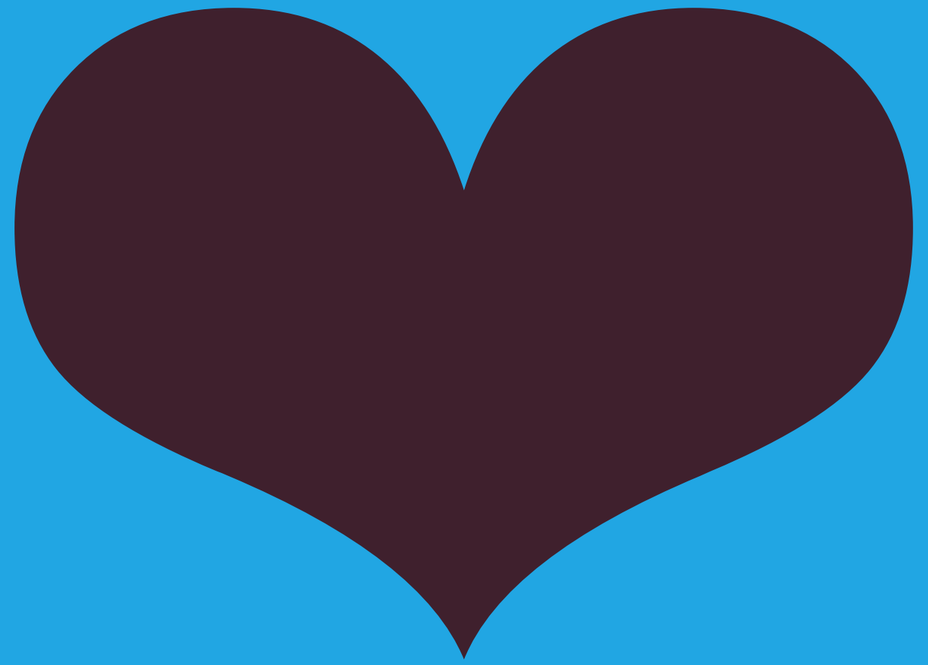
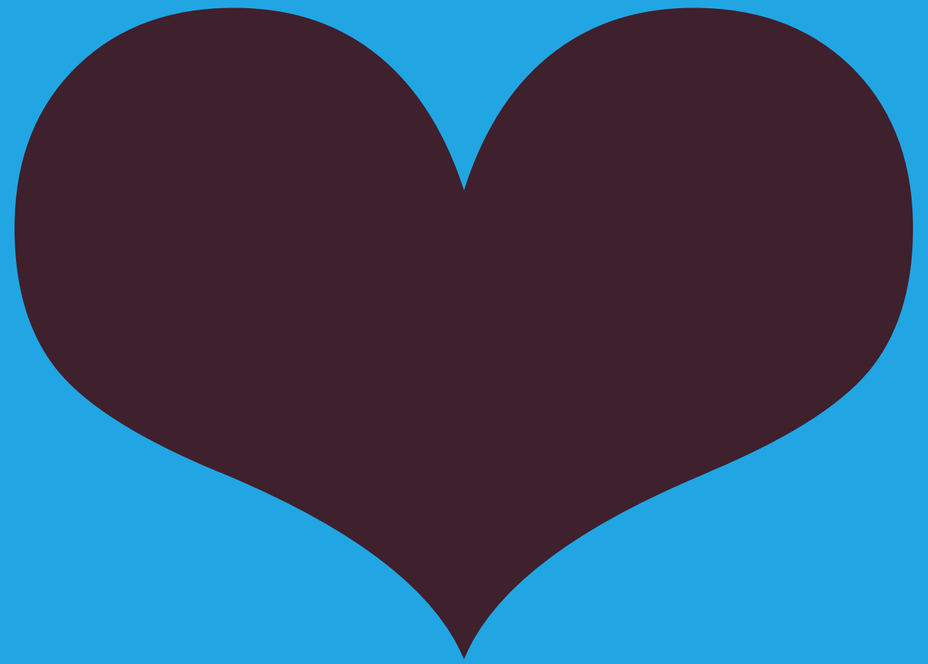
STORE

DATA LOADING MADE EASY-ISH

BRIAN PLUMMER - NY TIMES

WE  MAKING APPS

WE  OPEN SOURCE

WHY?

