Relazione Applicazioni Mobili

Autore: Emanuele Marzone 20049568

Informazioni iniziali

- **Repository Android (SSH):** git@gitlab.di.unipmn.it:Emarzone/tieniiltempo.git
- **Repository iOS (SSH):** git@gitlab.di.unipmn.it:Emarzone/app-ios.git
- **Demo:** Da svolgere su PC laboratorio
- **Credenziali di test:**
  - Caregiver 1: email: prova@gmail.com, password: Emanuele1
  - Caregiver 2: email: prova4@gmail.com, password: Corinna1
  - Utente 1: email: prova2@gmail.com, password: Mariorossi1 (associato caregiver 1)
  - Utente 2: email: prova3@gmail.com, password: Luigiverdi1 (associato caregiver 1)
  - Utente 3: email: prova5@gmail.com, password: Giovanni1 (associato caregiver 2)

Obbiettivo Applicazione Android

L'applicazione `Tieni il tempo` ha come obiettivo quello di fornire uno strumento utile ai **caregiver**, permettendo loro di assegnare attività quotidiane agli utenti e di monitorarne lo svolgimento.
Le attività sono suddivise in **sotto-attività sequenziali e parallele**, ognuna delle quali può includere una **descrizione**, un'**immagine di supporto** e un **commento**. Durante l'esecuzione, l'utente ha la possibilità di inviare **commenti con immagini** riferiti a specifiche sotto-task, così da facilitare la comunicazione e il supporto.

L'app integra inoltre una **chat con notifiche in tempo reale**, che permette a caregiver e utenti di rimanere sempre in contatto.
Al termine di ogni attività, il **caregiver** può inserire una **valutazione** e un **commento**, riconoscendo così l'impegno dell'utente e favorendo la motivazione, anche tramite un **sistema di premi**.

Questa soluzione rappresenta un **aiuto concreto** nella gestione delle attività quotidiane, soprattutto in quelle situazioni in cui non è sempre possibile la presenza diretta del caregiver.
L'app, infatti, si propone come una **guida semplice, intuitiva e veloce**, utile sia per **organizzare** che per **svolgere le attività in autonomia**.

Tabella funzionalità base (max 25/30)

Funzionalità	Sì/No/Parzialmente
1) Creare/modificare/cancellare attività e sotto-attività	Sì
2) Tempo esecuzione attività	Sì
3) Commento con foto ad una sotto-attività	Sì
4) Chat tra utenti	Sì
5) Più utenti per singolo caregiver	Sì

Funzionalità	Sì/No/Parzialmente
6) Ricezione notifiche	Sì
7) Invio localizzazione geografica	Sì

Tabella funzionalità avanzate (>25/30)

Funzionalità avanzata	Sì/No/Parzialmente
8) Sotto-attività in parallelo	Sì
9) Tempo esecuzione singole sotto-attività	Sì
10) Notifica al caregiver per superamento tempo	Sì
11) Inizio nuova attività tramite AlarmManager	Esonerato per creazione "FixTutorial" per IOS
12) Gamification (premi e ricompense)	Sì
13) Schermata statistiche	Sì

Strutture dati scelte

Per il progetto **"Tieni il tempo"** sono state definite diverse strutture dati principali, modellate in **Firebase Firestore**, ciascuna con campi specifici per supportare le funzionalità dell'app.

Utente (`User`)

Rappresenta sia i caregiver che gli utenti finali, distinguibili dal campo `userType`.

- `uid` : identificatore univoco dell'utente.
- `name` e `surname` : nome e cognome dell'utente.
- `email` : indirizzo email per login e comunicazioni.
- `userType` : indica se l'utente è **Caregiver** o **Utente**.

Attività (`Task`)

Ogni attività può essere composta da una o più sotto-attività e include informazioni sul tempo e sul completamento.

- `title` : titolo dell'attività.
- `estimatedDurationMinutes` : durata stimata in minuti.
- `status` : stato corrente dell'attività (es. "Completata").
- `startedAt` e `finishedAt` : timestamp di inizio e fine.
- `sortOrder` : campo utilizzato per ordinare automaticamente le attività.
- `userId` : identificatore dell'utente associato all'attività.
- `vote` : punteggio assegnato dal caregiver al completamento.

Sotto-attività (`subtask`)

Rappresentano le singole azioni che compongono un'attività.

- `description` : descrizione della sotto-attività.
- `comment` : commento associato.
- `createdAt`, `startedAt`, `finishedAt` : timestamp di creazione, inizio e fine.
- `imageUri` : contiene il percorso in cui è salvata l'immagine associata alla sotto-task
- `order` : ordine sequenziale tra le sotto-attività.
- `parallel` : indica se può essere svolta in parallelo ad altre sotto-attività.
- `position` : indica se richiede l'invio della posizione dell'utente.
- `status` : stato della sotto-attività (es. "Completata").
- `taskId` : riferimento all'attività padre.

Chat e Messaggi

Gestisce la comunicazione tra caregiver e utenti, con supporto a messaggi di testo, immagini e posizione.

- `id` : identificatore univoco dato dalla concatenazione degli id dei due utenti coinvolti.
- `messages` : lista dei messaggi, ciascuno con:
  - `senderId` : mittente del messaggio.
  - `text` : testo del messaggio.
  - `type` : tipo di messaggio ("TEXT", "LOCATION" o "IMAGE").
  - `imageUri` : eventuale immagine allegata.
  - `latitude` e `longitude` : coordinate geografiche se il messaggio è di tipo "LOCATION".
  - `timestamp` : momento di invio del messaggio.

Associazioni (`Association`)

Definisce le relazioni tra caregiver e utenti.

- `caregiverId` : identificatore del caregiver.
- `userId` : identificatore dell'utente associato al caregiver.

> Questa modellazione consente di gestire in modo flessibile tutte le funzionalità richieste, tra cui: gestione di attività e sotto-attività, commenti, voti, notifiche, chat e localizzazione, oltre al supporto a più utenti per un singolo caregiver.

Scelte implementative principali

Per prima cosa, ho scelto di suddividere il progetto in **package**, in modo da organizzare meglio le varie parti. In particolare, abbiamo:

- `auth`: contiene le activity dedicate alla schermata iniziale, login e registrazione.
- `caregiver`: contiene l'activity principale dei caregiver e le relative statistiche.
- `chat`: contiene le classi per modellare i dati, le activity e gli adapter.
- `profile`: contiene l'activity con le informazioni dei profili degli utenti dell'app.
- `task` e `subtask`: contengono sia le classi per modellare i dati sia le activity e gli adapter, gestendo le azioni sia dei caregiver sia degli utenti normali.
- `user`: contiene la classe per modellare i dati, l'activity principale degli utenti e le relative statistiche.

