Apputvikling 1-2-3

Arkitektur

Agenda

- Generelt om arkitektur
- MVVM + repository pattern
- Food Temprature app
  - Den skal basert på en tempratur velge en oppskrift du vil lage.
  - Den skal hente data fra et Food api og NowCast
  - Og vi vil følge androids guide to app architecture, for å lage appen

Høy kohesjon

Lav kobling

Høy kohesjon

Et objekt/modul som har moderat ansvar og utfører et begrenset antall oppgaver innenfor ett funksjonelt område har høy kohesjon.

Lav kobling

Høy kohesjon

Et objekt/modul som har moderat ansvar og utfører et begrenset antall oppgaver innenfor ett funksjonelt område har høy kohesjon.

Lav kobling

Et objekt/modul som er avhengig av få andre objekter/moduler har lav kobling.

Praktisk sett?

Høy kohesjon

Lav kobling

Praktisk sett?

Høy kohesjon

Vi skiller ulike ansvarsoppgaver i forskjellige klasser/moduler. (men de som er like holder vi sammen)

Lav kobling

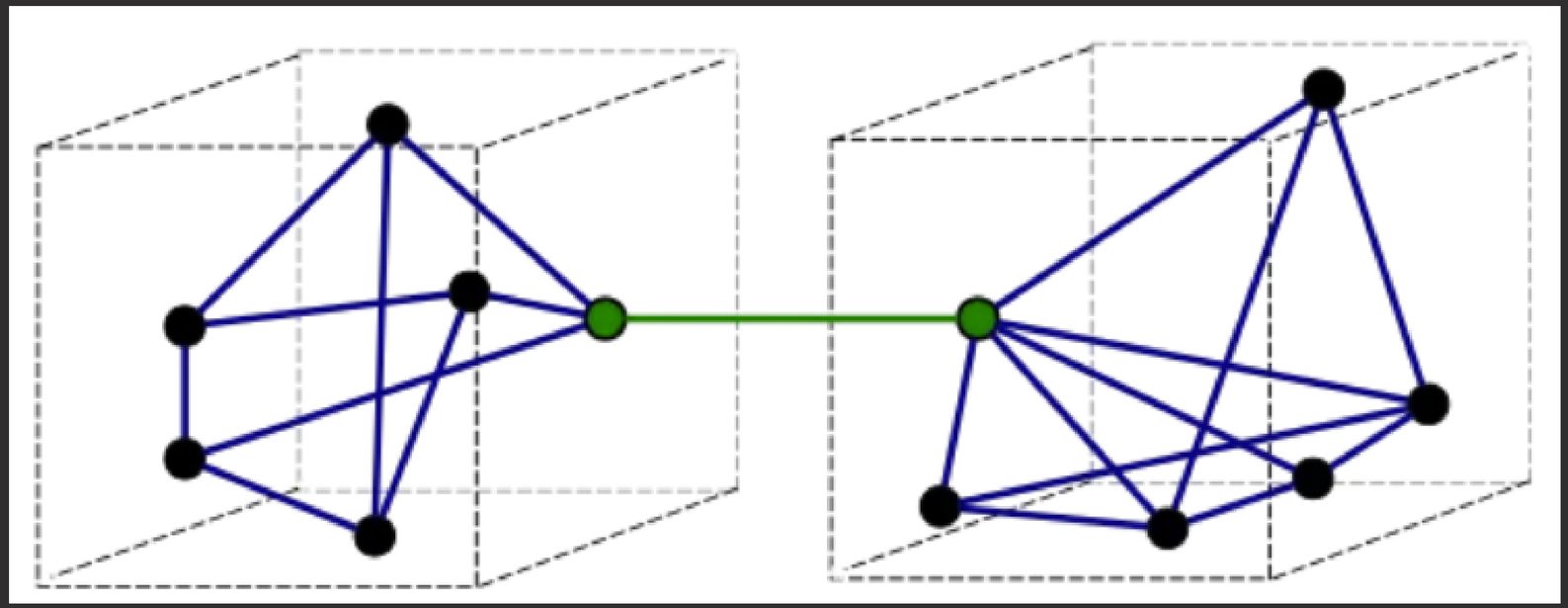
Praktisk sett?

