Bases de Datos y herramientas de desarrollo

Projecte Integrat de Software (PIS)

Universitat de Barcelona

victor.campello@ub.edu
c.izquierdo@ub.edu
carlos.martinisla@ub.edu

3-5 de Marzo de 2020

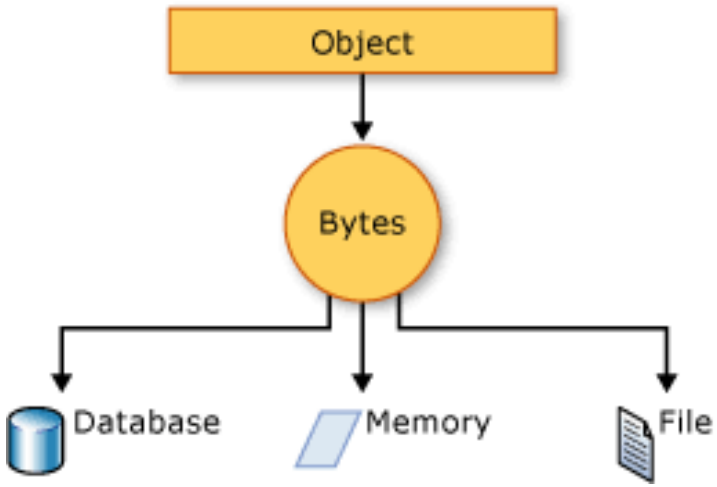
Overview

Recordatorio

- **Activity**: Interfaz en forma de ventana o subventana que el usuario ve y con la que interactúa. Por ejemplo, la ventana principal de vuestra app.
- **Service**: Operación con la que el usuario no interactúa, que se ejecuta de fondo. Por ejemplo, la operación de descargar un fichero de internet.
- **Broadcast Receiver**: Para recibir y enviar mensajes sobre eventos (mediante Intents) a otras aplicacions o al sistema de Android. Por ejemplo, cuando queda poca batería en el dispositivo.
- **Content Provider**: La forma de acceder a los datos del dispositivo y compartirlos entre aplicaciones.

Serialización

La serialización (o marshalling en inglés) consiste en un proceso de codificación de un objeto en un medio de almacenamiento (como puede ser un archivo, o un buffer de memoria) con el fin de transmitirlo a través de una conexión en red como una serie de bytes o en un formato humanamente más legible como XML o JSON, entre otros.



Identificar necesidades

1. ¿Cuánto espacio necesita nuestros datos?
2. ¿Qué tipo de datos necesitamos almacenar?
3. ¿Serán únicamente accesibles desde la aplicación?

Opciones de serialización y almacenaje de datos

1. **Bases de datos** Permite guardar grandes cantidades de datos. Pueden ser locales o en la nube. El segundo caso nos será útil para crear una aplicación social.
2. **GSON** Permite codificar una instancia de un objeto Java / Kotlin en formato JSON.
3. **SharedPreferences** Permite guardar una colección relativamente pequeña de pares clave-valor de forma local, proporciona métodos simples de lectura y escritura de esos pares.

Introducción a Firebase

Firebase se trata de una plataforma móvil creada por Google, cuya principal función es desarrollar y facilitar la creación de apps de elevada calidad de una forma rápida. Entre sus características principales encontramos:

- Crear bases de datos de manera sencilla
- Testear la aplicación
- Obtener analytics de la aplicación
- Controlar la monetización

Firebase

Implementación y Setup

Pasos a seguir para la implementación:
`https://firebase.google.com/docs/android/setup?hl=es-419`

Cloud Firestore

Una vez creado el proyecto y sincronizado con tu Android Studio, crearemos la base de datos con la herramienta **Cloud Firestore**.

