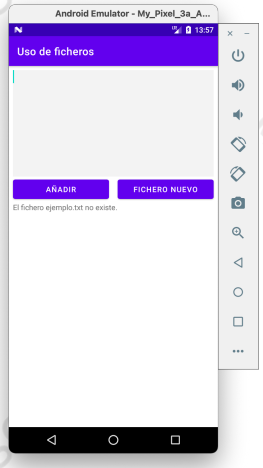
Anexo I. Manejo de ficheros



El uso de ficheros de texto plano (TXT, CSV, XML, etc) puede resultar obsoleto o anticuado, pero en más de una ocasión puede resultar una solución mediante una implementación fácil, no es difícil de tratar y puede resolver un problema que mediante el uso de otra tecnología podría resultar más compleja y costosa de mantener.

El siguiente proyecto de ejemplo muestra una aplicación que permite crear un fichero de texto si no existe, añadir información si ya existe y la lectura del mismo.

Para evitar solicitar permisos, los ficheros se crearán en la raíz de la propia aplicación, si se quisiese guardar en otro lugar, debería solicitarse el permiso WRITE_EXTERNAL_STORAGE en el *manifest* y gestionarlo desde la clase.

El primer paso será diseñar una *activity* que permita escribir en un *EditText* multilínea y muestre el contenido del fichero en un *TextView*.

A continuación, se muestra el código XML que forma la *activity* principal de la aplicación.

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

    <EditText
        android:id="@+id/etDatos"
        android:layout_width="0dp"
        android:layout_height="200dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:background="#F3F3F3"
        android:gravity="top"
        android:importantForAutofill="no"
        android:inputType="textMultiLine"
        android:scrollbars="vertical"
        app:layout_constraintBottom_toTopOf="@+id/btnSaveFileAppend"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:ignore="LabelFor"
        tools:text="Datos para guardar." />
```

```
    <TextView
        android:id="@+id/tvShowFile"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_marginStart="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginBottom="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/btnSaveFileAppend"
        tools:text="Aquí se muestra el contenido del fichero." />

    <Button
        android:id="@+id/btnSaveFileAppend"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginEnd="8dp"
        android:text="@string/btn_SaveFileAppend"
        app:layout_constraintEnd_toStartOf="@+id/btnSaveFileNew"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/etDatos" />

    <Button
        android:id="@+id/btnSaveFileNew"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginEnd="8dp"
        android:text="@string/btn_SaveFileNew"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/btnSaveFileAppend"
        app:layout_constraintTop_toBottomOf="@+id/etDatos" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

El siguiente paso será añadir el código necesario en la clase `MainActivity.kt`. En el método `onCreate()` se han añadido los métodos `setOnClickListener()` de los botones, un control de campo vacío y las llamadas a las funciones que realmente llevarán el peso de las acciones de lectura y escritura del fichero.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    // Comprueba si ya existe el fichero, si existe
    // se muestra su contenido al abrir la app.
    readFile()

```

```
    binding.btnSaveFileAppend.setOnClickListener {
        if (binding.etDatos.text.isNotEmpty()) {
            // Se separan las líneas del EditText para escribirlas
            // una a una, en realidad no sería necesario.
            val lineas: List<String> = binding.etDatos.text.split("\n")
            if (lineas.size > 0) {
                writeFile(lineas, false)
            }
            binding.etDatos.text.clear()
            readFile()
        } else {
            Toast.makeText(
                this,
                R.string.msg_warning1,
                Toast.LENGTH_SHORT
            ).show()
        }
    }

    binding.btnSaveFileNew.setOnClickListener {
        if (binding.etDatos.text.isNotEmpty()) {
            val lineas: List<String> = binding.etDatos.text.split("\n")
            if (lineas.size > 0) {
                writeFile(lineas, true)
            }
            binding.etDatos.text.clear()
            readFile()
        } else {
            Toast.makeText(
                this,
                R.string.msg_warning1,
                Toast.LENGTH_SHORT
            ).show()
        }
    }
}
```

Fíjate que se llama al método `readFile()` al inicio del método, éste se encargará de mostrar el contenido del fichero al inicio de la aplicación si existe, en caso contrario mostrará un mensaje indicado su ausencia.

El método `writeFile()`, que se muestra a continuación, es el encargado de la escritura de los datos, pasados mediante la variable `lineas`, el segundo argumento, se utilizará para indicar si se sobrescribe el fichero (*true*) o si debe añadirse la información (*false*), en ambos casos, el fichero se creará si no existe.

