Microsoft

JavaScript Async Context

Mike Kaufman

Principal Software Engineer
Chakra JavaScript Engine
Microsoft

Mark Marron

Principal Research Engineer
Microsoft Research

About This Talk

- What is Async Context?
- Why is Async Context important?
- Where is the JS community today?
- A Formal Model
- How Model Solves Problems
- Next Steps

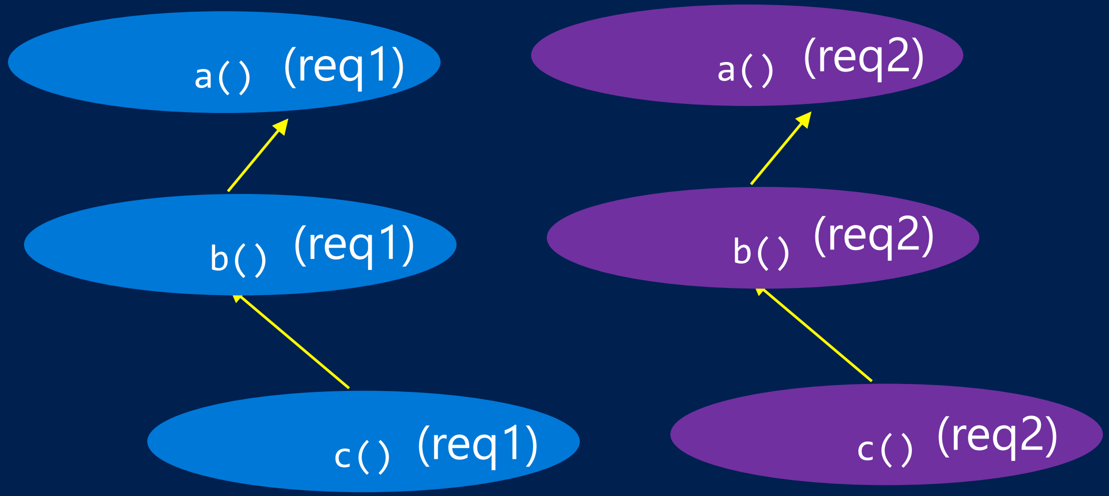
What is Async Context?

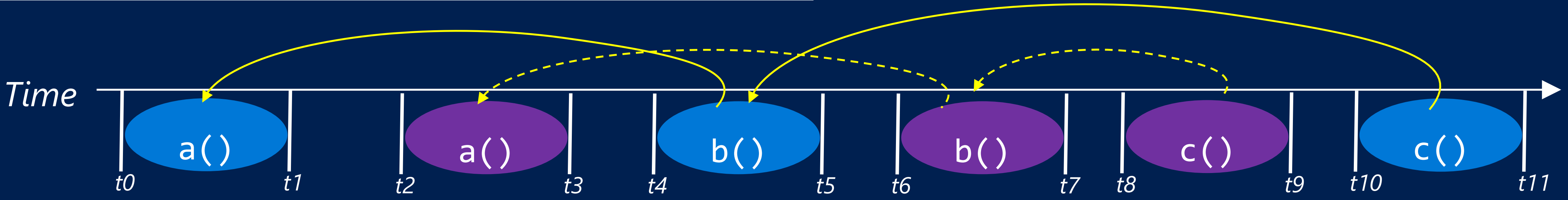
Ability to answer question **“How did I get here?”**, across async boundaries.

What is Async Context?

Ability to answer question **“How did I get here?”**, across async boundaries.

```
app.get('/', function a(req, res) {
  db.query(
    'SELECT * from PRODUCTS’,
    function b(data) {
      genPage(
        data, function c(content) {
          res.send(content);
        })
    })
});
```





Why Is Async Context Important?

- Foundational to JS programming model, yet:
  - No shared terminology & concepts
  - No ties to JS code
  - No Formal Specification
- Useful in applications & tooling:
  - Continuation local storage
  - Asynchronous call stacks
  - Async *step-into* in debugger
  - Memory leak detection
  - APM reporting
  - …

