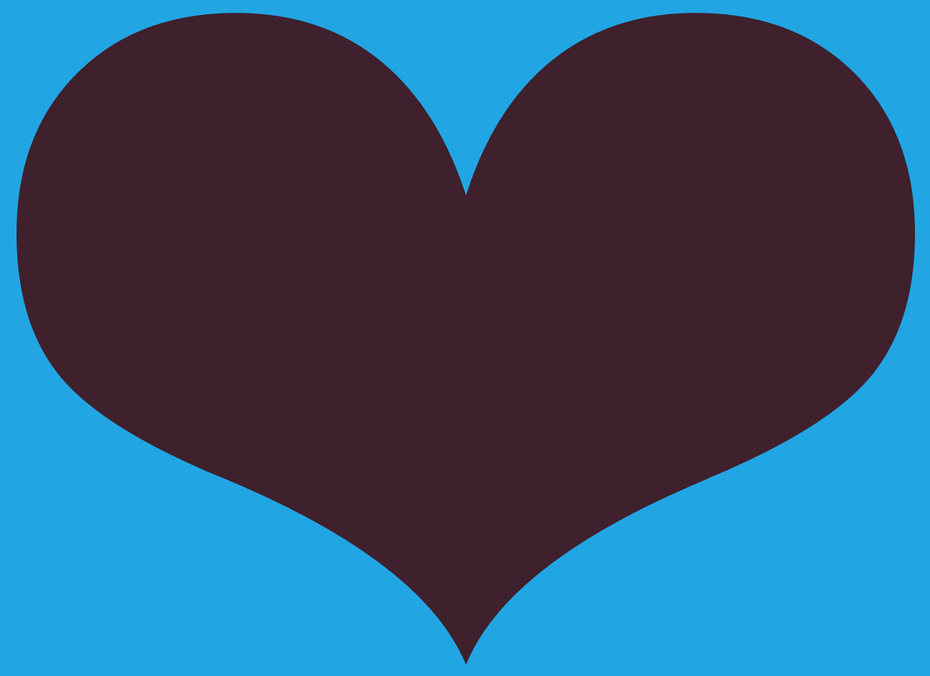
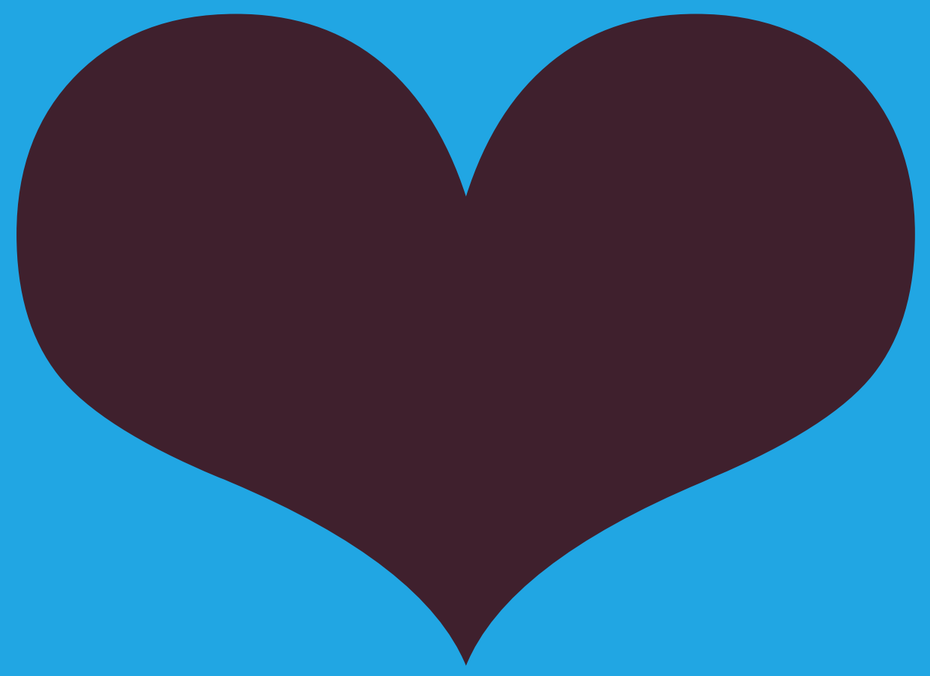
STORE

