C++ Apps
from Dropbox

Carousel

Mailbox

Check them out on the
Play Store or the App Store

Practical Cross-Platform C++ Development

Alex Allain (alexallain@dropbox.com)
Andrew Twyman (atwyman@dropbox.com)

You?

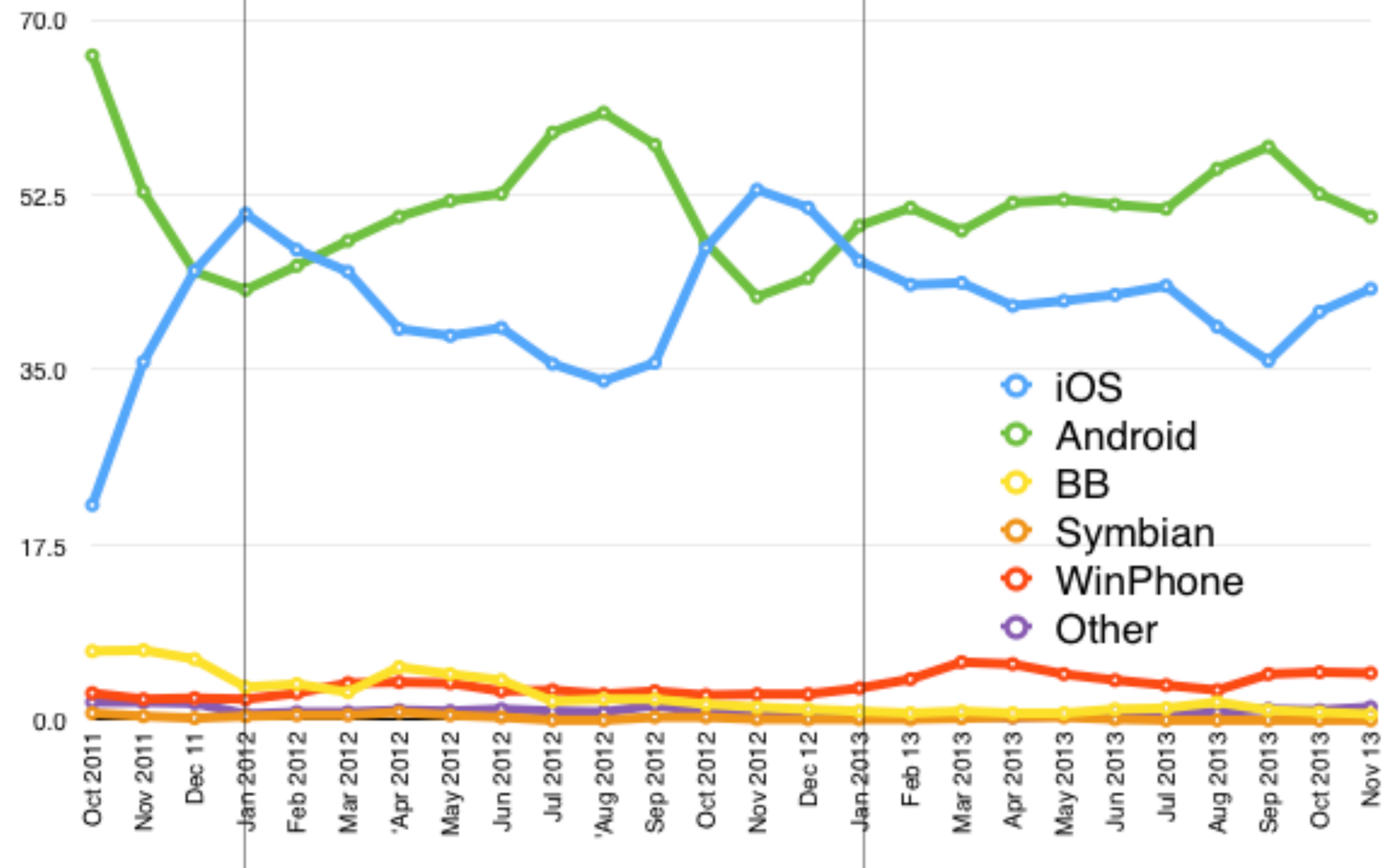
- Android
- iOS
- Other mobile
- Other talks
  - Dropbox + Microsoft

Goal for this talk: arm you to make your own (new) mobile apps

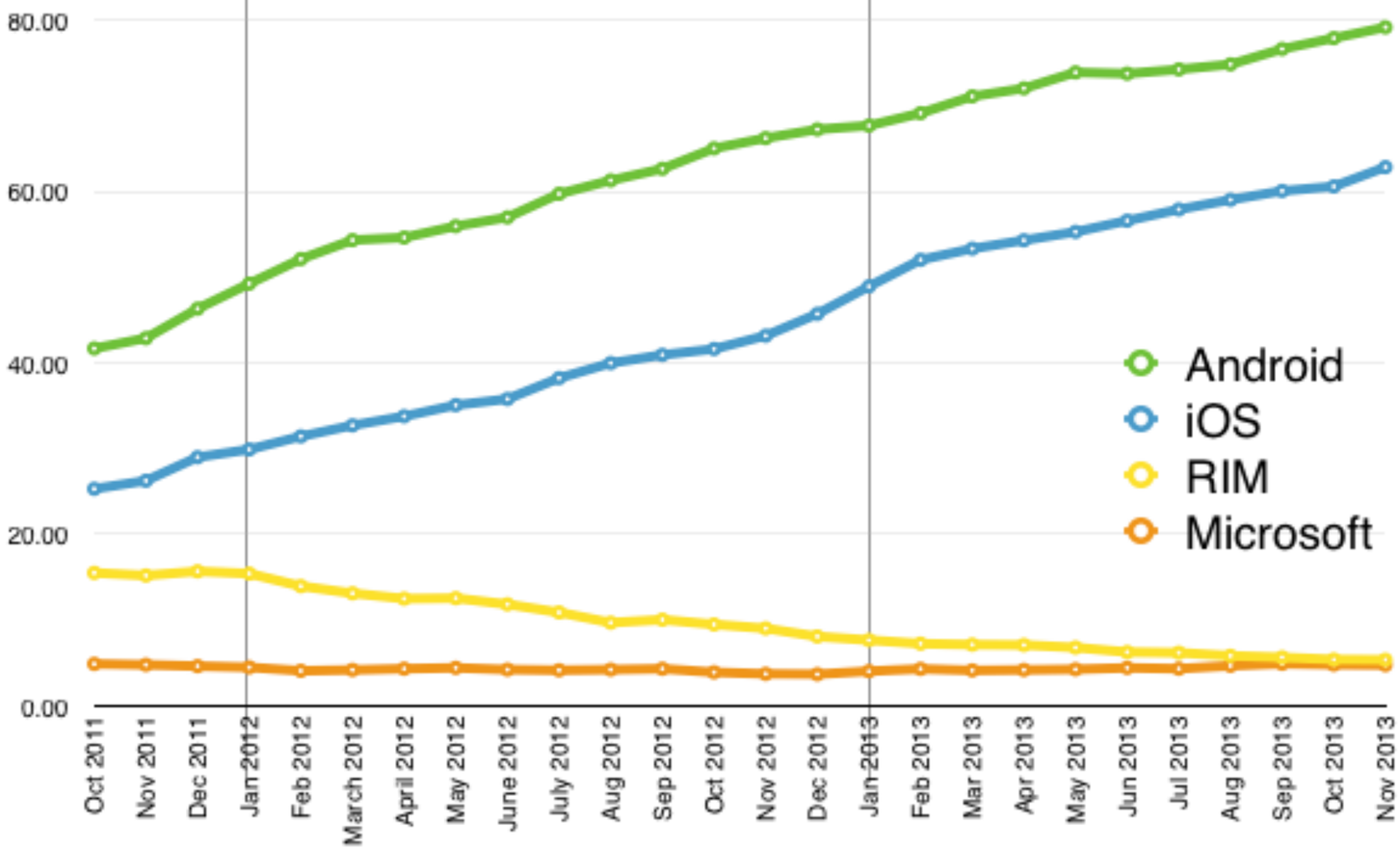
So you want to make an app?