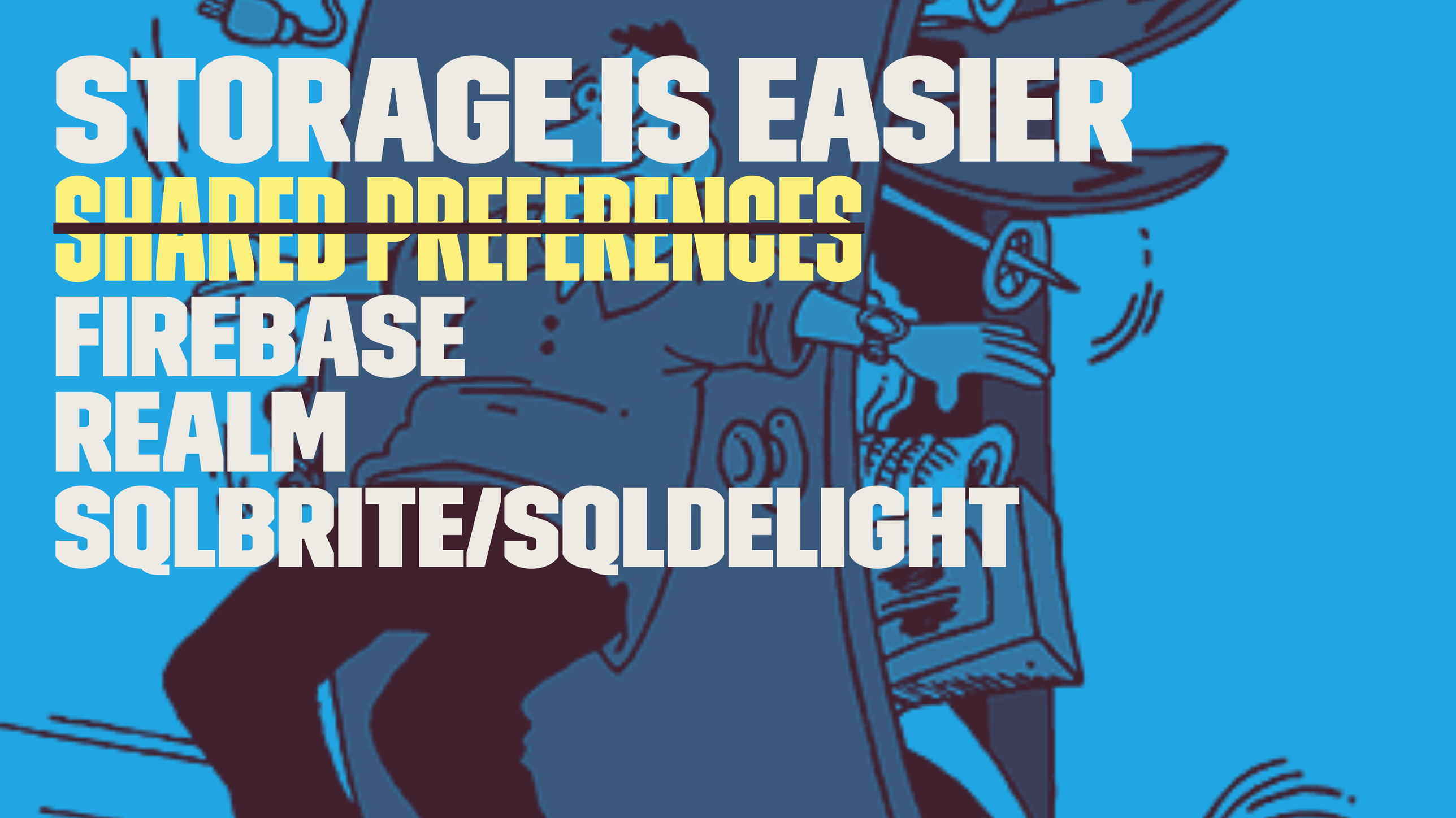
OPEN SOURCE MAKES LIFE EASIER

NETWORKING IS EASIER:

~~HTTPURLCONNECTION~~

VOLLEY

RETROFIT / OKHTTP


STORAGE IS EASIER
SHARED PREFERENCES
FIREBASE
REALM
SQLBRITE/SQLDELIGHT

PARSING IS EASIER

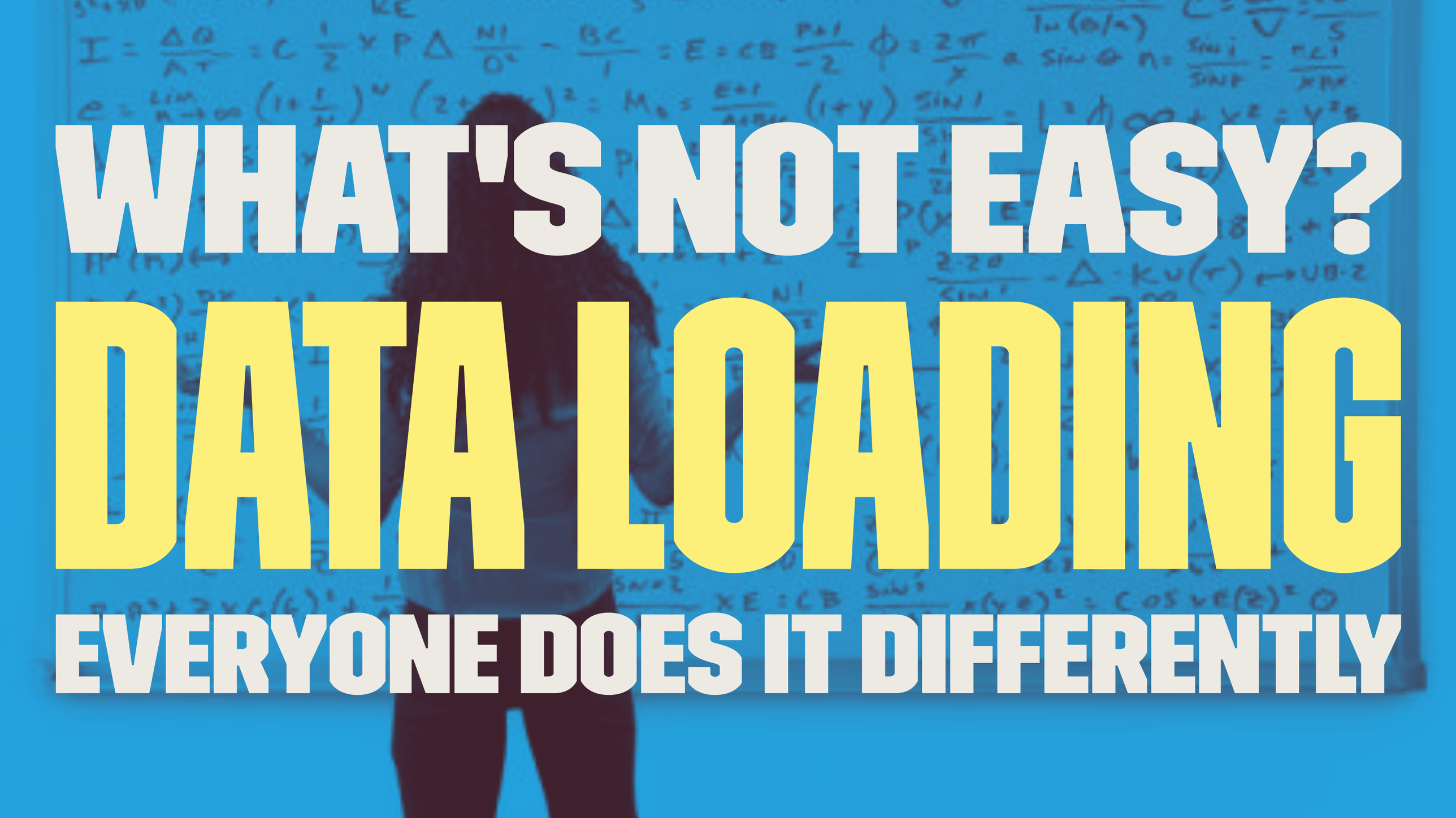
~~JSONOBJECT~~

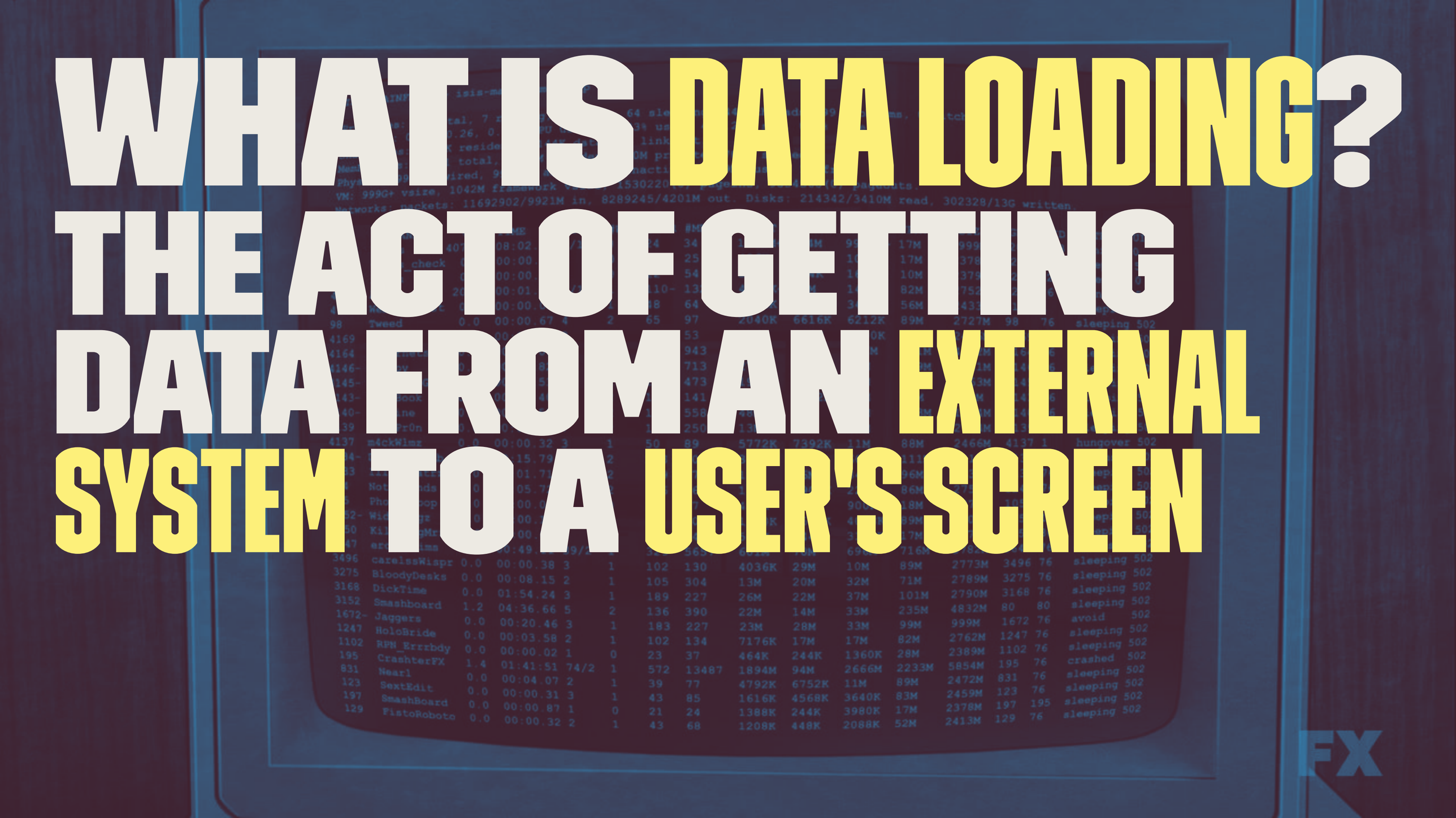
JACKSON

GSON

MOSHI


FETCHING, PERSISTING &
PARSING DATA HAS BECOME
EASY


WHAT'S NOT EASY?
DATA LOADING
EVERYONE DOES IT DIFFERENTLY


WHAT IS DATA LOADING?
THE ACT OF GETTING DATA FROM AN EXTERNAL SYSTEM TO A USER'S SCREEN
FX


LOADING IS COMPLICATED
ROTATION HANDLING IS A SPECIAL
SNOWFLAKE

NEW YORK TIMES BUILT STORE TO SIMPLIFY DATA LOADING

GITHUB.COM/NYTIMES/STORE


OUR GOALS

DATA SHOULD SURVIVE CONFIGURATION CHANGES

AGNOSTIC OF WHERE IT COMES FROM OR HOW IT IS NEEDED

ACTIVITIES AND PRESENTERS SHOULD STOP RETAINING MBS OF DATA

OFFLINE AS CONFIGURATION
CACHING AS STANDARD, NOT AN EXCEPTION

API SHOULD BE SIMPLE ENOUGH FOR AN INTERN TO USE, YET ROBUST ENOUGH FOR EVERY DATA LOAD.

HOW DO WE WORK WITH DATA AT NY TIMES?

80% CASE:
NEED DATA, DON'T CARE IF FRESH OR CACHED


NEED FRESH DATA
BACKGROUND
UPDATES &
PULL TO REFRESH

REQUESTS NEED TO BE ASYNC & REACTIVE

DATA IS DEPENDENT ON EACH OTHER, DATA LOADING SHOULD BE TOO

PERFORMANCE IS IMPORTANT

NEW CALLS SHOULD HOOK INTO IN FLIGHT RESPONSES

PARSING SHOULD BE DONE EFFICIENTLY AND MINIMALLY

PARSE ONCE AND CACHE THE RESULT FOR FUTURE CALLS

LET'S USE REPOSITORY PATTERN

BY CREATING REACTIVE AND PERSISTENT DATA STORES

REPOSITORY PATTERN

SEPARATE THE LOGIC THAT RETRIEVES THE DATA AND MAPS IT TO THE [VIEW] MODEL FROM THE [VIEW] LOGIC THAT ACTS ON THE MODEL.

