLASH_MODE_OFF
        }
        ImageCapture.FLASH_MODE_OFF -> {
            flashMode = ImageCapture.FLASH_MODE_ON
        }
    }
    setIconFlash()
    setFlash(flashMode)
    val sharedPreferences = requireActivity().
    getPreferences(Context.MODE_PRIVATE)
    val edit = sharedPreferences.edit()
    edit.putInt("flash",flashMode)
    edit.apply()
}
```

Variazione modalità flash

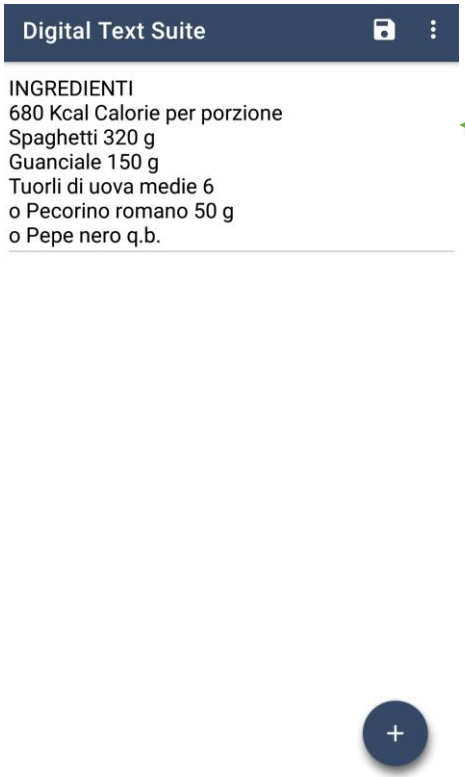
Ricordiamo inoltre la modalità scelta per la prossima volta

Avviando regolarmente - 15

```
private fun setIconFlash(){
    binding.imgBtnFlash.setImageDrawable(when(flashMode){
        ImageCapture.FLASH_MODE_ON ->
            ContextCompat.getDrawable(requireContext(),
            R.drawable.flash_on_48)
        ImageCapture.FLASH_MODE_OFF ->
            ContextCompat.getDrawable(requireContext(),
            R.drawable.flash_off_48)
        else -> ContextCompat.getDrawable(requireContext(),
            R.drawable.flash_auto_48)
    })
}
```

Ovviamente in base alla configurazione del flash, cambiamo anche l'icona

Risultato finale - 1



Schermata generale, vari elementi

Interazioni: elaborazione della nota, previo salvataggio

Risultato finale - 2

```
R.id.it_editable->{
    if(type==TextResultType.NOT_SAVED){
        Toast.makeText(this,getString(R.string.not_saved_edit),
        Toast.LENGTH_LONG).show()
    }
    else {
        .
        .
        if (editable) {
            type = TextResultType.EDITABLE
            binding.editTextTextMultiLine.isEnabled = true
            setType(menu)
        } else {
            if (originalText.equals(textResult)) {
                type = TextResultType.SAVED
                setType(menu)
            }
            binding.editTextTextMultiLine.isEnabled = false
        }
    }
}
```

Se vogliamo editare dobbiamo prima salvare il testo

Abilitiamo o disabilitiamo la possibilità di modificare il testo digitalizzato…

Risultato finale - 3

```
R.id.it_undo -> {
    binding.editTextTextMultiLine.text.clear()
    binding.editTextTextMultiLine.text.append(originalText)
    textResult=originalText
}
```

Possibilità di
tornare indietro
con le modifiche

Risultato finale - 4

```
R.id.it_delete -> {
    val alert = AlertDialog.Builder(this)
    alert.setTitle(R.string.alert_delete_title)
    alert.setMessage(R.string.alert_delete_message)
    alert.setPositiveButton(R.string.yes)
    { dialogInterface: DialogInterface, _:
      Int ->
        CoroutineScope(Dispatchers.IO).launch {
        dao.deleteNote(note)
            CoroutineScope(Dispatchers.Main).launch {
                dialogInterface.cancel()
                finish()
            }
    }
    }
    alert.setNegativeButton(R.string.no) { dialogInterface: DialogInterface, _:
        Int ->
        dialogInterface.cancel()
    }
    alert.show()
```

Possibilità di cancellare del tutto la nota: alert

Risultato finale - 5

```
R.id.it_copy -> {
    val clipboard = getSystemService(Context.CLIPBOARD_SERVICE) as ClipboardManager
    val clip = ClipData.newPlainText("note",textResult)
    clipboard.setPrimaryClip(clip)
    Toast.makeText(this,getString(R.string.copied),Toast.LENGTH_SHORT).show()
}
```

Possibilità di copiare la nota per intero

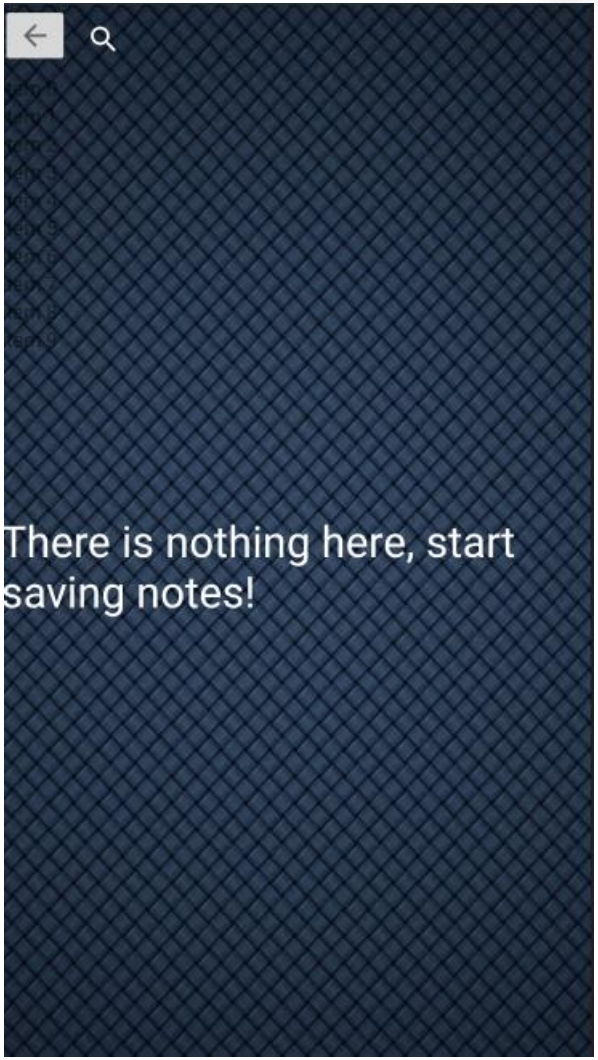
Extra: animazione tasti

FragmentAllFiles.kt

- All’interno di questo fragment vengono mostrate le directories e i files creati/salvati dall’utente
- Esistono due diverse modalità di visualizzazione

Fragment_all_files.xml

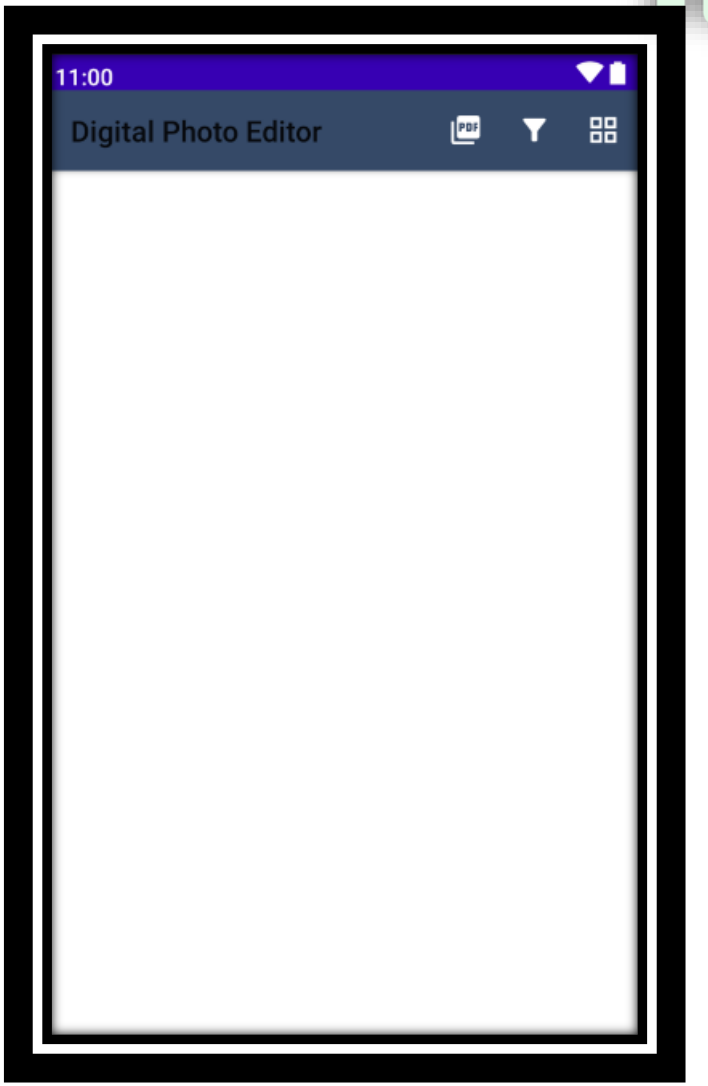
- La recycler view mostra le directories oppure i files all'interno della directory selezionata dall'utente.
- L' ImageButton viene mostrato all'utente solamente se si stanno mostrando i files nella directory, altrimenti è invisibile.
- La textView è visibile solamente se non sono presenti directories oppure non sono presenti files all'interno della directory selezionata.



Fragment_all_files.xml (land)

Menu_all_files.xml

- Il menu è strutturato in tre item

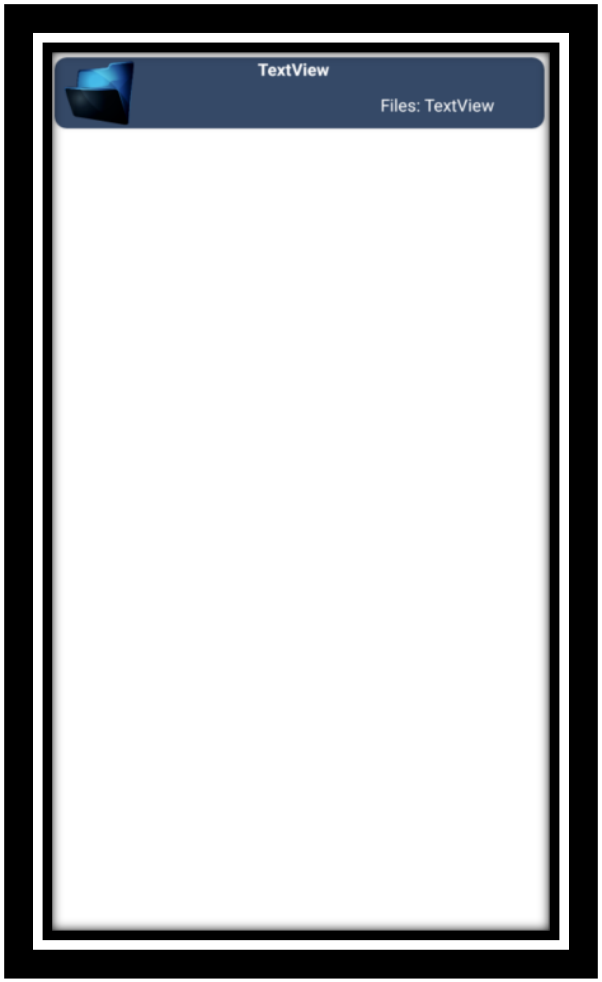


Item_directory_big.xml

- Il layout rappresenta il folder mostrato all'utente se viene scelta la modalità di visualizzazione DIR_BIG

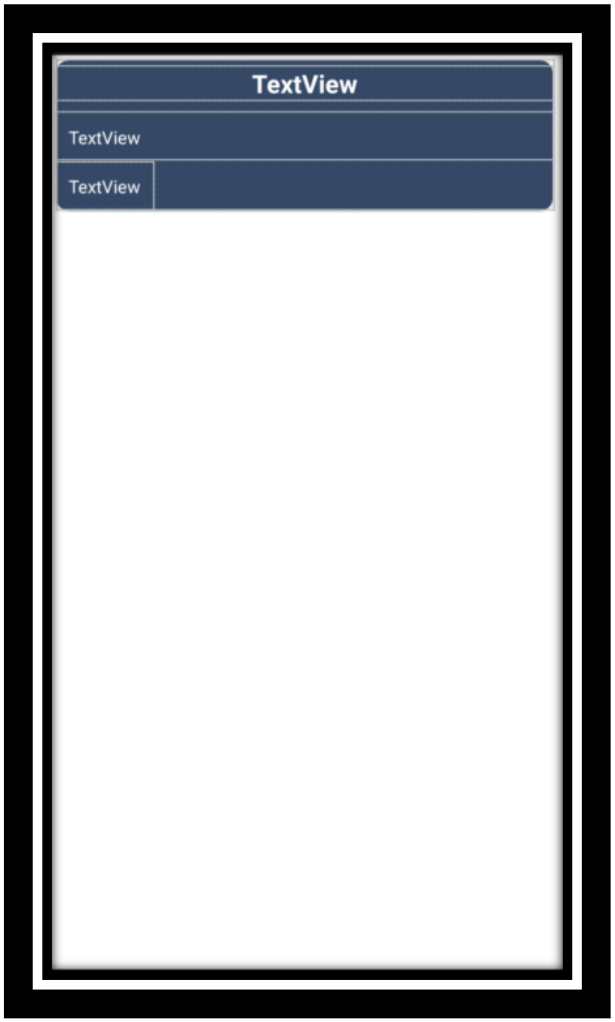
Item_directory_small.xml

- Il layout rappresenta il folder mostrato all'utente se viene scelta la modalità di visualizzazione DIR_SMALL



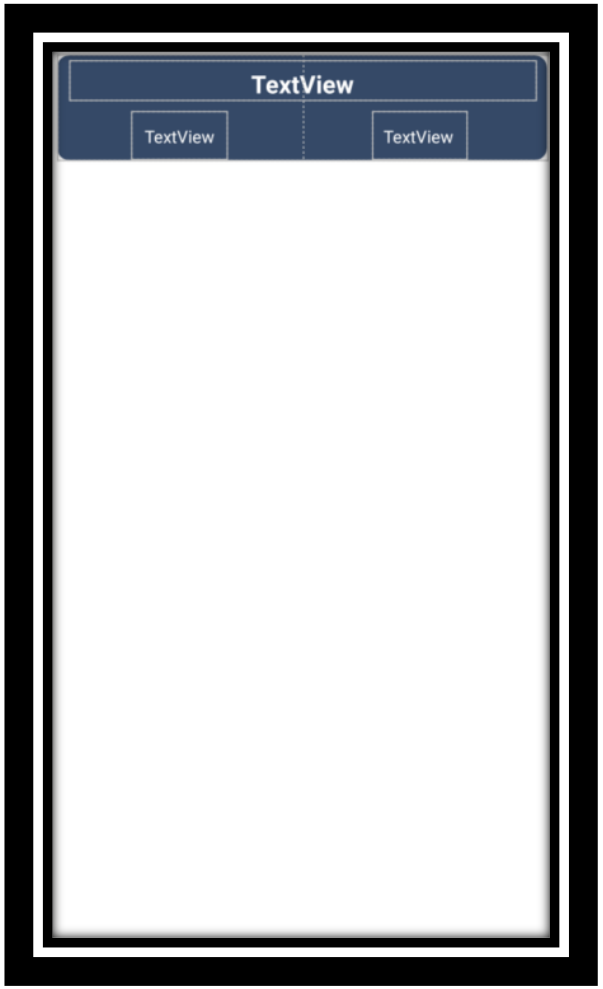
Item_file_big.xml

- Il layout rappresenta come vengono mostrati i files all'interno della directory se viene scelta la modalità di visualizzazione BIG



Item_file_small.xml

- Il layout rappresenta come vengono mostrati i files all'interno della directory se viene scelta la modalità di visualizzazione SMALL



Settaggio degli Observer

