

Android

Y amigos

Contenidos

- Activity & Fragment
- Sistemas de Construcción
- Conectividad
- Java puro y Java Android
- ¡Eh, esto funciona! - The App

Activity & Fragment

Cuándo usar cual
Beneficios
Problemas comunes
Patrones generales
Comunicación

Activities



Activities + Fragments



Why bother

- Una activity es componente con UI, generalmente única. Son los contextos más comunes
- Es normal que partes de la UI se reciclen: Banners de publicidad, cabeceras, etc...
- Podemos reutilizar partes de una vista separando en distintos ficheros usando `@include`
- ¿Qué ocurre con la lógica?

Sólo vista

```
<LinearLayout xmlns:android="." . ."
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <include layout="@layout/my_shared_layout"/>

</LinearLayout>
```

Vista + Lógica

Opción 1: Utilizando subclases de vistas

- Podemos definir nuevos atributos
- Controlar el método de dibujado
- Mucho control sobre el layout

```
class VistaCustom extends LinearLayout {  
    ...  
    //código para un banner publicitario  
    ...  
}
```

<http://developer.android.com/training/custom-views/index.html>

Vista + Lógica

```
<LinearLayout xmlns:android=". . ."
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:orientation="vertical">

    <com.presentacion.VistaCustom x:attr1="" ... />

</LinearLayout>
```

Vista + Lógica, ¿problemas?

- Alto acoplamiento entre el controlador y la vista
- Las vistas deberían tener poco/ningún estado, free bugs
- Ciclo de vida propio de una vista (no hay segundo plano)
- En general no tiene mucho sentido si no vamos a personalizar el método `onDraw()`.

Fragments

- Un fragmento es un componente lógico, vista opcional
- Las actividades se componen de varios fragmentos
- Un fragmento siempre vive en una actividad
- Los fragmentos se pueden componer de fragmentos
- A efectos prácticos son una máquina de estados
- Controlarlos correctamente es (~~@!#\$%~~) complicado

Vista + Lógica

Opción 2: Utilizando fragmentos

```
class FragmentCustom extends Fragment {  
    onCreateView(){  
        inflar layouts, inicializar vistas, etc.  
    }  
    ...  
    //código para un banner publicitario  
    ...  
}
```

<http://developer.android.com/guide/components/fragments.html>

Fragments

```
MyActivity extends Activity {  
  
    public void onCreate(bundle){  
        Fragment fragment = FragmentCustom.newInstance(3);  
        getFragmentManager()  
            .beginTransaction()  
            .add(R.id.base_fragment_container, fragment)  
            .commit();  
    }  
  
}
```

Fragments

```
Fragment fragment = FragmentCustom.newInstance(3);
```

- Los fragmentos **NO** pueden tener constructores
 - constructor + giro de pantalla = crash
- Todos los parámetros de los fragmentos deben estar en un Bundle.
- El patrón `MyFragment.newInstance(args)` imita un constructor

Fragments

```
public static UserFragment newInstance(int id) {  
    UserFragment user = new UserFragment(); //no params  
    Bundle args = new Bundle();  
    args.putInt(USER_ID, id % 100);  
    user.setArguments(args);  
    return user;  
}
```

Fragments

```
getFragmentManager()  
    .beginTransaction()  
    .add(R.id.base_fragment_container, fragment)  
    .commit();
```

Normalmente las actividades se componen en ejecución

También es posible usar la forma declarativa (xml)

