STU FEI
SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

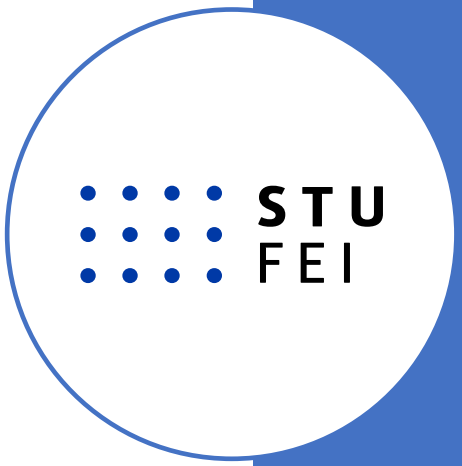
Mobilné výpočty

Ing. Maroš Čavojský, PhD.

SharedPreferences

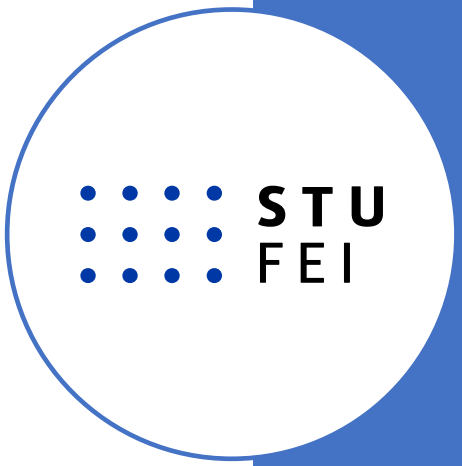
mechanizmus v systéme Android na ukladanie jednoduchých párov kľúč-hodnota a je bežne používaný na ukladanie primitívnych typov dát. Táto možnosť ukladania je vhodná na ukladanie malých množstiev dát, ako sú nastavenia používateľa, konfigurácie aplikácií alebo používateľskej relácie.

K SharedPreferences môžete pristupovať prostredníctvom ľubovoľného kontextu v systéme Android (napríklad Activity, Service alebo BroadcastReceiver). Najčastejšie sa však používa metóda getSharedPreferences() alebo getPreferences().



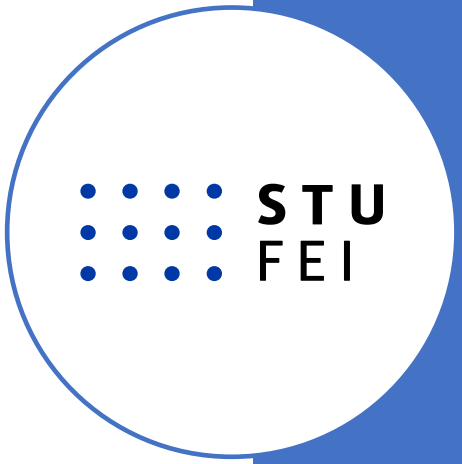
SharedPreferences

```
class PreferenceData private constructor() {

    private fun getSharedPreferences(context: Context?): SharedPreferences? {
        return context?.getSharedPreferences(
            shpKey, Context.MODE_PRIVATE
        )
    }

    companion object {
        @Volatile
        private var INSTANCE: PreferenceData? = null

        private val lock = Any()

        fun getInstance(): PreferenceData =
            INSTANCE ?: synchronized(lock) {
                INSTANCE
                    ?: PreferenceData().also { INSTANCE = it }
            }

        private const val shpKey = "eu.mcomputing.mobv.zadanie"
        private const val userKey = "userKey"

    }

}
```