- First, decide if it needs to be cross-platform

US smartphone sales market share, Oct 2011-Nov 2013



US smartphone total installed based Oct 2011 - Nov 2013



Source: Kantar, ComScore

So you want to make an app?

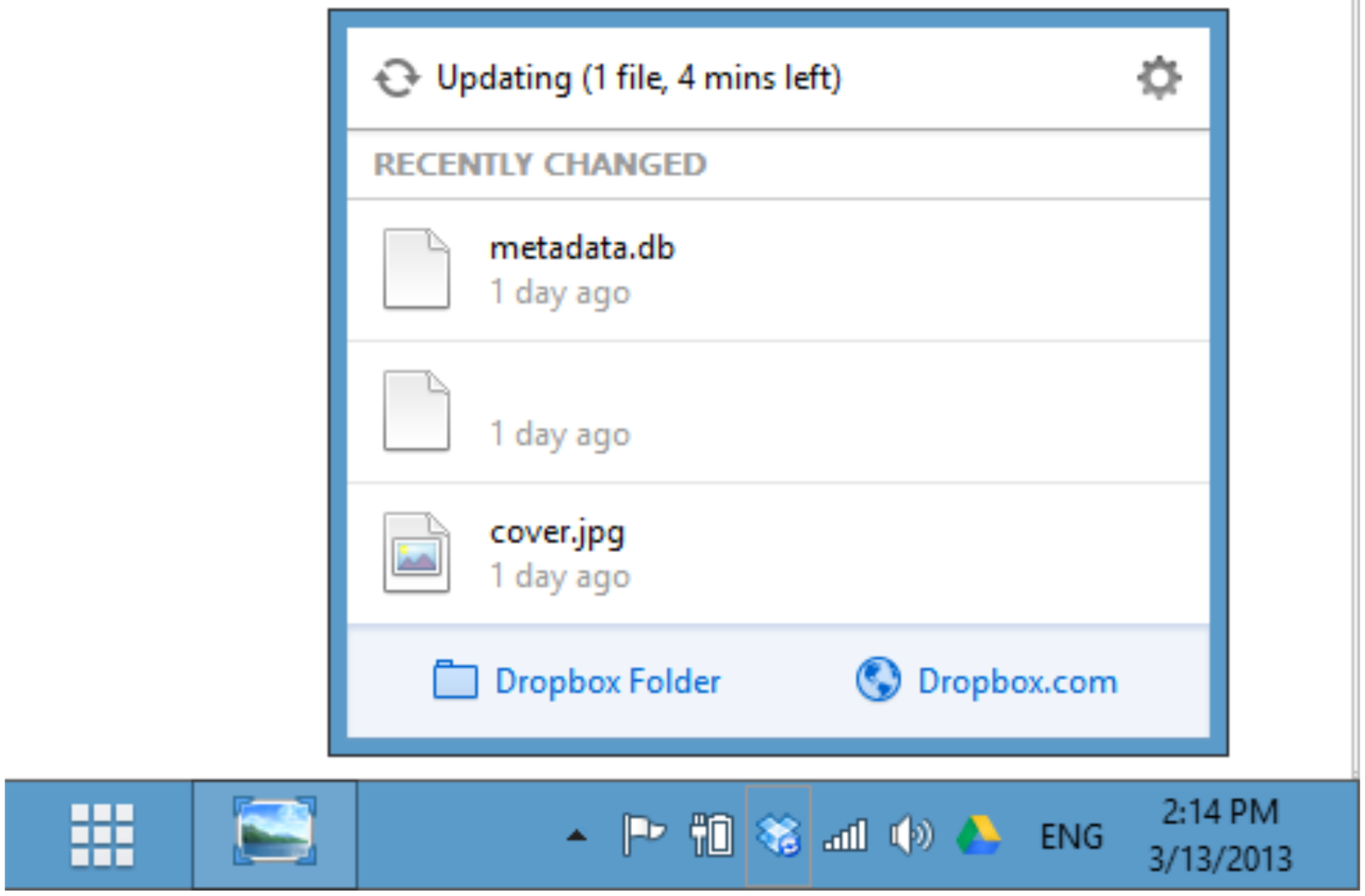
- First, decide if it needs to be cross-platform
- Second, decide your overall approach

Build everything native?