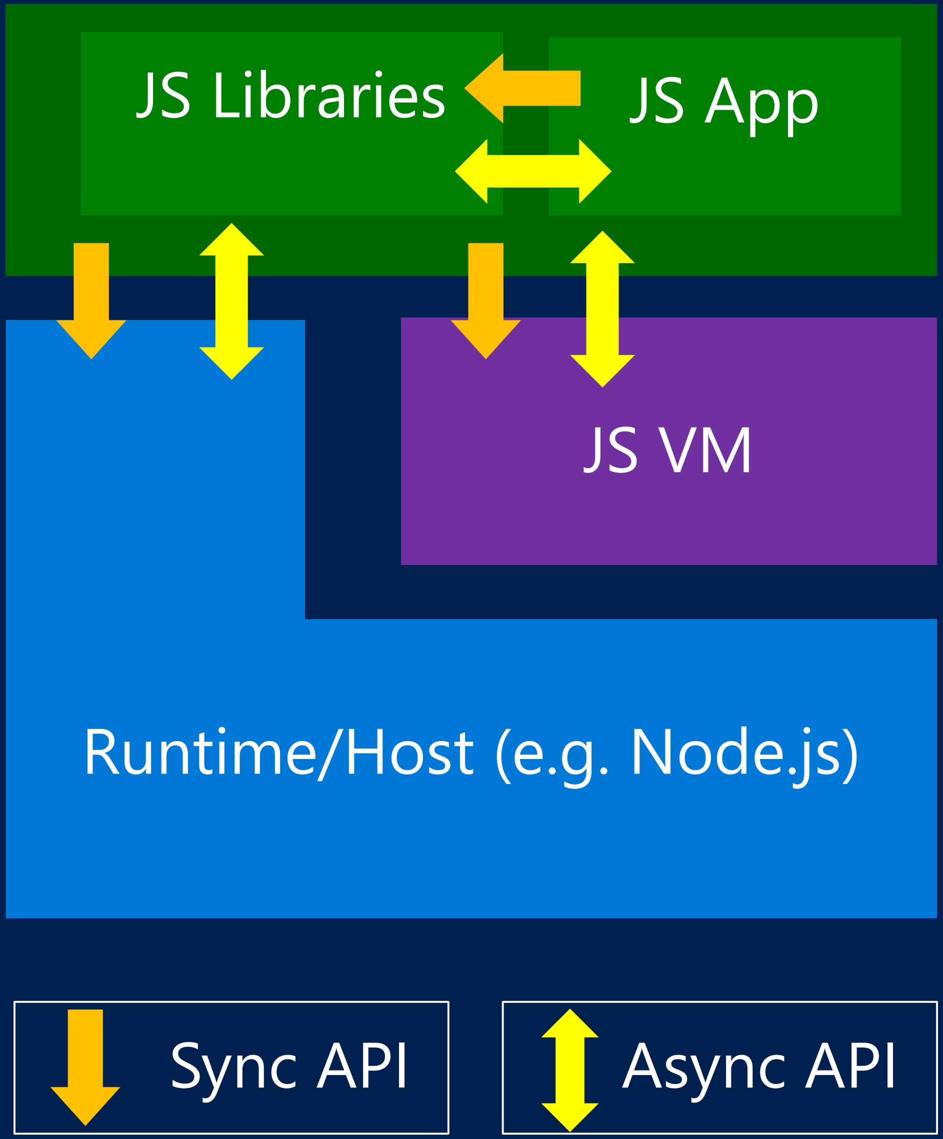
Where Is JS Community Today?

- Monkey Patching
  - Various ad-hoc approaches – module load interception and special cases for async-await
  - Breaks when monkey-patched APIs change
- Domains
  - Conflated async call flow w/ exception handling
  - Explicit enter()/leave() required
- Async-Hooks
  - Life-cycle events on Node.js "resources"
  - Exposes Node implementation details
  - Node.js only - No corollary to browser, other hosts.
  - Native->JS transitions and lifecycle events in model are expensive
- Academic Investigations
  - "Finding Broken Promises in Asynchronous JavaScript Programs" – Alimadadi *et al.*, 2018
  - "Semantics of Asynchronous JavaScript" – Loring *et al.*, 2017