Después de realizar el commit comienza el ciclo de vida del fragmento

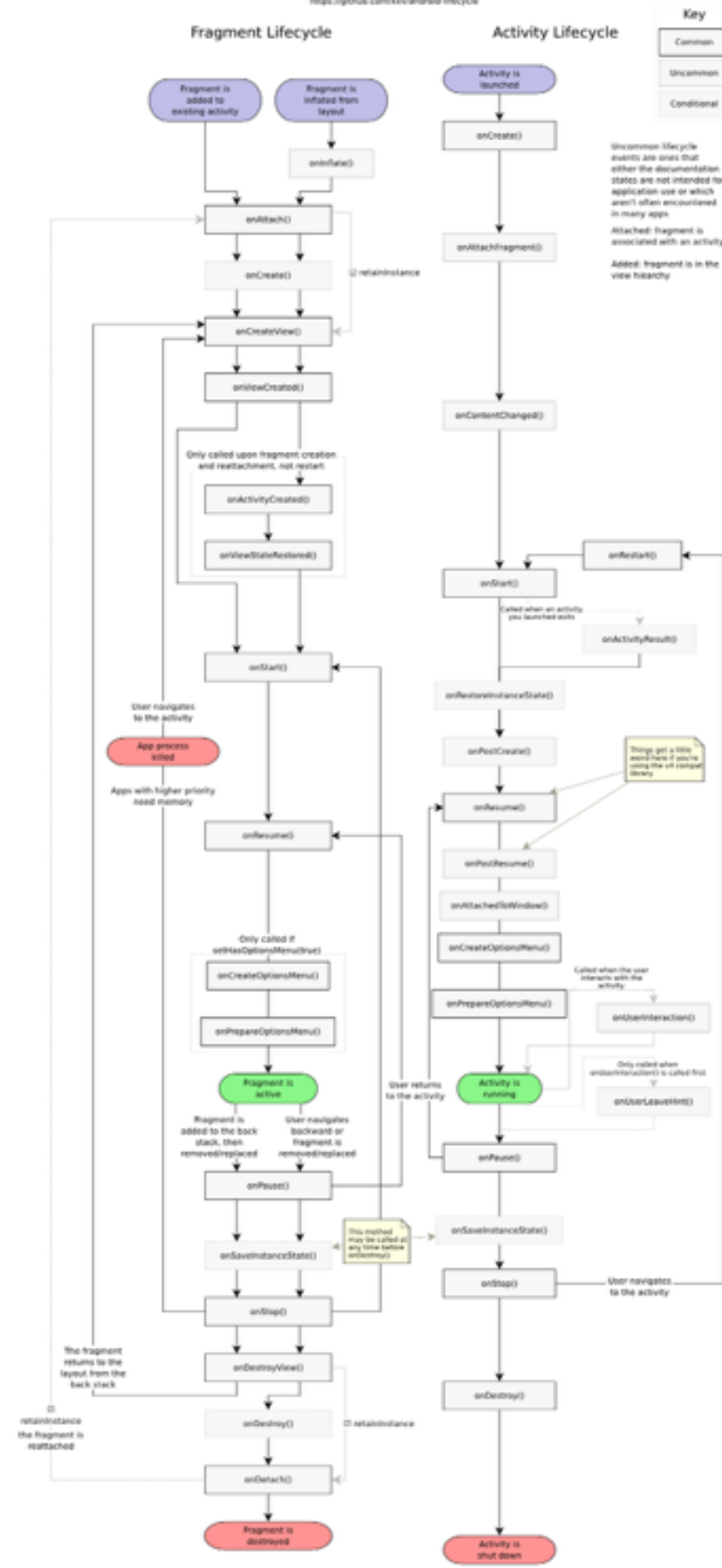
COMIENZA EL CICLO DE VIDA

Fragments

- El ciclo de vida es muy complejo, la documentación oficial es incompleta
- Android hace muchas suposiciones sobre como vas a usar un fragmento
- Los fragmentos tienen stack (igual que las actividades)
- El ciclo de vida completo se puede consultar en: <https://github.com/xxv/android-lifecycle>

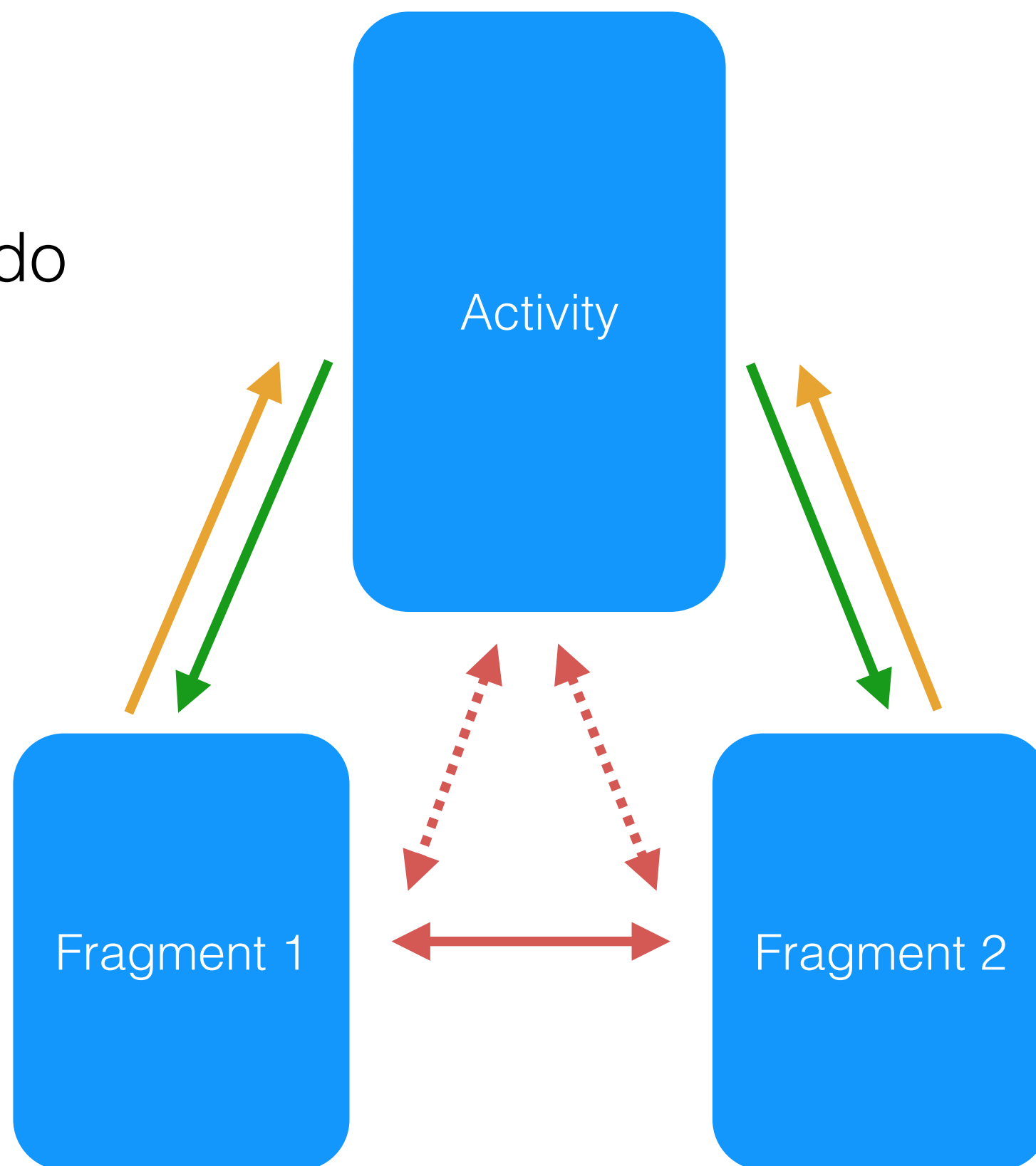
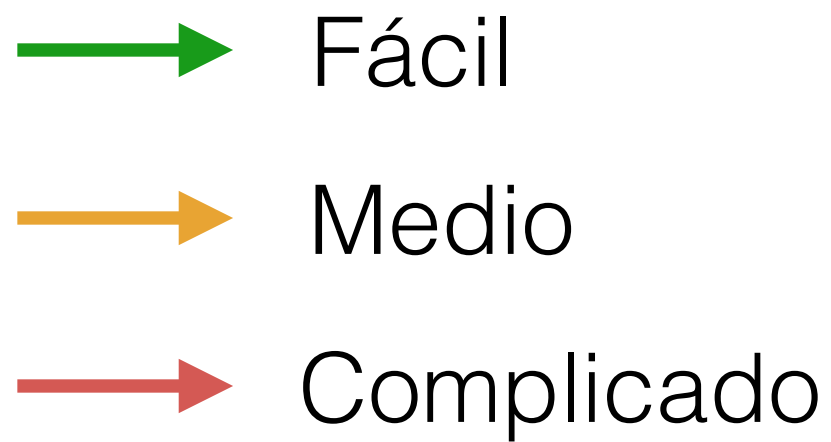
The Complete Android Activity/Fragment Lifecycle