DATA LOADING MADE EASY-ISH

MIKE NAKHIMOVICH - NY TIMES

WE  MAKING APPS

NY Times App, Consulting Apps, Hack Day Apps, Apps for Blog Posts and Talks

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

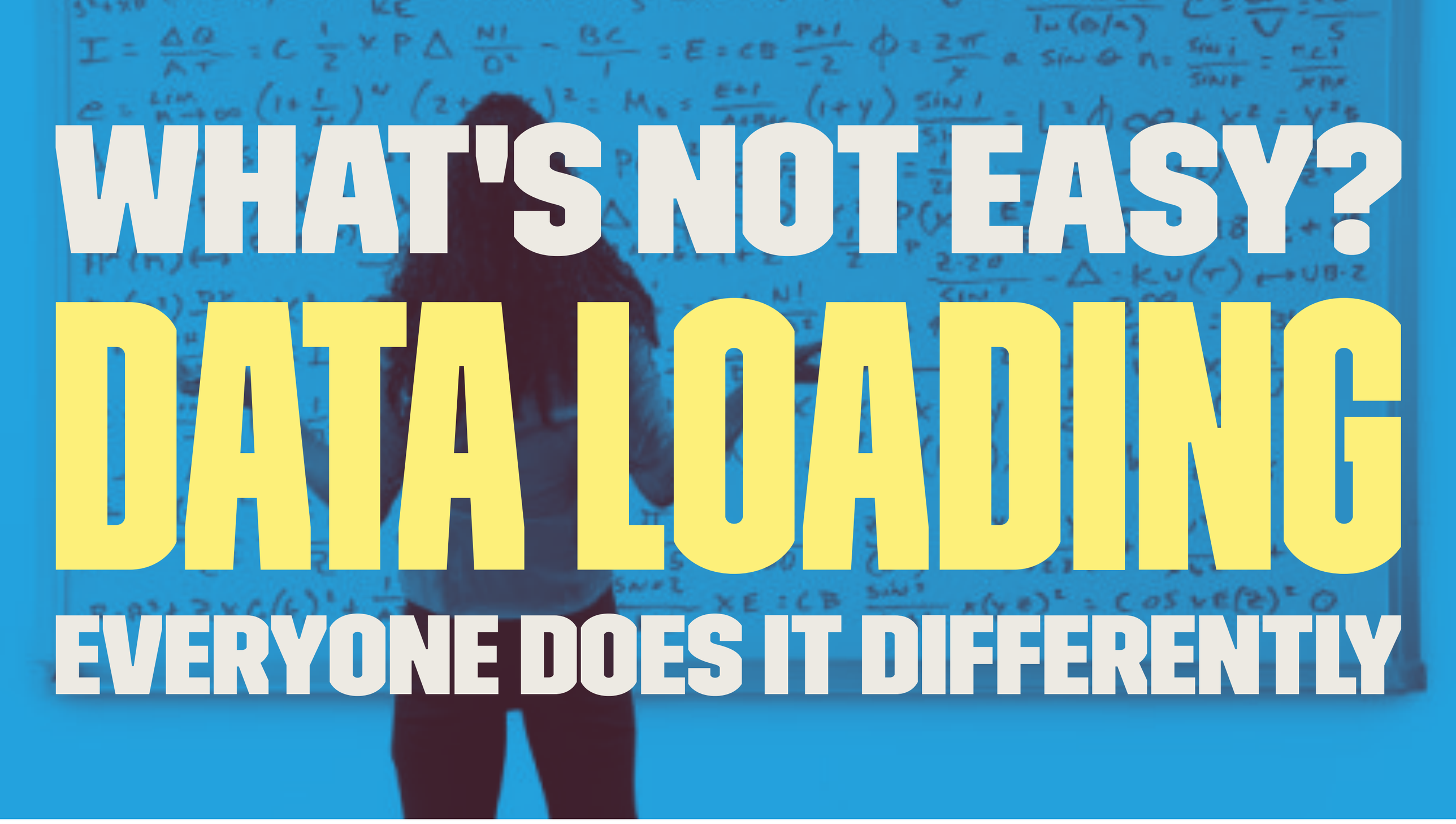
~~JSONOBJECT~~

JACKSON

GSON

MOSHI

FETCHING, PERSISTING & PARSING DATA HAS BECOME EASY



WHAT'S NOT EASY?

DATA LOADING

EVERYONE DOES IT DIFFERENTLY

Now raise your hand if you think the person sitting next to you does it in the same way

We know that we're gonna use these great network libraries but what is actually doing the network call? does my activity really need to make it? Who are what retains the data? how much do we need to serialize?

