Philipp Lackner
DROP TABLE

THE ESSENTIALS OF INDUSTRY-SCALE ANDROID DEVELOPMENT - REVEALED

WHY I NEEDED 13 YEARS TO MASTER THIS

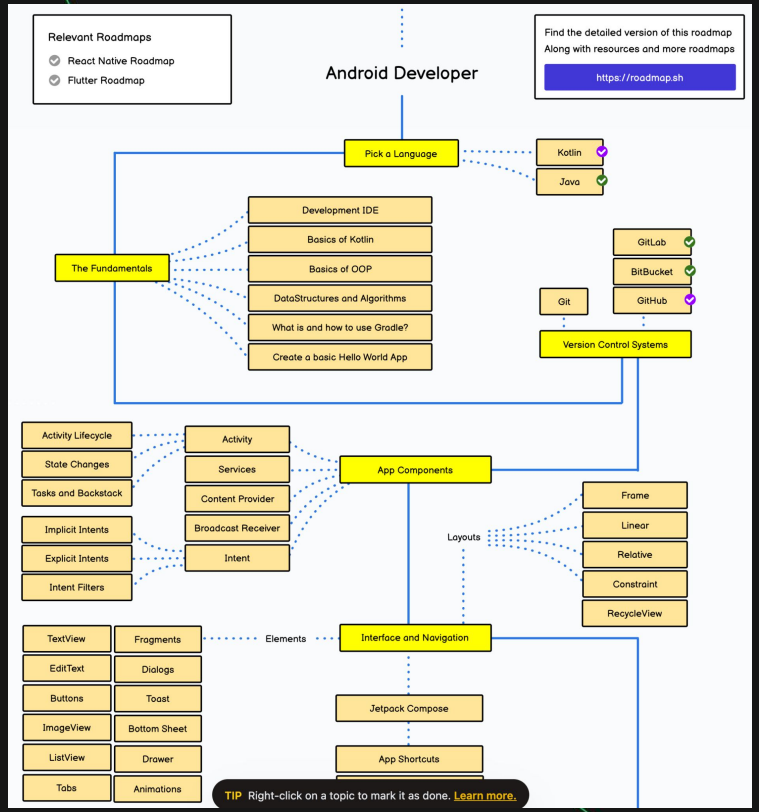
YOU'RE HERE BECAUSE...

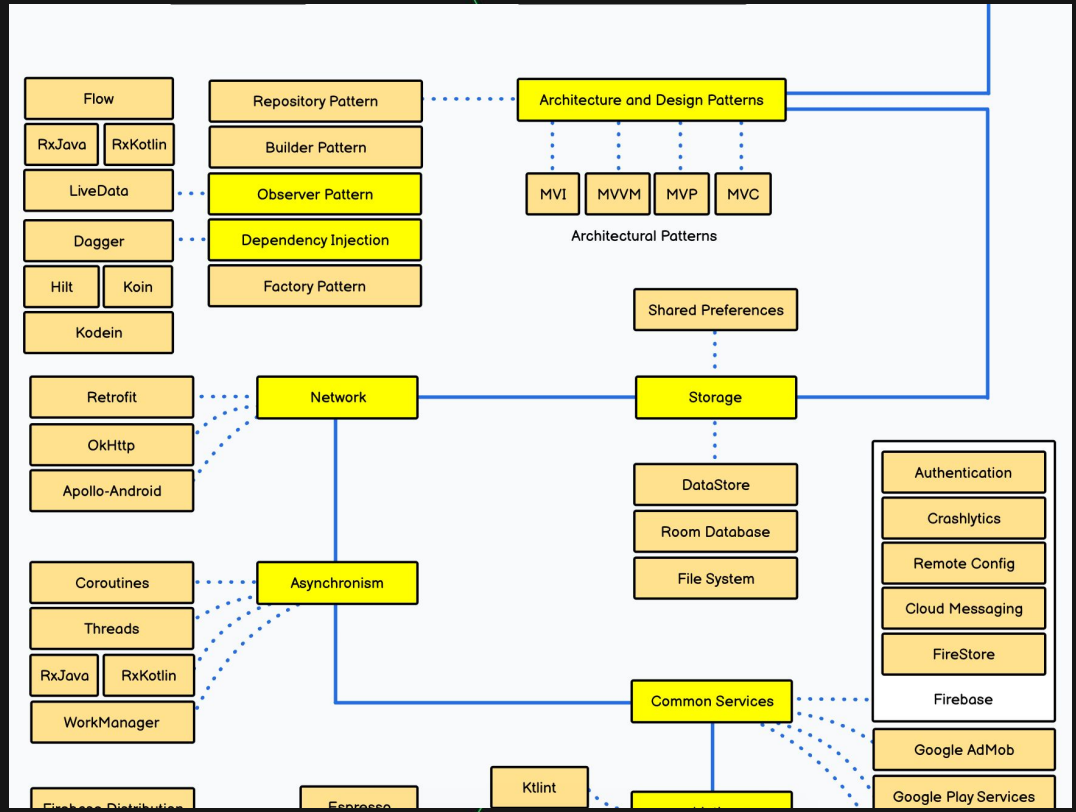
- → ... you learnt some concepts of Android development already, but struggle to glue them together to a working and scaling app
- → ... you are eager to learn more about coding, but feel like your way of learning isn't the most efficient
- → ... you want to learn what ANY software developer needs
- → ... Android dev is your passion, but in practice you don't yet feel very confident about it
- → ... you want to experience the reveal of Project Essentials live

HOW MOST DEVS LEARN ANDROID

Philipp Lackner
DROP TABLE

roadmap.sh





THESE ARE NOT THE ESSENTIALS!

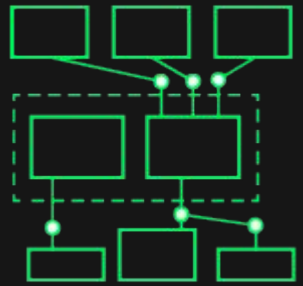
ESSENTIAL DEFINITION

"An essential in software development is a fundamental concept or building block that is critical to both architectural and technical coding practices, applicable across multiple fields. Mastery of these essentials is achieved through their practical application in real-world projects."