v0.9.0 2018-04-22 Steve Pomeroy <steveg@thelevelup.com>
CC BY-SA 4.0
<https://github.com/stevepomeroy/android-lifecycle>



Comunicación

- Los fragmentos necesitan información de la actividad (key events) - fácil
- Los fragmentos necesitan compartir y mutar información - tedioso
- ¿Cuál es la mejor forma de solucionarlo?



Comunicación

- Opción 1 - bad: Casting

```
public class UserFragment extends Fragment {  
    public void onAttach(Activity activity) {  
        super.onAttach(activity);  
        MainActivity ac = (MainActivity) activity;  
        ac.someMethod();  
    }  
}
```

Comunicación

- Opción 1 - bad: Casting

El fragmento ya no es reusable

Alto acoplamiento

Leaks de contextos = crash

Comunicación

- Opción 2 - meh: Interfaces

```
public class MainActivity extends Activity implements Actions {  
    public interface Actions {  
        void doSomething();  
    }  
}
```

Comunicación

- Opción 2 - meh: Interfaces

```
public class UserFragment extends Fragment {  
    public void onAttach(Activity activity) {  
        super.onAttach(activity);  
        Actions ac = (Actions) activity;  
        ac.doSomething();  
    }  
}
```

Comunicación

- Opción 2 - meh: Interfaces

El fragmento se puede reutilizar, siempre que la actividad implemente la interfaz

Bajo acoplamiento

Verbose, interfaz que no aporta mucho

Leaks de contextos = crash (Sigue siendo una referencia a la actividad)