- Dropbox did this

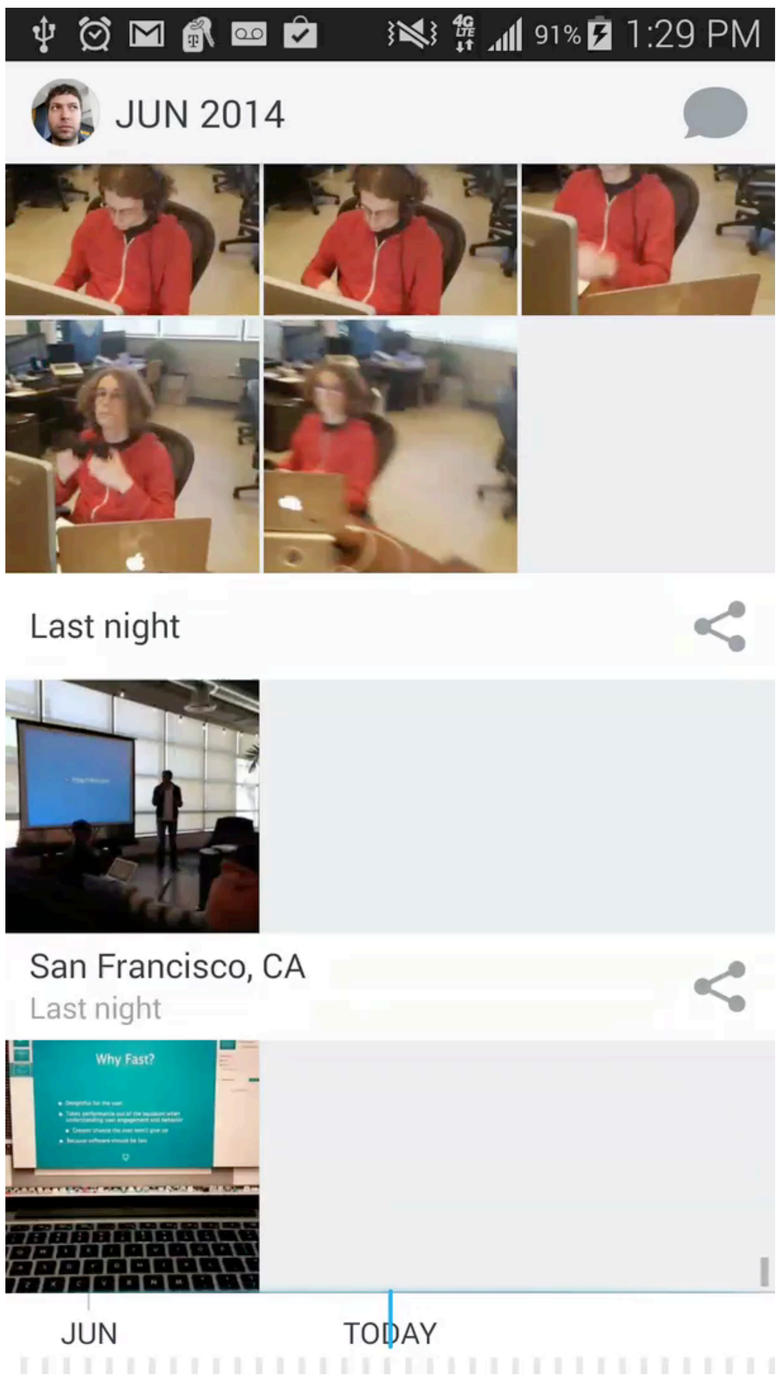
Build it entirely custom?

