

Android



App Arkitektur

Erlend Stenlund

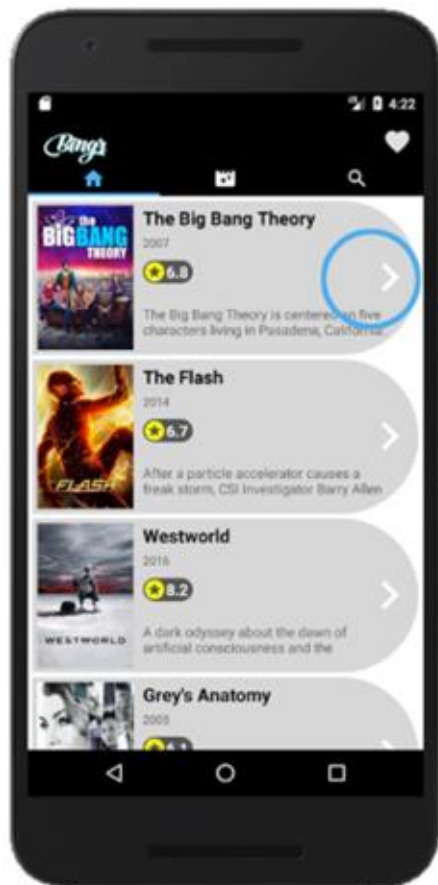
- Seniorutvikler fra Storebrand
- Java-programmør
- Master fra IFI
 - prosa
- Tidligere gruppelærer i IN2000
 - V19
 - V20
- Android apputvikling som hobby



Min bakgrunn fra IN2000

- Tok piloten IN2001 i 2018
- Case: **Film og Serie-app**
 - Data fra The Movie DataBase
- Kodet i Java med XML til layout 😐
- Teambuilding er viktig ❤️





Agenda

12.15 - 13.00

- Grunnprinsipper for arkitektur
- Tilstand
- Lagdelt arkitektur
 - UI-laget
 - ViewModels
 - Unidirectional Data Flow
 - State Hoisting
 - Data-laget
- Dependency Injection

13.15 - 14.00

- Live koding
 - Namage
 - WizApp

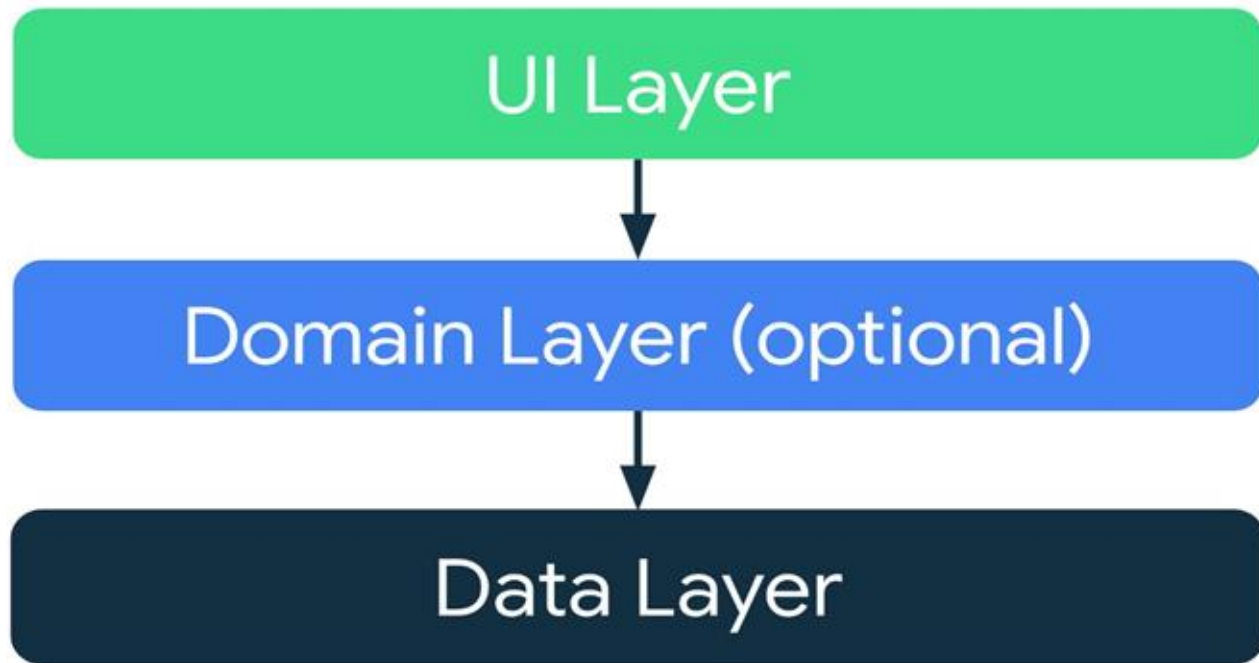




Hva kjennetegner god arkitektur?

- Robusthet
- Vedlikeholdbarhet
- Skalerbarhet
- Testbarhet
- Konsisitet
- **Separations of Concern**
 - tydelige ansvarsområder
 - modularisering
- **Single Source of Truth**
 - sentrale og pålitelige kilder til data
 - sikre at vi bruker den mest presiste og oppdaterte dataen

Lagdelt arkitektur



UI Layer

```
graph TD; subgraph UI_Layer [UI Layer]; UI_elements[UI elements] --> State_holders[State holders]; end; UI_Layer --> Domain_Layer[Domain Layer (optional)]; Domain_Layer --> Data_Layer[Data Layer];
```

UI elements

State holders

Domain Layer (optional)

Data Layer

Hva er tilstand?