THE REPOSITORY MEDIATES BETWEEN THE DATA LAYER AND THE [VIEW] LAYER OF THE APPLICATION.

WHY REPOSITORY?

MAXIMIZE THE AMOUNT OF CODE THAT CAN BE TESTED WITH AUTOMATION BY ISOLATING THE DATA LAYER

WHY REPOSITORY?

DATA SOURCE FROM MANY LOCATIONS WILL BE CENTRALLY MANAGED WITH CONSISTENT ACCESS RULES AND LOGIC

OUR IMPLEMENTATION
HTTPS://GITHUB.COM/NYTIMES/STORE

WHAT IS A STORE?

A CLASS THAT MANAGES THE FETCHING, PARSING, AND STORAGE OF A SPECIFIC DATA MODEL

TELL A STORE:
WHAT TO FETCH

TELL A STORE:
WHAT TO FETCH
WHERE TO CACHE

TELL A STORE:
WHAT TO FETCH
WHERE TO CACHE
HOW TO PARSE

TELL A STORE:

WHAT TO FETCH
WHERE TO CACHE
HOW TO PARSE
THE STORE HANDLES FLOW

STORES ARE OBSERVABLE

```
Observable<T> get( V key);

Observable<T> fetch(V key)

Observable<T> stream()

void clear(V key)
```


LET'S SEE HOW STORES
HELPED US ACHIEVE
OUR GOALS BY LOADING
A CURRYWURST

GET IS FOR HANDLING OUR 80% USE CASE

```
public final class CurrywurstActivity {
    Store<Currywurst, String> currywurstStore;
    void onCreate() {
        store.get("ketchup")
             .subscribe(currywurst -> show(currywurst));
        }
    }
```

ON CONFIGURATION CHANGE
YOU ONLY NEED TO SAVE/RESTORE YOUR KEY TO GET YOUR CACHED DATA

```
public final class CurrywurstActivity {
    void onRecreate(String topping) {
        store.get(topping)
             .subscribe(currywurst -> show(currywurst));
        }
    }
```

PLEASE DO NOT SERIALIZE A CURRYWURST
IT WOULD BE VERY MESSY

WHAT WE GAIN?:

FRAGMENTS/PRESENTERS DON'T NEED TO BE RETAINED

NO TRANSACTIONTOOLARGEEXCEPTIONS

ONLY KEYS NEED TO BE PARCELABLE

EFFICIENCY IS IMPORTANT:

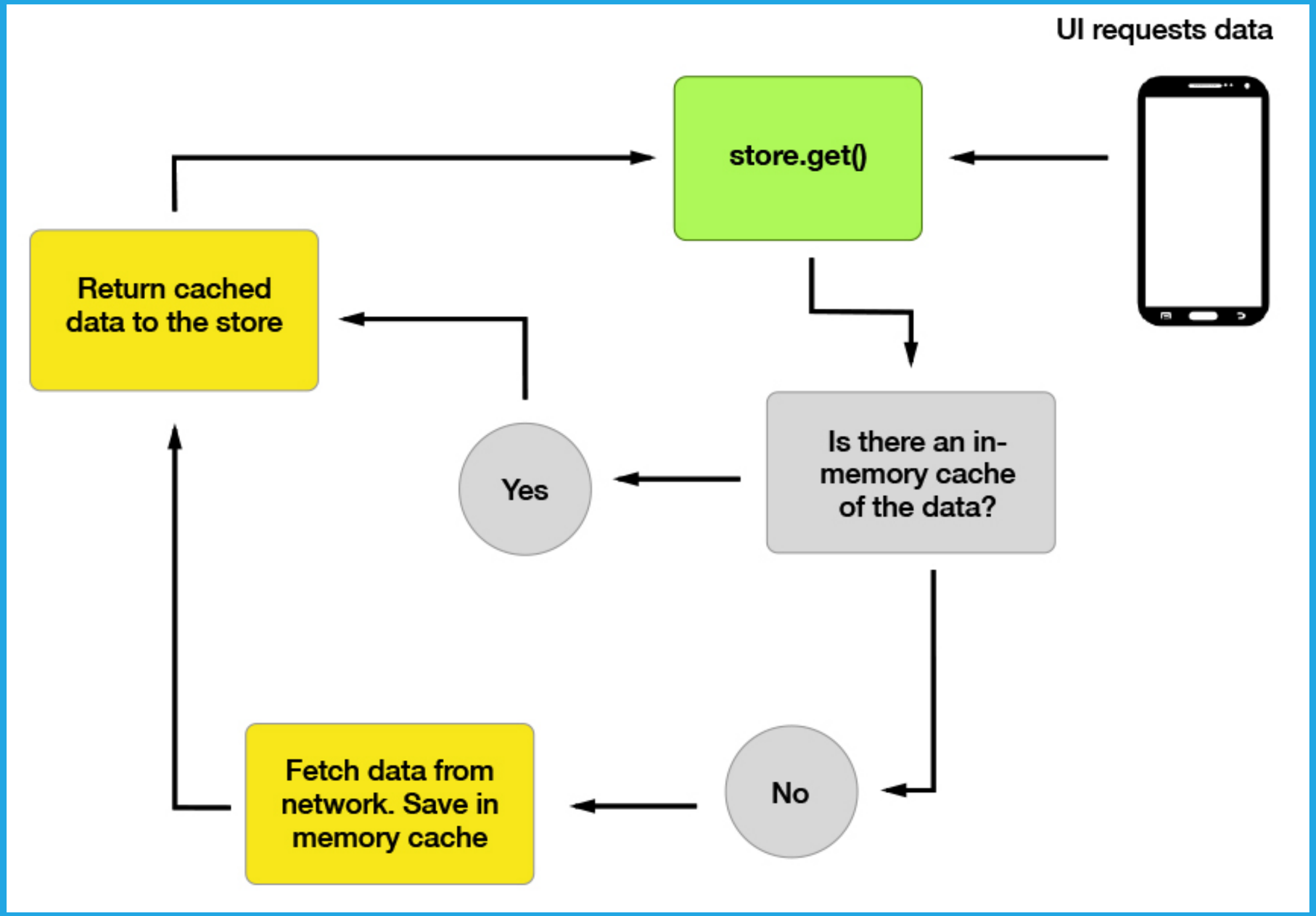
```
for(i=0;i<20;i++){
    store.get(topping)
                    .subscribe(currywurst -> getView()
                    .setData(currywurst));
}
```