→ THE ESSENTIALS ARE NOT ABOUT SPECIFIC LIBRARIES OR SDKs

THERE ARE TWO TYPES OF ESSENTIALS

Philipp Lackner
DROP TABLE



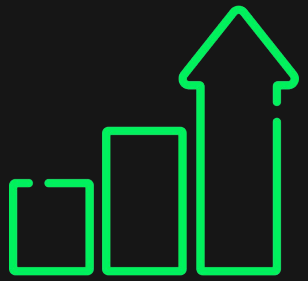
SOFTWARE ENGINEERING ESSENTIALS

TECHNICAL ESSENTIALS

WHY YOU SHOULD FOCUS ON THE ESSENTIALS

Philipp Lackner
DROP TABLE

YOU MASSIVELY SAVE TIME

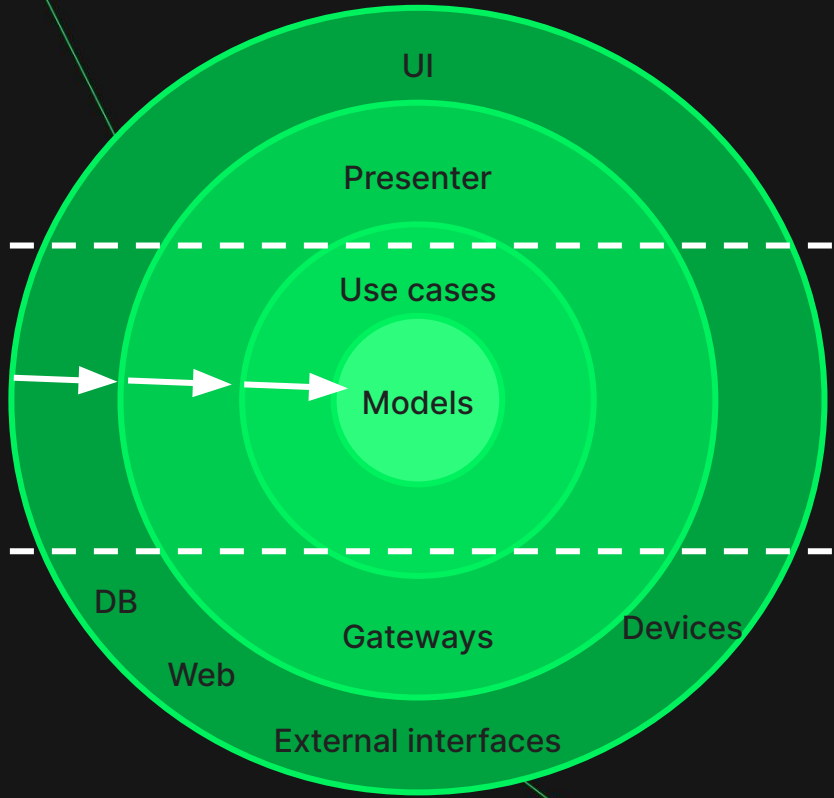


YOU GAIN MAXIMUM FLEXIBILITY

YOU STAND OUT ON THE MARKET

SEPARATION OF CONCERNS

Philipp Lackner
DROP TABLE



PRESENTATION

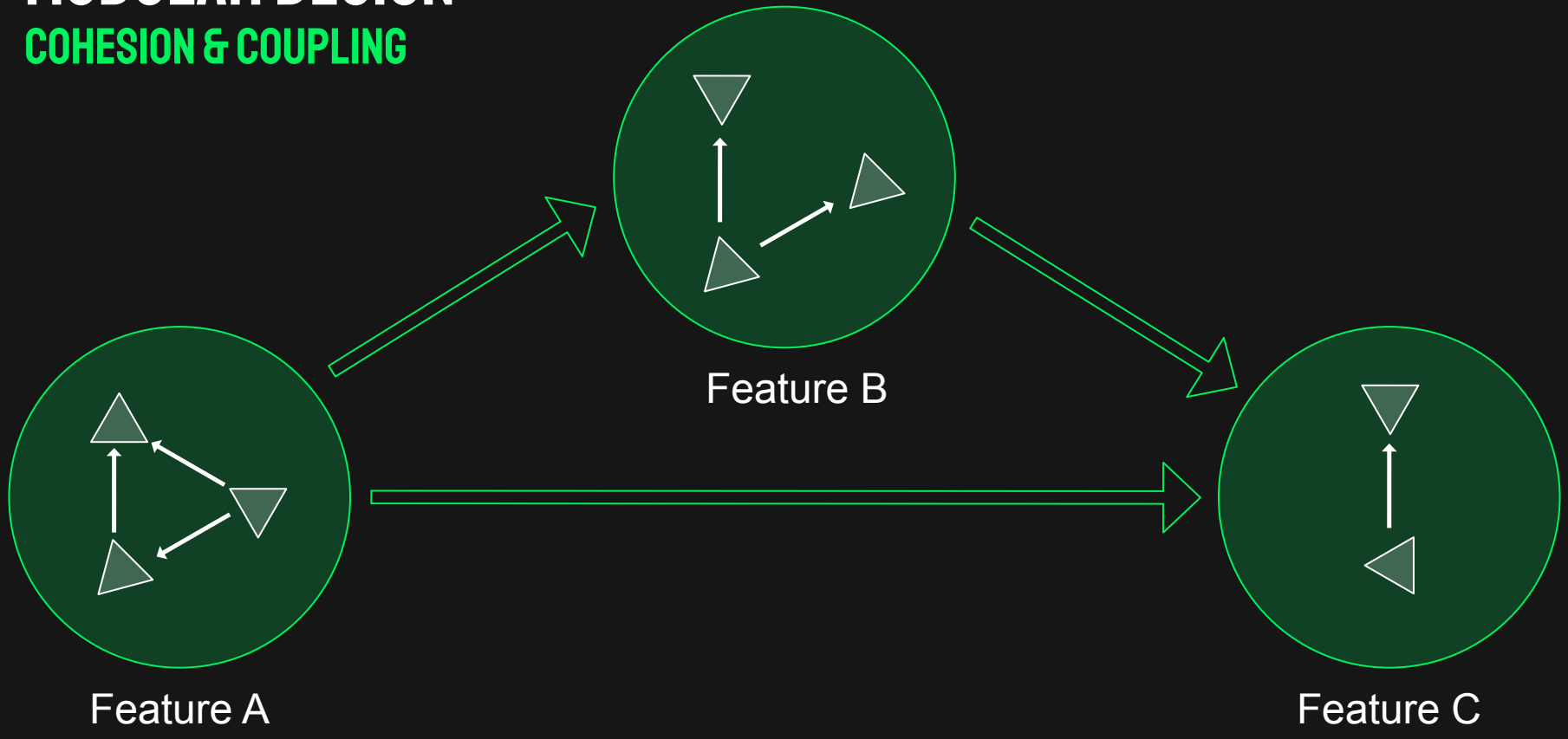
DOMAIN

DATA

MODULAR DESIGN

COHESION & COUPLING

Philipp Lackner
DROP TABLE



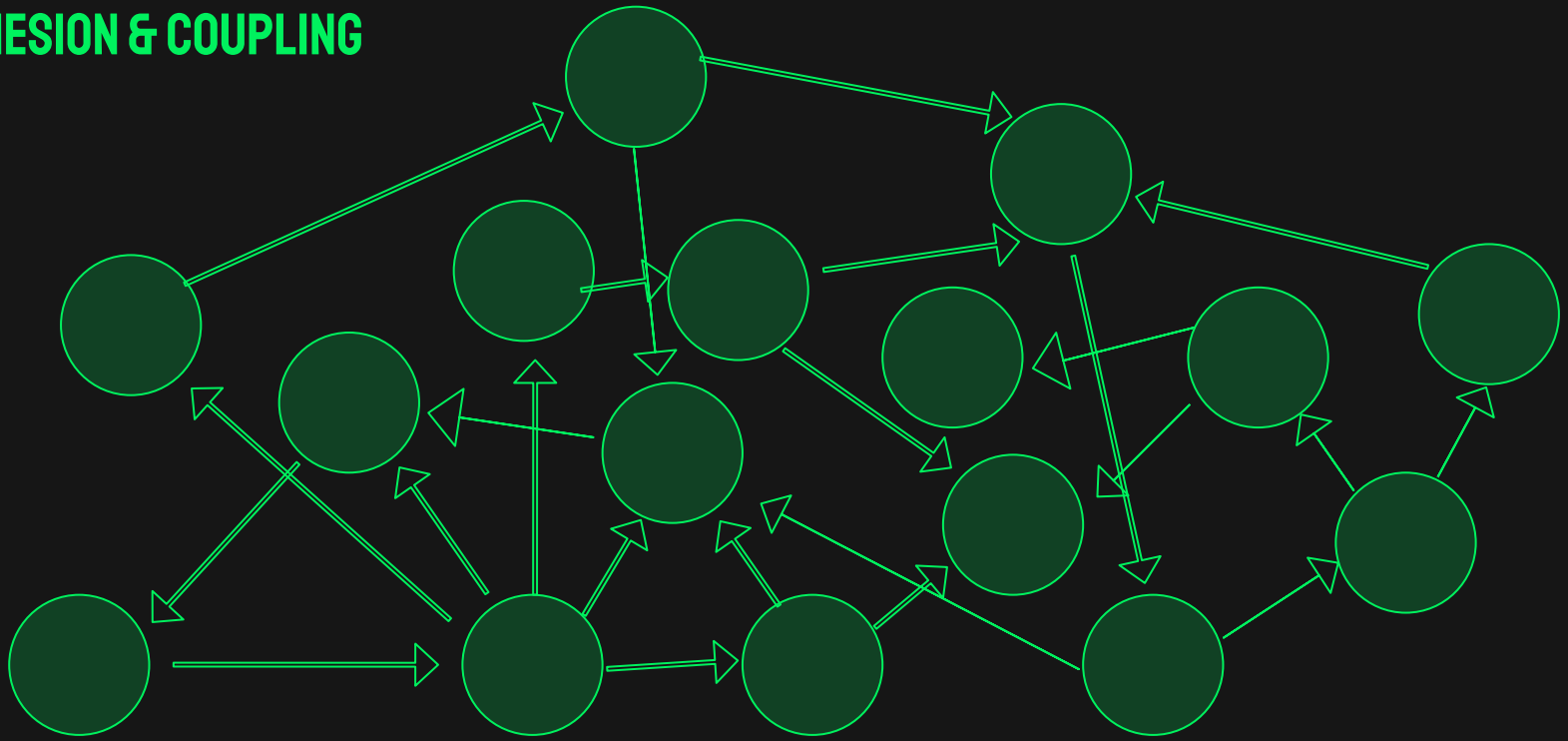
MODULAR DESIGN

COHESION & COUPLING

Philipp Lackner
DROP TABLE

MODULAR DESIGN

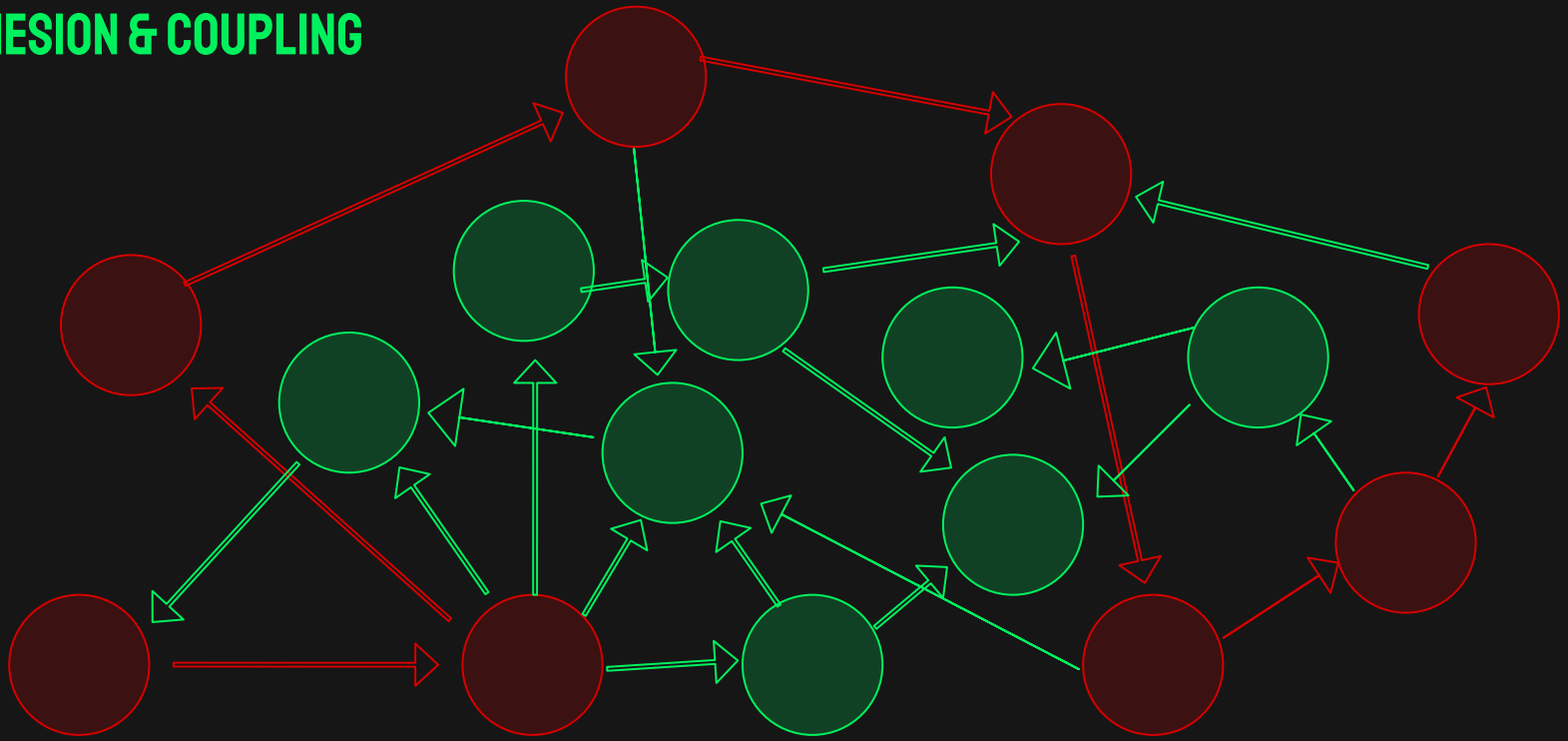
COHESION & COUPLING



Philipp Lackner
DROP TABLE

MODULAR DESIGN

COHESION & COUPLING

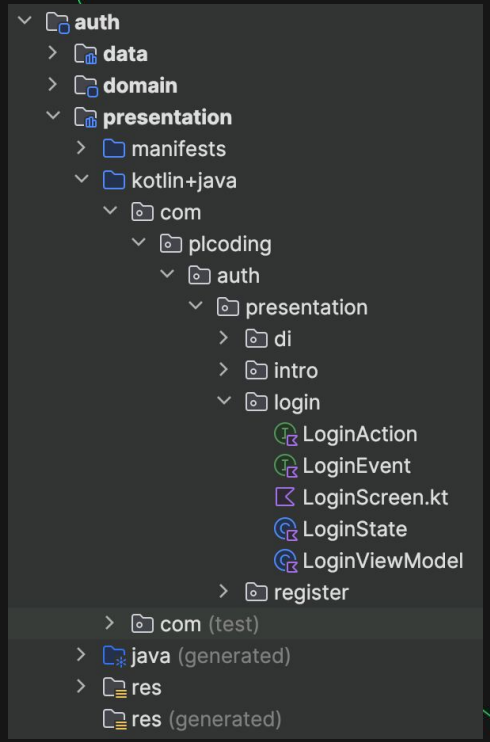


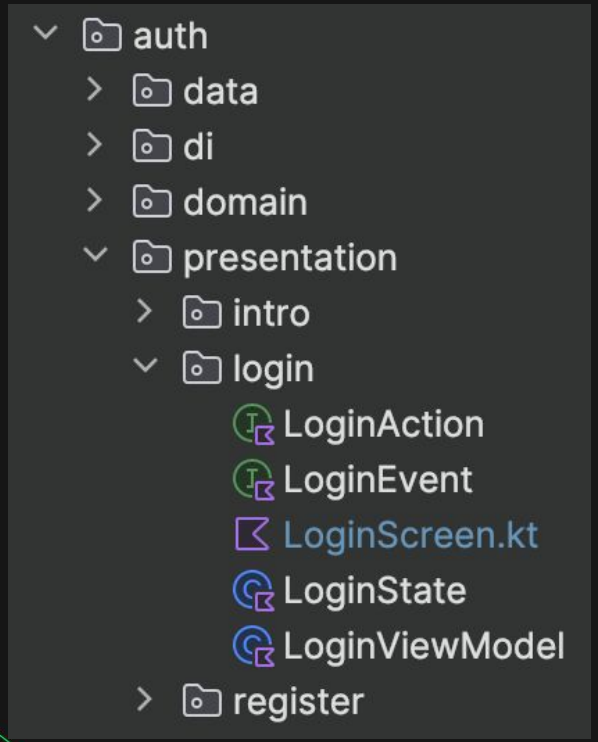
Philipp Lackner
DROP TABLE

WHAT ARE MODULES IN CODE?

COHESION & COUPLING

Philipp Lackner
DROP TABLE





- → Modules are typically “real” Gradle modules or normal packages
- → Gradle modules help you enforce a high cohesion and low coupling
- → Packages rely on the developer not violating these rules

DRY: DON'T REPEAT YOURSELF

Philipp Lackner
DROP TABLE

```
class RegisterViewModel(
    private val authApi: AuthApi
): ViewModel() {

    fun register(email: String, password: String) {
        val isValidEmail =
Patterns.EMAIL_ADDRESS.matcher(email).matches()

        val hasDigit = password.any { it.isDigit() }
        val hasLowercase = password.any { it.isLowerCase() }