Comunicación

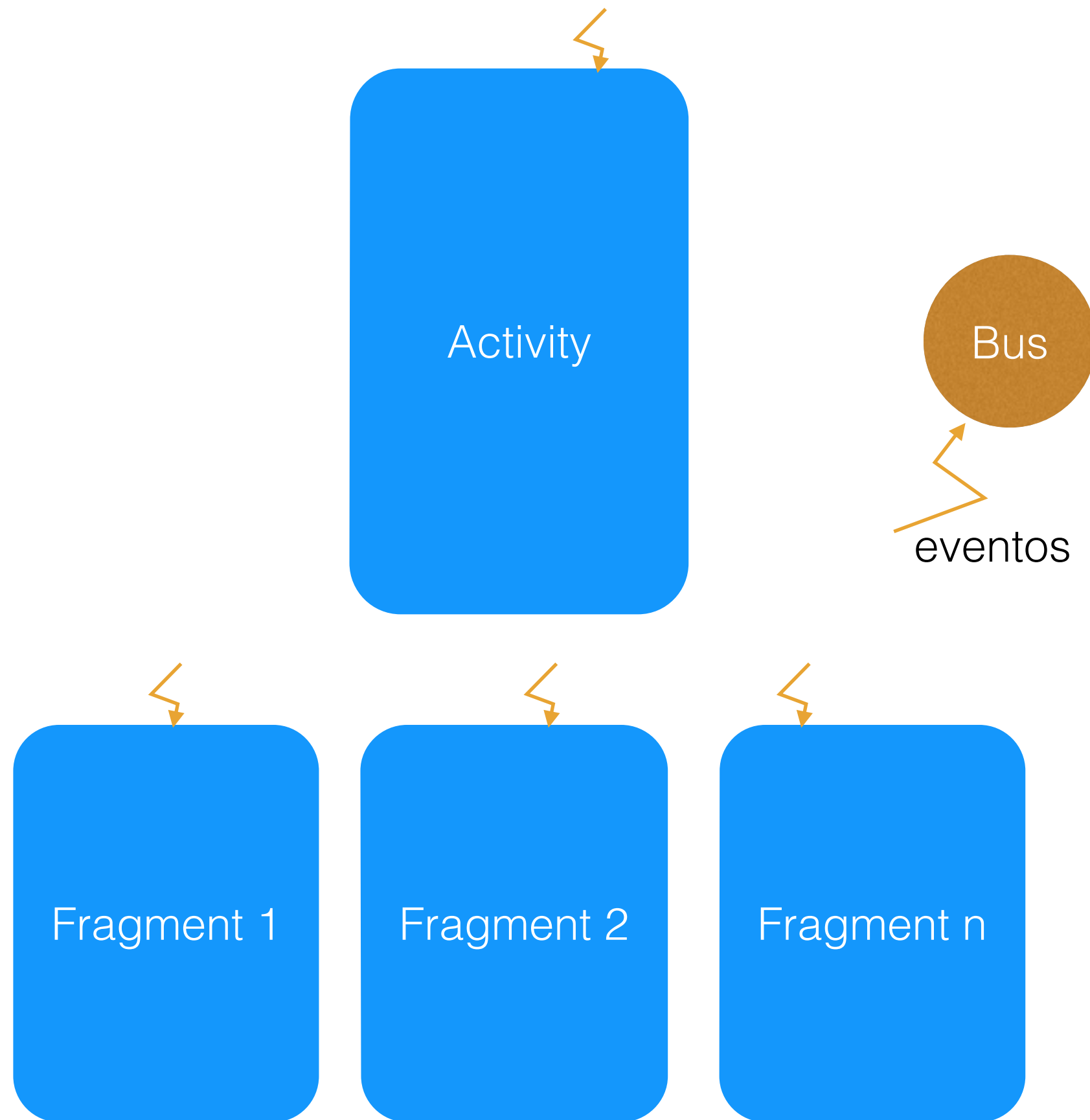
- Opción 3 - best: Bus de eventos

No es posible sin librerías externas (Otto)

La actividad se registra a eventos producidos por los fragmentos y viceversa

Los fragmentos se registran a eventos de otros fragmentos

No son necesarias referencias fuertes ni código extra



xq

¿Preguntas?

xq

Gradle & Android Studio

Sistemas de construcción

Ant

Maven

Gradle

Android Studio

Proyectos con Android
Studio



Why bother

- Los sistemas de construcción son imprescindibles
- Gestión de dependencias
- Variantes de construcción
- Paso automático de pruebas
- Todo lo que se te ocurra (scripts)

Conceptos básicos

- Android no tenía un sistema de construcción propio
- Originalmente basado en Ant (eclipse)
- Es posible usar Maven
- La mejor opción es Android Gradle Plugin

Ant

- Basado en tareas
- Definido en XML
- Muy libre
- No tiene gestión de dependencias por defecto
- Es difícil hacer tareas portables

Maven

- Similar a Ant respecto a las tareas (goals)
- Mucho más restringido, configuración de plugins
- Portable
- Gestión de dependencias
- Archivos Pom que llenan pantallas

Gradle

- Un paso intermedio entre Ant y Maven
- Basado en Tareas
- Muy flexible, utiliza groovy como lenguaje
- Muy conciso
- Gestión de dependencias
- Integración con el IDE - Android Studio

Android Plugin

build.gradle

```
apply plugin: 'com.android.application'
```

```
android {  
    compileSdkVersion 20  
    buildToolsVersion "20"  
  
    defaultConfig {  
        minSdkVersion 18  
        targetSdkVersion 20  
        versionCode 1  
        versionName "1.0"  
    }  
}
```

```
dependencies {  
}
```

xq

¿Preguntas?

xq

RED

Async Tasks

Loaders

Servicios

Retrofit

Volley

Imágenes

Patrones básicos



Why bother

- Hace falta conectividad en el 95% de las apps
- Todos los elementos básicos ya existen
- Integrar UI con callbacks es un infierno
- En Android es peor (¡gracias contextos!)
- Controlar hilos manualmente es incluso peor

