8 de septiembre

1. SISTEMAS DE COMUNICACIÓN

La comunicación es transmisión de señales mediante un código común al emisor y al receptor. Tenemos una jerarquía entre cliente y servidor. Es siempre el cliente el que empieza la comunicación.
Tipos de **conexiones**:

- Persistente: el canal queda abierto, cliente y servidor intercambian mensajes con independencia (una vez abierta la conexión se mantiene abierta y ambos intercambian, aunque haya sido el cliente el que la inicia). Por ejemplo, HTTP/2.
- Puntual: una vez abierto el canal se intercambian dos mensajes, la petición original y la respuesta, tras esto el canal se cierra. Por ejemplo, HTTP.

Ejemplo práctico Forvo: Al abrir forvo.com, el navegador (cliente) hace una petición inicial para el HTML. Este archivo contiene referencias a otros recursos (CSS, JavaScript, imágenes). El navegador debe entonces hacer peticiones puntuales adicionales para cada uno de esos recursos. Los navegadores suelen limitar el número de conexiones simultáneas a un mismo dominio (ej: 2-6) para evitar saturar el servidor.
Vamos a trabajar con conexiones TCP/UDP sobre IP
Como cliente, debemos conocer la dirección IP del servidor, el puerto al que conectarnos y el tipo de conexión (TCP/UDP)
Disponemos de 65536 puertos. Los inferiores a 1024 requieren de permisos de administrador (root). Los servicios estandarizados tienen un puerto de referencia (https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml)

- HTTP: 80
- HTTPS: 443
- FTP: 20-21
- DNS: 53
- SSH: 22
- MySQL: 3306
- SMTP: 25
- PostgreSQL: 5432
- LDAP: 389

El protocolo dictamina el lenguaje y los tipos de mensajes que se intercambiarán. Existen multitud de protocolos: HTTP, HTTP/2, FTP, DNS, SSH, etc. El lenguaje está asociado al protocolo. Hoy en día los más utilizados son:

- Plano (Text): TXT, CSV, TSV.
- Binario: Formatos compactos y eficientes para máquinas (ej: un archivo .png, .zip).
- XML (eXtensible Markup Language): Lenguaje de marcado extensible. Verboso pero muy estructurado.
- JSON (JavaScript Object Notation): Ligero, fácil de leer y escribir para humanos y de parsear/generar para máquinas. Es el estándar de facto en la web moderna.

REST (REpresentational State Transfer) es un estilo arquitectónico (no un protocolo estricto) para diseñar sistemas de comunicación cliente-servidor. Se basa en peticiones HTTP con una semántica clara.
Una petición REST se define generalmente por estos parámetros:

- URL (Endpoint): La dirección del recurso. Ej: /users/antonio
- Cabeceras (Headers): Metadatos de la petición (ej: tipo de contenido, autenticación).
    - Método HTTP (Verbo): La acción a realizar sobre el recurso.
        - GET: Obtener/leer un recurso.
        - POST: Crear un nuevo recurso o enviar datos para procesar.
        - PUT: Actualizar o reemplazar por completo un recurso existente.
        - DELETE: Eliminar un recurso.
        - PATCH: Actualizar parcialmente un recurso.
    - Tipo de Contenido (Content-Type): Define el formato de los datos enviados (ej: application/json).
- Cuerpo (Body): Los datos propiamente dichos (típicamente en JSON). Usado en POST, PUT, PATCH

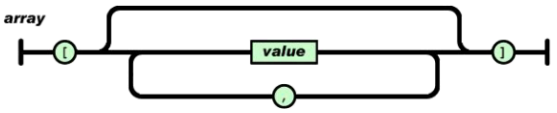
En general, hablaremos de una conexión a una dirección (URL) usando un verbo (tipo).
Listado de usuarios: GET /users
Crear usuario: PUT /users
De aquí obtenemos un usuario llamado antonio. Podemos interactuar con el sistema para realizar ciertas acciones:

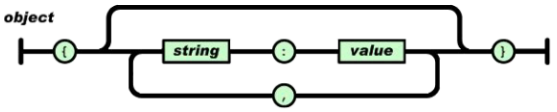
- Obtener información: GET /users/antonio
- Modificar sus datos: POST /users/antonio
- Borrar: DELETE /users/antonio

Usan la misma URL (/users/antonio) pero el tipo nos diferencia la acción (de aquí que nos refiramos a él como “verbo”). POST llevaría contenido, mientras que GET y DELETE no.

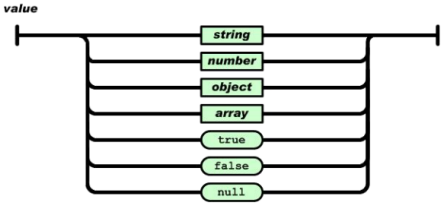
JSON (JavaScript Object Notation) es el lenguaje que vamos a usar para mandar información estructurada. Unión de dos especificaciones: JavaScript Programming Language + Standard ECMA-262 (3rd Edition). Es fácil de leer y escribir para humanos y para máquinas.
2 elementos estructurales (array y objeto), 5 tipos de valores nativos (string, numero, true, false y null ; o otro arrays u objeto anidado).