```
private fun setLiveData(){
    val directoryObserver = Observer<List<Directory>>(){
        setUI()
        setAdapterDirectory(it)
    }
    val listObserver = Observer<List<Note>>(){
        setUI()
        setAdapterFiles(it)
    }
    viewModel.listDirectory.observe(requireActivity(),directoryObserver)
    viewModel.listNotes.observe(requireActivity(),listObserver)
}
```

Adapters

- Sono stati implementati quattro diversi adapter: due adapter per quanto riguarda i files e due adapter per le directories
- Il motivo di ciò è perché esistono quattro modalità di visualizzazione, due per i files e due per le directories

setAdapterDirectory ()

```
private fun setAdapterDirectory(listDirectory: List<Directory>){
    binding.rvFiles.invalidate()
    Log.d("DIR_SIZE",listDirectory.size.toString())
    if(showMode == FilesShowMode.DIR_BIG){
        adapterDirectoryBig = DirectoryBigAdapter(listDirectory, requireContext(),this,spanCount)
        binding.rvFiles.adapter = adapterDirectoryBig
    }
    else if (showMode == FilesShowMode.DIR_SMALL){
        adapterDirectorySmall = DirectorySmallAdapter(listDirectory,requireContext(),this)
        binding.rvFiles.adapter = adapterDirectorySmall
    }
}
```

setAdapterFiles ()

```
private fun setAdapterFiles(listFiles: List<Note>){
    if(showMode == FilesShowMode.BIG){
        adapter = AllFilesBigAdapter(listFiles,requireContext(),this)
        binding.rvFiles.adapter=adapter
    }
    else{
        adapterSmall = AllFilesSmallAdapter(listFiles,requireContext(),this,this)
        binding.rvFiles.adapter=adapterSmall
    }
}
```

setUI()

```
val note = Note()
note.id = -1
viewModel.selectedNote.value = note
binding.rvFiles.invalidate()
binding.rvFiles.invalidateItemDecorations()
binding.rvFiles.adapter = null
binding.rvFiles.layoutManager = null
val layoutManager: RecyclerView.LayoutManager
for(i in 0 until binding.rvFiles.itemDecorationCount){
    binding.rvFiles.removeItemDecorationAt(i)
}
```

setUI() (2)

```
val orientation = requireContext().resources.configuration.orientation
if(orientation == Configuration.ORIENTATION_LANDSCAPE && showMode == FilesShowMode.SMALL){
    binding.previewLayout!!.isVisible = true
    binding.guidelineAllFiles!!.setGuidelinePercent(0.4f)
   childFragmentManager.beginTransaction()
        .replace(binding.previewLayout!!.id, FragmentNoteDetails(), DETAILS_FRAGMENT_TAG )
        .commit()
}
else{
    if(orientation == Configuration.ORIENTATION_LANDSCAPE){
        binding.previewLayout!!.isVisible = false
        binding.guidelineAllFiles!!.setGuidelinePercent(1f)
    }
}
```

setUI() (3)

```
when(showMode) {
    FilesShowMode.BIG -> {
        spanCount = if(orientation == Configuration.ORIENTATION_LANDSCAPE)
            4
        else
            2
        layoutManager =
            StaggeredGridLayoutManager(spanCount, StaggeredGridLayoutManager.VERTICAL)
        (layoutManager as StaggeredGridLayoutManager).gapStrategy=StaggeredGridLayoutManager.GAP_HANDLING_NONE
    }
    FilesShowMode.DIR_BIG -> {
        spanCount =  if(orientation == Configuration.ORIENTATION_LANDSCAPE)
            5
        else
            3
        layoutManager =
            StaggeredGridLayoutManager(spanCount, StaggeredGridLayoutManager.VERTICAL)
        (layoutManager as StaggeredGridLayoutManager).gapStrategy=StaggeredGridLayoutManager.GAP_HANDLING_NONE
    }
    FilesShowMode.DIR_SMALL -> {
        spanCount = 1
        layoutManager = LinearLayoutManager(requireContext())
    }
    FilesShowMode.SMALL -> {
        spanCount = 1
        layoutManager = LinearLayoutManager(requireContext())
    }
}
```

setUI() (4)

```
        binding.rvFiles.layoutManager = layoutManager
        if(showMode==FilesShowMode.DIR_BIG || showMode == FilesShowMode.BIG) {
           if (binding.rvFiles.itemDecorationCount == 0) {
              binding.rvFiles.addItemDecoration(
                 GridSpacingDecorator(
                    resources.getDimensionPixelSize(R.dimen.cardView_margin),
                    spanCount
                 )
              )
           }
         }
         else{
              if(binding.rvFiles.itemDecorationCount == 0){
                       binding.rvFiles.addItemDecoration(
LinearSpacingDecorator(resources.getDimensionPixelSize(R.dimen.cardView_margin),
        spanCount))
              }
        }
```

setUI() (5)

```
    if(showMode == FilesShowMode.DIR_BIG || showMode == FilesShowMode.DIR_SMALL){
        binding.imageButton.visibility=View.GONE
    }
    else{
        if(viewModel.listNotes.value != null)
        setAdapterFiles(viewModel.listNotes.value!!)
        binding.imageButton.visibility=View.VISIBLE
    }
    binding.imageButton.setOnClickListener{
        showMode = if(showMode == FilesShowMode.BIG)
            FilesShowMode.DIR_BIG
        else
            FilesShowMode.DIR_SMALL
        mActionMode = null
        selectedItem = ArrayList()
        setUI()
        loadData()
    }
    binding.sv.setOnQueryTextListener(MyQueryListener())

}
```

loadData()

```
private fun loadData(){
    Log.d("RESUME","RESUME")
    val dao = DbDigitalPhotoEditor.getInstance(requireContext()).digitalPhotoEditorDAO()
    CoroutineScope(Dispatchers.IO).launch{
        val results = dao.loadDirectories()
        val listDirectory = mutableListOf<Directory>()
        var size = 0
        var lastModify: Long
        for(res in results){
            size = dao.loadDirectorySize(res)
            lastModify = dao.getLastModifyDir(res)
            listDirectory.add(Directory(res,size, Date(lastModify)))
        }
        CoroutineScope(Dispatchers.Main).launch {
            viewModel.listDirectory.value = listDirectory
            binding.textView7.isVisible = listDirectory.isEmpty()
        }
    }
}
```

Metodi chiamati negli adapter

- All’interno delle classi adapter vengono chiamati i seguenti metodi della classe FragmentAllFiles:

selectedHandler

```
override fun selectedHandler(element: Any): Boolean {
    var isFirst = false
    if(selectedItem.isEmpty()){
        mActionMode = requireActivity().startActionMode(AllFilesActionModeCallback())
        isFirst = true
    }
    if(selectedItem.contains(element)) {
        selectedItem.remove(element)
        if(selectedItem.size==0)
            mActionMode!!.finish()
        return false
    }
    selectedItem.add(element)
    if(isFirst) {
        val note = Note()
        note.id = -1
        viewModel.selectedNote.value = note
        //Invoke reset only if the fragment exists
        (if (childFragmentManager.findFragmentByTag(DETAILS_FRAGMENT_TAG) == null) null
        else
            (childFragmentManager.findFragmentByTag(DETAILS_FRAGMENT_TAG) as FragmentNoteDetails))?.reset()
    }
    return true
}
```

```
fun isSelected(note: Note): Boolean{
    for(any in selectedItem){
        if(any is Note){
            if(any.id == note.id)
                return true
        }
    }
    return false
}
override fun isAnItemSelected(): Boolean {
    return selectedItem.isNotEmpty()
}
```

changeToFiles

```
fun changeToFiles(directory: String){
    CoroutineScope(Dispatchers.IO).launch{
        val files = DbDigitalPhotoEditor.getInstance(requireContext()).digitalPhotoEditorDAO().loadAllByDirectory(directory)
        CoroutineScope(Dispatchers.Main).launch{
            if(showMode == FilesShowMode.DIR_BIG)
                showMode = FilesShowMode.BIG
            else if(showMode == FilesShowMode.DIR_SMALL)
                showMode = FilesShowMode.SMALL
            setUI()
            actualDirectory = directory
            viewModel.listNotes.value=files.toMutableList()
            viewModel.sort(sortingType)
            selectedItem = ArrayList()
            mActionMode = null
        }
    }
}
```

Menu Contestuale

- Tenendo premuto su una directory o sul un file è possibile abilitare la selezione multipla degli elementi per poi dare all'utente la possibilità di eseguire determinate azioni

```
private inner class AllFilesActionModeCallback() : ActionMode.Callback{
    override fun onCreateActionMode(mode: ActionMode?, menu: Menu?): Boolean {
        if(showMode==FilesShowMode.DIR_BIG || showMode == FilesShowMode.DIR_SMALL)
            mode!!.menuInflater.inflate(R.menu.menu_onlong_digital,menu)
        else
            mode!!.menuInflater.inflate(R.menu.menu_onlong_files,menu)
        return true
    }

    override fun onPrepareActionMode(mode: ActionMode?, menu: Menu?): Boolean {
        return false
    }

    override fun onActionItemClicked(mode: ActionMode?, item: MenuItem?): Boolean {
        if(showMode==FilesShowMode.BIG || showMode == FilesShowMode.SMALL){
            //Files case
            when(item!!.itemId){
                R.id.it_files_delete -> deleteFiles(mode)
                R.id.it_files_merge -> mergeFiles(mode)
                else -> deleteFiles(mode)
            }
        }
        else{
            //Directory case
            deleteDirectories(mode)
        }
        return true
    }

    override fun onDestroyActionMode(mode: ActionMode?) {
        selectedItem = ArrayList()
        mActionMode=null
    }
```