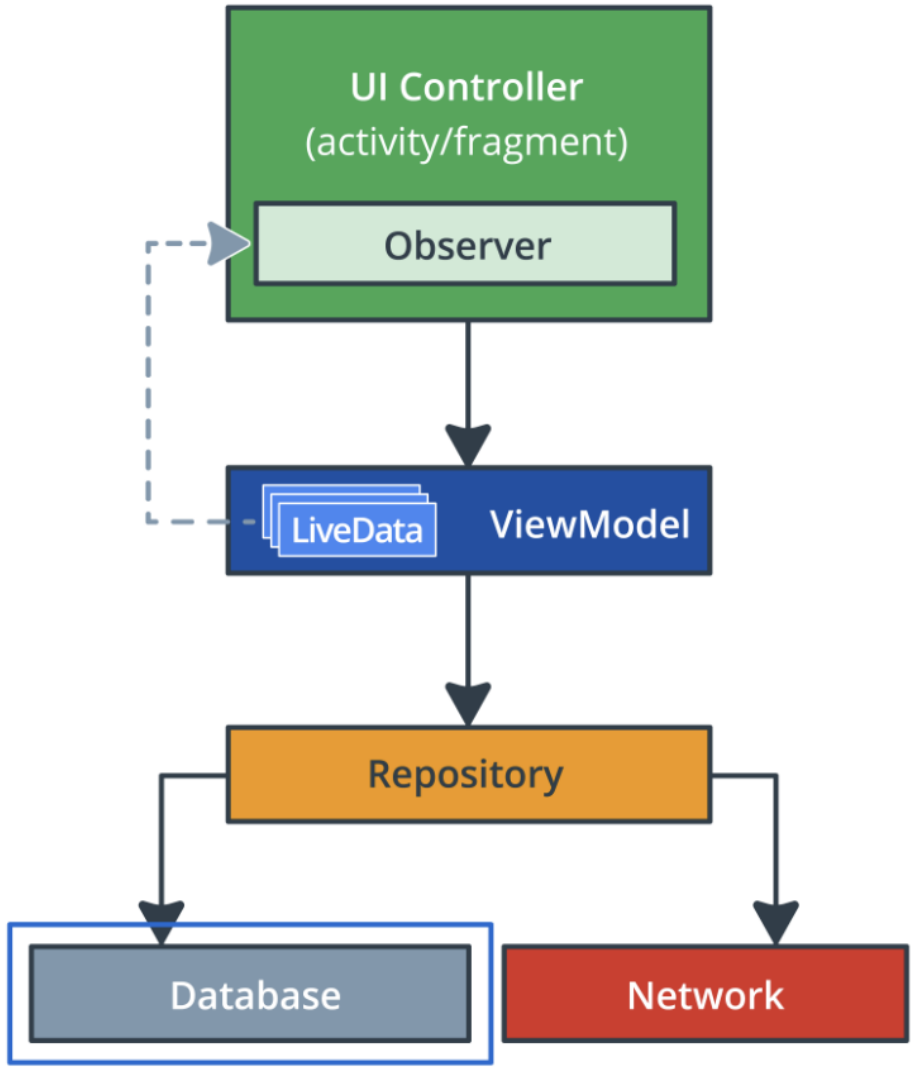


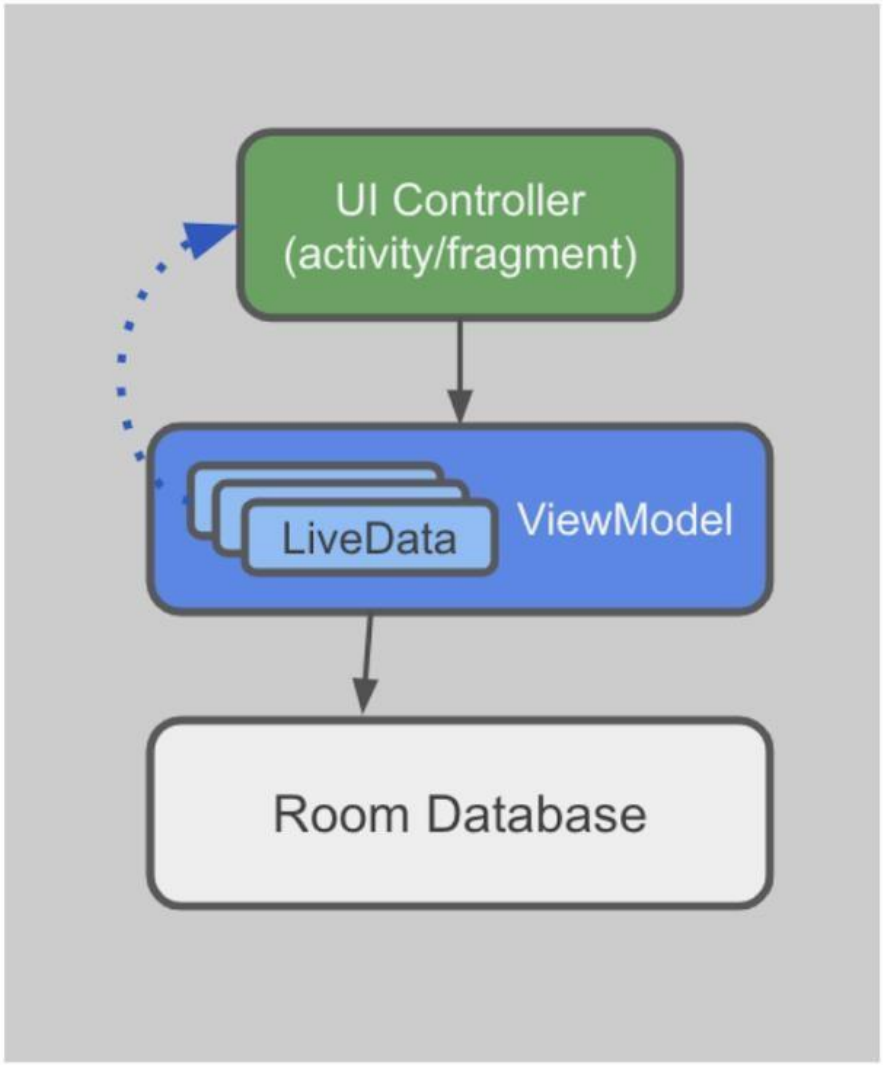
SharedPreferences

```
class PreferenceData private constructor() {
 ...

    fun clearData(context: Context?) {
        val sharedPref = getSharedPreferences(context) ?: return
        val editor = sharedPref.edit()
        editor.clear()
        editor.apply()
    }

    fun putUser(context: Context?, user: User?) {
        val sharedPref = getSharedPreferences(context) ?: return
        val editor = sharedPref.edit()
        user?.toJson()?.let {
            editor.putString(userKey, it)
        } ?: editor.remove(userKey)

        editor.apply()
    }

    fun getUser(context: Context?): User? {
        val sharedPref = getSharedPreferences(context) ?: return null
        val json = sharedPref.getString(userKey, null) ?: return null

        return User.fromJson(json)
    }
...
}
```