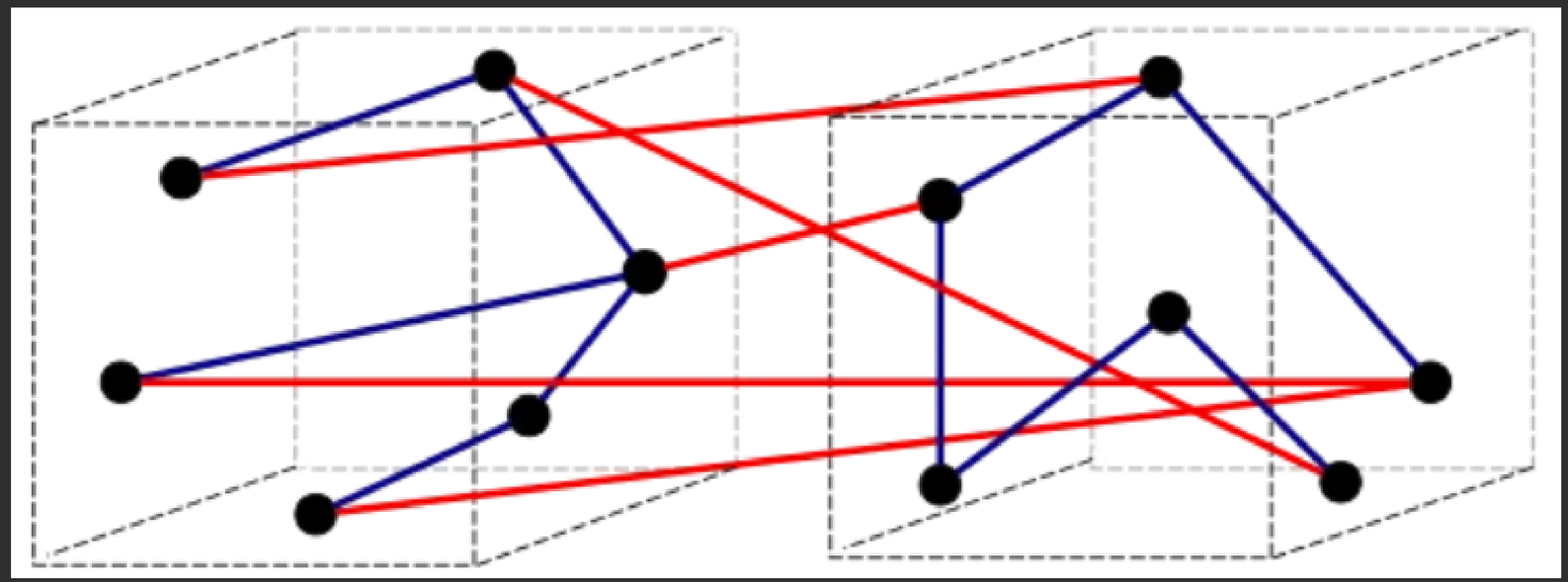
Høy kohesjon

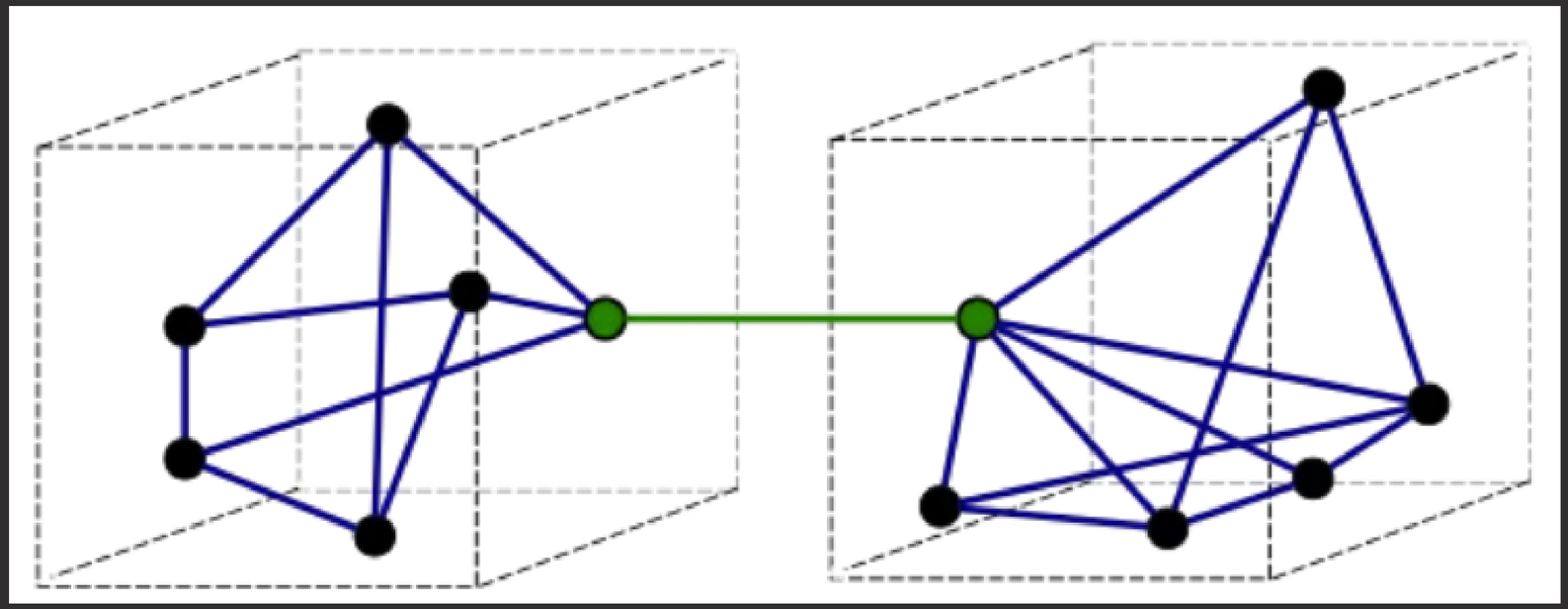
Vi skiller ulike ansvarsoppgaver i forskjellige klasser/moduler. (men de som er like holder vi sammen)

