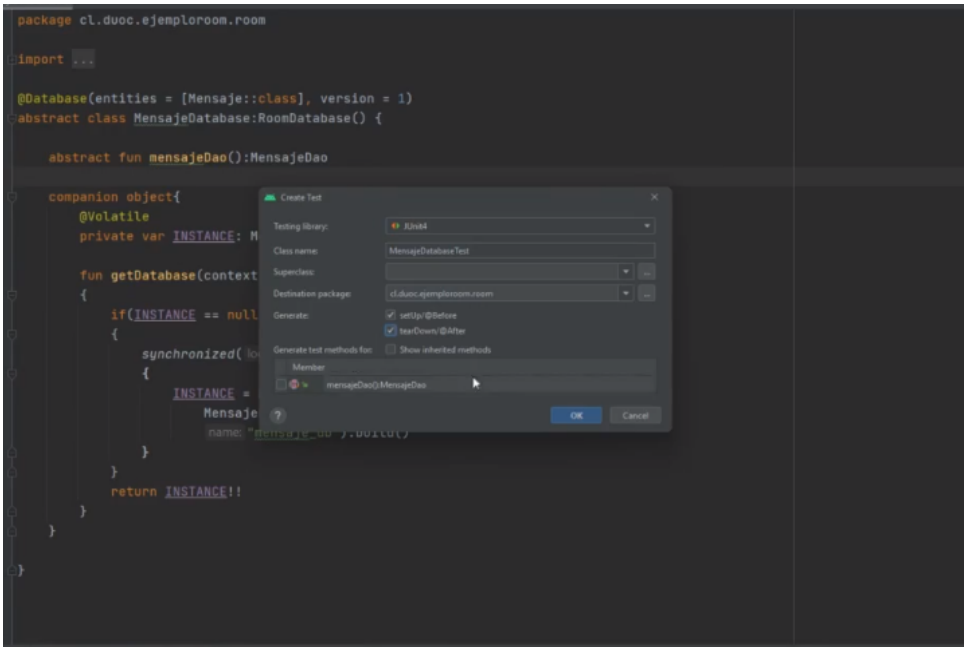
Evidencia Semana 16

Resumen Evidencia día 1 semana 16

El día de hoy se realizan consultas de los ejercicios propuestos y yo aproveché de hacer consultas sobre algunos conceptos que no tenía bien integrados sobre POO y consultar por otros proyectos que descargué para probar la máquina virtual ya que cuando se emula no se cargan las imagenes o se cae el emulador. No avancé mucho prácticando, pero pude despejar dudas de la materia con las que plantearon mis compañeros.

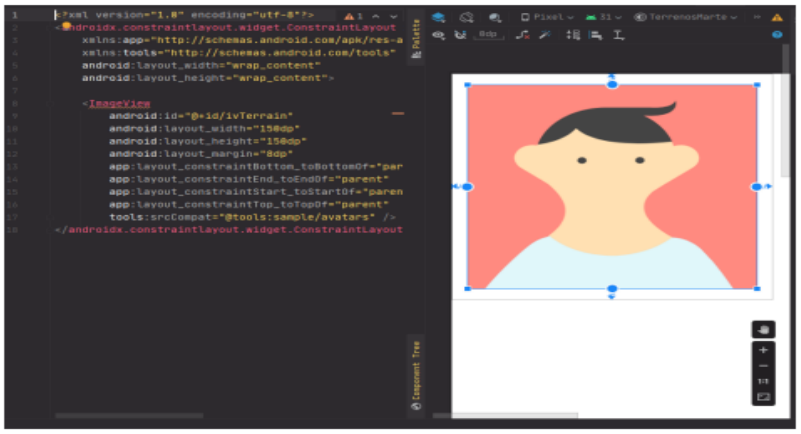
Resumen Evidencia día 2 semana 16

Hoy vimos pruebas instrumentales android test con el ejercicio de room, Mensaje database. Insert<test<Junit4 y activo el @after y @before y nos genera un archivo

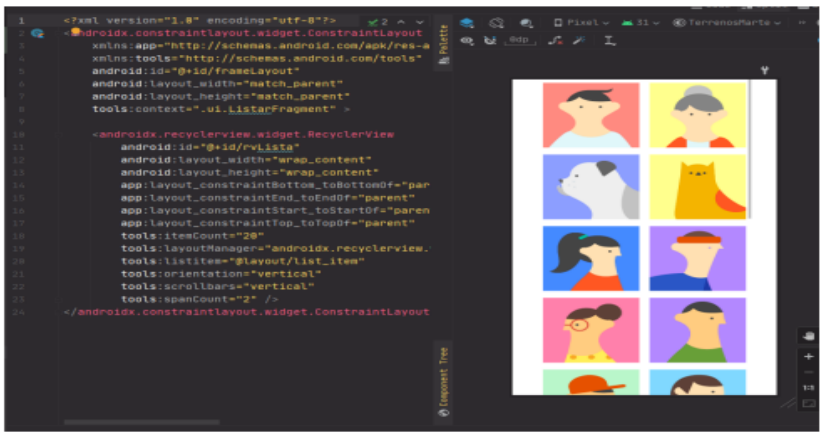


se trabaja en la continuación del ejercicio del día anterior, se resolvieron dudas de algunos compañeros,y se trabaja en el diseño de las vistas y el DataBase.

