



# LiveData and UI - Parte II

---

## Actualizando datos en la UI

---

### Competencias

- Observar LiveData en el proyecto.
- Construir una segunda actividad y ViewModel
- Actualizar la UI del proyecto con LiveData

### Introducción

Previamente, logramos dejar nuestra aplicación lista para mostrar datos que provengan directamente desde la BBDD a la UI a través de LiveData. Estos datos que van directo al adapter, no pueden ser modificados, ya que la clase LiveData es una instancia abstracta que no tiene métodos para ser modificable.

Afortunadamente, se creó una clase llamada MutableLiveData, es a través de esta que podemos modificar los valores provenientes de estos objetos gracias a métodos públicos que nos facilitaran las tareas.

La idea es que nuestra aplicación contenga una nueva actividad para ingresar Ítems, a medida que se llenan los datos, podremos saber el cálculo de la cantidad de dinero que saldrá nuestros consumo.

A continuación terminaremos la aplicación que te puede servir de base para otros proyectos.

### 1. Añadir una nueva actividad al Proyecto

Primero vamos a crear una nueva actividad, para eso a través de android studio añade una segunda actividad al proyecto. Utiliza el template de “Empty Activity”.

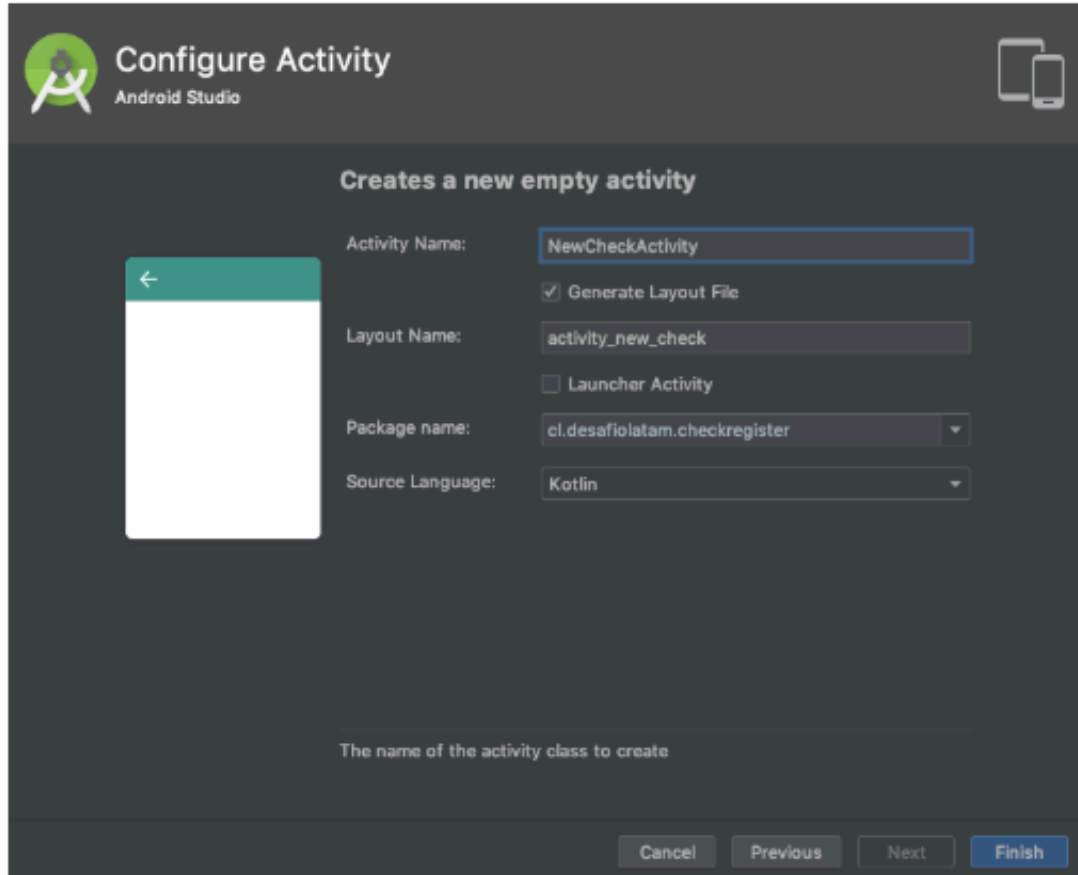


Imagen 1.

Revisa que la actividad nueva se haya registrado correctamente en el archivo de Manifest.xml y que su parent activity sea MainActivity.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="cl.desafiolatam.checkregister">

    <application
        android:name=".app.CheckApplication"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportRtl="true"
        android:theme="@style/AppTheme">
```

```

<activity
    android:name=".ui.MainActivity"
    android:label="@string/app_name"
    android:theme="@style/AppTheme.NoActionBar">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity android:name=".ui.NewCheckActivity"
    android:parentActivityName=".ui.MainActivity">
</activity>
</application>
</manifest>

```

## 2. Modificar el Layout que viene por defecto.

Ahora vamos a crear una interfaz gráfica para que los usuarios de la aplicación puedan ingresar el consumo. Modifica el archivo activity\_new\_check.xml.