- Dropbox did this



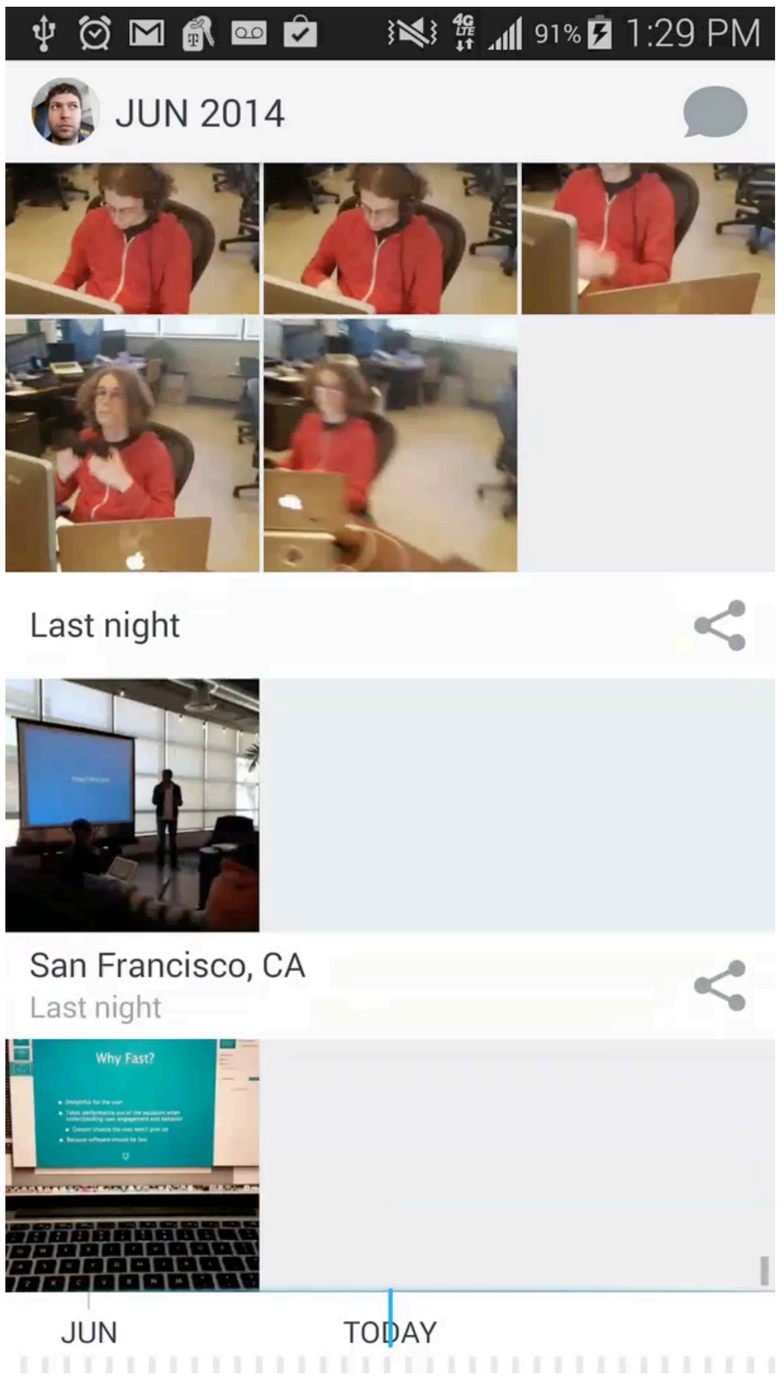
Native UI, xplatform core

- Dropbox did this



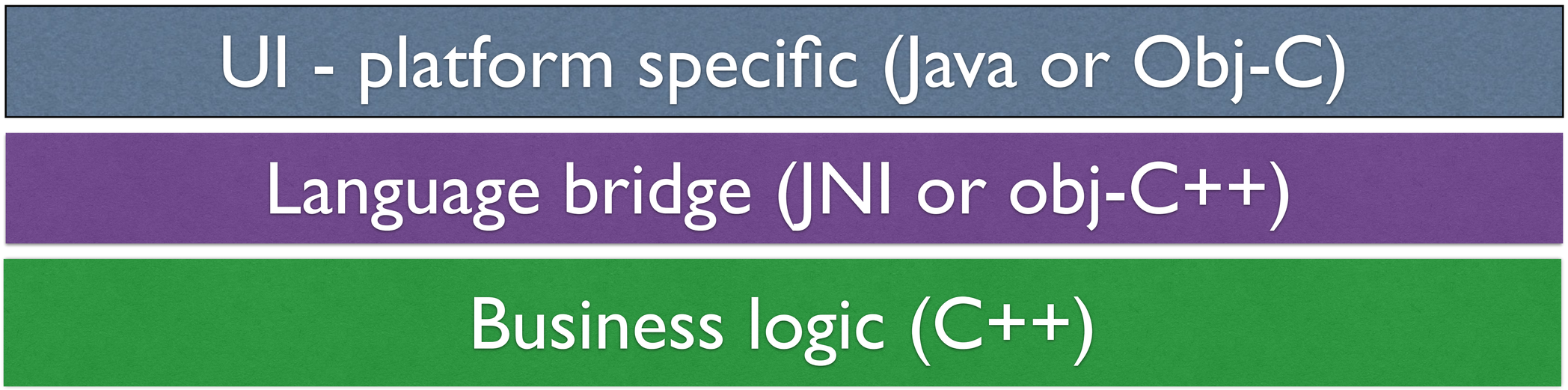
Native UI, xplatform core

- Dropbox did this
- Everything else in this talk follows from this design and discusses its consequences



So you want to a native UI but not write everything twice…

So you want to a native UI but not write everything twice…



Expanding that out…

- UI view layer
- **Language bridge**
- UI ViewModel
- Business logic library
- Libraries
  - **Networking**
  - **Threading**
  - Storage & caching
  - 3rd party libraries
  - **Specialized platform features**
- Basic environment
  - **C++ language features & standard library**
  - Build system

So you want specialized platform features?

So you want specialized platform features?

UI - platform specific

Platform services

Language bridge (JNI or Obj-C++)

Business logic (C++)

Principles of a good UI layer

- Keep it thin
- Follow native UI conventions and patterns
  - Native look and feel
  - Idiomatic code
- Call into library layer for logic
  - ViewModel in C++

Principles of your language bridge

- Keep it thin
  - Hard to get right
  - Filled with boilerplate – hides real logic
- Provide consistent interface across languages
  - Use types available in all languages
- You can auto-generate this

Sync progress…

- Motivation
- Cross-Platform Architecture
- **C++ Language Features & Standard Library**
- Networking & Threading
- Language Bridge
- Introducing Djinni
- Wrap Up

Our Situation

- We want to move fast
  - But sweat the details – can't lose data
- We're building (mostly) new apps
  - Not dealing with lots of legacy code
  - Not dealing with (very) old platforms
- We're building multiple apps
  - Benefits to shareable libraries

Setting the Baseline

- Platform support for C++ is uneven
- Different compiler support for C++11/14
- Different STL implementations
- Different bugs
- Different strict warnings

Our Approach

- Embrace the new
  - Newest **stable** tools we can obtain
  - Any language feature all compilers support
- Raise the baseline
  - We can’t raise the language
  - But we can fill gaps in the libraries
- Write for portability, refactor as necessary
  - e.g. fixed-width integers, UTF-8 strings

Our Tools

- **iOS & OS X:** clang 3.5, libc++ (XCode 5)
- **Android:** gcc 4.8, libstdc++ (NDK r9d)
- **OS X/Linux Python:** Either of the above
- **Windows (in progress):** Visual Studio 2013 or “14” (preview)

Raising the Libraries

- Implement missing (or broken) pieces
  - `optional<T>` (sadly not standard)
  - `make_unique` (until C++14)
  - `thread_local` (not on iOS)
  - `to_string` (not on Android)
  - `from_string` (`istringstream` hangs on Android)
- Extend with our own libraries
  - `run_on_exit`
  - `printf` into `std::string`
  - etc.

Sync progress…

- Motivation
- Cross-Platform Architecture
- C++ Language Features & Standard Library
- **Networking & Threading**
- Language Bridge
- Introducing Djinni
- Wrap Up

Networking & HTTP

- What layer should handle network I/O?
  - Use the frameworks on the platform?
  - Use a portable C++ libraries?
- Our apps have made different choices

C++ Libraries

- Good, portable libraries are available
  - websockets, libcurl, OpenSSL, etc.
- Benefits
  - Consistent features (and bugs) across platforms
  - High performance (no marshaling)
- But it’s up to you get all the details right

Platform Frameworks

- Mobile platforms have strong networking libraries
- Benefits
  - Security and feature parity with other apps
  - Lots of features for free
    - Standard: connection reuse, proxies
    - Specialized: background fetch, online/offline
  - No need to update
- But you're subject to the vagaries of the platform

HTTP as a Platform Service

- C++ code uses HttpRequestor class
  - Specialized to Dropbox API features
  - Synchronous interface for background threads
    - Uses an abstract interface to do raw HTTP
- Platform code implements the interface
  - Java: `HttpsUrlConnection`
  - Objective C: `NSURLConnection/NSURLSession`
  - Adapter code translates any differences

Background Threads

- How to manage background threads for libraries?
- We want C++ code to own thread lifetime
- Frameworks don't like threads they didn't create
  - Java: JNI state, class loader issues
  - Python: issues with thread-local storage
  - General: inconsistent debugging

Threads as a Platform Service

- Platform provides thread creation interface
  - Creates thread with name and priority
  - Calls back to C++ via a given lambda
- After creation, C++ owns the thread
  - Communication via `mutex/condition`
  - Termination when C++ calls `shutdown()`

Sync progress…

- Motivation
- Cross-Platform Architecture
- C++ Language Features & Standard Library
- Networking & Threading
- **Language Bridge**
- Introducing Djinni
- Wrap Up

Bridging to Java

- JNI = Java Native Interface
- Links Java to C or C++
- Designed for isolated calls across the boundary
  - CPU-intensive algorithms
  - Specialized hardware libraries
- Not only on Android