Scelte implementative funzionalità richieste

1. Creare/modificare/cancellare attività e sotto-attività

Queste funzionalità sono state implementate all'interno di `TaskListActivity.kt`, in particolare la creazione di una task e la sua modifica sono gestite all'interno di una sola funzione:

```
private fun showCreateTaskDialog(userId: String, taskToEdit: Task? = null) {
        val builder = AlertDialog.Builder(this)
        builder.setTitle(if (taskToEdit == null) "Crea nuova attività" else
"Modifica attività")

        val inflater = LayoutInflater.from(this)
        val dialogView = inflater.inflate(R.layout.dialog_create_task, null)
        val titleEditText = dialogView.findViewById<EditText>
(R.id.taskTitleEditText)
        val subtasksContainer = dialogView.findViewById<LinearLayout>
(R.id.subtasksContainer)
        val addSubtaskButton = dialogView.findViewById<Button>
(R.id.addSubtaskButton)
        val durationEditText = dialogView.findViewById<EditText>
(R.id.taskEstimatedDurationEditText)

        // Funzione helper per popolare le view delle sottotask, per evitare
codice duplicato
        val populateSubtasks = { documents:
List<com.google.firebase.firestore.DocumentSnapshot> ->
            for (doc in documents) {
                addSubtaskView(
                    subtasksContainer,
                    doc.getString("description") ?: "",
                    doc.getString("comment") ?: "",
                    doc.getString("imageUri"),
                    doc.getBoolean("parallel") ?: false,
                    doc.getBoolean("position") ?: false,
                    doc.getLong("order")?.toInt()
                )
            }
        }

        // Precompila dati esistenti se si sta modificando un'attività
        if (taskToEdit != null) {
            titleEditText.setText(taskToEdit.title)

durationEditText.setText(taskToEdit.estimatedDurationMinutes.toString())

            // Carica le sottotask esistenti.
            // Prima tenta la query ottimale che richiede un indice composito.
            db.collection("subtasks")
                .whereEqualTo("taskId", taskToEdit.id)
                .orderBy("order")
                .get()
```

```
                .addOnSuccessListener { result ->
                    // Successo: i documenti sono già ordinati da Firestore.
                    populateSubtasks(result.documents)
                }
                .addOnFailureListener { e ->
                    db.collection("subtasks")
                        .whereEqualTo("taskId", taskToEdit.id)
                        .get()
                        .addOnSuccessListener { result ->
                            // Ordiniamo manualmente la lista dei documenti in
base al campo "order", se l'ordinamento dalla query fallisce
                            val sortedDocs = result.documents.sortedBy {
it.getLong("order") ?: Long.MAX_VALUE }
                            populateSubtasks(sortedDocs)
                        }
                }
        } else {
            // Se è una nuova attività, aggiungo una sottotask vuota di default.
            addSubtaskView(subtasksContainer)
        }

        // Configurazione del pulsante per aggiungere nuove sottotask
        addSubtaskButton.setOnClickListener {
            addSubtaskView(subtasksContainer)
        }

        builder.setView(dialogView)

        // Pulsante "Salva"
        builder.setPositiveButton("Salva") { dialog, _ ->
            val title = titleEditText.text.toString().trim()
            if (title.isEmpty()) {
                Toast.makeText(this, "Il titolo è obbligatorio",
Toast.LENGTH_SHORT).show()
                return@setPositiveButton
            }

            val estimatedDuration = durationEditText.text.toString().toIntOrNull()
?: 10

            val taskRef = if (taskToEdit == null)
db.collection("tasks").document() else
db.collection("tasks").document(taskToEdit.id)

            if (taskToEdit == null) {
                // Logica per creare una NUOVA attività
                val taskData = hashMapOf(
                    "userId" to userId,
                    "title" to title,
                    "status" to "Da iniziare",
                    "estimatedDurationMinutes" to estimatedDuration,
                    "startedAt" to null,
                    "finishedAt" to null,
                    "vote" to null
```

```
                )
                // Calcola l'ordine di visualizzazione
                db.collection("tasks")
                    .whereEqualTo("userId", userId)
                    .get()
                    .addOnSuccessListener { tasks ->
                        val maxSortOrder = tasks.maxOfOrNull {
it.getLong("sortOrder") ?: 0 } ?: 0
                        taskData["sortOrder"] = maxSortOrder + 1

                        taskRef.set(taskData).addOnSuccessListener {
                            saveSubtasks(subtasksContainer, taskRef.id)
                            loadTasksForUser(userId)
                            Toast.makeText(this, "Attività salvata",
Toast.LENGTH_SHORT).show()
                        }
                    }
                    .addOnFailureListener {
                        Toast.makeText(this, "Errore nel calcolo dell'ordine",
Toast.LENGTH_SHORT).show()
                    }
            } else {
                // Logica per AGGIORNARE un'attività esistente
                db.collection("tasks").document(taskToEdit.id).get()
                    .addOnSuccessListener { snapshot ->
                        val currentStatus = snapshot.getString("status")
                        if (currentStatus != "Da iniziare") {
                            Toast.makeText(this, "Non puoi modificare un'attività
già iniziata", Toast.LENGTH_LONG).show()
                            return@addOnSuccessListener
                        }

                        val updates = mapOf(
                            "title" to title,
                            "estimatedDurationMinutes" to estimatedDuration
                        )

                        taskRef.update(updates).addOnSuccessListener {
                            saveSubtasks(subtasksContainer, taskRef.id)
                            loadTasksForUser(userId)
                            Toast.makeText(this, "Attività aggiornata",
Toast.LENGTH_SHORT).show()
                        }
                            .addOnFailureListener { e ->
                                Toast.makeText(this, "Errore durante
l'aggiornamento: ${e.message}", Toast.LENGTH_SHORT).show()
                            }
                    }
                    .addOnFailureListener { e ->
                        Toast.makeText(this, "Errore nel recupero stato:
${e.message}", Toast.LENGTH_SHORT).show()
                    }
            }
        }
```

```
        builder.setNegativeButton("Annulla") { dialog, _ -> dialog.dismiss() }
        builder.show()
    }
```

Questa funzione permette di mostrare una finestra di dialogo per la creazione e modifica di una task, al suo interno è presente il titolo della task, la durata stimata, un pulsante per aggiungere delle sottotask, le quali hanno i propri campi (descrizione, commento, immagine e posizione) e sono gestite da `addSubtaskView`. Quando creiamo una nuova task tutti i campi sono vuoti, mentre se ne modifichiamo una esistente (taskToEdit != null) prendiamo i dati della task e delle sottotask e precompiliamo i campi. Quando abbiamo finito di creare o modificare la task, possiamo salvarla cliccando sul pulsante apposito e i dati di quest'ultima e delle sue sottotask verranno usati per creare gli oggetti corrispondenti, infine il tutto viene caricato su firestore. Se la task è già stata iniziata dall'utente questa non può più essere modificata.

> Il codice di addSubtaskView è stato riportato nella sezione "3. Commento con foto ad una sotto-attività"

La cancellazione invece è gestita da quest'altra funzione:

```
private fun deleteTask(task: Task) {
        // Prima elimina tutte le sottotask associate e le loro immagini da
Firebase Storage
        db.collection("subtasks")
            .whereEqualTo("taskId", task.id)
            .get()
            .addOnSuccessListener { querySnapshot ->
                val batch = db.batch() //permette di salvare le operazioni da
svolgere ed effettuarle tutte in contemporanea (scritture e rimozioni)
                for (doc in querySnapshot.documents) {
                    // Elimina il file immagine associato da Firebase Storage
                    val imageUriString = doc.getString("imageUri")
                    imageUriString?.let { uriString ->
                        try {
                            val storageRefToDelete =
storage.getReferenceFromUrl(uriString)
                            storageRefToDelete.delete()
                                .addOnFailureListener { e ->
                                    // Logga l'errore, ma non blocca
l'eliminazione della sottotask
                                    e.printStackTrace()
                                }
                        } catch (e: Exception) {
                            e.printStackTrace()
                        }
                    }
                    batch.delete(doc.reference)
                }
                // Poi elimina l'attività stessa
                batch.delete(db.collection("tasks").document(task.id))
```

```
                batch.commit() // avvio lo svolgimento delle azioni raggruppate
                    .addOnSuccessListener {
                        taskList.remove(task)
                        taskAdapter.notifyDataSetChanged()
                        loadTasksForUser(userId)
                        Toast.makeText(this, "Attività e sottotask eliminate",
Toast.LENGTH_SHORT).show()
                    }
                    .addOnFailureListener { e ->
                        Toast.makeText(this, "Errore durante l'eliminazione:
${e.message}", Toast.LENGTH_SHORT).show()
                    }
            }
            .addOnFailureListener { e ->
                Toast.makeText(this, "Errore durante il recupero delle sottotask
per eliminazione: ${e.message}", Toast.LENGTH_SHORT).show()
            }
    }
```

Questa funzione è più esplicativa rispetto la precedente perchè deve svolgere solo la cancellazione delle attività. Per prima cosa andiamo ad eliminare dallo storage di firebase le immagini, successivamente salviamo sul batch l'operazione di cancellazione delle sottotask, una volta trovate tutte quelle associate ad una task, aggiungiamo anche la rimozione della task stessa al batch e poi facciamo il commit delle operazioni per svolgerle tutte in un colpo solo.

