1. Creating the database

데이터베이스 생성하기

안드로이드 API에서 SQLite를 사용할 때는 API가 제공하는 추상 클래스를 확장해야 했습니다. Room 라이브러리를 사용할 때도 동일한 원칙이 적용됩니다. 우리의 애플리케이션 데이터베이스는 `RoomDatabase` 클래스를 확장하여 선언됩니다. 하지만 Room은 더 강력하여 데이터베이스를 관리하는 추가 코드를 생성하고 더 많은 정보를 필요로 합니다.

우리는 이 데이터베이스에서 관리되는 엔티티와 현재 데이터베이스 버전을 나열하는 `@Database` 어노테이션을 제공해야 합니다. 우리가 설계한 DAO의 인스턴스를 제공하는 함수들은 이 데이터베이스 클래스에 구현됩니다.

애플리케이션 데이터베이스 클래스는 다음과 같이 생겼습니다:

kotlin 코드 복사

```kotlin
@Database(
    entities = [User::class, AuditEntry::class],
    version = 1
)
abstract class ApplicationDatabase: RoomDatabase() {
    abstract fun userDao(): UserDao
    abstract fun auditDao(): AuditDao
}
```

2. Singleton 패턴을 사용하여 데이터베이스 인스턴스 생성

데이터베이스를 생성하는 실제 작업은 Room 객체의 `databaseBuilder` 함수를 사용하여 수행됩니다. 여기에는 안드로이드 컨텍스트, 데이터베이스 클래스의 타입 참조 및 데이터베이스 이름이 필요합니다. 데이터베이스는 비용이 많이 드는 작업이므로, 데이터베이스 인스턴스를 하나만 사용하고 모든 애플리케이션 활동에서 재사용하는 것이 좋습니다. 가장 일반적인 스레드 안전한 싱글톤 패턴 구현은 다음과 같습니다:

kotlin 코드 복사

```kotlin
companion object {
    private const val DATABASE_NAME = "app.db"

    @Volatile private var instance: ApplicationDatabase? = null

    fun getDatabase(context: Context): ApplicationDatabase =
        instance ?: synchronized(this) {
            instance ?: Room.databaseBuilder(
                context.applicationContext,
                ApplicationDatabase::class.java,
                DATABASE_NAME
            ).build().also { instance = it }
        }
}
```

이 패턴은 두 번 확인된 잠금(double-checked locking) 패턴을 사용하여 여러 스레드가 동시에 데이터베이스를 사용하려고 할 때 단일 인스턴스만 생성되도록 합니다.

3. Using database views

데이터베이스 뷰 사용

이전의 단순한 쿼리를 대체하여 조인 쿼리를 사용했던 예제를 데이터베이스 뷰로 바꿀 수 있습니다. 데이터베이스 뷰는 SQL 쿼리를 미리 정의하고 이를 데이터베이스 객체로 사용하는 것입니다. 뷰를 사용하면 길고 복잡한 쿼리를 데이터베이스가 최적화하여 한 번만 처리할 수 있어 효율적입니다.

kotlin 코드 복사

```kotlin
@DatabaseView(
    "SELECT audit.userId, audit.timestamp, user.user_name, user.level " +
    "FROM audit " +
    "INNER JOIN user ON audit.userId = user.id"
)
data class FullAuditEntry(
    val userId: Int,
    val timestamp: Long,
    @ColumnInfo(name = "user_name") val userName: String,
    @ColumnInfo(name = "level") val userLevel: UserLevel
)

@Dao
interface AuditDao {
    @Query("SELECT * FROM fullauditentry")
    fun findAllWithUserDetails(): List<FullAuditEntry>
}
```

5. Converting object references to database types

객체 참조를 데이터베이스 유형으로 변환

객체 참조를 지원하지 않기 때문에 Room은 타입 컨버터를 정의하여 객체를 데이터베이스에서 지원하는 표현으로 저장할 수 있게 합니다. 예를 들어, `Instant` 타입을 `Long` 값으로 변환하여 저장하고 다시 변환할 수 있습니다.

