Cross-compiling C++ to JavaScript

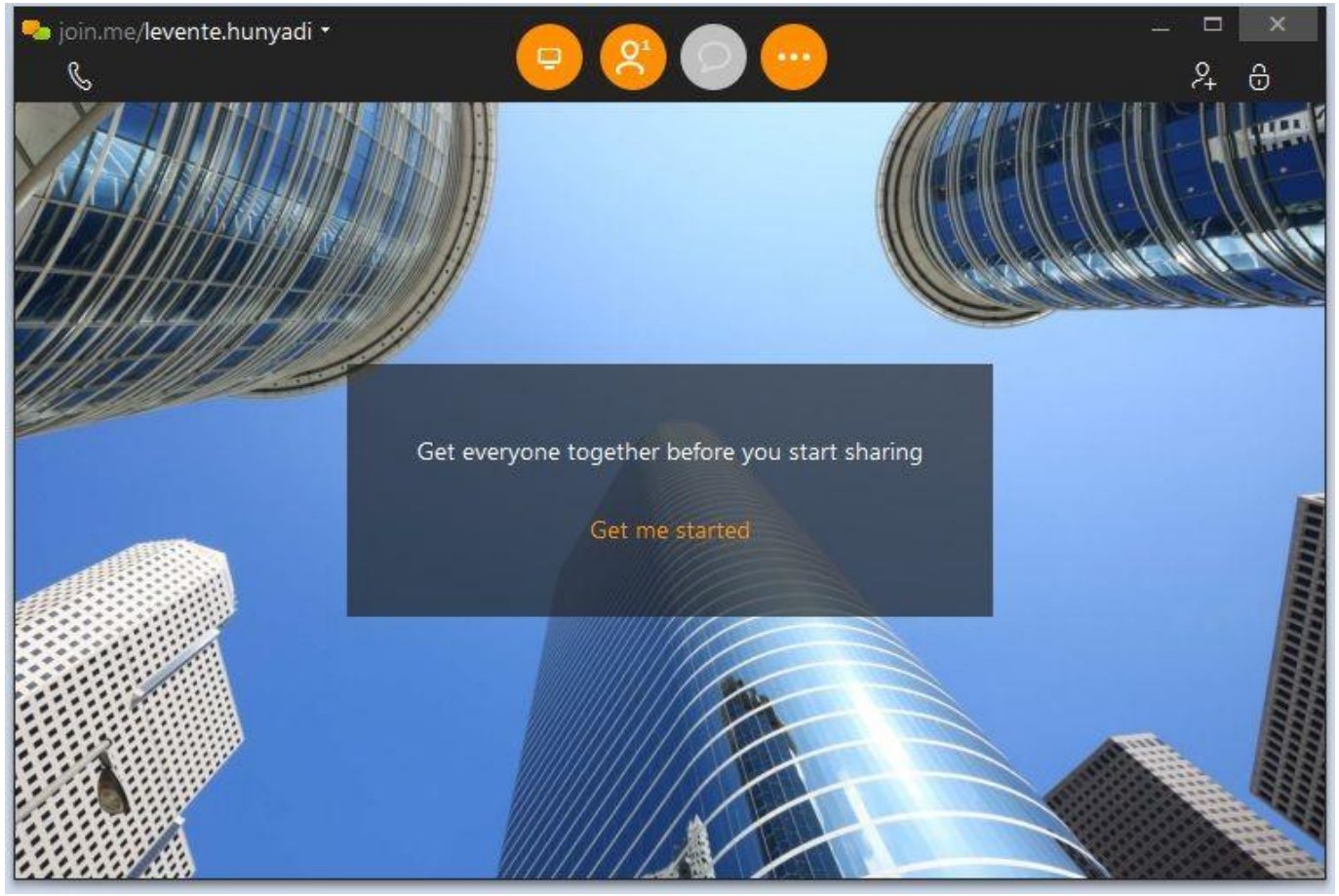
Challenges in porting the join.me common library to HTML5