HTTP vs Otros

- SOAP, CORBA, o cualquier otro popular en su tiempo.
- Demasiado “grandes”
- En ocasiones dependientes de un framework java (no android)
- REST (http en general) es muy fácil

Async Tasks

- For quick and dirty tasks
- Mejores para tareas de duración conocida, generalmente corta
- Lo único que aportan es un modelo de ejecución en segundo plano básico.
- Leaks de contextos (clases internas)

Async Tasks

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {  
    protected Long doInBackground(URL... urls) {  
        HttpPost request = new HttpPost(...);  
        // blah blah  
        return client.execute(request);  
    }  
  
    protected void onProgressUpdate(Integer... progress) {  
        setProgressPercent(progress[0]);  
    }  
  
    protected void onPostExecute(Long result) {  
        hacerAlgoEnLaUI(result);  
    }  
}
```

<http://developer.android.com/reference/android/os/AsyncTask.html>

Loaders

- ~~API creada por Satán~~
- Alternativa a las Async Tasks
- Lo mismo, pero ahora con un 300% más de código
- Sobreviven a cambios de contexto
- Su utilidad real es con bases de datos (cursores)

Servicios

- La forma Android de controlar el segundo plano
- Gestión automática del segundo plano (IntentService)
- La operación es fácil, obtener los resultados no tanto
- Las notificaciones permanentes de Android 4.2+

Suficientes soluciones Android

Retrofit

La forma declarativa

```
public interface GitHubService {  
    @GET("/users/{user}/repos")  
    List<Repo> listRepos(@Path("user") String user);  
}  
  
RestAdapter restAdapter = new RestAdapter.Builder()  
    .setEndpoint("https://api.github.com")  
    .build();  
  
GitHubService service = restAdapter.create(GitHubService.class);  
  
http://square.github.io/retrofit/
```

Retrofit

```
List<Repo> repos = service.listRepos("octocat");
```

- Parsing automático (json, xml)
- Expresión declarativa
- Muy fácil de extender en compilación
- Personalización restringida (en algunos puntos)

Volley

La forma constructiva

```
StringRequest strReq = new StringRequest(Method.GET,
    "www.google.com", new Response.Listener<String>() {

        @Override
        public void onResponse(String response) {
            Log.d(TAG, response.toString());
            pDialog.hide();
        }
    }, new Response.ErrorListener() {

        @Override
        public void onErrorResponse(VolleyError error) {
            VolleyLog.d(TAG, "Error: " + error.getMessage());
            pDialog.hide();
        }
    });

strReq.propiedades(...);
```

<https://android.googlesource.com/platform/frameworks/volley/>

Volley