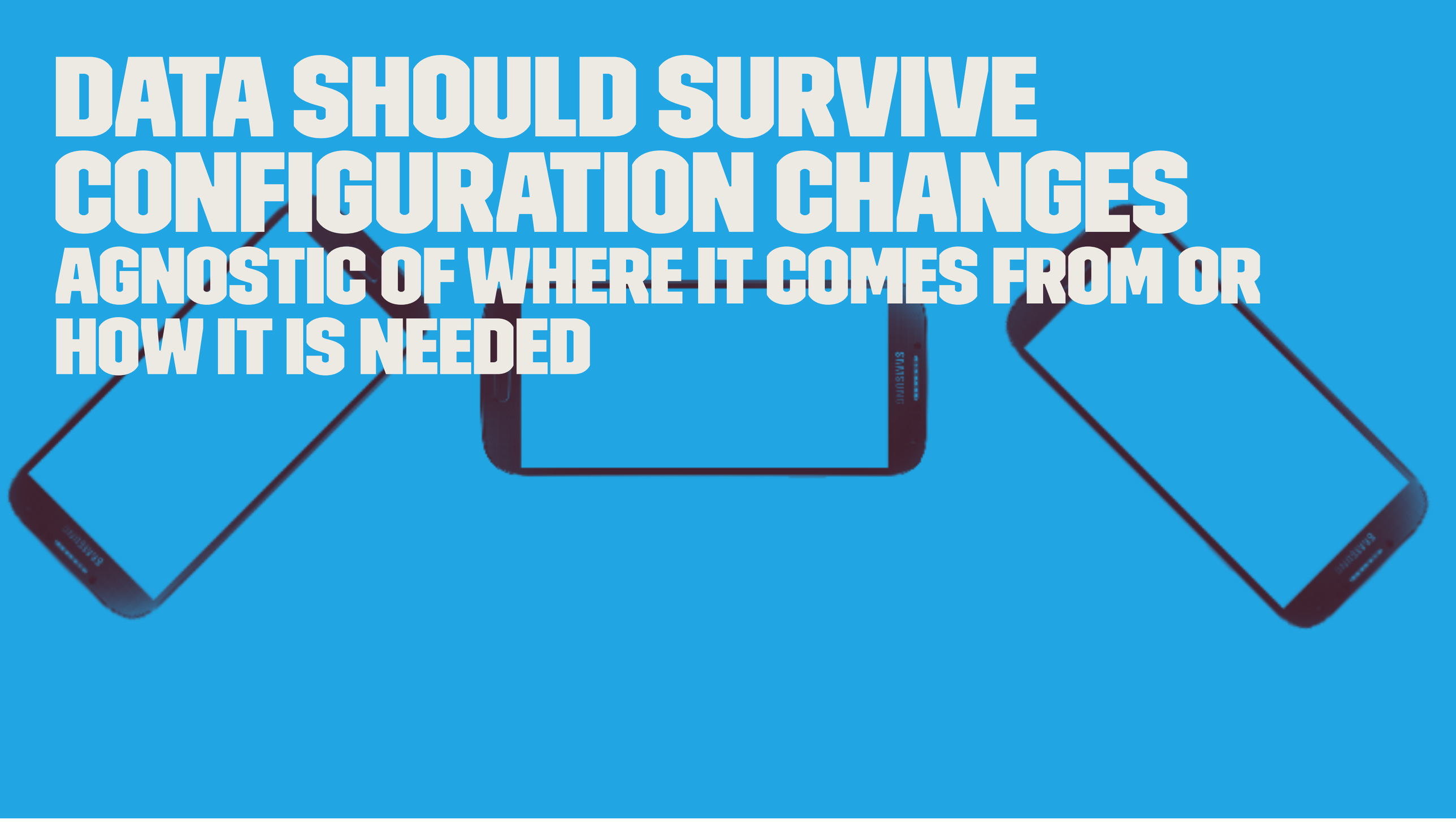
NEW YORK TIMES BUILT STORE TO SIMPLIFY DATA LOADING

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

Activities should deal with activity stuff, presenters should deal with presenting. Stores should be used to store 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.

Our team becomes 50% bigger every summer

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

Anytime we fetch an article we want comments 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?