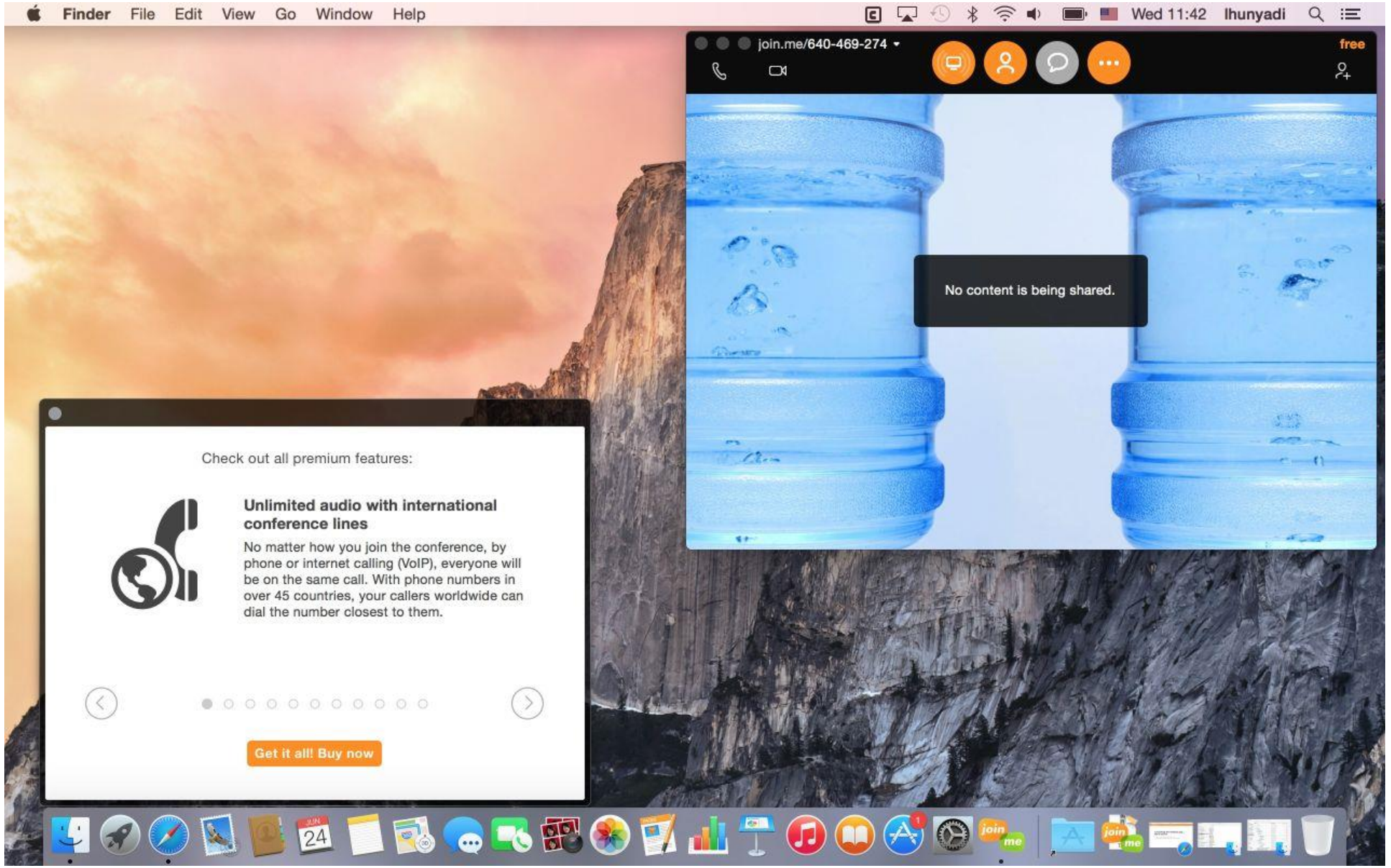
JUNE 24, 2015

LEVENTE HUNYADI

LogMeIn

join.me at a glance





join.me characteristics

Application deployed to several platforms

- Windows, Mac, Android, iOS, etc.