```
/**
 * Escribe el fichero de texto.
 * @tipo TRUE fichero nuevo, FALSE añade.
 */
```

```
fun writeFile(datos: List<String>, tipo: Boolean) {
    try {
        val salida: OutputStreamWriter
        if (tipo) {
            // Si el fichero no existe se crea,
            // si existe se sobrescribe.
            salida = OutputStreamWriter(
                openFileOutput(
                    getString(R.string.filename),
                    Activity.MODE_PRIVATE
                )
            )
        } else {
            // Si el fichero no existe se crea,
            // si existe se añade la información.
            salida = OutputStreamWriter(
                openFileOutput(
                    getString(R.string.filename),
                    Activity.MODE_APPEND
                )
            )
        }
        // Se escribe en el fichero línea a línea.
        for (d in datos) {
            salida.write(d + '\n')
        }
        // Se confirma la escritura.
        salida.flush()
        salida.close()

        Toast.makeText(
            this,
            getString(R.string.msg_correct),
            Toast.LENGTH_SHORT
        ).show()
    } catch (e: IOException) {
        Toast.makeText(this, e.message, Toast.LENGTH_LONG).show()
    }
}
```

La diferencia entre los modos de escritura se encuentra en el segundo parámetro del método `openFileOutput()`, en el que se indicará el modo de trabajo, este será un valor entero, y suele representarse con MODE_PRIVATE para sobrescribir, o MODE_APPEND para añadir.

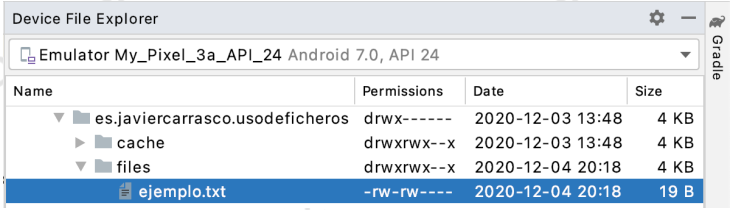
Tras la apertura del fichero, se escriben los datos utilizando el método `write()` sobre el objeto `OutputStreamWriter()` creado, se confirma la escritura con el método `flush()` y para finalizar se cierra el *stream* con el método `close()`.

Para terminar se muestra el método `readFile()`, que se encargará de comprobar la existencia del fichero y mostrar su contenido.

```
/**
 * Muestra el contenido del fichero de texto.
 */
fun readFile() {
    // Se comprueba si existe el fichero.
    if (fileList().contains(getString(R.string.filename))) {
        binding.tvShowFile.text = ""
        try {
            val entrada = InputStreamReader(
                openFileInput(getString(R.string.filename)))

            val br = BufferedReader(entrada)
            var linea = br.readLine()

            while (!linea.isNullOrEmpty()) {
                binding.tvShowFile.append(linea + "\n")
                linea = br.readLine()
            }