```
private fun deleteDirectories(mode: ActionMode?){
    val alertDialog = AlertDialog.Builder(requireContext())
    alertDialog.setTitle(R.string.delete_directories_alert_title)
    alertDialog.setMessage(R.string.delete_directories_alert_message)
    alertDialog.setPositiveButton(R.string.yes){ dialogInterface: DialogInterface, i: Int ->
        CoroutineScope(Dispatchers.IO).launch{
            val dao = DbDigitalPhotoEditor.getInstance(requireContext()).digitalPhotoEditorDAO()
            val listDir = ArrayList<Directory>()
            for(elem in selectedItem){
                if(elem is Directory){
                    dao.deleteDirectory(elem.name)
                    listDir.add(elem)
                }
            }
            CoroutineScope(Dispatchers.Main).launch{
                val newList = viewModel.listDirectory.value!!
                newList.removeAll(listDir)
                viewModel.listDirectory.value = newList
                dialogInterface.dismiss()
                mode!!.finish()
            }
        }
    }
    alertDialog.setNegativeButton(R.string.no){ dialogInterface: DialogInterface, _: Int ->
        dialogInterface.dismiss()
        mode!!.finish()
    }
    alertDialog.show()
}
```

```
private fun deleteFiles(mode: ActionMode?){
    val alertDialog = AlertDialog.Builder(requireContext())
    alertDialog.setTitle(R.string.delete_files_alert_title)
    alertDialog.setTitle(R.string.delete_files_alert_message)
    alertDialog.setPositiveButton(R.string.yes){ dialogInterface: DialogInterface, _: Int ->
        val dao = DbDigitalPhotoEditor.getInstance(requireContext()).digitalPhotoEditorDAO()
        CoroutineScope(Dispatchers.IO).launch{
            val listNote = ArrayList<Note>()
            for(elem in selectedItem){
                if(elem is Note)
                    listNote.add(elem)
            }
            dao.deleteAll(listNote)
            CoroutineScope(Dispatchers.Main).launch{
                val list = viewModel.listNotes.value!!
                list.removeAll(listNote)
                viewModel.listNotes.value = list
                dialogInterface.dismiss()
                mode!!.finish()
            }
        }
    }
    alertDialog.setNegativeButton(R.string.no){ dialogInterface: DialogInterface, _: Int ->
        dialogInterface.dismiss()
        mode!!.finish()
    }
    alertDialog.show()
}
```

```
private fun mergeFiles(mode: ActionMode?){
    val stringBuilder = StringBuilder()
    for(elem in selectedItem){
        if(elem is Note){
            stringBuilder.append(elem.text)
            stringBuilder.append("\n")
        }
    }
    val note = Note(stringBuilder.toString(),"","","und",System.currentTimeMillis(),false)
    val intent = Intent(requireContext(),TextResultActivity::class.java)
    intent.putExtra("result",note)
    intent.putExtra("type", TextResultType.NOT_SAVED.ordinal)
    requireContext().startActivity(intent)
    mode!!.finish()
}
```

Cambiare la modalità visualizzazione dei dati (1)

```
R.id.it_preview -> {
    selectedItem = ArrayList()
    mActionMode = null
    if (showMode == FilesShowMode.DIR_BIG || showMode == FilesShowMode.BIG) {
        Log.d("HERE", "HERE")
        item.icon = ContextCompat.getDrawable(
            requireContext(),
            R.drawable.ic_baseline_grid_view_24
        )
        if (showMode == FilesShowMode.DIR_BIG) {
            showMode = FilesShowMode.DIR_SMALL
            setUI()
            setAdapterDirectory(viewModel.listDirectory.value!!)
        } else {
            showMode = FilesShowMode.SMALL
            setUI()
            setAdapterFiles(viewModel.listNotes.value!!)
        }
    }
```

Cambiare la modalità visualizzazione dei dati (2)

```
    } else {
        item.icon =
            ContextCompat.getDrawable(
                requireContext(),
                R.drawable.ic_baseline_format_list_bulleted_24
            )
        if (showMode == FilesShowMode.DIR_SMALL) {
            showMode = FilesShowMode.DIR_BIG
            setUI()
            setAdapterDirectory(viewModel.listDirectory.value!!)
        } else {
            showMode = FilesShowMode.BIG
            setUI()
            setAdapterFiles(viewModel.listNotes.value!!)
        }
    }
}
```

```
R.id.it_filter -> {
    val sharedPreferences = requireActivity().getSharedPreferences("filter_all_files",
        Activity.MODE_PRIVATE)
    var filterMode = sharedPreferences.getInt("filterMode",FilterMode.BY_TEXT.ordinal)
    var sortingType = sharedPreferences.getInt("sortingType",SortingType.ALPHABETIC_ASC.ordinal)
    val dialog = FilterDialog.getInstance(filterMode,sortingType)
    dialog.setOnFilterSelected{ filter: FilterMode, sorting: SortingType ->
        selectedItem = ArrayList()
        mActionMode = null
        filterMode = filter.ordinal
        sortingType = sorting.ordinal
        this.filterMode = filter
        this.sortingType = sorting
```

```
        if(showMode == FilesShowMode.DIR_BIG || showMode == FilesShowMode.DIR_SMALL) {
            if(filter == FilterMode.BY_COUNTRY){
                Toast.makeText(requireContext(),requireContext().
                                getString(R.string.no_country_directory),Toast.LENGTH_LONG).show()
            }
            viewModel.filterDirectory(binding.sv.query.toString())
            viewModel.sortDirectories(sorting)
        }
        else {
            viewModel.filter(binding.sv.query.toString(), filter,actualDirectory)
            viewModel.sort(sorting)
        }
        val editor = sharedPreferences.edit()
        editor.putInt("filterMode",filterMode)
        editor.putInt("sortingType",sortingType)
        editor.apply()
    }
    dialog.show(parentFragmentManager,"FILTERDIALOG")
}
```

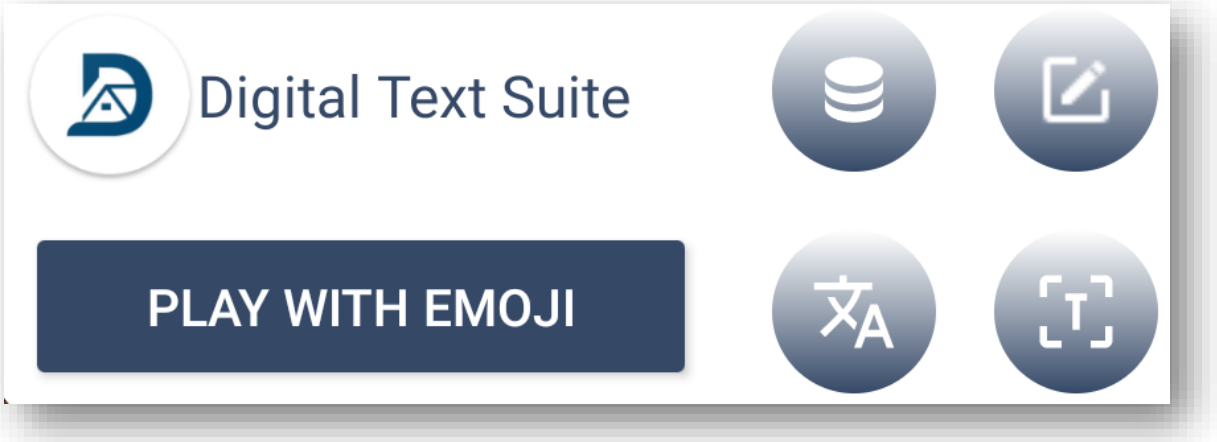
PDF Files

```
R.id.it_pdf -> {
    CoroutineScope(Dispatchers.IO).launch {
        val rootDir = RealMainActivity.rootDir
        val listFiles = getPdfFilesFromRootDir(rootDir)
        CoroutineScope(Dispatchers.Main).launch {
            val dialog = SelectPdfDialog.getInstance(listFiles)
            dialog.show(parentFragmentManager, "SELECTPDFDIALOG")
        }
    }

}
```

PDF Files (1)

```
private fun getPdfFilesFromRootDir(rootDir: File): List<File> {
    return rootDir.walk().filter{
        it.extension == "pdf"
    }.toList()
}
```

Widget



Widget (2)

- MyAppWidget.kt
- xml/my_app_widget_info.xml
- Layout/layout_my_app_widget.xml

Widget (3): RealMainActivity

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityRealMainBinding.inflate(layoutInflater)
    widget_metadata = intent.getIntExtra("fragment",-1)
    setContentView(binding.root)
    if(allPermissionsGranted()){
        init()
    }else{
        ActivityCompat.requestPermissions(this, REQUIRED_PERMISSIONS, REQUEST_CODE_PERMISSIONS)
    }
```

Widget (4): RealMainActivity

```
if(widget_metadata != -1) {
    binding.viewPagerMain.currentItem = widget_metadata
}
```

Digital Ink Recognition

- Permette all'utente di poter scrivere a mano su una whiteboard e digitalizzare il testo sotto forma di una Nota all'interno della nostra App.
- Le whiteboard possono essere memorizzate sul dispositivo per essere modificate successivamente
- Sono disponibili più di 300 linguaggi per la digitalizzazione.

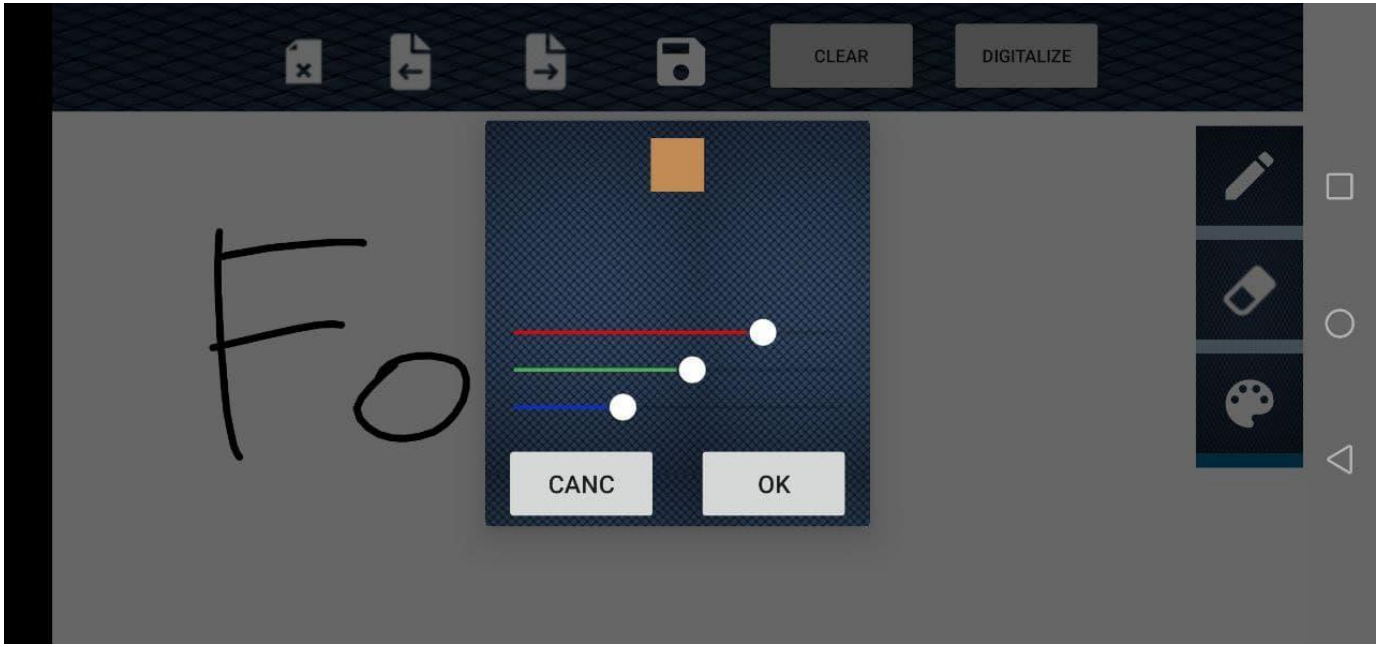
Digital Ink Recognition

- Nel secondo tab vengono mostrate tutte le whiteboard salvate e qui è possibile crearne una nuova.
- Le whiteboard digitalizzate hanno un bottone in basso a destra che mostra la nota associata.