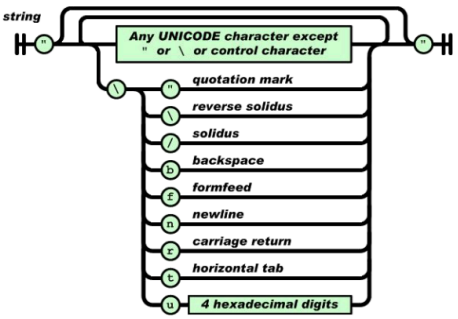
Ejemplo:
PUT /users
Fran, Naranjo
Francisco José, Naranjo
Esto se complica cuando queremos meter más información con más complejidad. Esto lo resuelve JSON.

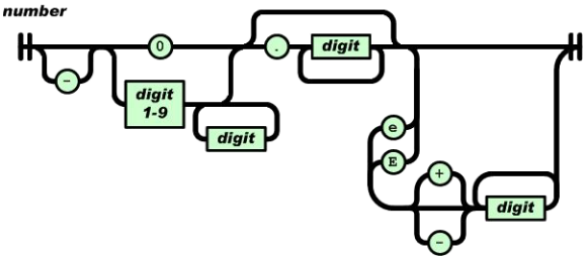




Un objeto es lo mismo que un array pero tienen elementos clave-valor.







Ejemplo:
{“key”: null} la clave existe pero no tiene contenido. Si intento acceder me dirá que el nulo.
{} la clave no existe. Si intentamos acceder nos dirá que no existe.

15 de septiembre

2. GIT

Sistema de control de versiones

- Herramienta para gestionar cambios en un proyecto a lo largo del tiempo.
- Registra todas las modificaciones: creación, actualización o borrado.
- Esencial para equipos de software (trabajo colaborativo) y para uso personal (historial y backups)

Ventajas:

- Escalabilidad: Permite trabajar en ramas (branches) y fusionarlas (merging).
- Historial completo: Trazabilidad de todos los cambios. Los datos no se borran, solo se registran cambios.
- Trabajo distribuido: Cada desarrollador tiene una copia local del repositorio

Repositorios

- Centralizado: Un único repositorio central. Todos envían cambios allí.
- Distribuido: Cada usuario tiene un repositorio local. Se sincroniza con un repositorio remoto

Conceptos clave

- *Commit*:

Cambio diferencial respecto al commit anterior. (se guarda en referencia a la ultima posicion y al borrar se “guarda” lo que se borra. No se guarda todo todo)

Los commits son siempre locales, pero se pueden enviar a otros repositorios.
Identificado con un hash único de 40 bytes.
Permite volver a cualquier estado anterior del proyecto.

- *Index (Staging):*

Zona intermedia donde se preparan los cambios antes de hacer commit.
No modifica archivos, solo registra la intención de cambio.

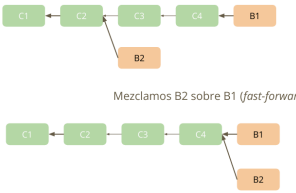
- *Branch (Rama):*

Es un puntero a un commit.
Permite trabajar en paralelo sin afectar la rama principal (main). Ejemplo: main y feature.

- *Tag:*

Es un nombre que le ponemos a un commit. Marca un commit específico (ej: versión 1.0). No permite nuevos commits (solo referencia).

- *Merge:*
  - Fusiona dos ramas. Genera un nuevo commit.
  - Fast-forward: Solo mueve el puntero de la rama (no crea commit extra).



Imagine there is a branch

`Master : A---> B --->`

There are two commits on it, A and B There is another feature branch, from B, it has the following commit History

`Feature : C--->D---->E`

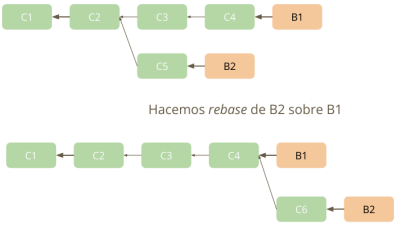
So the present it looks like this

```
`A--->B`
     `C--->D--->E `
```

So when you want to merge it, you can do it by fast forward merge by moving the head from B to E and incorporating the feature branch in the main branch and it looks like this

`A--->B--->C--->D--->E` with the header at E.

- *Rebase:*

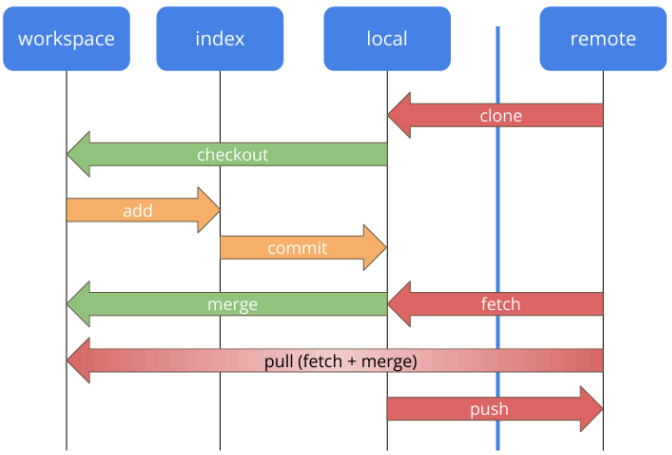


Reubica una rama sobre otra. Reescribe el historial (peligroso si se usa mal). Permite alterar, eliminar o unir commits (squash - útil para cuando ponemos información sensible donde no se debe).

- *Conflicto*: Fallo al fusionar ramas. Se debe resolver manualmente eligiendo la versión correcta.