- Enhver verdi som kan endres over tid
- Vi bruker **remember** for å lagre tilstand i Composables
 - Kan lagre både mutable og immutable objekter

@Composable

```
fun Component() {  
    val value1 = remember { mutableStateOf("") }  
    var value2 by remember { mutableStateOf("") }  
    val (value3, setValue) = remember { mutableStateOf("") }  
}
```

Tilstand med Jetpack Compose

- Med Jetpack Compose kan vi lage **Composable-funksjoner**
 - beskriver hvordan innholdet på skjermen skal se ut
 - ofte kalt **UI-komponenter**
- Composable funksjoner plasseres i et slags hierarkisk / tre-lignende struktur som utgjør hele appen
 - ofte kalt **UI-treet**

Rekomposisjoner

- Prosessen der en composable-funksjon blir kjørt på nytt
 - oppdaterer UI når tilstanden den avhenger av endres
- Compose avgjør hvilke deler av UI-treet som må oppdateres
 - kjører kun de nødvendige composable-funksjonene
 - hopper over øvrige composable-funksjoner
 - gjør apper mer effektive

Pop Quiz

Med gjeldende kode, hva vises på skjermen?

```
@Composable
fun Component() {
    val textState = remember { mutableStateOf("") }

    Text(text = textState.value)

    textState.value = "hvor har verdien endret seg?"
}
```



UI Layer

```
graph TD; subgraph UI_Layer [UI Layer]; UI_elements[UI elements] --> State_holders[State holders]; end; State_holders --> Domain_Layer[Domain Layer (optional)]; Domain_Layer --> Data_Layer[Data Layer];
```

UI elements

State holders

Domain Layer (optional)

Data Layer

ViewModel - kort forklart

ViewModel i Android er en klasse som **holder og administrerer UI-relatert data og logikk** på skjermnivå

ViewModel - helt konkret

```
class ExampleViewModel : ViewModel() { ... }
```

ViewModel - hvorfor trenger vi det?

- Bidrar til å separere forretningslogikk fra brukergrensesnittet
 - ofte kalt noe med **UiState**
- Brukes til å håndtere og lagre UI-relatert data på en **livssyklusbevisst** måte
 - F.eks.: skjerm-rotasjon
- Bruker **Flows** som UI kan observere på

MutableStateFlow

- Den interne tilstanden i en ViewModel
 - skal ikke eksponeres til UI
 - kan aksesseres med sin `value`-property
- Navngis med understrek først i variabel-navnet
- Må initialiseres med en verdi

```
class ExampleViewModel : ViewModel() {  
    private val _myStateFlow = MutableStateFlow<Int>(0)  
}
```

StateFlow

- `asStateFlow()` gjør tilstanden til en **read-only** flow
- `myStateFlow` er en eksponert variabel
 - UI kan observere, men ikke endre verdien
 - Er en **hot flow** 🔥

```
class ExampleViewModel : ViewModel() {  
    private val _myStateFlow = MutableStateFlow<Int>(0)  
    val myStateFlow: StateFlow<Int> = _myStateFlow.asStateFlow()  
}
```

ViewModel med UiState

```
class ExampleViewModel(  
    private val repository: Repository  
) : ViewModel() {  
  
    private val _uiState = MutableStateFlow<UiState>(UiState(null))  
    val uiState: StateFlow<UiState> = _uiState.asStateFlow()  
  
    fun getResult(id: String) {  
        viewModelScope.launch {  
            val result : String? = repository.getResult(id)  
            _uiState.value = UiState(result = result)  
        }  
    }  
}  
  
data class UiState(val result: String?)
```

ViewModel i Compose

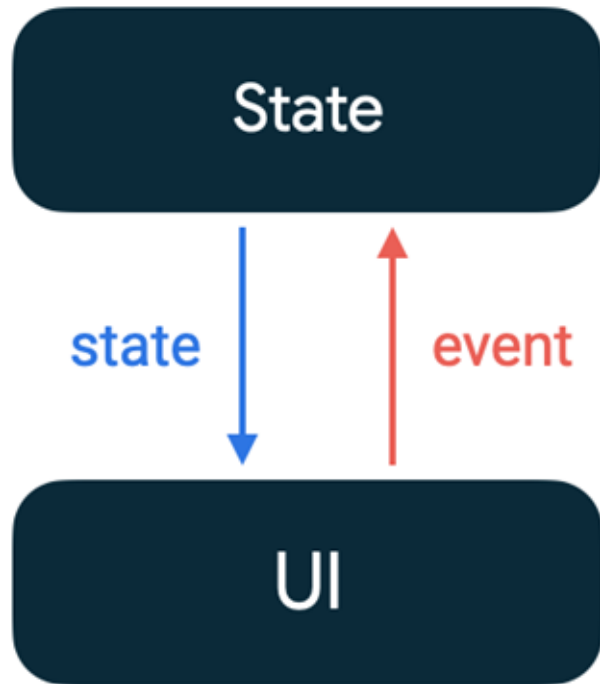
- Composables kan konsumere datastrøm fra en ViewModel
 - `collectAsState()`
 - `collectAsStateWithLifecycle()`