Digital Ink Recognition

- La barra a destra permette di selezionare gli strumenti di disegno. La barra in alto fornisce strumenti di gestione della whiteboard

Digital Ink Recognition



- E’ possibile cambiare colore della penna con un simpatico Dialog dove l’utente può costruire il colore combinando l’rgb

Digital Ink Recognition

- Entriamo nel merito del codice…
- Per realizzare questo caso d'uso abbiamo utilizzato il servizio di API offerto da MLKit.
- Le api di MLKit operano l'analisi su oggetti di tipo Ink e restituiscono il testo riconosciuto.
- Per consentire all'utente di scrivere sullo schermo abbiamo realizzato un oggetto di view: Whiteboard

Digital Ink Recognition

- Per la visualizzazione delle whiteboard memorizzate abbiamo utilizzato un Fragment il DigitalInkFragment
- Le whiteboard vengono memorizzate nel database mentre i metadati sono memorizzati in un json nello storage del dispositivo

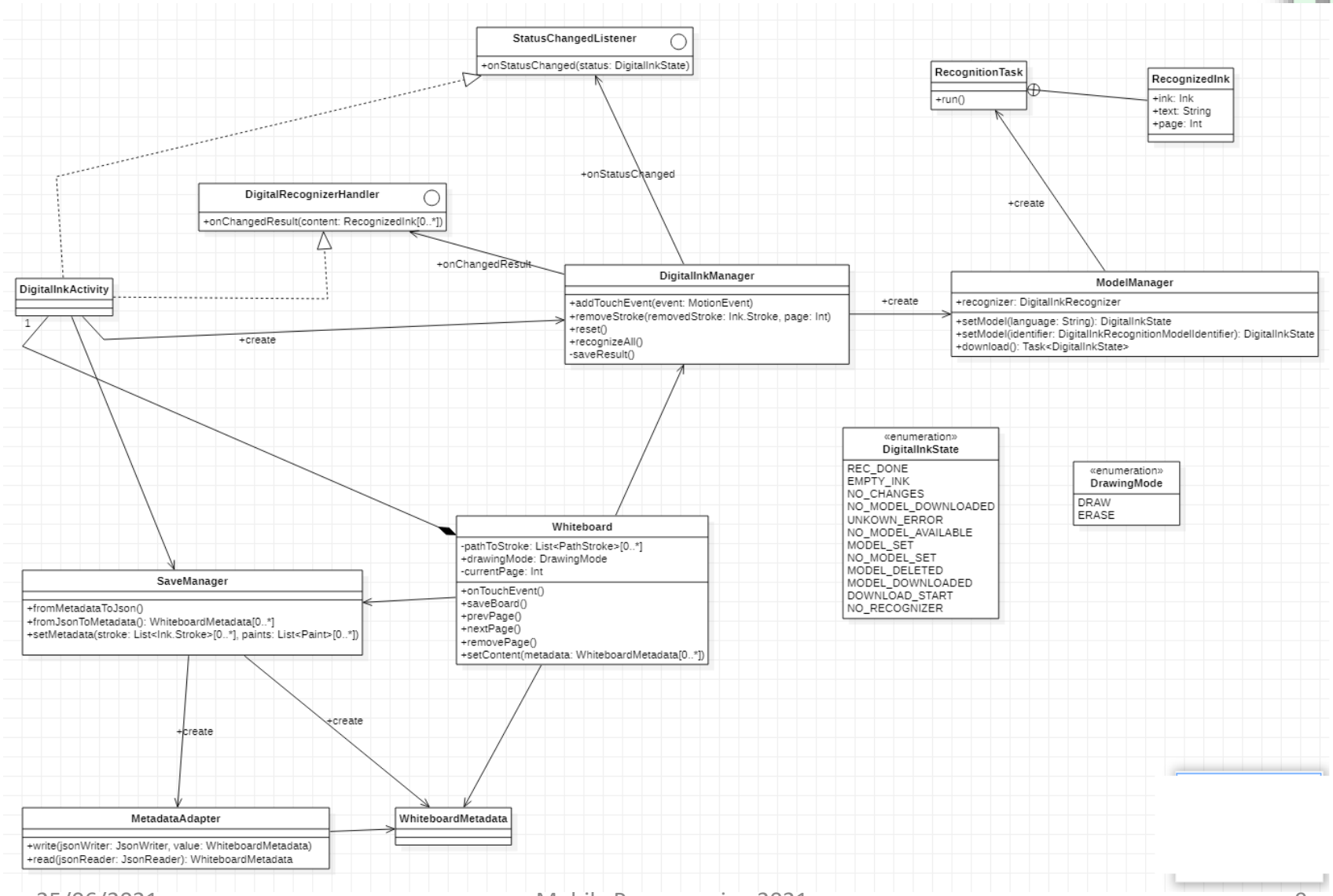
Digital Ink Recognition

- E’ stato realizzato anche un meccanismo di selezione multipla e rimozione per le whiteboard sfruttando le ActionMode
- Le whiteboard cambiano colore grazie ad un selector impostato come background

Digital Ink Recognition

- Per consentire l'interazione con la whiteboard è stata realizzata una attività apposita DigitalInkActivity
- Per una esperienza migliore questa activity ha una screen orientation in LANDSCAPE
- Per l'interazione con il manager l'activity implementa le interfacce StatusChangedListener e DigitalRecognizerHandler

DigitalInkRecognition



Whiteboard

- Whiteboard è una classe che estende View. Offre uno spazio bianco dove l'utente può disegnare.
- Una Whiteboard è autosufficiente e gestisce automaticamente la comunicazione con le classi responsabili della recognition
- Offre dei metodi per il salvataggio sullo storage del device e ripristino di dati salvati.

Whiteboard

- Per operare una Whiteboard ha bisogno di un DigitalInkManager.

```
fun setDigitalInkManager(digInkManager: DigitalInkManager){
    manager = digInkManager
}
```

- Per consentire all'utente di disegnare viene fatto l'override del metodo onTouchEvent

Whiteboard

```
override fun onTouchEvent(event: MotionEvent): Boolean {
    performClick()
    val action = event.actionMasked
    val x = event.x
    val y = event.y
    if(drawingMode == DrawingMode.DRAW) {
        when (action) {
            MotionEvent.ACTION_DOWN -> currentStroke.moveTo(x, y)
            MotionEvent.ACTION_MOVE -> currentStroke.lineTo(x, y)
            MotionEvent.ACTION_UP -> {
                currentStroke.lineTo(x, y)
                drawCanvas.drawPath(currentStroke, currentStrokePaint)
                paths[currentPage].add(currentStroke)
                lastStroke = currentStroke
                currentStroke = Path()
            }
            else -> {

            }
        }
        /**
         * Inform the manager of the event. The method returns true if and only if a new stroke has been added.
         * In this case the stroke is stored with the associated path
         */
        if(manager.addTouchEvent(event,currentPage)){
            pathToStroke[currentPage].add(PathStroke(lastStroke,manager.lastStroke,currentStrokePaint))
        }
        invalidate()
        return true
    }
```

Whiteboard

```
else{
   /**
    * Erase mode: Search among all the paths the one that contains the point corresponding to the touch event by
    * building a rectangle around it for each path and verifying that the rectangle contains the point
    */
   currentStroke = Path()
   val rect = RectF()
   for (i in 0 until pathToStroke[currentPage].size){
      pathToStroke[currentPage][i].path.computeBounds(rect,false)
      if(rect.contains(x,y)){
         /**
          * Notify the manager of the removal of the detected stroke
          */
         manager.removeStroke(pathToStroke[currentPage][i].stroke,currentPage)
         erase(i)
         break
      }
   }
   return false
}
```

Whiteboard

- E’ possibile aggiungere più di una pagina in una whiteboard, muoversi tra le pagine e rimuoverle tramite questi bottoni

Whiteboard

- Per aggiungere una pagina o spostarsi in avanti:

```
fun nextPage(){
   if(currentPage+1>=paths.size){
      paths.add(ArrayList())
      pathToStroke.add(ArrayList())
      manager.newPage()
   }
   currentStroke = Path()
   currentPage++
   onSizeChanged(
      canvasBitmap!!.width,
      canvasBitmap!!.height,
      canvasBitmap!!.width,
      canvasBitmap!!.height
   )
}
```

- Per spostarsi indietro:

```
fun prevPage(): Boolean{
   if(currentPage-1>=0)
      currentPage--
   else
      return false
   currentStroke = Path()
   onSizeChanged(
      canvasBitmap!!.width,
      canvasBitmap!!.height,
      canvasBitmap!!.width,
      canvasBitmap!!.height
   )
   return currentPage>0
}
```

Whiteboard

- Per rimuovere una pagina:

```
/**
 * This method allows you to remove the current page. If it is the only one present then a new one is created
 */
fun removePage(){
    paths.removeAt(currentPage)
    pathToStroke.removeAt(currentPage)
    if(currentPage == 0 && paths.isEmpty() && pathToStroke.isEmpty()){
        paths.add(ArrayList())
        pathToStroke.add(ArrayList())
    }
    else{
        if(currentPage>0)
            currentPage--
    }
    manager.deletePage(currentPage)
    currentStroke = Path()
    onSizeChanged(
        canvasBitmap!!.width,
        canvasBitmap!!.height,
        canvasBitmap!!.width,
        canvasBitmap!!.height
    )
    invalidate()
}
```

Whiteboard

- E’ possibile salvare una Whiteboard o ripristinarne una sfruttando i WhiteboardMetadata e la classe SaveManager. Una Whiteboard viene salvata in formato json.
- Offre un metodo saveBoard per il salvataggio e un metodo setContent per il ripristino

Whiteboard - saveBoard

```
/**
 * This method allows you to save the current whiteboard (and all its pages)
 * @param path is the file that must contain the metadata used to restore the whiteboard
 * @param imagePath is the file that must contain the whiteboard preview (Usually a front page image)
 */
fun saveBoard(path: File,imagePath: File){
    val listOfStrokes = mutableListOf<MutableList<Ink.Stroke>>()
    val listPaints = mutableListOf<MutableList<Paint>>()
    for(page in pathToStroke.indices) {
        listOfStrokes.add(mutableListOf())
        listPaints.add(mutableListOf())
        Log.d("PAGESTROKESIZE",pathToStroke[page].size.toString())
        for (i in pathToStroke[page].indices) {
            listOfStrokes[page].add(pathToStroke[page][i].stroke)
            listPaints[page].add(pathToStroke[page][i].paint)
        }
    }
    val saveManager = SaveManager()
    saveManager.path = path.absolutePath
    saveManager.setMetadata(listOfStrokes,listPaints)
    saveManager.fromMetadataToJson()
    val fileOutputStream = FileOutputStream(imagePath)
    val bitmap = Bitmap.createBitmap(canvasBitmap!!.width,canvasBitmap!!.height,Bitmap.Config.ARGB_8888)
    val canvas = Canvas()
    canvas.setBitmap(bitmap)
    for(p in pathToStroke[0]){
        canvas.drawPath(p.path,p.paint)
    }
    bitmap.compress(Bitmap.CompressFormat.PNG,100,fileOutputStream)
}
```

Whiteboard – setContent (1)

```
/**
 * This method allows you to restore a whiteboard instance
 * @param metadata The stored WhiteboardMetadata
 */
fun setContent(metadata: MutableList<WhiteboardMetadata>){
    metadata.sortBy{
        it.id
    }
    manager.setStartNumberPage(metadata.size)
    currentStroke = Path()
    paths = mutableListOf()
    pathToStroke = mutableListOf()
    currentPage = -1
    for(whiteboard in metadata) {
        currentPage ++
        paths.add(ArrayList())
        pathToStroke.add(ArrayList())
        val decoded = whiteboard.decodeMetadata()
        val listStroke = mutableListOf<Ink.Stroke>()
        for(elem in decoded){
            listStroke.add(elem.stroke)
        }
        manager.setInkStrokes(listStroke,currentPage)
```

Whiteboard – setContent (2)

```
for (stroke in whiteboard.decodeMetadata()) {
        for (i in stroke.stroke.points.indices) {
           if (i == 0) {
              currentStroke.moveTo(stroke.stroke.points[i].x, stroke.stroke.points[i].y)
           } else {
              currentStroke.lineTo(stroke.stroke.points[i].x, stroke.stroke.points[i].y)
           }
        }
        paths[currentPage].add(currentStroke)
        lastStroke = currentStroke
        currentStroke = Path()
        color = stroke.paint.color
        this.stroke = (stroke.paint.strokeWidth/(resources.displayMetrics.densityDpi/ DisplayMetrics.DENSITY_DEFAULT)).toInt()
        resetPaint()
        pathToStroke[currentPage].add(PathStroke(lastStroke, stroke.stroke,currentStrokePaint))
      }
   }
   currentPage = 0
   currentStroke = Path()
}
```

Whiteboard

- Una Whiteboard offre anche un metodo temporarySave per il salvataggio in un file in cache da utilizzare per una bufferizzazione temporanea della whiteboard.
- Il metodo utilizza gli stessi meccanismi descritti precedentemente per il salvataggio ma crea l'uri nella cache del dispositivo e restituisce l'uri corrispondente al file creato

DigitalInkManager

- DigitalInkManager è una classe che gestisce le operazioni di digital ink recognize
- Costruisce gli Ink da analizzare utilizzando le api di MLKit e le informazioni fornite dalla Whiteboard
- E’ responsabile della creazione e gestione del ModelManager per il download dei modelli
- Offre un metodo recognizeAll per la digitalizzazione degli Ink

DigitalInkManager

- Un oggetto di DigitalInkManager possiede uno stato che può variare nel tempo!
- Per gestire le variazioni di stato l'activity deve implementare l'interfaccia StatusChangedListener

```
interface StatusChangedListener {
   /** This method is called when the recognized content changes.  */
   fun onStatusChanged(status: DigitalInkState)
}
```