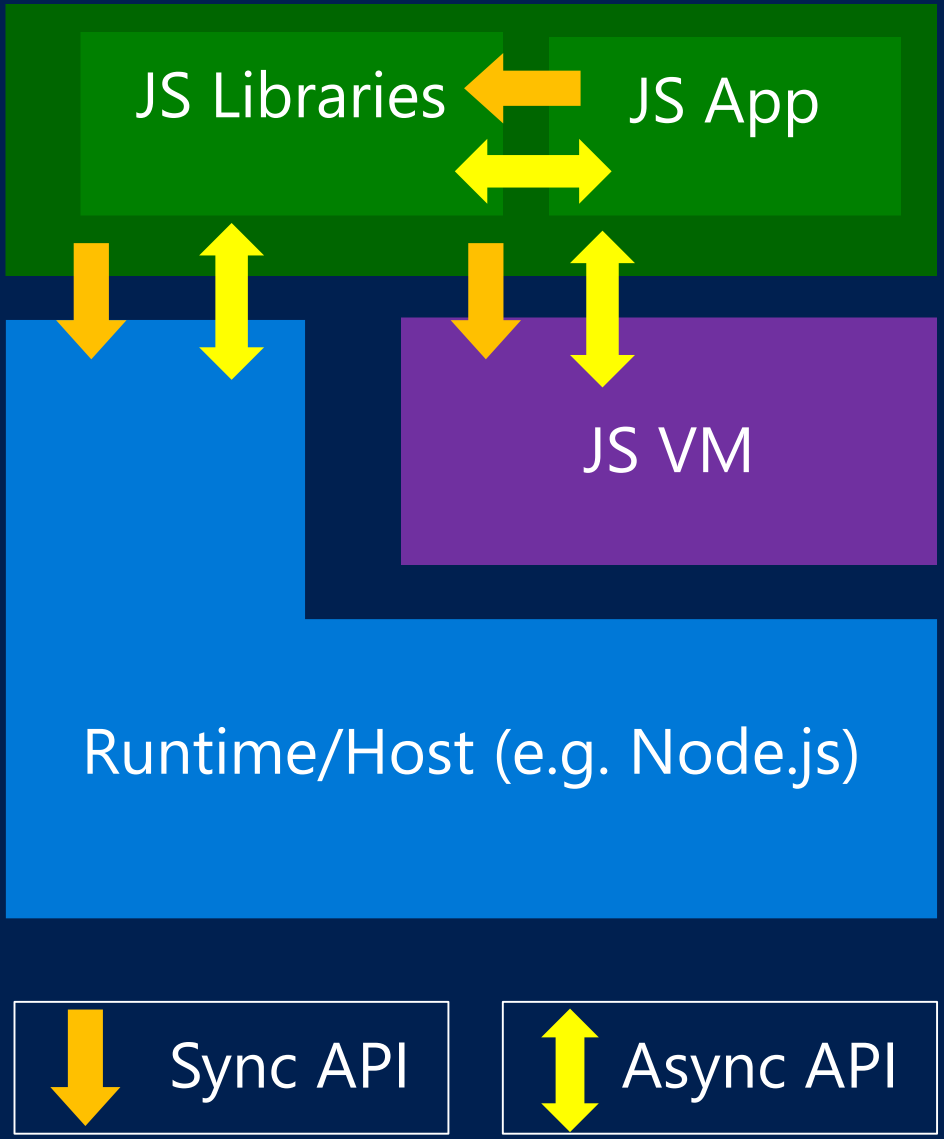
A Formal Model

- Define Concepts of Async Code Execution
  - Provide names for the concepts
- Provide Explicit Structure
  - Clear data structures that can be reasoned about
- No Polices
  - Policy experiments can be implemented over the structure
- Implementation at VM Level
  - Integration across the stack
  - Potential for syntactic constructs
  - Eliminate expensive callbacks
  - First-Class support as JavaScript evolves
    - E.g., address issues with Promises and Async/Await at the spec level

A JavaScript Application



A JavaScript Application



Async API:

- API that takes a function as a parameter, function is invoked asynchronously.

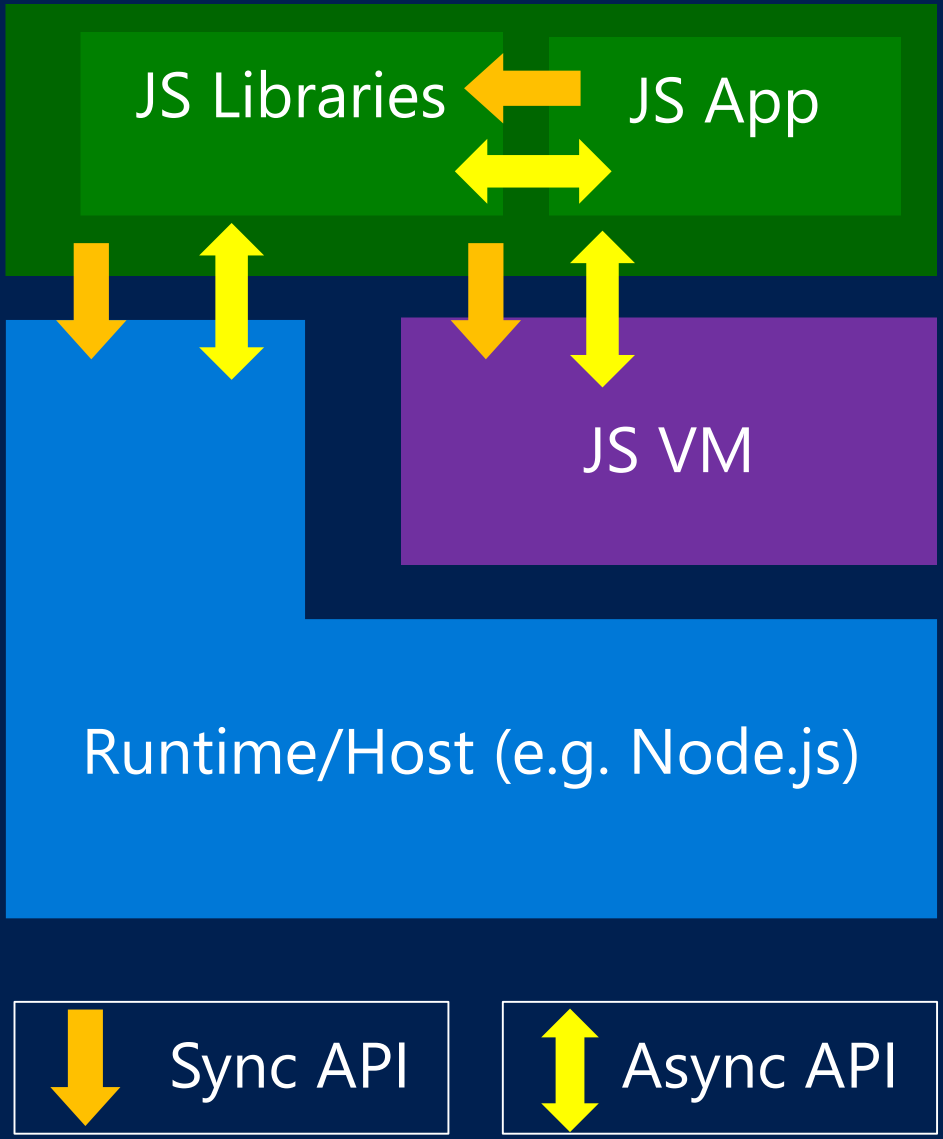
Goal:

- Provide ***primitive constructs*** that capture Async code flow at ***API boundaries***.

Why?

- API boundaries are close to programmer's mental model.
- Primitives are simple
  - "Correctness" gets pushed up stack