`@Composable`

```
fun Component(modifier: Modifier, viewModel: ExampleViewModel = ViewModel()) {  
    val uiState by viewModel.uiState.collectAsState()  
  
    Column(modifier = modifier) {  
        Button(onClick = { viewModel.getResult() }) {  
            Text(text = "Hent resultater")  
        }  
        uiState.result?.let {  
            ResultComponent(result = it)  
        }  
    }  
}
```

Unidirectional Data Flow

- **ViewModel** håndterer tilstand og logikk.
- Composables **observerer** på tilstand
 - `collectAsState()`
 - Viser data
- Sender **events** fra UI til ViewModel
 - `onClick()`
 - Oppdaterer tilstand



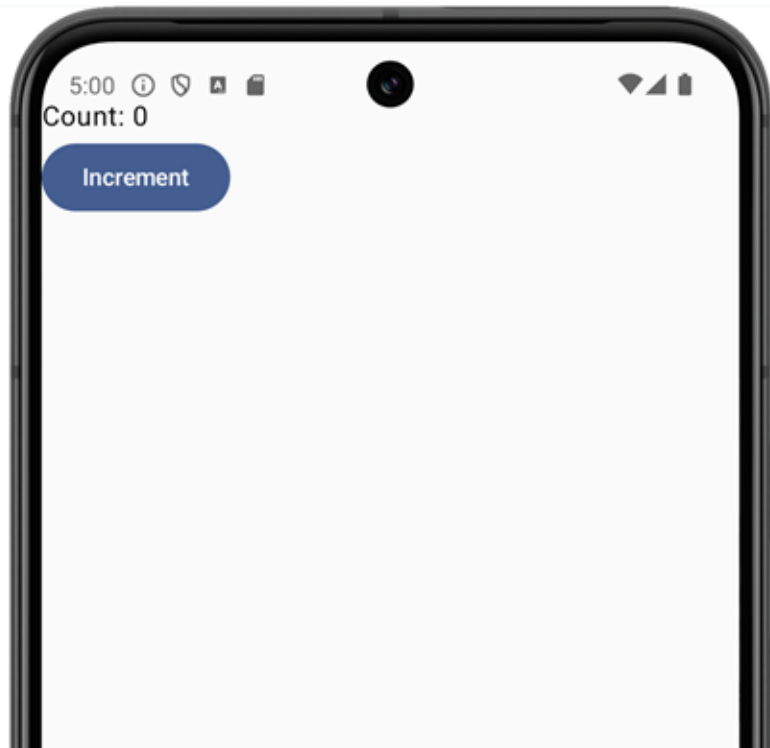
State hoisting

- Bruk av `remember` gjør composables **stateful**
- **Ulemper:**
 - Høy kobling (*avhengighet*) mellom tilstand og composables
 - Vanskelig å debugge
 - Begrenser gjenbruk
- State hoisting er en måte å **løfte tilstand** ut av composables
 - les: Flytt `remember` ut av `@Composable`
 - Dette gjør composables **stateless**

State hoisting

```
@Composable
fun CounterScreen() {
    Counter()
}
```

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }
    Column {
        Text("Count: $count")
        Button(onClick = { count++ }) {
            Text("Increment")
        }
    }
}
```



State hoisting

Vi kan introdusere Composables med to parametre:

- **value : T**
verdien som vises på skjermen
- **onValueChange: (T) -> Unit**
event som gir beskjed at verdien er oppdatert

```
@Composable
```

```
fun Component(value: String, onValueChange: (String) -> Unit) {  
    . . .  
}
```

State hoisting

```
@Composable
fun CounterScreen() {
    Counter()
}
```

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }
    Column {
        Text("Count: $count")
        Button(onClick = { count++ }) {
            Text("Increment")
        }
    }
}
```

```
@Composable
fun CounterScreen() {
    var count by remember { mutableStateOf(0) }
    Counter(count = count, onIncrement = { count++ })
    AnotherComponent(count = count)
}
```

```
@Composable
fun Counter(count: Int, onIncrement: () -> Unit) {
    Column {
        Text("Count: $count")
        Button(onClick = onIncrement) {
            Text("Increment")
        }
    }
}
```

State hoisting

- Løfte tilstand opp til **Lowest Common Ancestor** (LCA)
 - burde være nærmest der tilstand blir konsumert (lest / endret)
 - kan også være utenfor komposisjonen → ViewModel
- **Fordeler:**
 - flere composables kan få tilgang til en tilstand
 - bidrar til modularitet
- Ikke alltid nødvendig med å heise tilstanden
 - dersom composable trenger å ha intern tilstand
 - tilstanden trengs ikke utenfor komponenten