        val hasUppercase = password.any { it.isUpperCase() }
        val hasValidLength = password.length >= 9
        val isValidPassword = hasDigit && hasLowercase && hasUppercase &&
hasValidLength

        if(isValidEmail && isValidPassword) {
            viewModelScope.launch {
                val response = authApi.register(email, password)

                if(response.isSuccessful) {
                    // ...
                } else {
                    // ...
                }
            }
        }
    }
}
```

BAD

```
class ChangePasswordViewModel(
    private val authApi: AuthApi
): ViewModel() {

    fun changePassword(password: String) {
        val hasDigit = password.any { it.isDigit() }
        val hasLowercase = password.any { it.isLowerCase() }
        val hasUppercase = password.any { it.isUpperCase() }
        val hasValidLength = password.length >= 9
        val isValidPassword = hasDigit && hasLowercase && hasUppercase &&
hasValidLength

        if(isValidPassword) {
            viewModelScope.launch {
                val response = authApi.changePassword(password)

                if(response.isSuccessful) {
                    // ...
                } else {
                    // ...
                }
            }
        }
    }
}
```

DRY: DON'T REPEAT YOURSELF

Philipp Lackner
DROP TABLE

PROBLEMS OF REPEATED CODE

- Massively increased maintenance effort
- Higher risk of bugs
- Reduced readability

ADVANTAGES OF FOLLOWING DRY

+ Changes at a single place while affecting the whole codebase
+ Minimize risk of bugs, since single piece of code can be tested
+ Less code overall, increasing readability

HOW TO FOLLOW DRY?

