Datahantering – backend

TDP028 Entreprenöriell programmering

LINKÖPINGS UNIVERSITET

Översikt

- Navbar-navigering
- Null safety
- Datahantering
  - Provider
- Datahämtning
  - Async
  - Firestore
- Modulär kod

Hot reload

- *Hot reload* injicerar kod som förändrats, ritar om appen
  - Bevarar app state
  - *Web app stödjer inte hot reload just nu (2024)*
- *Hot restart* startar om appen
  - Förlorar app state (t.ex. inloggning, kundvagn)

Navbar

Navbar

- Lätt att navigera mellan skärmarna
- Bra om användarna vill navigera ofta mellan en grupp skärmar
  - Behöver inte stega framåt och backa flera steg till önskad skärm
  - Skärmarnas tillstånd (innehåll) bevaras

Kombinera GoRoute med Navbar

```
final _parentKey = GlobalKey<NavigatorState>();
final _shellKey = GlobalKey<NavigatorState>();
  GoRoute
    parentNavigatorKey = _parentKey
  ShellRoute
    GoRoute // Bottom Navigation
      parentNavigatorKey = _shellKey
    GoRoute // Bottom Navigation
      parentNavigatorKey = _shellKey
  GoRoute // Doesn't need Bottom Navigation
    parentNavigatorKey = _parentKey
```

Navbar

```
return Scaffold(
    body: widget.child,
    bottomNavigationBar: BottomNavigationBar(
      onTap: () => changeTab(currentIndex),
      currentIndex: currentIndex,
      items: const [
        BottomNavigationBarItem(icon: Icon(Icons.home), label: 'Home'),
        BottomNavigationBarItem(icon: Icon(Icons.chat), label: 'Chat'),
```

Byt tabb med go_router

```
void changeTab(int index) {
   switch(value){
     case 0:
       context.go(’/');
       break;
     case 1:
       context.go('/chat');
       break;
}
   setState(() {
     currentIndex = value;
```

ShellRoute

- För att kombinera navbar med go_router navigering
- Läs mer:
  - https://medium.com/@ahm4d.bilal/using-gorouters-shellroute-in-flutter-for-nested-navigation-777a9a20642f

Null safety

Null safety

default

```
String name   // Non-nullable type. Cannot be 'null'
String? name // Nullable type. Can be 'null' or string
```

I första fallet kommer kompilatorn kunna hjälpa till att upptäcka kod där name riskerar att bli null

Läs mer:

- https://dart.dev/null-safety#using-variables-and-expressions
- https://dart.dev/null-safety/understanding-null-safety

? = null-aware operator

```
String? notAString = null;
print(notAString?.length);
```

- Var beredd att var kan vara null
  - Hoppa över ’length’ och returnera null (hade annars fått NullPointer Exception och krasch)

?? (dubbelt frågetecken)

Om null …

Ex1:

String a = b ?? 'hello';

Ex2:

Text(items.getData ?? 'No data')

! (utropstecken)

- Aväand ! för att konvertera från nullable to non-nullable datatyp

```
String? iKnowItsNotNull = ””;
print(iKnowItsNotNull!.length);
```

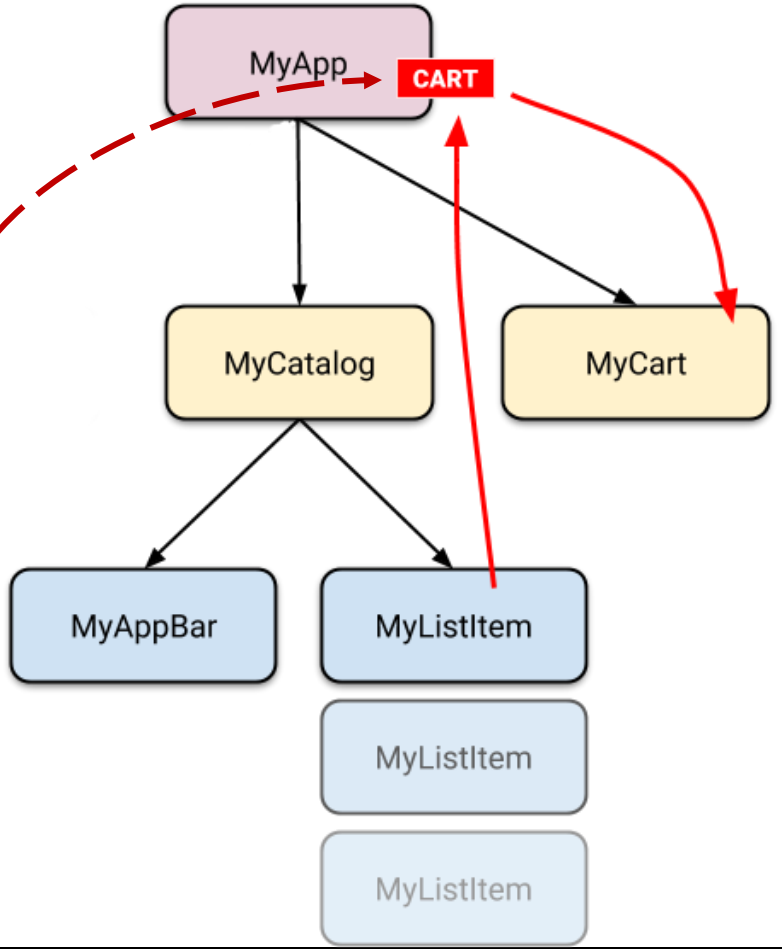
- Läs mer:
  - https://dart.dev/null-safety/understanding-null-safety#non-null-assertion-operator

State management

Tillståndshantering

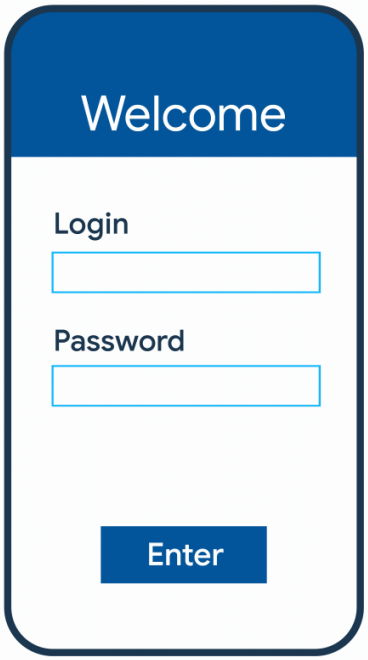
Fall när tillstånd bör lagras på app-nivå

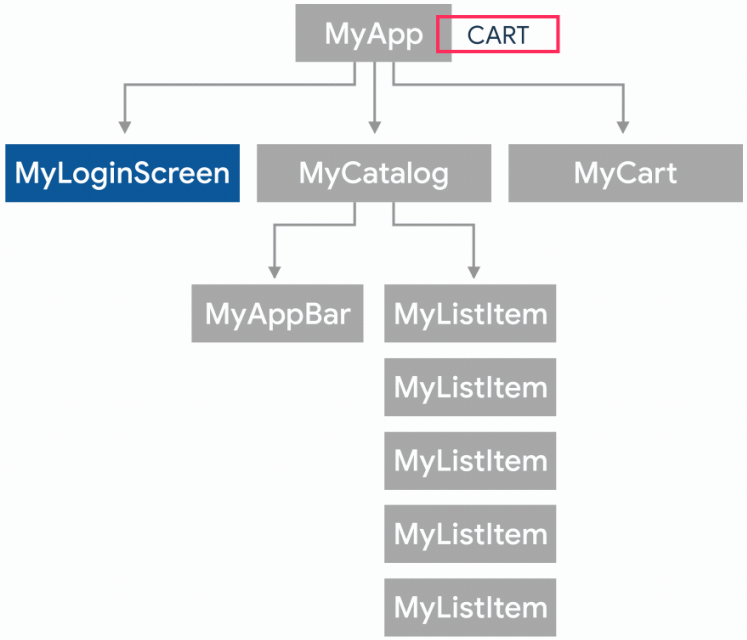
- När en widget behöver påverka annan del i trädet
- Ex.
  1. Anv. klickar på ListItem
  2. Cart uppdateras
  3. Triggar omritning av MyCart



Ett fel och ett rätt sätt att hantera state

- Behöver meddela topp-nivå (app-tillstånd) och sedan rita om de delar av skärmen som berörs





Fel sätt: Skicka ner callback i widget-trädet

```
class MyApp {
  void myTapCallback(Item item) {
    print('User tapped on $item');
  }

  @override
  Widget build(BuildContext context) {
    return SomeWidget(
        MyClickabelWidget(myTapCallback),
    );
  }
}
```

