CHICAGO
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2016

goto; conference

If You're Reading .then() It's Too Late

Luke Westby

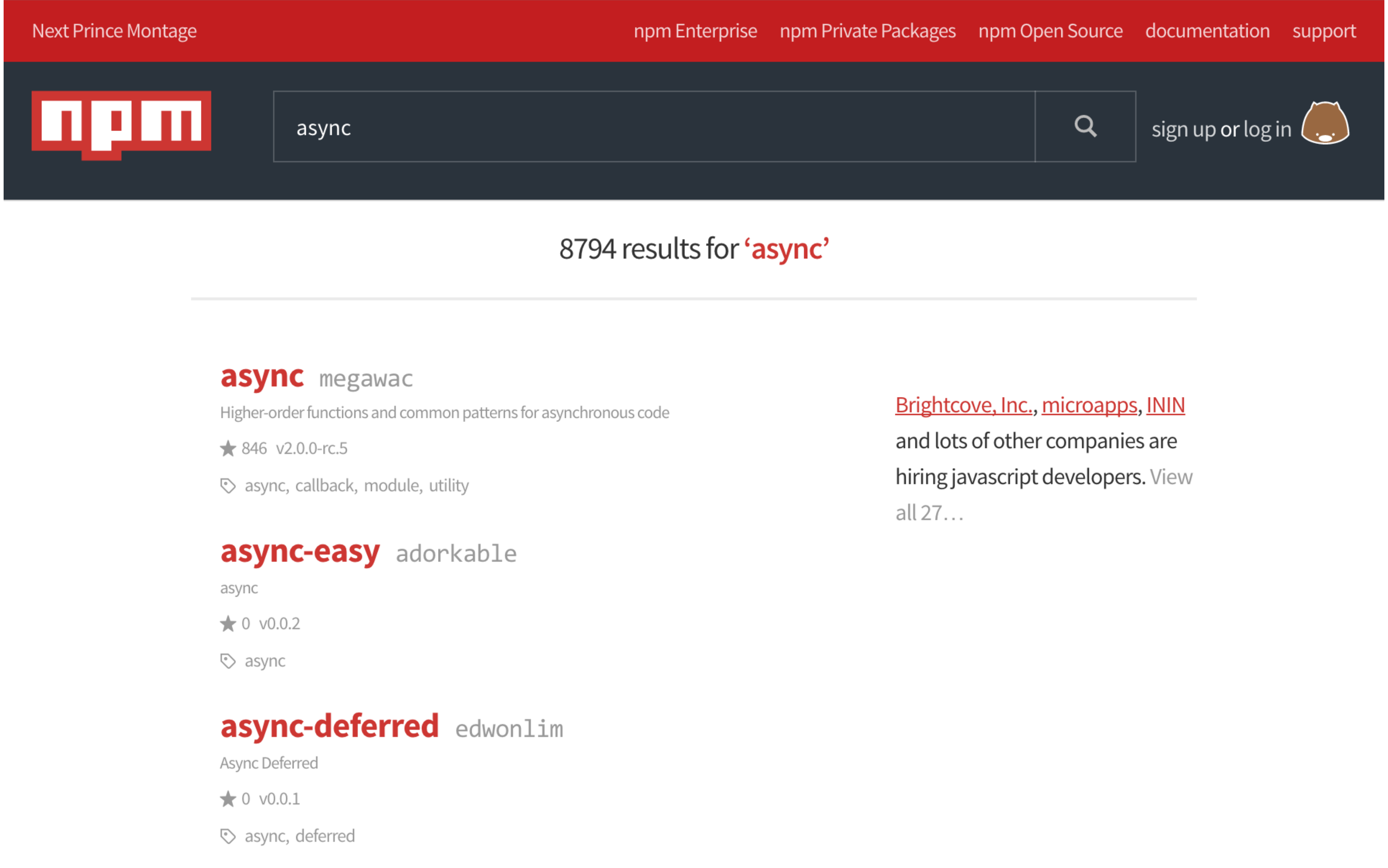
Handling effects in JavaScript

Effect:
Any interaction with the outside world

```
const request = (url) => {
  return fetch(url).then((r) => r.json());
};


const getUser = (id) => {
  return db("users").where({ id }).first()
    .then((user) => user.id);
};


const currentOffset = (el) => {
  return window.scrollY - el.scrollTop;
};
```