3. Add the Cloud Firestore Android library to your `app/build.gradle` file:

```
implementation 'com.google.firebase:firebase-firestore:21.4.0'
```

Inicializamos Cloud Firestore:

```
// Access a Cloud Firestore instance from your Activity
val db = FirebaseFirestore.getInstance()
```

Crear datos

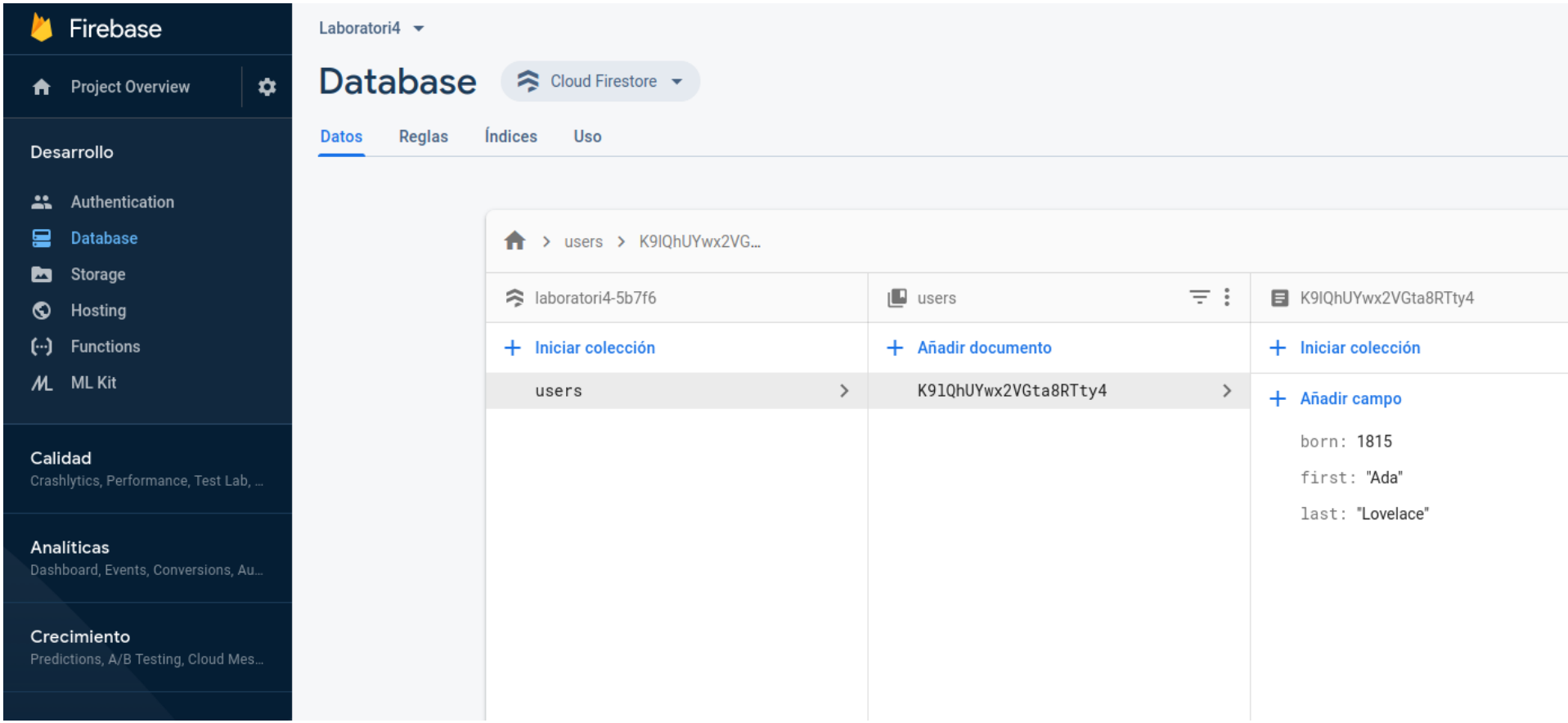
- Creamos Hash map (key, value) en la variable user
- Añadimos la nueva colección con una ID definidida ("users").
- **.add** añade los valores en la colección definida para una ID automática.

```
// Create a new user with a first and last name
val user = hashMapOf(
        "first" to "Ada",
        "last" to "Lovelace",
        "born" to 1815
)

// Add a new document with a generated ID
db.collection("users")
    .add(user)
    .addOnSuccessListener { documentReference ->
        Log.d(TAG, "DocumentSnapshot added with ID: ${documentReference.id}")
    }
    .addOnFailureListener { e ->
        Log.w(TAG, "Error adding document", e)
    }
```