Seguimiento de ramas remotas

Una rama local puede configurarse para "seguir" (track) una rama remota específica. Esto crea una relación automática entre ambas. La rama local "sabe" a qué rama remota debe enviar cambios (push) y de cuál recibirlos (pull/fetch). Las ramas de seguimiento remoto se nombran generalmente como remoto/rama (ej: origin/main).



COMANDOS	INFO
Configuración	
git config --global user.name “tu nombre”	Global: --global (afecta a todos los repositorios). Local: sin flag (solo para el repositorio actual).
git config --global user.email “tu@email.com”	
git config --global color.ui auto	

`cat <<EOF > .gitignore`
`/logs/*`
`!logs/.gitkeep`
`/tmp`
`*.swp`
`EOF`
	To ignore files, create a .gitignore file in your repository with a line for each pattern. File ignoring will work for the current and sub directories where .gitignore file is placed. In this example, all files are ignored in the logs directory (excluding the .gitkeep file), whole tmp directory and all files *.swp.

Proyecto	
git init [project name]	Create a new local repository in the current directory. If [project name] is provided, Git will create a new directory named [project name] and will initialize a repository inside it.
git clone	Downloads a project with the entire history from the remote repository.
Comunes	
git status	Displays the status of your working directory. Options include new, staged, and modified files. It will retrieve branch name, current commit identifier, and changes pending commit.
git add [file]	Add a file to the staging area. Use. in place of the full file path to add all changed files from the current directory down into the directory tree.
git diff [file]	Show changes between working directory and staging area
git diff --staged [file]	Shows any changes between the staging area and the repository
git checkout -- [file]	Discard changes in working directory. This operation is unrecoverable.
git reset [<path>...]	Revert some paths in the index (or the whole index) to their state in HEAD.
git commit -m “mensaje”	Create a new commit from changes added to the staging area. The commit must have a message!
git rm [file]	Remove file from working directory and staging area.
Guardar	
git stash	Put current changes in your working directory into stash for later use.
git stash pop	Apply stored stash content into working directory, and clear stash
git stash drop	Delete a specific stash from all your previous stashes
Branching model	
git branch [-a]	List all local branches in repository. With -a: show all branches (with remote).
git branch [branch_name]	Create new branch, referencing the current HEAD.
git rebase [branch_name]	Apply commits of the current working branch and apply them to the HEAD of [branch] to make the history of your branch more linear
git checkout [-b] [branch_name]	Switch working directory to the specified branch. With -b: Git will create the specified branch if it does not exist. Si hay cosas que no estén en el stage se desharán.
git merge [branch_name]	Join specified [branch_name] branch into your current branch (the one you are on currently).

git branch -d [branch_ name]	Remove selected branch, if it is already merged into any other. -D instead of -d forces deletion.
git show	Mezcla de log y diff
Historial	
git log [-n count]	List commit history of current branch. -n count limits list to last n commits
git log --oneline --graph --decorate	An overview with reference labels and history graph. One commit per line
git log ref .	List commits that are present on the current branch and not merged into ref. A ref can be a branch name or a tag name.
git log .ref	List commit that are present on ref and not merged into current branch.
git reflog	List operations (e.g. checkouts or commits) made on local repository.
git blame [file]	Muestra quién modificó cada línea.
Tagging commits	
git tag	List all tags
git tag [name] [commit sha]	Create a tag reference named name for current commit. Add commit sha to tag a specific commit instead of current one.
git tag -a [name] [commit sha]	Create a tag object named name for current commit.
git tag -d [name]	Remove a tag from local repository.
Deshacer cambios	
git reset [--hard] [target reference]	Switches the current branch to the target reference, leaving a difference as an uncommitted change. When --hard is used, all changes are discarded. It's easy to lose uncommitted changes with --hard.
git revert [commit sha]	Create a new commit, reverting changes from the specified commit. It generates an inversion of changes.
Sincronizar repositorios	
git fetch [remote]	Fetch changes from the remote, but not update tracking branches. Para actualizar es necesario hacer un merge.
git fetch --prune [remote]	Delete remote Refs that were removed from the remote repository.
git pull [remote]	Fetch changes from the remote and merge current branch with its upstream.
git push [--tags] [remote]	Push local changes to the remote. Use --tags to push tags. Mezcla los cambios locales en remoto y sólo es posible si el merge en remoto es de tipo fast-forward. Fallará en otro caso.
git push -u [remote] [branch]	Push local branch to remote repository. Set its copy as an upstream.
General para merge conflicts	

22 de septiembre

2. Android

FALTA ANDROID 2.1

Activity

Una Activity representa una pantalla en una aplicación Android. Cada aplicación está compuesta por una sucesión de actividades independientes. Cada actividad tiene su propio ciclo de vida.