Fel sätt: Callback

- Lagra allt i AppState och registrera callbacks från Widgets i appen
  - Callback = ”Kalla på min funktion vid uppdatering, tack!”
- Stor nackdel
  - appState-instansen måste skickas neråt för att widgets ska kunna regga sina callbacks

Fel: MyClickableWidget anropar callback

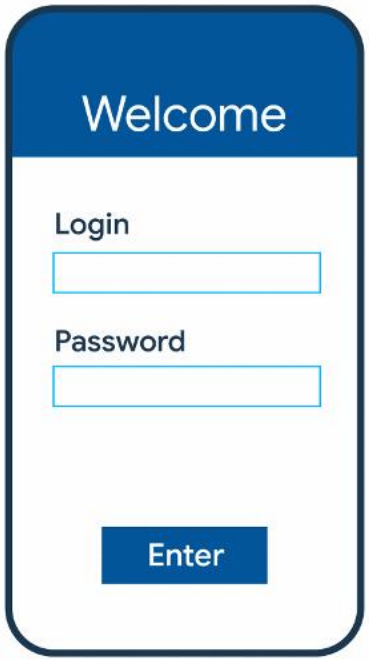
```
class MyClickableWidget extends StatelessWidget {

final int index;
final void Function(int) callBack;

const MyClickableWidget({super.key, required this.callBack});

@override
  Widget build(BuildContext context) {
    return SomeWidget(
          title: Text(...),
          onClick: () {callBack(index);}
```

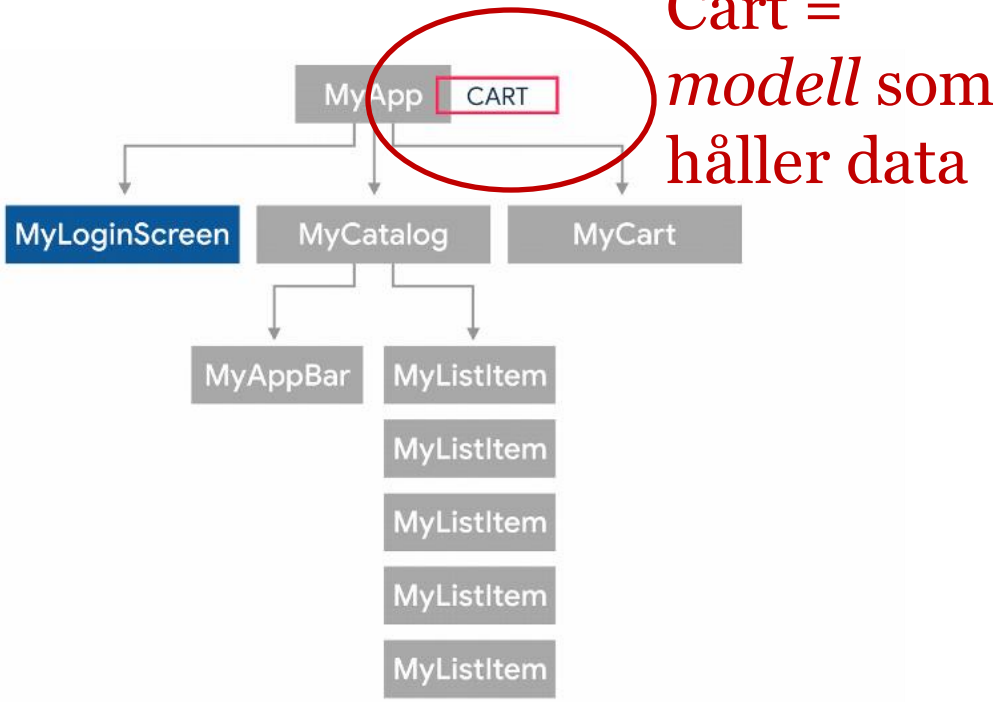
Fel: Rörigt att skicka in appState i widget-trädet

- Måste ofta skickas vidare i många led
- Rörigt att samla all information på ett ställe
  - Många tillståndsvariabler att hålla reda på
  - Många olika uppdateringar
  - Från många olika håll (vem har förstört …?)

Rätt sätt: Använd Provider

- Provider hjälper hantera *modellen* som håller data





Cart = *modell* som håller data

Provider biblioteket

- Skriv i terminal:

  > flutter pub add provider

Alt. lägg till ny dependency i pubspec.yaml:

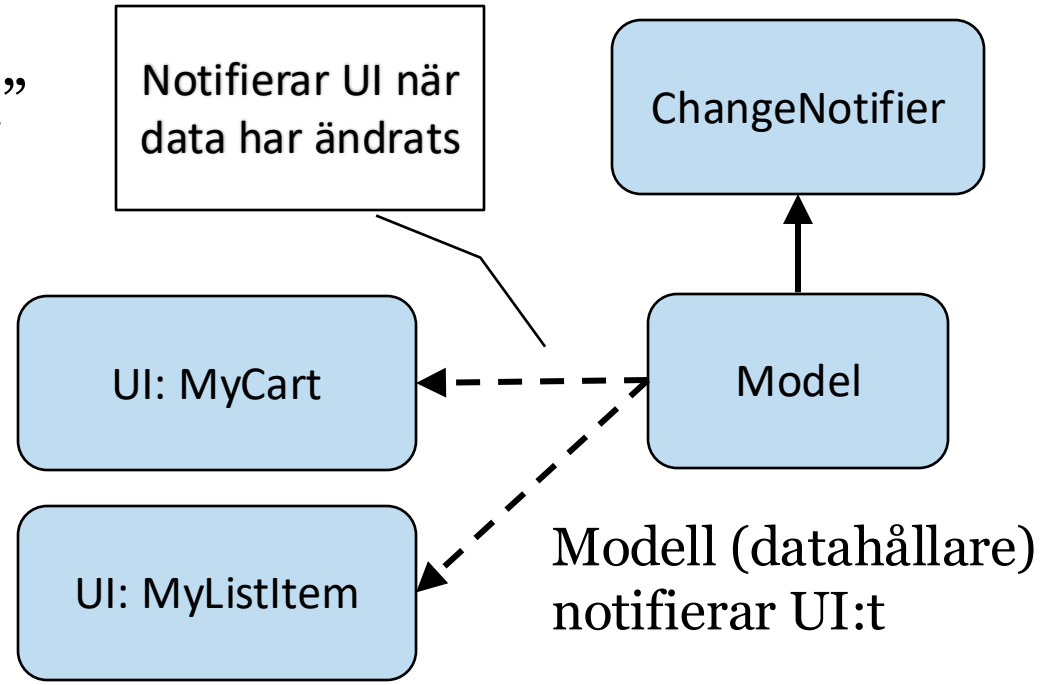
```
dependencies:
  provider: ^6.1.2    # ta senaste versionen
```

- Importera

```
import 'package:provider/provider.dart';
```

Modellen kan ärva ChangeNotifier

- Kommer notifiera de som har ”anmält” intresse
- Provider hjälper till med kontakt-förmedlingen
  - Fördel: Slipper explicita beroenden i koden

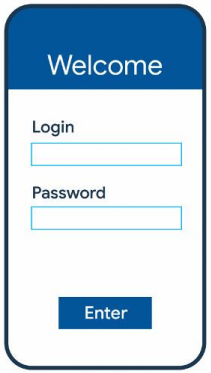


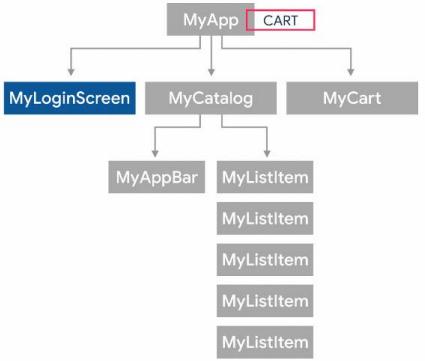
notifyListeners()

```
class CartModel extends ChangeNotifier {
  final List<String> _boughtItems = [];

  List<String> get items => _boughtItems;

  void buy(String merchName) {
    _boughtItems.add(merchName);
    notifyListeners();
  }
```

Wrap:a MyApp i en ChangeNotifierProvider

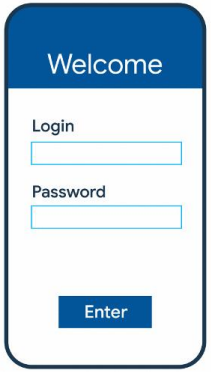
```
// ska ligga ovanför routing i widget-trädet
void main() {
  runApp(
    ChangeNotifierProvider(
      // hanterar instansen av vår modell
      create: (context) => CartModel(),
      child: const MyApp(),
    ),
  );
}
```