Philipp Lackner
DROP TABLE

1. DUPLICATE CODE IN SINGLE CLASS?

→ Create a function

2. DUPLICATE CODE IN DIFFERENT CLASSES?

→ Create a class

3. DUPLICATE CODE IN DIFFERENT MODULES?

→ Create a class in a shared module

KISS: KEEP IT SIMPLE, STUPID

Philipp Lackner
DROP TABLE

```
fun isValidPassword(password: String): Boolean {
    val regex = "^(?=.*[a-z])(?=.*[A-Z])(?=.*[\\d\\W]).{9,}\$\n".toRegex()
    return regex.matches(password)
}
```

KISS: KEEP IT SIMPLE, STUPID

Philipp Lackner
DROP TABLE

```
fun isValidPassword(password: String): Boolean {
    val hasDigitOrSpecialChar = password.any { !it.isLetter() }
    val hasLowercase = password.any { it.isLowerCase() }
    val hasUppercase = password.any { it.isUpperCase() }
    val hasValidLength = password.length >= 9

    return hasDigitOrSpecialChar && hasLowercase && hasUppercase &&
hasValidLength
}
```

KISS: KEEP IT SIMPLE, STUPID

Philipp Lackner
DROP TABLE

```
fun isValidPassword(password: String): Boolean {
    val regex = "^(?=.*[a-z])(?=.*[A-Z])(?=.*[\\d\\W]).
{9,}\$\n".toRegex()
    return regex.matches(password)
}
```

BAD

```
fun isValidPassword(password: String): Boolean {
    val hasDigitOrSpecialChar = password.any { !it.isLetter() }
    val hasLowercase = password.any { it.isLowerCase() }
    val hasUppercase = password.any { it.isUpperCase() }
    val hasValidLength = password.length >= 9

    return hasDigitOrSpecialChar && hasLowercase && hasUppercase &&
hasValidLength
}
```

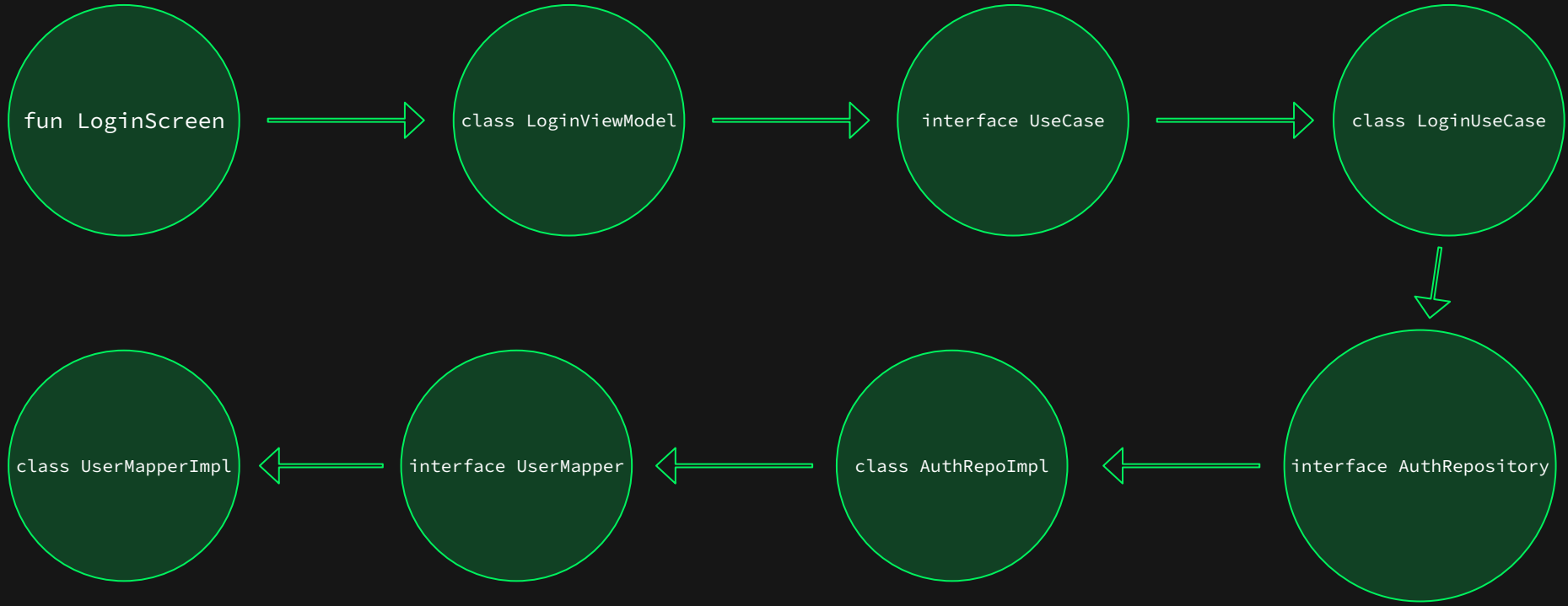
RIGHT

- → Code should not be more complex than it needs to be
- → Readability usually wins over minimal performance gain

KISS: KEEP IT SIMPLE, STUPID

ANOTHER BAD EXAMPLE

Philipp Lackner
DROP TABLE



KISS: KEEP IT SIMPLE, STUPID

GOOD EXAMPLE

Philipp Lackner
DROP TABLE

SOLID PRINCIPLES

S INGLE-RESPONSIBILITY

O PEN-CLOSED

L ISKOV-SUBSTITUTION

I NTERFACE-SEGREGATION

D EPENDENCY-INVERSION

Philipp Lackner
DROP TABLE

SINGLE-RESPONSIBILITY PRINCIPLE

Philipp Lackner
DROP TABLE

```
suspend fun upsertNote(note: Note): EmptyResult<DataError> {
    val localResult = localNoteDataSource.upsertNote(note)

    val remoteResult = remoteNoteDataSource.postNote(note)

    return when(remoteResult) {
        is Result.Error -> {
            val syncWork = OneTimeWorkRequestBuilder<NoteSyncWorker>()
                .setConstraints(Constraints(requiredNetworkType = NetworkType.CONNECTED))
                .setInputData(workDataOf("note" to note))
                .addTag(note.id)
                .build()

            WorkManager
                .getInstance(context)
                .enqueue(syncWork)
            Result.Success(Unit)
        }
        is Result.Success -> {
            localNoteDataSource.upsertNote(remoteResult.data).asEmptyDataResult()
        }
    }
}
```

SINGLE-RESPONSIBILITY PRINCIPLE

Philipp Lackner
DROP TABLE

```
suspend fun upsertNote(note: Note): EmptyResult<DataError> {
    val localResult = localNoteDataSource.upsertNote(note)

    val remoteResult = remoteNoteDataSource.postNote(note)

    return when(remoteResult) {
        is Result.Error -> {
            val syncWork = OneTimeWorkRequestBuilder<NoteSyncWorker>()
                .setConstraints(Constraints(requiredNetworkType = NetworkType.CONNECTED))
                .setInputData(workDataOf("note" to note))
                .addTag(note.id)
                .build()

            WorkManager
                .getInstance(context)
                .enqueue(syncWork)
            Result.Success(Unit)
        }
        is Result.Success -> {
            localNoteDataSource.upsertNote(remoteResult.data).asEmptyDataResult()
        }
    }
}
```

- → A class/function/module should have one reason to change
- → Functionality becomes isolated
- → Fewer dependencies per class, therefore less coupling

SINGLE-RESPONSIBILITY PRINCIPLE

Philipp Lackner
DROP TABLE

```
class NoteSyncScheduler(
    private val context: Context
) {
    fun scheduleSync(note: Note) {
        val syncWork = OneTimeWorkRequestBuilder<NoteSyncWorker>()
            .setConstraints(Constraints(requiredNetworkType = NetworkType.CONNECTED))
            .setInputData(workDataOf("note" to note))
            .addTag(note.id)
            .build()

        WorkManager
            .getInstance(context)
            .enqueue(syncWork)
    }
}
```

RIGHT

SINGLE-RESPONSIBILITY PRINCIPLE

Philipp Lackner
DROP TABLE

```
class NoteRepositoryImpl(
    private val scheduler: NoteSyncScheduler
) {

    suspend fun upsertNote(note: Note): EmptyResult<DataError> {
        val localResult = localNoteDataSource.upsertNote(note)

        val remoteResult = remoteNoteDataSource.postNote(note)

        return when(remoteResult) {
            is Result.Error -> {
                scheduler.scheduleSync(note)
                Result.Success(Unit)
            }
            is Result.Success -> {
                localNoteDataSource.upsertNote(remoteResult.data).asEmptyDataResult()
            }
        }
    }
}
```

RIGHT

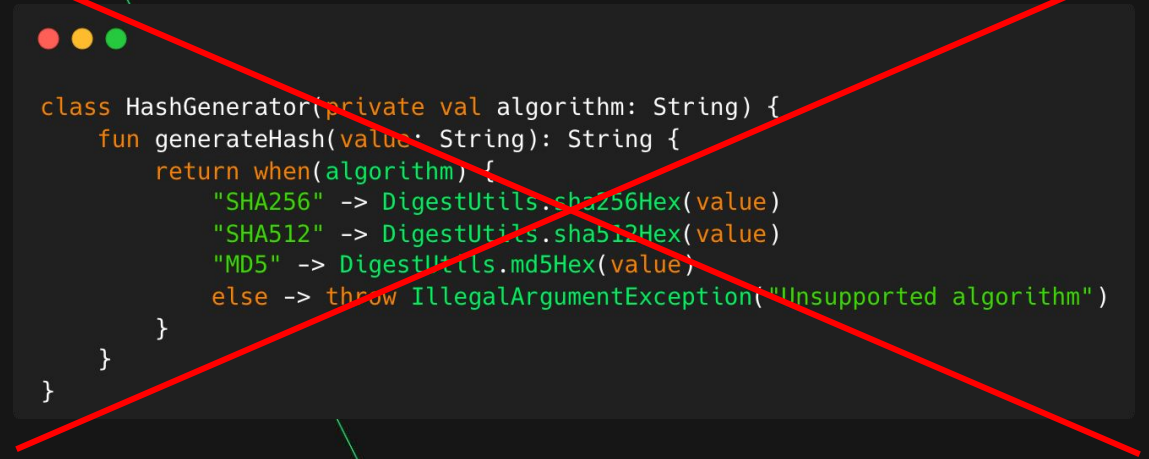
OPEN-CLOSED PRINCIPLE

Philipp Lackner
DROP TABLE

```
class HashGenerator(private val algorithm: String) {
    fun generateHash(value: String): String {
        return when(algorithm) {
            "SHA256" -> DigestUtils.sha256Hex(value)
            "SHA512" -> DigestUtils.sha512Hex(value)
            "MD5" -> DigestUtils.md5Hex(value)
            else -> throw IllegalArgumentException("Unsupported algorithm")
        }
    }
}
```