Vista item_layout



Vista Fragmento Listar



Código DataBase

```
package com.example.terrenosmarte.room
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import com.example.terrenosmarte.dao.TerrainDAO
import com.example.terrenosmarte.model.Terrain

        private const val DATA_BASE_NAME = "mars_db"
        @Database( entities = [Terrain::class], version= 3 )
        public abstract class MarsDB : RoomDatabase()
        {
        abstract fun getTerrainDao() : TerrainDAO
        companion object
        {
        @Volatile
        private var INSTANCE : MarsDB? = null
        fun getDatabase( context : Context) : MarsDB
        {
        val tempInstance = INSTANCE
        if ( tempInstance != null ){
        return tempInstance
        }
        synchronized(this){
        val instance = Room.databaseBuilder( context,
```

```
MarsDB::class.java,DATA_BASE_NAME).fallbackToDestructiveMigration().build()
INSTANCE = instance
return instance
}
}
}
}
```

Código Service

```
package com.example.terrenosmarte.service
import com.example.terrenosmarte.model.Terrain
import retrofit2.Call
import retrofit2.http.GET
    interface TerrainService {
    @GET("realestate")
    fun getTerrains() : Call<ArrayList<Terrain>>
    }
```

Resumen Evidencia día 3 semana 16

El día de hoy consistió en investigar sobre la librería MockServer, y tratar de implementarla en un proyecto, debido a que mi máquina no se encuentra operativa, solo me pude limitar a la búsqueda de información al respecto.

Por qué usar MockServer

MockServer le permite simular cualquier servidor o servicio a través de HTTP o HTTPS, como un servicio REST o RPC.

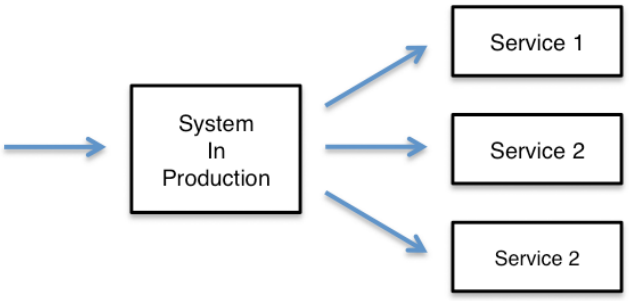
Esto es útil en los siguientes escenarios:

- pruebas
    - Recree fácilmente todo tipo de respuestas para dependencias HTTP, como servicios REST o RPC, para probar aplicaciones de manera fácil y efectiva.
    - aísle el sistema bajo prueba para garantizar que las pruebas se ejecuten de manera confiable y solo fallen cuando haya un error genuino. Es importante que solo se pruebe el sistema bajo prueba y no sus dependencias para evitar que las pruebas fallen debido a cambios externos irrelevantes, como fallas en la red o reinicio / redespliegue de un servidor.
    - Configure fácilmente respuestas simuladas de forma independiente para cada prueba para garantizar que los datos de la prueba estén encapsulados con cada prueba. Evite compartir datos entre pruebas que sean difíciles de administrar y mantener y que corran el riesgo de que las pruebas se infecten entre sí.
    - crear aserciones de prueba que verifiquen las solicitudes que ha enviado el sistema bajo prueba

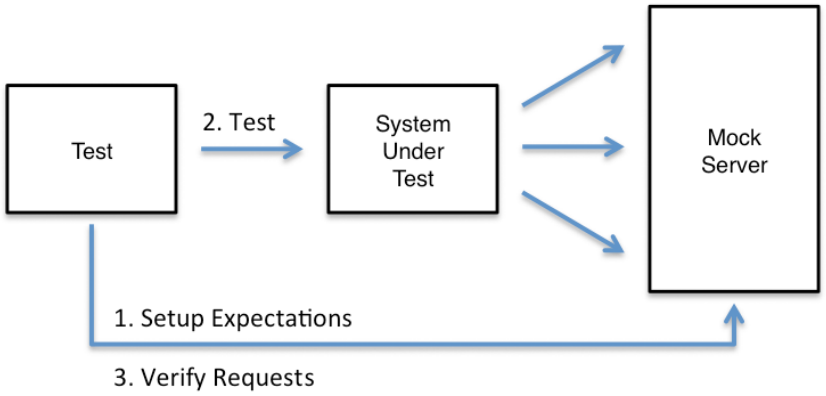
- desarrollo de desacoplamiento
    - comience a trabajar con una API de servicio antes de que el servicio esté disponible. Si una API o un servicio aún no está completamente desarrollado, MockServer puede simular la API, lo que permite que cualquier equipo que esté utilizando el servicio comience a trabajar sin retrasos.
    - aislar a los equipos de desarrollo durante las fases iniciales de desarrollo cuando las API / servicios pueden ser extremadamente inestables y volátiles. El uso de MockServer permite que el trabajo de desarrollo continúe incluso cuando falla un servicio externo
- aislar un solo servicio
    - Durante la implementación y la depuración, es útil ejecutar una sola aplicación o servicio o manejar un subconjunto de solicitudes en una máquina local en modo de depuración. Al usar MockServer, es fácil reenviar selectivamente solicitudes a un proceso local que se ejecuta en modo de depuración, todas las demás solicitudes se pueden reenviar a los servicios reales, por ejemplo, que se ejecutan en un entorno QA o UAT.