Lav kobling

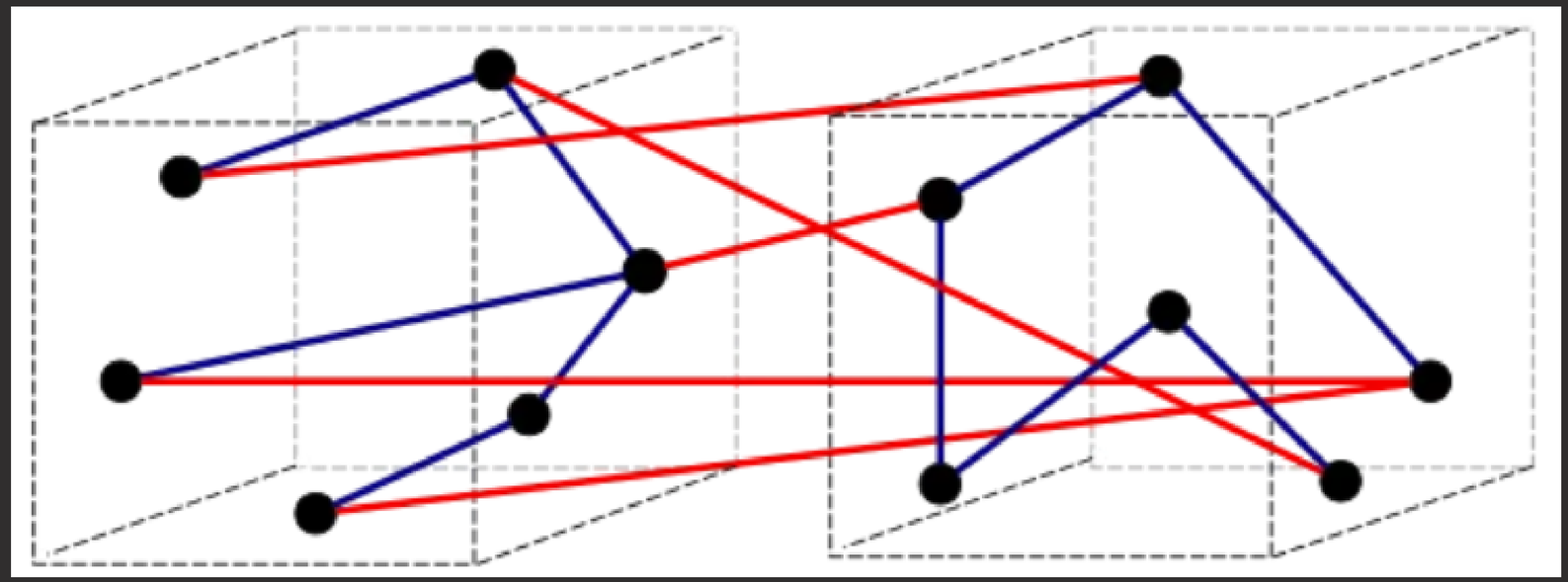
Vi lager få (oftest én) og enkle grensesnitt mellom klasser og moduler







Lav kobling,
høy hohesjon



høy kobling,
lav hohesjon

Hvis vi klarer å lage systemer med en god grad av lav kobling og høy kohesjon så får vi:

Systemer som er lette å teste, vedlikeholde og endre <3

MVVM + repository pattern

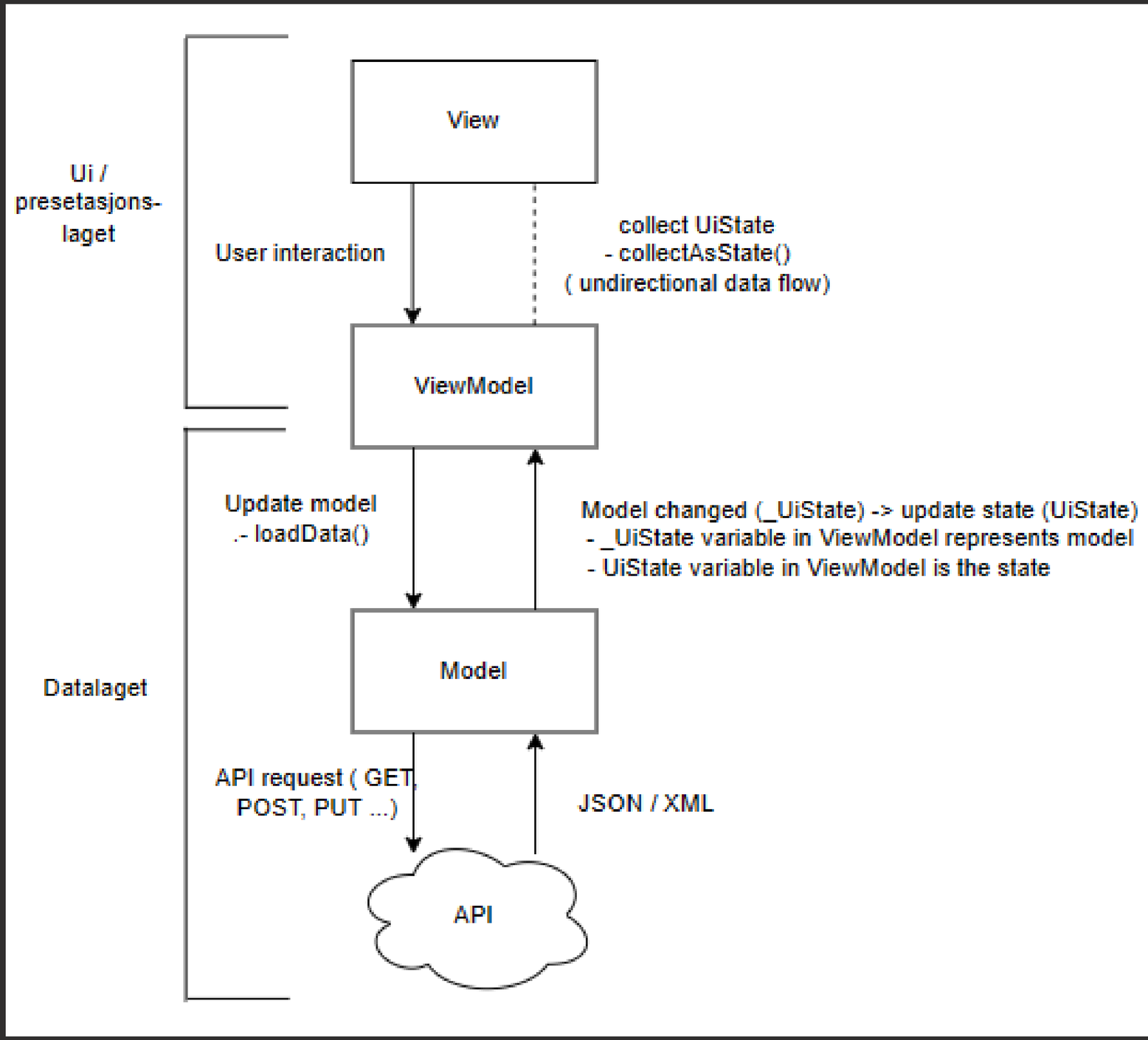
Gir en god grad av lav kobling og høy kohesjon