2. Tempo esecuzione attività e sotto-attività

Tempo attività:

Per gestire il tempo d'esecuzione di una attività ho deciso di sfruttare i campi `startedAt` e `finishedAt` definiti nella sua struttura dati. Questa operazione viene svolta all'interno del metodo `onBindViewHolder` dell'adapter della task che va a modificare i campi della view delle attività quando avvengono dei cambiamenti (questo metodo è presente sia nell'adapter delle task visualizzate dai caregiver sia da quelle visualizzate dagli utenti, riporto di seguito il metodo di quello del caregiver perchè la logica è uguale).

```
override fun onBindViewHolder(holder: TaskViewHolder, position: Int) {
        val task = tasks[position]
        holder.titleText.text = task.title

        // ometto parte del metodo per diminuire lo spazio occupato e focalizzarci
sul punto chiave della funzione richiesta
        .
        .
        .

        // una volta che lo stato di una task passa da "Da iniziare" a "In corso"
e infine a "completata"
        // si possono fare delle operazioni aggiuntive
        if (statusNormalized == "completata") {
```

```
            if (task.vote != null) {
                holder.voteButton.visibility = View.GONE
                holder.voteText.visibility = View.VISIBLE
                holder.voteText.text = "Voto: ${task.vote} stelle"
            } else {
                holder.voteButton.visibility = View.VISIBLE
                holder.voteButton.setImageResource(R.drawable.ic_star)

holder.voteButton.setColorFilter(holder.itemView.context.getColor(R.color.yellow))
                holder.voteButton.setOnClickListener {
                    onVoteClick(task)
                }
            }

            if (task.startedAt != null && task.finishedAt != null) {
                val durationSeconds = task.finishedAt!!.seconds -
task.startedAt!!.seconds
                val totalMinutes = (durationSeconds / 60).toInt()
                val estimated = task.estimatedDurationMinutes ?: 0

                holder.completionTimeText.visibility = View.VISIBLE
                holder.completionTimeText.text =
                    "Tempo impiegato:\n${totalMinutes} min su $estimated min
stimati"
            } else {
                holder.completionTimeText.visibility = View.GONE
            }
        } else {
            holder.voteButton.visibility = View.GONE
            holder.completionTimeText.visibility = View.GONE
        }



        val editable = statusNormalized == "da iniziare"
        holder.editButton.isEnabled = editable
        holder.editButton.alpha = if (editable) 1f else 0.3f
        if (editable) holder.editButton.setOnClickListener {
            onEditClick(task)
        }
        else holder.editButton.setOnClickListener(null)

        holder.deleteButton.setOnClickListener {
            onDeleteClick(task)
        }
    }
```

All'interno di onBindViewHolder dell'adapter, una volta che una task è competata prendendo i suoi campi startedAt e finishedAt, si fa la differenza in secondi, poi il risultato viene convertito in minuti, per una facilità di lettura, e inserito in una text view insieme al tempo stimato se presente (non è un campo obbligatorio, se omesso è stato fissato a 10 min).

> in questa funzione è presente anche la logica per mostrare il voto della task, quando gli viene assegnato dal caregiver (questo voto permette rispettando determinate condizioni di ottenere un premio riscattabile). Se non è ancora stato assegnato, in questo metodo viene mostrato un pulsante a stella per far votare il caregiver, dal lato dell'utente viene solo reso visibile o meno il voto.

Tempo sotto-attività

Anche per le sottotask ho sfruttato i campi `startedAt` e `finishedAt` presenti nelle loro strutture dati per calcolare il tempo d'esecuzione; questi campi vengono aggiornati quando viene premuto il pulsante per iniziare una sottotask e quando viene premuto quello per terminarla. Se la sotto-attività iniziata è la prima della lista fa cambiare anche lo stato alla task di cui fa parte e aggiunge il timestamp d'inizio anche a lei (questo aggiornamento viene fatto dalla funzione `updateTaskStatusOnSubtaskStart`). Quando sono finite tutte le sottotask viene reso cliccabile il tasto per terminare l'attività e questo mi permette di salvare anche il tempo di terminazione e cambiargli lo stato a completata.

Le funzioni che aggiornano i campi startedAt e finishedAt per le sottoTask sono `handleStartSubtask` e `handleEndSubtask`. Mentre la logica che calcola e mostra i tempi d'esecuzione di ogni sottotask è all'interno dell'`onBindViewHolder` nell'adapter delle sottotask. Non riporto i codici perchè le funzioni di aggiornamento dei campi sono semplici query a firestore, mentre la logica per calcolare in tempo effettivo è la stessa di quella mostrata precedentemente.

3. Commento con foto ad una sotto-attività

Essendo che il commento con foto poteva essere aggiunto sia dal caregiver che dall'utente normale riporto di seguito le due implementazioni svolte.

Implementazione caregiver:

Il commento con immagine è stato pensato come aggiunta diretta nella creazione della sottotask; infatti se la sottotask "prendi il latte" nella task "Fare la spesa" ha bisogno di un livello di specificità in più, si può compilare il campo commento inserendo per esempio "Prendi il latte Parmalat, se hai difficoltà fatti aiutare da un commesso" e nella foto (opzionalmente) inserire un immagine di riferimento.

```
private fun addSubtaskView(
        container: LinearLayout,
        description: String = "",
        comment: String = "", // questo campo fa parte di questa funzione
richiesta
        imageUri: String? = null, // questo campo fa parte di questa funzione
richiesta
        isParallel: Boolean = false,
        isPosition: Boolean = false,
        order: Int? = null
    ) {
        val inflater = LayoutInflater.from(this)
        val subtaskView = inflater.inflate(R.layout.item_subtask_input, container,
false)

        val descEditText = subtaskView.findViewById<EditText>
```

```
(R.id.subtaskDescriptionEditText)
        val commentEditText = subtaskView.findViewById<EditText>
(R.id.subtaskCommentEditText)
        val deleteButton = subtaskView.findViewById<ImageButton>
(R.id.deleteSubtaskButton)
        val loadImageButton = subtaskView.findViewById<ImageButton>
(R.id.loadImageButton)
        val imageView = subtaskView.findViewById<ImageView>(R.id.subtaskImageView)
        val parallelCheckbox = subtaskView.findViewById<CheckBox>
(R.id.subtaskParallelCheckbox)
        val positionCheckBox = subtaskView.findViewById<CheckBox>
(R.id.shareLocationCheckBox)

        descEditText.setText(description)
        commentEditText.setText(comment)
        parallelCheckbox.isChecked = isParallel
        positionCheckBox.isChecked = isPosition

        // Carica l'immagine se presente
        if (imageUri != null && imageUri.isNotEmpty()) {
            android.util.Log.d("TaskListActivity", "Caricamento immagine:
$imageUri")
            loadImageFromUrl(imageUri, imageView)
        } else {
            imageView.visibility = View.GONE
            imageView.tag = null
        }

        deleteButton.setOnClickListener {
            val currentImageUri = imageView.tag as? String
            currentImageUri?.let { uriString ->
                try {
                    val storageRefToDelete =
storage.getReferenceFromUrl(uriString)
                    storageRefToDelete.delete()
                        .addOnSuccessListener {
                            Toast.makeText(this, "Immagine associata eliminata da
Storage.", Toast.LENGTH_SHORT).show()
                        }
                        .addOnFailureListener { e ->
                            Toast.makeText(this, "Errore nell'eliminazione
dell'immagine da Storage: ${e.message}", Toast.LENGTH_LONG).show()
                            e.printStackTrace()
                        }
                } catch (e: Exception) {
                    e.printStackTrace()
                    Toast.makeText(this, "Errore nel riferimento all'immagine di
Storage.", Toast.LENGTH_SHORT).show()
                }
            }
            container.removeView(subtaskView)
        }

        loadImageButton.setOnClickListener {
```

```
            imagePickerCallback = { uri ->
                loadImageFromUrl(uri.toString(), imageView)
            }
            imagePickerLauncher.launch("image/*")
        }

        subtaskView.tag = order
        container.addView(subtaskView)
    }
```