8794 results for ‘async’

async megawac

Higher-order functions and common patterns for asynchronous code

★ 846 v2.0.0-rc.5

async, callback, module, utility

async-easy adorkable

async

★ 0 v0.0.2

async

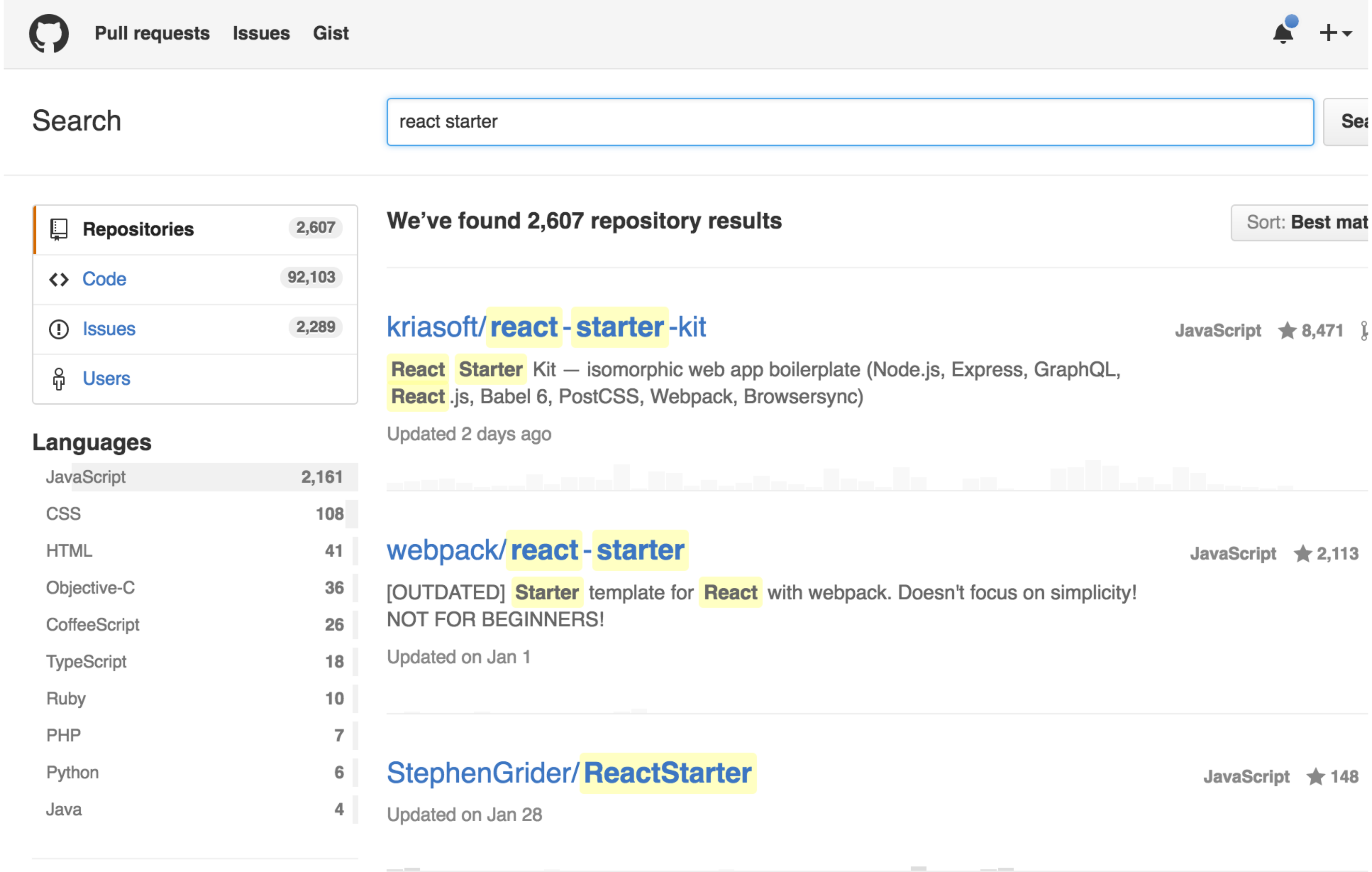
async-deferred edwonlim

Async Deferred

★ 0 v0.0.1

async, deferred

Brightcove, Inc., microapps, ININ and lots of other companies are hiring javascript developers. View all 27…