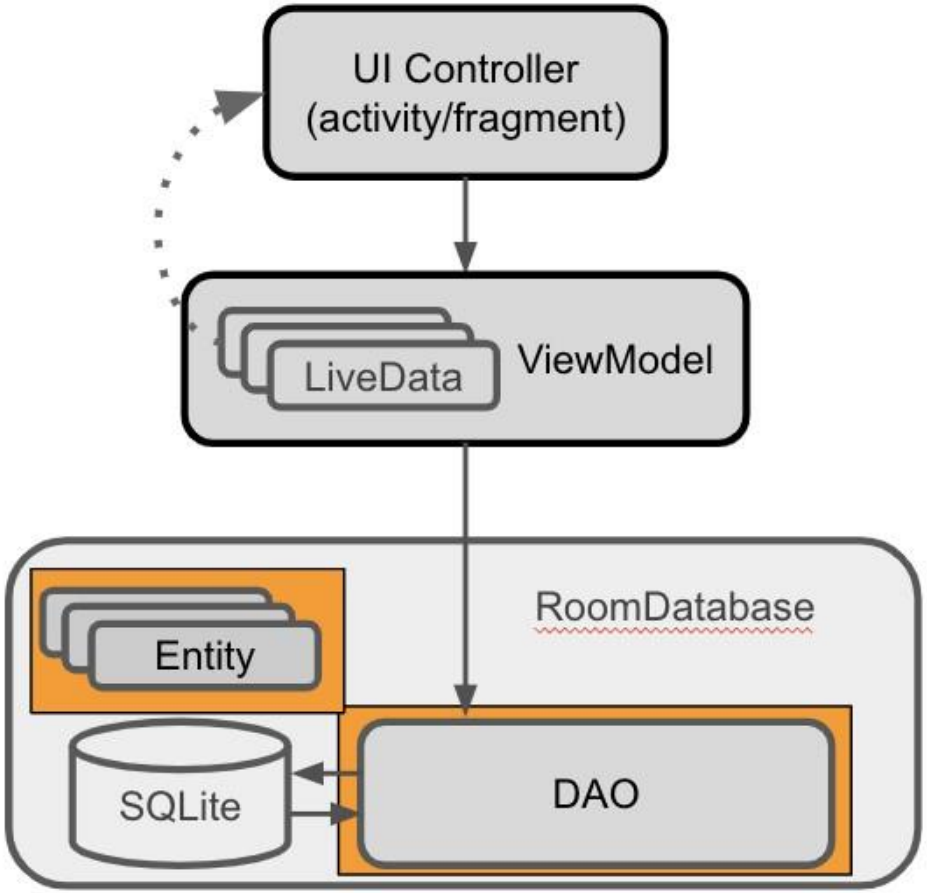


SQLite databáza - Room

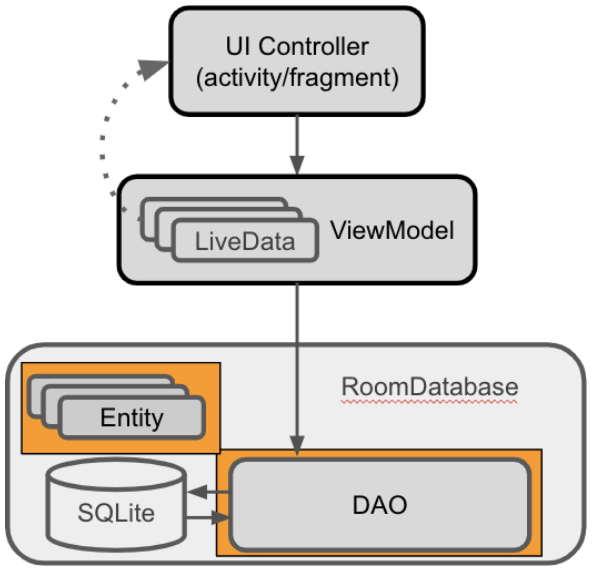


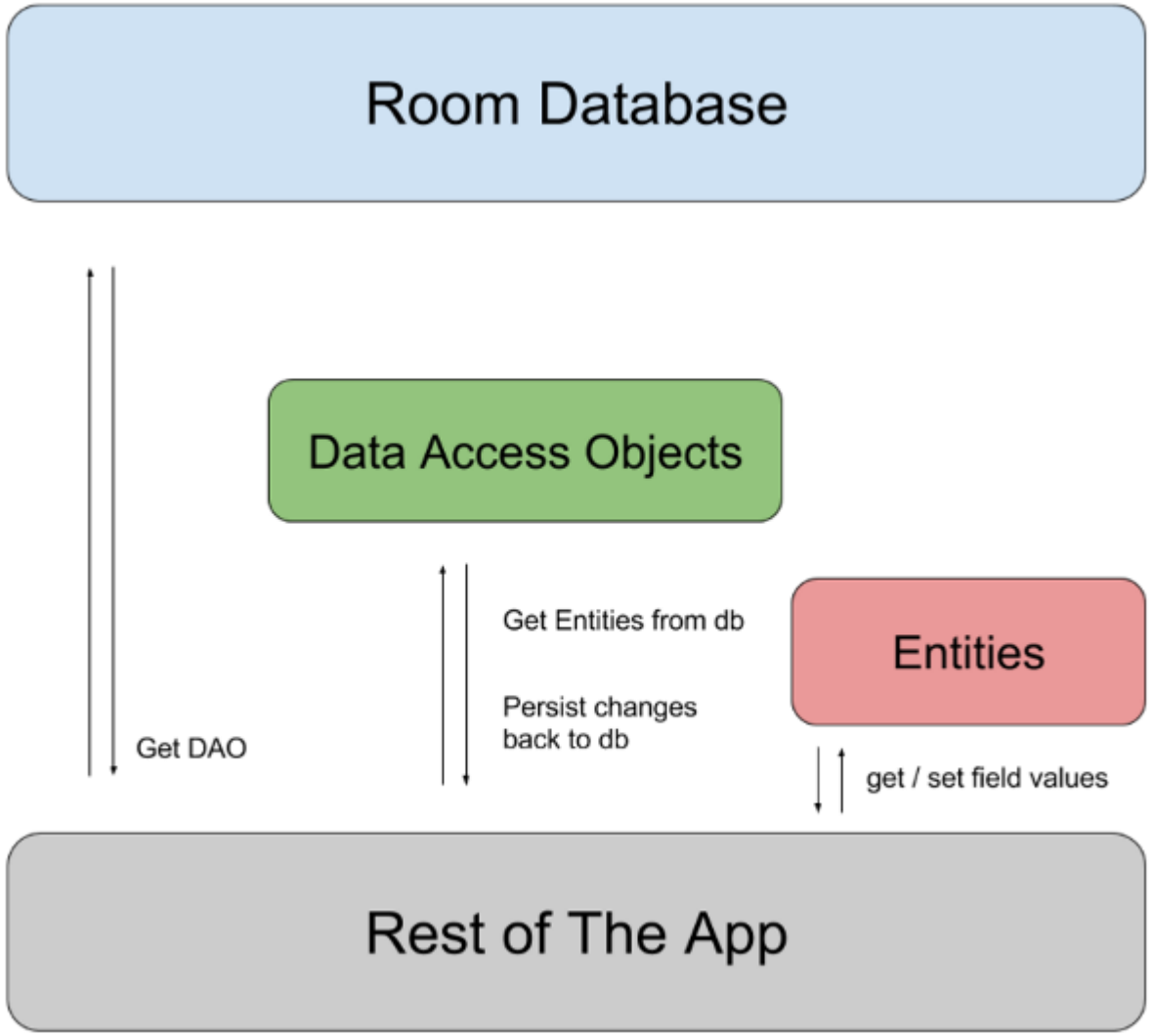


SQLite databáza - Room



SQLite databáza - Room

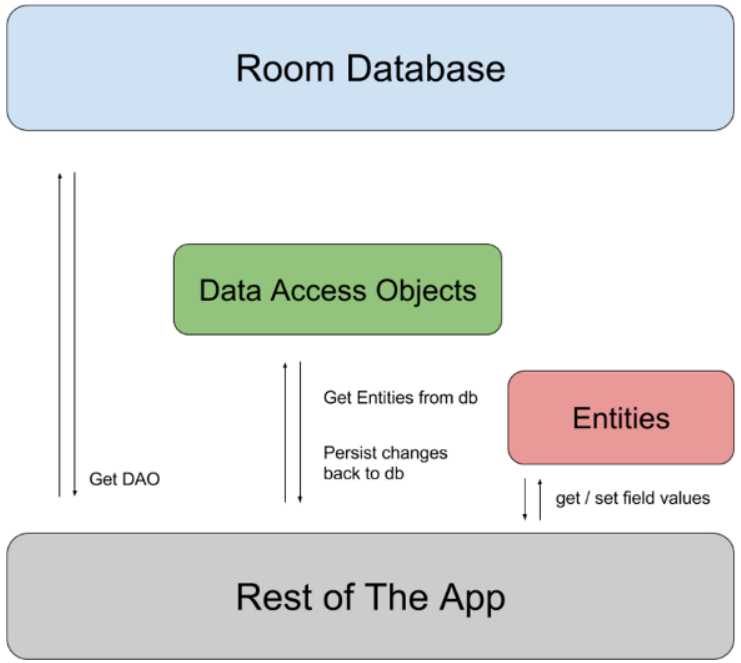




SQLite databáza - Room

```
@Database(entities = arrayOf(User::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```