Concepts



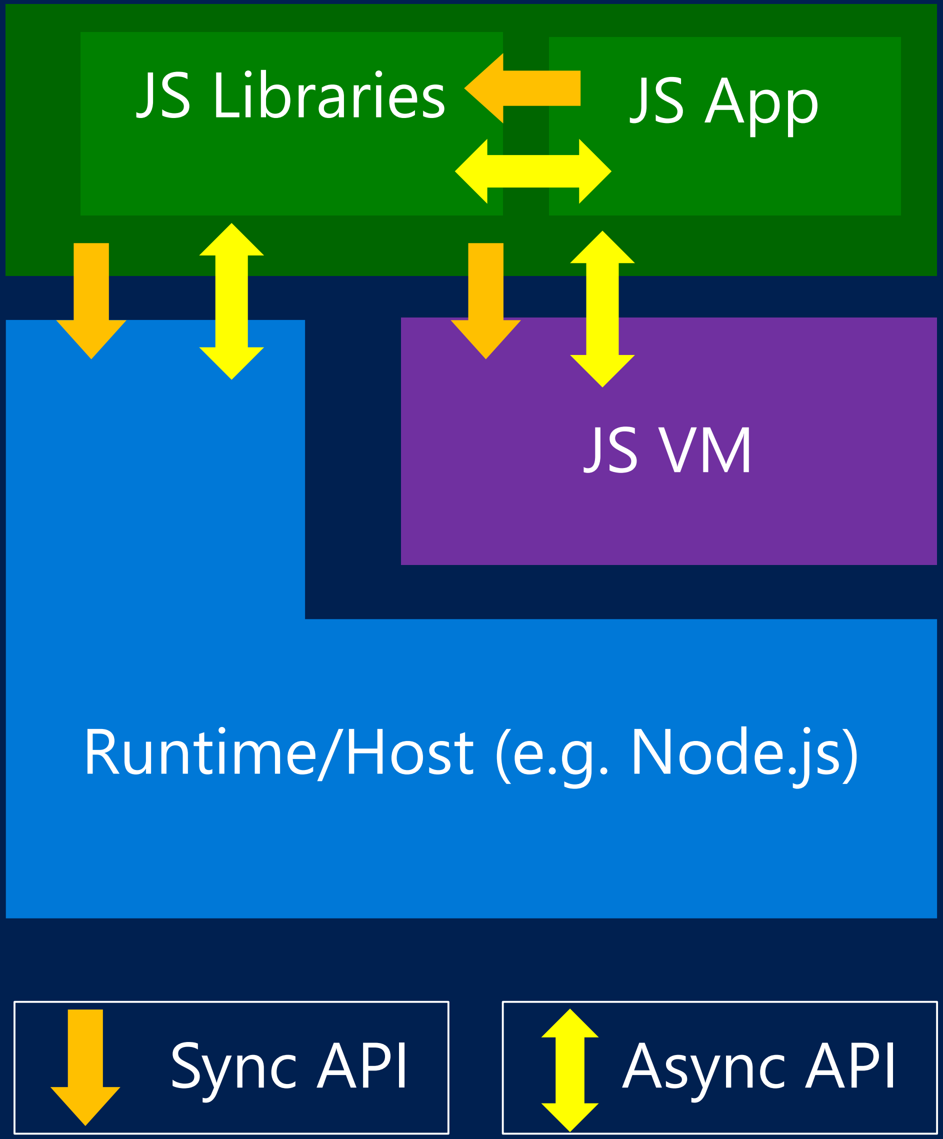
Continuation: special type of function that is passed into an Async API.

Context: structure created when a ***Continuation*** is invoked.

Assumption: All functions passed across Async API Boundaries are ***Continuations***

Invariant: All JS code executes inside a ***Context***.

Concepts



***Continuation*:** special type of function that is passed into an Async API.

Context*:** structure created when a ***Continuation is invoked.

Assumption: All functions passed across Async API Boundaries are ***Continuations***

Invariant: All JS code executes inside a ***Context***.