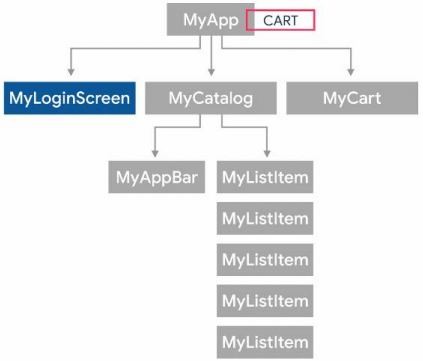




a. Vill ha uppdateringar: Consumer

```
return Consumer<CartModel>(
  builder: (context, cart, child) {
    return Text('Total price: ${cart.totalPrice}');
  },
);
```





b. Vill ha uppdateringar: watch

```
@override
Widget build(BuildContext context) {
  // want to be notified if Cart has changed
  CartModel cart = context.watch<CartModel>();

  return Scaffold(
     body: ListView.builder(
        itemCount: cart.items.length,
        itemBuilder: (context, index) =>
           ListTile(title: Text(cart.items[index])),
```

c. Vill ha passiv tillgång

```
// just want access, no notifications needed
CartModel cart =
    Provider.of<CartModel>(context, listen: false);

return Card(
  child: ListTile(
    title: Text(itemName),
    trailing: const Text('buy'),
    onTap: () => cart.add(itemName),
```

Använd hellre Consumer:
- kan läggas längre ner i trädet
- mindre risk för att klanta med context

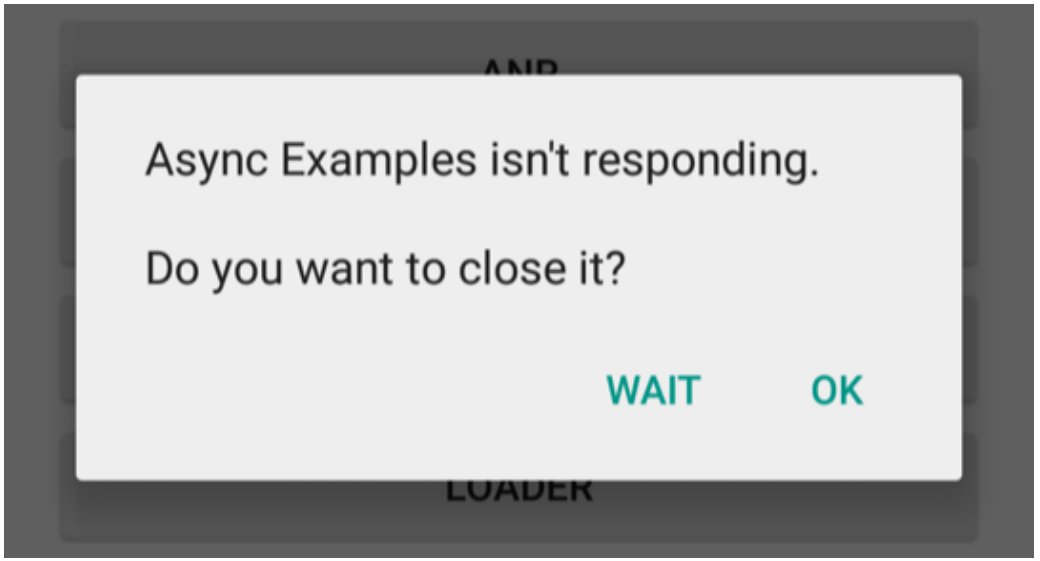
Provider

- Intelligent hantering av tillstånd hos data
  - Ger ’heads up’ när data har förändrats
- Läs mer:
  - https://docs.flutter.dev/data-and-backend/state-mgmt/simple
- Kodexempel:
  - https://github.com/flutter/samples/blob/main/provider_shopper/lib/main.dart

Async

Android: Application not responding (ANR)

- Appen ska inte vänta på långa nätverksanrop



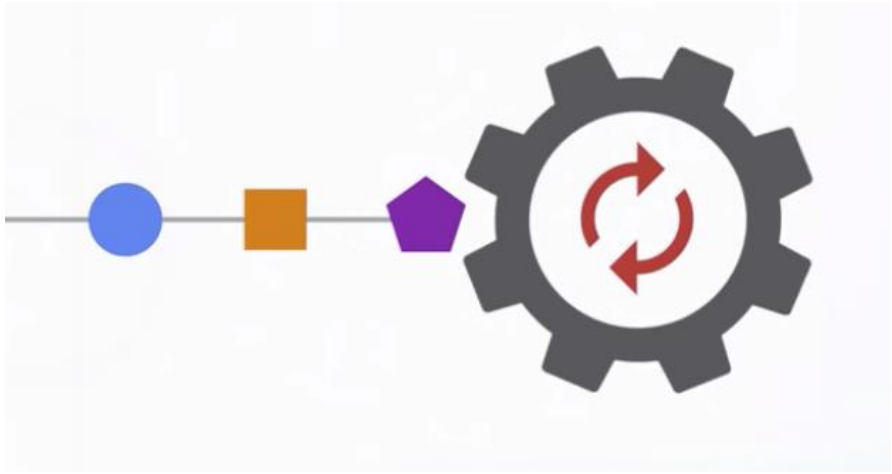
Exempel från Android

Trådar och virtuella processer (isolates)

- Trådar (threads)
  - Varje tråd fungerar som en egen parallell process
  - Flera trådar kan köras på en CPU-kärna
- Flutter har ’isolates’ med eget minne
  - Isolates kommunicerer via meddelanden

Event loop

- UI-tråden, ellerUI-isolate, kör main event-loop
  - Får inte blockeras av tung beräkning



- Läs mer:
  - https://www.youtube.com/watch?v=vl_AaCgudcY

Olika sätt att undvika blockering

1. Använd asynkrona widgets
2. Använd asynkrona bibliotek
3. Starta ny isolate manuellt

1. FutureBuilder

```
child: FutureBuilder<String>(
  future: _calculation, // a previously-obtained Future
  builder: (BuildContext context, AsyncSnapshot<String> snapshot) {
    List<Widget> children;
    if (snapshot.hasData) {
      children = <Widget>[
        const Icon(Icons.check_circle_outline),
        Text('Result: ${snapshot.data}'),
```

Pågående tung process

Builder som blir anropad med delresultat

1. FutureBuilder

- Tre fall att ta hand om, dvs. visa olika saker på skärmen:
  1. snapshot.hasData
  2. snapshot.hasError
  3. annars: vänteläge (inte klar ännu)
- Läs mer:
  - https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html

Future

- Påbörjat resultat från en process som fortfarande kör
- Ska typas
  - Future<String>
  - Kommer att bli en sträng när processen är färdig
- Future gör att event-loopen kan fortsätta
  - Man har ju redan ett slags svar i handen…
  - På de ställen i koden där man verkligen behöver svaret, måste man vänta in processen

Vänta på en långsam process

- async – await

```
Future<String> createOrderMessage() async {
  var order = await fetchUserOrder();
  return 'Your order is: $order';
}
```

Async – await

- Main UI-isolate blockeras inte, eftersom funktionen returnerar en future under tiden
  - return-satsen kommer dock att dröja
- Läs mer
  - https://dart.dev/libraries/async/async-await

2. Asynkrona bibliotek

```
var client = HttpClient();
try {
  HttpClientRequest request = await client.get('localhost', 80, '/file.txt');
  // Optionally set up headers...
  // Optionally write to the request object...
  HttpClientResponse response = await request.close();
  // Process the response
  final stringData = await response.transform(utf8.decoder).join();
  print(stringData);
} finally {
  client.close();
```

Felhantering

```
throw FormatException('Expected at least 1 section’);


try {
  breedMoreLlamas();
} catch (e) {
  print('Error: $e’);    // Handle the exception first
} finally {
  cleanLlamaStalls(); // Then clean up
}
```

Felhantering

- Läs mer:
  - https://dart.dev/language/error-handling

3. Manuella sättet

- Starta ny asynkron isolate
  - Kör det tunga jobbet
  - Den kommer att anropar din callback när klart
- Läs mer på:
  - https://docs.flutter.dev/perf/isolates