Comprobar los cambios

Téneis que ir a Firebase Console y observar si habéis introducido cambios.



Leer datos

El comando **.get** nos permite obtener los datos que hay dentro de la colección "users".

```
db.collection("users")
        .get()
        .addOnSuccessListener { result ->
            for (document in result) {
                Log.d(TAG, "${document.id} => ${document.data}")
            }
        }
        .addOnFailureListener { exception ->
            Log.w(TAG, "Error getting documents.", exception)
        }
```

Actualización

Como hemos visto en la diapositiva anterior cuándo creamos información en la base de datos, se generaba una ID automatica. En algunos casos, es mejor ordenar las carpetas con sus correspondientes nombres:

```
val city = hashMapOf(
        "name" to "Los Angeles",
        "state" to "CA",
        "country" to "USA"
)

db.collection("cities").document("LA")
        .set(city)
        .addOnSuccessListener { Log.d(TAG, "DocumentSnapshot successfully written!") }
        .addOnFailureListener { e -> Log.w(TAG, "Error writing document", e) }
                                                                    DocSnippe
```

Si usamos **.document** estaremos dando una ID determinada a nuestra entrada en la colección cities. El **.set** sirve para añadir toda la información en la documentación creada.

Updating

Si por ejemplo, queremos añadir algún campo que anteriormente no habíamos introducido en la información del usuario, con el siguiente código podemos añadir nuevas instancias. En este caso hemos creado la instancia **capital**, hemos seleccionado a que usuario queremos añadirle la instancia **'BJ'** y hemos seleccionado la opción **SetOptions.merge()** para unir los dos hashmaps.

```
// Update one field, creating the document if it does not already exist.
val data = hashMapOf("capital" to true)

db.collection("cities").document("BJ")
        .set(data, SetOptions.merge())
```

Obtener información: "Query"

Una vez tenemos datos introducidos en la base de datos, tambien necesitaremos consultar en algun momento los datos que tengamos almacenados. Igual que en SQL, hay que realizar una **query**.

```
val cities = db.collection("cities")

val data1 = hashMapOf(
        "name" to "San Francisco",
        "state" to "CA",
        "country" to "USA",
        "capital" to false,
        "population" to 860000,
        "regions" to listOf("west_coast", "norcal")
)
cities.document("SF").set(data1)

val data2 = hashMapOf(
        "name" to "Los Angeles",
        "state" to "CA",
        "country" to "USA",
        "capital" to false,
        "population" to 3900000,
        "regions" to listOf("west_coast", "socal")
)
cities.document("LA").set(data2)
```

Ejemplos de query

En la primera imagen, guardamos el valor de la query en una variable. En la segunda imagen, con la función **.get**, mostramos los valores obtenidos en la query. Para mostrar valores sin listados, es más sencillo crear variables y asociarlas a un View.

```
val capitalCities = db.collection("cities").whereEqualTo("capital", true)
```

```
db.collection("cities")
        .whereEqualTo("capital", true)
        .get()
        .addOnSuccessListener { documents ->
            for (document in documents) {
                Log.d(TAG, "${document.id} => ${document.data}")
            }
        }
        .addOnFailureListener { exception ->
            Log.w(TAG, "Error getting documents: ", exception)
        }
```