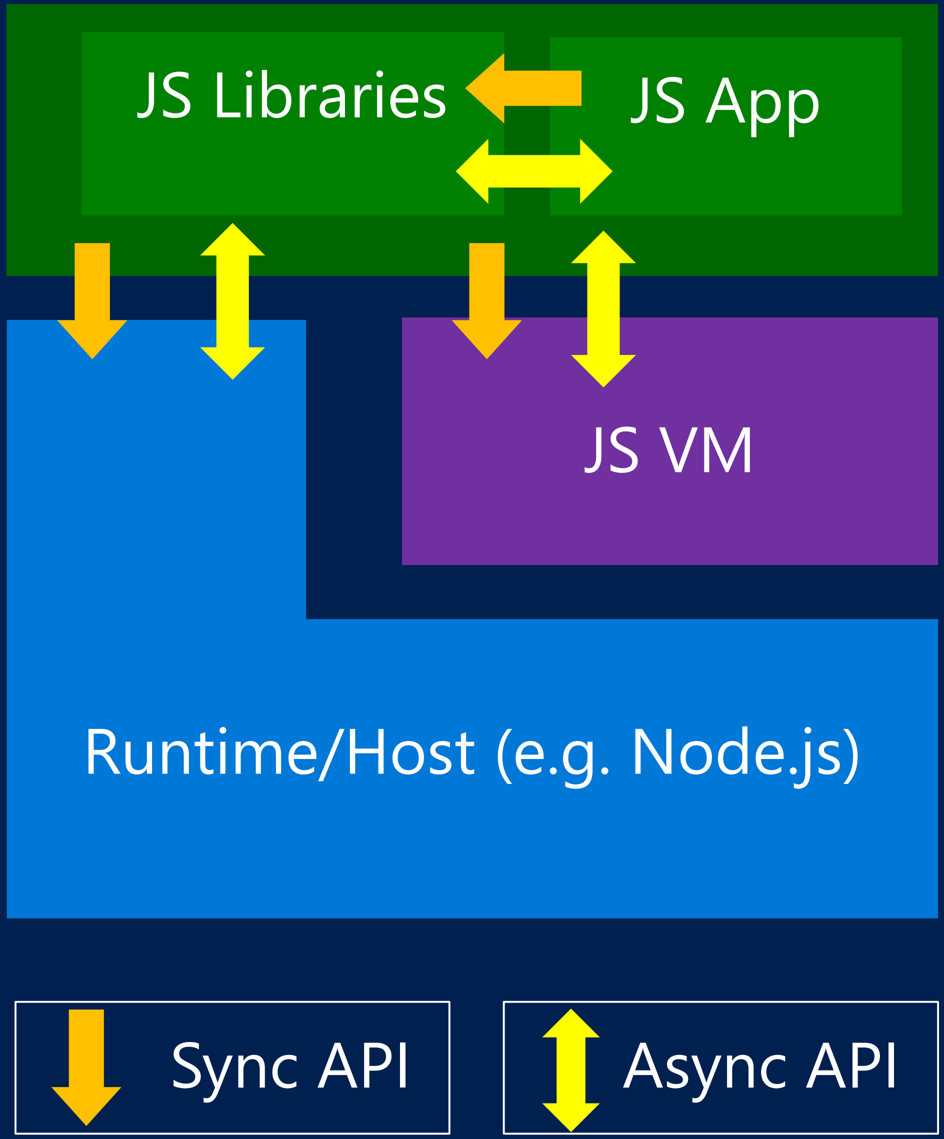
continuify() API

- VM-Provided API
- Given function f, transform f into a continuation
- typescript:
  - `continuify(f): Continuation`

Updated Host APIs:

```
function setTimeout(f, timeout) {
        const c = continuify(f);
        scheduleTimeout(c, timeout);
}

function nextTick(f) {
     const c = continuify(f);
     scheduleNextTick(c);
}
```

Concepts



***Continuation*:** special type of function that is passed into an Async API.

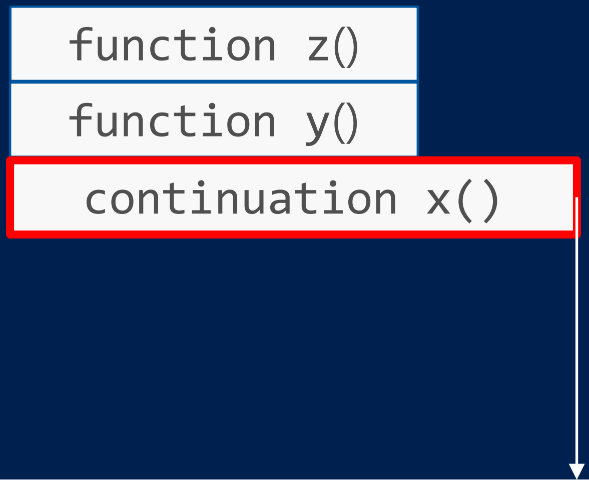
Context*:** structure created when a ***Continuation is invoked.

Assumption: All functions passed across Async API Boundaries are ***Continuations***

Invariant: All JS code executes inside a ***Context***.

All JS code executes inside a *Context*

```
function x() {y();}
function y() {z();}
function z() {}
setTimeout(x, 10)
```


function z()

function y()

continuation x()

```
/*
 * data associated with a Context
 *  that lets us answer “How did we get here”?
**/
interface Context {
  invocationID: number;
  continuation: Continuation;
   readyContext: Context;
}
```

```
interface Continuation {
  linkContext: Context;
}
```

An Example

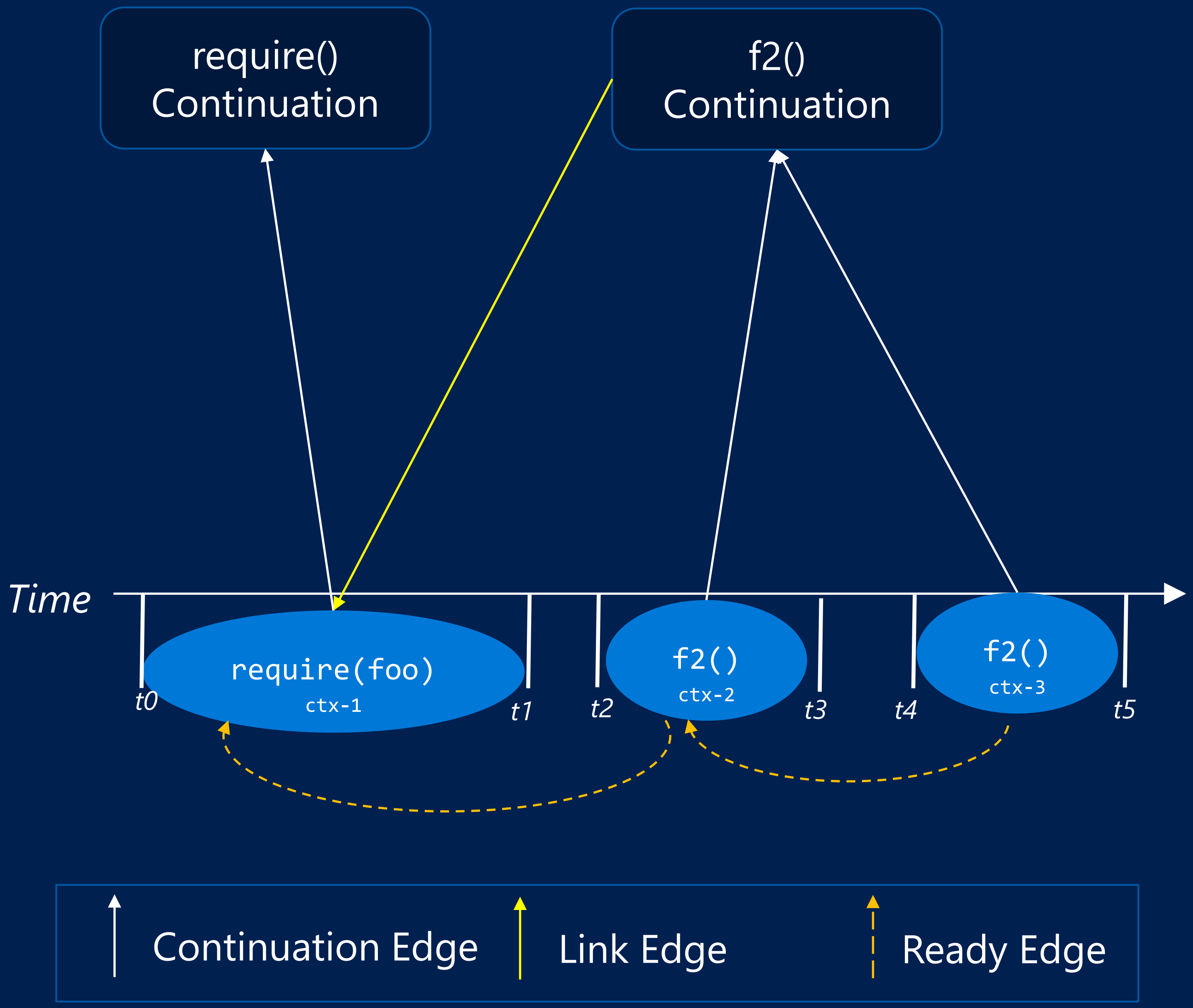
foo.js

```
function x() { ... }

function f2() {
  x();
}

function f1() {
  Promise.resolve()
    .then(f2)
    .then(f2);
}

f1();
```