```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)
}
```

SQLite databáza - Room

```
@Entity(tableName = "users")
data class User (
    @PrimaryKey val id: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

```
@Entity
data class User(
    @PrimaryKey val id: Int,
    val firstName: String?,
    val lastName: String?,
    @Ignore val picture: Bitmap?
)
```

- https://developer.android.com/training/data-storage/room/defining-data

SQLite databáza - Room

```
@Entity(foreignKeys = arrayOf(ForeignKey(
            entity = User::class,
            parentColumns = arrayOf("id"),
            childColumns = arrayOf("user_id"))
        )
)
data class Book(
    @PrimaryKey val bookId: Int,
    val title: String?,
    @ColumnInfo(name = "user_id") val userId: Int
)
```

- https://developer.android.com/training/data-storage/room/relationships

Room - M:N vzťah

```
@Entity
data class Playlist(
    @PrimaryKey var id: Int,
    val name: String?,
    val description: String?
)

@Entity
data class Song(
    @PrimaryKey var id: Int,
    val songName: String?,
    val artistName: String?
)
```

```
@Entity(tableName = "playlist_song_join",
        primaryKeys = arrayOf("playlistId","songId"),
        foreignKeys = arrayOf(
                        ForeignKey(entity = Playlist::class,
                                   parentColumns = arrayOf("id"),
                                   childColumns = arrayOf("playlistId")),
                        ForeignKey(entity = Song::class,
                                   parentColumns = arrayOf("id"),
                                   childColumns = arrayOf("songId"))
                                )
        )
data class PlaylistSongJoin(
    val playlistId: Int,
    val songId: Int
)
```

- https://developer.android.com/training/data-storage/room/relationships

SQLite databáza - Room

```
@Dao
interface MyDao {
    @Query("SELECT * FROM user WHERE age BETWEEN :minAge AND :maxAge")
    fun loadAllUsersBetweenAges(minAge: Int, maxAge: Int): Array<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :search " +
           "OR last_name LIKE :search")
    fun findUserWithName(search: String): List<User>
```

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
fun insertUsers(vararg users: User)
```

```
@Update
fun updateUsers(vararg users: User)
```

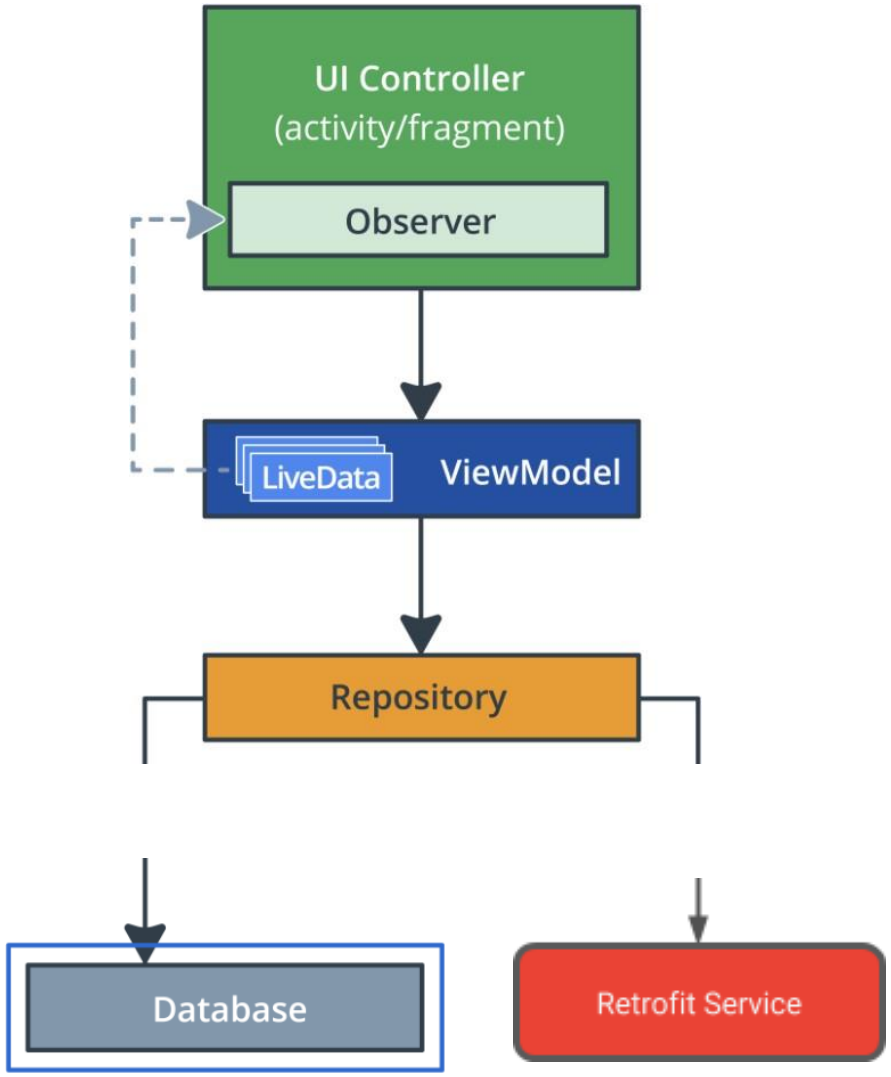
```
@Delete
fun deleteUsers(vararg users: User)
```

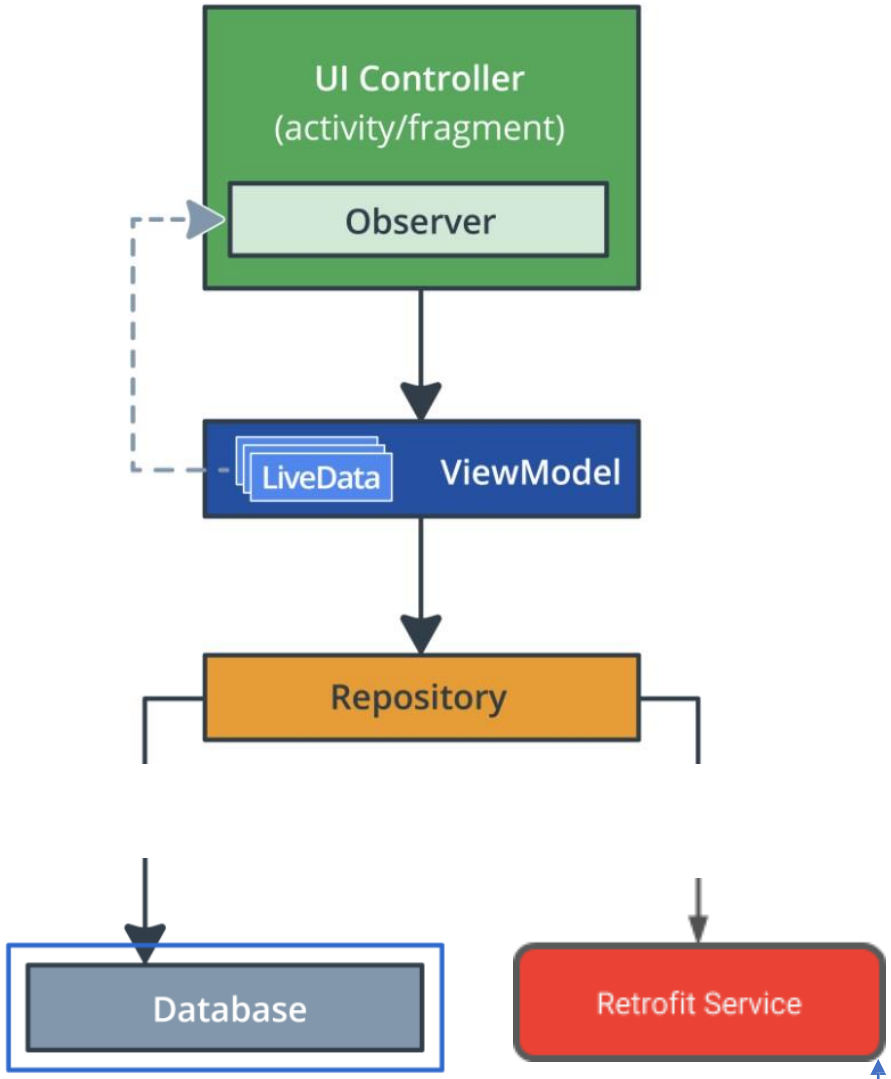
SQLite databáza - Room