```

<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data>
        <variable
            name="viewModel"
            type="cl.desafiolatam.checkregister.viewModel.
                CheckCreateUpdateViewModel" />
    </data>
    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".ui.NewCheckActivity">

        <TextView
            android:id="@+id/nameLabel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="16dp"
            android:text="Nombre de Item"
            android:textSize="18sp"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>

```

```

<EditText
    android:id="@+id/nameEditText"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:hint="Ingresa el nombre del Item"
    android:inputType="text"
    android:text="@={viewModel.name}"
    app:layout_constraintBaseline_toBaselineOf="@+id/nameLabel"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/nameLabel" />

<TextView
    android:id="@+id/unitPriceLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:text="Precio unidad"
    android:textSize="18sp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/nameLabel" />

<EditText
    android:id="@+id/unitPriceEditText"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:inputType="number"
    android:text="@={viewModel.singlePrice}"
    app:layout_constraintBaseline_toBaselineOf="@+id/unitPriceLabel"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/unitPriceLabel" />

<TextView
    android:id="@+id/quantityLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:text="Cantidad"
    android:textSize="18sp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/unitPriceLabel" />

<NumberPicker
    android:id="@+id/quantitySpinner"

```

```

        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        app:layout_constraintBaseline_toBaselineOf="@+id/quantityLabel"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/quantityLabel"/>

<TextView
    android:id="@+id/hitPointsLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:text="Total Actual"
    android:textSize="28sp"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/quantitySpinner"
    tools:text="Total actual" />

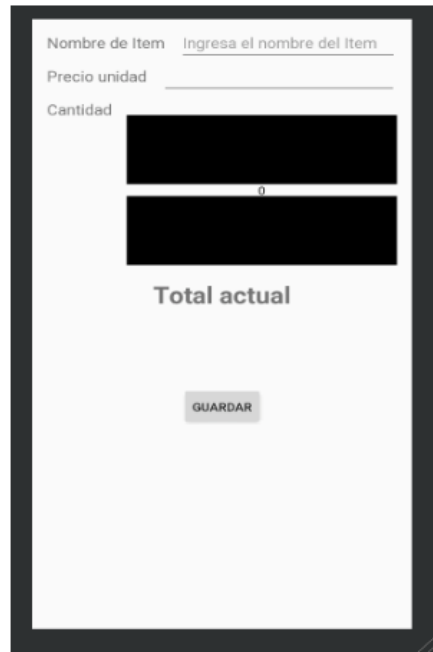
<TextView
    android:id="@+id/total_item"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@={` + viewModel.total}"
    android:textSize="56sp"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/hitPointsLabel" />

<Button
    android:id="@+id/saveButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:onClick="@{() -> viewModel.saveCheck()}"
    android:text="Guardar"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/total_item" />
</androidx.constraintlayout.widget.ConstraintLayout>
</layout>

```

Estamos utilizando un “layout” porque usaremos DataBinding para comunicar directamente nuestra vista con nuestro ViewModel.

El preview de este archivo debería verse así.



The image shows a mobile application preview with a light gray background and a dark gray border. It contains the following elements:

- Nombre de Item:** A text label followed by a text input field containing the placeholder text "Ingresa el nombre del Item".
- Precio unidad:** A text label followed by a text input field.
- Cantidad:** A text label followed by a spinner control. The spinner has a black bar over it, and the number "0" is visible below the bar.
- Total actual:** A text label.
- GUARDAR:** A button with the text "GUARDAR" in all caps.

Imagen 2. Preview de la aplicación.

La idea es que el usuario pueda ingresar el nombre de un Ítem, un valor inicial y luego seleccionar a través de un Spinner la cantidad de su consumo.

### 3. Agregar un Nuevo ViewModel

En el capítulo anterior utilizamos un ViewModel para mostrar los datos provenientes de la BBDD en nuestra vista, también a través de él podemos insertar y eliminar los elementos. Como Buena práctica, Para que nuestro proyecto sea más modular, escalable, vamos a generar un nuevo ViewModel para manejar los datos de la vista de la nueva actividad y añadir un nuevo ítem con sus respectivos atributos.

```
class CheckCreateUpdateViewModel(
    private val generator: CheckGeneratorTotal = CheckGeneratorTotal(),
    private val checkRepository: CheckRepository = CheckRepository()
) : ViewModel() {

    private val checkItemLiveData = MutableLiveData<Check>()
    private val saveLiveData = MutableLiveData<Boolean>()
    fun getCheckLiveData(): LiveData<Check> = checkItemLiveData
    fun getSaveLiveData(): LiveData<Boolean> = saveLiveData

    var name = ObservableField<String>("")
    var singlePrice = ObservableField<String>()
    var total = 0
    var quantity = 0
    lateinit var checkItem: Check

    fun updateCheckItem() {
        //      val attributes = CreatureAttributes(intelligence, strength,
        endurance)
        checkItem = generator.generateCheckTotal(
            name.get() ?: "",
            singlePrice.get() ?: "",
            quantity
        )
        checkItemLiveData.postValue(checkItem)
    }