- Lo stato del manager è uno dei valori dell'enum DigitalInkState

DigitalInkManager

- Per ricevere i risultati di una digital ink recognition l'activity deve implementare l'interfaccia DigitalRecognizerHandler

```
interface DigitalRecognizerHandler {
   /**
    * This method is called by the DigitalInkManager when a new result is ready
    */
   fun onChangedResult(content: MutableList<RecognitionTask.RecognizedInk>)
}
```

- L'activity deve inoltre registrarsi come handler usando gli appositi metodi esposti dal DigitalInkManager

ModelManager

- Questa classe è responsabile della gestione dei modelli di MLKit per la digital ink recognition
- E’ anche responsabile della creazione del recognizer di MLKit

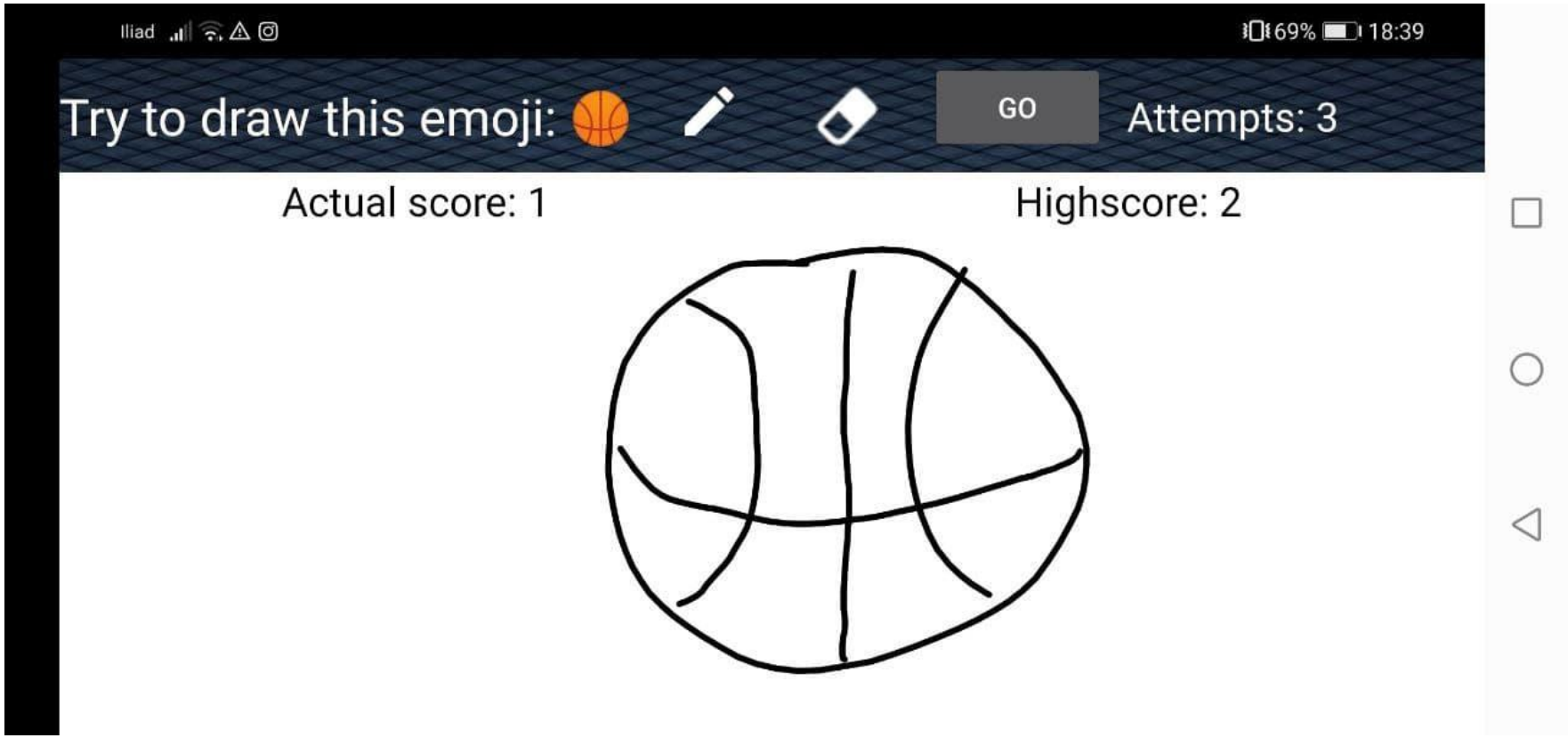
SaveManager

- Questa classe è responsabile delle operazioni di IO per una Whiteboard.
- Usa le api offerte da Gson per la lettura e scrittura di file Json
- E’ stata creata una classe MetadataAdapter per fornire un TypeAdapter a Gson per la scrittura in memoria di oggetti di tipo WhiteboardMetadata in formato Json

Play With Emoji

- Abbiamo realizzato un minigame chiamato Play With Emoji accessibile tramite Widget oppure tramite notifica. Il minigame consiste nel riuscire a disegnare una emoji suggerita dall'app
- Sfrutta l'oggetto Whiteboard e le classi relative alla digital ink recognition già descritte in precedenza
- E' stata realizzata una apposita Activity chiamata PlayWithEmojiActivity

PlayWithEmoji

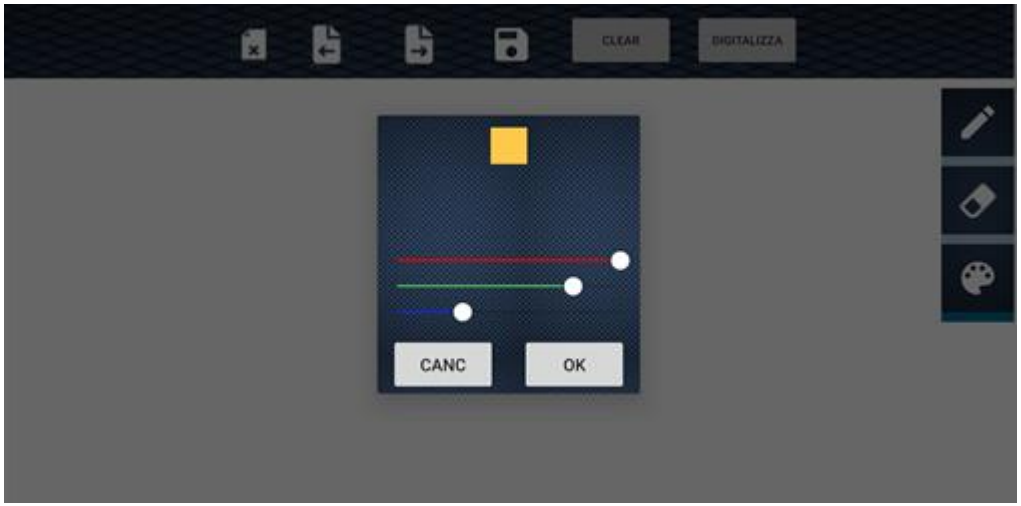


Notifiche

- L’app invia una notifica all’utente invitandolo a giocare al minigame PlayWithEmoji dopo 24 ore dall’ultima volta che ha aperto il gioco
- La notifica viene inviata con intervallo regolare di 24 ore a meno che l’utente non apra il minigame
- A tale scopo sono state realizzate 2 classi :
  - NotificationBuilder responsabile della creazione della notifica
  - NotificationReceiver responsabile del lancio della notifica ogni 24 ore

DialogColor -1

- Questo è un Fragment che estende la classe DialogFragment
- DialogColor permette all'utente di scegliere il colore della penna per disegnare sulla lavagna, tramite seekbar, combinando la tabella rgb



DialogColor -2

```
class ColorDialog : DialogFragment() {
    private var colorImageView: ImageView? = null
    private var redValue : Int = 0
    private var greenValue : Int = 0
    private var blueValue : Int = 0
    private var colors : Int = Color.BLACK
    private var colorListener: (colors : Int) -> Unit = {

    }
    private var cancelListener: () -> Boolean ={
        true
    }
    private lateinit var binding: ChooseColorDialogBinding

    companion object{
        fun getInstance(): ColorDialog{
            var instance : ColorDialog? = null
            if(instance == null){
                instance = ColorDialog()
            }
            return instance
        }
    }
```

DialogColor -3

```
override fun onCreateView( inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View {
    binding = ChooseColorDialogBinding.inflate(inflater)
    colorImageView = binding.ColorPicker
    colorImageView!!.setBackgroundColor(Color.BLACK)
    isCancelable = false
    setSeek()
    return binding.root
}
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    binding.ColorPicker
    binding.seekBarB
    binding.seekBarG
    binding.seekBarR
    setSeek()
    binding.btnOk.setOnClickListener{
        colorListener(colors)
        dismiss()
    }
    binding.btnCanc.setOnClickListener{
        if(cancelListener())
            dismiss()
    }
}
fun setOnColorSelected(listener: (color: Int)-> Unit){
    this.colorListener=listener
}
fun setOnCancelSelected(listener: () -> Boolean){
    cancelListener = listener
}
```

Gestione della seekbar -1

- Per gestire il cambiamento di valore della seekbar bisogna eseguire override dei metodi:
  - onProgressChanged(
    seekBar: SeekBar,
    progress: Int,
    fromUser: Boolean)//segnala il cambiamento del livelli
  - onStartTrackingTouch(
    seekBar: SeekBar?) //segnala inizio del tocco da parte dell'utente
  - onStopTrackingTouch(
    seekBar: SeekBar?) //segnala fine del tocco da parte dell'utente

Gestione della seekbar -2

```
private fun setSeek(){
    val seekBarR = binding.seekBarR
    val seekBarG = binding.seekBarG
    val seekBarB = binding.seekBarB
    seekBarR.setOnSeekBarChangeListener(mChangeListener)
    seekBarG.setOnSeekBarChangeListener(mChangeListener)
    seekBarB.setOnSeekBarChangeListener(mChangeListener)
}
private val mChangeListener: SeekBar.OnSeekBarChangeListener = object :
    SeekBar.OnSeekBarChangeListener {
    override fun onProgressChanged(
        seekBar: SeekBar,
        progress: Int,
        fromUser: Boolean
    ) {
        val viewId = seekBar
        when (viewId) {
            binding.seekBarR -> redValue = progress
            binding.seekBarG -> greenValue = progress
            binding.seekBarB -> blueValue = progress
        }
        colors = Color.rgb(redValue, greenValue, blueValue)
        colorImageView!!.setBackgroundColor(colors)
    }
    override fun onStartTrackingTouch(seekBar: SeekBar?) { }

    override fun onStopTrackingTouch(seekBar: SeekBar?) { }
}
```

Creazione del dialog

```
binding.btnPickColor.setOnClickListener{
    val colorPick = ColorDialog.getInstance()
    var colore : Int
    colorPick.setOnColorSelected {
        colore = it
        binding.whiteboard.drawingMode = DrawingMode.DRAW
        binding.selected2.setBackgroundColor(ContextCompat.getColor(this,R.color.unselected))
        binding.selected.setBackgroundColor(ContextCompat.getColor(this,R.color.selected_blue))
        binding.selected3.setBackgroundColor(ContextCompat.getColor(this,R.color.unselected))
        setColor(colore)
    }
    colorPick.setOnCancelSelected {
        binding.whiteboard.drawingMode = DrawingMode.DRAW
        binding.selected2.setBackgroundColor(ContextCompat.getColor(this,R.color.unselected))
        binding.selected.setBackgroundColor(ContextCompat.getColor(this,R.color.selected_blue))
        binding.selected3.setBackgroundColor(ContextCompat.getColor(this,R.color.unselected))
        true
    }
    colorPick.show(supportFragmentManager,"ColorDialog")
```

PenDialog

```
penTouch = 0
…
binding.btnPen.setOnClickListener{
    binding.whiteboard.drawingMode= DrawingMode.DRAW
    binding.whiteboard.isEnabled = true
    penTouch++
    …
    val penPick = PenDialog.getInstance()
    var value : Int
    penPick.setOnStrokeSelected {
        penTouch = 1
        value = it
        setStroke(value)
    }
    if(penTouch==2)
        penPick.show(supportFragmentManager,"PenDialog")
}
```

TextResultActivity -1

- Una volta creata una nota abbiamo la possibilità tramite un menu di scegliere se modificarla, salvarla, copiarla, o cancellarla
- Inoltre ci sono 3 diverse features:
  - Traduzione
  - Salvare in formato pdf
  - Metterla tra i preferiti