MAKES IT EASIER TO HAVE MULTIPLE DEVS WORKING ON SAME FEATURE

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

Stores bring together all those great open source libs from earlier

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

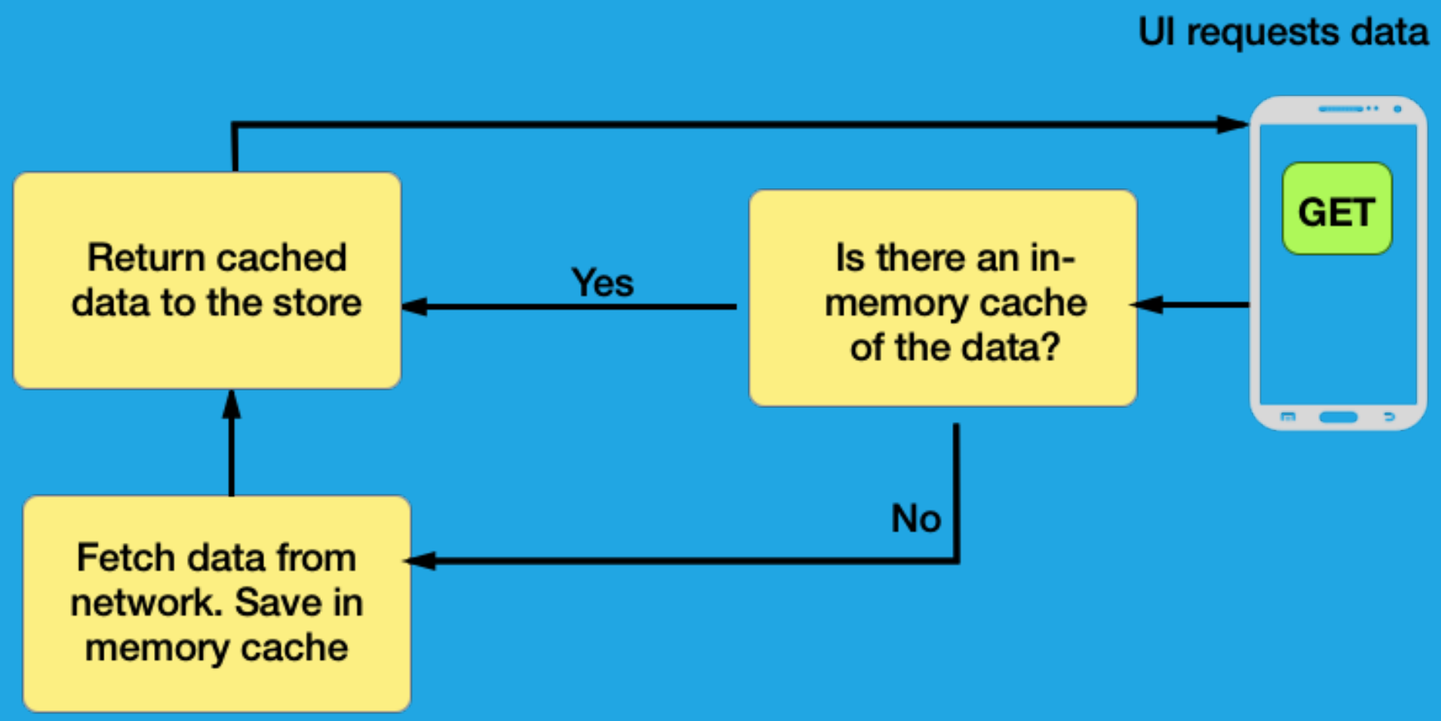
```
Observable<T> get( V key);

Observable<T> fetch(V key)

Observable<T> stream()

void clear(V key)
```


LET'S SEE HOW STORES
HELPED US ACHIEVE
OUR GOALS THROUGH A
PIZZA EXAMPLE

GET IS FOR HANDLING OUR 80% USE CASE

```
public final class PizzaActivity {
    Store<Pizza, String> pizzaStore;
    void onCreate() {
        store.get("cheese")
             .subscribe(pizza ->.show(pizza));
        }
    }
```

HOW DOES IT FLOW?



ON CONFIGURATION CHANGE
SAVE/RESTORE ONLY YOUR 🧀 AND GET YOUR CACHED 🍕

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

PLEASE DO NOT SERIALIZE A PIZZA
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(pizza -> getView()
                    .setData(pizza));
}
```

MANY CONCURRENT GET REQUESTS WILL STILL ONLY HIT NETWORK ONCE

FETCHING NEW DATA

```
public class PizzaPresenter {
    Store<Pizza, String> store;

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

HOW DOES STORE ROUTE THE REQUEST?