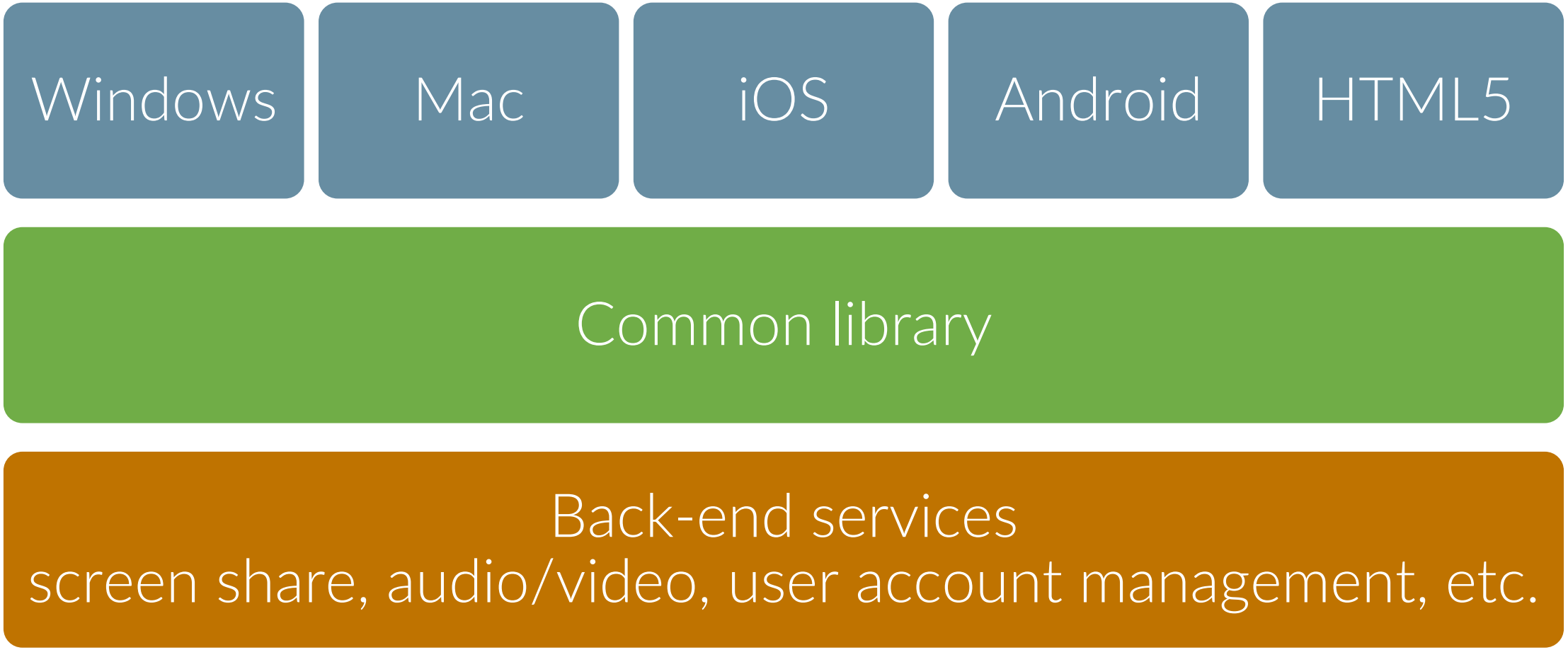
Native UI that looks natural

- Utilizes native functionality exposed by the platform

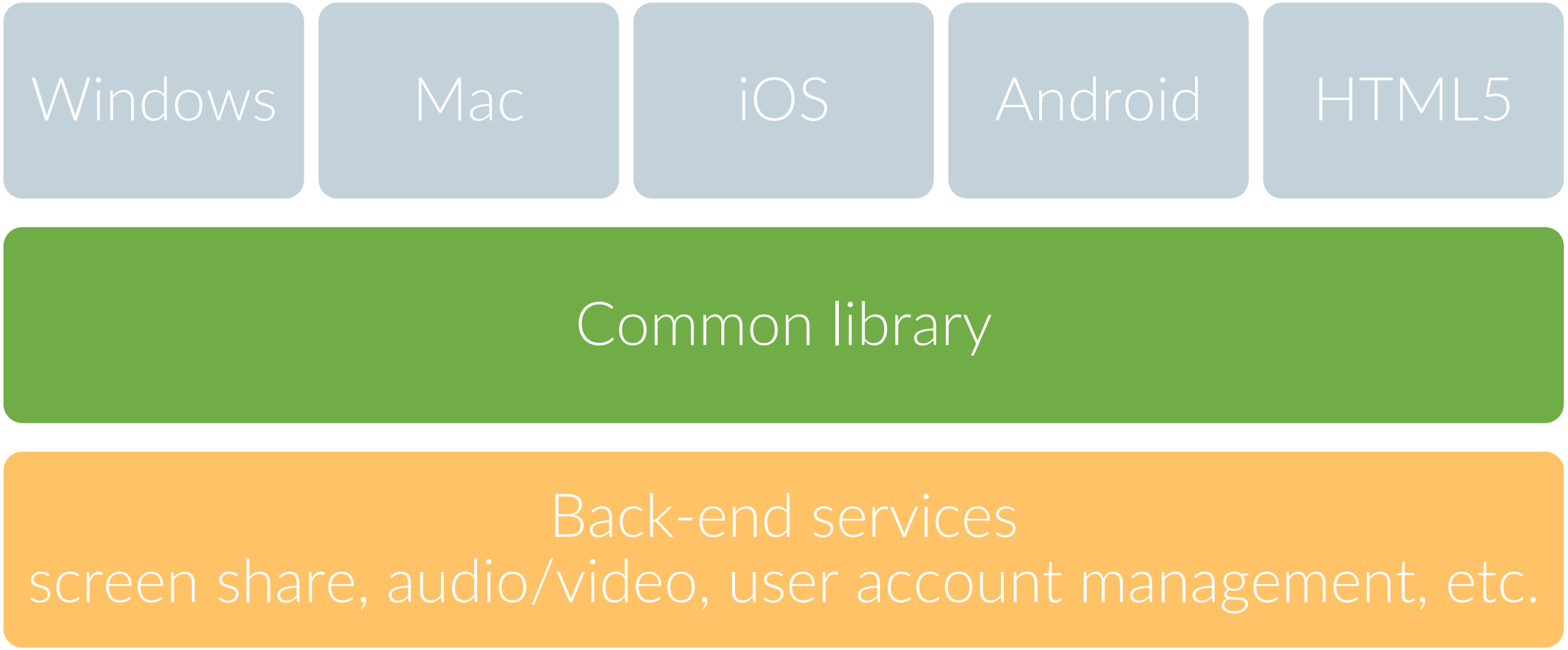
Common functionality in a single library shared across platforms

- Encapsulates application logic for sharing a screen or window, scheduling a meeting, using audio and video, etc.

Simplified architectural outline



Simplified architectural outline



C++ as the choice of language

Cross-platform

- Integrates into Windows, Mac, Android, iOS, etc. (with appropriate wrappers)

