Realms with Callable Boundary

API Overview

Caridy Patiño, Leo Balter, Rick Waldron

Primary Goals of the Realms proposal

- a new global object and a new set of intrinsics
- a separate module graph
- synchronous communication between both realms
- proper mechanism to control the execution of a program

API Interface

```
declare class Realm {
   constructor();
   importValue(specifier: string, bindingName: string): Promise<PrimitiveValueOrCallable>;
   evaluate(sourceText: string): PrimitiveValueOrCallable;
}
```

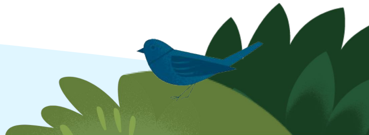
What's new?

No cross-realm object access.

The new Realms API enables a callable boundary cross-realms.

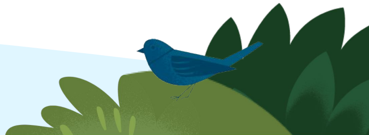
This callable boundary disallows access to any non-primitive values.

Callable objects can still be connected through auto wrapping.



No cross-realm object access

```
const realm = new Realm();

realm.evaluate('globalThis'); // Throws a TypeError

// or
realm.evaluate('[]'); // Throws a TypeError

// or
realm.evaluate('Object.prototype'); // Throws a TypeError
```



Primitives

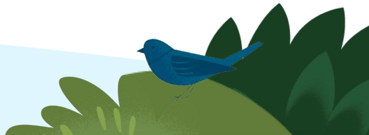
Not limited to strings & numbers

```
const realm = new Realm();

Symbol.for('x') === realm.evaluate('Symbol.for("x")'); // true
```

Realm Wrapped Function Exotic Object

- Has internals `[[Realm]]`, `[[WrappedTargetFunction]]`, and `[[Call]]`
- A new **Wrapped Function Exotic Object** is also created when the **Wrapped Function Exotic Object** returns a callable object.
- This enables *cross-realms callable boundaries.*

```
const r = new Realm();
const wrapped = r.evaluate('x => y => x * y');
const otherWrapped = wrapped(2);
otherWrapped(3); // 6
```



Callable Boundary

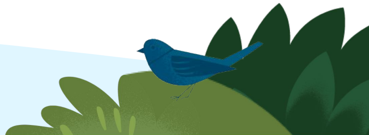
The new Realms API enables a callable boundary cross-realms.

The `[[Call]]` `internal` of a new **Wrapped Function Exotic Object** will call the function set at the same object's `[[WrappedTargetFunction]]` executed in the target's Realm.

Callable Boundary Desugaring

* using pseudo-code for the internals

```
const red = new Realm();
const doSomething = red.evaluate('x => x * 2');
doSomething(3);

doSomething.[[Call]] = function( thisArgument, argumentsList ) {
 let result, target = F.[[WrappedTargetFunction]] // x => x * 2

 try {
   result = target.call( GetWrappedValue(thisArgument), GetWrappedValue(argumentsList[0]) )
   return GetWrappedValue(result)
 } catch {
   throw new TypeError()
 }
}
```



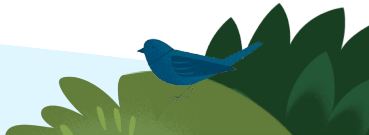
Auto wrapped functions

When one Realm sends a callable object, a new **Wrapped Function Exotic Object** is created in the other realm connected to it.

```
const realm = new Realm();
const wrapped = realm.evaluate('x => x * 2');
```

When the **Wrapped Function Exotic Object** is called, it chains the call to its connected function with the same arguments and returns its return.

```
wrapped(21); // returns 42
```