Lets take a closer look

UI-laget

View

- Ansvaret til **view** er å presentere state og murliggjøre brukerinterakjon.
- View observerer state I ViewModel.



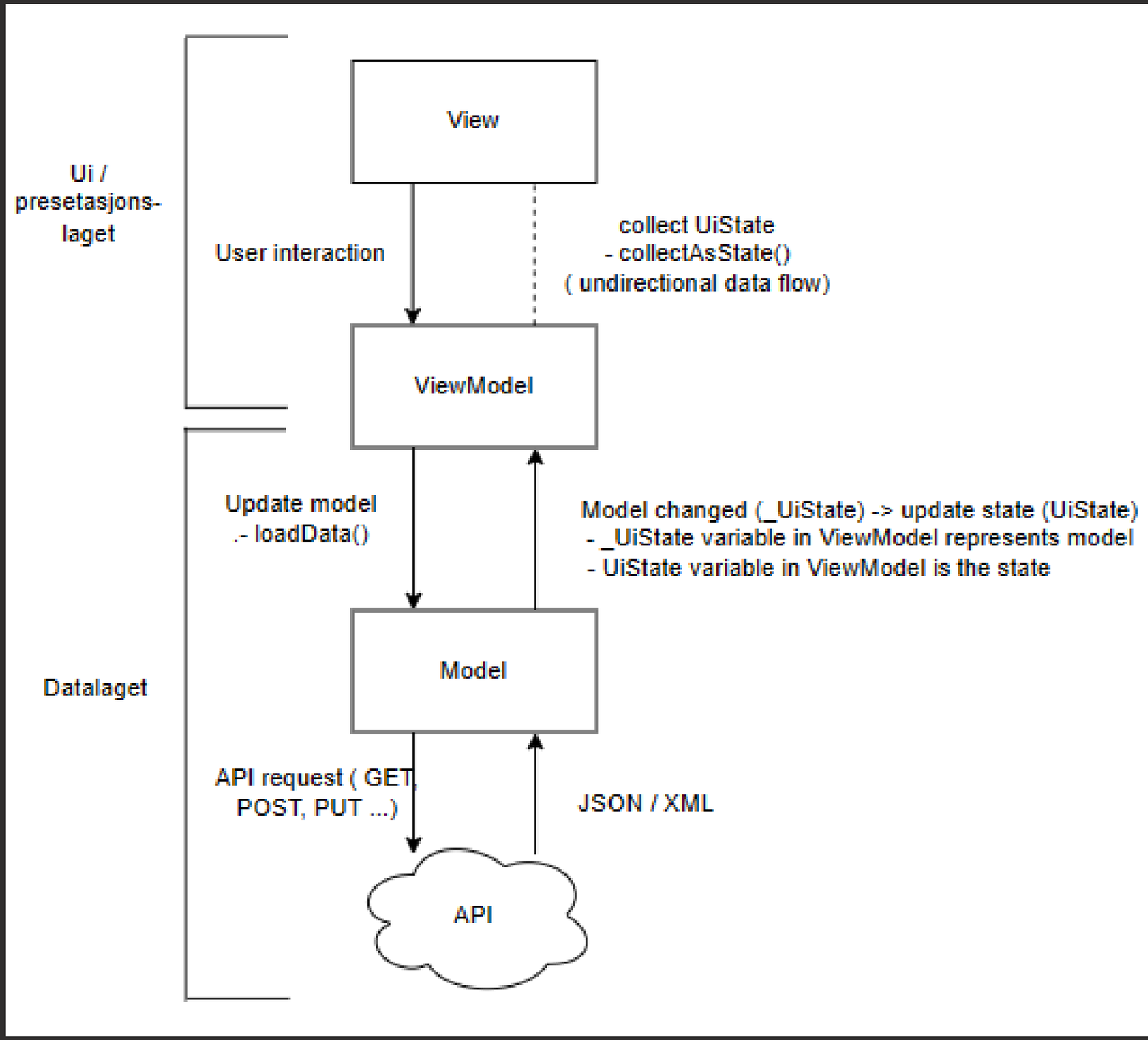
UI-laget

View

- Ansvaret til **view** er å presentere state og murliggjøre brukerinterakjon.
- View observerer state I ViewModel.