OPEN-CLOSED PRINCIPLE

Philipp Lackner
DROP TABLE



```
class HashGenerator(private val algorithm: String) {
    fun generateHash(value: String): String {
        return when(algorithm) {
            "SHA256" -> DigestUtils.sha256Hex(value)
            "SHA512" -> DigestUtils.sha512Hex(value)
            "MD5" -> DigestUtils.md5Hex(value)
            else -> throw IllegalArgumentException("Unsupported algorithm")
        }
    }
}
```

- → Classes should be open for extension, but closed for modification
- → You should be able to add new functionality without changing existing one
- → Classes are more decoupled

OPEN-CLOSED PRINCIPLE

Philipp Lackner
DROP TABLE

```
interface HashGenerator {
    fun generateHash(value: String): String
}

class Sha256HashGenerator: HashGenerator {
    override fun generateHash(value: String): String {
        return DigestUtils.sha256Hex(value)
    }
}
```

RIGHT

LISKOV-SUBSTITUTION PRINCIPLE

Philipp Lackner
DROP TABLE

```
abstract class WebClient {
    abstract fun get(url: String)
    abstract fun connectToSocket(url: String)
}

class WebSocketClient: WebClient() {
    override fun get(url: String) {
        throw RuntimeException("Web socket client doesn't support this")
    }

    override fun connectToSocket(url: String) {
        // ...
    }
}
```

LISKOV-SUBSTITUTION PRINCIPLE

Philipp Lackner
DROP TABLE

```
abstract class WebClient {
    abstract fun get(url: String)
    abstract fun connectToSocket(url: String)
}

class WebSocketClient: WebClient() {
    override fun get(url: String) {
        throw RuntimeException("Web socket client doesn't support this")
    }

    override fun connectToSocket(url: String) {
        // ...
    }
}
```

- → Objects of a superclass should be replaceable with objects of its subclasses
- → Functioning of the program should not be affected by replacement
- → Similar to SRP, it enhances code reusability

Philipp Lackner
DROP TABLE

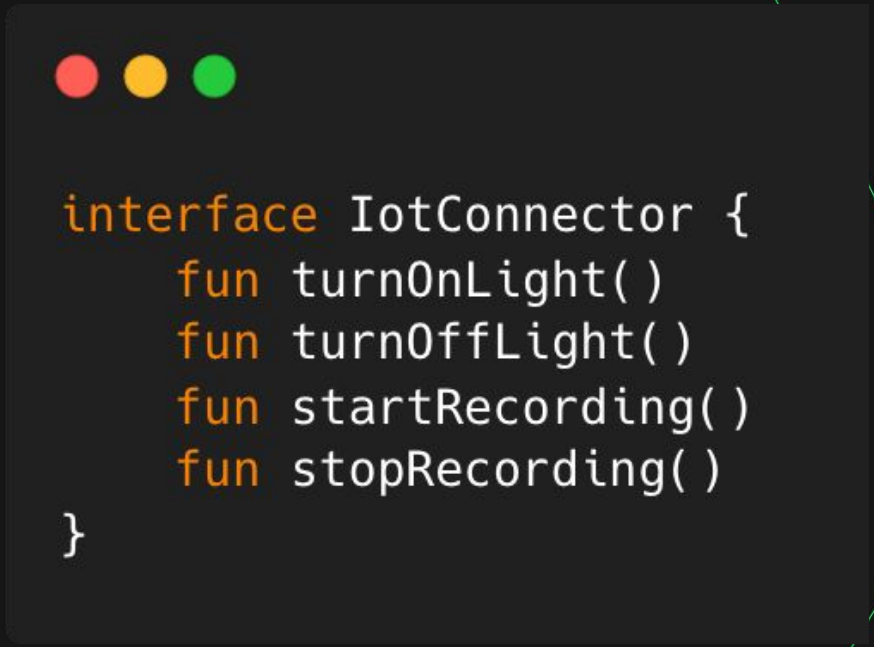
```
abstract class WebClient

abstract class HttpClient: WebClient() {
    abstract fun get(url: String)
}

class HttpClientImpl: HttpClient() {
    override fun get(url: String) {

    }
}
```

RIGHT

```
abstract class WebClient

abstract class WebSocketClient: WebClient() {
    abstract fun connectToSocket(url: String)
}

class WebSocketClientImpl: WebSocketClient() {
    override fun connectToSocket(url: String) {

    }
}
```

INTERFACE-SEGREGATION PRINCIPLE

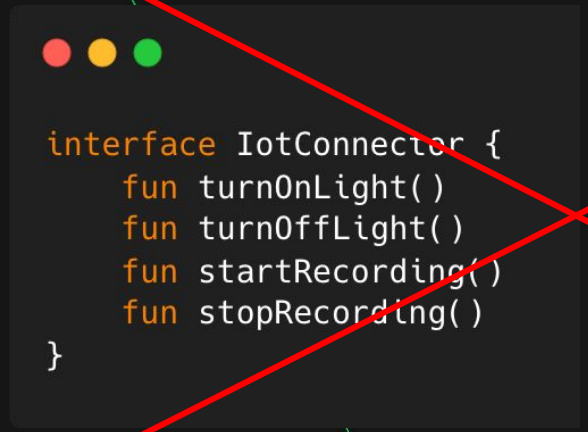
Philipp Lackner
DROP TABLE



```
interface IotConnector {
    fun turnOnLight()
    fun turnOffLight()
    fun startRecording()
    fun stopRecording()
}
```

INTERFACE-SEGREGATION PRINCIPLE

Philipp Lackner
DROP TABLE



```
interface IotConnector {
    fun turnOnLight()
    fun turnOffLight()
    fun startRecording()
    fun stopRecording()
}
```

```
class LightIotConnector: IotConnector {

    override fun turnOnLight() {
        // ...
    }

    override fun turnOffLight() {
        // ...
    }

    override fun startRecording() {}

    override fun stopRecording() {}
}
```

- → Classes should not be forced to implement functions they don't use
- → Following ISP makes code less complex and therefore easier to maintain

```
interface LightConnector {
    fun turnOnLight()
    fun turnOffLight()
}

interface CameraConnector {
    fun startRecording()
    fun stopRecording()
}
```

RIGHT

DEPENDENCY-INVERSION PRINCIPLE

Philipp Lackner
DROP TABLE

```
class RemoteNoteDataSource {
    private val firestore: FirebaseFirestore = FirebaseFirestore.getInstance()

    fun saveNote(note: Note) {
        firestore.collection("notes").document().set(note)
    }

    fun deleteNote(noteId: String) {
        firestore.collection("notes").document(noteId).delete()
    }
}
```

```
class SaveNoteUseCase(
    private val remoteNoteDataSource: RemoteNoteDataSource
) {
    suspend fun execute(note: Note) {
        // ...
    }
}
```

DEPENDENCY-INVERSION PRINCIPLE

Philipp Lackner
DROP TABLE

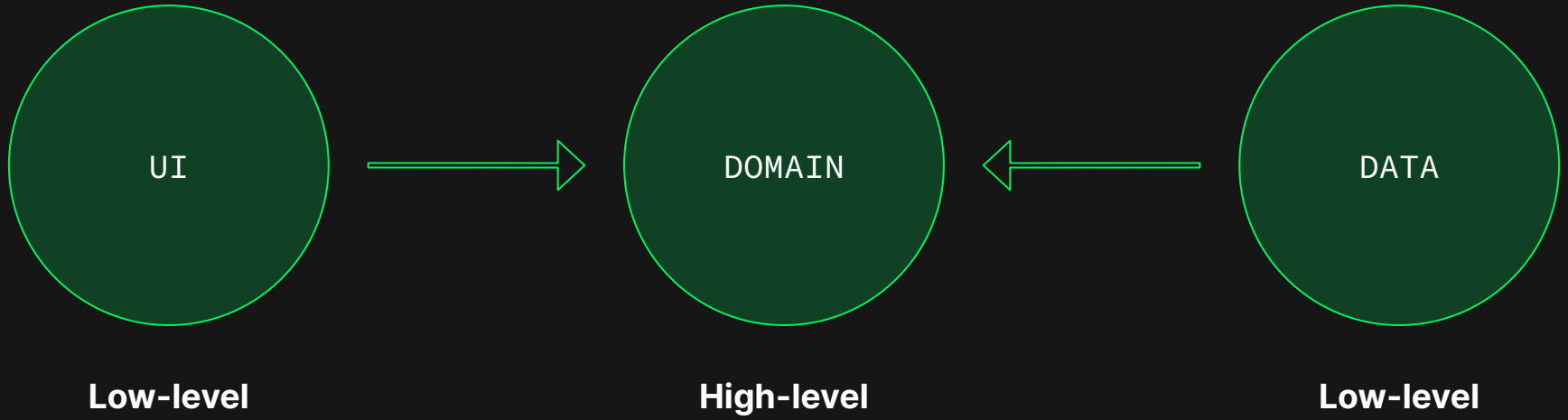
```
class RemoteNoteDataSource {
    private val firestore: FirebaseFirestore = FirebaseFirestore.getInstance()

    fun saveNote(note: Note) {
        firestore.collection("notes").document().set(note)
    }

    fun deleteNote(noteId: String) {
        firestore.collection("notes").document(noteId).delete()
    }
}
```

```
class SaveNoteUseCase(
    private val remoteNoteDataSource: RemoteNoteDataSource
) {
    suspend fun execute(note: Note) {
        // ...
    }
}
```