TextResultActivity -2

```
class TextResultActivity : AppCompatActivity() {
…
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityTextResultBinding.inflate(layoutInflater)
    setContentView(binding.root)
    note = intent.getParcelableExtra("result") ?: Note("","","","",System.currentTimeMillis(),false)
    textResult = note.text
    language = note.language
    originalText=textResult
    whiteboard = intent.getParcelableExtra("whiteboard") ?: DigitalizedWhiteboards()
    val ordinal = intent.getIntExtra("type",TextResultType.NOT_SAVED.ordinal)
    type = when(ordinal){
        TextResultType.SAVED.ordinal -> TextResultType.SAVED
        TextResultType.EDITABLE.ordinal -> TextResultType.EDITABLE
        else -> TextResultType.NOT_SAVED
    }
    initializeDB()
    setUI()
}
```

Creazione del menu -1

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.text_result_menu,menu)
    this.menu=menu!!
    setType(menu)
    return super.onCreateOptionsMenu(menu)
}
private fun setType(menu: Menu?){
    var itSave: MenuItem? = null
    var itDelete: MenuItem? = null
    var itUndo: MenuItem? = null
    for (item in menu!!.children){
        when(item.itemId){
            R.id.it_delete -> itDelete=item
            R.id.it_save -> itSave=item
            R.id.it_undo -> itUndo=item
        }
    }
    when(type){
        TextResultType.SAVED -> {
            itDelete!!.isVisible=true
            itSave!!.isVisible=false
            itUndo!!.isVisible=false
        }
        TextResultType.NOT_SAVED -> {
            itDelete!!.isVisible=false
            itSave!!.isVisible=true
            itUndo!!.isVisible=false
        }
        TextResultType.EDITABLE->{
            itDelete!!.isVisible=false
            itSave!!.isVisible=true
            itUndo!!.isVisible=true
        }
```

Creazione del menu -2

```
 override fun onOptionsItemSelected(item: MenuItem): Boolean
{
      when(item.itemId){
         R.id.it_save -> { }
         R.id.it_editable->{ }
         R.id.it_undo -> { }
         R.id.it_delete -> { }
         R.id.it_copy -> { }
   return super.onOptionsItemSelected(item)
}
```

Salvare la nota

```
CoroutineScope(Dispatchers.IO).launch {
    val directoryList = dao.loadDirectories()
    CoroutineScope(Dispatchers.Main).launch {
        val dialog = MakeDirectoryDialog.getInstance()
        dialog.setDirectoryList(directoryList)
        if (type == TextResultType.NOT_SAVED) {
            dialog.setOnDirectorySelected { directory: String, title: String ->
                note.text = textResult
                note.language = language
                note.directory = directory
                note.title = title
                CoroutineScope(Dispatchers.IO).launch {
                    dao.insertNote(note)
                    ...
                }
            }
        }
    }
}
```

Editare nota

- Prima di poter modificare la nota bisogna salvarla

```
if(type==TextResultType.NOT_SAVED){
        Toast.makeText(this,getString(R.string.not_saved_edit),Toast.LENGTH_LONG).show()
    }
        ....
if (editable) {
    type = TextResultType.EDITABLE
    binding.editTextTextMultiLine.isEnabled = true
    setType(menu)
}
```

Ritornare alla nota originale

```
R.id.it_undo -> {
    binding.editTextTextMultiLine.text.clear()
    binding.editTextTextMultiLine.text.append(originalText)
    textResult=originalText
}
```

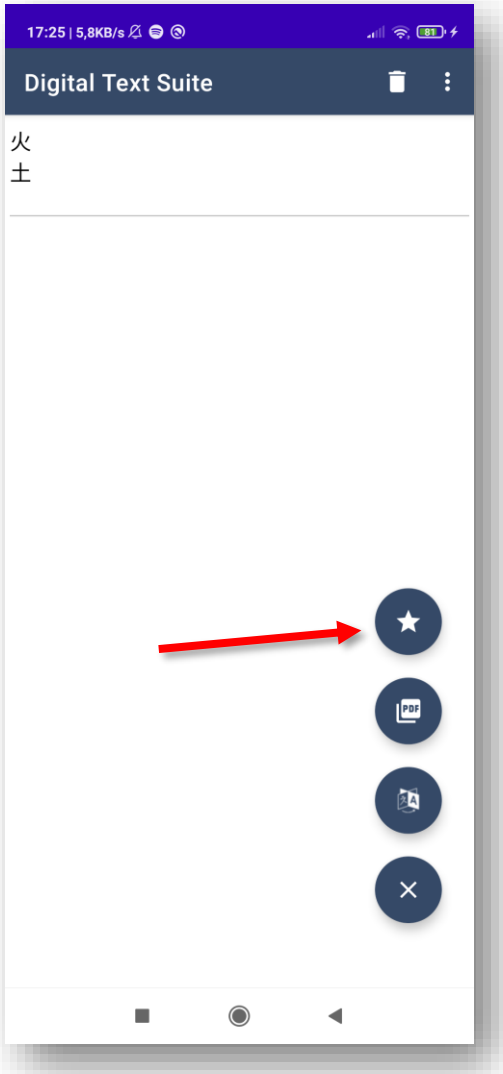
Eliminare la nota

```
CoroutineScope(Dispatchers.IO).launch {
dao.deleteNote(note)
    CoroutineScope(Dispatchers.Main).launch {
        dialogInterface.cancel()
        finish()
    }
}
```

Copiare la nota

```
R.id.it_copy -> {
    val clipboard = getSystemService(Context.CLIPBOARD_SERVICE) as ClipboardManager
    val clip = ClipData.newPlainText("note",textResult)
    clipboard.setPrimaryClip(clip)
```

Mettere la nota tra i preferiti



- Una feature che troviamo è quella di salvare le nostre note preferite per trovarle più facilmente nella sezione dedicata
- Nel bottone indicato apparirà la stella piena (salvata tra i preferiti), altrimenti solo contorno della stella

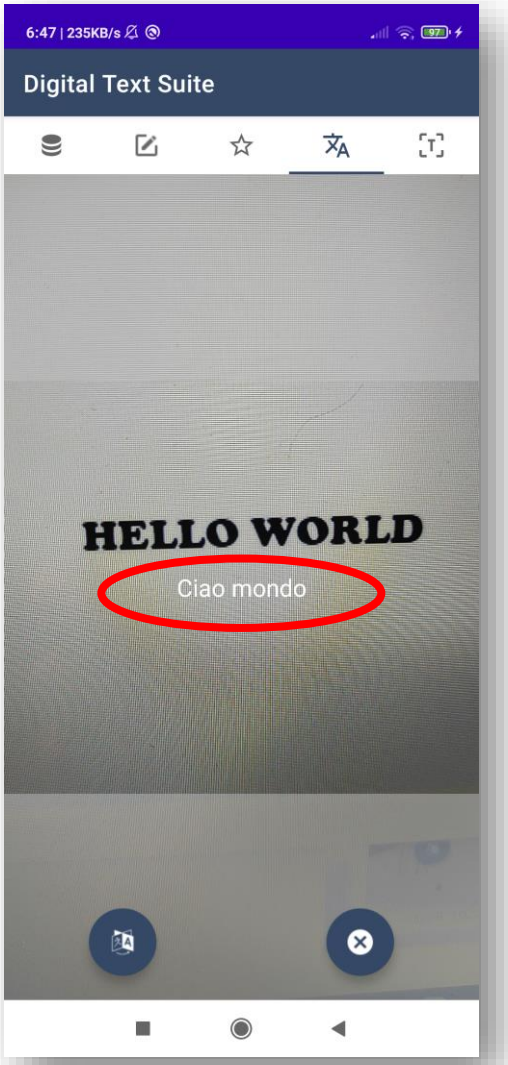
Salvare la nota tra i preferiti

```
binding.fabFavourite.setOnClickListener{
    CoroutineScope(Dispatchers.IO).launch{
        note.preferito = !note.preferito
        dao.updateNote(note)
        CoroutineScope(Dispatchers.Main).launch{
            binding.fabFavourite.setImageDrawable(if(note.preferito)
                ContextCompat.getDrawable(this@TextResultActivity,R.drawable.ic_baseline_star_24)
            else
                ContextCompat.getDrawable(this@TextResultActivity,R.drawable.favourite_icon_24))
        }
    }
}
```

Traduzione

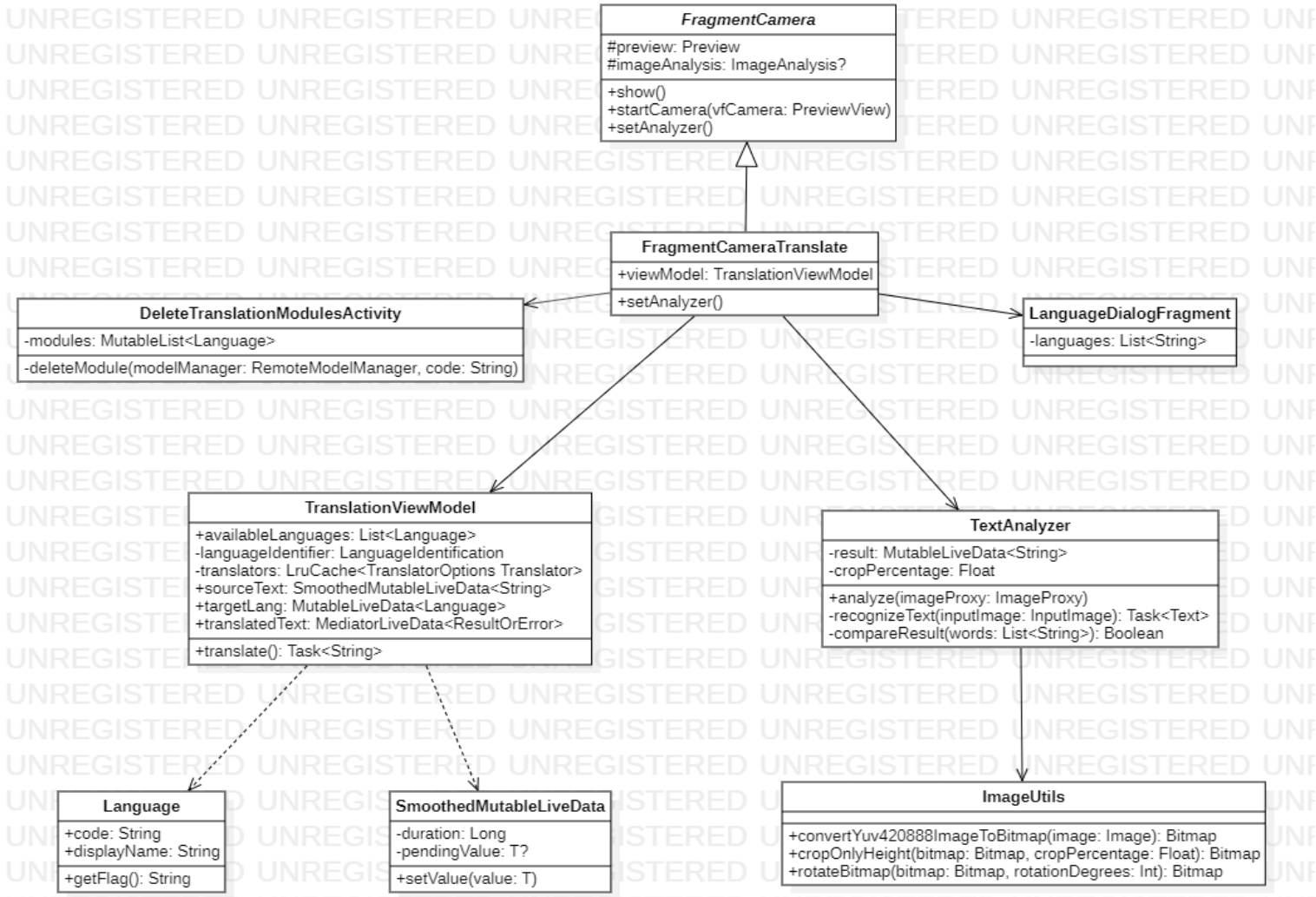
- Tra le features offerte da «Digital Text Suite», c’è la traduzione di testo
- Questa funzionalità viene offerta in due diverse modalità:
  - Traduzione Real-Time
  - Traduzione statica

Traduzione Real-Time



- È disponibile nel 4° tab del tab layout principale dell'app
- Utilizza la fotocamera per prendere in input stream di immagini e analizzarle
- Viene riconosciuto il testo presente nelle immagini e la lingua in cui è scritto
- È possibile selezionare una lingua target in cui tradurre il testo riconosciuto

Struttura


FragmentCamera
#preview: Preview
#imageAnalysis: ImageAnalysis?
+show()
+startCamera(vfCamera: PreviewView)
+setAnalyzer()
FragmentCameraTranslate
+viewModel: TranslationViewModel
+setAnalyzer()
DeleteTranslationModulesActivity
-modules: MutableList<Language>
-deleteModule(modelManager: RemoteModelManager, code: String)
LanguageDialogFragment
-languages: List<String>
TranslationViewModel
+availableLanguages: List<Language>
-languageIdentifier: LanguageIdentification
-translators: LruCache<TranslatorOptions Translator>
+sourceText: SmoothedMutableLiveData<String>
+targetLang: MutableLiveData<Language>
+translatedText: MediatorLiveData<ResultOrError>
+translate(): Task<String>
TextAnalyzer
-result: MutableLiveData<String>
-cropPercentage: Float
+analyze(imageProxy: ImageProxy)
-recognizeText(inputImage: InputImage): Task<Text>
-compareResult(words: List<String>): Boolean
Language
+code: String
+displayName: String
+getFlag(): String
SmoothedMutableLiveData
-duration: Long
-pendingValue: T?
+setValue(value: T)
ImageUtils
+convertYuv420888ImageToBitmap(image: Image): Bitmap
+cropOnlyHeight(bitmap: Bitmap, cropPercentage: Float): Bitmap
+rotateBitmap(bitmap: Bitmap, rotationDegrees: Int): Bitmap

FragmentCameraTranslate

- Questa classe è un Fragment che estende la classe astratta FragmentCamera, la quale mette a disposizione tutti i metodi necessari per il corretto funzionamento della fotocamera
- Sono state utilizzate le API della libreria di Android CameraX
- Si fa override del metodo `setAnalyzer()` per settare un TextAnalyzer come analizzatore delle immagini

FragmentCameraTranslate