```
AppController.getInstance().addToRequestQueue(strReq, tag_string_req);
```

- Parsing automático (json, xml)
- Construcción en ejecución
- **Caché automática**
- **Control absoluto** (convención > configuración)

Imágenes

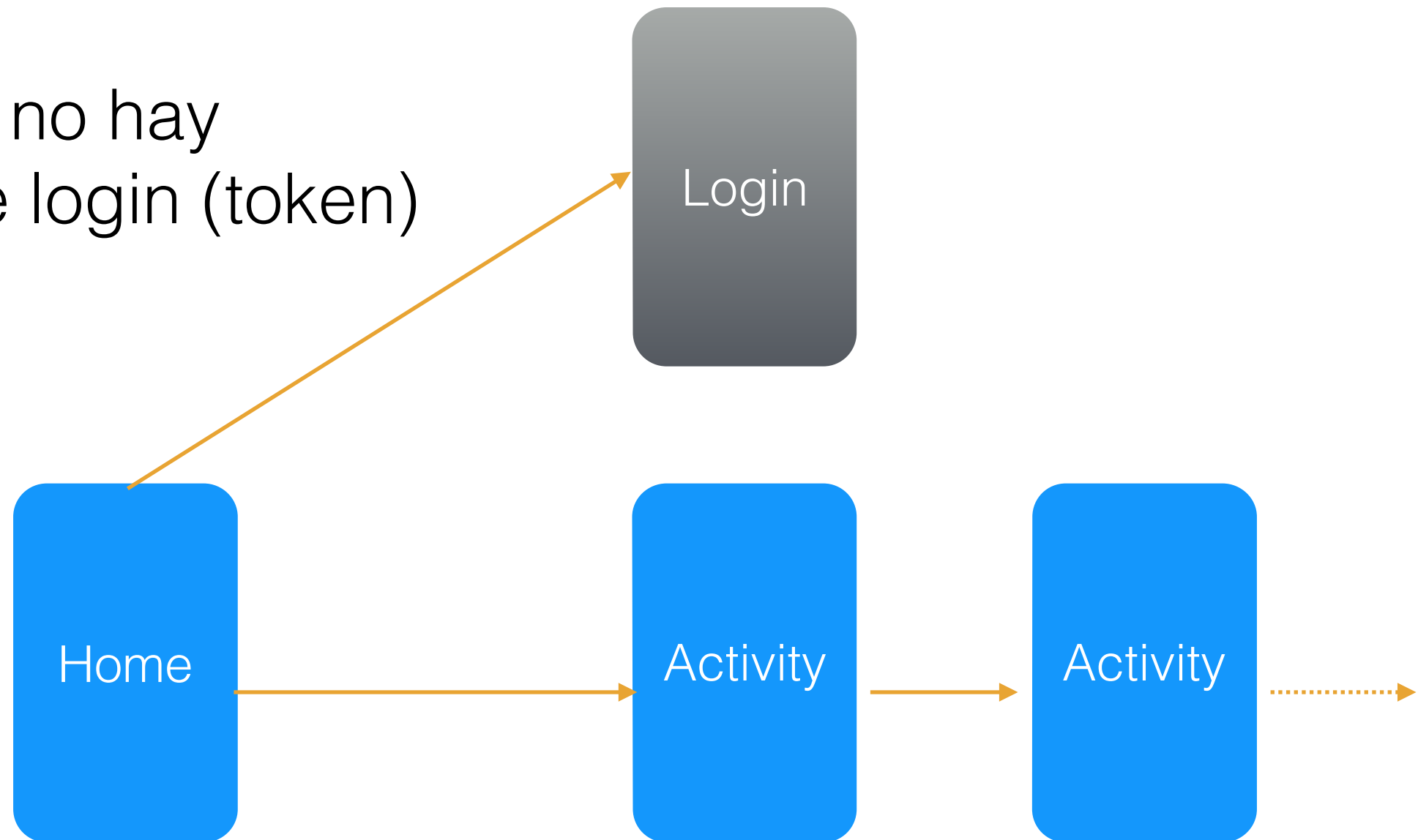
- Forma nativa - 0 beneficios, todos los problemas
- Picasso / Volley: <http://square.github.io/picasso/>
 - Caché
 - Descarga en segundo plano
 - Fallback
 - Desarrolladas para Android
- Universal Image Loader: <https://github.com/nostra13/Android-Universal-Image-Loader>

Controlando errores

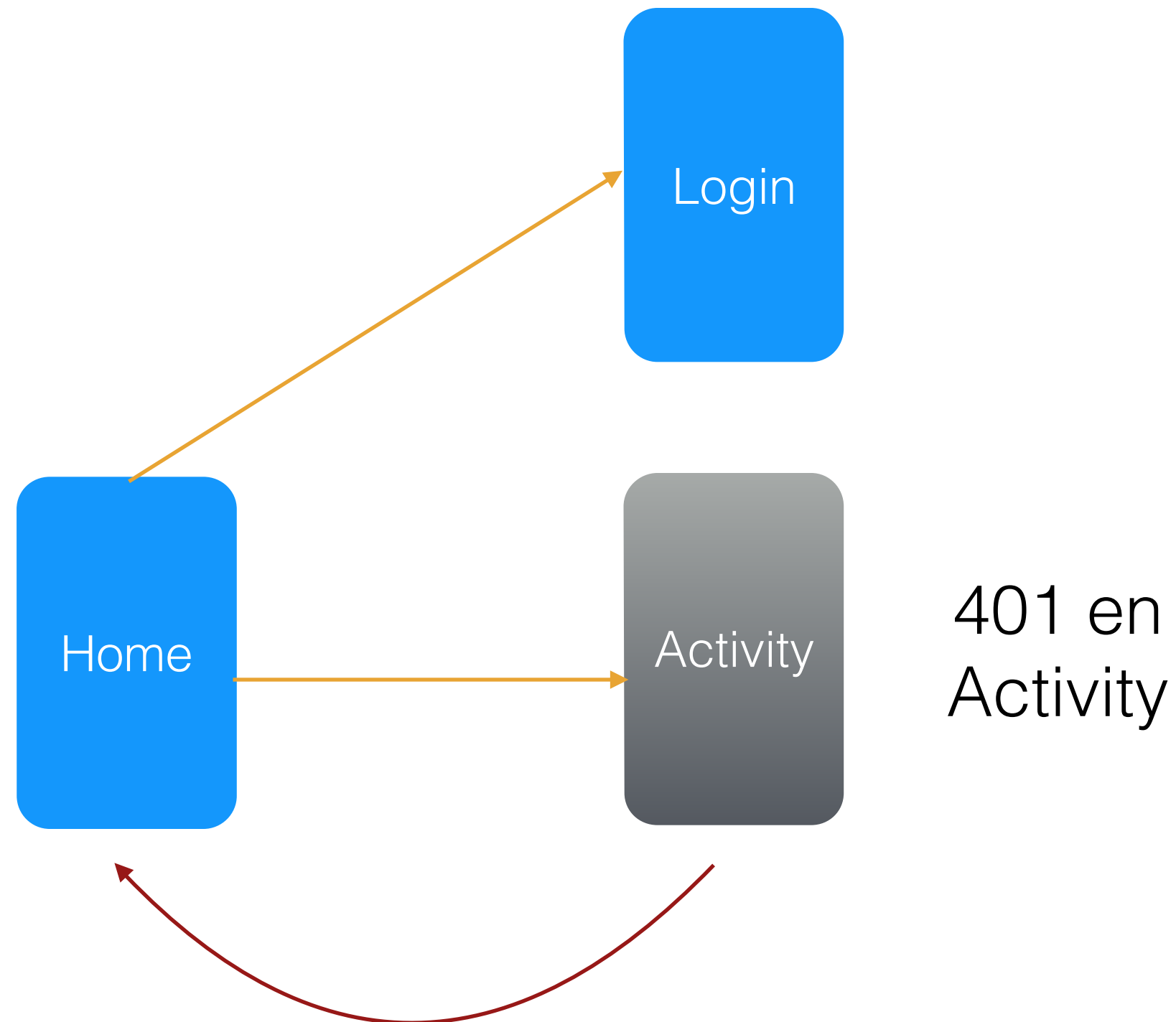
- Hay que asumir que todas las peticiones van a fallar (conectividad móvil)
- Hay que evitar dejar la UI en un estado poco consistente
- Debería haber algo de feedback hacia el usuario
- Casi todos los errores tienen como solución reintentar

401 - Unauthorized

Si no hay
datos de login (token)



401 - Unauthorized



Borrar datos de login + Intent a Home + CLEAR_TOP

xq

¿Preguntas?

xq

Java Puro y Java Android

Inyección de dependencias

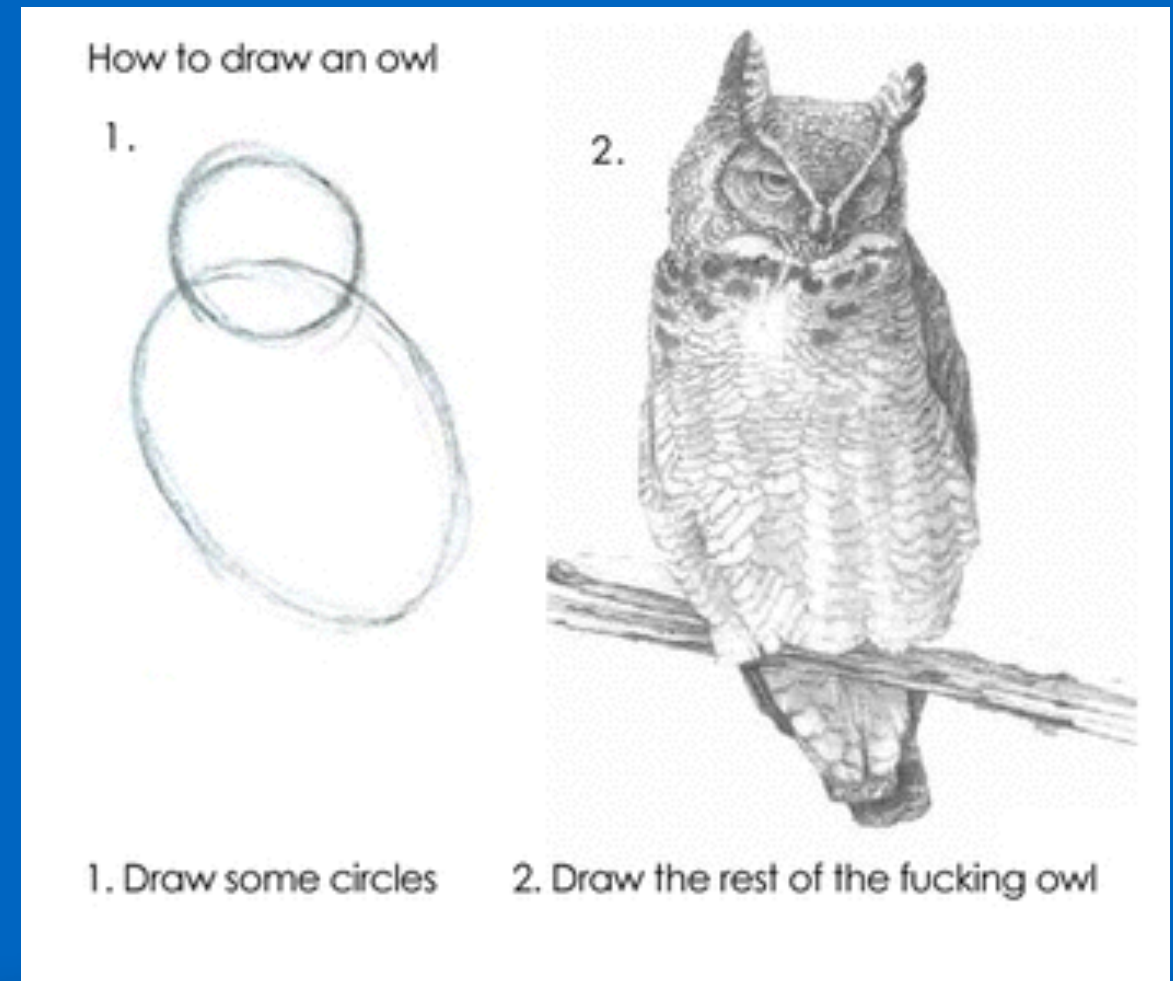
Dagger

Butterknife

Otto

Icepick

Ormlite



Why bother

- El SDK de Android es de muy bajo nivel