- → High-level modules should not depend on low-level modules
- → Abstractions should not depend on details
- → Changes in one part of the system won't have an effect on others

DEPENDENCY-INVERSION PRINCIPLE

Philipp Lackner
DROP TABLE

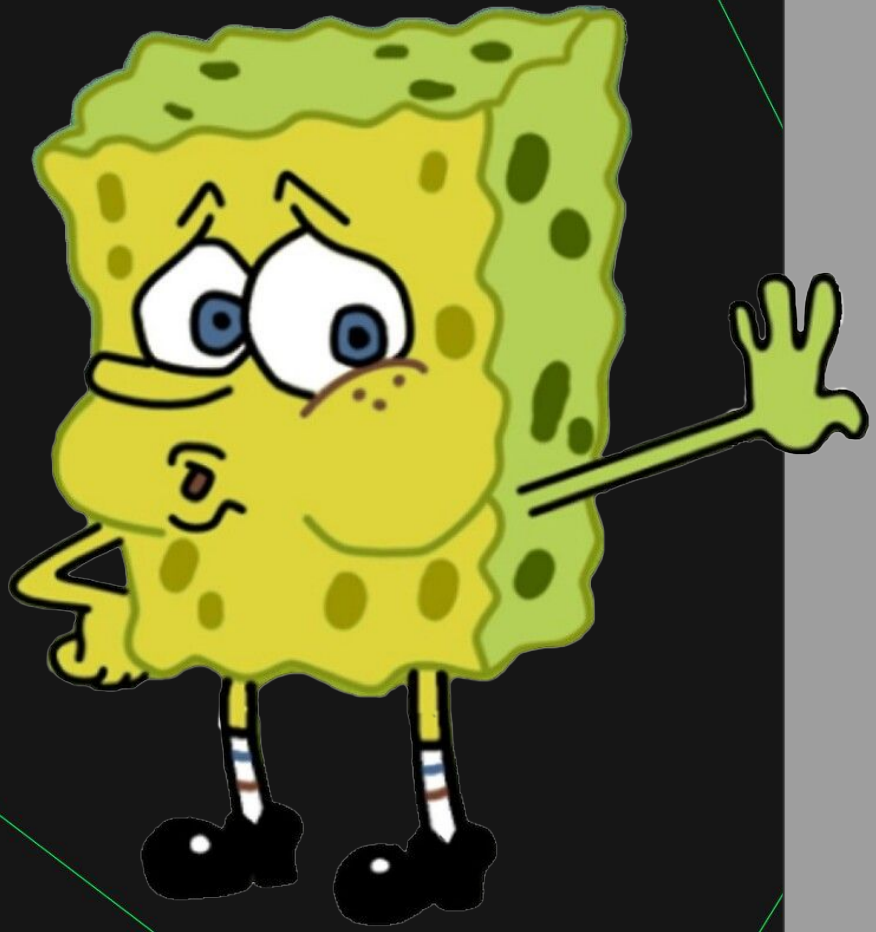


Philipp Lackner
DROP TABLE

```
interface RemoteNoteDataSource {
    fun saveNote(note: Note)
    fun deleteNote(noteId: String)
}

class FirebaseRemoteNoteDataSource : RemoteNoteDataSource {
    private val firestore: FirebaseFirestore = FirebaseFirestore.getInstance()

    override fun saveNote(note: Note) {
        firestore.collection("notes").document().set(note)
    }

    override fun deleteNote(noteId: String) {
        firestore.collection("notes").document(noteId).delete()
    }
}
```

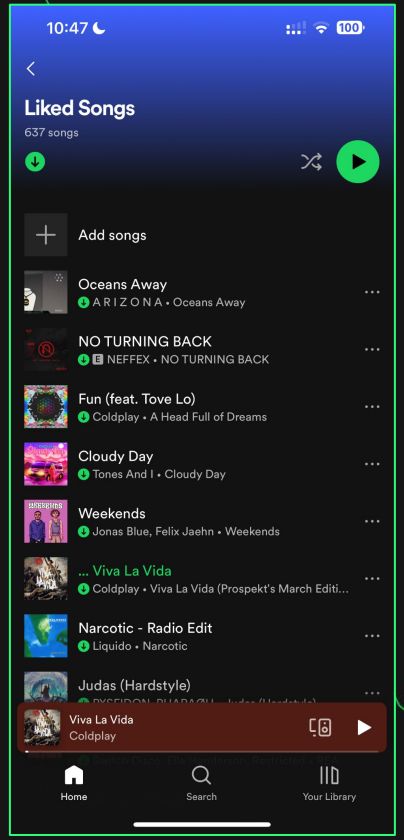
RIGHT

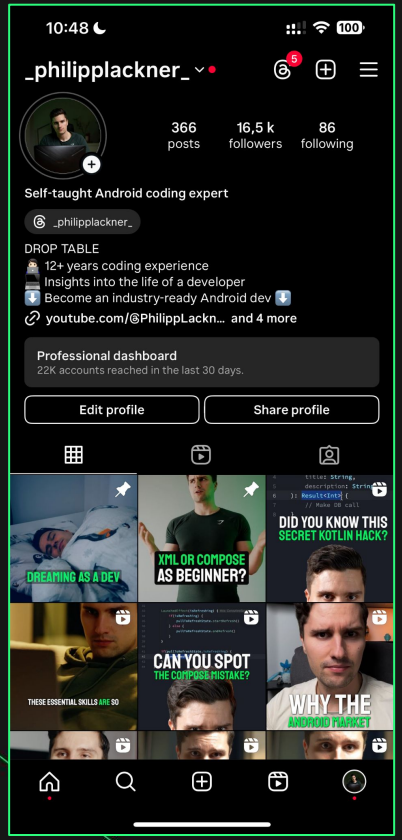
```
class SaveNoteUseCase(
    private val remoteNoteDataSource: RemoteNoteDataSource
) {
    suspend fun execute(note: Note) {
        // ...
    }
}
```