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Inicialización básica
    }
    @Override
    protected void onStart() {
        super.onStart();
        // La actividad está a punto de hacerse visible
    }
    @Override
    protected void onResume() {
        super.onResume();
        // La actividad se ha vuelto visible (está "activa")
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Otra actividad está tomando el foco
    }
    @Override
    protected void onStop() {
        super.onStop();
        // La actividad ya no es visible
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // La actividad está a punto de ser destruida
    }
}
```

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme">

        <!-- Activity principal (LAUNCHER) -->
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
```

```
            </intent-filter>
        </activity>

        <!-- Otra actividad -->
        <activity
            android:name=".SecondActivity"
            android:label="@string/title_second_activity" />
    </application>
</manifest>
```

Layout

Ordenación y representación de elementos en un activity. Formato XML <-> GUI. Se colocan en res > layou

Tipos principales:

LinearLayout

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Nombre:" />

    <EditText
        android:id="@+id/name_input"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <Button
        android:id="@+id/save_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Guardar" />
</LinearLayout>
```

RelativeLayout

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Botón 1"
        android:layout_alignParentTop="true"
        android:layout_alignParentStart="true" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Botón 2"
        android:layout_below="@id/button1"
        android:layout_toEndOf="@id/button1" />
</RelativeLayout>
```

Widgets

- TextView

El TextView es solo para mostrar información. El usuario NO puede modificar su contenido directamente. Si necesitas el texto, ya lo conoces porque tú lo estableciste, por ello no tiene sentido hacer un getText().

```
<TextView
    android:id="@+id/title_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Título"
    android:textSize="24sp" />
```

- EditText

EditText es un campo en el que el usuario va a poder escribir. En función de la versión se podrían ver representados de diferente forma.

```
<EditText
    android:id="@+id/email_input"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Introduce tu email"
    android:inputType="textEmailAddress" />
```

hint vs text: hint es una ayuda, el valor del campo es vacío y con text el campo tiene algo como valor.
Hay que hacer .getText().toString().

- Button

```
<Button
    android:id="@+id/action_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hacer acción"
    android:onClick="onButtonClick" />
```

- RadioButton
- CheckButton
- Switch
- ToggleButton
- ImageView

```
<ImageView
    android:id="@+id/logo_image"
    android:layout_width="100dp"
    android:layout_height="100dp"
    android:src="@drawable/logo" />
```

- ProgressBar

Trabajar con widgets desde código

Acceder y manipular widgets

```
public class MainActivity extends AppCompatActivity {
    private EditText nameInput;
    private Button saveButton;
    private TextView resultText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Referenciar widgets por ID
        nameInput = findViewById(R.id.name_input);
```

```
        saveButton = findViewById(R.id.save_button);
        resultText = findViewById(R.id.result_text);

        // Configurar evento del botón
        saveButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                String name = nameInput.getText().toString();
                resultText.setText("Hola, " + name + "!"); //Asignar un valor
            }
        });
    }
}

// No tiene sentido hacer un getText de un TextView, pero sí de un EditText.
```

Recursos y traducciones

Cada uno de los strings que vayamos a utilizar hay que meterlos en strings, que son ficheros guardados en res - values. Hay que crear strings por cada string que tengamos que usar, nunca se pone texto a pelo.
strings.xml (español - valores por defecto)

```
<resources>
    <string name="app_name">Mi Aplicación</string>
    <string name="welcome_message">Bienvenido a la aplicación</string>
    <string name="save_button">Guardar</string>
    <string name="cancel_button">Cancelar</string> //identificador es cancel_button y valor es Cancelar y
luego para usarlo es @string/cancel_button
</resources>
```

Para traducciones: hay que añadir otro fichero de strings con las traducciones. Hay que tener carpetas con el identificador values-XX (es, en, …). Se hace click derecho, new values resource file. Files = strings. Directory = values. Locale. Any Region.

Eventos y Listeners

Implementación de listeners

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main); //Vamos a utilizar activity_main
        EditText firstName = findViewById(R.id.first_name);
        firstName.setText(“Fran”);
        Button button = findViewById(R.id.my_button);
        // Listener anónimo
        button.setOnClickListener(new View.OpenSettingsClickListener(this)); //Definimos la
intencionalidad y le damos un contexto = actividad
    }
}

class OpenSettingsClickListener implements View.OnClickListener {
    private final Context context;

    public OpenSettingsClickListener(Context context) {
        this.context = context;
    }

    @Override
    public void onClick(View v) {
        Intent intent = new Intent(this.context, SettingActivity.class);
        this.context.startActivity(intent);
    }
```

```
}
```

El contexto es parte de la aplicación donde está corriendo Android. Es el sitio virtual donde esta corriendo la app. La actividad es un contexto. Cuando se haga un new OpenSettingsClickListener alguien me tiene que dar una clase de contexto.

Como cambiar de actividad: lo hacemos con los intent
Intent intent = new Intent (this, NextActivity.class);
startActivity(intent)

Los activity tienen un método finish() que cierra la actividad. Esto tendría sentido por ejemplo cuando se abra settings entonces main desaparezca.

```
class OpenSettingsClickListener implements View.OnClickListener {
    private final Activity activity;

    public OpenSettingsClickListener(MainActivity activity) { //para asegurarnos que solo se cierra main y
no cualquiera
        this.activity = activity;
    }

    @Override
    public void onClick(View v) {
        Intent intent = new Intent(this.context, SettingActivity.class);
        this.activity.startActivity(intent);
        this.activity.finish();
    }
}
```

Hay que tener cuidado con pasarle valores al setOnclickListener, hay que darle el objeto. Porque puede que cuando se ejecute este vacío el valor.

29 de septiembre
GRADLE
¿Qué es?

- Herramienta de automatización de construcción de proyectos.
- Usa un DSL (Domain-Specific Language) basado en Groovy.
- Utiliza un DAG (Grafo Acíclico Dirigido) para determinar el orden de ejecución de tareas.

¿Para qué se usa en Android Studio?

- Compilar código.
- Enlazar librerías.
- Firmar aplicaciones.
- Ejecutar procesos automáticos.
- Gestionar dependencias (librerias por ejemplo)

En proyectos Android, se utiliza principalmente el archivo **`build.gradle`** para:

- Configurar dependencias (librerías externas).
- Definir versiones del SDK.
- Establecer configuraciones de compilación.

Después de modificar hay que darle a sincronizar.
SHAREDPREFERENCES
¿Qué es? Almacenamiento ligero en XML privado de la app, clave-valor, solo para datos simples. No se comparten con otras aplicaciones. Solo admite tipos de datos primitivos y cadenas: boolean, float, int, long, String. No sirve para pasar datos entre Activities, solo para guardar configuración o preferencias, ya que para pasar datos están los intent.

Recuperar datos:

```java
// Abrir SharedPreferences
SharedPreferences settings = getSharedPreferences("Config", 0);
// Obtener valores guardados
boolean alarmActive = settings.getBoolean("alarm_active", false);
float distance = settings.getFloat("distance", 0.0f);
int maxIssues = settings.getInt("max_issues", 10);
long timestamp = settings.getLong("last_open_timestamp", 0);
String username = settings.getString("username", "");
```

Para pedir tenemos que dar en primer lugar el nombre de la variable que queremos recuperar y luego el valor por defecto. Así, si la clave no existe, devuelve el valor por defecto indicado.

Almacenar datos:

```java
// Abrir SharedPreferences
SharedPreferences settings = getSharedPreferences("Config", 0);
// Crear editor para modificar los datos
SharedPreferences.Editor editor = settings.edit();
// Guardar valores
editor.putBoolean("alarm_active", alarmActive);
editor.putFloat("distance", distance);
editor.putInt("max_issues", maxIssues);
editor.putLong("last_open_timestamp", timestamp);
editor.putString("username", username);
// Confirmar cambios
editor.commit();  // o editor.apply();
```

Cuando hacemos put no se guarda hasta que hagamos commit o apply.
Diferencia entre commit() y apply():

- commit() → guarda los cambios de forma sincrónica (devuelve true/false).
- apply() → guarda los cambios de forma asíncrona (no bloquea el hilo principal).

6 de octubre

BASES DE DATOS

Son sistemas que permiten almacenar información estructurada de manera organizada. Se gestionan mediante DBMS (DataBase Management System): motores de bases de datos
Tipos:

- RDBMS (Relational-DBMS): Bases de datos relacionales
- NoSQL (Not Only SQL): Bases de datos no relacionales

Estructura RDBMS

Jerarquía estricta:

1. Bases de datos → Contenedor principal
2. Tablas → Estructuras para datos específicos
3. Propiedades (schema) → Definición de columnas y tipos

SGLite en Android

RDBMS integrado en Android. Soporte nativo desde API 1. Implementa estándar SQL-92
Documentación oficial: sqlite.org/lang.html
Consideraciones:

- CRUD (Create, Read, Update, Delete) debe programarse manualmente
- Es responsabilidad del desarrollador implementar todas las operaciones

ORM (Object-Relational Mapper)

Crea relaciones entre bases de datos y objetos Java. Básicamente se trata de abstraer la estructura de una

tabla en una clase Java. Permite un lenguaje más evidente (DAO - Data Access Object)
Flujo ORM completo:

```
// 1. Creas la entidad básica
// 2. GreenDAO genera automáticamente durante el build
// 3. Obtienes el DAO para ejecutar queries
UserDao userDao = daoSession.getUserDao();
// 4. Operaciones CRUD automáticas
userDao.insert(user);      // INSERT
userDao.update(user);      // UPDATE
userDao.delete(user);      // DELETE
userDao.load(id);          // SELECT por ID
userDao.loadAll();         // SELECT *
```

Ejemplo:

```
UserDao dao = manager.getUserDao();
User user = dao.find(2);                    // Buscar usuario con ID 2
String name = user.getName();               // Obtener nombre → "Pepe"
user.setName("José");                       // Modificar nombre
user.save();                                // Guardar cambios
```

Ventajas de ORM:

- Código más legible y mantenible
- Reducción de código repetitivo
- Mayor seguridad contra inyección SQL
- Abstracción del motor de base de datos

GreenDAO - ORM para Android

Características principales:

- Eficiente y rápida
- Simplifica tareas costosas
- Especialmente diseñada para Android

Configuración gradle - Para gradle 8 o superior:

```
greendao {
    schemaVersion 1  // Versión del esquema de base de datos
}
```

Cada vez que incrementas el schemaVersion, GreenDAO modifica la base de datos.
GreenDAO NO SABE hacer migraciones automáticas complejas. Por defecto: ¡BORRA TODOS LOS DATOS!
Por eso hay que hacerlo manualmente con DaoMaster.OpenHelper.

<u>Gestión de constructores y build</u>

Importante no borrar constructores manualmente. Dejar que GreenDAO genere los constructores automáticamente. No añadir constructores manualmente.
Para generar automáticamente:

1. Tú defines la entidad básica con anotaciones
2. GreenDAO genera durante el build:
    a. User.java (completo con constructores)
    b. UserDao.java (para ejecutar queries)
    c. DaoSession.java (gestión de sesiones)

<u>Gestión de cambios y clean project</u>

Siempre que cambies el proyecto hacer: Build > Clean Project; Build > Rebuild Project
Por ejemplo, si:

- Cambias el schemaVersion en build.gradle
- Modificas anotaciones (@Id, @NotNull, etc.)
- Añades/Eliminas propiedades en la entidad

- Cambias la estructura de la base de datos
- Actualizas la versión de GreenDAO
- Cualquier modificación en las entidades

Conexión y configuración de base de datos

- SQLite no soporta acceso concurrente
- Solo una conexión activa por vez
- Estrategias recomendadas:
    - Conectar a nivel de aplicación
    - Una instancia por Activity con control estricto