Speeds up development

- Develop and maintain a single code base, deploy to several platforms

Porting the join.me common library to HTML5

LogMeIn

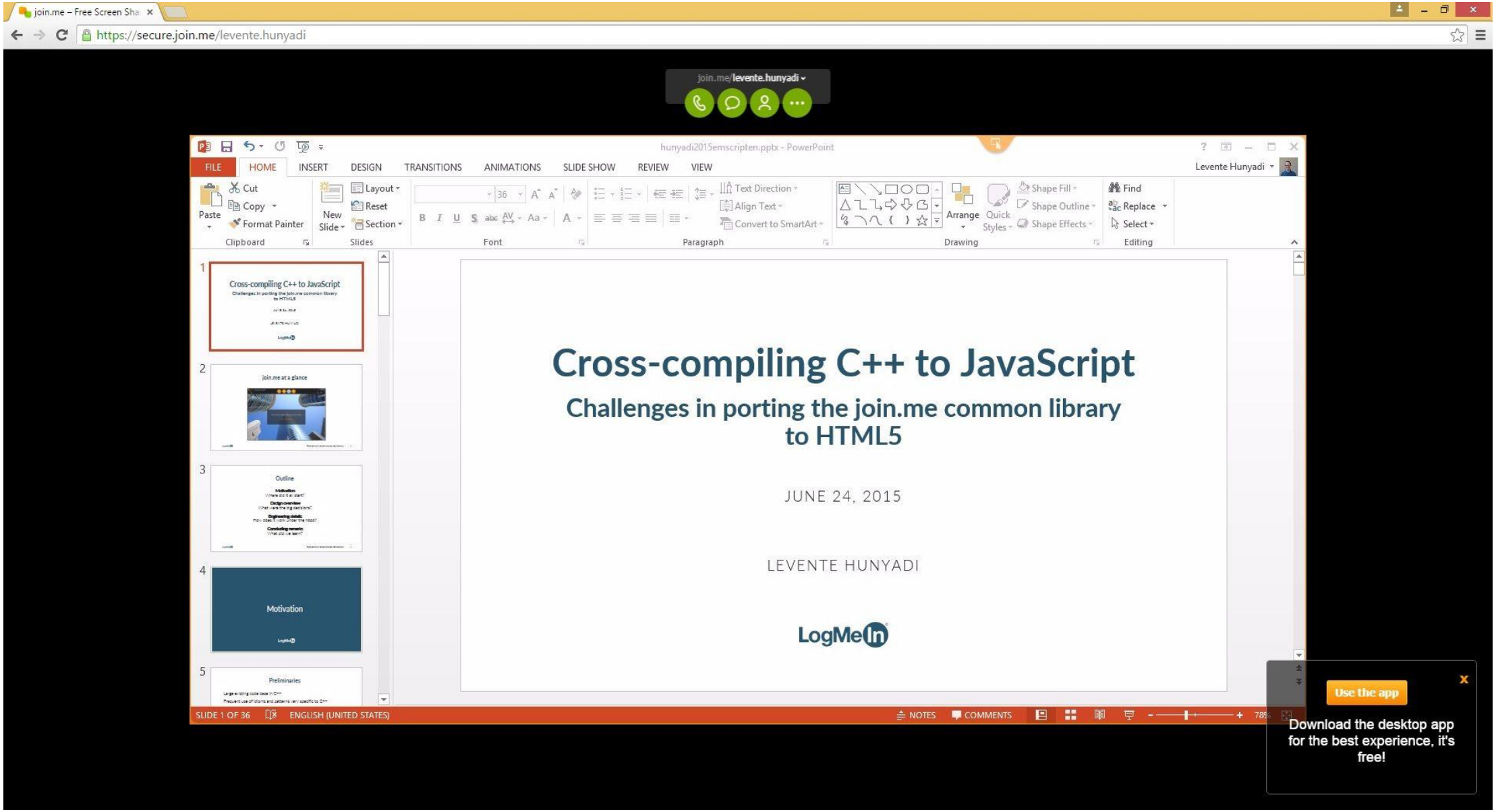
Motivation for HTML5

Join a meeting with a “single click” on the website

- Zero deployment

Universal browser support

- No special plug-ins needed for things to work out of the box



Preliminaries regarding common library

Large existing code base in C++

Frequent use of idioms and patterns very specific to C++

Code meant to be deployed to several platforms

Our principal problem:

Browsers run JavaScript, not C++

Some consequences

Large existing code base in C++

- Use a fully (or at least mostly) automated procedure

Frequent use of idioms and patterns very specific to C++

- Migrate code in a way that “eats everything”
  - truly multi-paradigm: our library code covers several fields including scheduling, low-level data manipulation, managing application state, notifications, etc.
  - modern C++ features like lambdas, *auto*, *decltype*, move semantics, etc.

Code meant to be deployed to several platforms

- Avoid introducing platform-specific behavior
  - if possible, no *#if defined(…)*

LogMeIn

Ingredients for our recipe

Compiles arbitrary C++ source code

- generates and operates on low-level primitives

Generates cross-browser compiled code

- no browser-specific dependencies

Does not sacrifice efficiency

- compiled code is fast, only cross-interface calls are relatively expensive

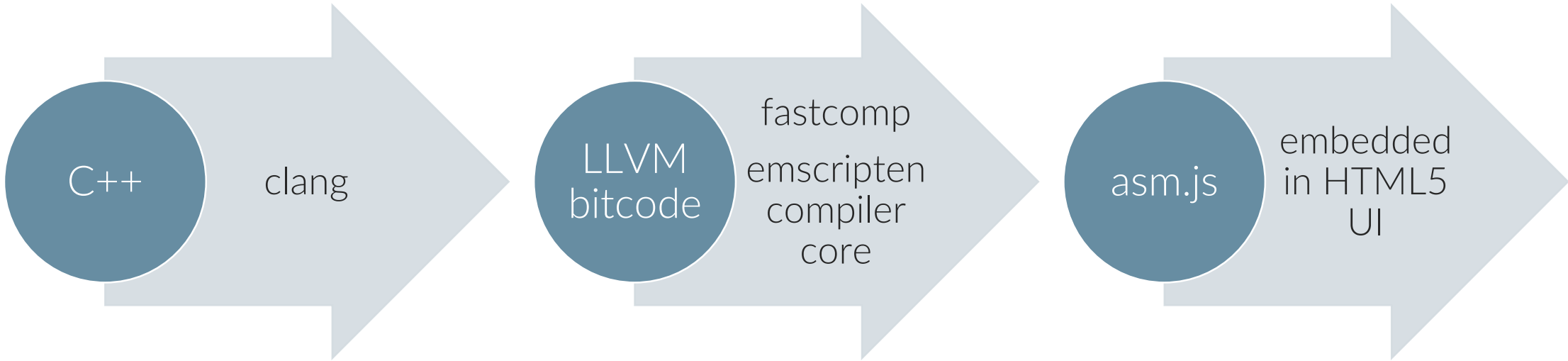
Our solution:

emscripten with embind

emscripten

Emscripten is an LLVM-based project that compiles C and C++ into highly-optimizable JavaScript in asm.js format. This lets you run C and C++ on the web at near-native speed, without plugins.

emscripten toolchain



Why emscripten?

Compiles arbitrary C++ code

- compiles advanced C++11 features our common library uses (emscripten is based on C++11 conforming clang compiler)
- compatible with any programming paradigm
- bidirectional interface with JavaScript (call JS in C++, call C++ in JS)

Why emscripten?