ViewModel

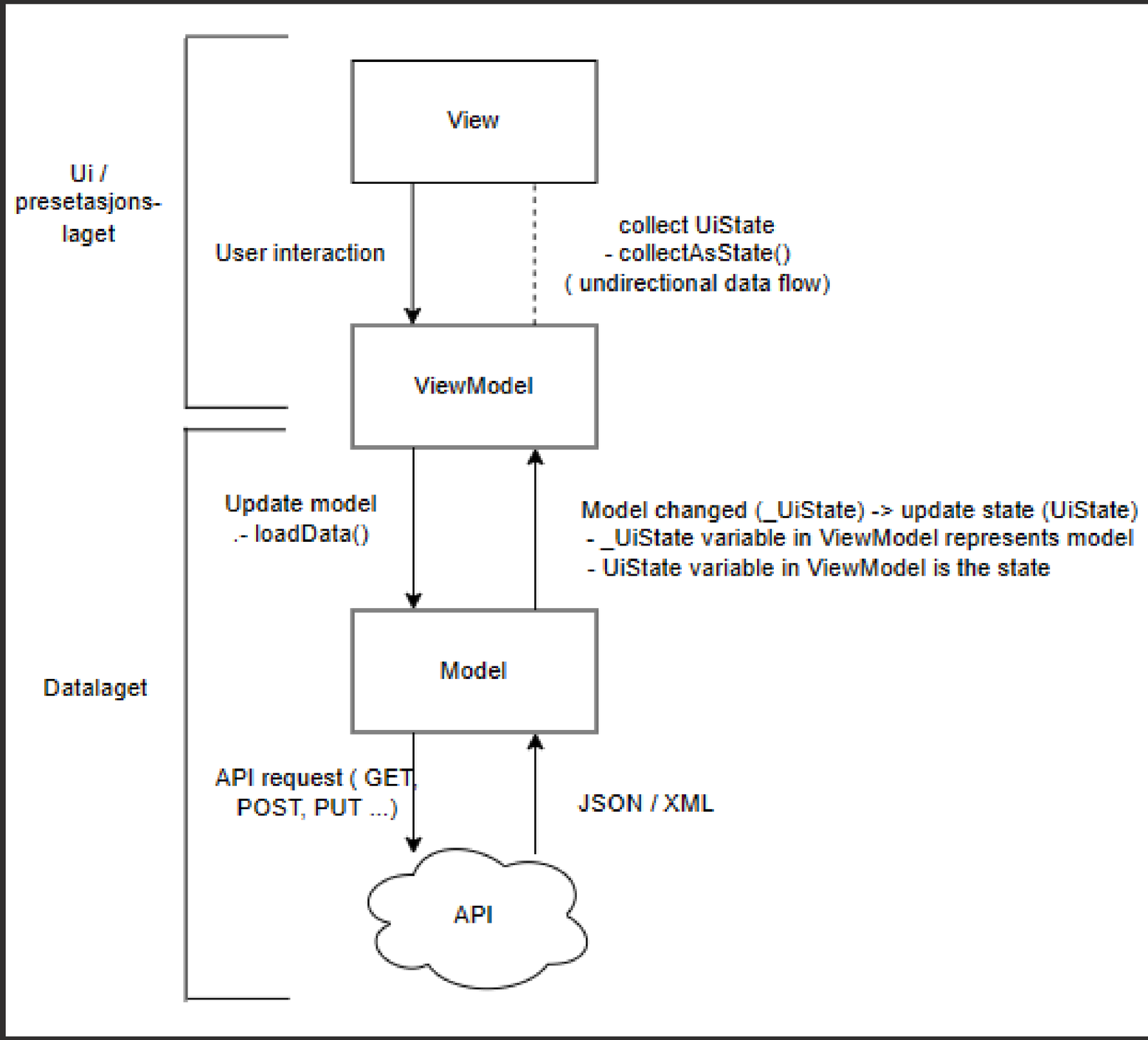
- **ViewModel** sitt ansvar er å presentere state til view, og også oppdatere state.
- Å oppdaterer state kan bla. Innebære å starte henting data, og også reagere på brukerinterkasjon som gjør at vi vil endre state.