In questa funzione di creazione di una sotto task, tra i campi compilabili obbligatori (come la descrizione) ho aggiunto la possibilità di inserire un commento e/o un immagine in maniera totalmente opzionale da parte del caregiver. Il commento se presente viene preso dall'editText, mentre l'immagine si può caricare cliccando un pulsante e la sua logica è gestita da `imagePickerLauncher` che apre il launcher per la selezione di un immagine, richiama il metodo `uploadImageToFirebaseStorage` che carica l'immagine sullo storage di firestore e poi `imagePickerCallback` serve per ottenere l'uri dell'immagine caricata da inserire nel campo della sottotask e per visualizzarla a schermo

> Sono presenti anche i campi per indicare se la task è parallela e se deve richiedere l'invio della posizione all'utente

Implementazione utente:

L'utente invece può commentare una sottotask solo quando sta svolgendo l'attività in cui è contenuta, quindi nella schermata in cui vengono mostrate è presente anche un pulsante in basso a destra che permette la selezione di una sottotask e fa apparire una finestra in cui commentarla e inserire un immagine (`showCommentImageDialog`)

```
private fun showCommentImageDialog(subtask: Subtask, taskTitle: String) {
        uploadedImageUrl = null // Resetta l'URL precedente

        val builder = AlertDialog.Builder(this)
        builder.setTitle("Commenta: ${subtask.description}")

        val inflater = this.layoutInflater
        val dialogView = inflater.inflate(R.layout.dialog_comment_image, null)
        val commentInput = dialogView.findViewById<EditText>(R.id.commentEditText)
        val selectImageButton = dialogView.findViewById<Button>
(R.id.selectImageButton)
        dialogTextView = dialogView.findViewById(R.id.imageSelectedTextView)

        builder.setView(dialogView)

        selectImageButton.setOnClickListener {
            // Definisce cosa fare una volta ottenuto l'URL di download
            imagePickerCallback = { downloadUri ->
                uploadedImageUrl = downloadUri.toString()
                dialogTextView.text = "Immagine caricata!"
                dialogTextView.visibility = View.VISIBLE
            }
```

```
            // Avvia il selettore
            imagePickerLauncher.launch("image/*")
        }

        builder.setPositiveButton("Invia") { dialog, _ ->
            val commentText = commentInput.text.toString().trim()

            if (commentText.isBlank()) {
                Toast.makeText(this, "Devi inserire almeno un commento",
Toast.LENGTH_SHORT).show()
            } else {
                // L'immagine è già stata caricata, ora inviamo solo il messaggio
                val header = "Commento a \"$taskTitle\" per
\"${subtask.description}\":"
                val messageText = "$header\n$commentText"
                findCaregiverAndSendMessage(messageText, uploadedImageUrl)
            }
        }
        builder.setNegativeButton("Annulla", null)
        builder.show()
    }
```

Anche in questa funzione il commento viene preso da un editText, mentre l'immagine viene gestita con la stessa logica di quella usata nel caregiver. La differenza è che il commento scritto dall'utente viene poi inviato nella chat con il suo caregiver, così da li possono direttamente iniziare una conversazione al riguardo.

4. Chat tra utenti

La chat è stata implementata sfruttando le associazioni, infatti un utente visualizzerà solo la chat con il proprio caregiver, mentre i caregiver potranno vedere tutte le chat con gli utenti che stanno seguendo. Dopo aver creato una chat, ho aggiunto anche i messaggi che sono associati ad essa e ho sviluppato dei file xml, sia per i "messaggi inviati" sia per quelli "ricevuti" per rendere più gradevole l'interfaccia. Inoltre ho aggiunto anche qui la possibilità di inviare immagini (sfruttando la logica già vista in precedenza) e la posizione (ho dovuto richiedere i permessi e poi ho inviato la latitudine e la longitudine del dispositivo, quando il messaggio viene cliccato si viene trasportati sull'app di Google Maps).

Questa funzione è contenuta nel package `chat` ed è stata sviluppata implementando le strutture dati `Chat` e `Message`, i loro adapter `ChatAdapter` e `MessageAdapter` e infine le due activity `ChatActivity` e `MessageActivity`.

Riporto di seguito solo i codici più significativi di questa funzione richiesta:

```
class ChatActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        .
        .
        .
        loadChatTitle() // aggiunge il titolo a video di "Chat di NomeUtente"
        loadAssociatedChats() // carica la recyclerView delle chat dell'utente
```

```
        setupBottomNavigation()
    }

      private fun loadChatsForUsers(userIds: List<String>) {
        val currentUserId = auth.currentUser?.uid ?: return

        if (userIds.isEmpty()) {
            chatRecyclerView.adapter = ChatAdapter(emptyList())
            chatRecyclerView.visibility = View.GONE
            emptyChatTextView.visibility = View.VISIBLE
            return
        }

        val chats = mutableListOf<Chat>()
        val loadedCount = 0

        // listener per aggiornare la UI in tempo reale
        val chatsCollection = db.collection("chats")

        for (userId in userIds) {
            // trovo i dati dell'utente (nome, cognome) per visualizzarlo nella
chat
            db.collection("users").document(userId).get().addOnSuccessListener {
userDoc ->
                val firstName = userDoc.getString("name") ?: "Utente"
                val lastName = userDoc.getString("surname") ?: ""
                val fullName = if (lastName.isNotBlank()) "$firstName $lastName"
else firstName

                // Calcola l'ID della chat room
                val chatRoomId = getChatRoomId(currentUserId, userId)

                // prendo l'ultimo messaggio in quella chat room
                chatsCollection.document(chatRoomId).collection("messages")
                    .orderBy("timestamp",
com.google.firebase.firestore.Query.Direction.DESCENDING)
                    .limit(1)
                    .addSnapshotListener { snapshot, error ->
                        if (error != null) {
                            // In caso di errore, viene mostrato un messaggio
                            updateChatList(chats, Chat(userId, fullName, "Errore
caricamento"), userIds.size)
                            return@addSnapshotListener // è un listener in tempo
reale di firestore, così quando avviene una modifica lui riesegue il codice e
aggiorna la ui senza ricaricarla manualmente
                        }

                        val lastMessage = if (snapshot != null &&
!snapshot.isEmpty) {
                            snapshot.documents[0].getString("text") ?: "..."
                        } else {
                            "Nessun messaggio" // Testo per chat vuote
                        }
```

```
                        updateChatList(chats, Chat(userId, fullName, lastMessage),
userIds.size)
                    }
            }.addOnFailureListener {
                updateChatList(chats, null, userIds.size) // in caso di errore non
aggiungiamo nessuna chat
            }
        }
    }

    private fun updateChatList(chats: MutableList<Chat>, newOrUpdatedChat: Chat?,
totalSize: Int) {
        // Rimuovo la vecchia versione della chat se esiste, per evitare duplicati
        if (newOrUpdatedChat != null) {
            chats.removeAll { it.userId == newOrUpdatedChat.userId }
            chats.add(newOrUpdatedChat)
        }

        // Aggiorna l'adapter solo quando abbiamo informazioni per tutte le chat
        if (chats.size >= totalSize) {
            // Ordina le chat per nome
            chats.sortBy { it.username }

            chatAdapter = ChatAdapter(chats)
            chatRecyclerView.adapter = chatAdapter

            if (chats.isEmpty()) {
                chatRecyclerView.visibility = View.GONE
                emptyChatTextView.visibility = View.VISIBLE
            } else {
                chatRecyclerView.visibility = View.VISIBLE
                emptyChatTextView.visibility = View.GONE
            }
        }
    }
}
```