Generates portable code, no browser-specific dependencies

- compiled code runs in a (sandboxed) pseudo-virtual machine
  - fixed-size (16 MB) memory for stack, heap and static data allocated in JavaScript memory area
  - no special add-on binaries or browser plug-ins required (“no-download” common library)
- external context (e.g. window) wrapped in C++ functions
  - C++ standard library *libc* implemented in JavaScript (e.g. *printf* writes to browser console)
  - JavaScript environment is almost totally opaque to C++

Why emscripten?

Compiled code is fast, only cross-interface calls are expensive

- C++ optimization techniques apply
  - LLVM optimization for C++ source code
  - advanced C++ optimization carry over to JavaScript domain (for free)
- harness high-efficiency techniques in JavaScript
  - compiled code runs in asm.js mode
  - memory model implemented with fast JavaScript typed arrays (Uint8Array)
- relatively little data passes C++/JavaScript boundary

Interesting design challenges

or

the join.me common library under the hood illustrated with two case studies

LogMeIn

More (or less) common programming techniques applied in the join.me common library

Task-based parallelism

- cross-thread communication with execution queues
- asynchronous operation scheduling

Extensive use of callbacks in continuation passing style

- what to do next is wrapped in a callback in C++, passed to and called from external code but executed in C++
- error-handling with status codes and state change, not exceptions

Abstract classes with virtual functions act as interfaces

Typical code sample from join.me

```
auto self = shared_from_this();
UpdateEarlyAccess(reason, [self, sessionDesc, reason](
        bool didTriggerUpdate, ICommonToUI::ErrorCode errorCodeForLaunchpad) {

    if (didTriggerUpdate) {
        return;
    }

    LMI::ENQUEUE(self->GetQueue(), [self, sessionDesc, errorCodeForLaunchpad]() {
        SendOp(self->GetUI(), ICommonToUI::OpReset, ..., sessionDesc, ...);
        // ...
        self->m_State = STATE_ON_LAUNCHPAD;
    });
});
```

Continuation passing style

```
auto self = shared_from_this();
UpdateEarlyAccess(reason, [self, sessionDesc, reason](
        bool didTriggerUpdate, ICommonToUI::ErrorCode errorCodeForLaunchpad) {

    if (didTriggerUpdate) {
        return;
    }

    LMI::ENQUEUE(self->GetQueue(), [self, sessionDesc, errorCodeForLaunchpad]() {
        SendOp(self->GetUI(), ICommonToUI::OpReset, ..., sessionDesc, ...);
        // ...
        self->m_State = STATE_ON_LAUNCHPAD;
    });
});
```

Cross-thread communication with queues

```
auto self = shared_from_this();
UpdateEarlyAccess(reason, [self, sessionDesc, reason](
        bool didTriggerUpdate, ICommonToUI::ErrorCode errorCodeForLaunchpad) {

    if (didTriggerUpdate) {
        return;
    }

    LMI::ENQUEUE(self->GetQueue(), [self, sessionDesc, errorCodeForLaunchpad]() {
        SendOp(self->GetUI(), ICommonToUI::OpReset, ..., sessionDesc, ...);
        // ...
        self->m_State = STATE_ON_LAUNCHPAD;
    });
});
```

Error handling with status codes and state

```
auto self = shared_from_this();
UpdateEarlyAccess(reason, [self, sessionDesc, reason](
        bool didTriggerUpdate, ICommonToUI::ErrorCode errorCodeForLaunchpad) {

    if (didTriggerUpdate) {
        return;
    }

    LMI::ENQUEUE(self->GetQueue(), [self, sessionDesc, errorCodeForLaunchpad]() {
        SendOp(self->GetUI(), ICommonToUI::OpReset, ..., sessionDesc, ...);
        // ...
        self->m_State = STATE_ON_LAUNCHPAD;
    });
});
```

Abstract classes with virtual functions

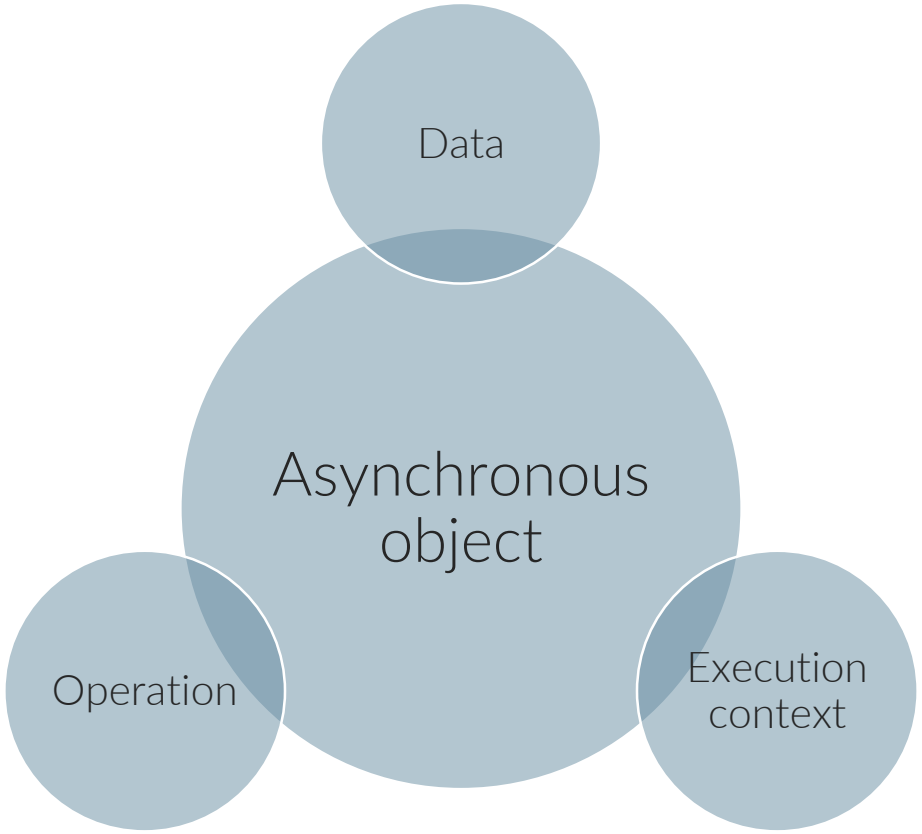
```
auto self = shared_from_this();
UpdateEarlyAccess(reason, [self, sessionDesc, reason](
        bool didTriggerUpdate, ICommonToUI::ErrorCode errorCodeForLaunchpad) {

    if (didTriggerUpdate) {
        return;
    }

    LMI::ENQUEUE(self->GetQueue(), [self, sessionDesc, errorCodeForLaunchpad]() {
        SendOp(self->GetUI(), ICommonToUI::OpReset, ..., sessionDesc, ...);
        // ...
        self->m_State = STATE_ON_LAUNCHPAD;
    });
});
```