```
//ESTABLECER CONEXIÓN
// Crear helper para la base de datos
DaoMaster.DevOpenHelper helper = new DaoMaster.DevOpenHelper(this, "dbname");
// Obtener base de datos escribible
Database db = helper.getWritableDb();
// Crear sesión DAO
daoSession = new DaoMaster(db).newSession();
```

Modelado de entidades

Creación de modelos:greenrobot.org/greendao/documentation/modelling-entities/

Ejemplo de entidad user:

```
@Entity  // Indica que es una entidad de base de datos
public class User {
    @Id(autoincrement = true)  // Identificador único autoincremental
    private Long id;

    @NotNull  // Campo obligatorio, no puede ser nulo
    private String name;

    private int children;  // Campo opcional

    // GreenDAO generará automáticamente:
    // - Constructores necesarios
    // - Getters y Setters
    // - Métodos auxiliares
}
```

Anotaciones comunes:

- @Entity: Marca la clase como entidad de base de datos
- @Id: Define la clave primaria
- @NotNull: Campo obligatorio
- @Property: Personalizar nombre de columna
- @Unique: Garantizar valores únicos

Query builder - Consultas Type-Safe

El dao nos dejara ejecutar queries. Query builder es una clase que nos va a ayudar a escribir queries pero nos deja ser más genéricos a la hora de escribir.

Por ejemplo, si quiero recuperar todos los usuarios que tienen hijos y que me lo de en orden ascendente.

- Antes con SQL tradicional:

```
select * from user where childre>0 order by name asc;
```

- Con query builder:

```
Sobre el dao que quieres actuar le pides un query builder
```

```
List<User> users = userDao.queryBuilder()
    .where(UserDao.Properties.Children.gt(0))    // WHERE children > 0
    .orderAsc(UserDao.Properties.Name)           // ORDER BY name ASC
    .list();                                     // Ejecutar y obtener lista
```

Ventajas del query builder:

1. Type-safe

```
// Correcto - detecta errores en compilación
.where(UserDao.Properties.Children.gt(0))
// Error de compilación si la propiedad no existe
.where(UserDao.Properties.Edad.gt(0)) // "Edad" no existe - ERROR
```

2. Prevención de SQL injection

```
// Seguro - parámetros escapados automáticamente
.where(UserDao.Properties.Name.eq(userInput))
```

3. Código más legible y mantenible
4. Refactoring seguro - IDE detecta usos de propiedades

Ejemplos:

```
// Usuarios con exactamente 2 hijos
List<User> users = userDao.queryBuilder()
    .where(UserDao.Properties.Children.eq(2))
    .list();
// Usuarios que empiezan con "A"
List<User> users = userDao.queryBuilder()
    .where(UserDao.Properties.Name.like("A%"))
    .list();
// Primeros 10 usuarios ordenados por nombre
List<User> users = userDao.queryBuilder()
    .orderAsc(UserDao.Properties.Name)
    .limit(10)
    .list();
// Múltiples condiciones
List<User> users = userDao.queryBuilder()
    .where(UserDao.Properties.Children.gt(0),
           UserDao.Properties.Name.like("J%"))
    .orderAsc(UserDao.Properties.Name)
    .orderDesc(UserDao.Properties.Children)
    .list();
// Consultas con OR
List<User> users = userDao.queryBuilder()
    .whereOr(UserDao.Properties.Children.eq(0),
             UserDao.Properties.Name.eq("Admin"))
    .list();
```

Buenas practicas:

- Ponerle nombre a cada propiedad @Property(nameInDb = “id”)

13 de octubre

En Android, los listados son una forma muy común de mostrar conjuntos de datos, como una lista de contactos, mensajes o productos. Para crear un listado funcional y dinámico, se utilizan dos elementos fundamentales: el ListView y el Adapter.

ListView

El ListView es un elemento del layout que muestra un listado vertical de elementos (llamados “celdas” o “items”).
Cada celda del ListView puede tener su propio diseño, lo que permite personalizar la forma en que se muestran los datos (por ejemplo, texto, imagen, iconos, etc.).
En XML, un ListView se declara normalmente así:

```
<ListView
    android:id="@+id/myListView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

Por sí solo, el ListView no sabe qué datos mostrar ni cómo mostrarlos. Ahí es donde entra en juego el Adapter.

Adapter

Un Adapter (adaptador) es una clase que actúa de intermediario entre los datos y la vista.
Su función es tomar los datos (por ejemplo, una lista de objetos Java) y convertirlos en vistas que el ListView pueda mostrar.
Podemos imaginar al Adapter como un “traductor” que transforma los datos en celdas visuales.
Existen varios tipos de adapters, dependiendo de la complejidad y personalización que se necesite:

- ArrayAdapter → útil para listas sencillas de texto.
- SimpleAdapter → permite mostrar datos con estructuras simples en layouts personalizados.
- BaseAdapter → el más flexible, ideal para personalizar completamente las celdas.

Cuando necesitamos un control total sobre cómo se muestra cada elemento, usamos BaseAdapter, que requiere implementar algunos métodos fundamentales.