Search

react starter

Repositories	2,607
Code	92,103
Issues	2,289
Users	

Languages

JavaScript	2,161
CSS	108
HTML	41
Objective-C	36
CoffeeScript	26
TypeScript	18
Ruby	10
PHP	7
Python	6
Java	4

We've found 2,607 repository results

kriasoft/react-starter-kit — JavaScript ★ 8,471

React Starter Kit — isomorphic web app boilerplate (Node.js, Express, GraphQL, React.js, Babel 6, PostCSS, Webpack, Browsersync)

Updated 2 days ago

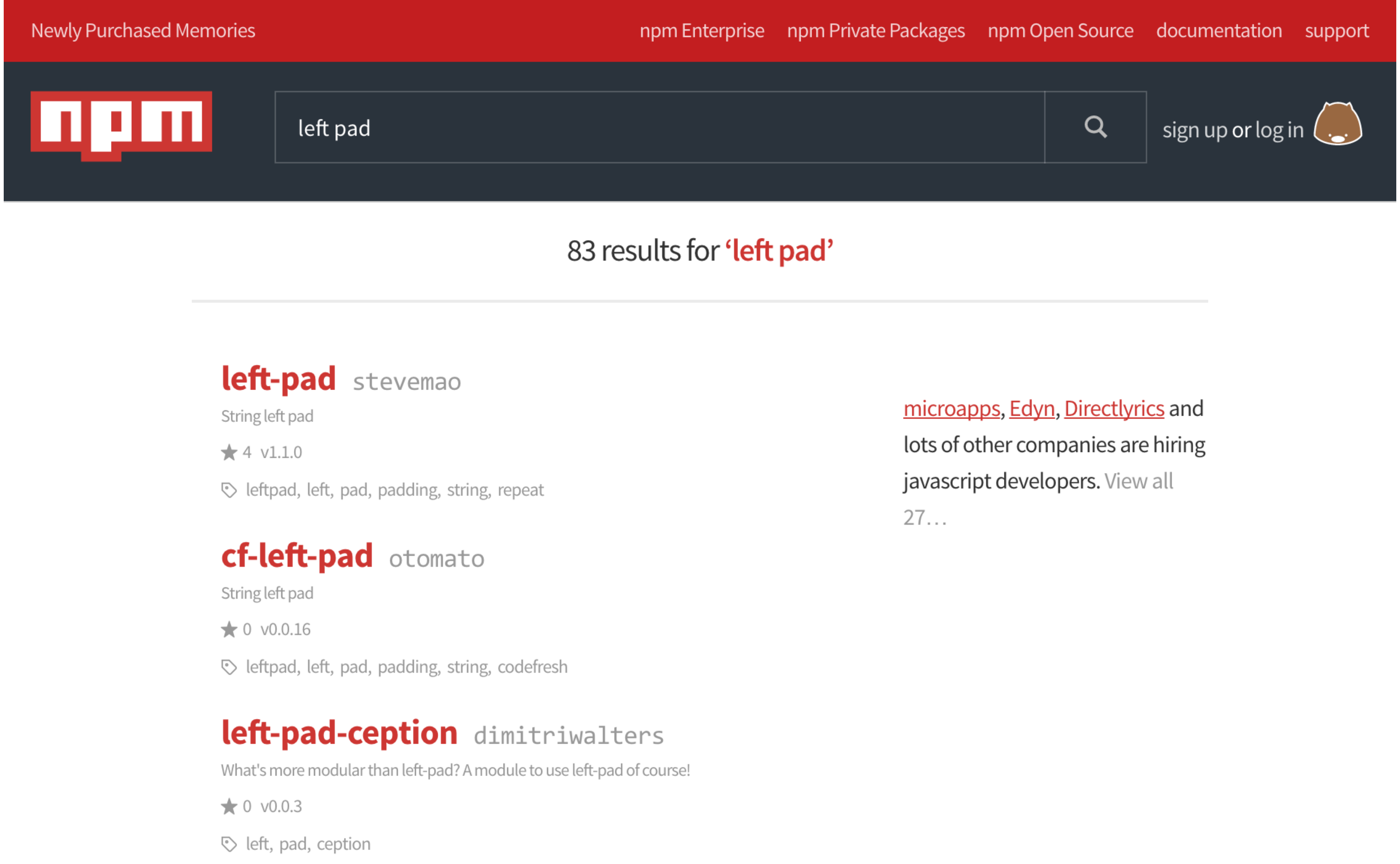
webpack/react-starter — JavaScript ★ 2,113