Data Layer

Application
data

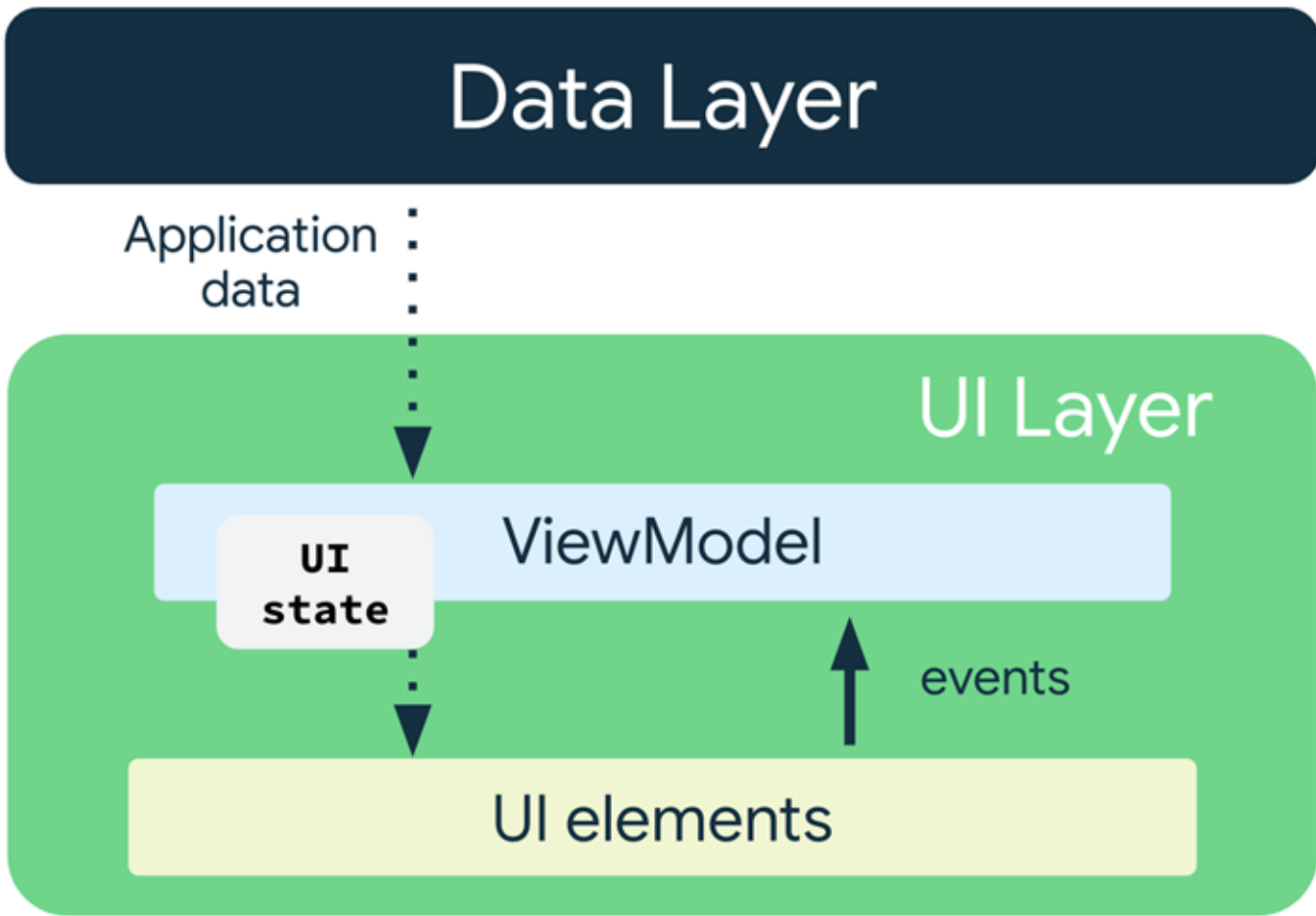
UI Layer

ViewModel

**UI
state**

events

UI elements



UI Layer



Domain Layer (optional)