in `loadChatsForUsers` prendiamo in input una lista di IdUtenti, così facendo creiamo un elemento nella recyclerView per ogni utente con cui si può chattare. Usando gli id ottengo anche il loro nome e cognome così da poterlo visualizzare e poi facendo una query sui messaggi della chat tra l'utente corrente e l'utente associato prendo l'ultimo messaggio inviato e lo mostro nella cardview sotto al nome. Questi tre paramatrei sono inviati poi a `updateChatList` che li usa per aggiornare la UI e ordinare le chat per nome.

```
class MessageActivity : AppCompatActivity() {
    .
    .
    .
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_message)
```

```
        // Recupero i dati e verifico la validità
        chatWithUserId = intent.getStringExtra("USER_ID") ?: ""
        val chatWithUserName = intent.getStringExtra("USER_NAME") ?: "Utente"

        if (chatWithUserId.isEmpty() || currentUserId.isEmpty()) {
            Toast.makeText(this, "Errore: utente non valido.",
Toast.LENGTH_SHORT).show()
            finish()
            return
        }

        // Determino l'ID della chat room e la inizializzo
        chatRoomId = getChatRoomId(currentUserId, chatWithUserId)
        setupUI(chatWithUserName)

        // Imposto il listener per inviare i messaggi
        sendButton.setOnClickListener {
            val messageText = messageInput.text.toString().trim()
            if (messageText.isNotEmpty()) {
                val message = Message(senderId = currentUserId, text =
messageText, type = MessageType.TEXT)
                sendMessage(message)
            }
        }

        sendAttachmentButton = findViewById(R.id.sendAttachmentButton)

        sendAttachmentButton.setOnClickListener {
            showAttachmentDialog()
        }

        createNotificationChannel()
        listenForMessages()
    }


    // logica d'invio messaggi
    private fun sendMessage(message: Message) {
        db.collection("chats").document(chatRoomId).collection("messages")
            .add(message)
            .addOnSuccessListener {
                if (message.type == MessageType.TEXT) {
                    messageInput.text.clear()
                }
            }
            .addOnFailureListener { e ->
                Toast.makeText(this, "Errore durante l'invio: ${e.message}",
Toast.LENGTH_SHORT).show()
            }
    }



    private var isInitialLoad = true
```

```
    private fun listenForMessages() {
         messageListener =
db.collection("chats").document(chatRoomId).collection("messages")
            .orderBy("timestamp", Query.Direction.ASCENDING)
            .addSnapshotListener { snapshots, error ->
                if (error != null) return@addSnapshotListener
                if (snapshots == null) return@addSnapshotListener

                val newMessages = snapshots.documentChanges.filter {
                    it.type ==
com.google.firebase.firestore.DocumentChange.Type.ADDED
                }.map {
                    it.document.toObject(Message::class.java)
                }

                if (isInitialLoad) {
                    // Primo caricamento: aggiungo tutti i messaggi senza
notificare
                    messageList.clear()
                    messageList.addAll(snapshots.toObjects(Message::class.java))
                    isInitialLoad = false
                } else {
                    // Caricamenti successivi: aggiungo solo i nuovi messaggi
                    messageList.addAll(newMessages)

                    if (chatRoomId != currentOpenChatId) {
                        val incomingMessages = newMessages.filter { it.senderId !=
currentUserId }
                        for (msg in incomingMessages) {
                            db.collection("users").document(msg.senderId).get()
                                .addOnSuccessListener { doc ->
                                    val senderName = doc.getString("name") ?:
"Utente"
                                    val senderSurname = doc.getString("surname")
?: "Chat"
                                    val userName = "$senderName $senderSurname"
                                    showNotification(userName, msg.text)
                                }
                                .addOnFailureListener {
                                    showNotification("Nuovo messaggio", msg.text)
                                }
                        }
                    }
                }

                messageAdapter.notifyDataSetChanged()  // l'adapter legge la lista
dei messaggi aggiornata
                messagesRecyclerView.scrollToPosition(messageList.size - 1)
            }
    }
}
```

Nel metodo on create sfrutto i valori che vengono passati dall'intent precedente per usarli per popolare il titolo della pagina in cui ci si può scambiare i messaggi, successivamente aggiorno la UI, imposto i listener, creo il canale per le notifiche e poi mi metto in ascolto sui nuovi messaggi in arrivo. Proprio nella funzione `listenForMessages` *creo degli snapshotListeners sui messaggi della chat così da avere il tutto sempre aggiornato in automatico; in più una volta aggiunto un nuovo messaggio aggiorno la lista con cui ho creato l'adapter e usando* `notifyDataSetChanged` *viene ricaricata la lista dei messaggi e visualizzata. Se è il primo caricamento (quando l'utente apre la chat) carico i messaggi e non mando notifiche, se invece è un caricamento successivo e la chat non è aperta invio delle notifiche push con* `showNotification`*. Per inviare un messaggio invece uso* `sendMessage` *che prende un messaggio in input e lo aggiunge alla lista dei messaggi della chat corrente, l'id univoco della chat viene ottenuto grazie a* `getChatRoomId` *che fa un controllo sugli id degli utenti e restituisce l'id univoco composto, (se il messaggio è un testo pulisce anche l'editText).*

5. Più utenti per singolo caregiver

Per implementare l'associazione di un caregiver a uno o più utenti, ho creato dentro la classe `CaregiverMainActivity` una funzione per mostrare la finestra di dialogo in cui inserire l'ID dell'utente e poi quando viene premuto il pulsante per confermare chiamo il metodo `checkAndAssociateUser` a cui passo l'id e controllo se l'associazione è effettuabile (l'utente non deve essere già associato ad altri)

```
 private fun checkAndAssociateUser(userId: String) {
        val caregiverId = auth.currentUser?.uid ?: return

        // Controlla se esiste un documento user con quell'ID e se è un utente
(non caregiver)
        db.collection("users").document(userId).get()
            .addOnSuccessListener { userDoc ->
                if (!userDoc.exists()) {
                    Toast.makeText(this, "Utente non trovato.",
Toast.LENGTH_SHORT).show()
                    return@addOnSuccessListener //listener per eseguire le
operazioni in maniera asincrona
                }

                val role = userDoc.getString("userType") ?: ""
                if (role != "Utente") {
                    Toast.makeText(this, "L'ID inserito non appartiene a un
utente.", Toast.LENGTH_SHORT).show()
                    return@addOnSuccessListener
                }

                db.collection("associations")
                    .whereEqualTo("userId", userId)
                    .get()
                    .addOnSuccessListener { documents ->
                        if (!documents.isEmpty) {
                            Toast.makeText(this, "Utente già associato ad un
caregiver.", Toast.LENGTH_SHORT).show()
                        } else {
                            // Crea nuova associazione
                            val association = hashMapOf(
```

```
                                "caregiverId" to caregiverId,
                                "userId" to userId
                            )
                            db.collection("associations").add(association)
                                .addOnSuccessListener {
                                    Toast.makeText(this, "Utente associato con
successo!", Toast.LENGTH_SHORT).show()
                                    loadAssociatedUsers() // aggiorna lista
                                }
                                .addOnFailureListener {
                                    Toast.makeText(this, "Errore durante
l'associazione.", Toast.LENGTH_SHORT).show()
                                }
                        }
                    }
                    .addOnFailureListener {
                        Toast.makeText(this, "Errore durante il controllo.",
Toast.LENGTH_SHORT).show()
                    }
            }
            .addOnFailureListener {
                Toast.makeText(this, "Errore nel recupero dati utente.",
Toast.LENGTH_SHORT).show()
            }
    }
```

Grazie a questo codice controllo se l'id inserito esiste ed è quello di un utente, dopodiche procedo a verificare se ci sono già associazioni con quell'id e se non sono presenti ne creo una nuova. Così facendo ho implementato l'associazione Caregiver-Utente, controllando che il dato inserito sia corretto e potendo associare più utenti ad uno stesso caregiver se questi sono disponibili.

6. Ricezione notifiche e Notifica al caregiver per superamento tempo

Per implementare questa funzione ho scelto di usare dei listener sul database di firestore che fanno partire le notifiche quando osservano dei cambiamenti, ad esempio nelle chat quando arriva un nuovo messaggio e nelle activity in corso quando queste superano il tempo stimato.