            br.close()
            entrada.close()
        } catch (e: IOException) {
            Toast.makeText(this, e.message, Toast.LENGTH_LONG).show()
        }
    } else {
        binding.tvShowFile.text = getString(
            R.string.msg_warning2,
            getString(R.string.filename))
    }
}
```

Resaltar la condición del `if`, que permite conocer si un fichero concreto está creado, el resto consiste en crear el `InputStreamReader` del fichero mediante `openFileInput` y leer, aquí sí, línea a línea el fichero. Fíjate que en ambas acciones, lectura y escritura, se utiliza `try-catch` para el control de errores utilizando `IOException`.

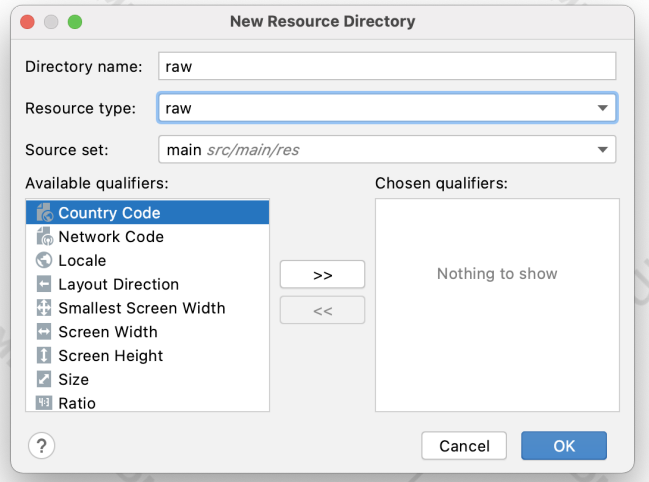
Para localizar el fichero creado en el dispositivo se deberá utilizar el *Device File Explorer*, lo puedes encontrar en el menú ***View > Tool Windows*** o en el panel vertical derecho. Destacar que esta opción únicamente funcionará con el emulador en marcha. Los ficheros creados se podrán encontrar en ***data > data > nombre aplicación > files***.



Cargar datos desde RAW

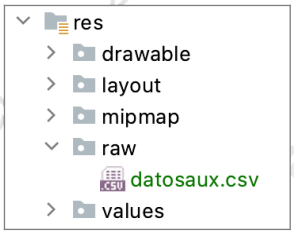
En ocasiones, puede resultar interesante tener una serie de datos preparados para realizar una carga inicial, esto puede hacerse desde un fichero ubicado en ***res > raw***. Este fichero acompañará a la aplicación en todo momento, y sólo se utilizará cuando sea necesario, por ejemplo, cuando la base de datos esté vacía y se quieran mostrar datos, o falle una descarga, etc.

El primer paso será crear el recurso ***raw*** en la sección de *resources* del proyecto.



Para acceder a este cuadro de dialogo puedes utilizar el menú contextual pulsando con el botón derecho sobre la carpeta *res* y seleccionar la opción ***New > Android Resource Directory***.

Una vez tengas el directorio ***raw*** creado, puedes añadir el fichero(s) que necesites para hacer uso de él. Desde ese momento, podrás acceder a dicho fichero como si se tratase de un recurso más, haciendo uso de `R.raw.nombrefichero` (el nombre del fichero debe estar completamente en minúsculas), por ejemplo, en el caso del fichero de muestra que se ve en la imagen, su recurso sería `R.raw.datosaux`, no se utiliza la extensión si la tuviese.



> ***IMPORTANTE**: Los ficheros ubicados en el recurso **raw** pueden leerse, pero no escribirse. Motivo por el que no es necesario el uso de permisos especiales.*

El código que se muestra a continuación es muy parecido al utilizado anteriormente para la lectura del archivo, pero varía en la forma de crear el *stream* para su lectura.

```
try {
    val entrada = InputStreamReader(
        resources.openRawResource(R.raw.datosaux)
    )

    val br = BufferedReader(entrada)
    var linea = br.readLine()

    while (!linea.isNullOrEmpty()) {
        Log.i("FILE", linea)
        linea = br.readLine()
    }

    br.close()
    entrada.close()
} catch (e: IOException) {
    Log.e("ERROR IO", e.message.toString())
}
```

Leer ficheros XML

Como ya sabrás, el formato XML (*eXtensible Markup Language*, lenguaje de marcas extensible) es un formato de texto delimitado por etiquetas, utilizado habitualmente para el intercambio de información entre sistemas.

Existen diferentes formas de analizar un fichero XML, desde la más básica haciéndolo de forma “manual” a herramientas capaces de generar estructuras de clases Java nativas. Pero aquí se centrará la atención en dos.

- **SAX** (*Simple API for XML*), que permite leer el fichero de forma secuencial (de principio a fin) sin conservar en memoria información sobre los datos que se habían leído anteriormente.
- **DOM** (*Document Object Model*), útil para muchos casos, pero con un alto consumo de memoria, ya que almacena todo el árbol del documento.

Para los ejemplos que se desarrollarán a continuación, se utilizará el siguiente fichero XML almacenado en la carpeta RAW del proyecto Android.