It gets ugly…

- Impedance mismatch
  - Requires marshaling of data
  - Lots of manual steps
  - Manual resource management
- No type safety 😱
  - Call C++ via magic names
  - Call Java via reflection
  - Hope you pass the right number of args

```
public class JniDemo {
    static {
        System.loadLibrary("JniDemo");
    }

    public static native int demo(int arg);
}
```

```
extern "C" JNIEXPORT jint JNICALL
Java_com_dropbox_example_JniDemo_demo(
        JNIEnv * env,
        jclass clazz,
        jint arg)
{
    return JniDemo::demo(arg);
}
```

```
extern "C" JNIEXPORT jint JNICALL
Java_com_dropbox_example_JniDemo_demo(
        JNIEnv * env,
        jobject thiz,
        jint arg)
{
    jclass my_class = env->GetObjectClass(thiz);
    jmethodID my_meth = env->GetMethodID(my_class, "getCpp", “()J");
    jlong cpp_long = env->CallLongMethod(thiz, my_method, arg);
    JniDemo * cpp_obj = reinterpret_cast<JniDemo*>(cpp_long);

    return cpp_obj->demo(arg);
}
```

```
extern "C" JNIEXPORT jstring JNICALL
Java_com_dropbox_example_JniDemo_demo(
        JNIEnv * env,
        jobject thiz,
        jstring arg)
{
    jclass my_class = env->GetObjectClass(thiz);
    jmethodID my_meth = env->GetMethodID(my_class, "getCpp", “()J");
    jlong cpp_long = env->CallLongMethod(thiz, my_method, arg);
    JniDemo * cpp_obj = reinterpret_cast<JniDemo*>(cpp_long);

    const char * arg_cstr = env->GetStringUTFChars(arg, nullptr);

    std::string result = cpp_obj->demo(arg_cstr);

    env->ReleaseStringUTFChars(arg, arg_cstr);
    jstring jresult = env->NewStringUTF(result.c_str());
    return jresult;
}
```

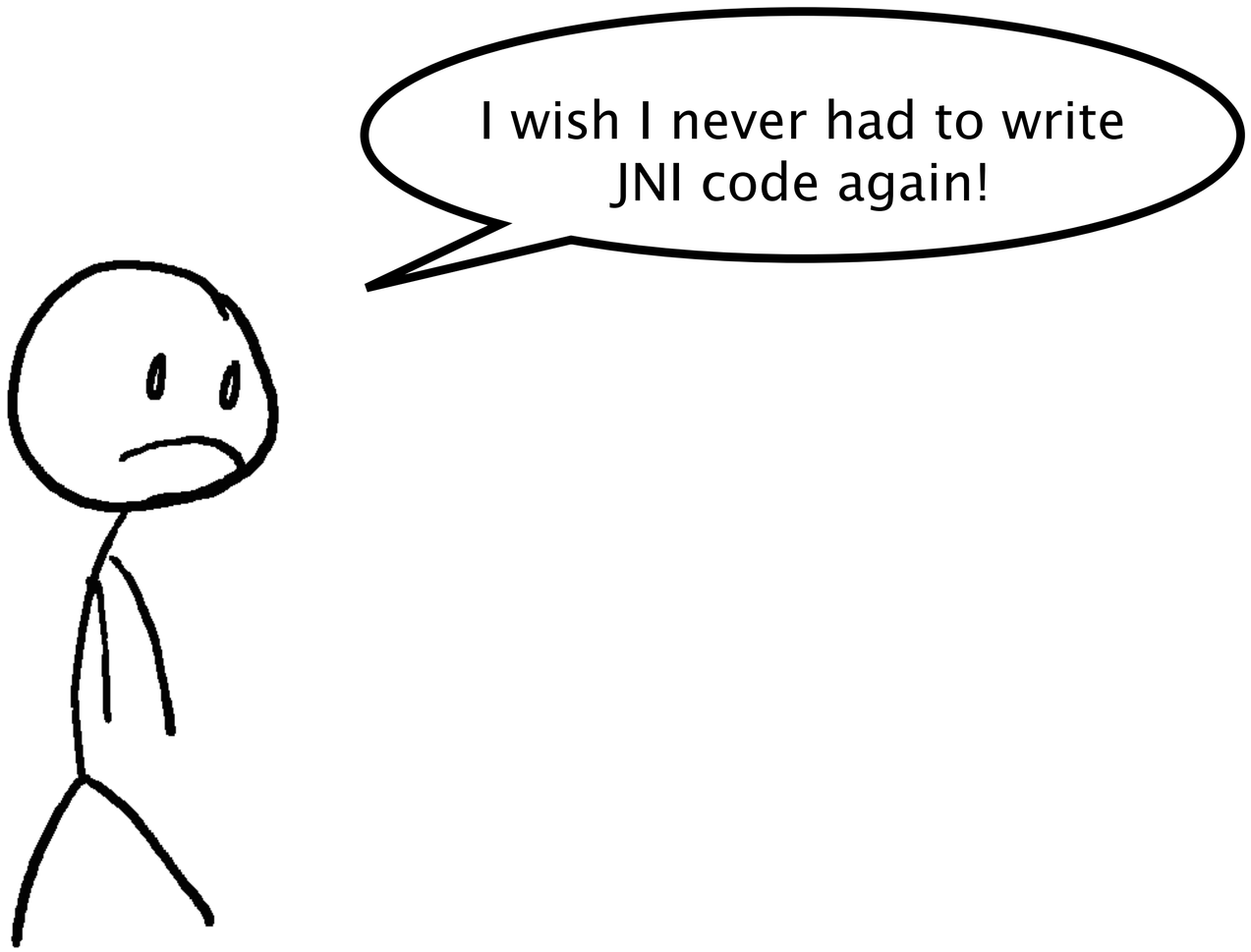
```
extern "C" JNIEXPORT jstring JNICALL
Java_com_dropbox_example_JniDemo_demo(
        JNIEnv * env,
        jobject thiz,
        jstring arg)
{
    jclass my_class = env->GetObjectClass(thiz);
    jmethodID my_meth = env->GetMethodID(my_class, "getCpp", “()J");
    jlong cpp_long = env->CallLongMethod(thiz, my_method, arg);
    JniDemo * cpp_obj = reinterpret_cast<JniDemo*>(cpp_long);

    jsize length = env->GetStringLength(arg);
    const char16_t * arg_chars = reinterpret_cast<const char16_t*>(
        env->GetStringChars(arg, nullptr));
    std::u16string arg_u16(arg_chars, length):
    std::string cpp_arg = miniutf::to_utf8(arg_u16);
    env->ReleaseStringChars(arg, arg_chars);

    std::string result = cpp_obj->demo(cpp_arg);

    std::u16string result_u16 = miniutf::to_utf16(result);
    const jchar * result_jchars =
        reinterpret_cast<const jchar*>(result_u16.data());
    jstring result = env->NewString(result_jchars, result_u16.length());
    return jresult;
}
```