Operadores de consulta

Estos son los tres operadores de consulta en Android/Firebase.

```
citiesRef.whereEqualTo("state", "CA")
citiesRef.whereLessThan("population", 100000)
citiesRef.whereGreaterThanOrEqualTo("name", "San Francisco")
```

Válido: Filtros de rango en solo un campo

Web | Swift | Objective-C | Java Android | Kotlin Android | Java

```
citiesRef.whereGreaterThanOrEqualTo("state", "CA")
        .whereLessThanOrEqualTo("state", "IN")
citiesRef.whereEqualTo("state", "CA")
        .whereGreaterThan("population", 1000000)
```

No válido: Filtros de rango en diferentes campos

Web | Swift | Objective-C | Java Android | Kotlin Android | Java

```
citiesRef.whereGreaterThanOrEqualTo("state", "CA")
        .whereGreaterThan("population", 100000)
```

GSON

JSON es un formato de datos de texto muy popular que se usa para intercambiar datos en las aplicaciones web y móviles modernas. Por ejemplo, en SQL, Azure o extremos de la API Rest. Si deseamos acceder a datos de otras BD (SQL Server), necesitaremos crear esos archivos JSON con la función (añadir datos/leer datos) a través de nuestro código Kotlin. Para eso existe **GSON library**, que nos permite traducir classes Kotlin a JSON, y viceversa.

Implementación

Lo primero que tenemos que hacer es añadir la libreria a las dependencias de **app/build.gradle**, tal que así:

```
dependencies {
    ...
    implementation 'google.com.code.gson:gson:2.8.5'
}
```

Después hemos de crear una clase usuario.

```
data class Usuario(val _first_name: String? = null,
                   val _last_name: String? = null,
                   val _email: String?,
                   val _address: String? = null,
                   val _country: String? = null)
```

Parsing

Creamos una instancia de nuestra clase:

```
val estudiante = Usuario(
    _first_name: "Cristian",
    _last_name: "Izquierdo",
    _email: "c.izquierdo@ub.edu",
    _address: "UB",
    _country: "Spain,")
```

Parseamos a JSON con la función de la libreria **Gson().toJson()**.

```
val estudiantejson = Gson().toJson(estudiante)
```

A este proceso de transformación a JSON format se le llama **serialización (o marshalling)**.El proceso inverso (**deserialización o unmarshalling** se realiza con la función **Gson().fromJson()**

SHARED PREFERENCES

Android proporciona otro método alternativo diseñado específicamente para administrar la serielización de datos: las preferencias compartidas o shared preferences. Cada preferencia se almacenará en forma de clave-valor, es decir, cada una de ellas estará compuesta por un identificador único (p.e. email) y un valor asociado a dicho identificador (p.e. prueba@email.com). Además, los datos se guardarán en ficheros XML, formato interpretable de la misma forma que JSON.Este método es apto para guardar **pequeños volúmenes de información**, como por ejemplo el estado de la aplicación.

Ejercicio

1. Crea una acivity "login" con nombre, apellido y email y su correspondiente EditText para cada campo.
2. Añade un botón 'Save' que guarde en Shared Preferences los campos anteriores.
3. Crea una acivity "recovery" con nombre, apellido y email en tres diferentes TextView.
4. Crea una función que intente leer las Shared Prefrences en "OnCreate" de la activity "login", las recupere si es posible y lance un intent hacia la activity "recovery", mostrando las Shared Preferences en los correspondientes TextView de la activity "recovery".

Ejercicio

1. Al hacer click en el botón "Save", guardaremos nuestros datos.
2. En el método "OnCreate" comprobaremos si los campos deseados existen.
3. Si existen, recuperaremos la información en una nueva Activity.