```
@Dao
interface MyDao {
    @Query("SELECT * FROM user WHERE age BETWEEN :minAge AND :maxAge")
    fun loadAllUsersBetweenAges(minAge: Int, maxAge: Int): Array<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :search " +
           "OR last_name LIKE :search")
    fun findUserWithName(search: String): List<User>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertUsers(vararg users: User)

    @Update
    fun updateUsers(vararg users: User)

    @Delete
    fun deleteUsers(vararg users: User)
```

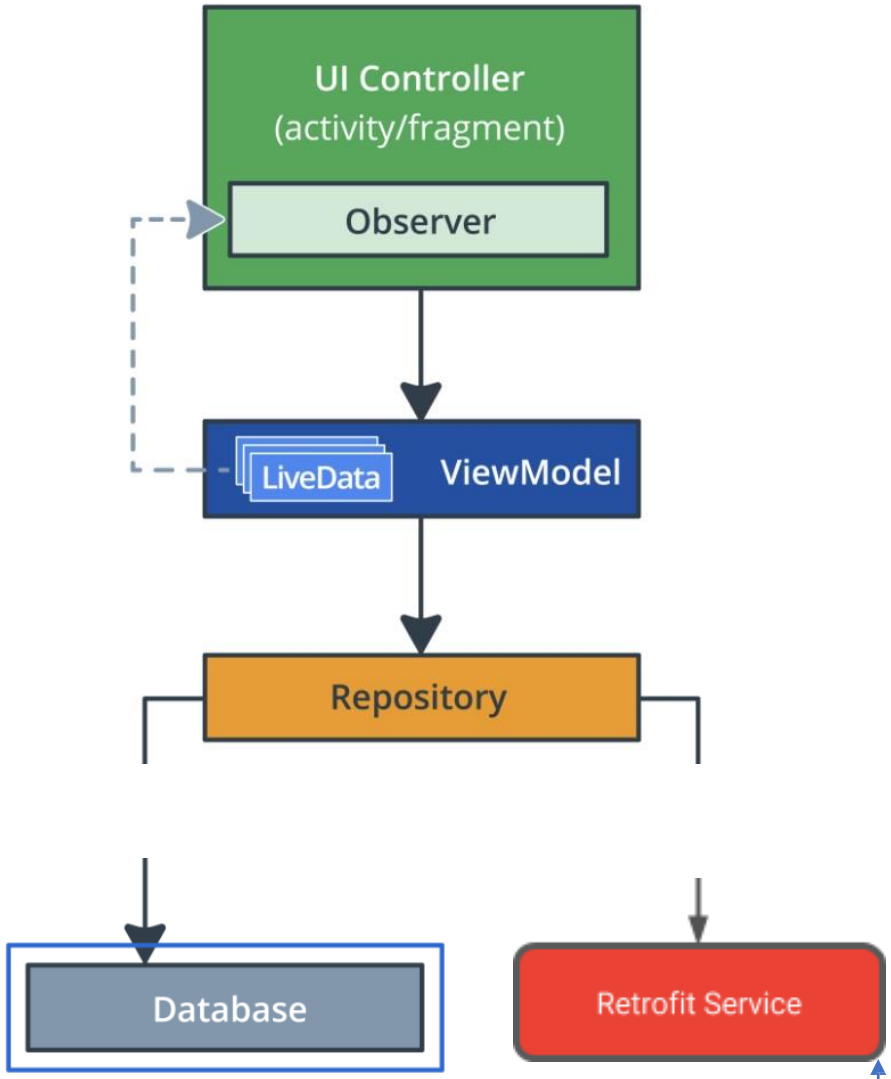
Coroutines + Room

```
@Dao
interface MyDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertUsers(vararg users: User)

    @Update
    suspend fun updateUsers(vararg users: User)

    @Delete
    suspend fun deleteUsers(vararg users: User)