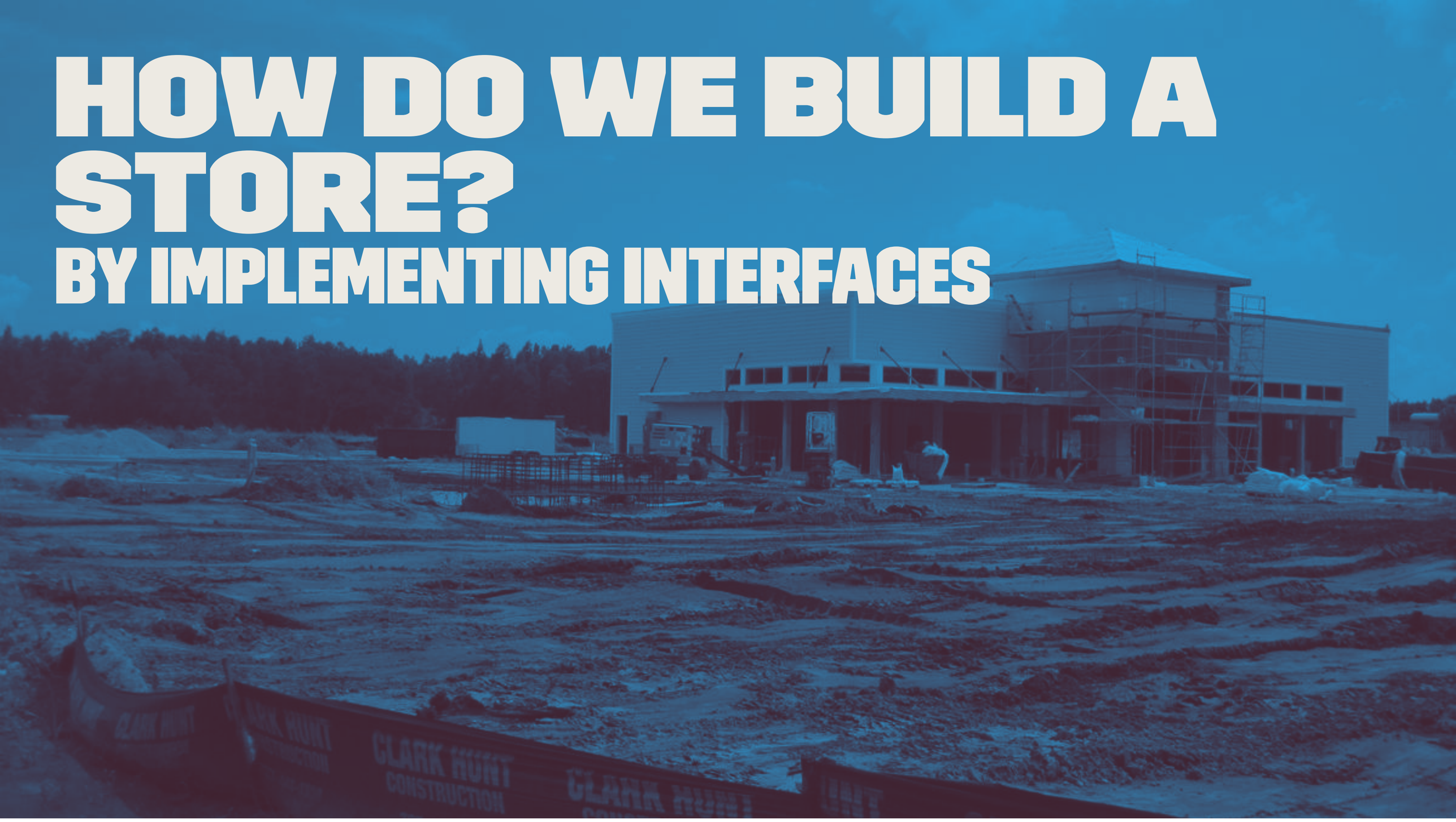
FETCH ALSO THROTTLES MULTIPLE REQUESTS AND BROADCAST RESULT

STREAM LISTENS FOR EVENTS

```
public class PizzaBar {
    Store<Pizza, String> store;
    void showPizzaDone() {
         store.stream()
         .subscribe(pizza ->
              getView().showSnackBar(pizza));
    }
}
```