MANY CONCURRENT GET REQUESTS WILL STILL ONLY HIT NETWORK ONCE

FETCHING NEW DATA

```
public class CurrywurstPresenter {
    Store<Currywurst, String> store;

    void onPTR() {
        store.fetch("ketchup")
                .subscribe(currywurst ->
                    getView().setData(currywurst));
            }
        }
```

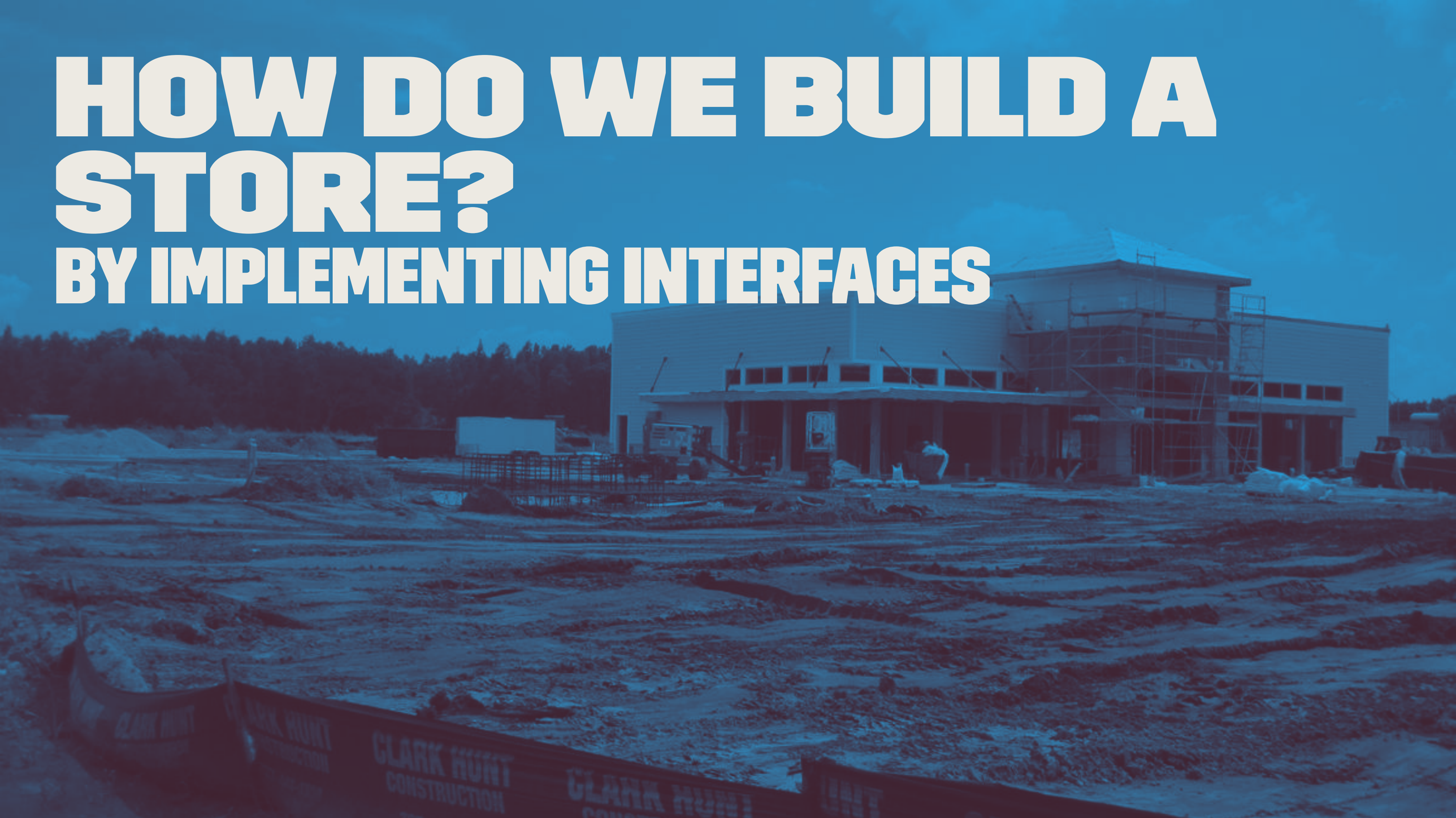

UI requests data
store.get()
Return cached data to the store
Yes
Is there an in-memory cache of the data?
No
Fetch data from network. Save in memory cache

STREAM LISTENS FOR EVENTS

```
public class CurrywurstBar {
    Store<Currywurst, String> store;
    void showCurrywurstDone() {
        store.stream()
        .subscribe(currywurst -> showSnackBar(currywurst));
    }
}
```

HOW DO WE BUILD A STORE?

BY IMPLEMENTING INTERFACES



INTERFACES

```
Fetcher<Raw, Key>{
    Observable<Raw> fetch(Key key);
}
Persister<Raw, Key>{}
    Observable<Raw> read(Key key);
    Observable<Boolean> write(Key key, Raw raw);
}
Parser<Raw, Parsed> extends Func1<Raw, Parsed>{
     Parsed call(Raw raw);
}
```

FETCHER DEFINES HOW A STORE WILL GET NEW DATA

```
Fetcher<Currywurst,String> currywurstFetcher =
new Fetcher<Currywurst, String>() {
    public Observable<Currywurst> fetch(String topping) {
        return currywurstApi.order(topping);
};
```

BECOMING OBSERVABLE

```
Fetcher<Currywurst,String> currywurstFetcher =
topping ->
Observable.fromCallable(() -> client.fetch(topping));
```

PARSERS HELP WITH FETCHERS THAT DON'T RETURN VIEW MODELS

SOME PARSERS TRANSFORM

```
Parser<Currywurst, CurrywurstBox> boxParser = new Parser<>() {
    public CurrywurstBox call(Currywurst currywurst) {
        return new CurrywurstBox(currywurst);
    }
};
```

