Storing app data reliably, performantly and easily

Kurt Nelson
Mobile Platform at Uber

Last time at DroidconSF...

Last time at DroidconSF...

The Dangers of SharedPreferences

SharedPreferences

It's still in the framework, and still kind of works.

Still avoided by both Uber and Google

Apps written at Uber, Google & Facebook are deployed at scale.

When working on an app "at scale", there are some extra issues you have to worry about.

Storage at Scale

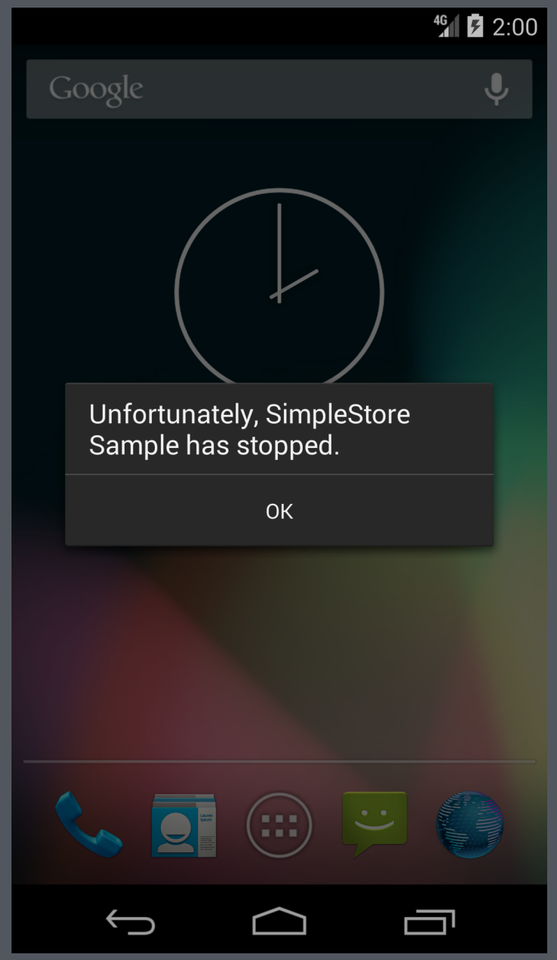
What is storage at scale?

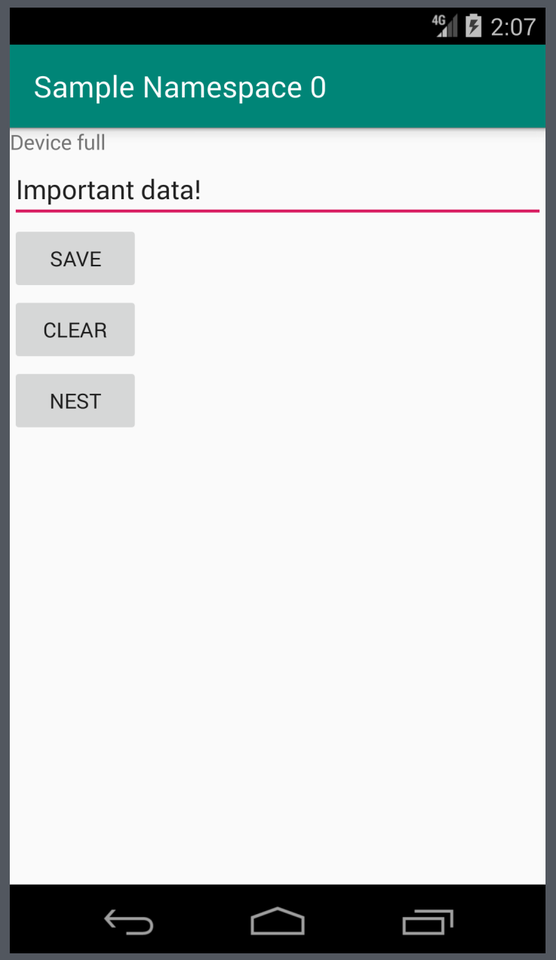
Performing I/O operations on millions of devices, millions of times a day, globally.