Botón save

```
fun savePrefs(view: View) {

    val name_text = name_sharedprefs.text.toString()
    val last_name_text = last_name_sharedprefs.text.toString()
    val email_text = email_sharedprefs.text.toString()

    val editor =
        getSharedPreferences( name: "MyPrefs", Context.MODE_PRIVATE).edit()
    editor.putString("name", name_text)
    editor.putString("last_name", last_name_text)
    editor.putString("email", email_text)

    editor.apply()

    Toast.makeText( context: this@MainActivity_sharedprefs,  text: "New user saved!", Toast.LENGTH_SHORT).show()

}
```

Comprobar SharedPrefs y Intent

```
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main_sharedprefs)
if (getSharedPreferences( name: "MyPrefs", Context.MODE_PRIVATE).contains("name")){
    Toast.makeText( context: this@MainActivity_sharedprefs, text: "User found!"
        , Toast.LENGTH_SHORT).show()

    val name = getSharedPreferences( name: "MyPrefs", Context.MODE_PRIVATE).getString("name", "")
    val last_name = getSharedPreferences( name: "MyPrefs", Context.MODE_PRIVATE).getString("last_name", "")
    val email = getSharedPreferences( name: "MyPrefs", Context.MODE_PRIVATE).getString("email", "")

    val intent = Intent( packageContext: this, MainActivity_sharedprefs_recovery::class.java)
    intent.putExtra( name: "name", name)
    intent.putExtra( name: "last_name", last_name)
    intent.putExtra( name: "email", email)

    startActivity(intent)
}
```

Mostrar SharedPrefs

```
class MainActivity_sharedprefs_recovery : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main_sharedprefs_recovery)
        val name_text:String = intent.getStringExtra( name: "name")
        val last_name_text:String = intent.getStringExtra( name: "last_name")
        val email_text:String = intent.getStringExtra( name: "email")

        recovery_name.text = name_text
        recovery_lastname.text = last_name_text
        recovery_email.text = email_text

    }
}
```

Git basics

Git es un proyecto open-source que permite mantener un control de versiones de nuestro código. Es el estándar en desarrollo de software. Existen varias plataformas que usan este sistema: GitHub, GitLab, Bitbucket, etc. Nosotros usaremos GitHub.
Git es un sistema distribuido, es decir, cuando trabajamos con el código tenemos una copia en local (*fork*) y cuando hemos terminado de hacer cambios, podemos subir los cambios al servidor con facilidad.
Podéis encontrar un manual con varios detalles sobre Git en:

https://git-scm.com/book/

Git basics

Empezamos a usar Git una vez tenemos claras las características de nuestra aplicación y las funcionalidades que queremos añadir.
Git dispone de un herramienta en línea de comandos *git* con la cual nos comunicaremos con los diferentes servidores y haremos todas las operacions en nuestro proyecto.
Los comandos básicos son:

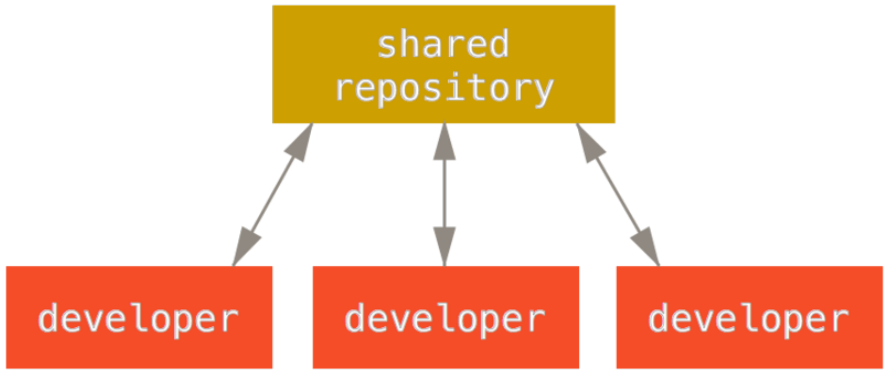
- *git init*: Inicializa un repositorio desde cero. Crea la carpeta **.git**, que contiene los ficheros necesarios para el repositorio.
- *git clone <url>*: Descarga un repositorio existente en una nueva carpeta con el mismo nombre.
- *git add <nombre del fichero>*: Añade el fichero o ficheros a la lista de cambios que vamos a incluir en nuestra confirmación o commit.
- *git commit -m <mensaje>*: Confirma los ficheros añadidos y crea un nuevo evento en visible para el resto de desarrolladores.
- *git status*: Muestra los cambios sin confirmar, tanto añadidos como sin añadir.