Notifiche per la chat

Per le notifiche dei messaggi ricevuti in chat ho dovuto, richiedere i permessi di invio delle notifiche, successivamente ho creato un canale per le notifiche con `createNotificationChannel` e poi ho implementato una funzione per inviare le notifiche (`showNotification`). Quest'ultima viene richiamata quando la chat non è aperta e viene ricevuto un nuovo messaggio in `listenForMessages` (codice riportato in precedenza), essendo che in questa funzione c'è uno snapshotListener, anche quando l'activity non è in primo piano, quando ci sono aggiornamenti su firestore, arrivano le notifiche. Inoltre ho eliminato le notifiche quando siamo "online" sulla chat andando ad effettuare un controllo sfruttando un companion object (`currentOpenChatId`), che non è altro che un elemento statico comune per tutte le istanze di una classe in kotlin, il quale diventa diverso da null solo se apriamo la chat.

```
    private fun createNotificationChannel() {
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
                val name = "Chat Notifications"
                val descriptionText = "Notifiche per i messaggi ricevuti"
                val importance = NotificationManager.IMPORTANCE_HIGH
                val channel = NotificationChannel(CHANNEL_ID, name, importance).apply
    {
                    description = descriptionText
                }
                val notificationManager: NotificationManager =
                    getSystemService(NotificationManager::class.java)
                notificationManager.createNotificationChannel(channel)
            }
        }

    @SuppressLint("MissingPermission", "NotificationPermission", "ObsoleteSdkInt")
        private fun showNotification(senderName: String, messageText: String) {
            val notificationId = chatRoomId.hashCode()

            val notification = NotificationCompat.Builder(this, CHANNEL_ID)
                .setSmallIcon(R.drawable.ic_chat)
                .setContentTitle(senderName)
                .setContentText(messageText)
                .setPriority(NotificationCompat.PRIORITY_HIGH)
                .setAutoCancel(true)
                .build()

            with(NotificationManagerCompat.from(this)) {
                notify(notificationId, notification)
            }
        }
```

Notifiche al caregiver quando un utente impiega più tempo del previsto

Per questo tipo di notifiche ho scelto un approccio leggermente diverso, infatti ho inserito, quando carico le task associate ad un utente, una funzione che controlla se tra quelle in corso qualcuna ha superato la durata stimata, se ciò succede viene creato un canale delle notifiche e viene inviata la notifica per aggiornare il caregiver. Essendo che questa logica viene fatta partire quando il caregiver apre le task di un utente la notifica è limitata a quell'activity, perchè non c'è un controllore che periodicamente verifica la differenza tra durata effettiva e stimata delle task. Ma questo non è un problema perchè così facendo il caregiver viene subito avvisato della durata superiore quando controlla le attività.

```
    @SuppressLint("MissingPermission")
    private fun showTaskOverdueNotification(taskTitle: String) {
        val builder = NotificationCompat.Builder(this, "TASK_OVERDUE_CHANNEL")
            .setSmallIcon(R.drawable.ic_stats)
            .setContentTitle("Task in ritardo")
            .setContentText("La task \"$taskTitle\" ha superato la durata
stimata!")
```

```
            .setPriority(NotificationCompat.PRIORITY_HIGH)
            .setAutoCancel(true)

        with(NotificationManagerCompat.from(this)) {
            notify(System.currentTimeMillis().toInt(), builder.build()) // id
unico
        }
    }

    private fun checkTaskOverdue(task: Task) {
        val startedAt = task.startedAt ?: return
        val estimatedMillis = task.estimatedDurationMinutes * 60 * 1000L
        val currentTime = System.currentTimeMillis()

        if (currentTime > startedAt.toDate().time + estimatedMillis && task.status
!= "Completata") {
            showTaskOverdueNotification(task.title)
        }
    }
```

Ho omesso la creazione del canale delle notifiche perchè uguale.

7. Invio localizzazione geografica

Per l'invio della localizzazione geografica ho dovuto richiedere i permessi di geolocalizzazione all'utente (`requestLocationPermissionLauncher`), installare i google play services e inizializzare un launcher per usarli (`fusedLocationClient`). Quando richiedo di inviare la posizione, mi salvo latitudine e longitudine che inserisco in un messaggio che inoltro alla chat tra utente e caregiver. Essendo che cliccando sul messaggio non apro una mappa interna all'applicazione, ma faccio aprire Google Maps, non ho avuto bisogno delle sue API; così ottengo anche tutti i vantaggi dell'app di google tra cui le indicazioni stradali.

> Non ho usato le API di Google Maps, ma ho usato le API di Fused Location che sono sempre state sviluppate da Google, ma permettono di sfruttare i google play services per ottenere le coordinate del dispositivo in maniera semplice e ottimizzata.

```
    // Client per la geolocalizzazione
    private val fusedLocationClient by lazy {
        LocationServices.getFusedLocationProviderClient(this)
    }

    // Launcher per la richiesta dei permessi di localizzazione
    private val requestLocationPermissionLauncher =
registerForActivityResult(ActivityResultContracts.RequestPermission()) {
isGranted: Boolean ->
        if (isGranted) {
            sendLocationNow()
        } else {
            Toast.makeText(this, "Permesso di localizzazione negato.",
Toast.LENGTH_SHORT).show()
        }
```

```
    }

    private fun checkLocationPermissionAndSend() {
        when {
            ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED ->
{
                Toast.makeText(this, "Acquisizione posizione...",
Toast.LENGTH_SHORT).show()
                sendLocationNow()
            }
            else -> {

requestLocationPermissionLauncher.launch(Manifest.permission.ACCESS_FINE_LOCATION)
            }
        }
    }

    private fun sendLocationNow() {
        try {

fusedLocationClient.getCurrentLocation(Priority.PRIORITY_HIGH_ACCURACY,
CancellationTokenSource().token)
                .addOnSuccessListener { location ->
                    if (location != null) {
                        //restituisce null o la prima occorrezza nella lista che
rispetta le condizioni tra {}
                        val subtaskInProgress = subtasks.firstOrNull { it.position
&& it.status == "In corso" }
                        val description = if (subtaskInProgress != null) {
                            "Posizione attuale per la sottotask:
\"${subtaskInProgress.description}\""
                        } else {
                            "La mia posizione attuale"
                        }
                        findCaregiverAndSendLocation(description,
location.latitude, location.longitude)
                    } else {
                        Toast.makeText(this, "Impossibile ottenere la posizione.
Attiva il GPS.", Toast.LENGTH_LONG).show()
                    }
                }
        } catch (e: SecurityException) {
            Toast.makeText(this, "Errore di sicurezza. Attiva il GPS.",
Toast.LENGTH_LONG).show()
        }
    }
```

Questi pezzi di codice sono gli elementi principali della richiesta e invio della posizione geografica. Con `requestLocationPermissionLauncher` *riesco a richiedere e controllare se abbiamo i permessi (*`checkLocationPermissionAndSend`*), dopodichè in* `sendLocationNow` *usiamo il client di fused location per ottenere la posizione corrente con accuratezza alta e se la richiesta va bene, cerco il caregiver associato*

all'utente, calcolo l'id univoco della loro chat e invio il messaggio con i dati della posizione. Così facendo il caregiver riceverà la notifica, andrà sulla chat e cliccando sul messaggio verrà trasportato su google maps in cui potra ottenere anche le indicazioni verso l'utente.

> La stessa logica è stata riportata anche in MessageActivity per poter inviare la posizione direttamente dalla chat, oltre che dalla pagina di svolgimento delle subtask

8. Sotto-attività in parallelo

Per poter eseguire anche le attività in parallelo oltre a quelle in sequenza, ho implementato una logica di controllo sulla possibilità d'inizio delle sottotask. L'idea alla base è avere un campo nella creazione delle sottotask che indica che è parallela e permettere a tutte le sottotask parallele nello stesso blocco di essere eseguite (quindi o tra l'inizio e la prima sottotask sequenziale o tra una sottotask sequenziale e l'altra). Ho scelto di permettere a solo le sottotask in un unico blocco di poter essere eseguite contemporaneamente per una facilità di individuazione visiva, infatti se per esempio avessimo una sottoativitià sequenziale tra quelle parallele, una volta terminate le sottoativitià prima di essa quella, diventa eseguibile e potrebbe portare confusione nell'utente, dividendole in blocchi, invece, diventa più intuitivo e diretto.