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<Pizza,String> pizzaFetcher =
new Fetcher<Pizza, String>() {
    public Observable<Pizza> fetch(String topping) {
        return pizzaSource.fetch(topping);
};
```

BECOMING OBSERVABLE

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

PARSERS HELP WITH FETCHERS THAT DON'T RETURN VIEW MODELS

SOME PARSERS TRANSFORM

```
Parser<Pizza, PizzaBox> boxParser = new Parser<>() {
    public PizzaBox call(Pizza pizza) {
        return new PizzaBox(pizza);
    }
};
```

OTHERS READ STREAMS

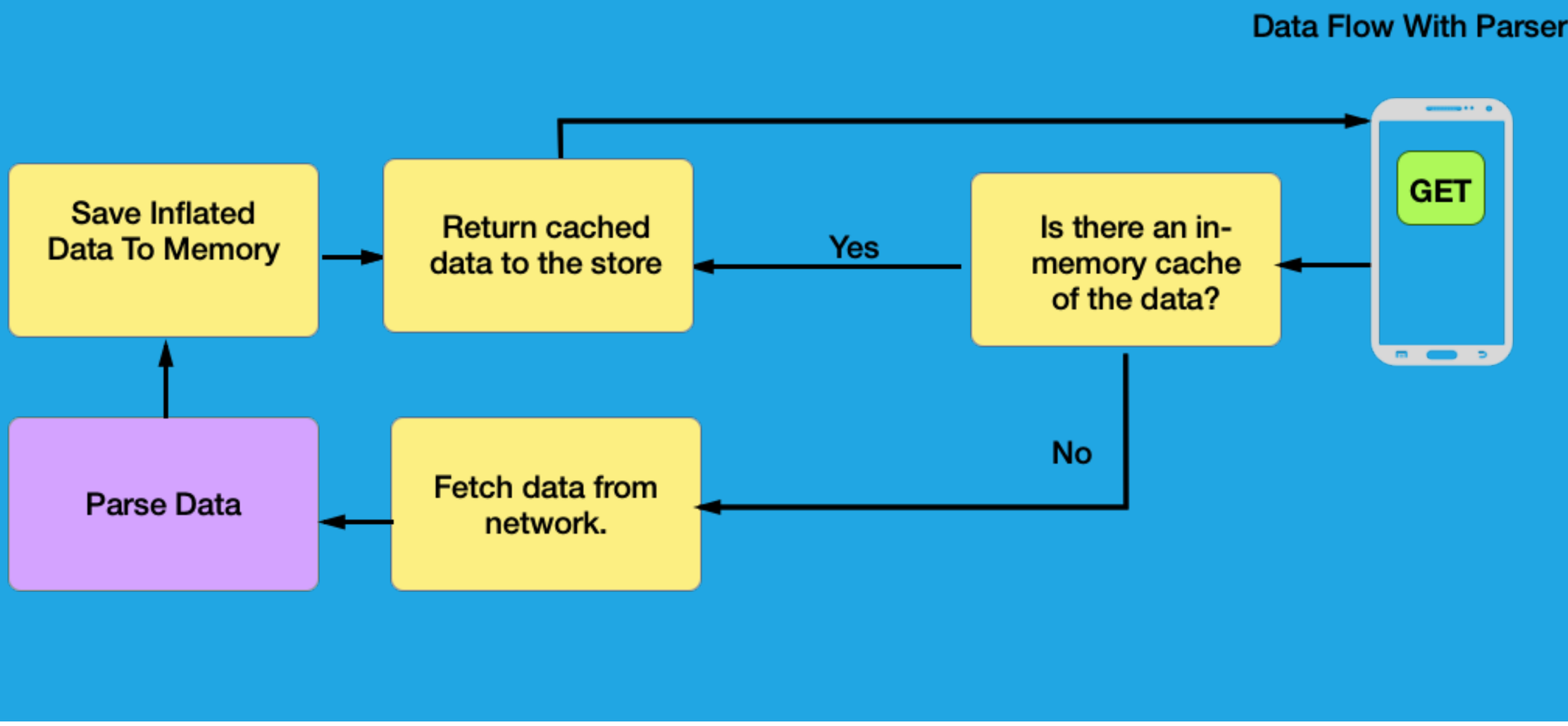
```
Parser<BufferedSource, Pizza> parser = source -> {
    InputStreamReader reader =
    new InputStreamReader(source.inputStream());