OTHERS READ STREAMS

```
Parser<BufferedSource, Currywurst> parser = source -> {
    InputStreamReader reader =
    new InputStreamReader(source.inputStream());
    return gson.fromJson(reader, Currywurst.class);
}
```

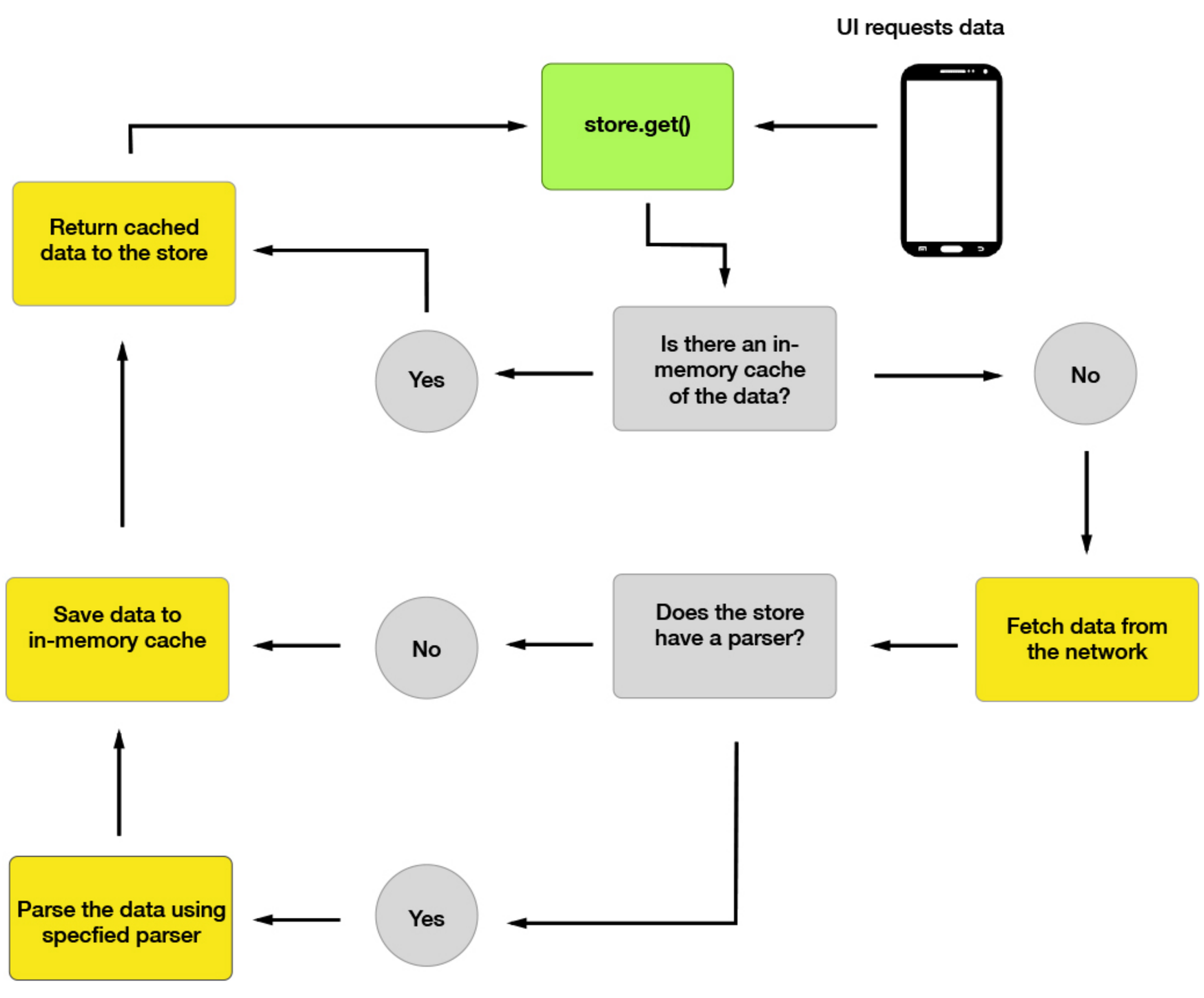
MIDDLEWARE IS FOR THE COMMON JSON CASES

```
Parser<BufferedSource, Currywurst> parser
= GsonParserFactory.createSourceParser(gson,Currywurst.class)

Parser<Reader, Currywurst> parser
= GsonParserFactory.createReaderParser(gson,Currywurst.class)

Parser<String, Currywurst> parser
= GsonParserFactory.createStringParser(gson,Currywurst.class)
```

```
'com.nytimes.android:middleware:CurrentVersion'
'com.nytimes.android:middleware:jackson:CurrentVersion'
'com.nytimes.android:middleware-moshi:CurrentVersion'
```


UI requests data
store.get()
Return cached data to the store
Yes
Is there an in-memory cache of the data?
No
Save data to in-memory cache
No
Does the store have a parser?
Fetch data from the network
Parse the data using specfied parser
Yes

ADDING OFFLINE WITH PERSISTERS

FILE SYSTEM RECORD PERSISTER

```
FileSystemRecordPersister.create(
fileSystem,key -> "currywurst"+key, 1, TimeUnit.DAYS);
```

FILE SYSTEM IS KISS STORAGE

```
 interface FileSystem {
   BufferedSource read(String var) throws FileNotFoundException;
   void write(String path, BufferedSource source) throws IOException;
   void delete(String path) throws IOException;
}

compile 'com.nytimes.android:filesystem:CurrentVersion'
```