Burlarse de las dependencias y verificar la solicitud

Dado un sistema con dependencias de servicio, de la siguiente manera:

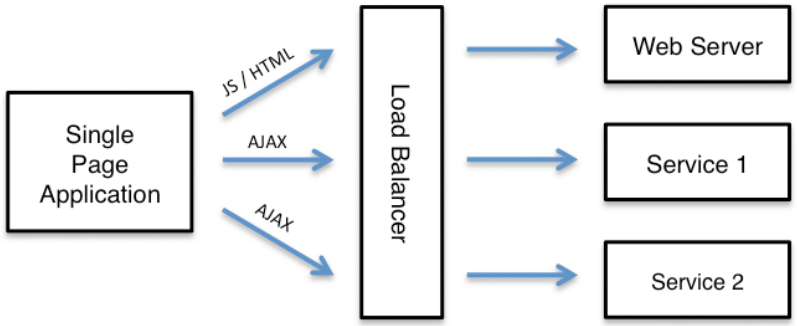


MockServer podría usarse para simular las dependencias del servicio, de la siguiente manera:

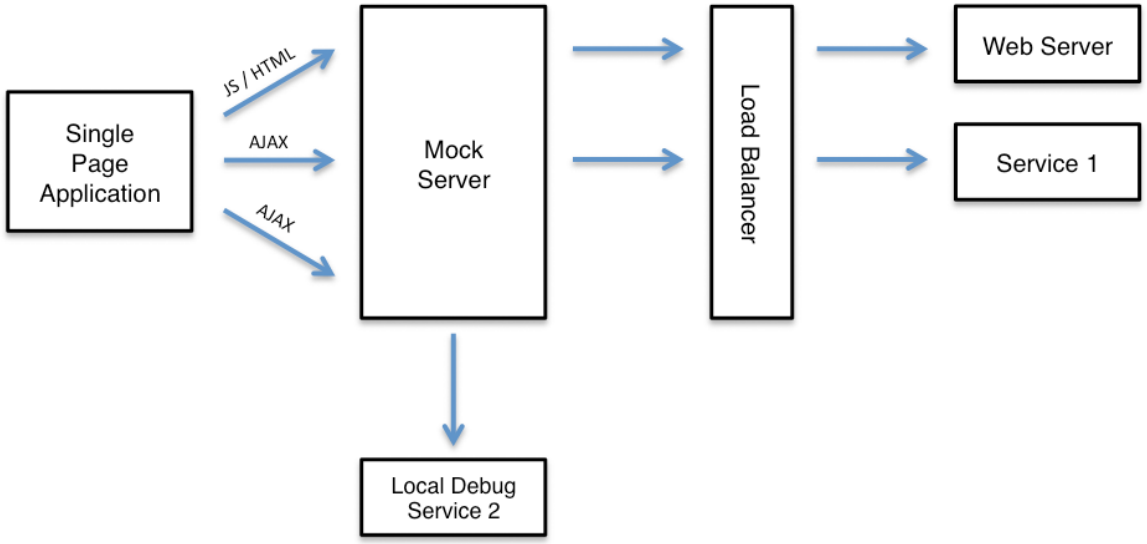


Aislamiento de un solo servicio / aplicación

Una aplicación de una sola página puede cargar recursos estáticos como HTML, CSS y JavaScript desde un servidor web y también realizar llamadas AJAX a uno o más servicios separados, de la siguiente manera:



Para aislar un solo servicio AJAX, para desarrollo o depuración, MockServer puede reenviar selectivamente solicitudes específicas a la instancia local del servicio:



Reflexión: Esta semana los temas se enfocaron en pruebas unitarias e instrumentales y en terminar los proyectos que se han ido desarrollando durante el módulo, siendo bastante honesta me ha costado mucho desarrollar los conceptos que se vieron en clases debido a la dificultad de no tener la máquina virtual operativa o un computador personal que pueda aguantar el software de android studio y poder practicar, me siento en desventaja pero tratando de aprender de las consultas de compañeros y el material adicional que he investigado en internet.