An Example

foo.js

```
function x() { ... }

function f2() {
  x();
}

function f1() {
  Promise.resolve()
    .then(f2)
    .then(f2);
}

f1();
```



require() Continuation

f2() Continuation

Time

t0 t1 t2 t3 t4 t5

require(foo)
ctx-1

f2()
ctx-2

f2()
ctx-3

Continuation Edge

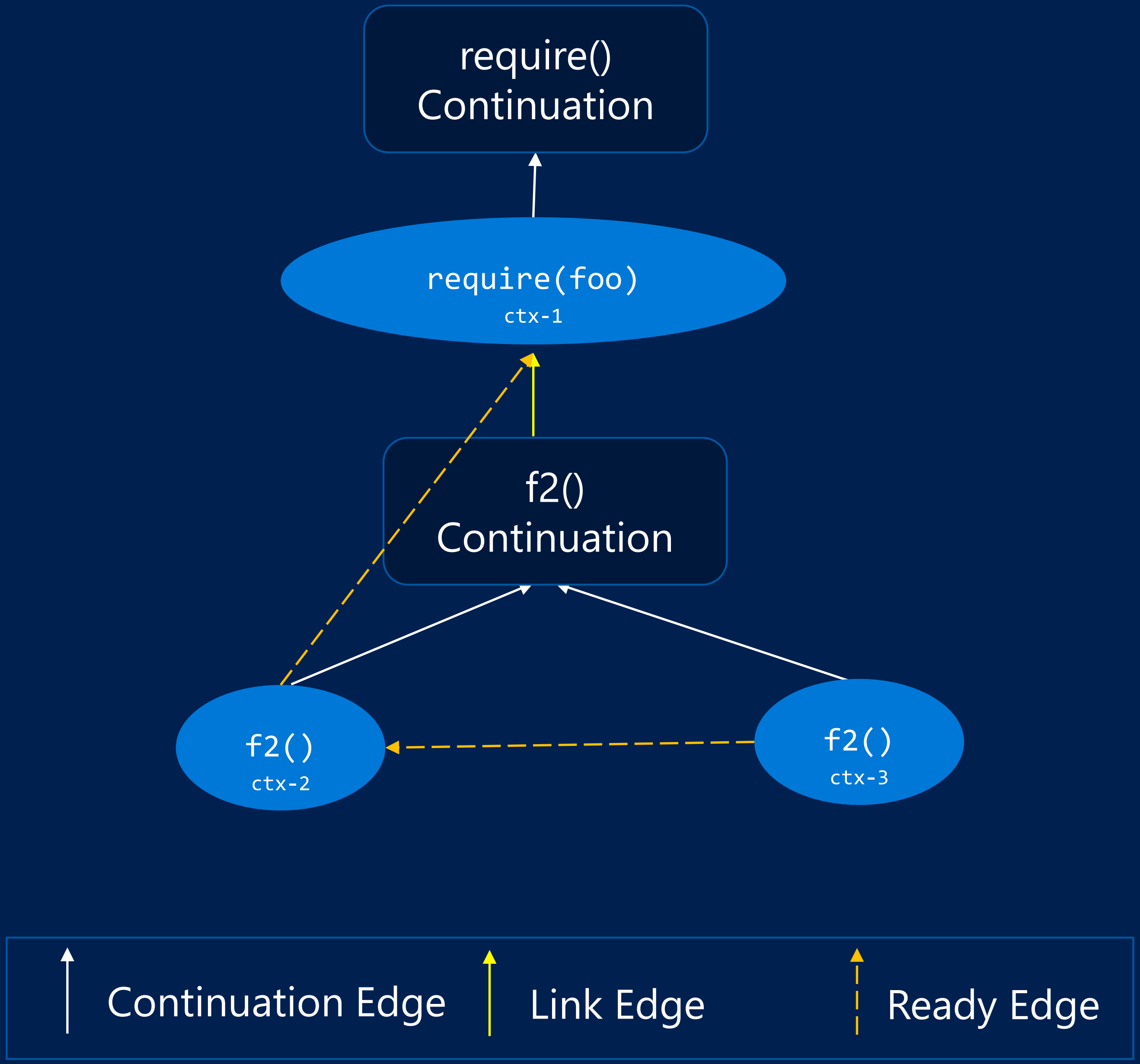
Link Edge

Ready Edge

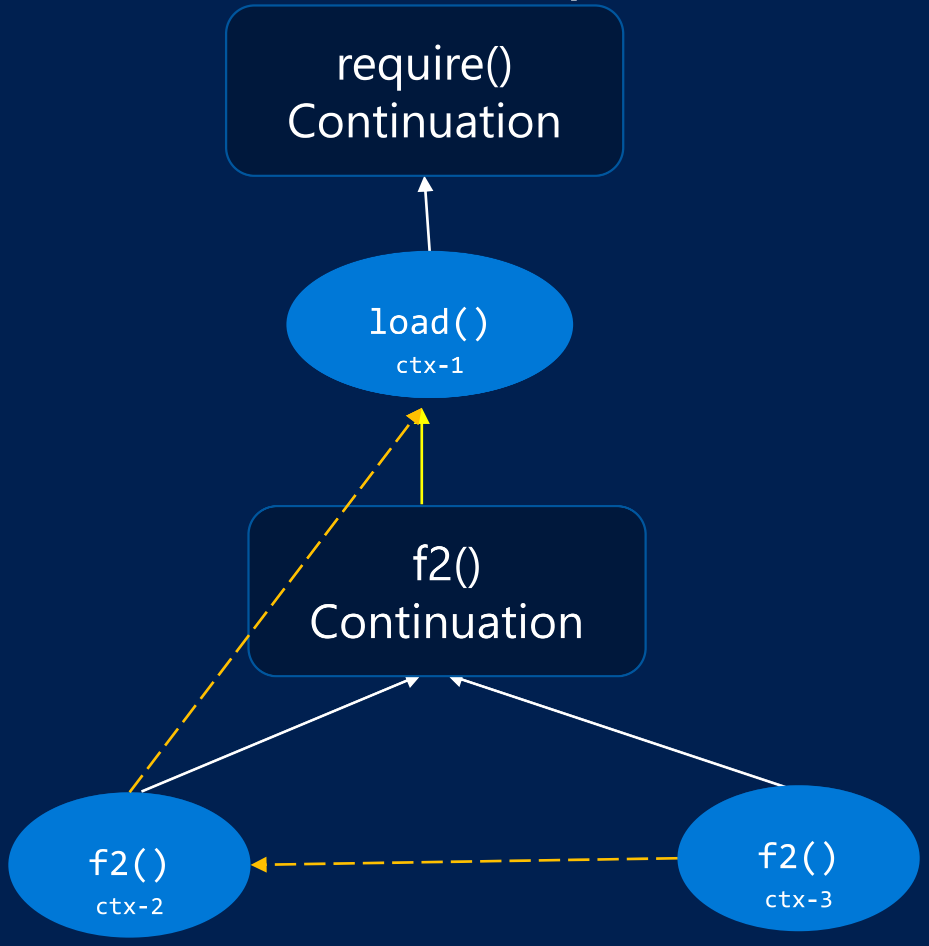
A Graph

foo.js

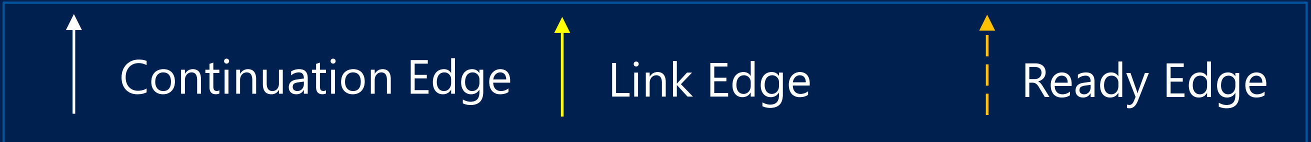
```
function x() { ... }

function f2() {
  x();
}

function f1() {
  Promise.resolve()
    .then(f2)
    .then(f2);