- Algunos patrones de Java no aconsejables (singletons)
- Mucho control explícito -> Código duplicado
- Las AbstractFactoryFactory se crean por un motivo
- ...Pero inyectar dependencias está desaconsejado por Android, sobrecarga el runtime

Inyección de dependencias

- Es un patrón (no una librería)
- Nuestros componentes (activity, fragment) satisfacen sus dependencias sin un solo “new”
- Hay muchas formas de abstraer las dependencias
- Métodos factoría, Factories, Prototipos...

Inyección de dependencias

- Es un patrón (no una librería)
- Nuestros componentes (activity, fragment) satisfacen sus dependencias sin un solo “new”
- Hay muchas formas de abstraer las dependencias
- Métodos factoría, Factories, Prototipos...

Dagger en Android

- Múltiples servicios, actividades, etc. necesitan compartir estado
- Algunas librerías necesitan singletons o objetos con vidas largas
- Android es *muy difícil* de probar
- Dagger + Flavors = Happiness

Dagger

- ObjectGraph: Gestor e inyector de dependencias central
- @Module + @Provides: Mecanismo para proveer dependencias
- @Inject: Mecanismo para requerir dependencias
- Algunas cosas más, magia y convenciones
- goo.gl/Sjs0w0

Módulos

- Los módulos son clases que proveen dependencias