    @Query("SELECT * FROM user")
    suspend fun loadAllUsers(): Array<User>
}
```


UI Controller
(activity/fragment)
Observer
LiveData
ViewModel
Repository
Database
Retrofit Service

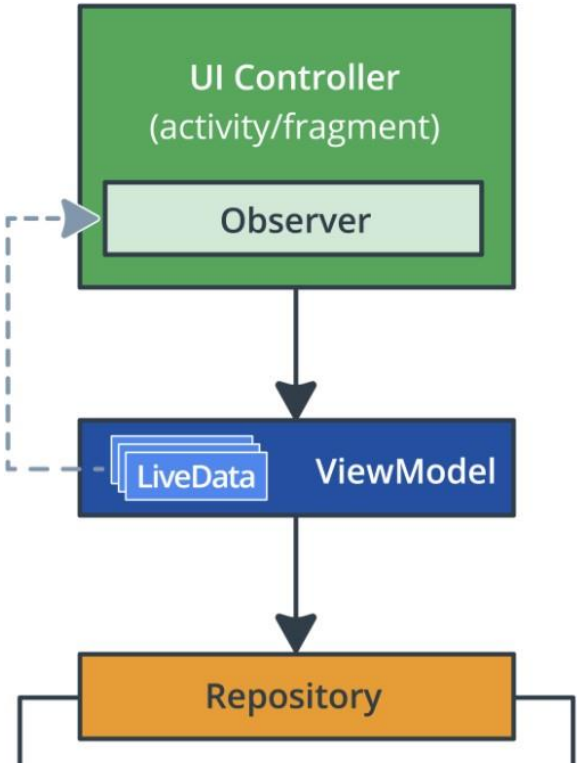



```
@GET("realestate")
    suspend fun getProperties():
Response<List<MarsResponse>>
```




```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insertImages(items:List<MarsItem>)

@Query("SELECT * FROM images")
fun getImages(): LiveData<List<MarsItem>>
```

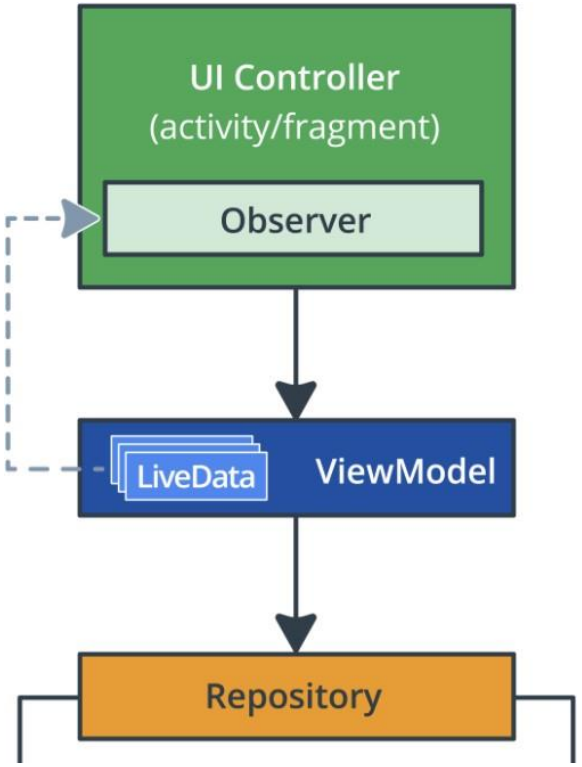
```
@GET("realestate")
    suspend fun getProperties():
Response<List<MarsResponse>>
```




```
suspend fun insertImages(items: List<MarsItem>)
{ dao.insertImages(items)}

fun getImages(): LiveData<List<MarsItem>> = dao.getImages()
```


```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insertImages(items:List<MarsItem>)

@Query("SELECT * FROM images")
fun getImages(): LiveData<List<MarsItem>>
```

```
@GET("realestate")
    suspend fun getProperties():
Response<List<MarsResponse>>
```

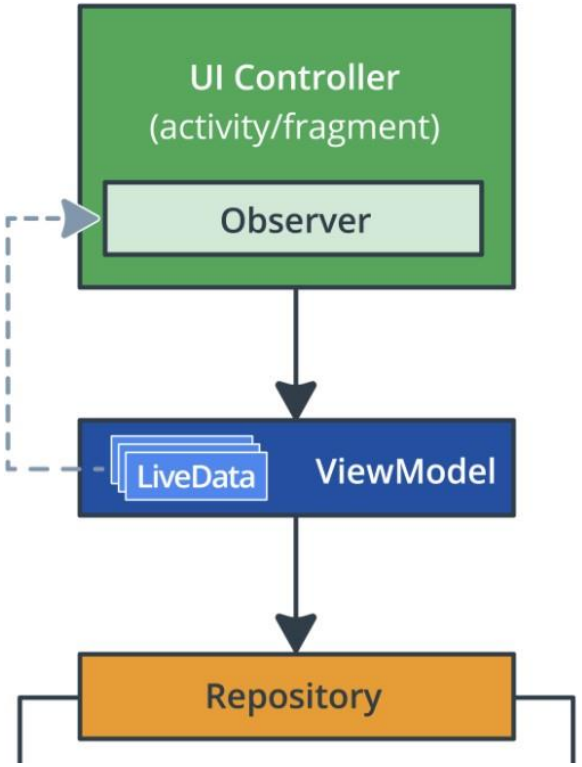



```
suspend fun insertImages(items: List<MarsItem>)
{ dao.insertImages(items)}

fun getImages(): LiveData<List<MarsItem>> = dao.getImages()
```


```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insertImages(items:List<MarsItem>)

@Query("SELECT * FROM images")
fun getImages(): LiveData<List<MarsItem>>
```

```
fun getMars(): LiveData<List<MarsItem>> = cache.getImages()
suspend fun loadMars(
        onError: (error: String) -> Unit)


{...
cache.insertImages(it.map { item ->
     MarsItem(item.price, item.id, item.type, item.img_src)
    })
 ...}
```