Riporto di seguito la logica di controllo per l'esecuzione delle sottattivita in sequenza e in parallelo

```
private fun canStartSubtask(subtask: Subtask, position: Int): Boolean {
        if (subtask.status != "Da iniziare") return false

        if (subtask.parallel) {
            // PARALLELA: può essere avviata se:
            // 1. Tutte le sequenziali precedenti sono completate
            // 2. Non c'è nessuna sequenziale in corso (in qualsiasi posizione)

            val allPrevSequentialsCompleted = subtasks
                .take(position)
                .filter { !it.parallel }
                .all { it.status == "Completata" }

            val noSequentialInProgress = subtasks
                .none { !it.parallel && it.status == "In corso" }

            return allPrevSequentialsCompleted && noSequentialInProgress

        } else {
            // SEQUENZIALE: può essere avviata se:
            // 1. Nessuna altra subtask è in corso
            // 2. Tutte le subtask precedenti sono completate (parallele E
sequenziali)

            val anyInProgress = subtasks.any { it.status == "In corso" }
            if (anyInProgress) return false

            val allPrevCompleted = subtasks
                .take(position)
                .all { it.status == "Completata" }
```

```
            return allPrevCompleted
        }
    }
```

L'idea di questa funzione è semplice, bisogna controllare lo stato della sottotask e il tipo. Se la sottotask ha come stato "Da iniziare" allora si può valutare se può essere resa eseguibile o meno. Capito ciò si entra in un if-else in cui, eseguo una logica se la sottotask è parallela o un'altra se è sequenziale. Se la sotto-attività è parallela devo controllare se prima e dopo non ci sono sotto-attività sequenziali in corso (devono essere "completate" quelle prima e non ce ne devono essere "In corso" dopo) se ciò è soddisfatto tutte le sottotask parallele in quel blocco diventano eseguibili. Per quelle sequenziali basta controllare che non ci sia una sottotask prima ancora in corso e che siano tutte con stato "Completata".

9. Gamification

Per rendere più coinvolgente il completamento delle task ho pensato di introdurre il concetto di premi ogni qual volta una task viene valutata 5 stelle. Così facendo se l'utente finisce la task nel tempo prestabilito e il caregiver decide di valutarlo a pieni voti, l'utente otterrà la possibilità di riscattare il premio. Questo concetto viene descritto nella schermata delle statistiche dell'utente. I premi sono cumulabili e la tipologia di premio può essere scelta di volta in volta di comune accordo tra le parti (ad esempio si mettono d'accordo che con X premi guadagnati li può trasformare in un gioco per una console, oppure riscatta un premio diretto e gli prende un gelato, etc...). Una volta scelto il premio il caregiver, nella stessa schermata in cui può aggiungere, modificare, eliminare e votare le task può anche riscattare i premi associati all'utente una volta che vengono consegnati.

> La logica per rendere visibile o meno il pulsante del voto di una task è stata riportata precedentemente nel codice della sezione 2. Tempo attività e sotto-attività

Riporto di seguito le funzioni per votare una task e riscattare un premio che sono all'interno di `TaskListActivity`.

```
private fun showVoteDialog(task: Task) {
        val builder = AlertDialog.Builder(this)
        builder.setTitle("Dai un voto all'attività")

        val container = LinearLayout(this).apply {
            orientation = LinearLayout.VERTICAL
            setPadding(50, 30, 50, 10)
        }

        // RatingBar centrato
        val ratingContainer = FrameLayout(this)
        val ratingBar = RatingBar(this, null, android.R.attr.ratingBarStyle).apply
{
            numStars = 5 // numero di stelle totali
            stepSize = 0.5f // possibilità di mettere anche i mezzi voti
            rating = task.vote?.toFloat() ?: 1f
            // ho usato il float altrimenti il numero di stelle era diverso da 5
(il valore di default è 1)
            layoutParams = FrameLayout.LayoutParams(
```

```
                FrameLayout.LayoutParams.WRAP_CONTENT,
                FrameLayout.LayoutParams.WRAP_CONTENT
            ).apply {
                gravity = Gravity.CENTER
            }
        }
        ratingContainer.addView(ratingBar)
        container.addView(ratingContainer)

        // Campo per il commento
        val commentInput = EditText(this).apply {
            hint = "Aggiungi un commento (opzionale)"
        }
        container.addView(commentInput)

        builder.setView(container)

        builder.setPositiveButton("Conferma") { _, _ ->
            val newVote = ratingBar.rating
            val commentText = commentInput.text.toString().trim()

            val taskRef = db.collection("tasks").document(task.id)

            taskRef.get().addOnSuccessListener { docSnapshot ->
                val userId = docSnapshot.getString("userId")
                if (userId == null) {
                    Toast.makeText(this, "Impossibile recuperare l'utente della
task", Toast.LENGTH_SHORT).show()
                    return@addOnSuccessListener
                }

                // Salva voto in Firestore
                taskRef.update("vote", newVote)
                    .addOnSuccessListener {
                        Toast.makeText(this, "Voto salvato",
Toast.LENGTH_SHORT).show()

                        // Se presente, invia un commento in chat all'utente
                        if (commentText.isNotEmpty()) {
                            val caregiverId =
FirebaseAuth.getInstance().currentUser?.uid ?: return@addOnSuccessListener

                            sendMessageToChat(
                                senderId = caregiverId,
                                recipientId = userId,
                                text = "Hai ricevuto un voto di $newVote su
\"${task.title}\":\n$commentText",
                                imageUrl = null
                            )
                        }

                        // Aggiorna premi se voto = 5
                        if (newVote == 5f) {
                            val prizeDocRef =
```

```
db.collection("prizes").document(userId)

                            prizeDocRef.get().addOnSuccessListener { document ->
                                if (document.exists()) {
                                    val currentCount = document.getLong("count")
?: 0
                                    prizeDocRef.update("count", currentCount + 1)
                                } else {
                                    prizeDocRef.set(mapOf("count" to 1))
                                }
                            }.addOnFailureListener { e ->
                                Toast.makeText(this, "Errore premi: ${e.message}",
Toast.LENGTH_SHORT).show()
                            }
                        }

                        loadTasksForUser(userId)
                    }
                    .addOnFailureListener { e ->
                        Toast.makeText(this, "Errore nel salvataggio del voto:
${e.message}", Toast.LENGTH_SHORT).show()
                    }
            }
        }

        builder.setNegativeButton("Annulla", null)
        builder.show()
    }

    private fun showRedeemDialog() {
        val builder = AlertDialog.Builder(this)
        builder.setTitle("Riscatta un premio per l'utente")

        val layout = layoutInflater.inflate(R.layout.dialog_redeem_prize, null)
        val prizeCountText = layout.findViewById<TextView>(R.id.prizeCountText)
        val redeemButton = layout.findViewById<Button>(R.id.confirmRedeemButton)

        val prizeDocRef = db.collection("prizes").document(userId)

        prizeDocRef.get().addOnSuccessListener { document ->
            var currentCount = document.getLong("count")?.toInt() ?: 0

            fun updateUI() {
                if (currentCount > 0) {
                    prizeCountText.text = "Premi disponibili: $currentCount"
                    redeemButton.isEnabled = true
                } else {
                    prizeCountText.text = "Nessun premio disponibile"
                    redeemButton.isEnabled = false
                }
            }

            updateUI()
```

```
            redeemButton.setOnClickListener {
                if (currentCount > 0) {
                    currentCount--
                    prizeDocRef.update("count", currentCount)
                        .addOnSuccessListener {
                            updateUI()
                            Toast.makeText(this, "Premio riscattato!",
Toast.LENGTH_SHORT).show()
                        }
                        .addOnFailureListener { e ->
                            Toast.makeText(this, "Errore: ${e.message}",
Toast.LENGTH_SHORT).show()
                        }
                }
            }

            builder.setView(layout)
            builder.setNegativeButton("Chiudi", null)
            builder.show()
        }.addOnFailureListener { e ->
            Toast.makeText(this, "Errore nel recupero premi: ${e.message}",
Toast.LENGTH_SHORT).show()
        }
    }
```

In queste funzioni è presente la logica principale di assegnazione e riscossione del premio. Infatti per votare una task mostro una finestra di dialogo con una rating bar per visualizzare la tipica schermata di valutazione con le stelle. Quando il caregiver valuta la task, viene memorizzato il voto, aggiornato il campo corrispondente della task associata all'utente con una chiamata a firestore e se è uguale a 5 si va a creare, se non esiste, un nuovo documento chiamato `prizes` associato all'utente che contiene un contatore che salva il numero di premi riscattabili dall'utente (il numero sarà visualizzabile dall'utente dalla schermata delle sue statistiche, mentre dal caregiver nella finestra di dialogo di riscossione). Il contatore viene aggiornato ogni qualvolta la condizione è rispettata. Durante la votazione si può aggiungere un commento e se è presente questo viene inviato come messaggio in chat.