Wraps any Callable Objects

Any object with a [[Call]] internal

Not limited to ordinary functions

- Function
- arrow functions
- bound functions
- Proxy wrapped functions

Wrapped in Both Directions

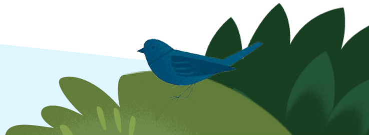
The API allows sending and receiving callable objects

```
const realm = new Realm();
const doSomething = realm.evaluate('(x, cb) => cb(x * 2)');

doSomething( 2, (done => console.log(done)) );

// doSomething.[[WrappedTargetFunction]] === (x, cb) => cb(x * 2);

// cb.[[WrappedTargetFunction]] === done => console.log(done);
```

Non Callable Objects #1

Any attempt to access Non Callable Object values will throw a TypeError.

```
const realm = new Realm();

try {
   realm.evaluate('[]');
} catch (err) {
   err.constructor === TypeError; // evaluates to true
}
```



Non Callable Objects #2

Wrapped functions can't receive non-callable objects

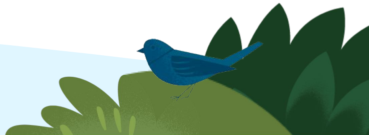
```
const realm = new Realm();
realm.evaluate('globalThis.called = false');

const doSomething = realm.evaluate('() => globalThis.called = true');
try {
   doSomething({});
} catch (err) {
   err.constructor === TypeError; // evaluates to true
   realm.evaluate('globalThis.called'); // evaluates to false
}
```

Non Callable Objects #3

```
const realm = new Realm();
const doSomething = realm.evaluate(`(wrappedTainted) => {
   try {
       wrappedTainted();
   } catch (err) {
       return err.constructor === TypeError;
   }
}`);

const tainted = () => { return {}; };
doSomething(tainted); // returns true
```

Non Callable Objects #4

```
const realm = new Realm();
const doSomething = realm.evaluate(`(wrappedArray) => {
   try {
       wrappedArray(); // would return a new array
   } catch (err) {
       return err.constructor === TypeError;
   }
}`);

doSomething(Array); // returns true
```



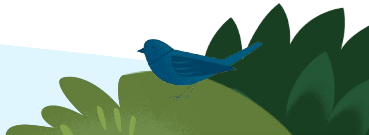
Abrupt Completion Wrapping

Abrupt completions are wrapped into a TypeError

```
const realm = new Realm();

try {
  realm.evaluate('throw new Error("custom")');
} catch (err) {
  err.constructor === TypeError; // evaluates to true
}
```

Wrapped functions won't carry properties

```
const realm = new Realm();
function fn() { return 42; }
fn.secret = 'confidential';

const doSomething = realm.evaluate(`
  (wrappedFn) => {
    wrappedFn.secret; // undefined
    return Object.prototype.hasOwnProperty.call(wrappedFn, 'secret');
  }
`);

doSomething(fn); // returns false
```

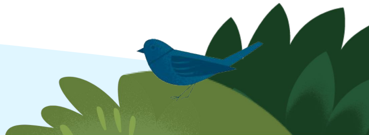


Realm.prototype.importValue

```
const realm = new Realm();
const sum = await realm.importValue('./my-framework', 'sum');

sum(2, 3); // 5
```

Realm.prototype.importValue

- `Realm.prototype.importValue` is analogous to dynamic `import()`
- It returns a promise that eventually resolves to a value of an exported name of a specified module namespace.
- The resolved value is not dynamically mapped to the module namespace.
- The resolved value goes through GetWrappedValue. Functions are subject to wrapping.