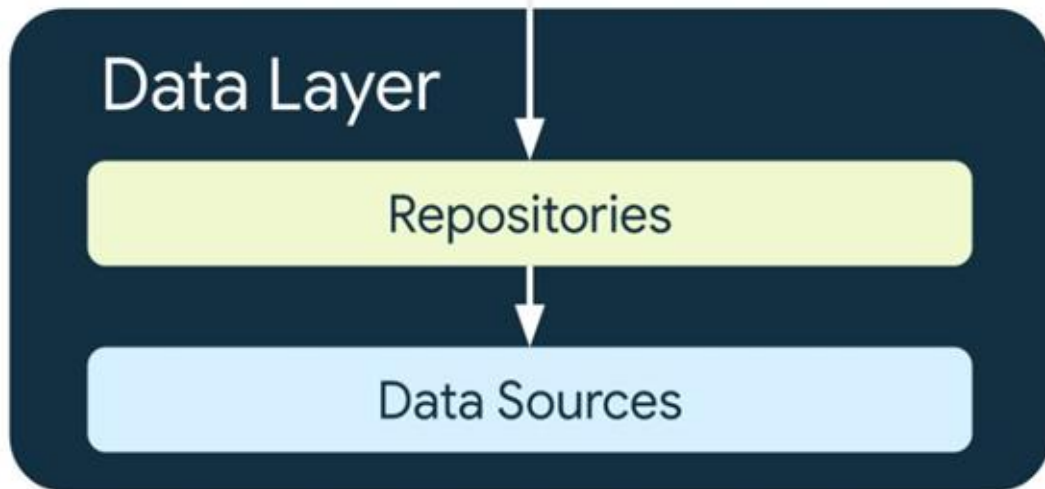
Data Layer



Repositories



Data Sources



Data Source

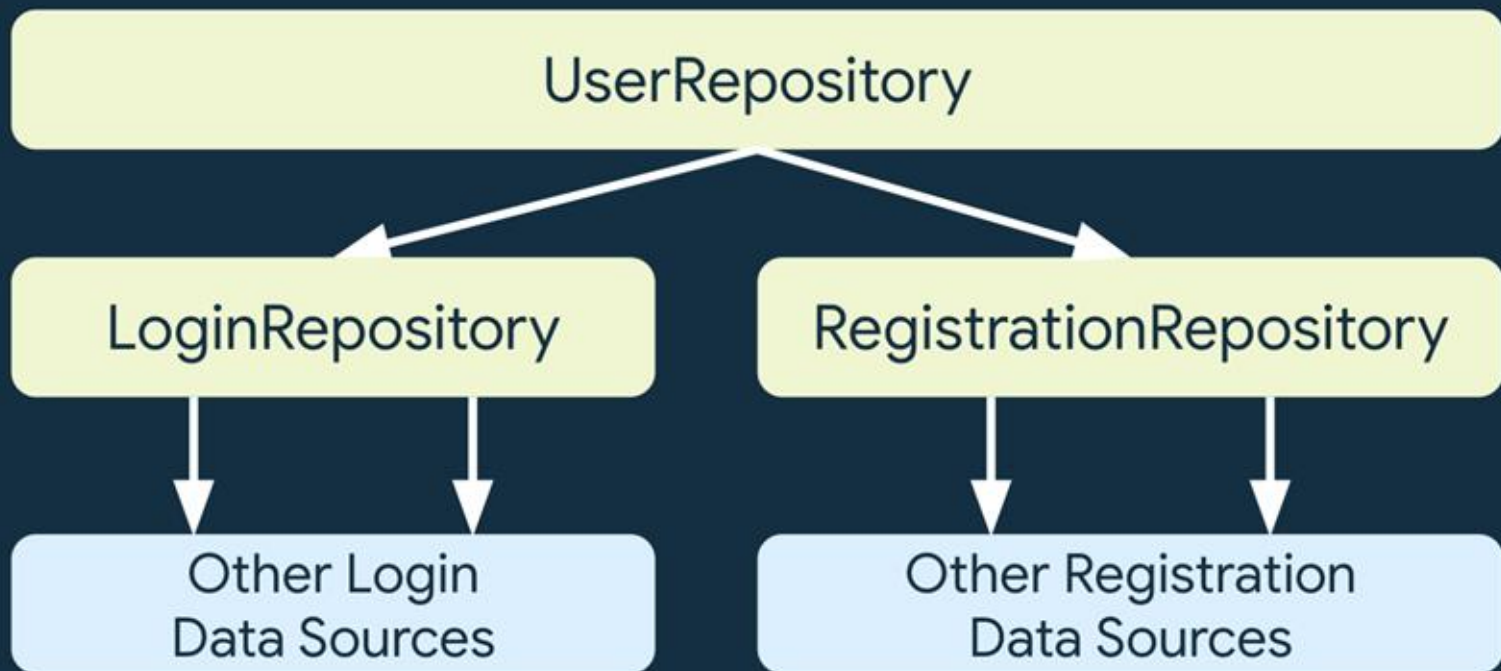
- Klasser som har ansvaret for å hente data fra ulike kilder
 - **API**
 - **databaser**
 - filer
- Eksponerer funksjoner som kan brukes av repository
 - bør være **suspend** funksjoner
 - bør være thread safe

Repository

- En ordnet måte å **hente data**
 - kan kombinere data fra 0 → flere datakilder
 - kan kombinere data fra 0 → flere repositories
- Eksponerer ofte suspend funksjoner

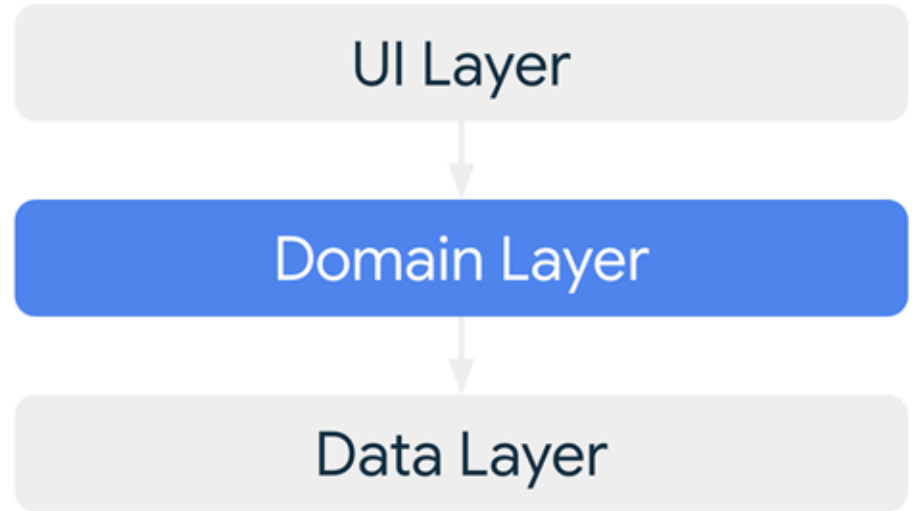
```
class Repository(private val datasource: Datasource) {  
  
    suspend fun getData(name: String): Response {  
        val id = datasource.getIdFromName(name)  
        return datasource.getResult(id)  
    }  
}
```

Data Layer



Domenelaget

- Ansvarlig for **avansert forretningslogikk**
 - operasjoner på data
- Fordeler ansvar på "use cases"