Per riscattare un premio, i due dopo essersi accordati, possono rimuovere i punti guadagnati in app con la finestra di dialogo apposita presente all'interno della sezione di gestione delle task di un utente. Questa finestra fa una semplice query a firestore sul documento prizes dell'utente mostrando i dati e un pulsante per decrementare il contatore; se cliccato aggiorna il valore sul database.

10. Schermata statistiche

Per le statistiche ho implementato due versioni una per gli utenti (`UserStatisticsActivity`) e una per i caregiver (`CaregiverStatisticsActivity`), ognuna che mostra dei dati diversi. Per il caregiver le statistiche mostrano: il numero di utenti associati, il numero delle task totali create, le task da inziare, in corso e completate degli utenti e la percentuale degli utenti che completano le task in tempo. Per l'utente le statistiche mostrano: il numero di task totali, il numero di task completate, in corso e da iniziare, la percentuale delle task completate in tempo, il tempo medio di completamento, i premi correnti ottenuti e il numero di premi riscattabili. Sotto alle statistiche dell'utente è presente anche una piccola didascalia che spiega come si fa ad ottenere i premi.

Queste due classi sono molto semplici, la parte principale sono due funzioni di caricamento dei dati (`caricaStatisticheUtente` e `caricaStatisticheCaregiver`), una per classe. Essendo che le funzioni sono solo delle chiamate a firestore con delle elaborazioni sui dati non le riporto.

> Gli elementi cardine delle funzioni sono, una chiamata a firebase sulla collezione di utenti, con cui ottengo l'id e il nome; una sulla collezione di task in cui ottengo il numero totale, il numero di quelle completate, in corso e da inziare più, tramite elaborazioni sui campi, anche le percentuali e la durata media di completamento; infine una chiamata alla collezione delle associations nelle statistiche dei caregiver per ottenere il numero di utenti associati.

Obbiettivo Progetto IOS

L'obbiettivo che si vuole raggiungere con lo sviluppo di questa parte è l'apprendimento delle conoscenze base della programmazione swift, imparando quali sono i suoi elementi base, la struttura del linguaggio di programmazione e capire come comunicano gli elementi di UI con quelli della logica effettiva dell'applicazione.

Imprevisti

Durante l'anno accademico 2024/2025 il tutorial è stato aggiornato per garantire la compatibilità con i più recenti sistemi operativi. Tuttavia, i Mac a disposizione dell'università non supportano più gli aggiornamenti, rendendo di fatto impossibile seguire la guida ufficiale online. Dopo averne discusso con il docente, ho individuato un metodo alternativo che mi ha consentito di completare il progetto, integrare le modifiche richieste e redigere un file di supporto `FixTutorial`, così da permettere anche ad altri di seguirne i passaggi nonostante le limitazioni tecniche.

Modifiche principali effettuate

Le modifiche principali che ho effettuato per personalizzare l'applicazione sono state tradurre tutte le stringhe, etichette e titoli dell'applicazione in italiano, modificare gli "scrums" con colori, titoli e partecipanti scelti da me.

Ecco di seguito parti codici che mostrano alcune delle modifiche:

```
extension DailyScrum {
    static let sampleData: [DailyScrum] =
    [
        //sono stati modificati i titoli, gli attendees (pertecipanti) e i colori
        DailyScrum(title: "Tutorial IOS",
                   attendees: ["Emanuele", "Andrea", "Corinna", "Jonathan"],
                   lengthInMinutes: 10,
                   theme: .bubblegum),
        DailyScrum(title: "Progetto Android",
                   attendees: ["Katie", "Giovanni", "Euna", "Luis", "Darla"],
                   lengthInMinutes: 5,
                   theme: .sky),
        DailyScrum(title: "Stesura Relazione",
                   attendees: ["Chella", "Chris", "Mario", "Eden", "Karla",
"Lindsey", "Aga", "Chad", "Jenn", "Sarah"],
```

```
                    lengthInMinutes: 5,
                    theme: .poppy)
        ]
    }

    struct DetailView: View {
        @Binding var scrum: DailyScrum //mettere il binding

        @State private var editingScrum = DailyScrum.emptyScrum
        @State private var isPresentingEditView = false

        var body: some View {
                List {
                    //modificate le due etichette
                    Section(header: Text("Informazioni Riunione")) {
                        NavigationLink(destination: MeetingView(scrum: $scrum)) {
                            Label("Inizia la riunione", systemImage: "timer")
                                .font(.headline)
                                .foregroundColor(.accentColor)
                        }
                        HStack {
                            Label("Durata", systemImage: "clock")
                            Spacer()
                            Text("\(scrum.lengthInMinutes) minutes")
                        }
                        .accessibilityElement(children: .combine)
                        HStack {
                            Label("Tema", systemImage: "paintpalette")
                            Spacer()
                            Text(scrum.theme.name)
                                .padding(4)
                                .foregroundColor(scrum.theme.accentColor)
                                .background(scrum.theme.mainColor)
                                .cornerRadius(4)
                        }
                        .accessibilityElement(children: .combine)
                    }
                    Section(header: Text("Partecipanti")){
                        ForEach(scrum.attendees){ attendee in
                            Label(attendee.name, systemImage: "person")
                        }
                    }
                    Section(header: Text("Storico")){
                        if scrum.history.isEmpty {
                            Label("Ancora nessun meeting", systemImage:
    "calendar.badge.exclamationmark")
                        }
                        ForEach(scrum.history) { history in
                                HStack {
                                Image(systemName: "Calendario")
                                Text(history.date, style: .date)
                            }
                        }
                    }
```

```
            }.navigationTitle(scrum.title)
            .toolbar {
                        Button("Modifica") {
                            isPresentingEditView = true
                            editingScrum = scrum
                        }
                    }
                    .sheet(isPresented: $isPresentingEditView) {
                        NavigationStack {
                            DetailEditView(scrum: $editingScrum, saveEdits:
{dailyscrum in scrum = editingScrum}) //aggiunto anche il saveEdits
                                    .navigationTitle(scrum.title)
                                    .toolbar {
                                        ToolbarItem(placement: .cancellationAction) {
                                            Button("Cancella") {
                                                isPresentingEditView = false
                                            }
                                        }
                                        ToolbarItem(placement: .confirmationAction) {
                                            Button("Fatto") {
                                                isPresentingEditView = false
                                                scrum = editingScrum
                                            }
                                        }
                                    }
                        }
                    }
                }
            }

struct DetailView_Previews: PreviewProvider {
    static var previews: some View {
        NavigationStack{
            DetailView(scrum: .constant(DailyScrum.sampleData[0])) // modificare
anche questo rispetto al tutorial
        }

    }
}
```

> Alcuni commenti presenti nel codice riportato sono stati usati per la creazione del documento per superare i problemi del tutorial online

Elementi principali usati

Tra gli elementi di swift usati, quelli più importanti secondo me sono stati: `HStack`, `VStack`, in minor parte anche `ZStack`, che permettono di orientare gli elementi nello schermo in orizzontale, verticale e cambiare la profondità degli elementi (spostarli avanti o dietro ad altri); le `list` con le `section` che permette di avere degli elementi visualizzati come liste e suddivisi in sezioni con un titolo ed un etichetta; infine i `navigationStack` e i `navigationLink` che permettono di creare il contenitore in cui si può passare ad un'altra view e il collegamento alla stessa.