}

f1();
```



Async Call Graph

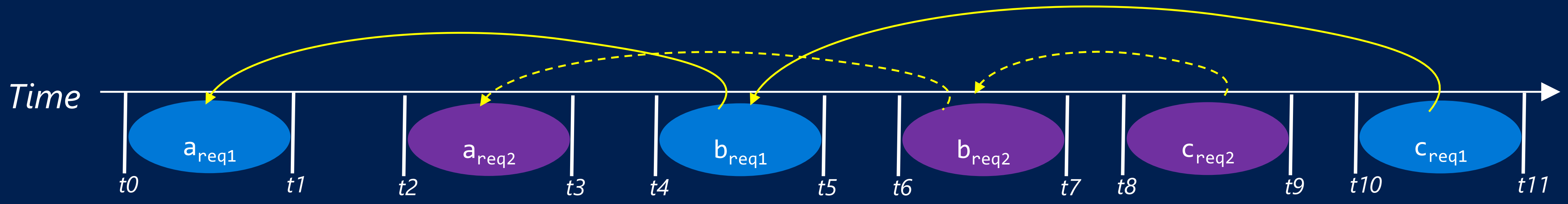


- Directed Acyclic Graph (DAG)
- Nodes: ***Continuations*** and ***Contexts***
- Edges:
  - ***Continuation Edge***
    - Edge from ***Context*** -> ***Continuation*** invoked to create it.
  - ***Link Edge***:
    - Edge from ***Continuation*** -> ***Context*** where ***Continuation*** was created
    - Generally, ***Context*** where Async API called
  - ***Ready Edge***:
    - For promises, edge from ***Context*** to the ***Context*** where previous promise was resolved.