Domain Layer

GetLatestNewsWithAuthors
Use Case



FormatDateUseCase

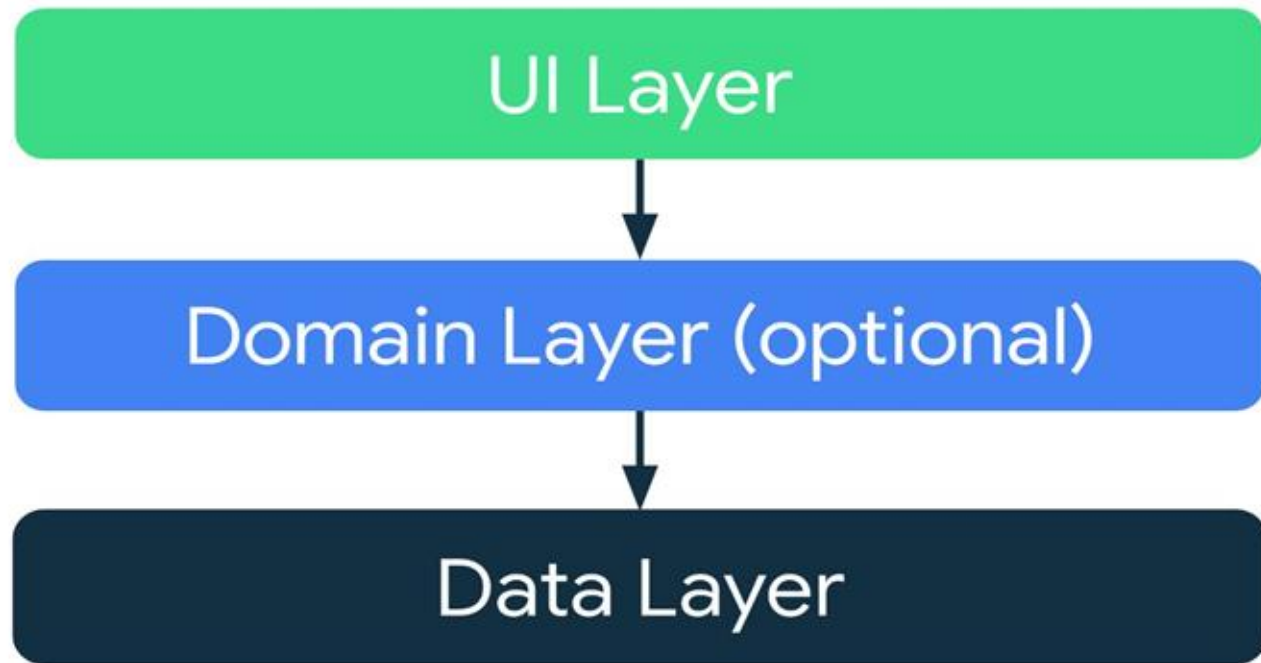
Data Layer

NewsRepository

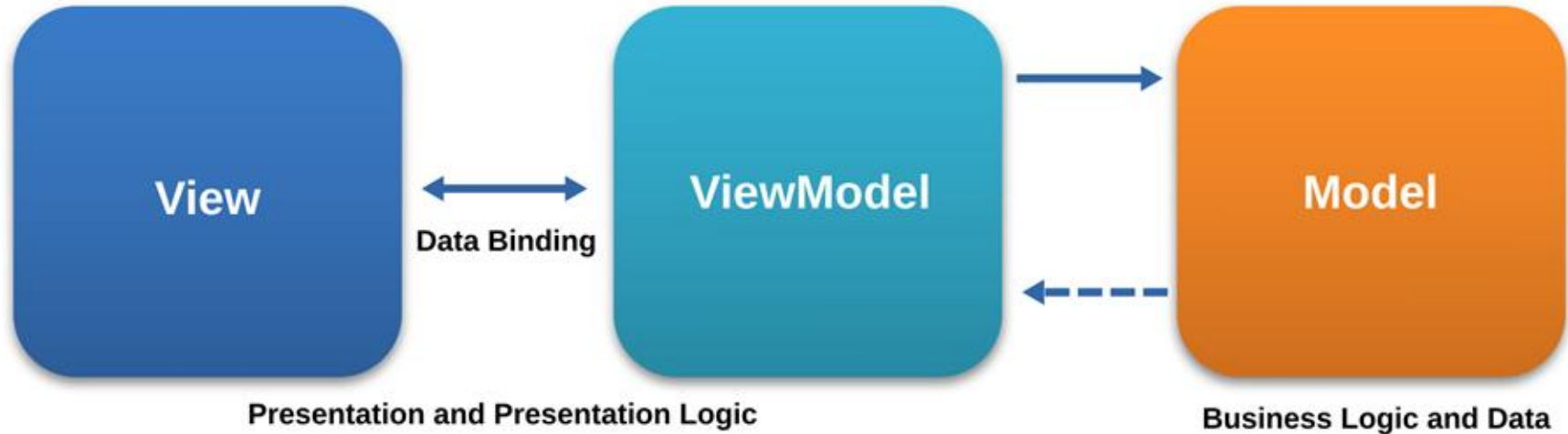
AuthorsRepository



Lagdelt arkitektur



MVVM - Model View ViewModel



MVVM - Model View ViewModel

- Benytter
 - **Model** (data/domenelogikk)
 - **View** (UI)
 - **ViewModel** (binder UI og data, holder tilstand)
- ViewModel kan ha flere flows for UI-tilstand
- Lav terskel for utviklere
 - mye eksempelkode og bibliotekstøtte
- Passer godt for enkle og middels komplekse skjermer
- Svært utbredt i Android og matcher Googles anbefalte arkitektur