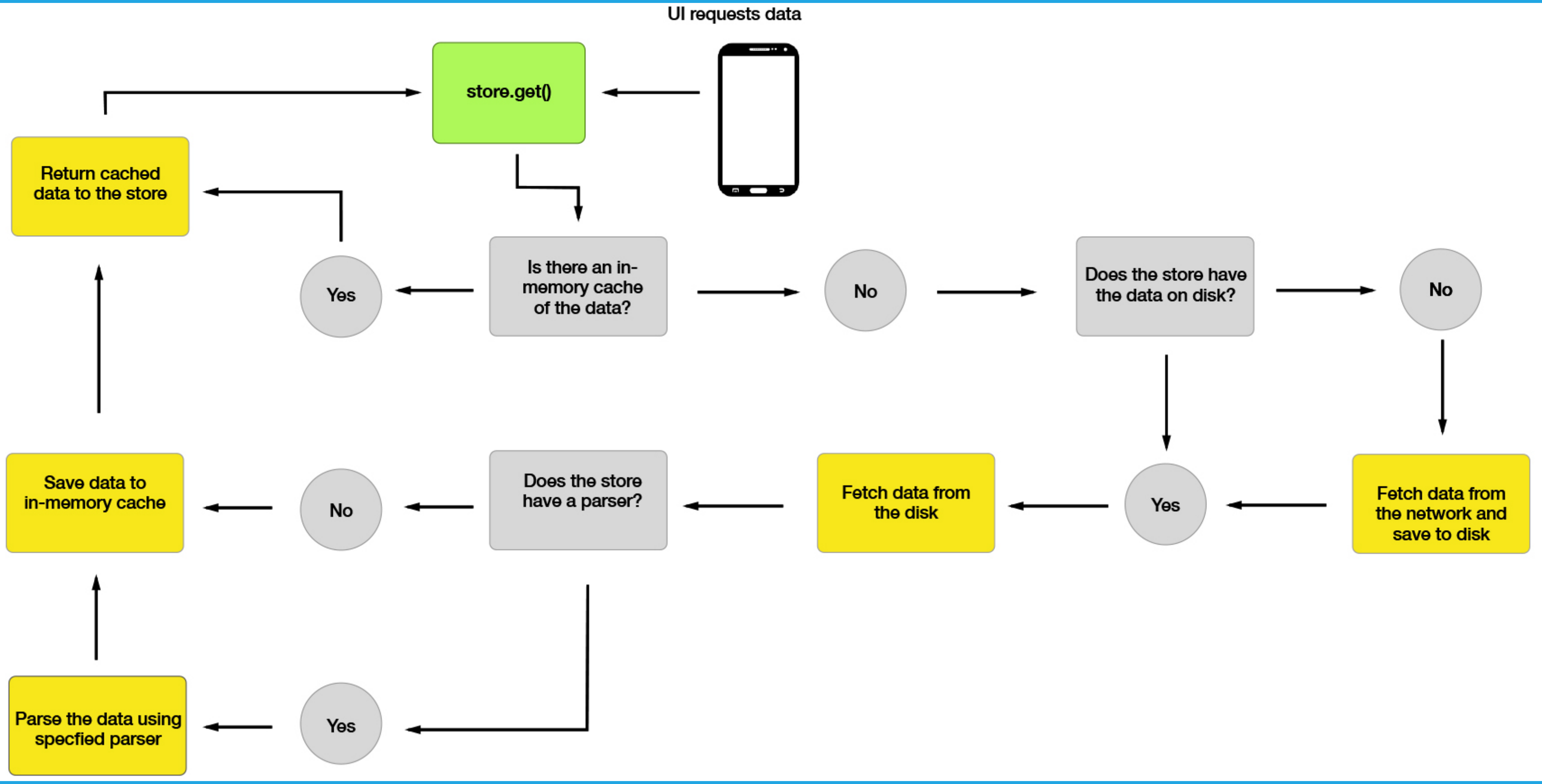
DON'T LIKE OUR PERSISTERS? NO PROBLEM! YOU CAN IMPLEMENT YOUR OWN

PERSISTER INTERFACES

```
Persister<Raw, Key> {
    Maybe<Raw> read(Key key);
    Single<Boolean> write(Key key, Raw raw);
 }

Clearable<Key> {
    void clear(Key key);
}

RecordProvider<Key> {
    RecordState getRecordState( Key key);
```


UI requests data
store.get()
Return cached data to the store
Yes
Is there an in-memory cache of the data?
No
Does the store have the data on disk?
No
Save data to in-memory cache
No
Does the store have a parser?
Fetch data from the disk
Yes
Fetch data from the network and save to disk
Parse the data using specfied parser
Yes


WE HAVE OUR COMPONENTS!
LET'S BUILD AND OPEN A STORE

MULTIPARSER

```
List<Parser> parsers=new ArrayList<>();
parsers.add(GsonParserFactory.createSourceParser(gson, Currywurst.class));
parsers.add(boxParser);
```

STORE BUILDER

```
List<Parser> parsers=new ArrayList<>();
parsers.add(GsonParserFactory.createSourceParser(gson, Currywurst.class));
parsers.add(boxParser);

StoreBuilder<String,BufferedSource,CurrywurstBox> parsedWithKey()
```

ADD OUR PARSER

```
List<Parser> parsers=new ArrayList<>();
parsers.add(GsonParserFactory.createSourceParser(gson, Currywurst.class));
parsers.add(boxParser);

StoreBuilder<String,BufferedSource,CurrywurstBox> parsedWithKey()
        .fetcher(topping -> currywurstSource.fetch(topping))
```

NOW OUR PERSISTER

```
List<Parser> parsers=new ArrayList<>();
parsers.add(GsonParserFactory.createSourceParser(gson, Currywurst.class));
parsers.add(boxParser);

StoreBuilder<String,BufferedSource,CurrywurstBox> parsedWithKey()
        .fetcher(topping -> currywurstSource.fetch(topping))
        .persister( FileSystemRecordPersister.create( fileSystem,
           key -> "prefix"+key, 1, TimeUnit.DAYS))
```

AND OUR PARSERS

```
List<Parser> parsers=new ArrayList<>();
parsers.add(GsonParserFactory.createSourceParser(gson, Currywurst.class));
parsers.add(boxParser);


StoreBuilder<String,BufferedSource,CurrywurstBox> parsedWithKey()
        .fetcher(topping -> currywurstSource.fetch(topping))
        .persister( FileSystemRecordPersister.create( fileSystem,
           key -> "prefix"+key, 1, TimeUnit.DAYS))
        .parsers(parsers)
```

TIME TO OPEN A STORE

```
List<Parser> parsers=new ArrayList<>();
parsers.add(GsonParserFactory.createSourceParser(gson, Currywurst.class));
parsers.add(boxParser);

Store<CurrywurstBox, String> currywurstStore =
StoreBuilder<String,BufferedSource,CurrywurstBox> parsedWithKey()
        .fetcher(topping -> currywurstSource.fetch(topping))
        .persister( FileSystemRecordPersister.create( fileSystem,
       key -> "prefix"+key, 1, TimeUnit.DAYS))
        .parsers(parsers)
        .open();
```