kotlin 코드 복사

```kotlin
class InstantConverters {
    @TypeConverter
    fun fromTimestamp(value: Long?): Instant? {
        return value?.let { Instant.ofEpochMilli(it) }
    }

    @TypeConverter
    fun instantToTimestamp(instant: Instant?): Long? {
        return instant?.toEpochMilli()
    }
}

@Database(
    entities = [User::class, AuditEntry::class],
    views = [FullAuditEntry::class],
    version = 1
)
@TypeConverters(InstantConverters::class)
abstract class ApplicationDatabase: RoomDatabase() {
    abstract fun userDao(): UserDao
    abstract fun auditDao(): AuditDao
}
```

6. Using database migrations

데이터베이스 마이그레이션 사용

애플리케이션 업그레이드 시 기존 데이터베이스가 사용 가능하도록 마이그레이션 절차가 필요합니다. Room은 마이그레이션 핸들러를 등록하여 이 작업을 수행할 수 있습니다. 다음 예제는 `User` 엔티티에 `passwordHash` 필드를 추가하고 데이터베이스 버전을 업데이트하는 방법을 보여줍니다:

kotlin 코드 복사

```kotlin
@Entity(tableName = "user")
data class User (
    // ...
    @ColumnInfo(name = "pwd_hash") val passwordHash: String
)

@Database(
    entities = [User::class],
    version = 2
)
abstract class ApplicationDatabase: RoomDatabase() {
    // ...
    companion object {
        // ...
        internal val MIGRATION_1_2 = object : Migration(1, 2) {
            override fun migrate(database: SupportSQLiteDatabase) {
                database.execSQL(
                    "ALTER TABLE user ADD COLUMN pwd_hash TEXT NOT NULL DE
                )
            }
        }
    }
}
```

```
fun getDatabase(context: Context): ApplicationDatabase =
    instance ?: synchronized(this) {
        instance ?: Room.databaseBuilder(
            context.applicationContext,
            ApplicationDatabase::class.java,
            DATABASE_NAME
        ).addMigrations(MIGRATION_1_2).build().also { instance = it }
    }
```

7. Running queries outside of the main thread

메인 스레드 외부에서 쿼리 실행

Room 라이브러리를 UI 스레드에서 사용하면 앱이 길게 잠길 수 있기 때문에, 모든 데이터베이스 작업을 메인 스레드 외부에서 실행해야 합니다. 이를 위해 코루틴을 사용하여 비동기적으로 작업을 처리할 수 있습니다.

kotlin 코드 복사

```kotlin
class UsersViewModel(application: Application) : AndroidViewModel(applica
    private val database = ApplicationDatabase.getDatabase(getApplication<

    fun findUsers(consumer: (List<User>) -> Unit) {
        viewModelScope.launch {
            val users = withContext(Dispatchers.IO) {
                database.userDao().findAll()
            }
            consumer(users)
        }
    }
}

// Activity에서 ViewModel 사용 예제
val model: UsersViewModel by viewModels()
model.findUsers { users ->
    // 사용자 데이터를 UI에 표시
}
```

8. Testing Room-based databases

Room 기반 데이터베이스 테스트

Room 기반 데이터베이스를 테스트하는 방법 중 하나는 인메모리 데이터베이스를 사용하는 것입니다. 이는 실제 디스크에 쓰지 않고 메모리에서만 작동하므로 테스트가 빠르고 안전합니다.

kotlin 코드 복사

```kotlin
val db = Room.inMemoryDatabaseBuilder(context, ApplicationDatabase::class
val dao = db.userDao()

dao.insert(
    User(1, "regular", UserLevel.NORMAL),
    User(2, "super", UserLevel.SUPERUSER)
)

val users = dao.findAll()
assertThat(users, containsInAnyOrder(
    User(1, "regular", UserLevel.NORMAL),
    User(2, "super", UserLevel.SUPERUSER)
))
```

마이그레이션도 테스트해야 합니다. Room 테스트 라이브러리는 이전 스키마를 다시 생성하고 마이그레이션을 실행하여 데이터베이스가 안정적인지 확인할 수 있게 해줍니다.