```
override fun setAnalyzer() {
    try{
        val analyzer = TextAnalyzer(requireContext(), lifecycle,
viewModel.sourceText, CROP_PERCENTAGE)
        if (imageAnalysis != null) {
            imageAnalysis.also { it!!.setAnalyzer(cameraExecutor, analyzer) }
        }else{
            imageAnalysis = ImageAnalysis.Builder()
                .setTargetResolution(Size(1280, 720))
                .setBackpressureStrategy(ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)
                .build()
                .also { it.setAnalyzer(cameraExecutor, analyzer) }
        }
    }catch(exception: IllegalStateException){

    }
}
```

TextAnalyzer

- Implementa l'interfaccia ImageAnalysis.Analyzer
- Utilizza delle API di MLKit per ottenere un riconoscitore di testo, con cui processare le immagini
- Dell'immagine in input, viene analizzata solo una parte; infatti viene prima ruotata nella giusta direzione e poi ne viene fatto un crop, tagliando una percentuale della parte superiore e inferiore. Per queste operazioni, si utilizza la classe ImageUtils

TextAnalyzer

```
override fun analyze(imageProxy: ImageProxy) {
    val mediaImage = imageProxy.image ?: return
    val rotationDegrees = imageProxy.imageInfo.rotationDegrees

    val bitmap = ImageUtils.convertYuv420888ImageToBitmap(mediaImage)
    val rotatedBitmap = ImageUtils.rotateBitmap(bitmap, rotationDegrees)
    val croppedBitmap = ImageUtils.cropOnlyHeight(rotatedBitmap, cropPercentage)

    recognizeText(InputImage.fromBitmap(croppedBitmap, 0))
        .addOnCompleteListener {
            // close the imageProxy to allow following images to be analyzed
            imageProxy.close()
        }
}
```

TextAnalyzer

- Per evitare continui cambiamenti nel testo analizzato, sì è cercato di stabilire una threshold dinamica per il cambiamento del testo riconosciuto
- La threshold è calcolata in base alla differenza tra il testo precedentemente riconosciuto e quello attuale
- Tale differenza tiene conto sia del numero di parole che della differenza tra le parole stesse
- Implementata col metodo `compareResult()`

TextAnalyzer

```
private fun compareResult(words: List<String>): Boolean {
    if (result.value == null) {
        return true
    }
    val actualWords = result.value!!.split(" ")
    val size = Math.min(words.size, actualWords.size)
    val maxSize = Math.max(words.size, actualWords.size)
    // reference value of number of words
    val ref = (size * DIFFERENCE_PERCENTAGE)
    val refInt: Int
    if (ref - ref.toInt() >= 0.5f)
        refInt = ref.toInt() + 1
    else
        refInt = ref.toInt()
    var count = 0
    // checking differences among words
    for (i in 0 until size) {
        if (!words[i].equals(actualWords[i]))
            count++
        // if enough different ...
        if (count >= refInt)
            return true
    }
    // else if size is enough different ...
    if (maxSize - size + count >= refInt)
        return true
    return false
}
```

TranslationViewModel

- È la classe in cui si ha il riferimento a tutte le componenti necessarie per la traduzione:
  - Testo sorgente
  - Lingua sorgente
  - Lingua destinazione
- La lingua sorgente viene ottenuta utilizzando un LanguageIdentifier, presente nelle librerie di MLKit, che analizza il testo sorgente

TranslationViewModel

- Per effettuare la traduzione, vengono utilizzati degli oggetti di tipo Translator di MLKit
- Sul translator settato correttamente con lingua d'origine e di destinazione, viene invocato il metodo `downloadModelIfNeeded()`
- Tale metodo provvede a scaricare dalla rete i modelli di traduzione della lingua indicata, qualora non siano presenti sul dispositivo

TranslationViewModel

```
val translatorOptions = TranslatorOptions.Builder()
    .setSourceLanguage(sourceLangCode)
    .setTargetLanguage(targetLangCode)
    .build()

// get the translator
val translator = translators[translatorOptions]


modelDownloadTask = translator.downloadModelIfNeeded().addOnCompleteListener {
    modelDownloading.setValue(false)
}


return modelDownloadTask.onSuccessTask {
    // translate the source text
    translator.translate(text)
}.addOnCompleteListener {
    translating.value = false
}
```

TranslationViewModel

- Per quanto riguarda il testo sorgente, il testo tradotto, le lingue sorgente e target, sono tutti dei MutableLiveData
- Tuttavia, sono state usate due varianti particolari di tale tipo di dato, di cui una custom:
  - SmoothedMutableLiveData (custom)
  - MediatorLiveData, per avere aggiornamenti quando cambia un singolo componente di un gruppo di MutableLiveData

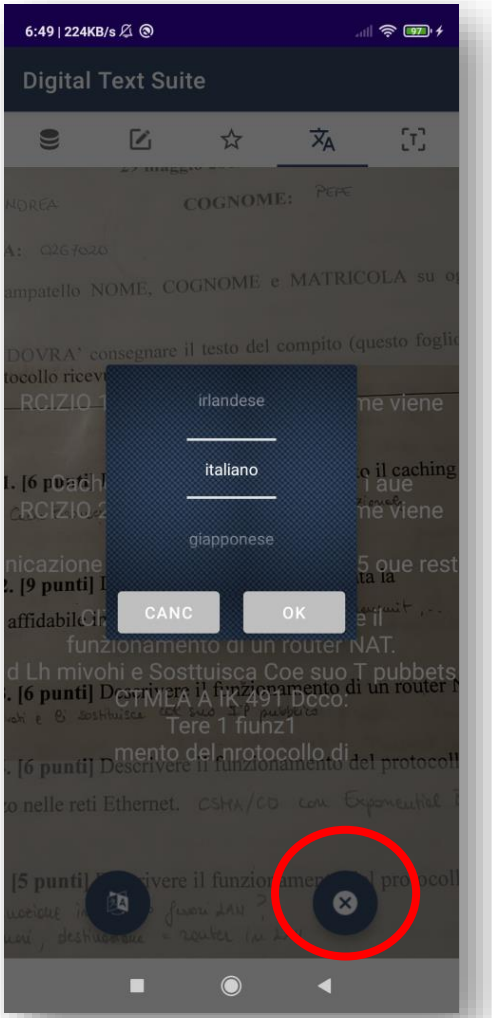
Scelta della lingua

- La lingua di destinazione è selezionabile tramite il bottone in basso a sinistra nel layout della traduzione real-time
- Apparirà il dialog in figura, che permetterà di scegliere tra oltre 50 idiomi disponibili

Scelta della lingua

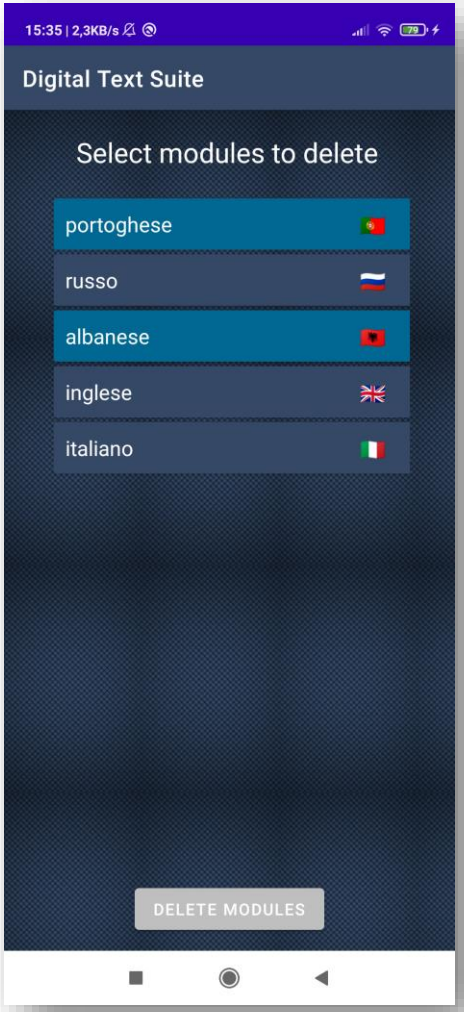
- Il dialog è stato realizzato estendendo la classe DialogFragment
- Presenta un Number Picker per la scelta della lingua cui è stato assegnato uno stile personalizzato
- In generale, per la gestione delle lingue, è stato creato un mapping tra gli idiomi riconosciuti da MLKit e una classe custom `Language`, cui è stato aggiunto il metodo `getFlag()` per il mapping tra lingue e una bandiera di un paese che meglio la rappresentasse

Rimozione dei modelli di traduzione



- Cliccando sul FloatingActionButton in basso a destra, si aprirà una nuova activity in cui sarà possibile eliminare i modelli per la traduzione precedentemente scaricati
- Ogni modello pesa all'incirca 30MB

DeleteTranslationModulesActivity



- Il layout dell'activity consta sostanzialmente di una RecyclerView, in cui è possibile una selezione multipla degli items da eliminare
- Si usa il meccanismo delle Coroutine per verificare quali modelli sono presenti nel dispositivo; ne viene poi fatto il mapping per costruire una lista di Language da passare all'adapter della RecyclerView per la visualizzazione

DeleteTranslationModulesActivity

```
val modelManager = RemoteModelManager.getInstance()
// get translation models stored on the device
CoroutineScope(Dispatchers.IO).launch {
    Tasks.await(modelManager.getDownloadedModels(TranslateRemoteModel::class.java)
        .addOnSuccessListener{ models ->
            models.forEach {
                modules.add(Language(it.language))
            }
        }
        .addOnFailureListener{
            Log.d("Modules", "Unable to detect downloaded modules")
            finish()
        })
```

```
CoroutineScope(Dispatchers.Default).launch {
    deleteModule(modelManager, it.code)
}
```

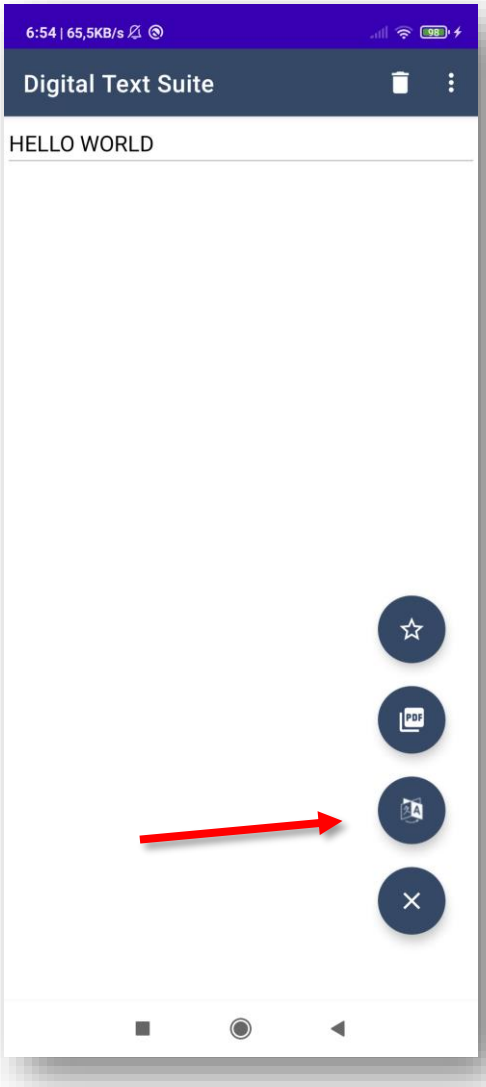
DeleteTranslationModulesActivity