[OUTDATED] Starter template for React with webpack. Doesn't focus on simplicity! NOT FOR BEGINNERS!

Updated on Jan 1

StephenGrider/ReactStarter — JavaScript ★ 148

Updated on Jan 28



Newly Purchased Memories

npm Enterprise npm Private Packages npm Open Source documentation support

left pad

sign up or log in

83 results for 'left pad'

left-pad stevemao

String left pad

★ 4 v1.1.0

leftpad, left, pad, padding, string, repeat

cf-left-pad otomato

String left pad

★ 0 v0.0.16

leftpad, left, pad, padding, string, codefresh

left-pad-ception dimitriwalters

What's more modular than left-pad? A module to use left-pad of course!

★ 0 v0.0.3

left, pad, ception

microapps, Edyn, Directlyrics and lots of other companies are hiring javascript developers. View all 27…

Some things really should be the responsibility of the language

Callbacks

```
doSomeWork((error, firstResult) => {
  if (error) return respond(error, 500);
  else {
    processFirstResult(firstResult, (secondError, secondResult) => {
    if (secondError) return respond(secondError, 500);
    else {
      if (secondResult.prop) {
        doFirstBranchWork(secondResult, (firstBranchError, firstBranchResult) => {
          if (firstBranchError) respond(firstBranchError, 500);
          else respond(firstBranchResult, 200);
        });
      } else {
        doSecondBranchWork(secondResult, (secondBranchError, secondBranchResult) => {
          if (secondBranchError) respond(secondBranchError, 500);
          else respond(secondBranchError);
        });
      }
    }
  });
});
```

```
doSomeWork((error, result) => {
  if (error) respond(error, 500);
  else respond(result, 200);
});
```

Promises

```
doSomeWork()
  .then((result) => processFirstResult(result))
  .then((result) => {
    return result.prop ?
      doFirstBranchWork(result) :
      doSecondBranchWork(result);
  })
  .then((result) => respond(result, 200))
  .catch((error) => respond(error, 500));
```

🙁

```
const myPromise = new Promise((resolve, reject) => {
  doSomeWorkCallbackStyle((error, result) => {
    if (error) reject(error);
    else resolve(result);
  });
});
```

executor

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

Parameters

executor

A function that will be passed to other functions via the arguments `resolve` and `reject`. The `executor` function is executed immediately by the Promise implementation which provides the `resolve` and `reject` functions (the executor is called before the `Promise` constructor even returns the created object). The `resolve` and `reject` functions are bound to the promise and calling them fulfills or rejects the promise, respectively. The executor is expected to initiate some asynchronous work and then, once that completes, call either the `resolve` or `reject` function to resolve the promise's final value or else reject it if an error occurred.

If we’re doing async work, effects are (almost) always involved

Promises represent the *creation* of async work

If a Promise is present, it we can pretty safely assume the presence of a side-effect

side-effects

This is super easy to test:

```
(user) => user.id
```

This is less easy to test:

```
db("users").first()
 .then((user) => user.id)
```

https://www.youtube.com/watch?v=6EdXaWfoslc

```
const getUser = (id) => {
  return db("users").where({ id }).first();
};
```

vs.

```
const getUser = (id) => ({
  action: "db",
  table: "users",
  operations: [ ["where", { id }], ["first"] ]
});
```

Somewhere else…

```
const dbRunner = ({ table, operations }) => {
  const query = db(table);
  return operations.reduce(/* ... */, query);
};
```

Declare effects in application code

Pause execution of the application code to execute the effect

Resume application code when the effect has produced a result

React

```
type ReactDOMElement = {
  type: string,
  props: {
    children: ReactNodeList,
    className: string,
    ...
  },
  key: string | boolean | number | null,
  ref: string | null
};
```