Datalaget

Model

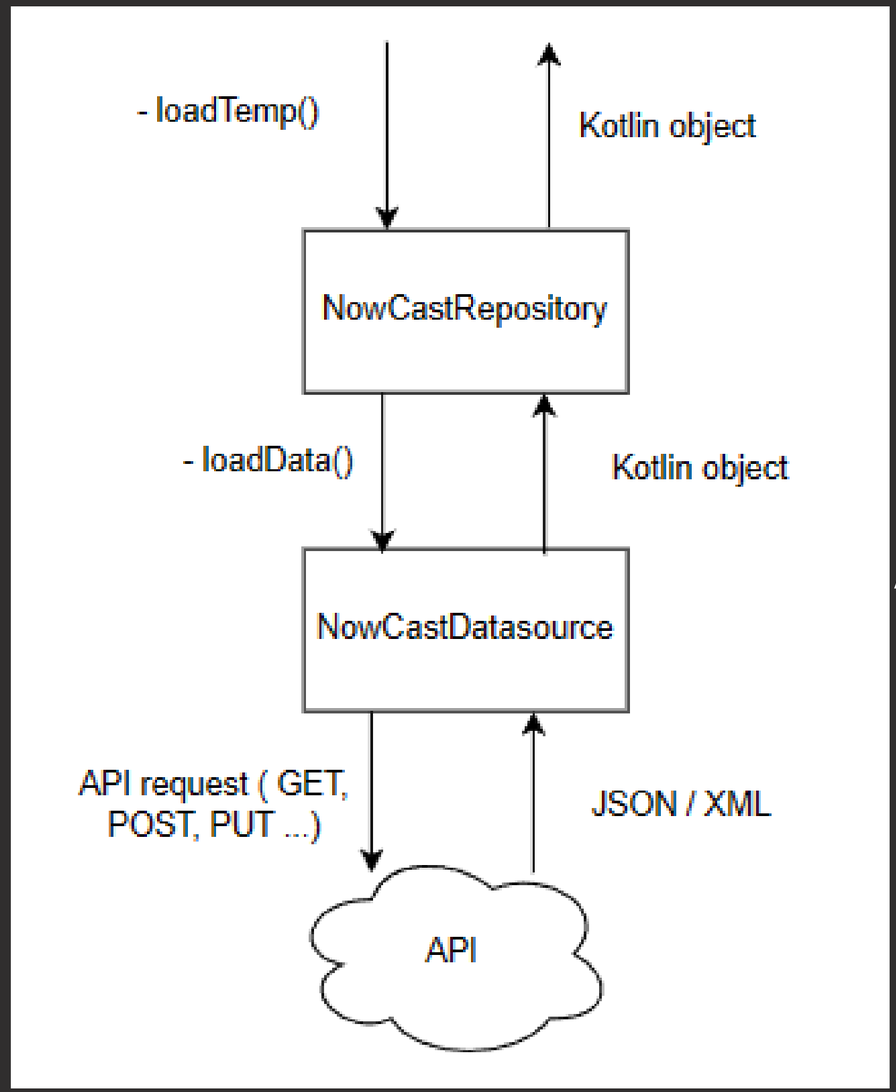
- **Model** sitt ansvar er å hente og behandle data, og presentere denne dataen til viewmodel.
- Typisk ønsker vi å dele opp Model I flere klasser med ulikt ansvar for høyere kohesjon :)
  - **Repositories:** en repo-klasse for hver ulike type data
  - **Datasoruces:** en source-klasse for hver type data (faktisk GET-kall)
  - **Domenelag:** (optional) egen klasse for behandling av vanlige usecases.
  - Behandling av data burde skje i klassen for den datatypen.



Datalaget

Model

- **Model** sitt ansvar er å hente og behandle data, og presentere denne dataen til viewmodel.
- Typisk ønsker vi å dele opp Model I flere klasser med ulikt ansvar for høyere kohesjon :)
  - **Repositories:** en repo-klasse for hver ulike type data.
  - **Datasoruces:** en source-klasse for hver type data (faktisk GET-kall)
  - **Domenelag:** (optional) egen klasse for behandling av vanlige usecases.
  - Behandling av data burde skje i klassen for den datatypen.



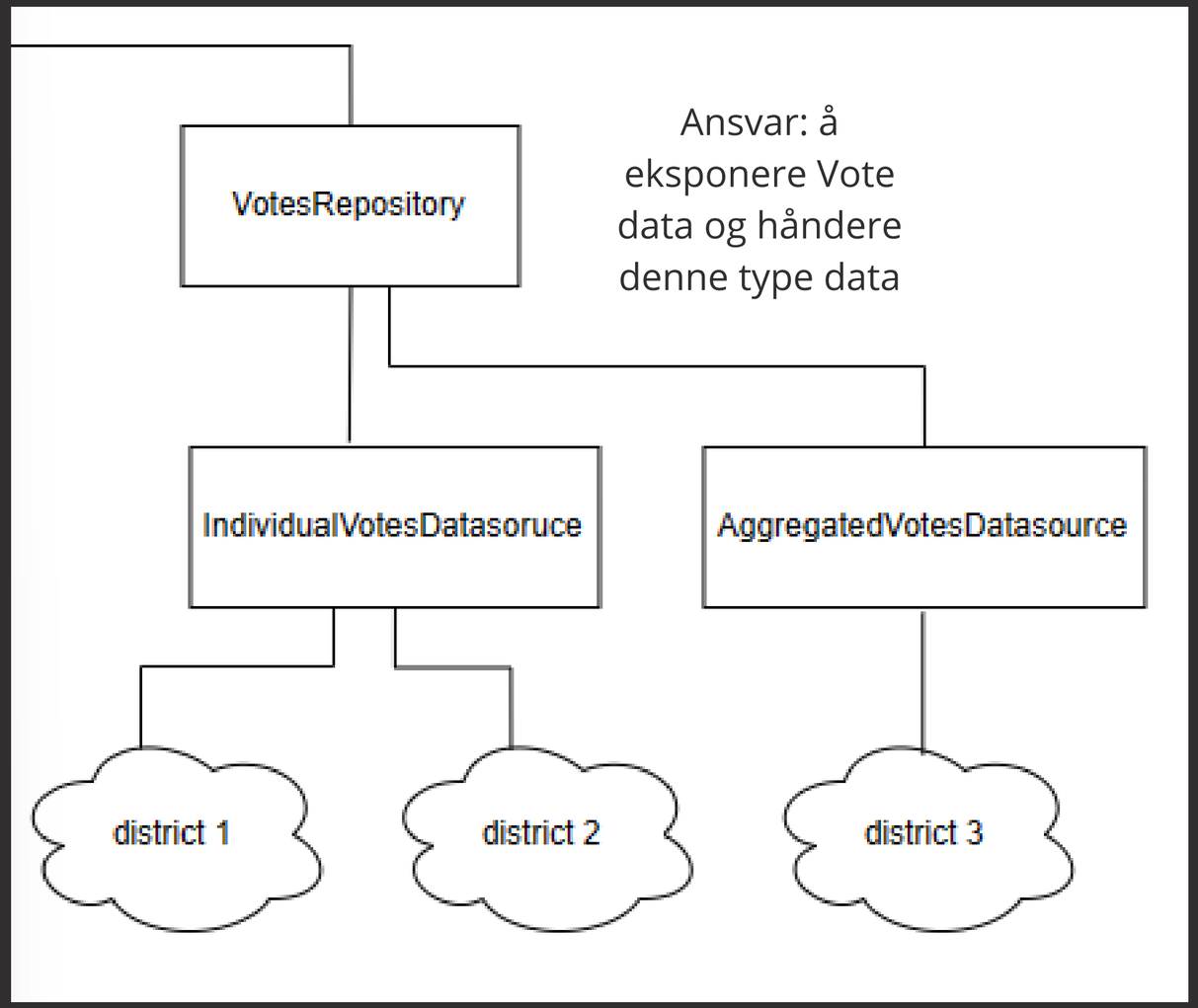
Ansvar: å eksponere og håndere en type data