    fun quantitySelected(quantitySelected: Int) {
        quantity = quantitySelected
        updateCheckItem()
    }

    fun saveCheck() {
        checkRepository.insertCheckItem(checkItem)
        saveLiveData.postValue(true)
    }
}
```

Para que esta vez podamos modificar los datos provenientes de LiveData, vamos a utilizar las clases MutableLiveData, las cuales nos permitirán a través de sus métodos públicos, poder añadir datos a los objetos LiveData. Observa también, que hemos creado métodos para hacer Update a los cálculos de los items(updateCheckItem) y también para guardar (saveCheck). Recuerda que las vistas están unidas por DataBinding.

Solo nos queda Unir este ViewModel a la vista correspondiente.

#### 4. Observando unidireccionalmente con LiveData en la actividad NewCheckActivity.

```
class NewCheckActivity : AppCompatActivity() {
    private lateinit var viewModel: CheckCreateUpdateViewModel
    lateinit var binding: ActivityNewCheckBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = DataBindingUtil.setContentView(this,
            R.layout.activity_new_check
        )
        viewModel =
            ViewModelProviders.of(this).get(CheckCreateUpdateViewModel::class.java)
        binding.viewModel = viewModel
        observableIntVariable()
        configureLiveDataObservers()
    }

    private fun observableIntVariable() {
        quantitySpinner.maxValue = 50
        quantitySpinner.minValue = 0
        quantitySpinner.wrapSelectorWheel = true
        quantitySpinner.setOnValueChangedListener { numberPicker, i, i2 ->
            viewModel.quantitySelected(numberPicker.value)
        }
    }

    private fun configureLiveDataObservers() {
        viewModel.getCheckLiveData().observe(this, Observer { checkItem ->
            checkItem?.let {
                total_item.text = it.totalItem
                nameEditText.setText(checkItem.name)
                unitPriceEditText.setText(checkItem.singlePrice)
            }
        })

        viewModel.getSaveLiveData().observe(this, Observer { saved ->
            saved?.let {

```



```

        if (it) {
            Toast.makeText(this, "Guardado Exitosamente",
                Toast.LENGTH_SHORT).show()
            finish()
        } else {
            Toast.makeText(this, "Error al guardar",
                Toast.LENGTH_SHORT).show()
        }
    }
}
}
}

```

El método observableIntVariable controla el numberPicker y cada vez que se cambia su valor realiza una llamada al método del viewModel que pasa el valor para realizar el cálculo.

Si ejecutamos nuestra aplicación podremos añadir Items, calcular su precio en tiempo real, guardar estos datos en la persistencia del teléfono, y todos lo estamos realizando con un patrón de diseño robusto y además potenciado por los patrones de arquitectura.

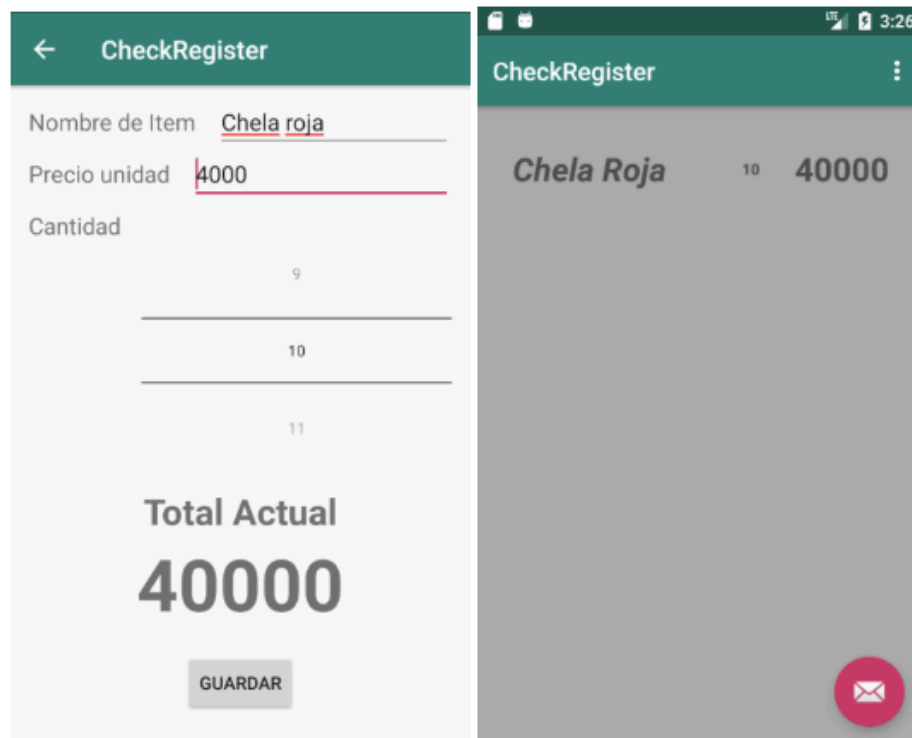


Imagen 3. Agregando items.

Con esto concluimos la unidad de LiveData y UI. Para seguir profundizando estos temas, te recomendamos ver los ejemplos CodeLabs de google para profundizar tus conocimientos.