Git basics

- *git fetch <remote>*: Consulta los cambios con el repositorio remoto. Es como un botón de actualizar.
- *git merge <remote>*: Fusiona los cambios remotos con tus cambios locales. Aquí pueden surgir conflictos si varias personas han modificado el mismo fichero "a la vez".
- *git pull <remote>*: Ejecuta *git fetch* seguido de *git merge*. Se suele usar este comando, salvo si se quieren ver los cambios antes de descargarlos.
- *git push origin master*: Sube todos los cambios al repositorio remoto. Importante: solo se permite esta acción cuando tenemos todos los cambios remotos.
- *git checkout <branch>*: Cambia de una rama a otra, deshace los cambios locales en ficheros o crea nuevas ramas (usando **-b** delante de *<branch>*).

Flujo de trabajo

A continuación veremos el flujo de trabajo que utilizaremos en esta asignatura: flujo centralizado.



Tendremos un repositorio centralizado en un servidor y todos trabajaremos a la vez en él.

John y Jessica

Veamos el siguiente ejemplo, para entender cómo contribuir a un repositorio.
Tenemos dos desarrolladores, John y Jessica. Estos empiezan por clonarse un repositorio existente y modificar o crear ficheros:

```
# John's Machine
$ git clone john@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'remove invalid default value'
[master 738ee87] remove invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

John y Jessica

Jessica modifica un fichero, lo confirma y lo sube al repositorio:

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