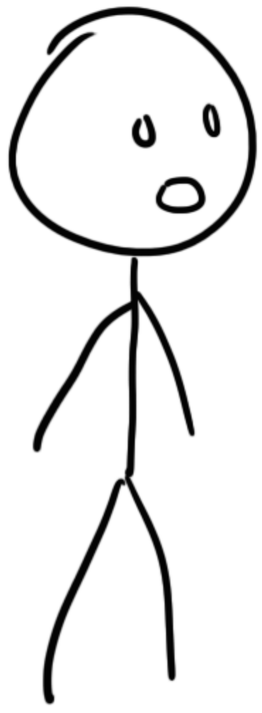
```
Java_com_dropbox_example_JniDemo_demo(
        JNIEnv * env,
        jobject thiz,
        jstring arg)
{
    jclass my_class = env->GetObjectClass(thiz);
    if (env->ExceptionCheck())
        return nullptr; // Arbitrary value!
    jmethodID my_meth = env->GetMethodID(my_class, "getCpp", “()J");
    if (env->ExceptionCheck())
        return nullptr; // Arbitrary value!
    jlong cpp_long = env->CallLongMethod(thiz, my_method, arg);
    if (env->ExceptionCheck())
        return nullptr; // Arbitrary value!
    JniDemo * cpp_obj = reinterpret_cast<JniDemo*>(cpp_long);

    jsize length = env->GetStringLength(arg);
    const char16_t * arg_chars = reinterpret_cast<const char16_t*>(
        env->GetStringChars(arg, nullptr));
    std::u16string arg_u16(arg_chars, length):
    std::string cpp_arg = miniutf::to_utf8(arg_u16);
    env->ReleaseStringChars(arg, arg_chars);

    std::string result = cpp_obj->demo(cpp_arg);

    std::u16string result_u16 = miniutf::to_utf16(result);
    const jchar * result_jchars =
        reinterpret_cast<const jchar*>(result_u16.data());
    jstring result = env->NewString(result_jchars, result_u16.length());
    if (env->ExceptionCheck())
        return nullptr; // Arbitrary value!
    return jresult;
}
```

```
    return nullptr; // Arbitrary value!
jmethodID my_meth = env->GetMethodID(my_class, "getCpp", “()J");
if (env->ExceptionCheck())
    return nullptr; // Arbitrary value!
jlong cpp_long = env->CallLongMethod(thiz, my_method, arg);
if (env->ExceptionCheck())
    return nullptr; // Arbitrary value!
JniDemo * cpp_obj = reinterpret_cast<JniDemo*>(cpp_long);

jsize length = env->GetStringLength(arg);
const char16_t * arg_chars = reinterpret_cast<const char16_t*>(
    env->GetStringChars(arg, nullptr));
try {
    std::u16string arg_u16(arg_chars, length):
    std::string cpp_arg = miniutf::to_utf8(arg_u16);
    env->ReleaseStringChars(arg, arg_chars);

    std::string result = cpp_obj->demo(cpp_arg);

    std::u16string result_u16 = miniutf::to_utf16(result);
    const jchar * result_jchars =
        reinterpret_cast<const jchar*>(result_u16.data());
    jstring result = env->NewString(result_jchars, result_u16.length());
    if (env->ExceptionCheck())
        return nullptr; // Arbitrary value!
    return jresult;
  } catch (const std::exception& e) {
    env->ReleaseStringChars(arg, arg_chars);
    jclass runtime_ex = env->FindClass("java/lang/RuntimeException");
    env->ThrowNew(runtime_ex, e.what());
    return nullptr; // Arbitrary value!
  }
}
```

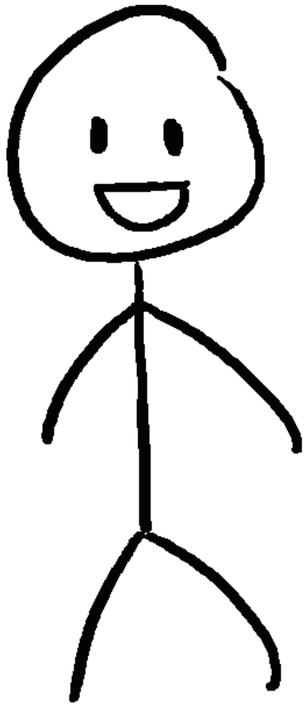
What's my point?

- JNI is tedious and error-prone
  - Even when doing simple things
- Lots of details means lots of mistakes
  - Forget to check errors
  - Forget to free resources
  - Wrong number of arguments
- C++ can make it simpler (somewhat)
  - e.g. RAII for resources






Granted!



```
jni_demo = interface +c {
    demo(arg: string): string;
}
```

Bridging to Objective-C

- Objective-C++
  - Unofficial overlay of 2 languages
- Lots of power without much code
  - Direct composition: objects from both languages can own each other
  - Use your favorite techniques from either language

Objective-C++ is Complex

- 2 types of exceptions
- 2 types ref-counting
- 3 types of bugs: know 2 languages to fix them
- So we limit Objective-C++ to the boundary

```objc
@interface ObjCppDemo : NSObject

- (id)initWithInteger:(NSInteger)i;
- (NSInteger)demoWithInteger:(NSInteger)arg;
- (NSString *)demoWithString:(NSString *)arg;

@end
```

```objc
- (NSInteger)demoWithInteger:(NSInteger)arg {
    return _cppObj.demo(arg);
}
```

```objc
- (NSString *)demoWithString:(NSString *)arg {
    std::string arg_cpp([arg UTF8String]);
    string result = _cppObj.demo(arg_cpp);
    return [[NSString alloc] initWithUTF8String:result.c_str()];
}
```