Backend – Firestore

Bra startpunkt

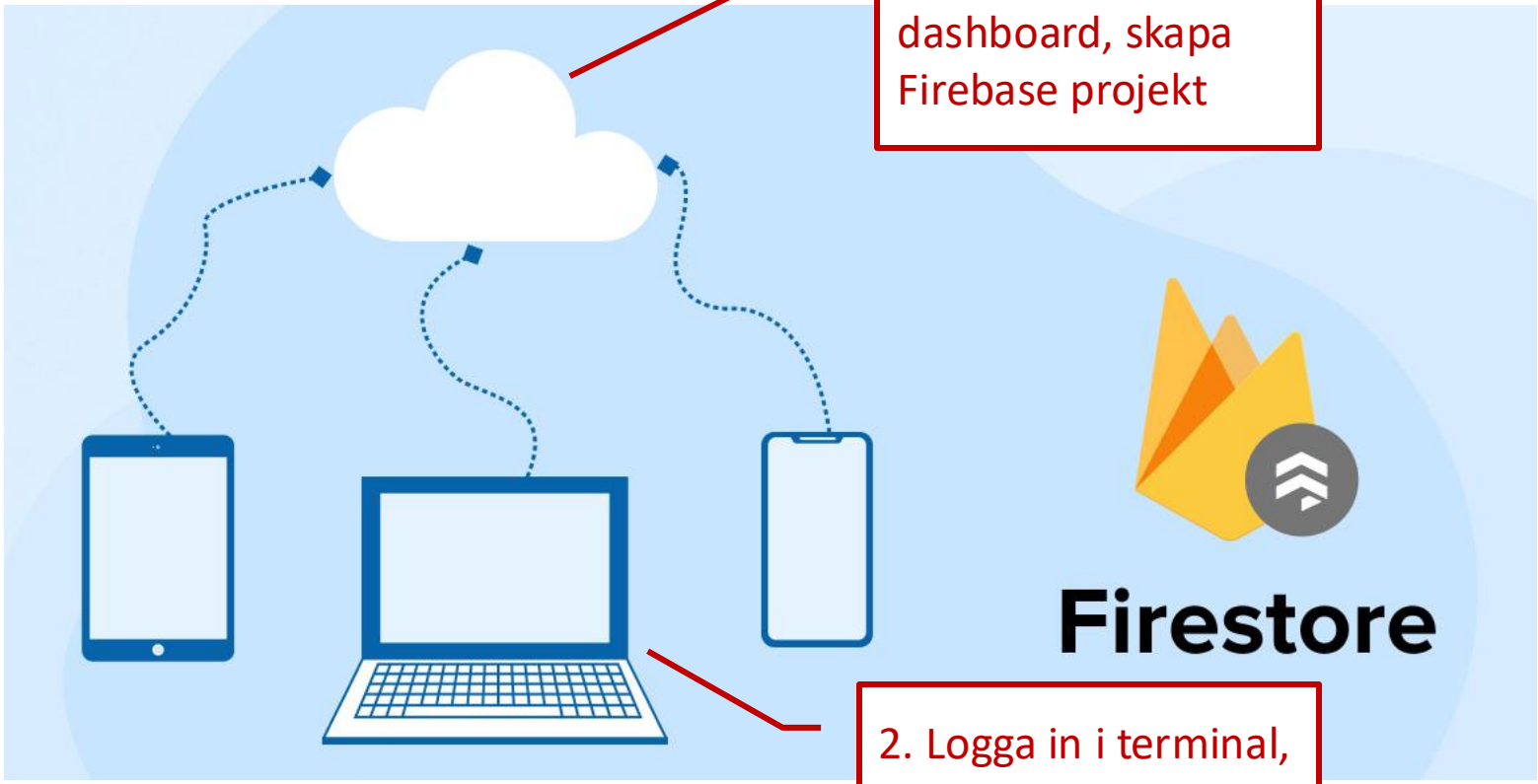
- Sätt upp Firebase:
  - https://firebase.google.com/docs/flutter/setup?platform=ios
- Quickstart för Firestore:
  - https://firebase.google.com/docs/firestore/quickstart

Cloud Firestore vs Realtime database

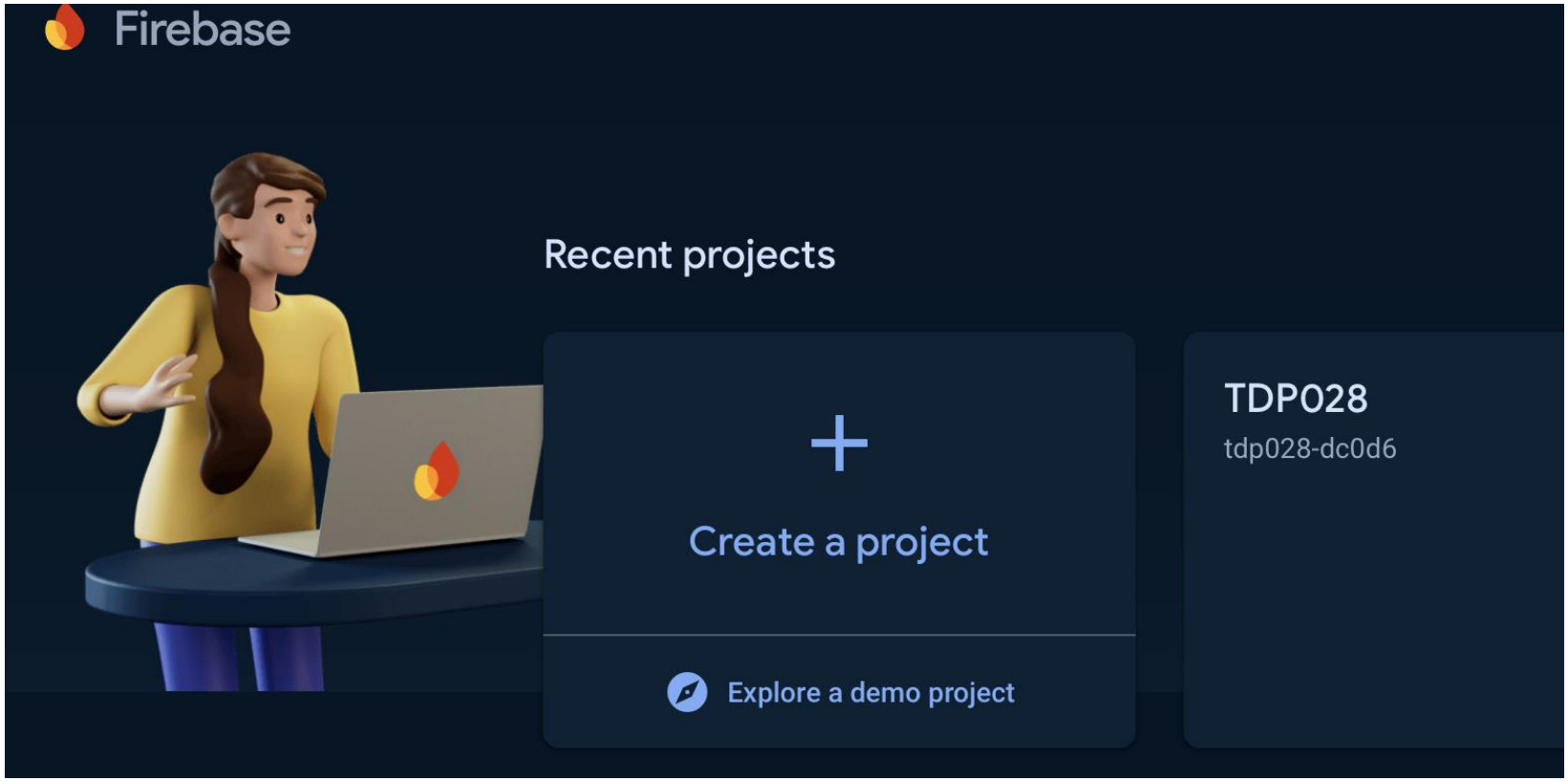
- Firestore
  - Dokument, ger mer struktur
- Realtime database
  - json, lite enklare koncept