```
<?xml version="1.0" encoding="utf-8"?>
<pizzas>
  <pizza nombre="Barbacoa" precio="8">
    <ingrediente>Salsa Barbacoa</ingrediente>
    <ingrediente>Mozzarella</ingrediente>
    <ingrediente>Pollo</ingrediente>
    <ingrediente>Bacon</ingrediente>
    <ingrediente>Ternera</ingrediente>
  </pizza>
  <pizza nombre="Margarita" precio="6">
```

```
    <ingrediente>Tomate</ingrediente>
    <ingrediente>Jamón</ingrediente>
    <ingrediente>Queso</ingrediente>
  </pizza>
  <pizza nombre="Cuatro quesos" precio="10">
    <ingrediente>Tomate</ingrediente>
    <ingrediente>Queso Havarti</ingrediente>
    <ingrediente>Queso Azul</ingrediente>
    <ingrediente>Cabrales</ingrediente>
    <ingrediente>Mozzarella</ingrediente>
  </pizza>
  <pizza nombre="Cuatro estaciones" precio="9">
    <ingrediente>Tomate</ingrediente>
    <ingrediente>Jamón</ingrediente>
    <ingrediente>Champiñones</ingrediente>
    <ingrediente>Pepperoni</ingrediente>
    <ingrediente>Mozzarella</ingrediente>
  </pizza>
</pizzas>
```

También se creará la *data class* `Pizza` como modelo para que resulte más sencillo recuperar la información.

```
data class Pizza(val nom: String, val p: Int) {
    var nombre: String = nom
    var precio: Int = p
    var ingredientes: MutableList<String> = ArrayList()
}
```

Para ambos ejemplos, la clase `MainActivity` contará con el siguiente método para representar la lista de datos obtenida.

```
// Método encargado de mostrar en el TextView el contenido de la MutableList.
private fun escribirPizzas(pizzas: MutableList<Pizza>) {
    binding.textView.text = null

    pizzas.forEach {
        binding.textView.append("Pizza: ${it.nombre} | Precio: ${it.precio}€\n")

        binding.textView.append("Ingredientes:\n")
        it.ingredientes.forEach {
            binding.textView.append("- $it\n")
        }

        binding.textView.append("\n")
    }
}
```

SAX

SAX sólo permite lectura secuencial del fichero, de principio a fin, no se encarga de almacenar toda la estructura de datos en memoria. Por eso, es recomendable utilizarlo para análisis sencillos, en los que se desee volcar un fichero XML a otro tipo de fichero de datos o a una estructura dinámica en memoria creada para tal caso.

Los pasos para aplicar SAX son lo siguientes:

- Se crea un objeto de tipo `SAXParserFactory`.
- A partir de él, se creará un `SAXParser`.
- Crear un manejador de eventos, un `DefaultHandler` y sobrecargar los métodos `startElement()` (apertura de etiqueta), `characters()` (recoge el texto contenido entre la apertura y el cierre de una etiqueta) y `endElement()` (cierre de etiqueta).
- Llamar al método `parse` del *parser*, indicándole el fichero que debe analizar y el manejador de eventos que utilizará.