```
@GET("realestate")
    suspend fun getProperties():
Response<List<MarsResponse>>
```



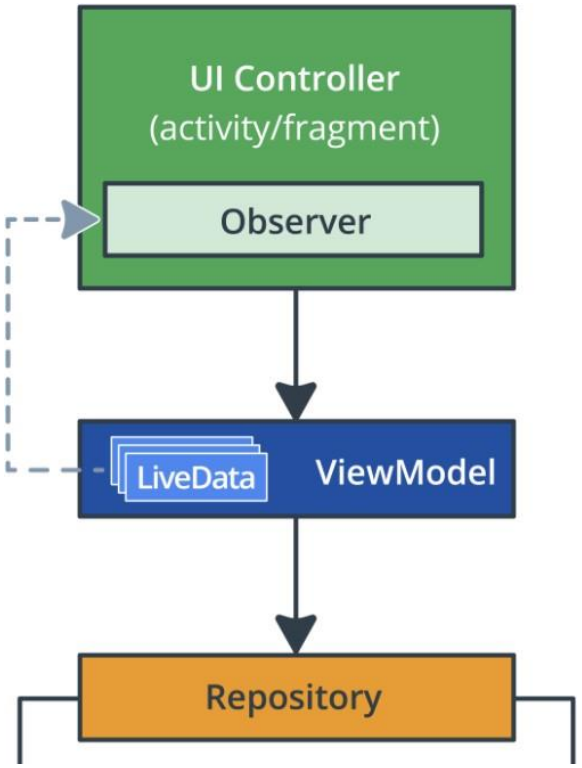

```
val images: LiveData<List<MarsItem>>
    get() = repository.getMars()

fun loadMars() {
    viewModelScope.launch {
        repository.loadMars { error.postValue(it) }
    }
}
```

```
fun getMars(): LiveData<List<MarsItem>> = cache.getImages()
suspend fun loadMars(
        onError: (error: String) -> Unit)

{...
cache.insertImages(it.map { item ->
     MarsItem(item.price, item.id, item.type, item.img_src)
    })
 ...}
```

```
suspend fun insertImages(items: List<MarsItem>)
{ dao.insertImages(items)}

fun getImages(): LiveData<List<MarsItem>> = dao.getImages()
```


```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insertImages(items:List<MarsItem>)

@Query("SELECT * FROM images")
fun getImages(): LiveData<List<MarsItem>>
```

```
@GET("realestate")
    suspend fun getProperties():
Response<List<MarsResponse>>
```

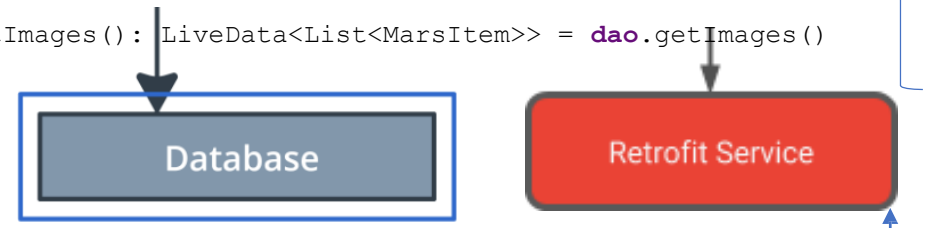



```
marsViewModel.images.observe(this) { adapter.data = it }

marsViewModel.error.observe(this){
          Toast.makeText(context,it,Toast.LENGTH_SHORT).show()
}
```

```
val images: LiveData<List<MarsItem>>
    get() = repository.getMars()

fun loadMars() {
    viewModelScope.launch {
        repository.loadMars { error.postValue(it) }
    }
}
```

```
suspend fun insertImages(items: List<MarsItem>)
{ dao.insertImages(items)}

fun getImages(): LiveData<List<MarsItem>> = dao.getImages()
```

```
fun getMars(): LiveData<List<MarsItem>> = cache.getImages()
suspend fun loadMars(
        onError: (error: String) -> Unit)

{...
cache.insertImages(it.map { item ->
     MarsItem(item.price, item.id, item.type, item.img_src)
    })
 ...}
```




```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insertImages(items:List<MarsItem>)

@Query("SELECT * FROM images")
fun getImages(): LiveData<List<MarsItem>>
```

```
@GET("realestate")
    suspend fun getProperties():
Response<List<MarsResponse>>
```

WorkManager

- Android Jetpack architecture component
- Recommended solution to execute background work (immediate or deferred)
- Opportunistic and guaranteed execution
- Execution can be based on certain conditions

Declare WorkManager dependencies

```
implementation "androidx.work:work-runtime-ktx:$work_version"
```

Important classes to know

- `Worker` - does the work on a background thread, override `doWork()` method
- `WorkRequest` - request to do some work
- `Constraint` - conditions on when the work can run
- `WorkManager` - schedules the `WorkRequest` to be run

Define the work

```
class UploadWorker(appContext: Context, workerParams: WorkerParameters) :
        Worker(appContext, workerParams) {

    override fun doWork(): Result {

        // Do the work here. In this case, upload the images.
        uploadImages()

        // Indicate whether work finished successfully with the Result
        return Result.success()
    }
}
```

Extend CoroutineWorker instead of Worker

```
class UploadWorker(appContext: Context, workerParams: WorkerParameters) :
        CoroutineWorker(appContext, workerParams) {

    override suspend fun doWork(): Result {

        // Do the work here (in this case, upload the images)
        uploadImages()

        // Indicate whether work finished successfully with the Result
        return Result.success()
    }
}
```

WorkRequests

- Can be scheduled to run once or repeatedly
  - `OneTimeWorkRequest`
  - `PeriodicWorkRequest`
- Persisted across device reboots
- Can be chained to run sequentially or in parallel
- Can have constraints under which they will run

Schedule a OneTimeWorkRequest

Create `WorkRequest`:

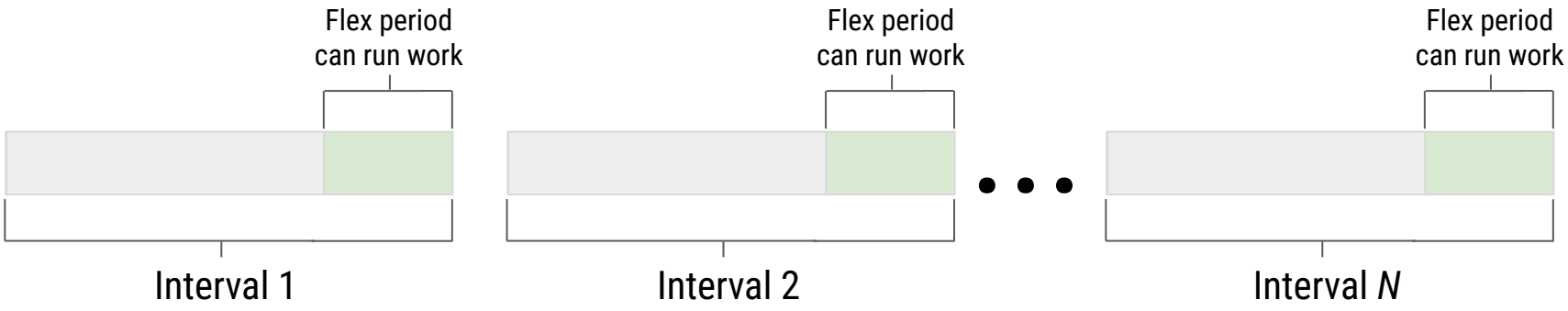
```
val uploadWorkRequest: WorkRequest =
   OneTimeWorkRequestBuilder<UploadWorker>()
       .build()
```

Add the work to the `WorkManager` queue:

```
WorkManager.getInstance(myContext)
    .enqueue(uploadWorkRequest)
```

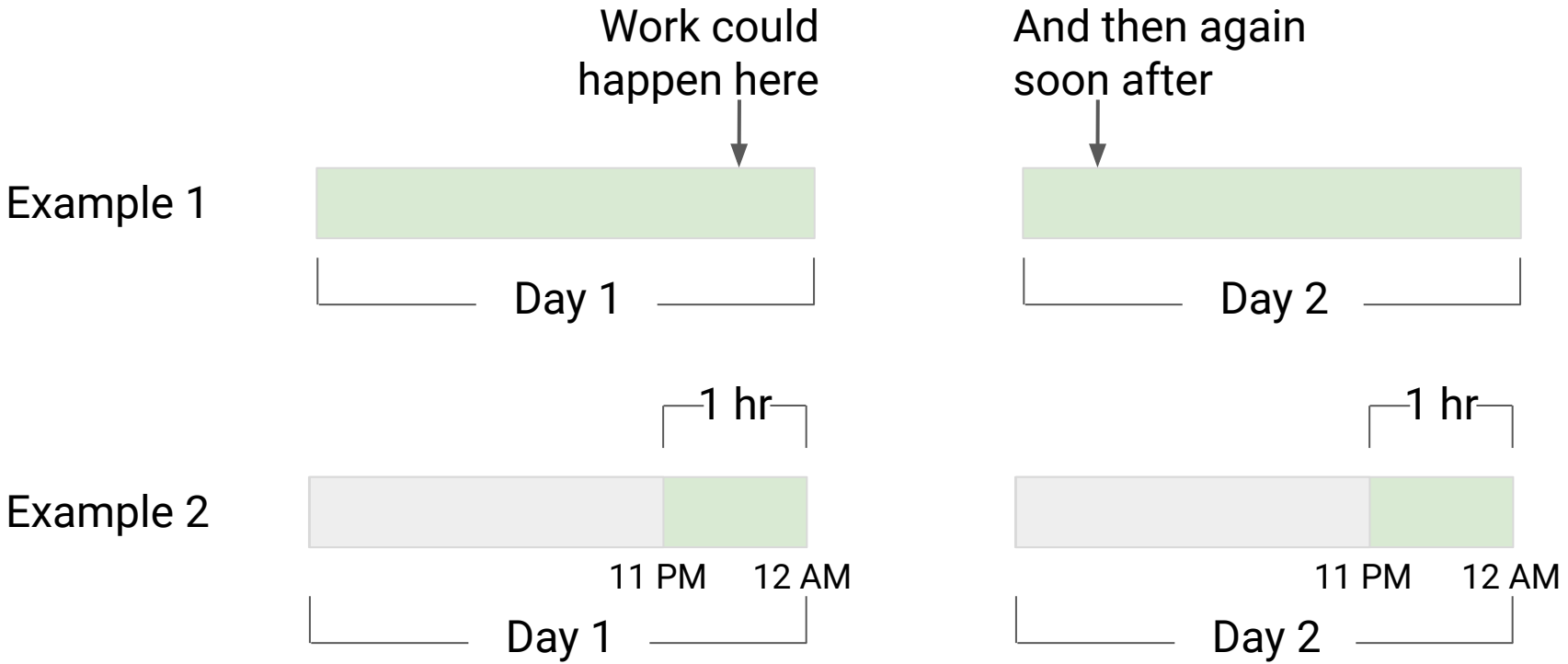
Schedule a PeriodicWorkRequest

- Set a repeat interval
- Set a flex interval (optional)



Specify an interval using `TimeUnit` (e.g., `TimeUnit.HOURS`, `TimeUnit.DAYS`)

Flex interval



PeriodicWorkRequest example

```
val repeatingRequest = PeriodicWorkRequestBuilder<RefreshDataWorker>(
        1, TimeUnit.HOURS,     // repeatInterval
        15, TimeUnit.MINUTES   // flexInterval
    ).build()
```

Enqueue periodic work

```
WorkManager.getInstance().enqueueUniquePeriodicWork(
    "Unique Name",
    ExistingPeriodicWorkPolicy.KEEP, // or REPLACE
    repeatingRequest
)
```

Work input and output

Define Worker with input and output

```
class MathWorker(context: Context, params: WorkerParameters):
      CoroutineWorker(context, params)  {

    override suspend fun doWork(): Result {
        val x = inputData.getInt(KEY_X_ARG, 0)
        val y = inputData.getInt(KEY_Y_ARG, 0)
        val result = computeMathFunction(x, y)
        val output: Data = workDataOf(KEY_RESULT to result)
        return Result.success(output)
    }
}
```

Result output from doWork()

Result status	Result status with output
`Result.success()`	`Result.success(output)`
`Result.failure()`	`Result.failure(output)`
`Result.retry()`	

Send input data to Worker

```
val complexMathWork = OneTimeWorkRequest<MathWorker>()
    .setInputData(
        workDataOf(
            "X_ARG" to 42,
            "Y_ARG" to 421,
        )
    ).build()

WorkManager.getInstance(myContext).enqueue(complexMathWork)
```

WorkRequest constraints

Constraints

- `setRequiredNetworkType`
- `setRequiresBatteryNotLow`
- `setRequiresCharging`
- `setTriggerContentMaxDelay`
- `requiresDeviceIdle`

Constraints example

```
val constraints = Constraints.Builder()
    .setRequiredNetworkType(NetworkType.UNMETERED)
    .setRequiresCharging(true)
    .setRequiresBatteryNotLow(true)
    .setRequiresDeviceIdle(true)
    .build()

val myWorkRequest: WorkRequest = OneTimeWorkRequestBuilder<MyWork>()
    .setConstraints(constraints)
    .build()
```