Cycle.js

http://jsbin.com/numuladaqe/1/edit?js,output

In particular, lines 10-13

```
const actionHandlers = {
  [Actions.begin](request) {
    return Actions.dbCall({
      table: “users",
      operations: [
        ["where", { id: request.params.id }],
        ["first"],
      ]
    });
  },
  [Actions.dbCallSuccess](request, user) {
    return Actions.respond(user.id, 200);
  },
  [Actions.dbCallFailure](request, error) {
    return Actions.respond(error, 500);
  }
};
```

```
server.route({
  method: “GET",
  path: “/users/{id}",
  handler(request, reply) {
    runActions(runners, actionHandlers, request)
      .then((result) => {
        reply(result.data).statusCode(result.status);
      })
      .catch((error) => {
        reply(error.message).statusCode(error.status);
      });
  }
});
```

```
const actionHandlers = {
  [actions.begin](request) {
    return Actions.dbCall({
      table: "users",
      operations: [
        ["where", { id: request.params.id }],
        ["first"],
      ]
    });
  },
  [actions.dbCallSuccess](request, user) {
    return Actions.respond(user.id, 200);
  },
  [actions.dbCallFailure](request, error) {
    return Actions.respond(error, 500);
  }
};
```

```
const actionHandlers = {
  [0](request) {
    return dbCall({
      table: "users",
      operations: [
        ["where", { id: request.params.id }],
        ["first"],
      ]
    });
  },
  [1](request, user) {
    return respond(user.id, 200);
  },
  error(request, error) {
    return respond(error, 500);
  }
};
```

```
server.route({
  method: “GET",
  path: “/users/{id}",
  handler(request, reply) {
    runActions(runners, handler(request))
      .then((result) => {
        reply(result.data).statusCode(result.status);
      })
      .catch((error) => {
        reply(error.message).statusCode(error.status);
      });
  }
});
```

Generators

```
const handler = function* (request) {
  try {
    const user = yield {
      type: “db",
      table: “users",
      operations: [/* ... */]
    };

    return respond(user.id, 200);
  } catch (error) {
    respond(error.message, 500);
  }
};
```

```
const iterator = handler({ params: { id: 1 } });

let next = iterator.next();
assertDeepEqual(next.value, { type: "db", /* ... */ });

next = iterator.next({ id: 1, name: "Luke", /* ... */ });
assertDeepEqual(next.value, respond(1, 200));

// or

next = iterator.throw(Error("NOPE!"));
assertDeepEqual(next.value, respond("Luke", 500));
```

Two problems:

1. Nested calls are uncomfortable
2. Parallelization isn’t a given

```
const handler = function* (request) {
  try {
    const user = yield {
      type: "db",
      table: "users",
      operations: [/* ... */]
    };

    return respond(user.id, 200);
  } catch (error) {
    respond(error.message, 500);
  }
};
```

```
const handler = function* (request) {
  try {
    const userId = ?? getUserId(request); ??
  } catch (error) {
    respond(error.message, 500);
  }
};
```

```javascript
const handler = function* (request) {
  try {
    const userId = yield spawn(getUserId, request);
  } catch (error) {
    respond(error.message, 500);
  }
};
```

Only one level of continuation is possible with Generators

What if we could yield deeply?

One-shot delimited continuations with effect handlers

Sebastian Markbåge

https://esdiscuss.org/topic/one-shot-delimited-continuations-with-effect-handlers

```
const funcWithEffect = (request) => {
  const user = perform { type: “db”, /* ... */ };
  return user.id;
}

try {
  const result = funcWithEffect(request);
} catch effect -> [{ type, ...details }, continuation] {
  runners[type](details)
    .then((result) => continuation(result))
    .catch((error) => { throw error; });
}
```

If this can be implemented efficiently,
it should be implemented

Again, some things really should be the
responsibility of the language

@luke_dot_js

@lukewestby

hello@lukewestby.com

Raise

Questions?

CHICAGO

INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE 2016

goto; conference

Please

Remember to rate this session

Thank you!

follow us @gotochgo

Conference: May 24th-25th / Workshops: 23th-26th