```
// Devuelve en una MutableList el contenido del fichero XML.
private fun leerPizzas(): MutableList<Pizza> {
    val pizzas: MutableList<Pizza> = ArrayList()

    try {
        val saxParserFactory = SAXParserFactory.newInstance()
        val saxParser = saxParserFactory.newSAXParser()

        val manejadorEventos = object : DefaultHandler() {
            var etiquetaActual = ""
            var contenido = ""
            lateinit var pizza: Pizza

            // Método que se llama al encontrar inicio de etiqueta: '<'.
            override fun startElement(uri: String?, localName: String?,
                qName: String?, attributes: Attributes?
            ) {
                // Si el tag es "pizza", empieza una nueva y se guarda su nombre y precio.
                if (qName != null) {
                    etiquetaActual = qName

                    if (etiquetaActual == "pizza") {
                        Log.d("XML-Pizza: ", attributes!!.getValue("nombre"))
                        pizza = Pizza(
                            attributes.getValue("nombre"),
                            attributes.getValue("precio").toInt()
                        )
                        pizzas.add(pizza) // Se añade a la lista.
```

```
                    }
                }
            }

            // Obtiene el dato entre '>' y '<'.
            override fun characters(ch: CharArray?, start: Int, length: Int) {
                contenido = String(ch!!, start, length)
            }

            // Se llama al encontrar un fin de etiqueta: '>'.
            override fun endElement(uri: String?, localName: String?, qName: String?) {
                if (etiquetaActual != "") {
                    Log.d("XML-Ingrediente", contenido)
                    pizza.ingredientes.add(contenido)
                    etiquetaActual = ""
                }
            }
        }

        // Cuerpo de la función: trata de analizar el fichero deseado
        // Llamará a startElement(), endElement() y character()
        saxParser.parse(resources.openRawResource(R.raw.pizzas), manejadorEventos);
    } catch (e: ParserConfigurationException) {
        e.printStackTrace();
    } catch (e: IOException) {
        e.printStackTrace();
    } catch (e: SAXException) {
        e.printStackTrace();
    }
    return pizzas
}
```

DOM

Si se prefiere memorizar todo el árbol del documento, de modo que se pueda acceder a cualquier dato en cualquier momento, la alternativa es emplear el DOM. En este caso, los pasos serán:

- Crear un objeto de tipo `DocumentBuilderFactory`.
- A partir de él, crear un `DocumentBuilder`.
- Crear también un `Document` y, con *parse*, indicarle el nombre del fichero que debe analizar.
- Obtener la lista de nodos con `getElementsByTagName()`, que se podrá recorrer con *for*.
- Si ese nodo tiene atributos, se podrán obtener con `attributes.getNamedItem(nombre)`.
- Sus elementos estarán accesibles con `nodeName`.

```kotlin
// Devuelve en una MutableList el contenido del fichero XML.
private fun leerPizzas(): MutableList<Pizza> {
    val pizzas: MutableList<Pizza> = ArrayList()

    try {
        val factory: DocumentBuilderFactory = DocumentBuilderFactory.newInstance()
        val builder: DocumentBuilder = factory.newDocumentBuilder()
        val doc: Document = builder.parse(resources.openRawResource(R.raw.pizzas))
        val raiz: Element = doc.getDocumentElement()
        val items: NodeList = raiz.getElementsByTagName("pizza")

        for (i in 0..items.length - 1) { // Recorre todos los elementos principales.
            val nodoPizza: Node = items.item(i)
            // Se crea la pizza utilizando las propiedades de tag "pizza".
            val pizza = Pizza(
                nodoPizza.attributes.getNamedItem("nombre").nodeValue,
                nodoPizza.attributes.getNamedItem("precio").nodeValue.toInt()
            )

            for (j in 0..nodoPizza.childNodes.length - 1) { // Recorre los hijos.
                val nodoActual = nodoPizza.childNodes.item(j)

                // Comprueba si es un elemento de tipo Node.
                if (nodoActual.nodeType == Node.ELEMENT_NODE) {
                    if (nodoActual.nodeName.equals("ingrediente")) {
                        Log.d("XML-ingrediente", nodoActual.childNodes.item(0).nodeValue)
                        pizza.ingredientes.add(nodoActual.childNodes.item(0).nodeValue)
                    }
                }
            }

            pizzas.add(pizza)
        }
    } catch (e: ParserConfigurationException) {
        e.printStackTrace()
    } catch (e: IOException) {
        e.printStackTrace()
    }
    return pizzas
}
```