Case study 1

Task-based parallelism

Traditional OO

Task-based parallelism



Typical code sample from join.me

```
LMI::ENQUEUE(self->GetQueue(), [self, sessionDesc, errorCodeForLaunchpad]() {
    SendOp(self->GetUI(), ICommonToUI::OpReset, ..., sessionDesc, ...);
    // ...
    self->m_State = STATE_ON_LAUNCHPAD;
});
```

- statement does not block, returns immediately
- operation is posted on execution queue
- callback is invoked when operation is scheduled

Our C++ library vs. JavaScript

- asynchronous
- uses callbacks
- supports timed operations

- asynchronous
- uses callbacks
- supports timed operations

Our C++ library vs. JavaScript

- native hosts are multi-threaded (parallel execution)
- native implementations use lock-free queues
- synchronous operations are supported

- JavaScript is single-threaded (global execution order)
- browsers have their own scheduler (*setTimeout* and *setInterval*)
- synchronous operations are **not** supported

LogMeIn

Our C++ library vs. JavaScript

Implementation

- assign the same global thread to all execution contexts in JavaScript
- use a special-purpose queue for JavaScript exposed over our asynchronous object interface
  - *SendOp(…)* uses *window.setImmediate(fn)*
  - *SendDelayedOp(…)* uses *window.setTimeout(fn, delay)*
- re-write all synchronous operations in our common library as asynchronous operations to be compatible with JavaScript paradigm

End result: full transparency

- no need to be aware whether we target a native platform or HTML5
- uses most efficient task scheduling available on the target platform

Case study 2

Communicating over external interfaces

Traditional OO interface vs. JavaScript

- exposes interface with virtual functions
- mixes value types and (ownership-retaining) reference (pointer) types
  - int, enum, std::string
  - T&
  - std::shared_ptr<T>

- no true interfaces or virtual functions (uses prototyping instead)
- only value types and ownership-ignorant reference types
  - string
  - object

Interacting with OO interface from JS

1) Call a common library operation from JavaScript front-end

   a JavaScript function must be able to call a member function implemented in C++

2) Implement a library interface in JavaScript front-end

   a C++ member function must be able to call a JavaScript function bound to an object

   cross-language inheritance at work: JavaScript “derives” from C++ class (e.g. TCP socket interface in common library implemented with web sockets in browser)

Interacting with OO interface from JS

1) Call a common library operation from JavaScript front-end

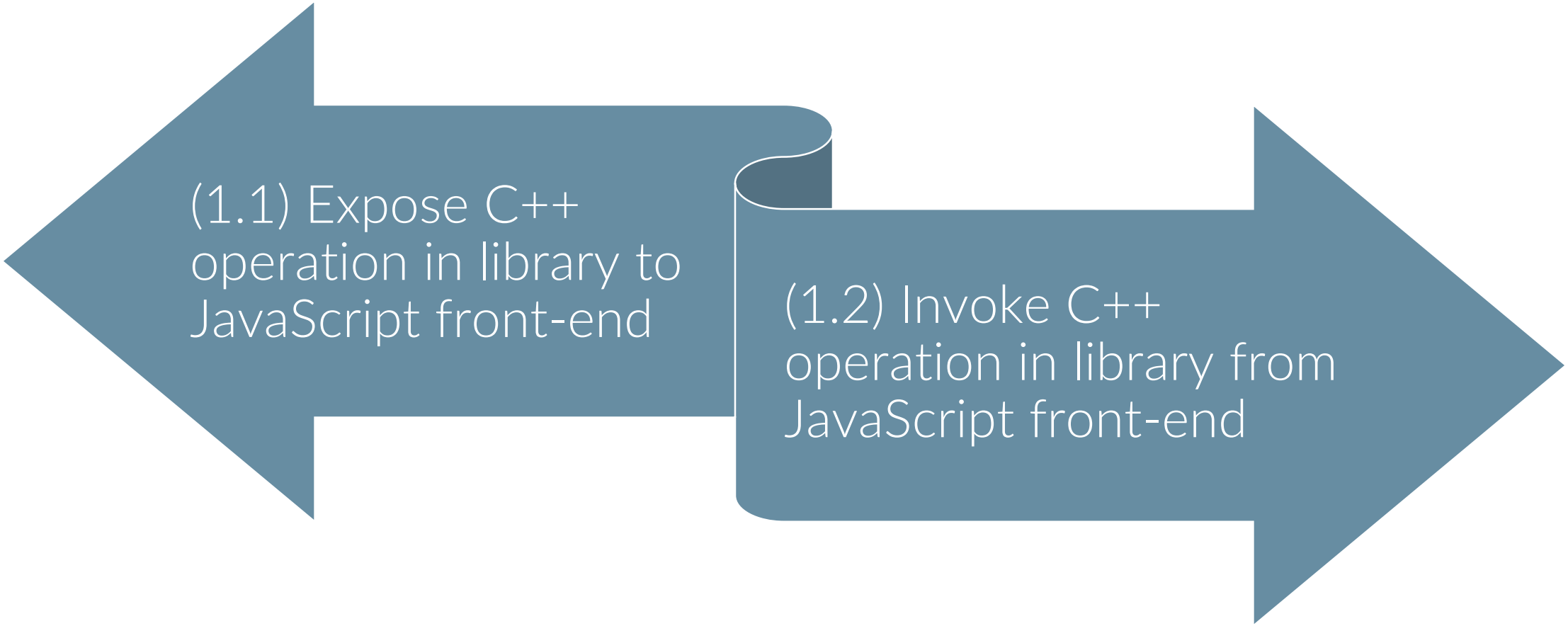
   a JavaScript function must be able to call a member function implemented in C++