Métodos fundamentales de un BaseAdapter

Al extender la clase BaseAdapter, debemos sobrescribir cuatro métodos obligatorios:

- getCount()

Indica cuántos elementos hay en la lista.
Devuelve el tamaño del conjunto de datos.

```
 public int getCount() {
    return this.users.size();
}
```

- getItem(int position)

Devuelve el objeto de datos correspondiente a la posición indicada.

```
public User getItem(int position) {
    return this.users.get(position);
}
```

- getItemId(int position)

Devuelve un identificador único para el elemento.
Si no usamos un id real, podemos devolver simplemente la posición.

```
public long getItemId(int position) {
    return position;
}
```

- getView(int position, View convertView, ViewGroup parent)

Este método crea o actualiza la vista (celda) que se muestra en la lista.
Aquí es donde se “infla” el layout de cada fila y se asignan los valores correspondientes.

```
 public View getView(int position, View convertView, ViewGroup parent) {
    if (convertView == null) {
        LayoutInflater inflater = LayoutInflater.from(this.context);
        convertView = inflater.inflate(R.layout.row, null);
```

```
    }
    // Configurar los valores de la celda según los datos
    // Ejemplo: TextView tvName = convertView.findViewById(R.id.name);
    // tvName.setText(users.get(position).getName());
    return convertView;
}
```

El parámetro convertView se reutiliza para evitar crear nuevas vistas innecesariamente, mejorando el rendimiento.
Cuando convertView es null, significa que no existe una vista reciclada y debemos “inflar” una nueva a partir del layout XML.
Ejemplo
Supongamos que tenemos una lista de objetos User. Podríamos crear un adapter así:

```
public class UserListAdapter extends BaseAdapter {
    private final List<User> users;
    private final Context context;

    public UserListAdapter(Context context, List<User> users) {
        this.context = context;
        this.users = users;
    }
    @Override
    public int getCount() {
        return users.size();
    }
    @Override
    public User getItem(int position) {
        return users.get(position);
    }
    @Override
    public long getItemId(int position) {
        return position;
    }
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        if (convertView == null) {
            LayoutInflater inflater = LayoutInflater.from(context);
            convertView = inflater.inflate(R.layout.row_user, parent, false);
        }

        User user = getItem(position);
        TextView nameView = convertView.findViewById(R.id.userName);
        nameView.setText(user.getName());

        return convertView;
    }
}
```

Este adapter se encarga de tomar cada objeto `User` y mostrarlo dentro del layout `row_user.xml`, que podría incluir el nombre, foto, o cualquier otro dato.
Eventos en Listas
Las listas reaccionan automáticamente al evento de clic, sin necesidad de agregar botones dentro de cada

elemento. Para manejar las acciones del usuario, se usan listeners especiales:

- AdapterView.OnItemClickListener

Detecta cuándo el usuario pulsa un elemento de la lista.
Es similar al `setOnClickListener()` de un botón, pero recibe también la posición del ítem clicado.

```
 listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        User clickedUser = users.get(position);
        Toast.makeText(context, "Has pulsado a " + clickedUser.getName(),
Toast.LENGTH_SHORT).show();
    }
});
```

- AdapterView.OnItemLongClickListener

Detecta una pulsación larga sobre un elemento, útil para mostrar menús contextuales o confirmar acciones.
En resumen

- El ListView muestra la lista en pantalla.
- El Adapter convierte los datos en vistas.
- El BaseAdapter permite una personalización total de las celdas.

Se pueden añadir listeners para gestionar interacciones del usuario.
Gracias a esta estructura, Android puede mostrar grandes volúmenes de datos de forma eficiente y flexible, permitiendo diseños muy personalizados y optimizados en rendimiento.

20 de octubre

PETICIONES REST EN ANDROID

Las peticiones REST (Representational State Transfer) son la forma más común de comunicación entre una aplicación Android y un servidor web. A través de ellas, la app puede enviar datos, recibir información o sincronizarse con servicios externos como bases de datos o APIs. Para trabajar con este tipo de comunicación, Android se apoya en librerías que simplifican el proceso de enviar peticiones HTTP y manejar respuestas JSON.

Trabajar con JSON y objetos Java

Una de las formas más cómodas de manejar datos en Android es usando JSON (JavaScript Object Notation). Sin embargo, trabajar directamente con texto JSON puede ser incómodo y propenso a errores. Por eso, lo habitual es convertir el JSON en objetos Java y viceversa. Esto permite usar tipos de datos, getters, setters y toda la potencia de Java, pero sin dejar de comunicarnos en formato JSON con el servidor. Para realizar esta conversión (llamada serialización y deserialización), se utilizan librerías especializadas.

Librerías para convertir JSON ↔ Java

Las más conocidas son:

- **Jackson**: una librería muy completa y rápida para procesar JSON.
- **Gson**: creada por Google, más ligera y sencilla de usar.

En Android normalmente se usa Gson, porque se integra fácilmente y no requiere mucha configuración. Para añadirla, se hace a través de Gradle, en el archivo build.gradle del módulo de la app. En dependencies:
El objetivo de Gson es doble:

1. Convertir un objeto Java en JSON (cuando queremos enviar información al servidor).
2. Convertir un texto JSON recibido en un objeto Java (cuando recibimos datos del servidor).