```
private fun deleteModule(modelManager: RemoteModelManager, code: String) {
    val model = TranslateRemoteModel.Builder(code).build()
    modelManager.deleteDownloadedModel(model).addOnFailureListener {
        Toast.makeText(this, "Deletion of model $code failed", Toast.LENGTH_SHORT).show()
    }
}
```

Traduzione statica

- È possibile anche effettuare la traduzione del testo in maniera statica, a partire da una nota precedentemente creata
- Questa feature è disponibile nella TextResultActivity

Traduzione statica



- Cliccando sul FAB indicato in figura, apparirà lo stesso dialog visto in precedenza per la scelta della lingua target
- Ancora una volta, viene utilizzato il meccanismo delle Coroutines per creare la lista di lingue da passare al dialog

Translator

- Questa volta, per effettuare la traduzione si utilizza la classe `Translator`, in quanto non si ha più bisogno dei MutableLiveData
- La traduzione vera e propria avviene sempre tramite le classi Translator di MLKit, ma stavolta il testo e i codici della lingua sorgente e destinazione vengono direttamente passati come parametri alla funzione `translate()`

Translator

- La funzione translate() restituisce un Task<String>, il quale, come in precedenza, fa il download dei modelli di traduzione, se necessari, e provvede a tradurre il testo
- Tuttavia, per ottenere il risultato bisogna attendere che il Task venga completato, quindi viene bloccata la UI, ma appare una ProgressBar accompagnata da una TextView per avvisare l'utente che la traduzione è in corso

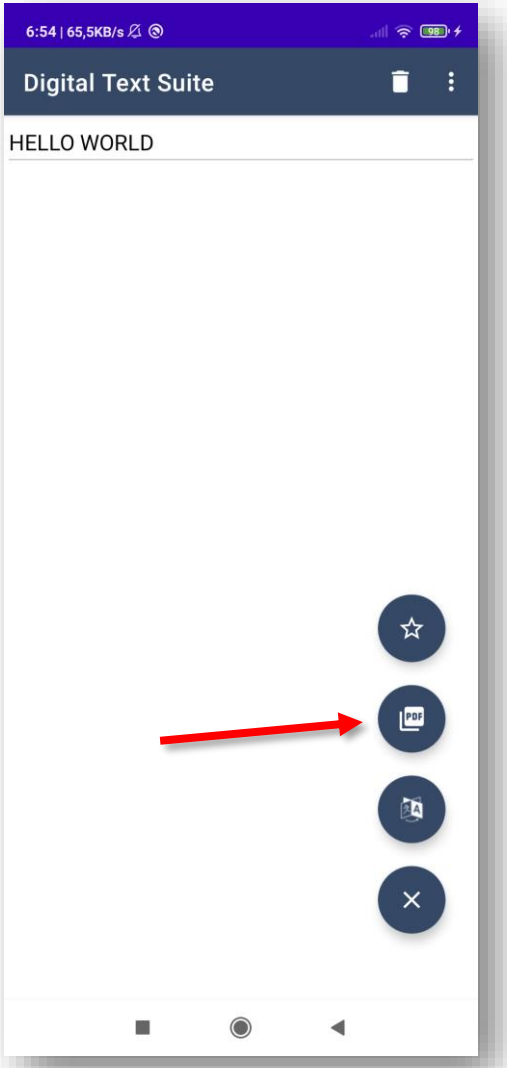
TextResultActivity

```
setNormalLayoutEnable(false)
binding.grpTranslation.visibility = View.VISIBLE
CoroutineScope(Dispatchers.Default).launch {
    val translator = Translator()

    // waiting the translation, blocking UI
    val newRes = Tasks.await(translator.translate(textResult, language, targetLang))
    CoroutineScope(Dispatchers.Main).launch {
        binding.grpTranslation.visibility = View.GONE
        val intent = Intent(this@TextResultActivity,TextResultActivity::class.java)
        val newNote = Note(newRes,"","",targetLang,System.currentTimeMillis(),false)
        intent.putExtra("result",newNote)
        intent.putExtra("type",TextResultType.NOT_SAVED)
        startActivity(intent)
        setNormalLayoutEnable(true)
    }
}
```

TextResultActivity

- Una volta ottenuto il testo tradotto, viene creata una nuova nota (classe `Note`)
- Dopodiché viene lanciata una Intent verso un'altra istanza di TextResultActivity, passando la nuova nota creata come dato

Salvataggio in PDF



- Un’altra feature offerta dalla applicazione è il salvataggio del testo di una nota in formato PDF
- Per la creazione e la scrittura su un documento PDF si è fatto uso della libreria «itextpdf»

Salvataggio in PDF

- Per consentire il salvataggio di files nell'external storage, nella RealMainActivity, vengono create delle cartelle
- Si ottiene dapprima una cartella di root dell'applicazione con il nome dell'app stessa
- Nella cartella di root viene creata una cartella «pdf», al cui interno verrà inserita la cartella di «Default», in cui si potranno salvare i files
- All'utente viene comunque data la possibilità di creare altre sottocartelle di «pdf», in cui salvare i propri files

Creazione cartelle per PDF

```
lateinit var rootDir : File
lateinit var pdfDir : File
lateinit var pdfDefaultDir : File
```

```
rootDir = getOutputDirectory()
pdfDir = File(rootDir, getString(R.string.pdf_dir)).apply { mkdir() }
pdfDefaultDir = File(pdfDir, getString(R.string.default_dir)).apply { mkdir() }
```

```
private fun getOutputDirectory(): File {
    val mediaDir = externalMediaDirs.firstOrNull()?.let {
        File(it, resources.getString(R.string.app_name)).apply { mkdirs() }
    }
    return if (mediaDir != null && mediaDir.exists())
        mediaDir else filesDir
}
```

Creazione del file PDF

- Nella TextResultActivity, alla pressione del FAB per il PDF, verrà mostrato un Dialog per la scelta della cartella di destinazione
- Si usa ancora il meccanismo delle Coroutines per andare a leggere dall'external storage quali sono le cartelle precedentemente create, così da passarle al Dialog
- Per la creazione del file PDF è stata creata una apposita classe: `PdfManager`

Creazione del file PDF

```
binding.fabPrintPdf.setOnClickListener {
    CoroutineScope(Dispatchers.IO).launch {
        val stringDirectoryList: MutableList<String> = mutableListOf()
        RealMainActivity.pdfDir.listFiles()?.forEach { stringDirectoryList.add(it.name) }
        stringDirectoryList.add(getString(R.string.new_dir))
        CoroutineScope(Dispatchers.Main).launch {
            val pdfDialog = MakeDirectoryDialog.getInstance()
            pdfDialog.setDirectoryList(stringDirectoryList)

            pdfDialog.setOnDirectorySelected { directory: String, title: String ->
                val text = textResult
                val dir = File(RealMainActivity.pdfDir, directory)
                CoroutineScope(Dispatchers.IO).launch {
                    dir.apply { mkdirs() }
                    PdfManager.transformToPdf(title, text, dir)
                    CoroutineScope(Dispatchers.Main).launch {
                        Toast.makeText(
                            this@TextResultActivity,
                            getString(R.string.pdf_saved),
                            Toast.LENGTH_SHORT
                        ).show()
                    }}}
            pdfDialog.setOnCancelListener {
                true
            }
            pdfDialog.show(supportFragmentManager, "DIALOGPDF")
        }}}
```

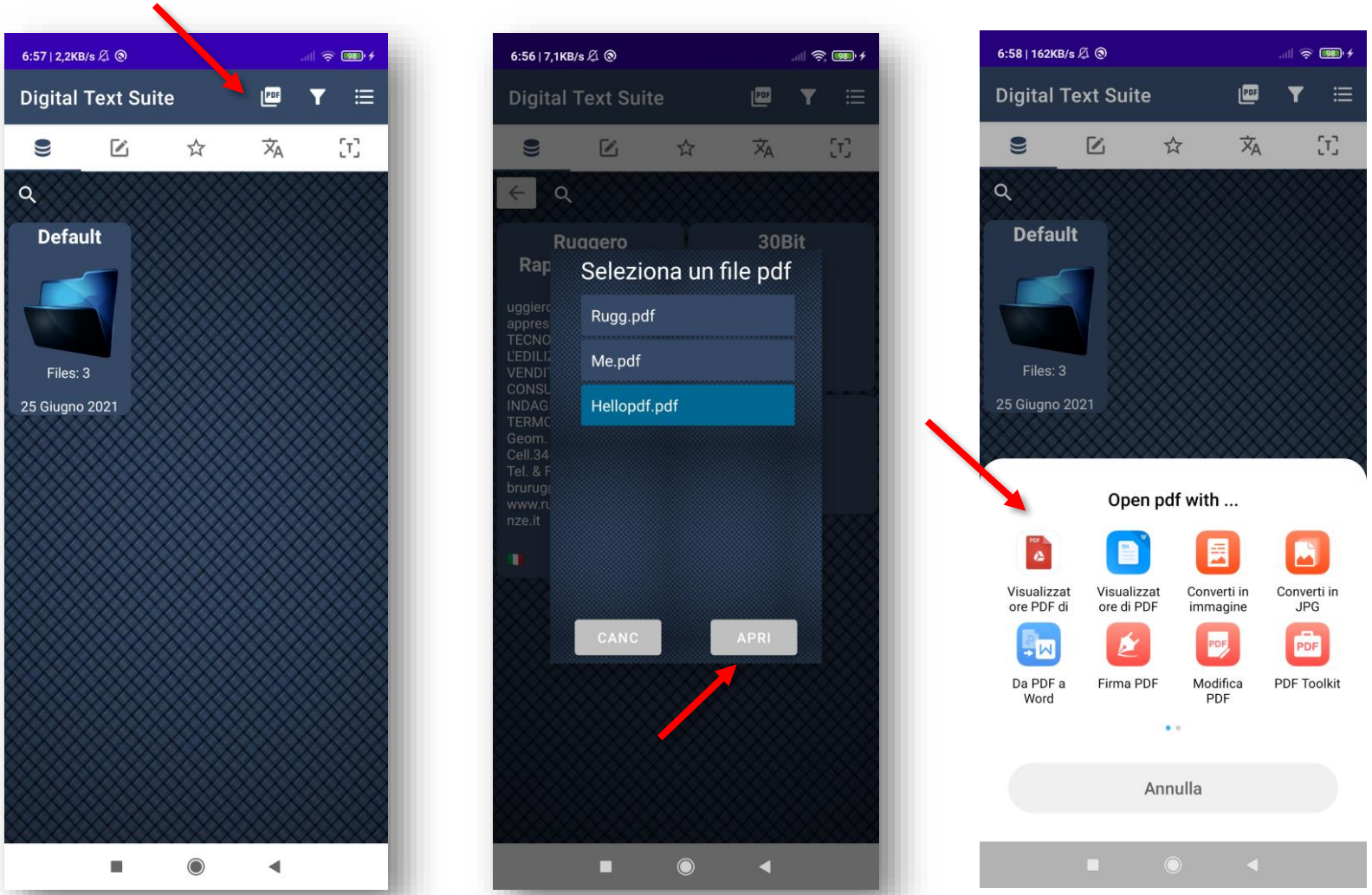
PdfManager

- Questa classe mette a disposizione il metodo `transformToPdf()`, che permette di creare un file pdf con un determinato nome in una determinata cartella, contenente il testo passato come ulteriore parametro
- Utilizza le API della libreria «itextpdf» per costruire un documento a partire da un `PdfDocument` aperto in scrittura
- Il testo viene scritto sul documento creando un oggetto di tipo Paragraph, il quale viene aggiunto al documento

PdfManager

```
fun transformToPdf(title: String, text: String, folder: File) {

    val fileLocation = File(folder, title + PDF_EXTENSION).canonicalPath

    // create an instance of PdfDocument at fileLocation location
    val pdfDocument = PdfDocument(PdfWriter(fileLocation))
    pdfDocument.defaultPageSize = PageSize.A4

    val document = Document(pdfDocument)
    val paragraph = Paragraph(text)
    paragraph.setFontSize(16f)
    paragraph.setTextAlignment(TextAlignment.LEFT)
    document.add(paragraph)

    // This will create a file at your fileLocation, specified while creating
    // PdfDocument instance
    document.close()
}
```

Come visualizzare i PDF



Come visualizzare i PDF

- Si può selezionare il file da visualizzare tramite il Dialog mostrato nella seconda delle figure precedenti, che al suo interno contiene una RecyclerView con tutti i files PDF creati con «DigitalTextSuite»
- Selezionato il file da aprire, viene lanciata una Intent con un Chooser, il quale permette di scegliere con quale applicazione aprire il file

Come visualizzare i PDF

```
CoroutineScope(Dispatchers.IO).launch {
    val rootDir = RealMainActivity.rootDir
    val listFiles = getPdfFilesFromRootDir(rootDir)
    CoroutineScope(Dispatchers.Main).launch {
        val dialog = SelectPdfDialog.getInstance(listFiles)
        dialog.show(parentFragmentManager, "SELECTPDFDIALOG")
    }
}

private fun getPdfFilesFromRootDir(rootDir: File): List<File> {
    return rootDir.walk().filter{
       it.extension == "pdf"
    }.toList()
}
```

```
intent.setDataAndTypeAndNormalize(data, "application/pdf")
intent.flags = Intent.FLAG_ACTIVITY_CLEAR_TOP
intent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
startActivity(Intent.createChooser(intent, "Open pdf with ..."))
```