Ansvar: Parse og hente en type data

Datalaget

Model

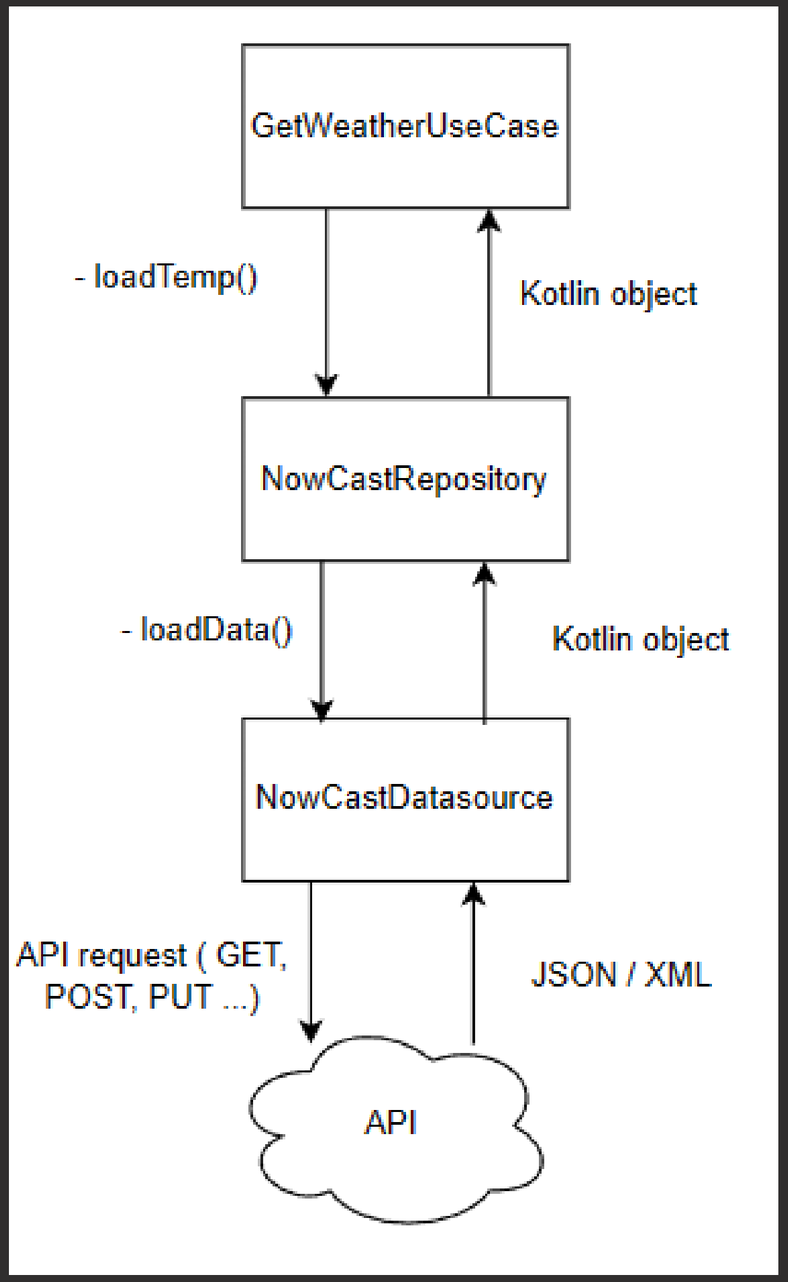
- **Model** sitt ansvar er å hente og behandle data, og presentere denne dataen til viewmodel.
- Typisk ønsker vi å dele opp Model I flere klasser med ulikt ansvar for høyere kohesjon :)
  - **Repositories:** en repo-klasse for hver ulike type data.
  - **Datasources:** en source-klasse for hver type data (faktisk GET-kall)
  - **Domenelag:** (optional) egen klasse for behandling av vanlige usecases.
  - Behandling av data burde skje i klassen for den datatypen.



Datalaget

Model

- **Model** sitt ansvar er å hente og behandle data, og presentere denne dataen til viewmodel.
- Typisk ønsker vi å dele opp Model I flere klasser med ulikt ansvar for høyere kohesjon :)
  - **Repositories:** en repo-klasse for hver ulike type data.
  - **Datasources:** en source-klasse for hver type data (faktisk GET-kall)
  - **Domenelag:** (optional) egen klasse for behandling av vanlige usecases.
  - Behandling av data burde skje i klassen for den datatypen.