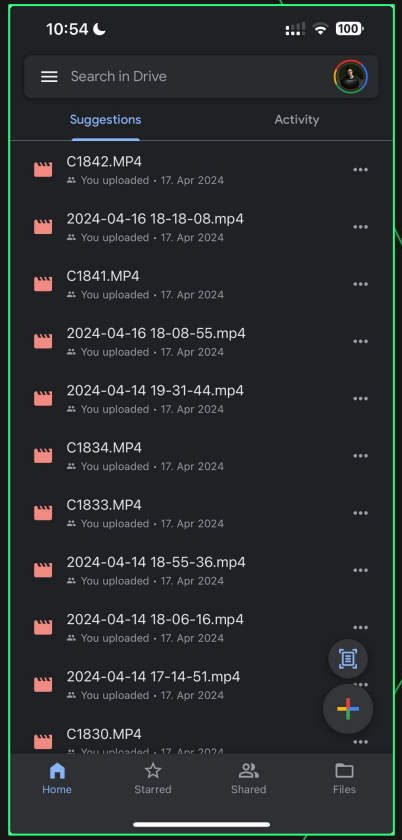


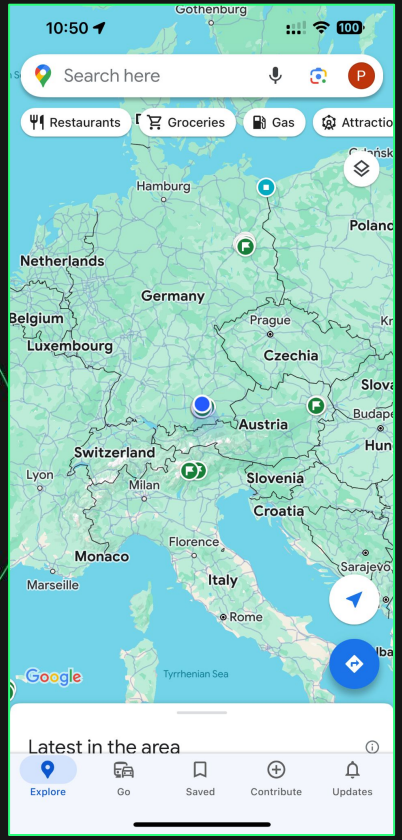
LET'S TAKE A LOOK AT POPULAR APPS

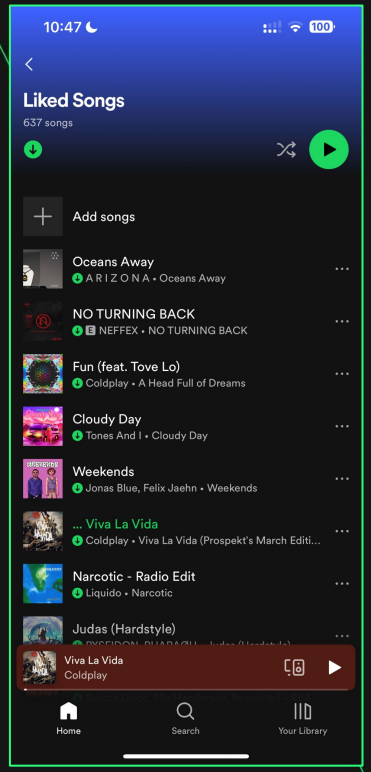
Philipp Lackner
DROP TABLE

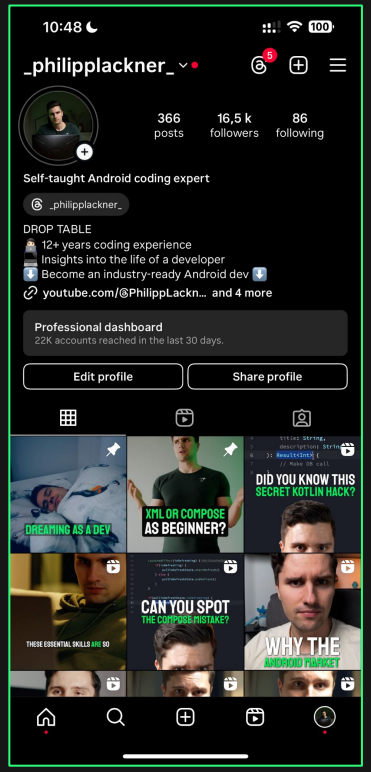


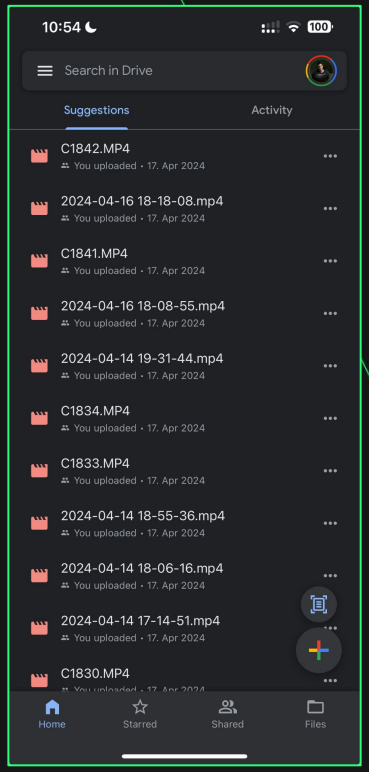


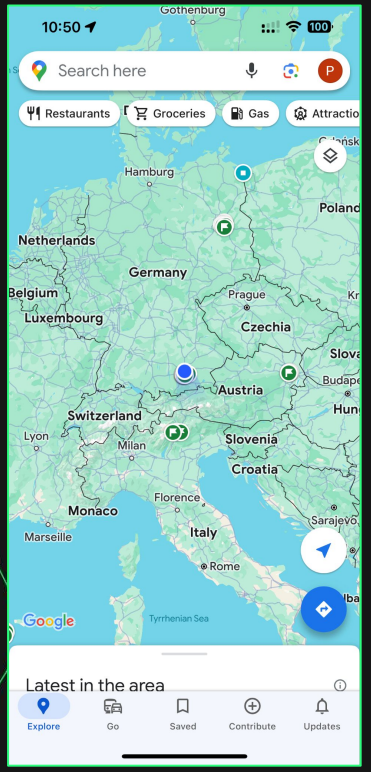












WHAT DO ALL THESE APPS HAVE IN COMMON?

DECLARATIVE UI

Philipp Lackner
DROP TABLE

- A way to build UI by describing **what** should be on the screen
- The framework takes care of the rest
- Used across fields
  - Web development (React, Angular, Vue, etc.)
  - Mobile development (Compose, SwiftUI)

```
@Composable
fun PostFilterButton(
    text: String,
    selected: Boolean,
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    Button(
        onClick = onClick,
        shape = RoundedCornerShape(50f),
        colors = ButtonDefaults.buttonColors(
            containerColor = if (selected) SparkyOrange else
Color.Transparent,
            contentColor = SparkyDarkGray
        ),
        border = if (selected) {
            null
        } else BorderStroke(width = 0.25.dp, color = SparkyDarkGray),
        modifier = modifier
            .padding(vertical = 16.dp)
    ) {
        Text(
            text = text
        )
    }
}
```

DATABASES

Philipp Lackner
DROP TABLE

- Any software that has to store data in a structured format needs a database
- Often used as a cache for faster data access
- Most popular DB frameworks for Android
  - Room
  - Realm
  - SQLDelight

PEOPLE

ID	NAME	SURNAME	AGE
1	Jodie	Tucker	34
2	Jayden	Archer	56
3	Grace	Wheeler	18
4	Freddie	Humphries	56

APIs

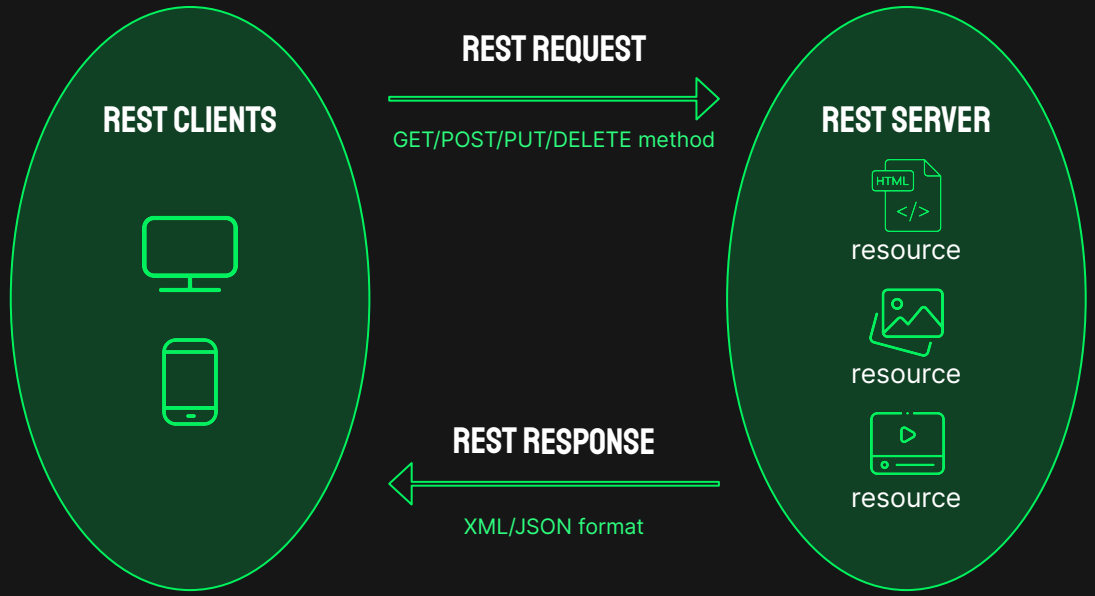
Philipp Lackner
DROP TABLE

- API = Application Programming Interface
- Most people think, API == REST API (Wrong!)