    return gson.fromJson(reader, Pizza.class);
}
```

MIDDLEWARE IS FOR THE COMMON JSON CASES

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

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

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

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

DATA FLOW WITH A PARSER



ADDING OFFLINE WITH PERSISTERS

FILE SYSTEM RECORD PERSISTER

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

FILE SYSTEM IS KISS STORAGE

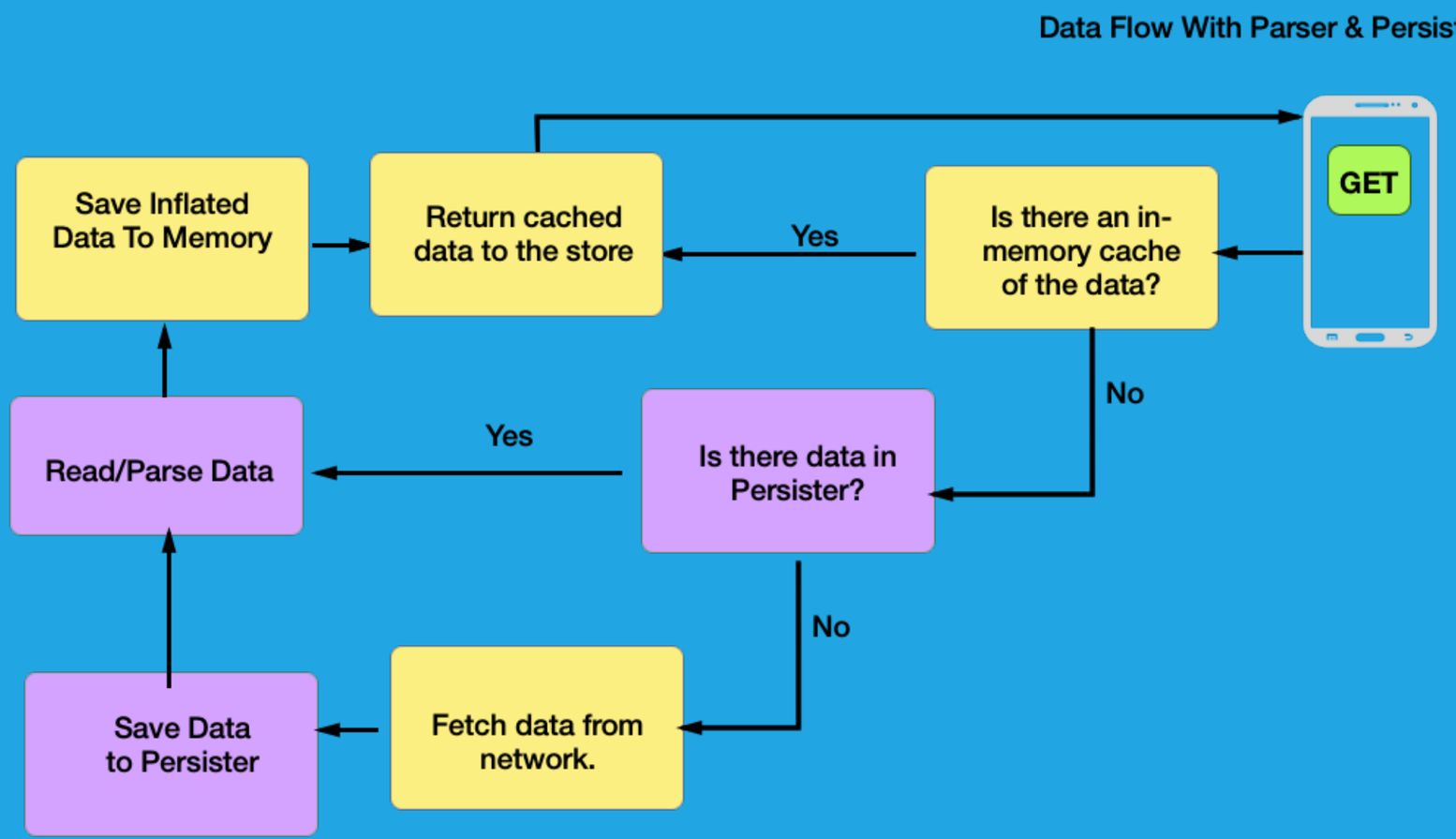
```
 interface FileSystem {
   BufferedSource read(String var) throws FileNotFoundException;
   void write(String path, BufferedSource source) throws IOException;
   void delete(String path) throws IOException;
}