Module specifier and exported name required

```
// ./inside-code.js
export { runTests } from 'test-framework';
import './my-tests.js';

// from the incubator Realm
const r = new Realm();
const runTests = await r.importValue('./inside-code.js', 'runTests');
```

Module specifier and exported name required

(with the module blocks proposal)

```
module insideCode {
 export { runTests } from 'test-framework';
 import './my-tests.js';
}

const r = new Realm();
const runTests = await r.importValue(insideCode, 'runTests');
```

Realms Caveats

- `Realm.prototype.evaluate` is subject to some CSP directives, i.e. unsafe-eval.
- `Realm.prototype.importValue` is also subject other CSP directives, i.e. default-src.
- Functions are never unwrapped. Every evaluation wraps callables into a new wrapped function exotic.
- Wrapped Function Exotics don't have a `[[Construct]]`, and won't chain these.
- Wrapped Function Exotics' `[[Call]]` won't coerce `thisArgument` to object, this is done in regular functions' `[[Call]]`
- Wrapped Function Exotics' `thisArgument` is also subject to `GetWrappedValue`.

Resolutions

- The current proposal might be limited on cross-realm object access
- Although, it enables a proper virtualization mechanism
- The API still provides enough tools to implement membranes on top
- The wrapped exotic functions enable cross-realms callbacks in either direction

Status

- Rendered Spec
- Explainer
- SES feedback: onboard
- WIP
    - Implementer's feedback
    - TAG Review
    - Proof of Concept Membrane on top

Thank You

Outstanding discussions

Web Globals

By default, Realms includes ECMAScript intrinsics, but an instantialization hook allows the host to add more properties to the global object.

- Properties must be configurable
- Properties must not have authority, meaning they can't perform I/O or create side effects of mutation status

Module Graph

Realms need an independent resolution of modules to avoid leaking access to object values cross-realms.

- The host might reuse the module's graph I/O, but needs to instantiate a different module evaluation for each Realm.
- It is imperative for this proposal that modules work seamlessly per realm, without connection to any values from other realms. Otherwise, virtualization becomes compromised.

SharedArrayBuffer

The current realms API offer no mechanism to access shared memory buffers. This is not required for many use cases, but it become a need in a future.

We believe the API allows future extensions for such access.

For now, there is no special treatment of objects and their internals in the function wrapping model.

import without fetching a name

RFC: Realm.prototype.import

I don't often need a binding from user-code injected into a realm.

```
const r = new Realm();

const runTests = await r.importValue('test-framework', 'run');
await r.importValue('./my-tests.js', '???????');

runTests(done => console.log(done));
```

Status quo: a binding name is always required

import without fetching a name

Alternative #1

use importValue + import

```
const r = new Realm();

const runTests = await r.importValue('test-framework', 'run');
await r.import('./my-tests.js');

runTests(done => console.log(done));
```

import without fetching a name

Alternative #2

Use import with options bag, ergonomic to import assertions

```
const r = new Realm();

const runTests = await r.import('test-framework', { binding: 'run' });
await r.import('./my-tests.js');

runTests(done => console.log(done));
```

import without fetching a name

Alternative #3

Use a module namespace resolver

```
const r = new Realm();

const realmModule = await r.import('test-framework');
await r.import('./my-tests.js');

const runTests = realmModule.get('run');
runTests(done => console.log(done));
```

import without fetching a name

Alternative #1:

use importValue + import

```
const runTests =
  await r.importValue(
    'test-framework', 'run'
  );
await r.import('./my-tests.js');
```

Alternative #2:

import with options bag

```
const runTests =
  await r.import('test-framework', {
    binding: 'run'
  });
await r.import('./my-tests.js');
```

Alternative #3:

Use a module namespace resolver

```
const realmModule =
  await r.import('test-framework');
await r.import('./my-tests.js');

const runTest =
  realmModule.get('run');
```

Status Quo

importValue can still import a layered module

```
// ./inside-code.js
export { runTests } from 'test-framework';
import './my-tests.js';

// from the incubator Realm
const r = new Realm();
const runTests = await r.importValue('./inside-code.js', 'runTests');
```

Status Quo

importValue can still import a layered module

(with module blocks)

```
module insideCode {
 export { runTests } from 'test-framework';
 import './my-tests.js';
}

const r = new Realm();
const runTests = await r.importValue(insideCode, 'runTests');
```

Bikeshed

Bikeshed: ?

?