- Läs mer:
  - https://blog.flutter.wtf/firestore-gui-tools/

Behöver sättas upp från två håll

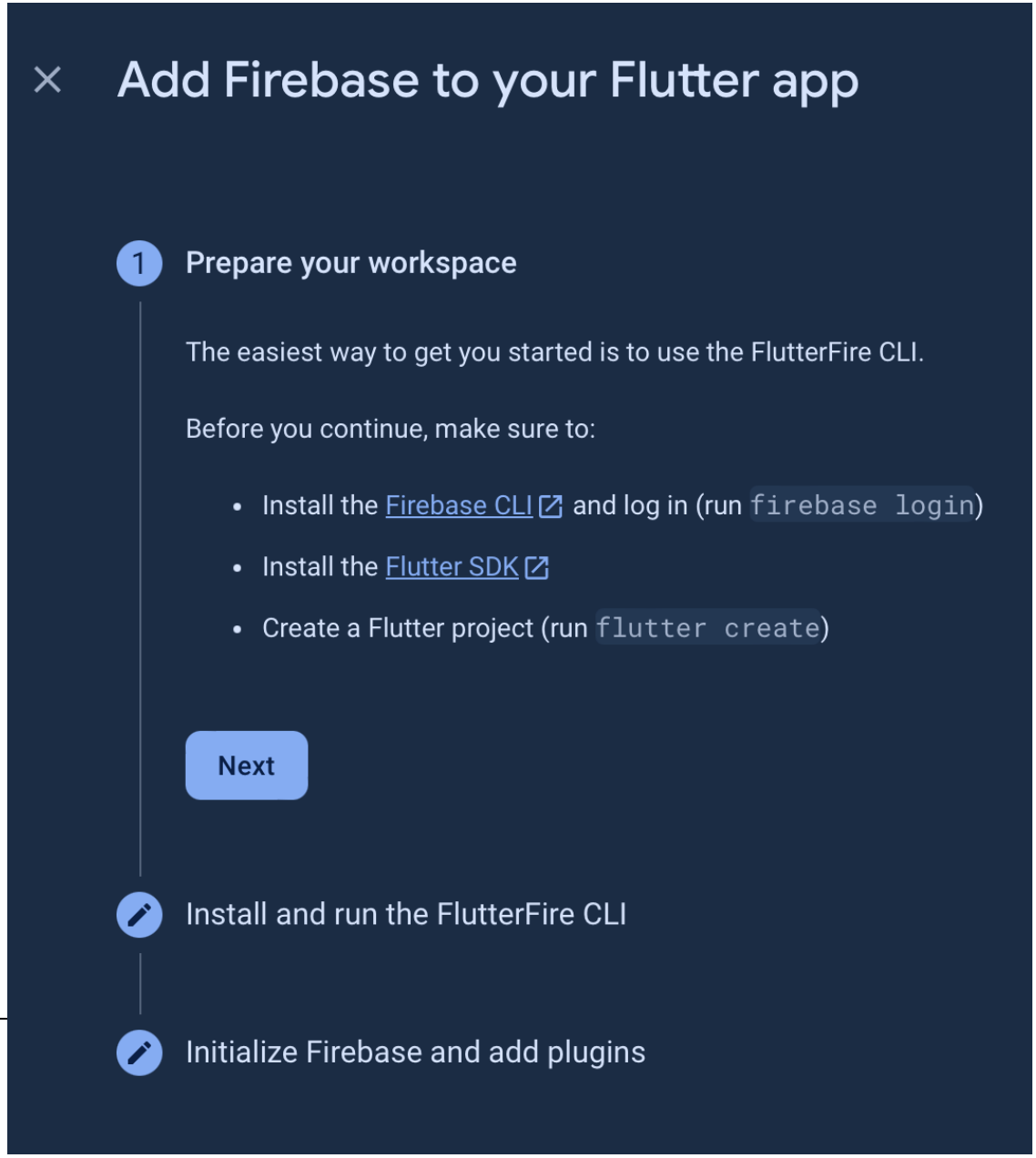


1. Skapa Firestore projekt (på webben)



https://console.firebase.google.com/...

Sätta upp Firebase

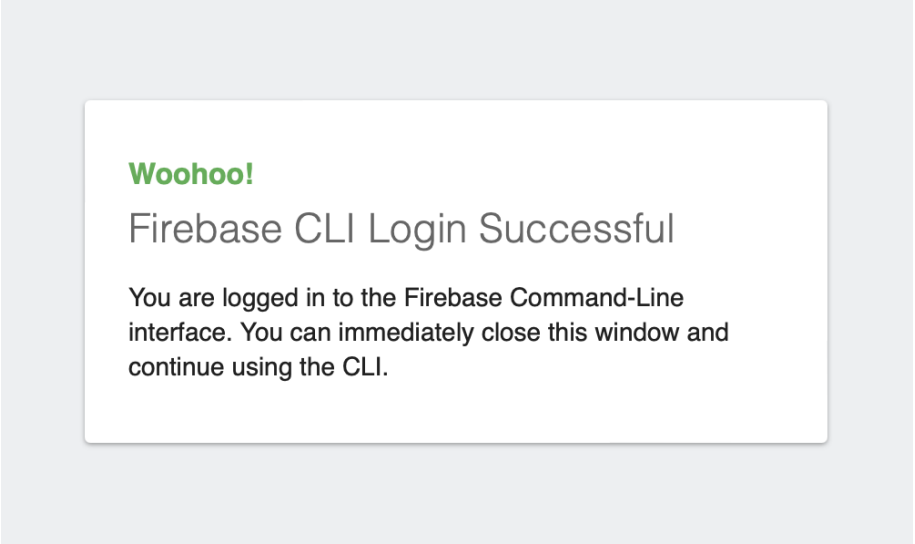


2. Från utvecklarhåll

- Öppna terminal
- Installera Firebase CLI (Command Line Interface)

  > curl -sL https://firebase.tools | bash

  > firebase login



3. Från utvecklarhåll (forts)

- Nu är moln och utvecklarmiljö ihopkopplade
- Installera FlutterFire CLI

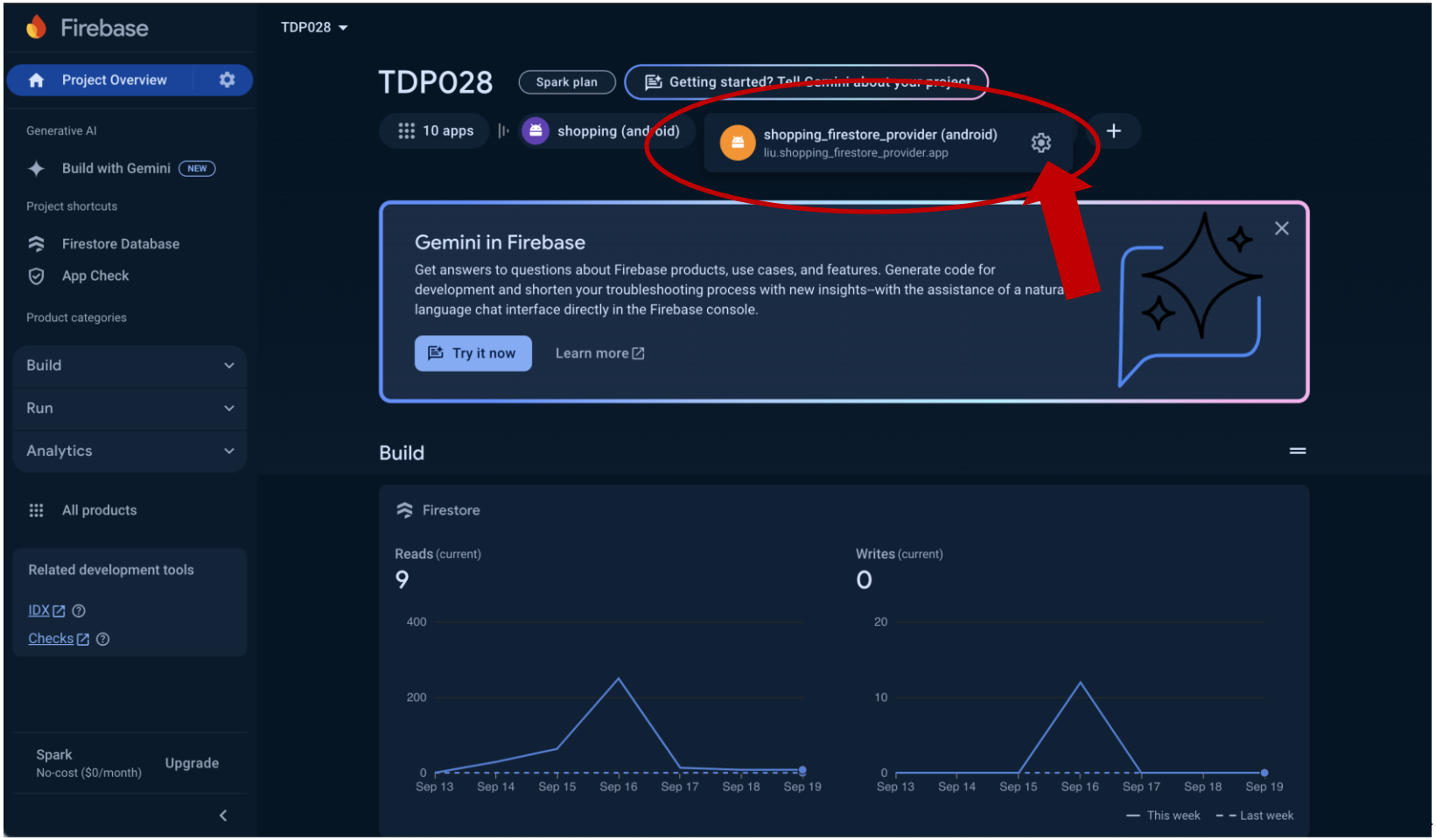
  > dart pub global activate flutterfire_cli

  > export PATH="$PATH":"$HOME/.pub-cache/bin"