```
@Module
public class NetworkModule {
    @Provides @Singleton
    public OkHttpClient provideOkHttpClient() {
        return new OkHttpClient();
    }

    @Provides @Singleton
    public TwitterApi provideTwitterApi(OkHttpClient client) {
        return new TwitterApi(client);
    }
}
```

Inyección de atributos

- La inyección ocurre cuando el objeto está *vivo*
- Los objetos se inyectan a sí mismos (no hay magia)
- También se pueden inyectar constructores

```
public class TweeterActivity extends Activity {  
    @Inject Tweeter tweeter;  
    @Inject Timeline timeline;  
  
    // ...  
}
```

ObjectGraph

```
ObjectGraph og = ObjectGraph.create(  
    new NetworkModule(),  
    new TwitterModule("JakeWharton")  
);  
  
// Using constructor injection:  
TweeterApp app = og.get(TweeterApp.class);  
  
// Using field injection:  
TweeterApp app = new TweeterApp();  
og.inject(app);
```

ButterKnife

- Genera código repetitivo (boilerplate) mediante anotaciones (p. ej: `@InjectView`)
- Sustituye el método `findViewById` por `@InjectView` => menos código y mejor estructurado
- Evita clases anónimas para manejar acciones como el click de un botón (`@OnClick`), click en un elemento de una lista (`@OnItemClick`), etc...

ButterKnife

- Para hacer efectivas las anotaciones, es necesario utilizar el método `ButterKnife.inject`
- En una activity:

```
...  
setContentView(R.layout.activity_main)  
ButterKnife.inject(this);  
...
```

- En un fragment:

```
View onCreateView(LayoutInflater inflater, ViewGroup container,  
Bundle state) {  
    View view = inflater.inflate(R.layout.fancy_fragment,  
container, false);  
    ButterKnife.inject(this, view);  
    return view;  
}
```

ButterKnife

- "Injectar" vistas:

```
// Sin ButterKnife  
  
private ListView listView;  
...  
listView = (ListView) findViewById(R.id.myListView);
```

```
// Con ButterKnife  
  
@InjectView(R.id.myListView) ListView listView;
```

ButterKnife

- Manejar on click de un botón:

```
// Sin ButterKnife
findViewById(R.id.myButton).setOnClickListener(
    new View.OnClickListener() {
        public void onClick(View view) {
            foo();
        }
    });
// Repetir el proceso para el botón R.id.mySecondButton
```

```
// Con ButterKnife
@OnClick({R.id.myFirstButton, R.id.mySecondButton})
public void onButtonClicked(Button button) {
    foo();
}
```


Otto

```
Bus bus = new Bus(); //Singleton Java es la mejor opción  
bus.register(this); //Un fragment, una activity
```

```
bus.post(new MiEvento(42));
```

```
@Subscribe public void miEvento(MiEvento event) {  
    foo();  
}
```

```
http://square.github.io/otto/
```

Otto

- Los métodos `@subscribe` se ejecutan en la UI
- Todos los subscriptores ejecutan sus acciones antes de que `bus.post(...)` retorne
- Se pueden producir eventos de subscripción al bus y hacer bootstrapping (fragmentos dinámicos)

Más

- Icepick: <https://github.com/frankiesardo/icepick>
- Ormlite: <http://ormlite.com/>
- GreenDAO: <http://greendao-orm.com/>
- Todo lo demás: <http://android-arsenal.com/free>


xq

¿Preguntas?

xq

GRACIAS

pabloogc@gmail.com


pilasVacías