CONFIGURING CACHES

CONFIGURING MEMORY POLICY

```
StoreBuilder
        .<String, BufferedSource, Currywurst>parsedWithKey()
        .fetcher(topping -> currywurstSource.fetch(topping))
        .memoryPolicy(MemoryPolicy
                .builder()
                .setMemorySize(10)
                .setExpireAfter(24)
                .setExpireAfterTimeUnit(HOURS)
                .build())
        .open()
```

REFRESH ON STALE - BACKFILLING THE CACHE

```
Store<Currywurst, String> currywurstStore = StoreBuilder
        .<String, BufferedSource, Currywurst>parsedWithKey()
        .fetcher(topping -> currywurstSource.fetch(topping))
        .parsers(parsers)
        .persister(persister)
        .refreshOnStale()
        .open();
```

NETWORK BEFORE STALE POLICY

```
Store<Currywurst, String> currywurstStore = StoreBuilder
        .<String, BufferedSource, Currywurst>parsedWithKey()
        .fetcher(topping -> currywurstSource.fetch(topping))
        .parsers(parsers)
        .persister(persister)
        .networkBeforeStale()
        .open();
```


STORES IN THE WILD

INTERN PROJECT: BEST SELLERS LIST

```
public interface Api {
    @GET
    Observable<BufferedSource>
    getBooks(@Path("category") String category);
```

```
Store<Books, BarCode> provideBooks(FileSystem fileSystem,
Gson gson, Api api) {
    return StoreBuilder.<BarCode, BufferedSource, Books>
            parsedWithKey()
           .fetcher(category -> api.getBooks(category))
           .persister(create(fileSystem))
           .parser(createSourceParser(gson, Books.class))
           .open();
}
```

```
public class BookActivity{
...
onCreate(...){
  bookStore
    .get(category)
    .subscribe();
}
```

HOW DO BOOKS GET UPDATED?

```
public class BackgroundUpdater{
  ...
 bookStore
    .fetch(category)
    .subscribe();
  }
```

DATA AVAILABLE WHEN SCREEN NEEDS IT

UI CALLS GET, BACKGROUND SERVICES CALL FRESH

LIVEDATA, NO PROBLEM

```
public class WurstLiveData extends WurstLiveData<Wurst> {

    protected void onActive() {

      curryworstStore
       .get("ketchup")
       .subscribe(new SingleObserver<Currywurst>() {
            ......
            @Override
            public void onSuccess(Currywurst currywurst) {
              setValue(currywurst);
            }
            ......
       });
```

```
LiveData<Currywurst> getCurrywurst(String topping) {

        MutableLiveData<Currywurst> data = new
        MutableLiveData<>();

        curryworstStore
          .get(topping)
          .subscribe(new SingleObserver<Currywurst>() {
              ......
              @Override
              public void onSuccess(Currywurst currywurst) {
                data.setValue(currywurst);
              }
              ......
          });
```

DEPENDENT CALLS

MAP ONE STORE GET TO ANOTHER

```
public Observable<SectionFront> getSectionFront(String name) {
        return feedStore.get()
                .map(feed -> feed.getSectionMeta(name))
                .flatMap(meta -> sfStore.get(meta)));
    }
```


RELATIONSHIPS BY OVERRIDING
GET/FETCH

VIDEO STORE RETURNS SINGLE VIDEOS
PLAYLIST STORE RETURNS PLAYLIST
HOW DO WE GET A PLAYLIST WITH ALL VIDEOS?

STEP 1: CREATE VIDEO STORE

```
Store<Video, Long> provideVideoStore(VideoFetcher fetcher,
                                     Gson gson) {
    return StoreBuilder.<Long, BufferedSource, Video>parsedWithKey()
            .fetcher(fetcher)
            .parser(createSourceParser(gson, Video.class))
            .open();
}
```

STEP 2: CREATE PLAYLIST STORE

```
public class VideoPlaylistStore extends RealStore<Playlist, Long> {

final Store<Video, Long> videoStore;

public VideoPlaylistStore(@NonNull PlaylistFetcher fetcher,
                          @NonNull Store<Video, Long> videoStore,
                          @NonNull Gson gson) {
    super(fetcher,
          NoopPersister.create(),
          createSourceParser(gson, Playlist.class));
    this.videoStore = videoStore;
}
```

STEP 3: OVERRIDE STORE.GET

```
public class VideoPlaylistStore extends RealStore<Playlist, Long> {

    final Store<Video, Long> videoStore;

    @Override
    public Observable<Playlist> get(Long playlistId) {
        return super.get(playlistId)
                .flatMap(playlist ->
                  from(playlist.videos())
                        .concatMap(video -> videoStore.get(video.id()))
                        .toList()
                        .map(videos ->
                                builder().from(playlist)
                                        .videos(videos)
                                        .build()));
    }
```


LISTENING FOR CHANGES

STEP 1: SUBSCRIBE TO STORE, FILTER WHAT YOU NEED

```
sectionFrontStore.stream()
                 .observeOn(scheduler)
                 .subscribeOn(Schedulers.io())
                 .filter(this::sectionIsInView)
                 .subscribe(this::handleSectionChange);
```

STEP 2: KICK OFF A FRESH REQUEST TO THAT STORE

```
public Observable<SectionFront> refreshSections(final String sectionName) {
    return feedStore.get()
                    .flatMapIterable(this::sectionsToUpdate)
                    .map(section->sectionFrontStore.fetch(section)));
}
```

STORE3 FEATURES

store.getRefreshing(key) -

will subscribe to get() which returns a single response, but unlike Get, Get Refreshing will stay subscribed. Anytime you call store.clear(key) anyone subscribed to getRefreshing(key) will resubscribe and force a new network response.

store.getWithResult(key) -

a Result model which returns the parsed object accompanied by its source(network/cache). allows you to distinguish if the cache is from memory or disk

WOULD LOVE CONTRIBUTIONS & FEEDBACK!

THANKS FOR LISTENING