REMOTE APIs

Philipp Lackner
DROP TABLE

- ➔ Remote APIs allow communication to external servers, commonly via HTTP (REST APIs)
- ➔ 83% of Android apps require internet access
- ➔ Most popular libraries for Android
  - → Retrofit
  - → Ktor



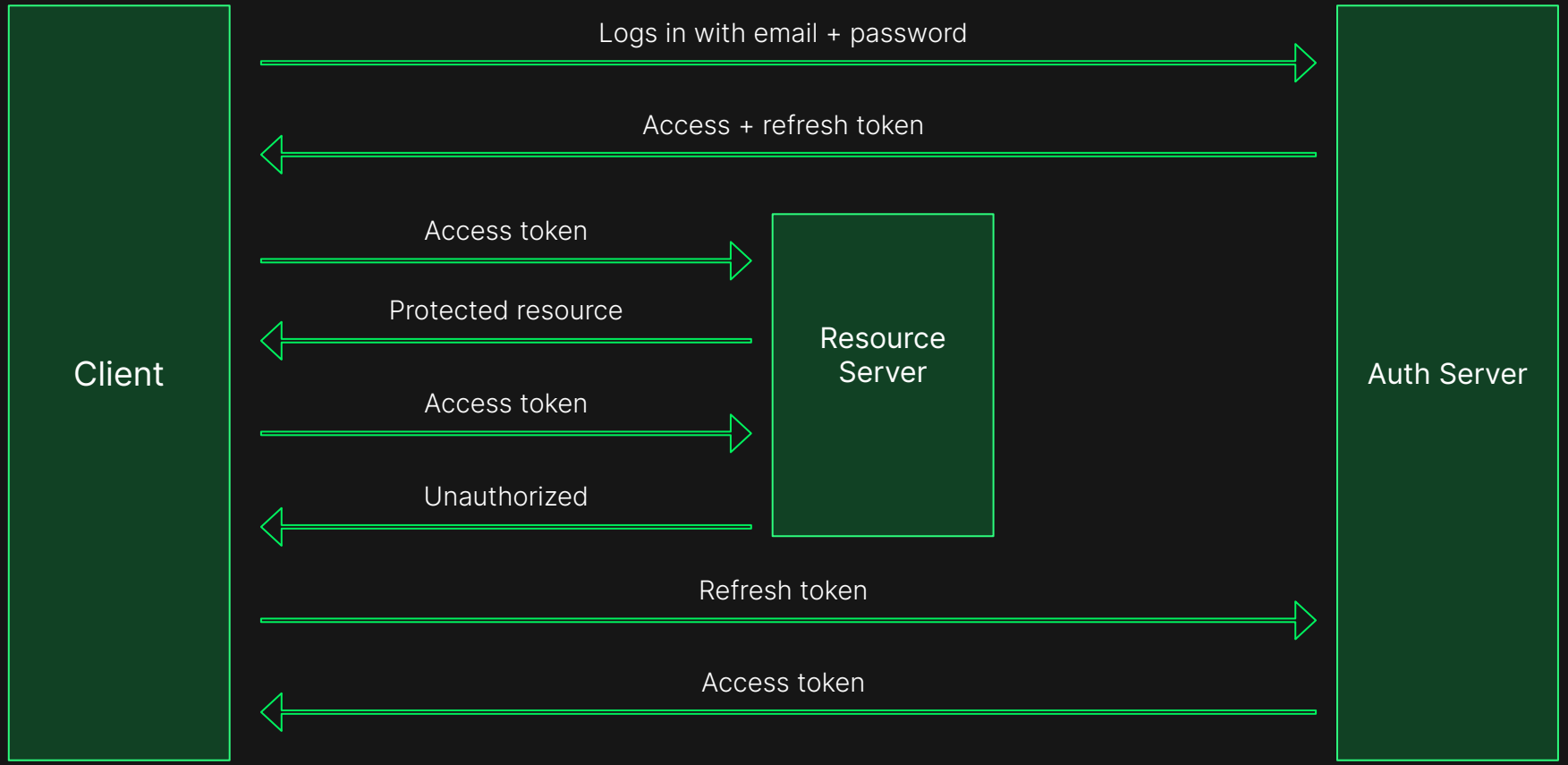
AUTHENTICATION

Philipp Lackner
DROP TABLE

- ➔ Let’s you know who is using the software
- ➔ Applies to various fields
- ➔ Enables many more crucial concepts, such as data sync
- ➔ Lots of different mechanisms
- ➔ One of the most widely-used authentication mechanisms is OAuth

AUTHENTICATION

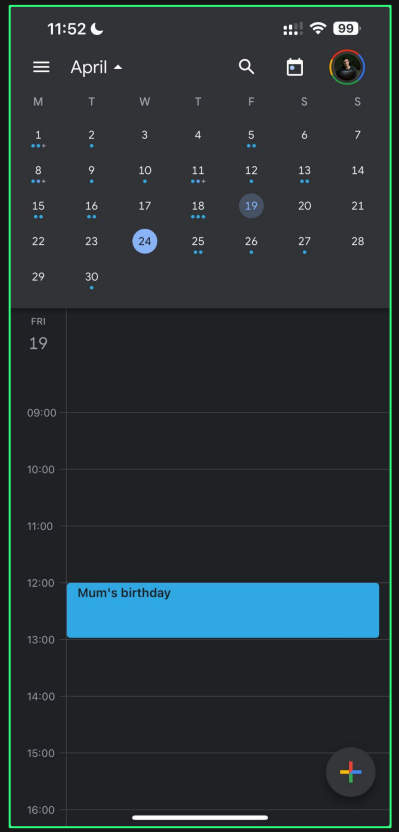
Philipp Lackner
DROP TABLE



OFFLINE-FIRST DEVELOPMENT

Philipp Lackner
DROP TABLE

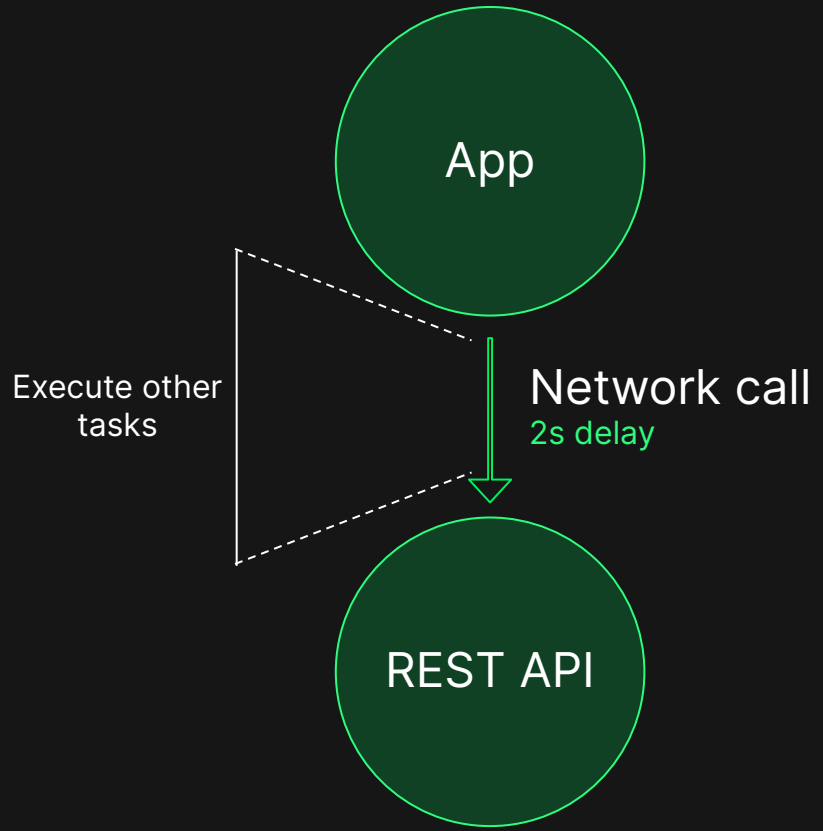
- ➔ Paradigm to maximize the usefulness of software when being offline
- ➔ Mostly relevant for mobile apps
- ➔ Complexity is usually VERY underestimated by developers
  - → Dealing with failed syncs
  - → Dealing with multiple devices
  - → Dealing with multiple logins
- ➔ Not only meant to allow offline usage, but also to make data access faster
- ➔ Even backends often have caching mechanisms



ASYNC / REACTIVE PROGRAMMING

Philipp Lackner
DROP TABLE

- Async programming allows execution of tasks independent of the program flow to wait for other operations to complete
- Reactive programming allows to automatically react to changes
- Both are concepts applicable to ALL fields of software development
- On Android most commonly implemented via coroutines & Flows



ASYNC / REACTIVE PROGRAMMING

HOW MANY PROBLEMS CAN YOU FIND?

Philipp Lackner
DROP TABLE

```
suspend fun syncPendingNotes(sessionStorage: SessionStorage): EmptyResult<DataError> {
    val userId = sessionStorage.get()?.userId
        ?: return Result.Error(DataError.Local.NO_USER)

    val createdNotes = notePendingSyncDao.getAllNotePendingSyncEntities(userId)

    createdNotes.forEach { note ->
        try {
            remoteNoteDataSource.postNote(note)

            notePendingSyncDao.deletePendingSync(note.id)
        } catch (e: Exception) {
            return Result.Error(DataError.Network.UNKNOWN)
        }
    }

    return Result.Success(Unit)
}
```

1.

2.

3.

PROJECT MANAGEMENT

MULTI-MODULE ARCHITECTURE & GRADLE

- ➔ Multi-module architecture splits your codebase into libraries
  - → Reusing code across platforms
  - → Enforcing clear boundaries
  - → Faster builds
  - → Better separation in a team
- ➔ Not learning Gradle will result in a huge mess