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

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

PERSISTER INTERFACES

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

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

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

DATA FLOW WITH A PERSISTER



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

MULTIPARSER

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

STORE BUILDER

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

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

ADD OUR PARSER

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

StoreBuilder<String,BufferedSource,Pizza> parsedWithKey()
        .fetcher(topping -> pizzaSource.fetch(topping))
```

NOW OUR PERSISTER

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

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

AND OUR PARSERS

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

StoreBuilder<String,BufferedSource,Pizza> parsedWithKey()
        .fetcher(topping -> pizzaSource.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(boxParser);
parsers.add(GsonParserFactory.createSourceParser(gson, Pizza.class));

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


CONFIGURING CACHES

CONFIGURING MEMORY POLICY

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

REFRESH ON STALE - BACKFILLING THE CACHE

```
Store<Pizza, String> pizzaStore = StoreBuilder
        .<String, BufferedSource, Pizza>parsedWithKey()
        .fetcher(topping -> pizzaSource.fetch(topping))
        .parsers(parsers)
        .persister(persister)
        .refreshOnStale()
        .open();
```

NETWORK BEFORE STALES POLICY

```
Store<Pizza, String> pizzaStore = StoreBuilder
        .<String, BufferedSource, Pizza>parsedWithKey()
        .fetcher(topping -> pizzaSource.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);
```

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

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

HOW DO BOOKS GET UPDATED?

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

DATA AVAILABLE WHEN SCREEN NEEDS IT
UI CALLS GET, BACKGROUND SERVICES CALL FRESH

If the background update fails UI will fallback to network

DEPENDENT CALLS

SIMPLE CASE USING RXJAVA OPERATORS
MAP STORE RESULT TO ANOTHER STORE RESULT

```
public Observable<SectionFront> getSectionFront(@NonNull final String name) {
        return feedStore.get()
                .map(latestFeed -> latestFeed.getSectionOrBlog(name))
                .flatMap(section -> sfStore.get(SectionFrontId.of(section)));
    }
```

MORE COMPLICATED EXAMPLE

VIDEO STORE RETURNS SINGLE VIDEOS,
PLAYLIST STORE RETURNS PLAYLIST,
HOW DO WE COMBINE?


RELATIONSHIPS BY OVERRIDING
GET/FETCH

STEP 1: CREATE VIDEO STORE

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

STEP 2: CREATE PLAYLIST STORE

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

final Store<CherryVideoEntity, Long> videoStore;

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

STEP 3: OVERRIDE STORE.GET()

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

    final Store<CherryVideoEntity, 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.subscribeUpdates()
                 .observeOn(scheduler)
                 .subscribeOn(Schedulers.io())
                 .filter(this::sectionIsInView)
                 .subscribe(this::handleSectionChange);
```

STEP 2: KICK OFF A FRESH REQUEST TO THAT STORE

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

BONUS: HOW WE MADE STORE

RXJAVA FOR API AND FUNCTIONAL NICETIES

```
public Observable<Parsed> get(@Nonnull final Key key) {
        return Observable.concat(
            cache(key),
            fetch(key)
        ).take(1);
    }

    public Observable<Parsed> getRefreshing(@Nonnull final Key key) {
        return get(key)
                .compose(repeatOnClear(refreshSubject, key));
    }
```

GUAVA CACHES FOR MEMORY CACHE

```
memCache = CacheBuilder
            .newBuilder()
            .maximumSize(maxSize())
            .expireAfterWrite(expireAfter(),timeUnit())
            .build();
```

IN FLIGHT THROTTLER TOO

```
inflighter.get(key, new Callable<Observable<Parsed>>() {
                    public Observable<Parsed> call() {
                        return response(key);
                    }
                })
```

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

WHY GUAVA CACHES?
THEY WILL BLOCK ACROSS THREADS SAVING PRECIOUS MBS OF DOWNLOADS WHEN MAKING SAME REQUESTS IN PARALLEL

FILESYSTEM:
BUILT USING OKIO TO ALLOW STREAMING FROM NETWORK TO DISK & DISK TO PARSER
ON FRESH INSTALL NY TIMES APP SUCCESSFULLY DOWNLOADS AND CACHES 60MB OF DATA (INSERT EMOJI)


WOULD LOVE
CONTRIBUTIONS &
FEEDBACK!
THANKS FOR LISTENING