Storage Reliability

Why is storage reliability important?

Storage errors should not crash the user





Storage Reliability

Applications need to know when IO fails in a timely manner

Async callbacks improve the user experience

Explain fixable errors to the user

When `SharedPreferences#apply()` fails, you never find out

Storage Reliability

```
SharedPreferences sharedPref = context.getSharedPreferences("app", Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putString("auth_token", authToken);
editor.apply();

context.startActivity(MainScreenActivity.class);
finish();
```

Block progress on failure when appropriate

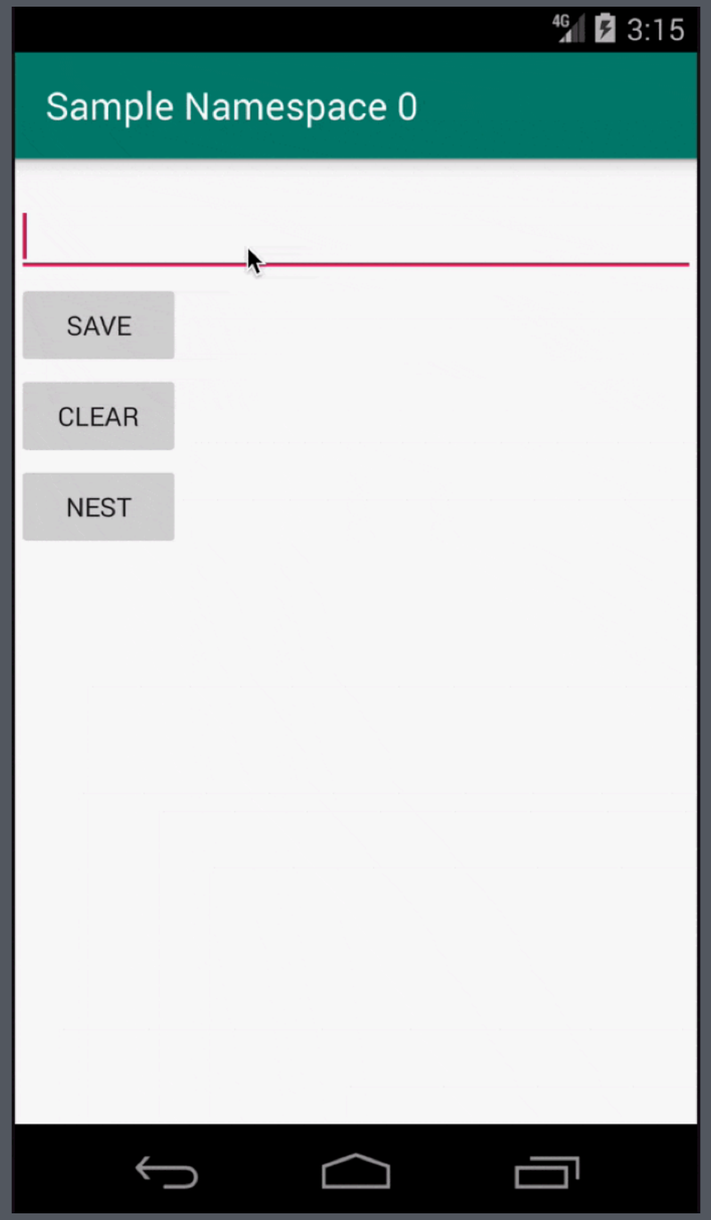
Essential for legal and privacy status, such as a EULA or GDPR opt-out.