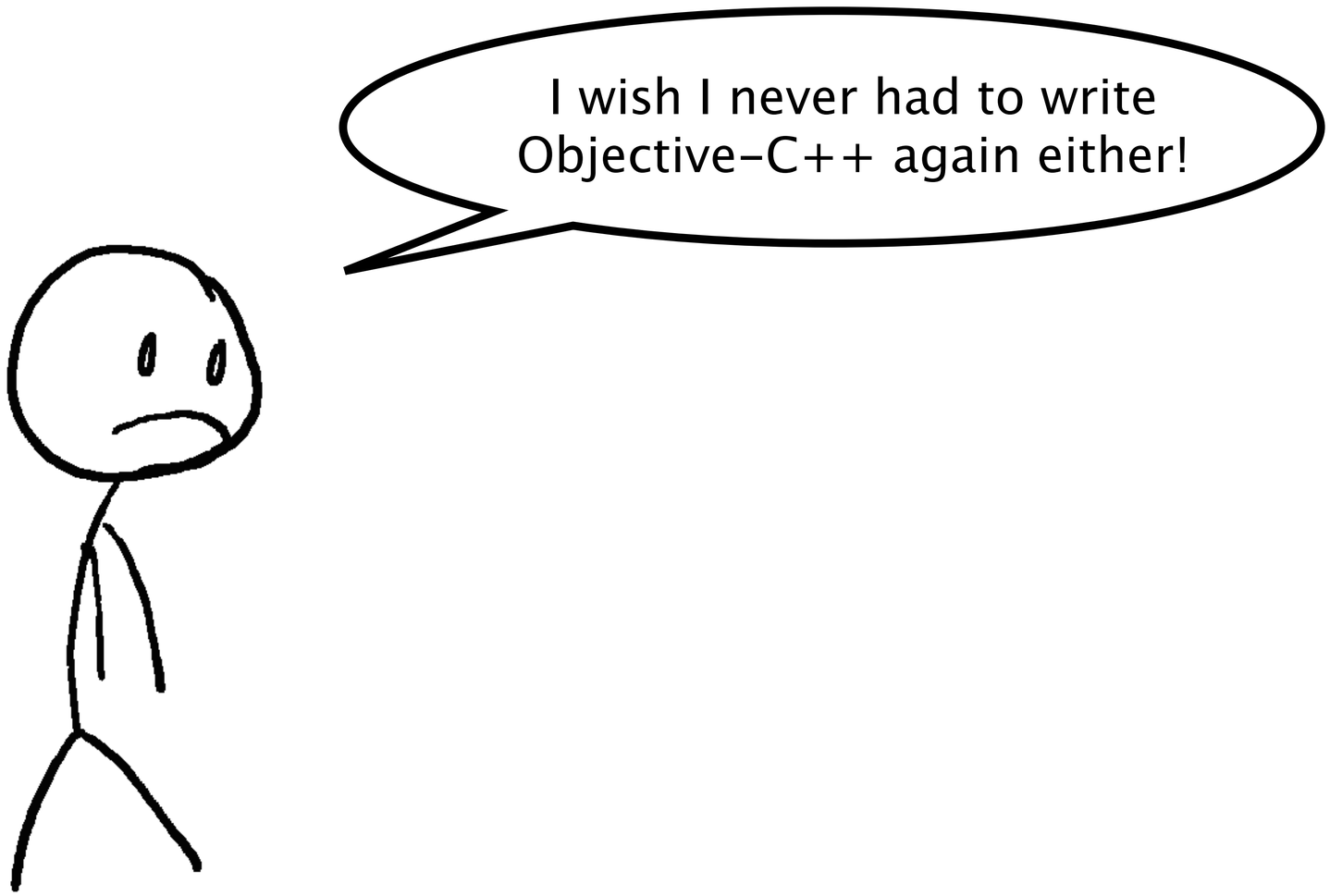
```
- (NSString *)demoWithString:(NSString *)arg {
    std::string arg_cpp(
        [arg UTF8String],
        [arg lengthOfBytesUsingEncoding:NSUTF8StringEncoding]);
    string result = _cppObj.demo(arg_cpp);
    return [[NSString alloc] initWithBytes:result.data()
                                    length:result.length()
                                  encoding:NSUTF8StringEncoding];
}
```

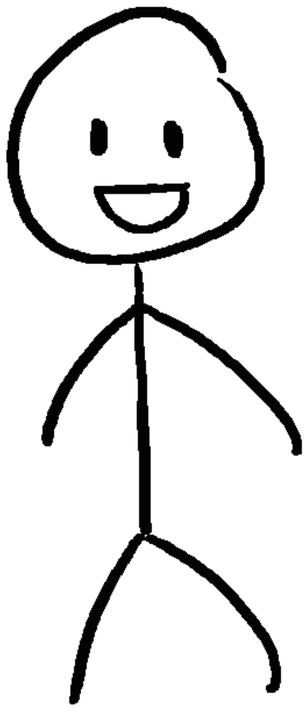
```objc
- (NSString *)demoWithString:(NSString *)arg {
    try {
        std::string arg_cpp(
            [arg UTF8String],
            [arg lengthOfBytesUsingEncoding:NSUTF8StringEncoding]);
        string result = _cppObj.demo(arg_cpp);
        return [[NSString alloc] initWithBytes:result.data()
                                        length:result.length()
                                      encoding:NSUTF8StringEncoding];
    } catch (const std::exception& e) {
        [NSException raise:@"DemoException" format:@"%s", e.what()];
        __builtin_unreachable();
    }
}
```

```objc
- (NSString *)demoWithString:(NSString *)arg error:(NSError **)error {
    try {
        std::string arg_cpp(
            [arg UTF8String],
            [arg lengthOfBytesUsingEncoding:NSUTF8StringEncoding]);
        string result = _cppObj.demo(arg_cpp);
        return [[NSString alloc] initWithBytes:result.data()
                                        length:result.length()
                                      encoding:NSUTF8StringEncoding];
    } catch (const std::exception& e) {
        if (is_fatal(e)) {
            [NSException raise:@"DemoException" format:@"%s", e.what()];
            __builtin_unreachable();
        }
        if (error) {
            NSString * what = [NSString stringWithUTF8String:e.what()];
            *error = [NSError errorWithDomain:@"DemoErrorDoman"
                                         code:42
                                     userInfo:@{@"what": what}];
        }
        return nil; // Defined error return value.
    }
}
```

What's my point?

- Easier than JNI
  - Objective-C++ allows code to overlap
  - Memory model is more compatible
- But complex
  - Worry about the semantics of 2 languages
  - Lots of tedious wrapper functions






```
jni_demo = interface +c {
    demo(arg: string): string;
}
```

Sync progress…

- Motivation
- Cross-Platform Architecture
- C++ Language Features & Standard Library
- Networking & Threading
- Language Bridge
- **Introducing Djinni**
- Wrap Up

What is Djinni?

- Pronounced “genie”
  - Derived (loosely) from “Dropbox JNI”
- Generates bridge code between languages
- Currently supports
  - Java <-> C++ (via JNI)
  - Obj-C <-> C++ (via Obj-C++)

What does Djinni do?

- You define your interface (using a custom IDL)
- Djinni generates bridge code automatically
  - Java proxy class & JNI marshaling
  - Obj-C interface and Obj-C++ marshaling
  - C++ abstract base class
- You provide the concrete implementation

Djinni IDL

- Interfaces: like abstract base classes
  - Can be implemented in any language
  - Call both ways
- Records: like structs
  - Value types containing pure data
  - Can contain primitive types, other records
- Enumerations: like scoped enum
  - auto-generated in all languages

Primitive Types

- `bool`
- Fixed-Width Integers: `i8`, `i16`, `i32`, `i64`
- Double-Precision Floating-Point: `f64`
- `string`
- `binary`
- `list<T>`, `set<T>`, `map<K,V>`
- `optional<T>`

Other Features

- static methods
- constants
- auto-generated comparators for records
- room for custom extensions
- interface comments in output
- configurable naming conventions per language
  - foo_bar, fooBar, FooBar, mFooBar, etc.
  - C++ namespace / Java package names

```
# An enum for specialized use.
wish_difficulty = enum {
    easy;
    medium;
    hard;
}

# A simple data struct.
wish = record {
    difficulty: wish_difficulty;
    request: string;
} deriving (eq, ord)
```

```
# A platform service, called by C++
djinni = interface +j +o {
    const max_wishes: i32 = 3;

    grant_wish(my_wish: wish): bool;
    past_wishes(): set<wish>;

    # Factory method
    static rub_lamp(): djinni;
}
```

Why this approach?