2) Implement a library interface in JavaScript front-end

   a C++ member function must be able to call a JavaScript function bound to an object

   cross-language inheritance at work: JavaScript “derives” from C++ class (e.g. TCP socket interface in common library implemented with web sockets in browser)

(1) Call a library operation in C++ from JavaScript front-end



(1.1) Expose library operation in C++ to JS

- expose class type by registering a unique name with *em::class_(…)*
- add binding using emscripten method *.function()* to register signature
- use embind utility functions in common to help with type mapping, e.g. UTF-8 strings

(1.1) Expose library operation in C++ to JS

```
em::class_<IUIToCommon>("IUIToCommon")
    .smart_ptr<std::shared_ptr<IUIToCommon>>("IUIToCommonSPtr")
    .function("OpPresenterSwitch_Initiate",
        EMBIND_METHOD(IUIToCommon::OpPresenterSwitch_Initiate))
    .function("OpPresenterSwitch_Cancel",
        EMBIND_METHOD(IUIToCommon::OpPresenterSwitch_Cancel))
;
```

(1.1) Expose library operation in C++ to JS "Dark magic" inside

```
template<typename MemberFunctionType, MemberFunctionType memberFunction, typename ReturnType,
    typename ClassType, typename... Args>
struct MethodInvoker {
    static ReturnType invoke(ClassType& obj, typename TypeTransformer<Args>::type... args) {
        return (obj.*memberFunction)(reverse_transform_argument<Args>(args)...);
    }
};

template<typename MemberFunctionType, MemberFunctionType memberFunction, typename ReturnType,
    typename ClassType, typename... Args>
constexpr auto wrap_method(ReturnType(ClassType::*)(Args...)) -> ReturnType(*)(ClassType&, typename
TypeTransformer<Args>::type...)
{
    return &MethodInvoker<MemberFunctionType, memberFunction, ReturnType, ClassType, Args...>::invoke;
}

#define EMBIND_METHOD(method) wrap_method<decltype(&method),&method>(&method)
```

(1.1) Expose parameter types and callbacks in C++ to JS

- add binding for function parameter types with *em::class_(...)*
- add binding for properties of newly registered types with *.property(...)*, *.field(...)*
- add binding for new callback function signatures (unique globally, no aliases)
- add binding for enumerations (both new- and old-style enums)

(1.1) Expose parameter types and callbacks in C++ to JS

```
em::value_object<Point>("Point")
    .field("x", &Point::x)
    .field("y", &Point::y)
;

typedef std::function<void(ErrorCode)> Callback;
em::class_<ITCPSocket::Callback>("TCPCallback")
    .constructor<const ITCPSocket::Callback&>()
    .function("Invoke", &ITCPSocket::Callback::operator())
;
```

(1.2) Invoke library operation in C++ from JS

- use global object *Module* to access C++ class as a prototype
- use keyword *new* to instantiate an object
- use the instance almost as a regular JavaScript object

(1.2) Invoke library operation in C++ from JS

```
AsyncConnect: function (address, port, bufferSize, callback) {
    var address = 'wss://' + address + ':' + port + '/endpoint';
    var connectCallback = new Module.TCPCallback(callback);
    var socket = new WebSocket(address);
    socket.binaryType = 'arraybuffer';
    socket.onopen = function () {
        connectCallback.Invoke(0);
        connectCallback.delete();
        connectCallback = null;
    };
}
```

(1.2) Invoke library operation in C++ from JS

```
AsyncConnect: function (address, port, bufferSize, callback) {
    var address = 'wss://' + address + ':' + port + '/endpoint';
    var connectCallback = new Module.TCPCallback(callback);
    var socket = new WebSocket(address);
    socket.binaryType = 'arraybuffer';
    socket.onopen = function () {
        connectCallback.Invoke(0);
        connectCallback.delete();
        connectCallback = null;
    };
}
```