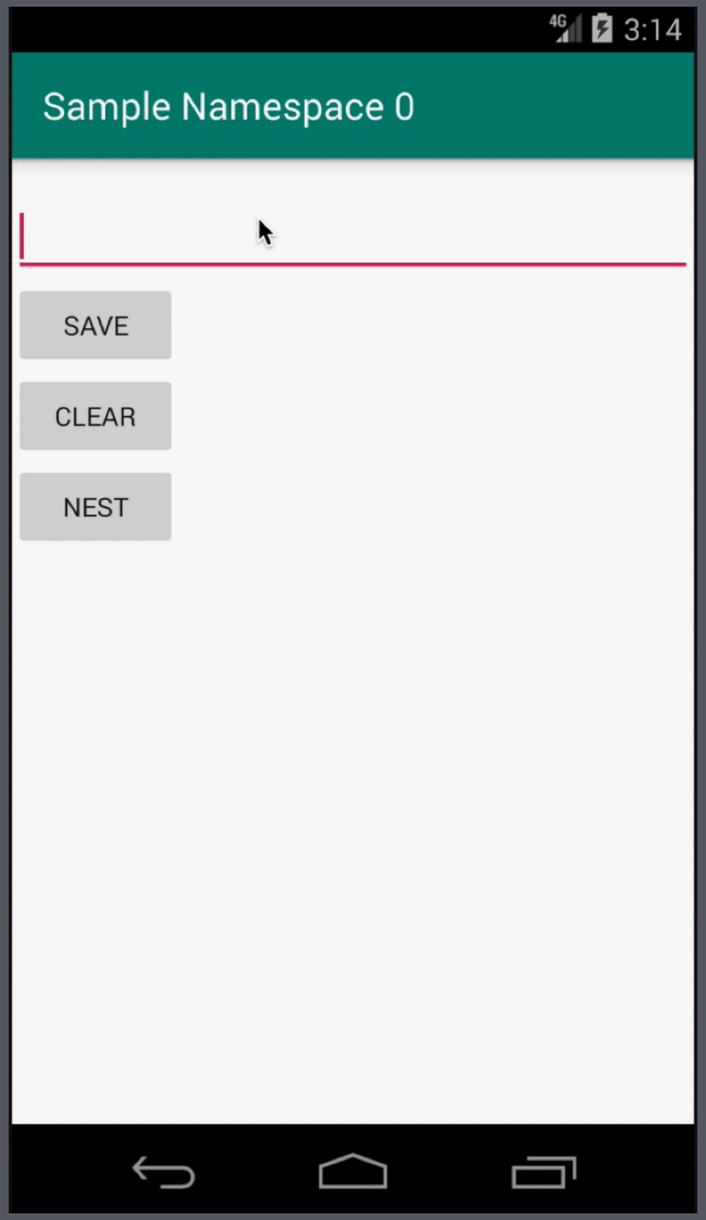
Storage Performance

Why is storage performance important?

The user experience should always be smooth.

Storage can cause frame drops.


3:15
Sample Namespace 0
SAVE
CLEAR
NEST


3:14
Sample Namespace 0
SAVE
CLEAR
NEST

It is easy to drop frames

Running at 60fps gives the device 16 milliseconds to render each frame.

Even on a modern device, the p95 storage read latency is already in this neighborhood. On low end devices, this can be drastically slower.

Storage Performance

Different types of data have different performance needs.

Cache Performance Optimizations

If a cache read is latent, maybe it should be eagerly treated as a miss?

If a cache write fails, does anyone care?

Storage Performance

Critical disk operations should block progress with a spinner until complete

API Ergonomics

API Ergonomics

- Understandable by new developers
- Common async abstraction
- Using correctly is easy
- Consistant behavior for primitives and objects
- Future proof serialization

I want a key-value store that ensures all of this.

I want a key-value store that ensures all of this.

So I built it. Twice.

SimpleStore

Simple yet performant asynchronous file storage for Android

SimpleStore

https://github.com/uber/simple-store

Write a key

```
val store = SimpleStoreFactory.create(context, MY_NAMESPACE)
store.use {
  Futures.addCallback(
    it.putString("key", "value"),
    object : FutureCallback<String> {
      override fun onSuccess(result: String) {
        // update the UI
      }

      override fun onFailure(t: Throwable) {
        Log.e("MyActivity", "Something went wrong", t)
      }
    }, mainExecutor()
  )
}
```

Namespaces

A namespace is the base unit that can be opened and closed.