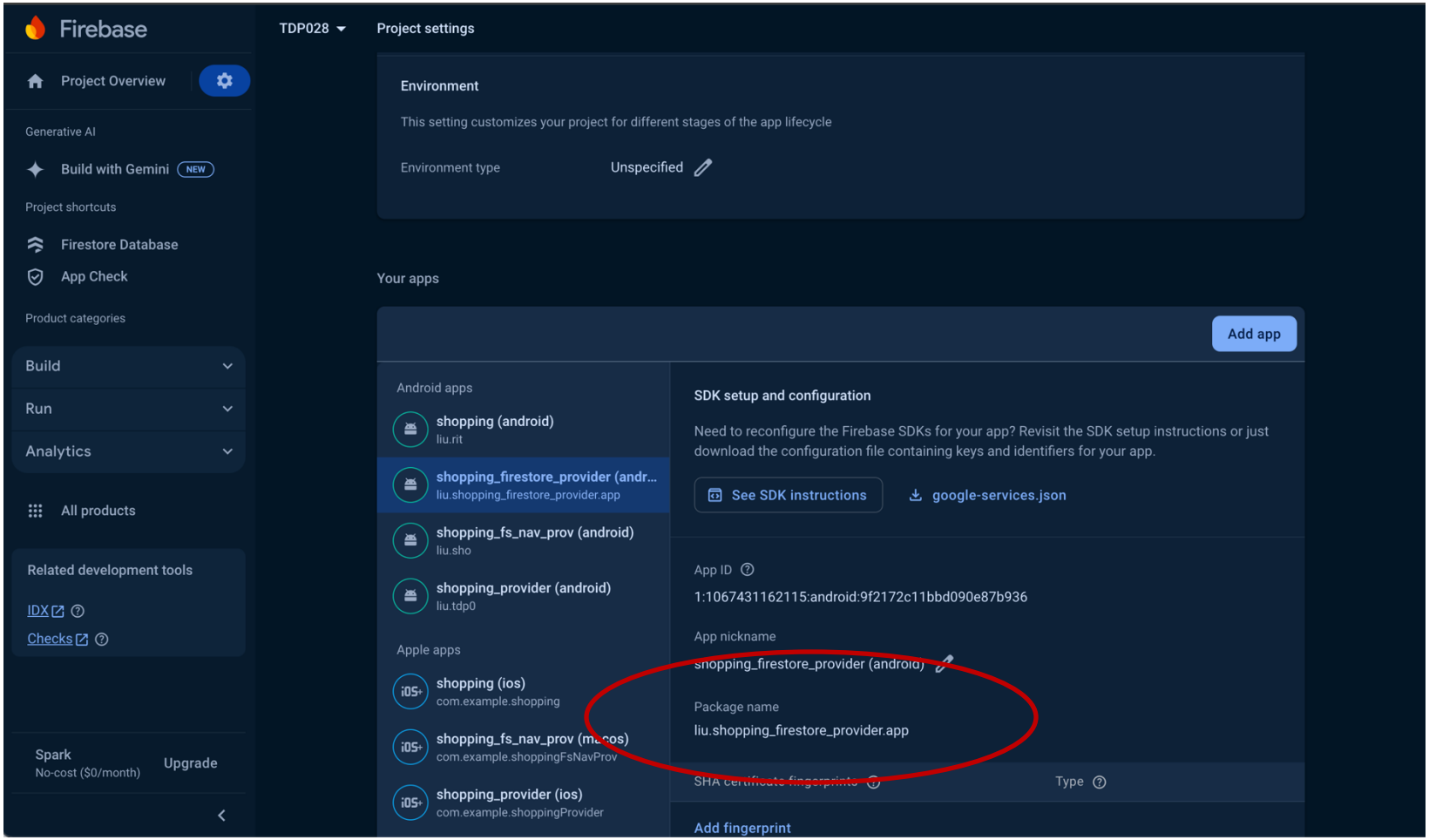
  > flutterfire configure --project=*tdp028-dc0d6*

Projektet som skapades i Firestore

https://console.firebase.google.com/...



https://console.firebase.google.com/...



> flutterfire configure (forts)

- Which platforms should your configuration support (use arrow keys & space to select)?
  - Alla alternativ är valda från början
    - Mellanslag för att ’toggle:a’
  - Tryck bara Enter för att välja alla plattformar

Från utvecklarhåll (forts)

Installera Firebase-core

> flutter pub add firebase_core

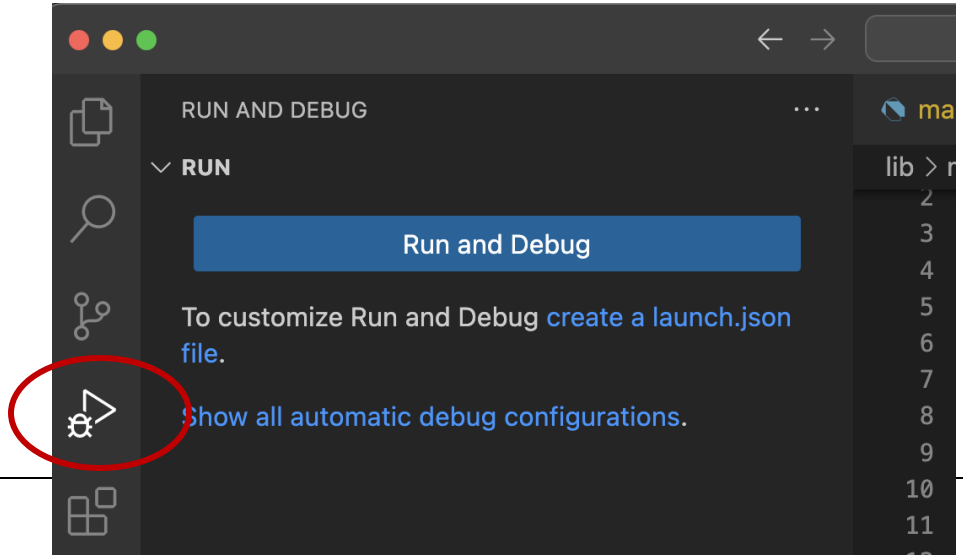
Firestore

- Sätt upp Firestore plugin till Flutter

> flutter pub add cloud_firestore

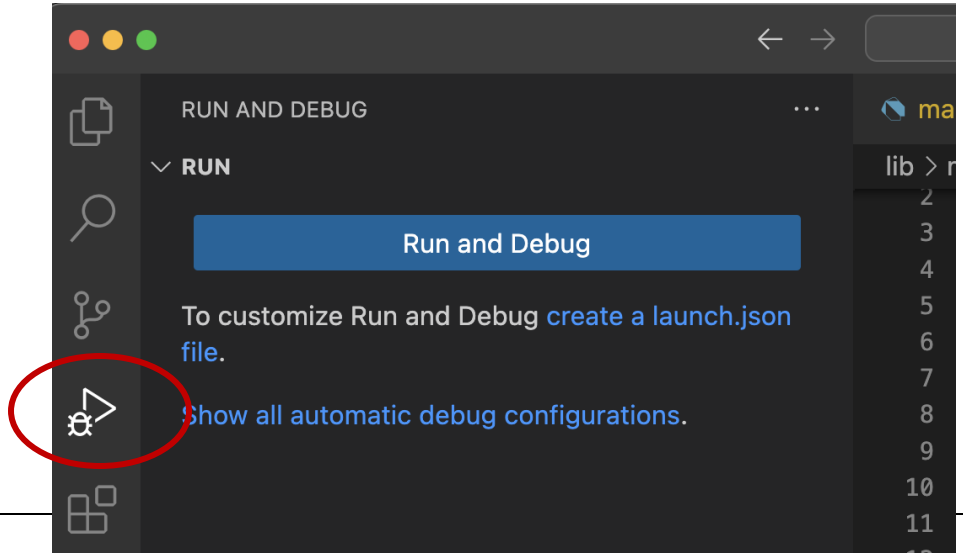
- Bygg om (kompilera om) appen:

> Kör appen



Kör appen

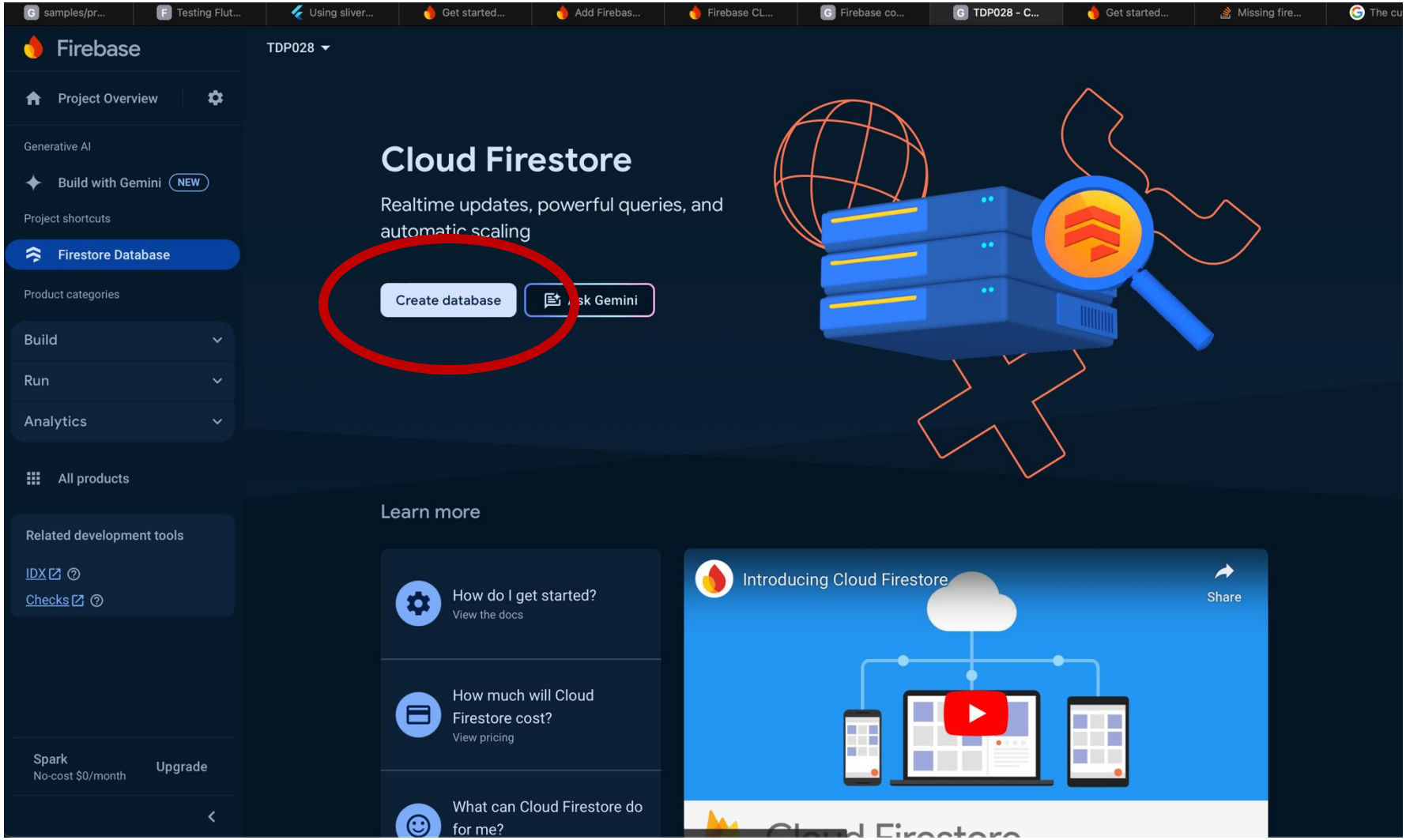
- OBS! Kompilering till iOS och MacOS tar en evighet! För att snabba upp lite:
  - https://firebase.google.com/docs/firestore/quickstart#set_up_your_development_environment



Importera och initialisera *Firebase*

```
import 'package:firebase_core/firebase_core.dart';
import 'package:[your_package]/firebase_options.dart';

Future<void> initializeFirebase() async {
    await Firebase.initializeApp(
      options: DefaultFirebaseOptions.currentPlatform,
    );
}
```

Dags att skapa databasen



Importera och initialisera *Firestore*

- Firestore är själva databasen (Firebase är en samling molntjänster)

```
import 'package:cloud_firestore/cloud_firestore.dart';

final db = FirebaseFirestore.instance;
```

Referens till en samling eller dokument

```
CollectionReference cartCollRef = db.collection('cart');

DocumentReference cartDocRef =
    db.collection('cart').document(...);
```

- Mer om hur Firestore är organiserad
  - https://www.youtube.com/watch?v=v_hR4K4auoQ

Läs vidare

- https://firebase.google.com/docs/firestore/query-data/listen

Möjliga småproblem

Vid problem

- Kör flutter doctor
  - Kolla att alla installationer är OK
- Googla sedan på det felmeddelande du har fått
  - Oftast något namn eller någon version i en inställningsfil som är fel

Inkoppling Firestore till Android platform

- Vid frågan om vilket app-namn
  - Testa att bara göra return
    - Kommer då att få rätt app-namn (enligt de inställningsfiler som finns i projektet)
  - Om man skriver själv, trunkeras namnet, och blir ”fel”

Ex. Problem att köra på Android

- Problem med Android – Firebase kopplingen
  - "No matching client found for package name"
- Dubbelkolla appens namn i följande filer:
  - androidManifest.xml
  - build.gradle
  - google-services.json
- Ska matcha namnet i Firestore-projektet

Ex. Problem med Chrome

- Fastnar i “Waiting for debug connection..”
- Testa att köra
  - // rensa allt kompilerat från projektet
  - flutter clean
- Och sedan
  - // bygg om och kör i verbose mode
  - flutter run --verbose -d chrome
- Vid behov
  - flutter pub cache repair

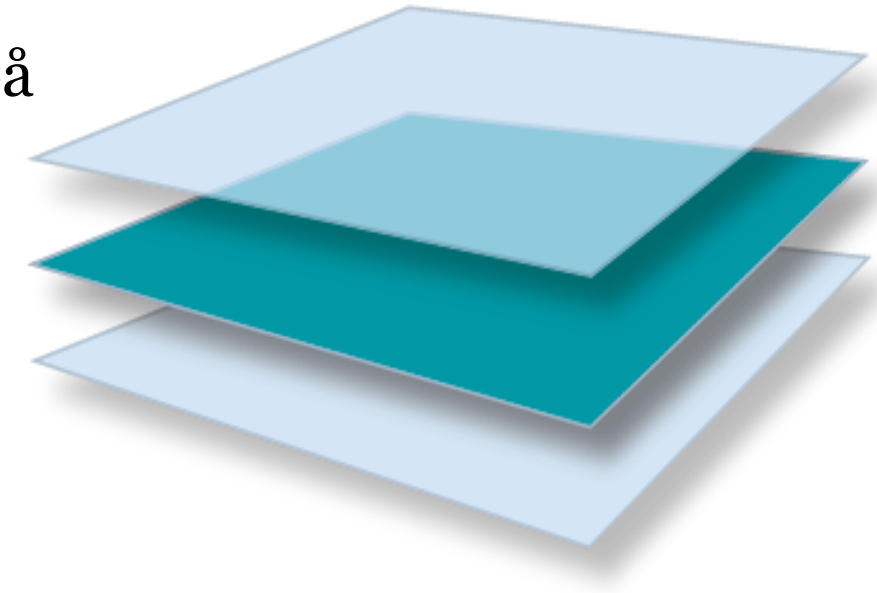
Modulär kod

Modulär kod

- Ofta naturligt att dela upp koden i skärmar
  - main_screen mapp
    - Kod för widget:en
    - Hjälp-klasser
    - Datastrukturer för att hålla data
    - Kod för att hämta data
  - second_screen mapp
    - …

Återanvändbara skärmbitar: Widgets

- Återanvända
- Flytta
- Lätt ändra på



Flutter-app arkitektur

- Tre lager:
  - Presentation layer
  - Business logic layer
  - Data layer
- Referenser (beroenden) ska aldrig gå åt sidan
  - Viktigt att UI kan bytas ut (ritas om) utan att arkitektur-lagren nedanför får hängande pekare

Beroenden

- Varje widget ska vara självtillräcklig
  - Utbytbar, flyttbar, lätt att ändra
- Beroenden bör gå rakt uppåt i widget-trädet
  - Widgets ska *inte* kommunicera med varandra direkt
  - Ingen ”sidledes” kommunikation i widget-trädet
- Idealet
  - Widgets ska bara veta om sin context

Dependency injection

Dependency injection

- Vill inte ha direkta beroenden
  - T.ex. inte veta hur en instans ska skapas
- Objekt skapas istället externt
  - Skickas med färdigt till ”ägarens” konstruktor
  - Eller via setters
  - Eller till build via context (t.ex. m.h.a. provider-biblioteket)

Exempel på injection

```
class Car {
    Engine? engine;
    List<Wheel> wheels;

    Car(this.wheels, {this.engine});

    void setEngine(Engine newEngine) {
        engine = newEngine;
    }
}

Car car = Car(wheels).setEngine(engine);
```