Interacting with OO interface from JS

1) Call a common library operation from JavaScript front-end

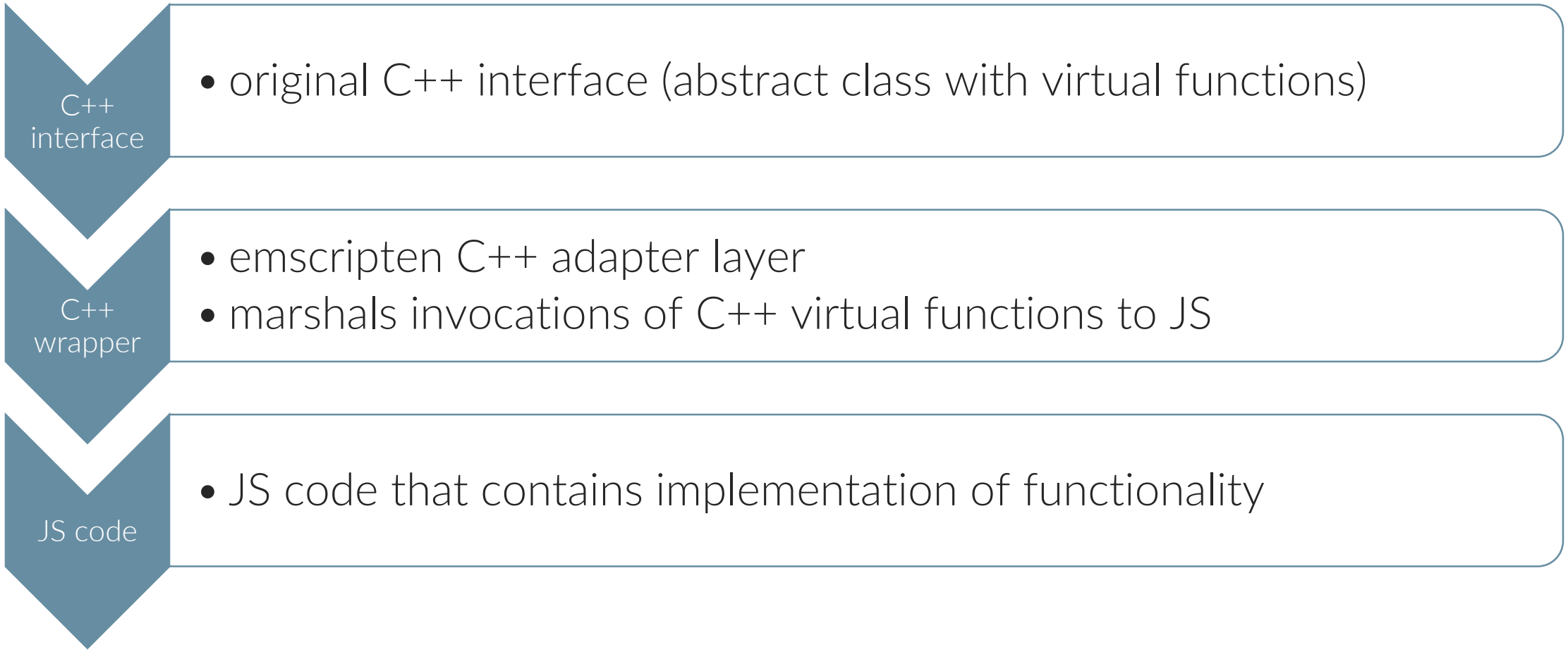
   a JavaScript function must be able to call a member function implemented in C++

2) Implement a library interface in JavaScript front-end

   a C++ member function must be able to call a JavaScript function bound to an object

   cross-language inheritance at work: JavaScript “derives” from C++ class (e.g. TCP socket interface in common library implemented with web sockets in browser)

(2) Implement a library interface in JavaScript



LogMeIn

(2) Implement a library interface in JavaScript

(2.1) C++

- add a new wrapper class
- add special macro to generate emscripten constructor
- add member functions to map virtual function calls to JavaScript calls (no virtual methods in JavaScript)

(2.2) JavaScript

- use the function *.implement(…)* to bind to C++ interface

(2.1) Annotating an interface for external calls in C++

```
struct CommonToUI : public em::wrapper<ICommonToUI> {
    EMSCRIPTEN_WRAPPER(CommonToUI);

    void OpChatMessageDelivered(std::string msg) override {
        return call<void>("OpChatMessageDelivered", msg);
    }

    // ...
};
```

(2.1) Register an interface for external calls in C++

```
em::class_<ICommonToUI>("ICommonToUI")
    .allow_subclass<CommonToUI>("CommonToUI")
    .function("OpChatMessageDelivered",
        EMBIND_METHOD(ICommonToUI::OpChatMessageDelivered),
        em::pure_virtual()
    )

    // ...
;
```

(2.2) Pass an object implementing an interface in JS for external calls in C++

```
var iCommonToUI = {
    OpChatMessageDelivered: function (targetId, msgTime, msg) {
        console.log('OpChatMessageDelivered');
    }
}

var jsCommonToUI = Module.WrapCommonToUI(
    Module.ICommonToUI.implement(iCommonToUI)
);
var service = new Module.Facade(
    jsCommonToUI, jsPlatform, jsBasicDependencyProvider
);
```

Pitfalls in cross-compiled code

- no *long* type (only *Number*, which is equivalent to C++ *double*)
- passing long arrays is inefficient (involves data copies)
  - memory views to elide copies for reads/writes (*emscripten::memory_view<T>*)
- no garbage collection in JavaScript for objects originating from C++
  - must clean everything up in JavaScript (use *delete()* method)
- native JavaScript in C++ (*emscripten::val*)
  - conversion functions in common library to an interval JSON class type
- synchronous (blocking) operations are not supported

Parting thoughts

LogMeIn

Things we learned

Very effective code re-use is possible

- limited emscripten-specific code in common library
- limited rewrite required in common library to be compatible with HTML5

Tracking down errors originating from C++ is not easy

- no interactive debugger
- many nested lambdas leave little clue as to what could be going on

Inadequate compile-time verification of type binding correctness

- undefined or ill-defined types discovered only at run time when parameters are passed

Thank you!

Levente Hunyadi
levente.hunyadi@logmein.com

LogMeIn