Only one reference to a specific namespace is allowed to exist at a time.

Ordering is guarenteed within a namespace.

Write a key (RxJava)

```
SimpleStoreFactory.create(context, MY_NAMESPACE).use {
  Single.fromFuture(it.putString("some_key", "Foo value"))
      .observeOn(uiThread())
      .subscribe(updateView, handleError)
}
```

Fetch a key (RxJava)

```
SimpleStoreFactory.create(context, MY_NAMESPACE).use {
  Single.fromFuture(simpleStore.getString("some_key"))
      .observeOn(uiThread())
      .subscribe(updateView, showError)
}
```

Design Principles

1. Fundamentally async
2. Never emits null
3. Serializes as a `byte[]`
4. Releases all resources when desired
5. Supports configuring based on data type

Fundamentally Async

IO operations are interrupt driven.

Fundamentally Async

Shared thread pool to enqueue IO operations across all callsites

Internally uses `OrderedExecutor`

Why not RxJava?

`Single.fromFuture` instantly goes from ListenableFuture to RxJava.

Never Emits Null

`null` in an Rx-stream or `ListenableFuture` is undesirable

0-length `byte[]` equivalent of the value instead

Serializes to byte[]

Base class only handles `String` and `byte[]`

Allows choosing your own object serialization

Releases Resources

After `close()` is called, SimpleStore releases the in-memory cache.

Releases Resources

When a namespace is closed, resources are freed

Optional LRU mode for in-memory cache

Supports Configurations

Caches

Non-critical

Critical

The Next New Thing

Supports Configurations

When opening a namespace, an optional configuration can be passed.

Used to place on disk

Supports Configurations

Cache

```
SimpleStoreFactory.create(context,
   "<namespace>",
   NamespaceConfig.CACHE)
```

In the future, we can optimize memory use for you.

Supports Configurations

Non-critical

```
SimpleStoreFactory.create(context, "<namespace>")
```

Default mode, good for data that is not on a hot-path.

Supports Configurations

Performance Critical

```
SimpleStoreFactory.create(context,
   "<namespace>",
   NamespaceConfig.CRITICAL)
```

Keeps data in memory as long as the store is open

Supports Configurations

The Next New Thing

Android Q's media scoping can be implemented down the line.

Primitives are in the box

If you want to store more than just `String` and `byte[]`, `PrimitiveSimpleStore` implements basic serialization of primitives for you.

If you're only interested in storing primitives, you can leave the talk now.

More than primitives!

SimpleStore for Protocol Buffers

SimpleStore for Protocol Buffers

Companion artifact of `simplestore-proto` enables storing protoc generated POJOs directly.

SimpleStore for Protocol Buffers

Atomically load complex models from disk.

gRPC users are already using protos for models.

SimpleStore for Protocol Buffers

Uses Google's Lite protoc plugin, contibutions of other runtimes welcome.

```
val proto = Demo.Data.newBuilder().setField(editText.text.toString()).build()
Futures.addCallback(
    simpleStore.put("some_thing", proto),
    object : FutureCallback<Demo.Data> {
      override fun onSuccess(payload: Demo.Data?) {
        editText.setText("")
        button.isEnabled = true
        editText.isEnabled = true
        loadMessage()
      }

      override fun onFailure(t: Throwable) {
        textView.text = t.toString()
        button.isEnabled = true
        editText.isEnabled = true
      }
    },
    mainExecutor()
)
```

Modular & flexible.

Use whatever serialization format you like.

Possible Enhancements

- Co-routine wrapper
- Kotlin native compatible interface
- Automatic cache eviction
- Pipelining within a namespace
- Key auditing & cleanup

Contributions welcome!

Thanks!

Kurt Nelson, Uber Mobile Platform

@kurtisnelson

http://github.com/uber/simple-store