```
# Jessica's Machine
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

John y Jessica

Si John intenta ahora subir sus cambios, tendrá un error. Recordad que el comando *push* solo te permite subir los cambios si tienes todos los cambios del remoto (los cambios de Jessica).

```
# John's Machine
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

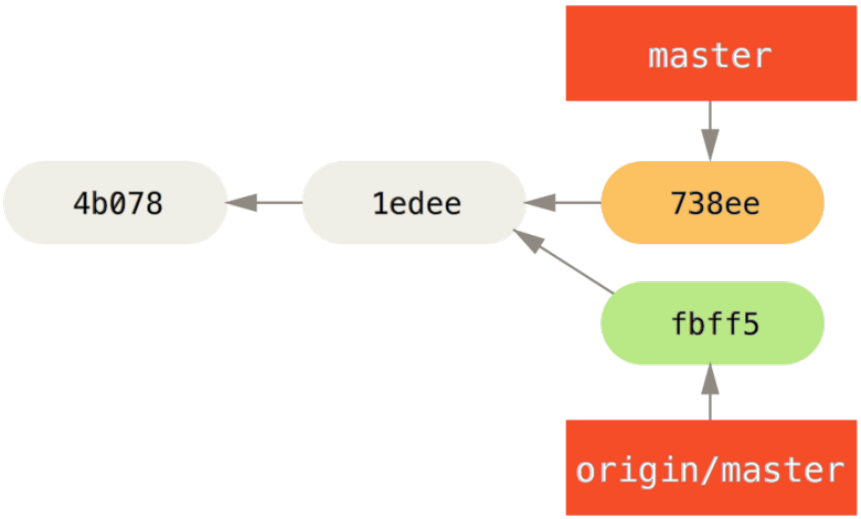
Git no permite combinar los cambios en remoto. Se deben combinar localmente antes de subirlos.

John y Jessica

John necesita descargarse los cambios de Jessica, combinarlos con los suyos y subirlo todo.

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

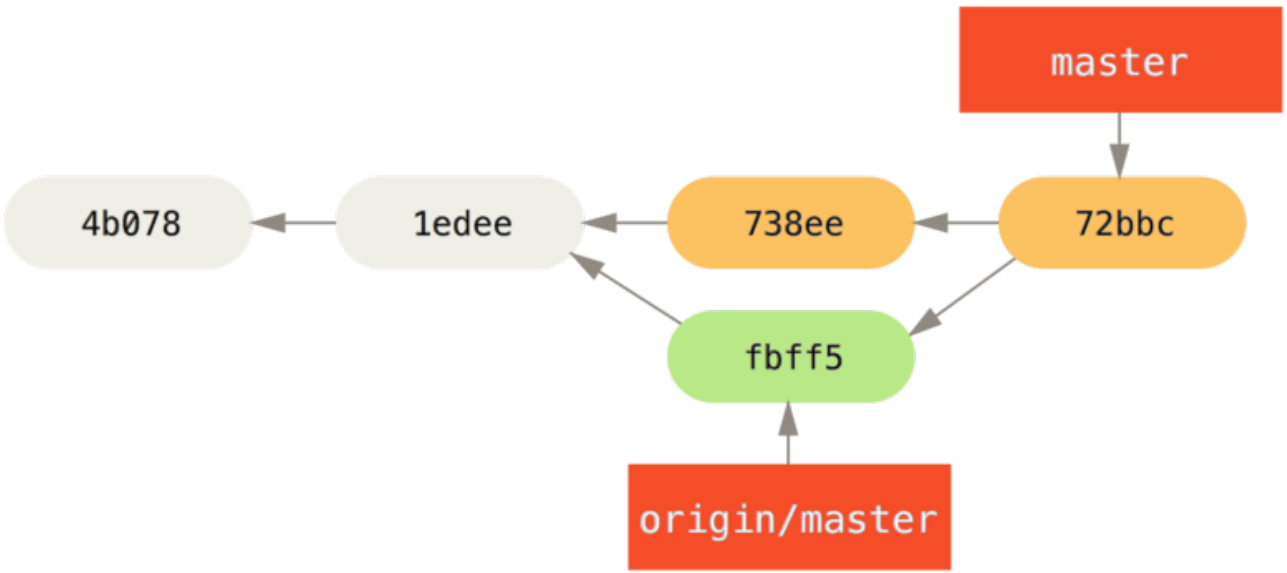
En este momento, para John, el repositorio tiene la siguiente configuración:



John y Jessica

Ahora John puede combinar los cambios de los dos commits. Esto lo hará Git automáticamente si usamos *git pull origin*.

```
$ git merge origin/master
Merge made by the 'recursive' strategy.
 TODO |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```