- Why not extract from source-code (e.g. SWIG)?
- Why not serialized RPC?
- Clearly defined interface
  - Looks native on all sides
- Clear set of supported types and operations
  - Intersection of what all languages can support
- Explicitly language-agnostic on both sides
  - Can re-target the back-end too

Example Output

- How are we doing for time?
- Let's look at what's generated from a real example
- Focus on what you interact with (not the glue)

```
photo = record {
    # server id of the photo
    id: i64;

    filename: string;
}

photo_model = interface +c {
    # returns the currently sync'd photos
    get_photos(): list<photo>;
}
```

photo.hpp

```
#pragma once

#include <cstdint>
#include <string>
#include <utility>

struct Photo final {
    /** server id of the photo */
    int64_t id;

    std::string filename;

    Photo(
          int64_t id,
          std::string filename) :
    id(std::move(id)),
    filename(std::move(filename)) {
    }
};
```

photo_model.hpp

```
#pragma once

#include "photo.hpp"
#include <vector>

class PhotoModel {
public:
    virtual ~PhotoModel() {}

    /** returns the currently sync'd photos */
    virtual std::vector<Photo> get_photos() = 0;
};
```

Photo.java

```
package com.dropbox.carousel;

import com.dropbox.djinni.DjinniGenerated;

@DjinniGenerated
public final class Photo {

    /*package*/ final long mId;

    /*package*/ final String mFilename;

    public Photo(
                long id,
                String filename) {
        this.mId = id;
        this.mFilename = filename;
    }

    /** server id of the photo */
    public long getId() {
        return mId;
    }

    public String getFilename() {
        return mFilename;
    }
}
```

PhotoModel.java (Part 2)

```
package com.dropbox.carousel;

import com.dropbox.djinni.DjinniGenerated;
import java.util.ArrayList;
import java.util.concurrent.atomic.AtomicBoolean;


@DjinniGenerated
public abstract class PhotoModel {
    /** returns the currently sync'd photos */
    public abstract ArrayList<Photo> getPhotos();

    @DjinniGenerated
    public static final class NativeProxy extends PhotoModel
    {
        private final long nativeRef;
        private final AtomicBoolean destroyed =
                new AtomicBoolean(false);

        /** bridging implementation elided */
    }
}
```

DBPhoto.h

```objc
#import <Foundation/Foundation.h>

@interface DBPhoto : NSObject

/** server id of the photo */
@property (nonatomic, readonly) int64_t id;

@property (nonatomic, readonly) NSString *filename;

@end
```

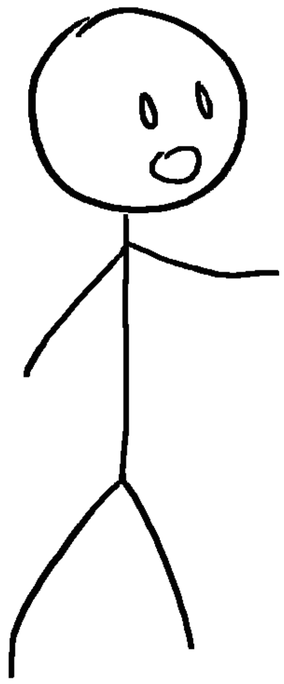
DBPhotoModel.h

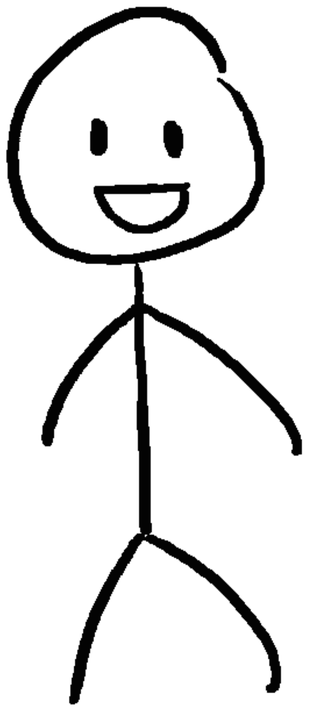
```
#import <Foundation/Foundation.h>
@class DBPhoto;

@protocol DBPhotoModel

/** returns the currently sync'd photos */
- (NSMutableArray *)getPhotos;

@end
```

Other Generated Code

- C++: Nothing
- Java:
  - Implementation of `NativeProxy`
  - `NativePhoto.hpp, NativePhoto.cpp`
  - `NativePhotoModel.hpp, NativePhotoModel.cpp`
- Objective-C:
  - `DBPhoto+Private.h, DBPhoto.mm`
  - `DBPhotoModelCppProxy.h`
  - `DBPhotoModelCppProxy+Private.h`
  - `DBPhotoModelCppProxy.mm`

Djinni, I wish you were free!





github.com/dropbox/djinni

Try this out!

- Djinni is now open-source
  - `github.com/dropbox/djinni`
  - Apache 2.0 license
  - Includes sample app with cross-platform build (using gyp)
- In beta
  - But we're shipping products with it
  - Please send us feedback
  - Help make it better

Sync progress…

- Motivation
- Cross-Platform Architecture
- C++ Language Features & Standard Library
- Networking & Threading
- Language Bridge
- Introducing Djinni
- **Wrap Up**

Conclusions

- Multi-platform is a reality in mobile (& desktop)
- Cross-platform has worked for us
  - Minimize duplication
  - Link native UI with shared logic
- C++11/14 is up for the challenge
  - Need to push the envelope on each platform
- Hopefully we gave you a place to start
  - Djinni can help with the tedious stuff

Future Work

- We’re far from done
- More platforms: Windows is next
- Bridge generation for more languages
- More platform features available to C++
- More tooling to make it easier
- Build out more application building blocks

- Our goal: zero cost for cross-platform dev

More Info

- These slides: bit.ly/djinnitalk
- Djinni: github.com/dropbox/djinni
- Other open source: github.com/dropbox
- Blog post coming soon: tech.dropbox.com
- Email us:
  - alexallain@dropbox.com
  - atwyman@dropbox.com
  - j4cbo@dropbox.com
- Sound fun? We <3 C++ and people who <3 C++.
  Email alexallain@dropbox.com