Dependency injection

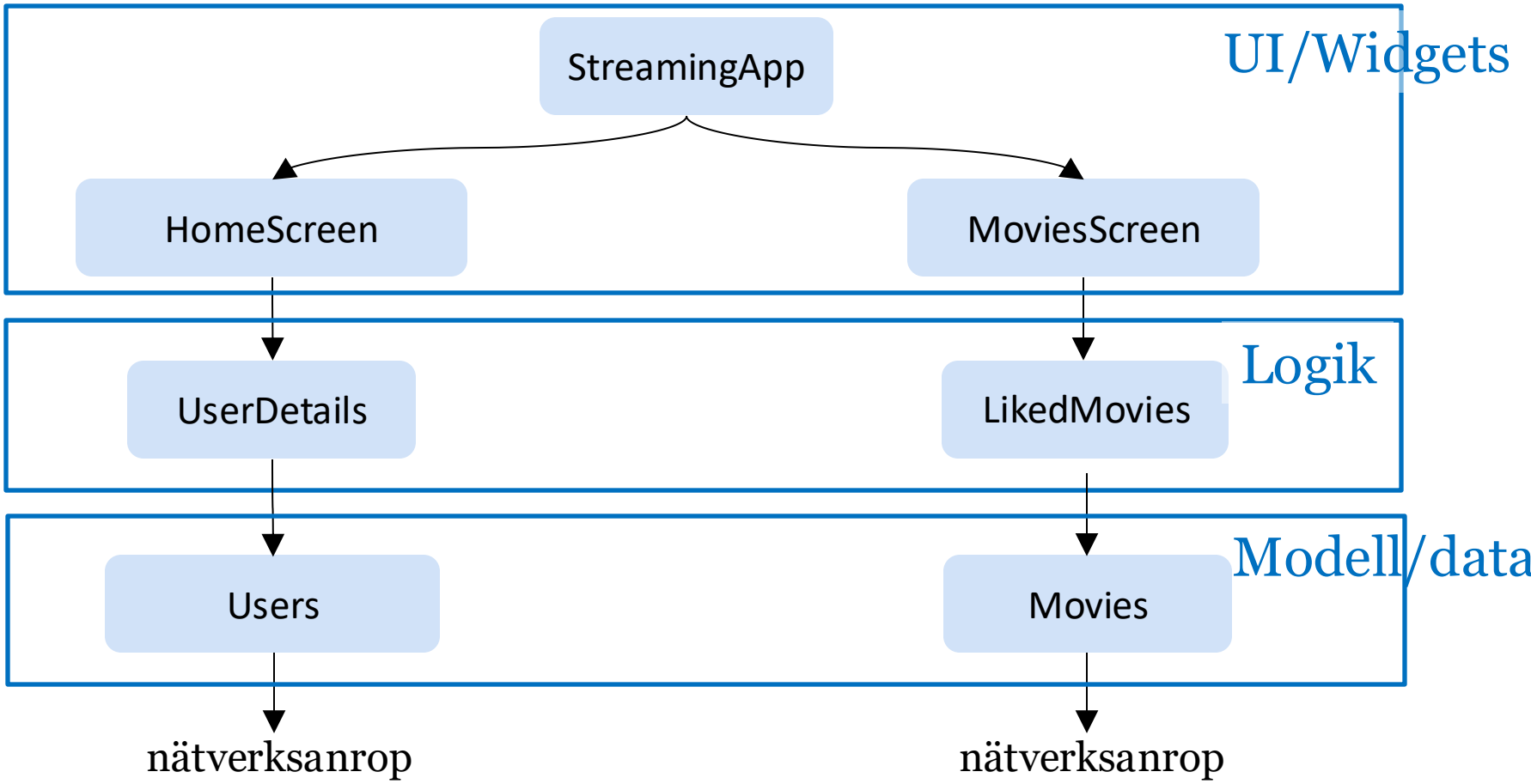
- Fördelar
  - Tydligare vilka andra klasser som denna klass behöver
    - Lättare att underhålla
  - Lättare att byta ut komponenten
  - Lättare att testa (skicka in mocks)

Model View ViewModel (MVVM) och Repository designmönstren

MVVM

- Vill ha så lite fasta beroenden som möjligt mellan UI och data-lagret
- ViewModel i Android
  - Injiceras till UI:t
  - Notifierar UI:t när data ändrats
- Provider i Flutter
  - Kan skapa providers var som helst i widget-trädet
  - Kan notifiera när data ändrats

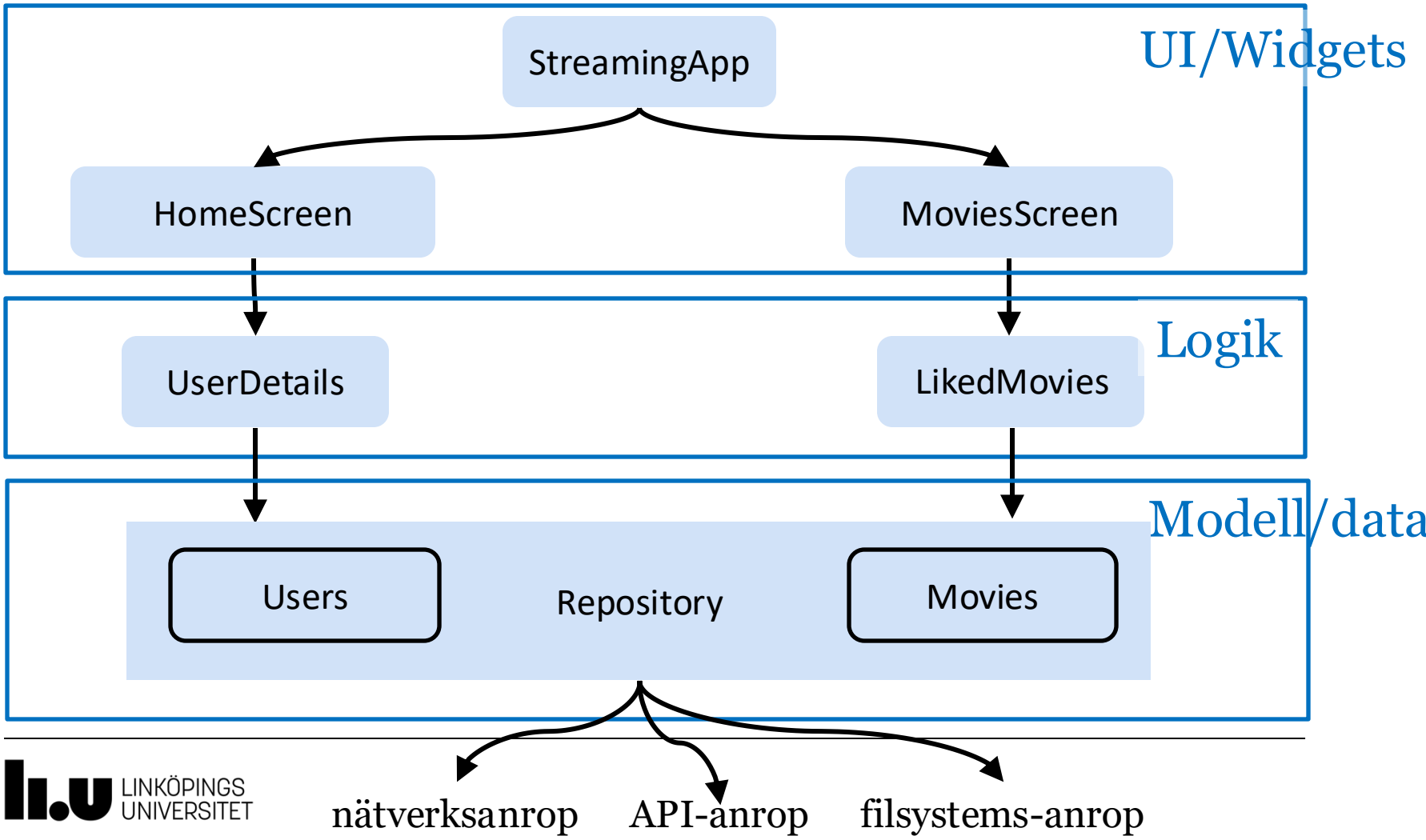
App-struktur



Inte helt optimalt

- Infrastruktur (nätverk, filhantering, etc.) utströdda
- Lätt att införa buggar
  - Glömmer att ändra på ett ställe
- Arbetsamt
  - Byta från Firestore till egen databas
    - Behöver ändra på flera ställen
- Svårt att testa, eftersom nätverksanrop tätt kopplade till (inbyggda i) varje data-klass

Bättre app-struktur



Läs mer

- Repository pattern:
  - https://fluttersamples.com
- Youtube:
  - https://www.youtube.com/watch?v=x6C20zhZHw8

rita.kovordanyi@liu.se

www.liu.se

LINKÖPINGS UNIVERSITET