Solution: Understanding Perf Timings

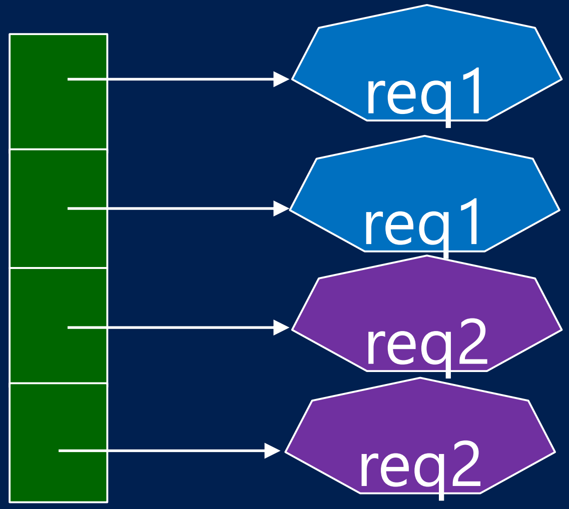
- Annotate Continuations & Contexts with timing data
- Compute
  - Sum of elapsed times in Contexts in a specific async subtree
    - Req1: (*t1-t0*) + (*t5-t4*) + (*t11-t10*)
    - Req2: (*t3-t2*) + (*t7-t6*) + (*t9-t8*)
  - Wall clock time from start of an HTTP request to end:
    - Req1: *t11–t0*
    - Req2: *t9-t2*



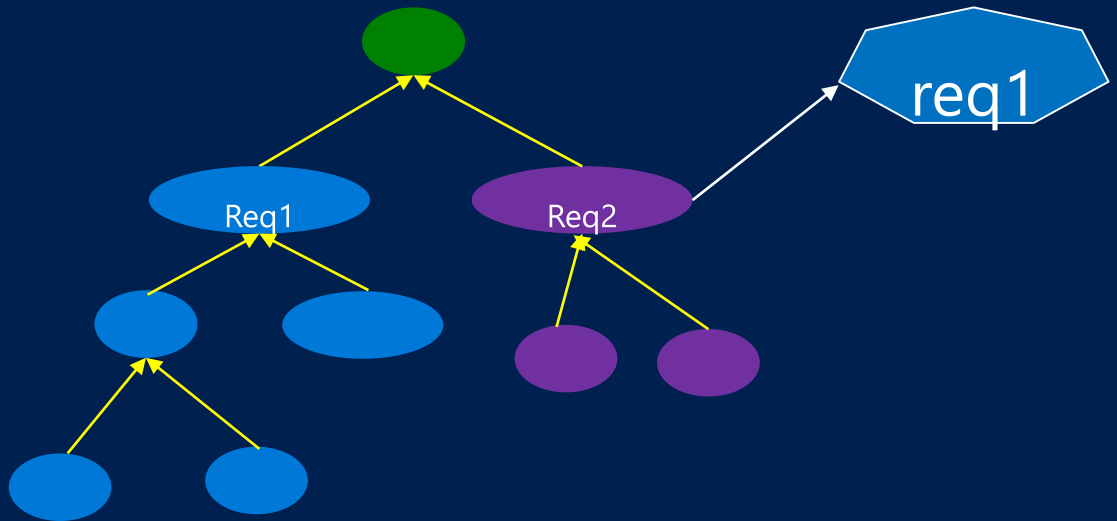
Solution: Memory Leak Detection

- Tag JS heap objects with their "allocation context".
- Apply heuristics to identify potential leaks.

Ex: Single array referencing objects allocated in different context sub-trees



Ex: Objects allocated in one sub-tree, but referenced from another.



Next Steps

- Get Feedback
  - Let us know what you think
- Implementation
  - Investigate efficient VM-level implementation
    - Support model for Promises, Async/Await
    - Provide APIs
  - Host-level
    - Update hosts Async APIs to “continuify” parameters
  - Measure Perf
- ECMA-262 Integration
  - Expand definition of “Execution Context” to include Async Context
  - Update Promise AbstractOperations, async/await to support model
  - Opportunity for syntactic support
    - `continuation x(a, b) { }`

Thank You!

- Feedback
  - Mike Kaufman
    - mike.kaufman@microsoft.com
  - Mark Marron
    - marron@microsoft.com

- Get Involved
  - Node.js Diagnostics Working Group
  - http://github.com/nodejs/diagnostics

- Deep Dive
  - Diagnostics Breakout @ Collab Summit
  - Friday Oct. 12, 3:30 – 5:00

Backup

Continuation Model vs Async Hooks

Async Hooks	Continuation Model
• Variety of “Resources”	• Only “Resource” is Continuation
• Observer model used to infer async structure. • Callbacks from Native to JS Code	• Fixed Definitions of async structure • No callbacks
• Node.js only	• Host-indepdent. • 1st-class concept through VM & host
• Exposes low-level Resources	• No low-level exposure

User Space Queueing

- TBD
  - Complete this slide
- Async APIs defined in JS user space
- Implementation manages its own callback/dispatch logic
  - E.g., database drivers
- In these examples, we have multiple Continuation() frames on the stack
  - Model supports this very nicely
- TBD
  - Example
  - Picture

Solution: CLS

- A simple key/value store
  - get/set API
- Writes occur on current context
- Reads occur by walking “path to the root” of graph
  - Trivially follow “link context” edges.
  - More Complex use cases possible

```
set(key: string, value: any) {
    let curr: Context = Context.GetCurrent();
    let props = curr['CLS'];
    if (!props) {
        props = {};
        curr['CLS'] = props;
    }
    props[key] = value;
}

get(key: string): any {
    let curr: Context = Context.GetCurrent();
    while (curr) {
        let props = curr['CLS'];
        if (props && key in props) {
            return props[key];
         }
         curr = curr.continuation.linkingContext;
    }
    return undefined;
}
```