John y Jessica

Finalmente, John será capaz de subir el merge que ha hecho localmente

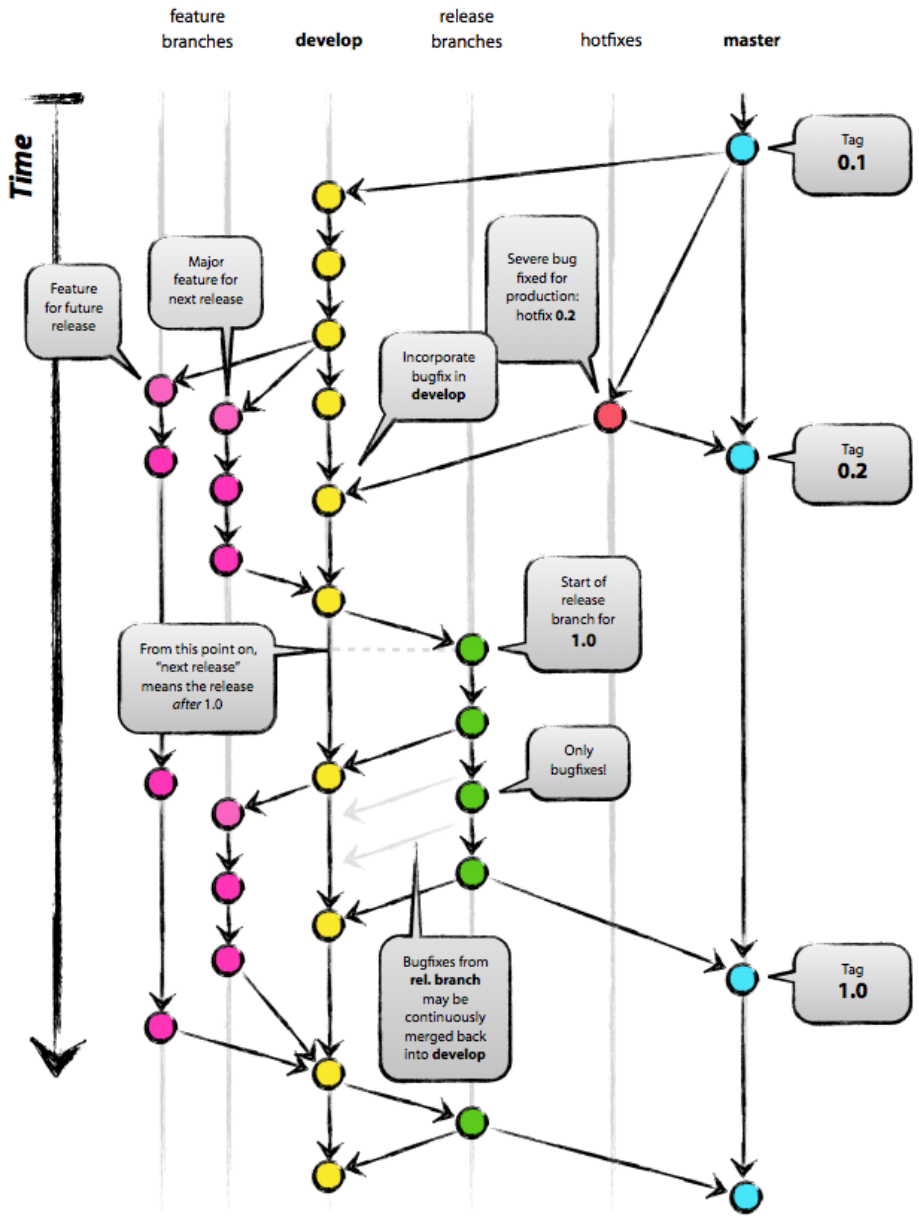
```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

En la práctica

En la práctica, el proceso habitual debería ser el de John y Jessica, añadiendo alguna rama más si fuese necesario.
Un ejemplo de un flujo de trabajo real que usaría un experto se puede encontrar aquí:

nvie.com/posts/a-successful-git-branching-model/

En la práctica



En la práctica

Durante el desarrollo de vuestro proyecto no necesitaremos tantas ramas, como en el caso anterior, ya que seréis un máximo de 4 desarrolladores. Además, vuestra aplicación es un proyecto nuevo, por lo que no tendrá una versión en producción que puedan estar usando los usuarios mientras desarrolláis nuevas características.

Importante: necesitáis tener en cuenta dos cosas

- La entrega bisemanal debe estar en la rama **master**. No miraremos otras ramas.
- Debéis etiquetar el *commit* hasta el cual se considera la entrega bisemanal de la siguiente forma:

  git tag -a sprintX
  git push origin sprintX

  donde X será el número de entrega del sprint.

Vuestro tablero

En el trello deberéis añadir una columna (**Backlog**) con todas las ideas o tareas que tenéis pendientes y una columna por cada Sprint (**SprintX**) con las tareas que desarrollaréis en esa entrega. También os puede ayudar tener una columna de **Work in progress** para saber qué está haciendo cada uno, en caso que haya tareas que dependan entre ellas o para evitar hacer la misma tarea.

End