Det er ingen satte “regler” for hvordan model skal settes opp
MEN: bruk anbefalingene til andorid -> fordel ansvar på en logisk måte

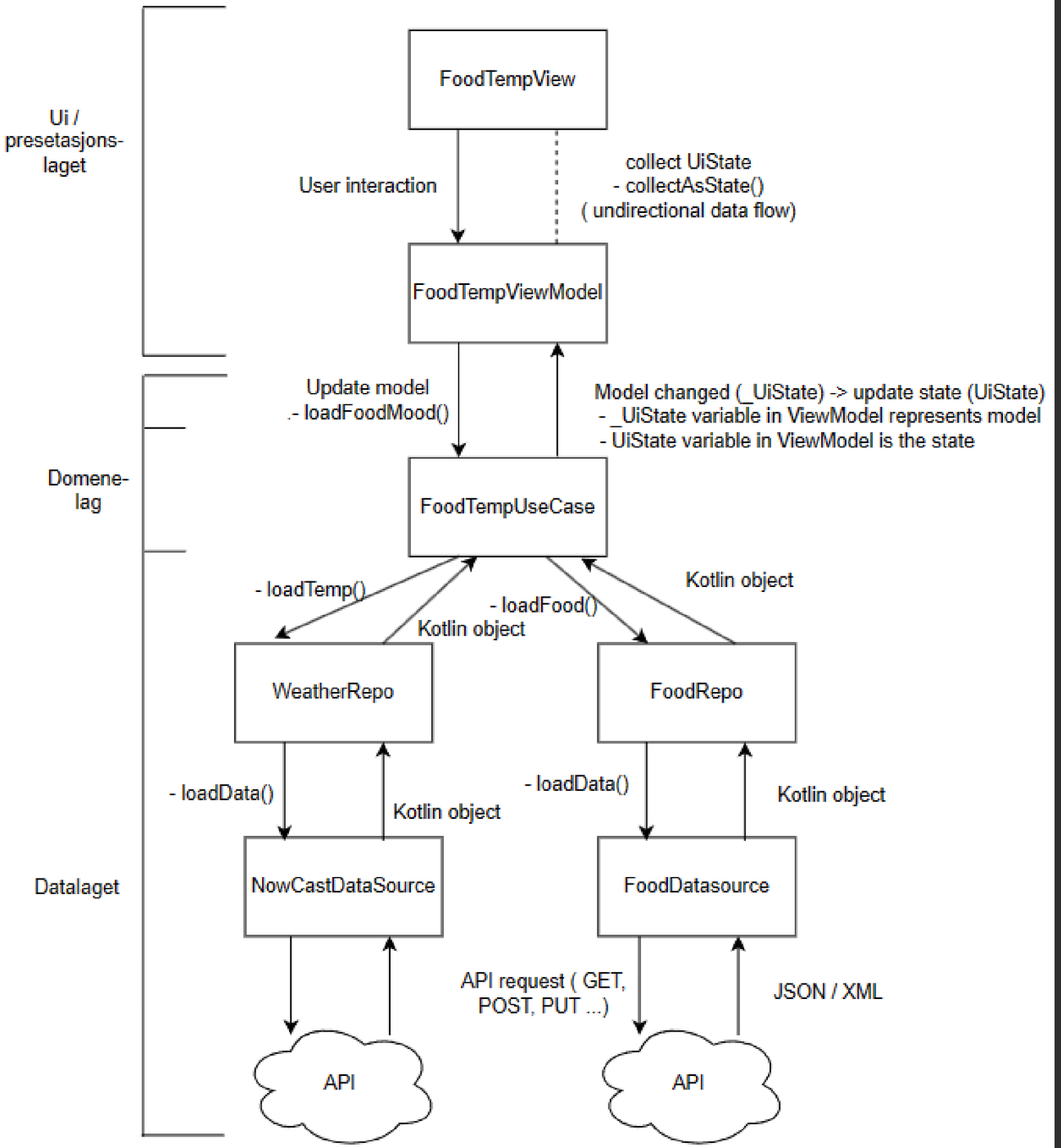
Food Temprature app

- Den skal basert på tempraturen akkurat nå velge en oppskrift du vil lage.
- Den skal hente data fra et Food api og NowCast
- Og vi vil følge androids guide to app architecture, for å lage appen. aka. det over :)

Food Temprature app

- En skjerm
  - FoodTempView
  - FoodTempViewModel
- To datakilder
  - NowCastDataSource og WeatherRepo
  - FoodDataSource of FoodRepo
- Finne mat basert på temperatur
  - Er et vanlig (eneste) use case for appen vår = FoodTempUseCase
  - Denne “beregningen” gjør vi også i denne klassen.

Vi tegner opp


Ui / presetasjons-laget
FoodTempView
User interaction
collect UiState
- collectAsState()
(undirectional data flow)
FoodTempViewModel
Update model
.- loadFoodMood()
Model changed (_UiState) -> update state (UiState)
- _UiState variable in ViewModel represents model
- UiState variable in ViewModel is the state
Domene-lag
FoodTempUseCase
- loadTemp()
Kotlin object
- loadFood()
Kotlin object
WeatherRepo
FoodRepo
- loadData()
Kotlin object
- loadData()
Kotlin object
Datalaget
NowCastDataSource
FoodDatasource
API request (GET, POST, PUT ...)
JSON / XML
API
API