Klasse med definert avhengighet

```
class Car {  
  
    private val engine = Engine()  
  
    fun start() {  
        engine.start()  
    }  
}  
  
fun main(args: Array) {  
    val car = Car()  
    car.start()  
}
```

Klasser med tildelt avhengighet

- Constructor Injection

```
class Car(private val engine: Engine) {  
    fun start() {  
        engine.start()  
    }  
}  
  
fun main(args: Array) {  
    val engine = Engine()  
    val car = Car(engine)  
    car.start()  
}
```

- Field Injection

```
class Car {  
    lateinit var engine: Engine  
  
    fun start() {  
        engine.start()  
    }  
}  
  
fun main(args: Array) {  
    val car = Car()  
    car.engine = Engine()  
    car.start()  
}
```

Dependency Injection (DI)

- Vi gjør ofte **manuell** dependency injection
 - blir komplekst å koble mange avhengigheter
 - boilerplate
- Dependency Injection kan automatiseres
 - kan skrive mer gjenbrukbar kode
 - gjør refaktoring lettere
 - gjør testing enklere

Dependency Injection (DI)

```
@HiltViewModel
```

```
class ExampleViewModel @Inject constructor(  
    private val repository1: Repository1,  
    private val repository2: Repository2,  
    private val repository3: Repository3,  
    private val repository4: Repository4  
) : ViewModel() {  
    . . .  
}
```

```
ExampleViewModel()
```

Dagger

- Bibliotek for DI
- Genererer kode under kompileringstid
- `@Inject`
- Brukes gjerne i store applikasjoner
- Mye boilerplate

Hilt

- Bibliotek for DI laget for Android
- Basert på Dagger
- Forenkler DI uten å måtte konfigurere Dagger manuelt
- Bruker annoteringer som:
 - `@HiltAndroidApp`
 - `@HiltViewModel`
- Litt mindre boilerplate

Hilt & Dagger Annotations

v2.33 - @AndroidDev

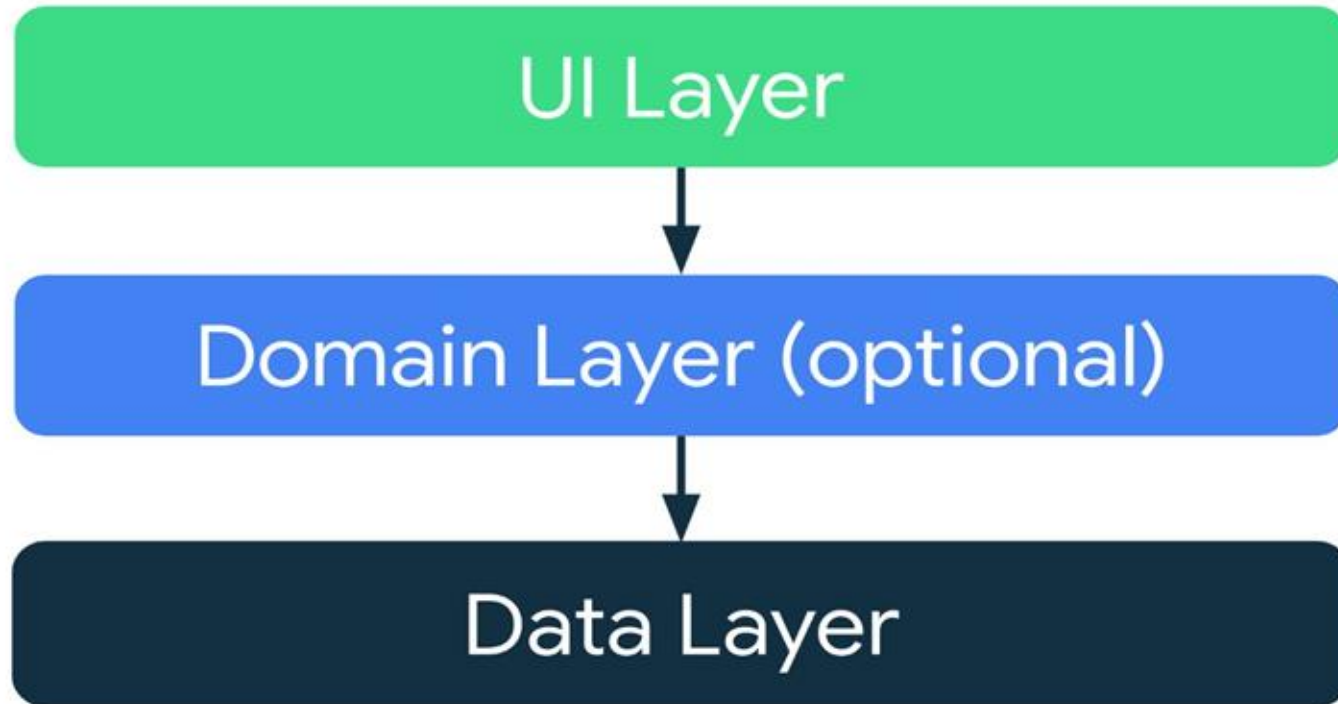
Annotation	Usage	Code Sample
@HiltAndroidApp	Kicks off Hilt code generation. Must annotate the Application class.	<pre>@HiltAndroidApp class MyApplication : Application() { ... }</pre>
@AndroidEntryPoint	Adds a DI container to the Android class annotated with it. This requires using Hilt's Gradle Plugin.	<pre>@AndroidEntryPoint class MyActivity : AppCompatActivity() { ... }</pre>
@Inject	Constructor Injection. Tells which constructor to use to provide instances and which dependencies the type has.	<pre>class AnalyticsAdapter @Inject constructor(private val service: AnalyticsService) { ... }</pre>
	Field injection. Populates fields in @AndroidEntryPoint annotated classes. Fields cannot be private.	<pre>@AndroidEntryPoint class MyActivity : AppCompatActivity() { @Inject lateinit var adapter: AnalyticsAdapter ... }</pre>
@HiltViewModel	Tells Hilt how to provide instances of an Architecture Component ViewModel.	<pre>@HiltViewModel class MyViewModel @Inject constructor(private val adapter: AnalyticsAdapter, private val state: SavedStateHandle) : ViewModel() { ... }</pre>