Android Asynchronous Http Client

Para realizar peticiones HTTP (como GET, POST, PUT o DELETE), se usa la librería Android Asynchronous Http Client (de LoopJ), disponible en: http://loopj.com/android-async-http. Esta herramienta es muy útil porque maneja automáticamente la asincronía de las peticiones. Esto significa que las solicitudes al servidor se hacen en segundo plano, sin bloquear la interfaz del usuario. Así, la app puede seguir funcionando mientras

espera la respuesta del servidor. También se puede añadir mediante Gradle, tal como se indica en su documentación oficial.

Estructura de una petición REST

Android trabaja por eventos, es decir, la aplicación reacciona a acciones que ocurren: por ejemplo, cuando el usuario pulsa un botón o cuando llega una respuesta del servidor. Por eso, una petición REST se compone de dos fases principales:

1. Enviar la petición:
   Se configuran los parámetros necesarios (URL, tipo de método —POST, GET, etc.— y los datos).
   Luego, la petición se envía en un hilo independiente.
2. Recibir la respuesta o el error:
   Cuando llega la respuesta, se dispara un evento que debemos gestionar.
   Esto se hace implementando una clase que maneje el resultado, distinguiendo entre éxito y fallo.

Manejo de la respuesta

Para gestionar la respuesta se crea una clase que extiende de AsyncHttpResponseHandler.
Esta clase tiene dos métodos clave:

- onSuccess() → se ejecuta cuando la petición ha sido válida.
- onFailure() → se ejecuta cuando ocurre un error.

El criterio de éxito o error depende del código HTTP que devuelve el servidor:

- 2xx o 3xx → peticiones correctas (éxito).
- 4xx o 5xx → peticiones erróneas (fallo).

Ejemplo

El ejemplo básico propuesto es una petición al módulo “welcome”, que simplemente devuelve un mensaje de bienvenida. El proceso consiste en:

1. Crear la clase de datos que vamos a enviar y la clase de datos que recibimos.

```
public class WelcomeRequest {
    private String name;

    public WelcomeRequest(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
public class WelcomeResponse {
    private String message;

    public String getMessage() {
        return message;
    }
}
```

Cuando usamos Gson para convertir esta clase a JSON, obtenemos algo como:

```
{"name": "Pepe"}
```

2. Crear la clase que manejará la respuesta.

```
public class WelcomeResponseHandler extends AsyncHttpResponseHandler {
    private final Gson gson;
    private final Context context;

    public WelcomeResponseHandler(Context context, Gson gson) {
        this.context = context;
        this.gson = gson;
    }

    @Override
    public void onSuccess(int statusCode, Header[] headers, byte[] responseBody) {
        if (statusCode == 200) {
            // Convertir el JSON de la respuesta a un objeto WelcomeResponse
            WelcomeResponse response = this.gson.fromJson(
                new String(responseBody),
                WelcomeResponse.class
            );

            // Mostrar el mensaje recibido
            Toast.makeText(context, response.getMessage(), Toast.LENGTH_SHORT).show();
        } else {
            // Otros códigos de éxito (redirecciones, etc.)
            Toast.makeText(context, "Respuesta inesperada del servidor",
Toast.LENGTH_SHORT).show();
        }
    }

    @Override
    public void onFailure(int statusCode, Header[] headers, byte[] responseBody,
Throwable error) {
        Toast.makeText(context, "Error en la conexión: " + error.getMessage(),
Toast.LENGTH_SHORT).show();
    }
}
```

3. Enviar la petición usando AsyncHttpClient:

```
Gson gson = new Gson();
AsyncHttpClient client = new AsyncHttpClient();

try {
    client.post(
        this, // contexto de la Activity
        "http://api.battleship.tatai.es/v2/welcome", // URL del endpoint
        new StringEntity(gson.toJson(new WelcomeRequest("Pepe"))), // JSON con nombre
        "application/json", // tipo de contenido
        new WelcomeResponseHandler(this, gson) // manejador de la respuesta
    );
} catch (UnsupportedEncodingException e) {
    throw new RuntimeException(e);
```

```
}
```

4. Aquí se ve el flujo completo:
    a. Creas el JSON a partir del objeto WelcomeRequest usando Gson.
    b. Envías la petición al servidor con AsyncHttpClient.post().
    c. La petición se ejecuta en segundo plano (hilo de red asíncrono).
    d. Cuando el servidor responde:
        i. Si el estado es correcto (200 OK), se llama a onSuccess().
        ii. Si hay error (4xx o 5xx), se llama a onFailure().
    e. En onSuccess(), Gson convierte el JSON recibido a un objeto WelcomeResponse.
    f. Finalmente, se muestra el mensaje en la interfaz (por ejemplo, con un Toast).

Permisos en Android

Android es muy estricto con la seguridad, por lo que hay que declarar explícitamente los permisos que necesita la aplicación. Para poder realizar peticiones por Internet, se debe añadir esta línea al archivo:

```
AndroidManifest.xml:
<uses-permission android:name="android.permission.INTERNET" />
```

Si no se incluye este permiso, las peticiones REST fallarán, ya que Android bloqueará cualquier intento de conexión externa (entrará onFailure throwable..)

API del proyecto

Cada proyecto o servidor suele tener su propia API, que describe:

- Las direcciones (endpoints) de cada módulo.
- Qué método HTTP usa (GET, POST, etc.).
- Qué parámetros se envían y qué datos se esperan de vuelta.

Consultar la especificación de la API es esencial para saber cómo deben estructurarse las peticiones y qué respuestas se recibirán.