PAUSE



Liten repetisjon



Data Layer

Application
data

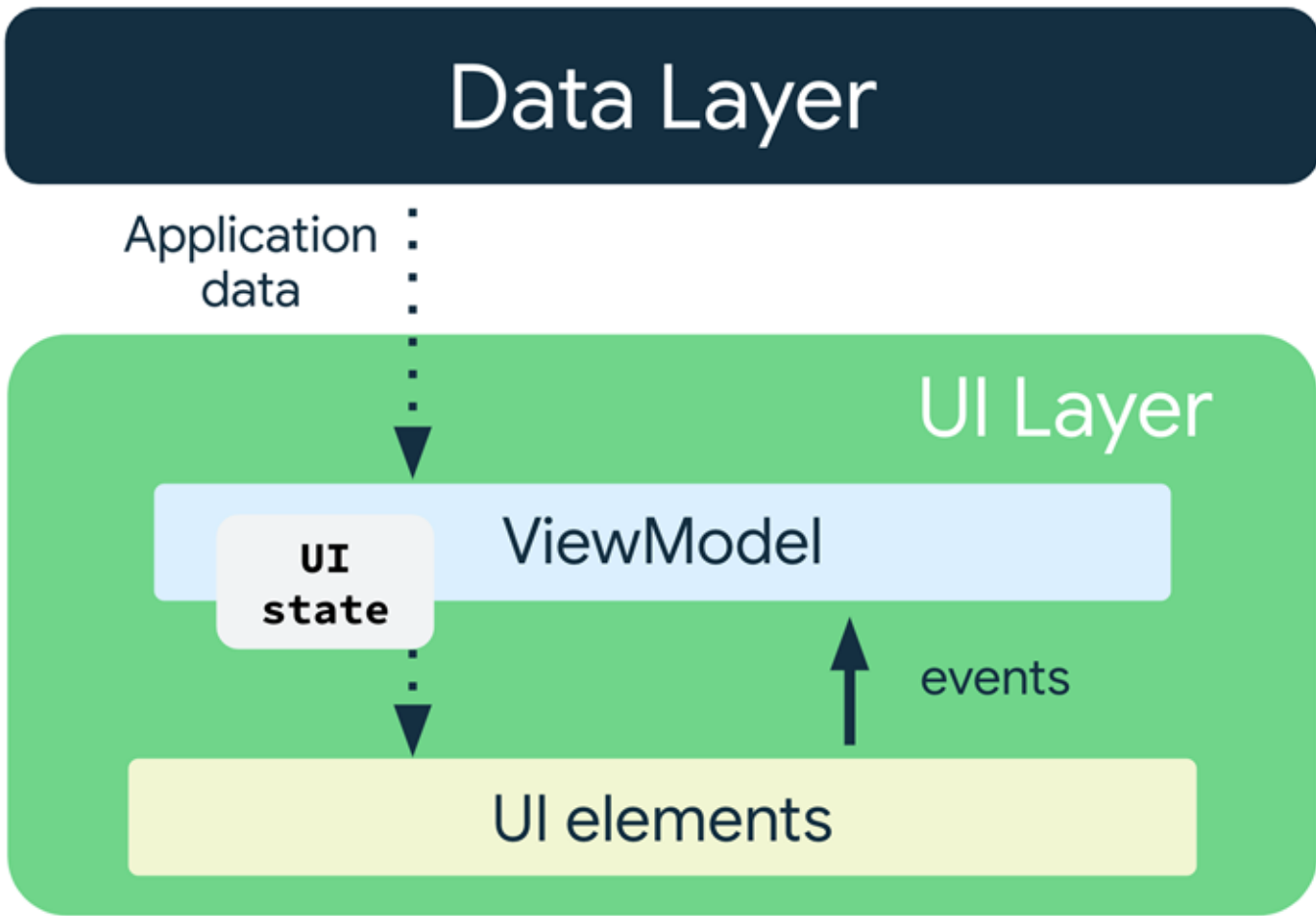
UI Layer

ViewModel

**UI
state**

events

UI elements



Namage

- Ta input fra bruker
 - Lagre i en state
- Definere ViewModel med alders-resultater
- Hente alder via API
- Vise oversikt over registrerte alders-resultater
- <https://github.com/erlenste/Namage>

WizApp 🧐

- Wizard App
- Hente data fra API
- Definere ViewModel med
 - Resultat
 - Innlastning
 - Feilhåndtering
- Dependency Injection
- <https://github.com/erlenste/WizApp>

Lykke til med prosjektarbeidet!

Erlend Stenlund
erlend.stenlund@storebrand.no