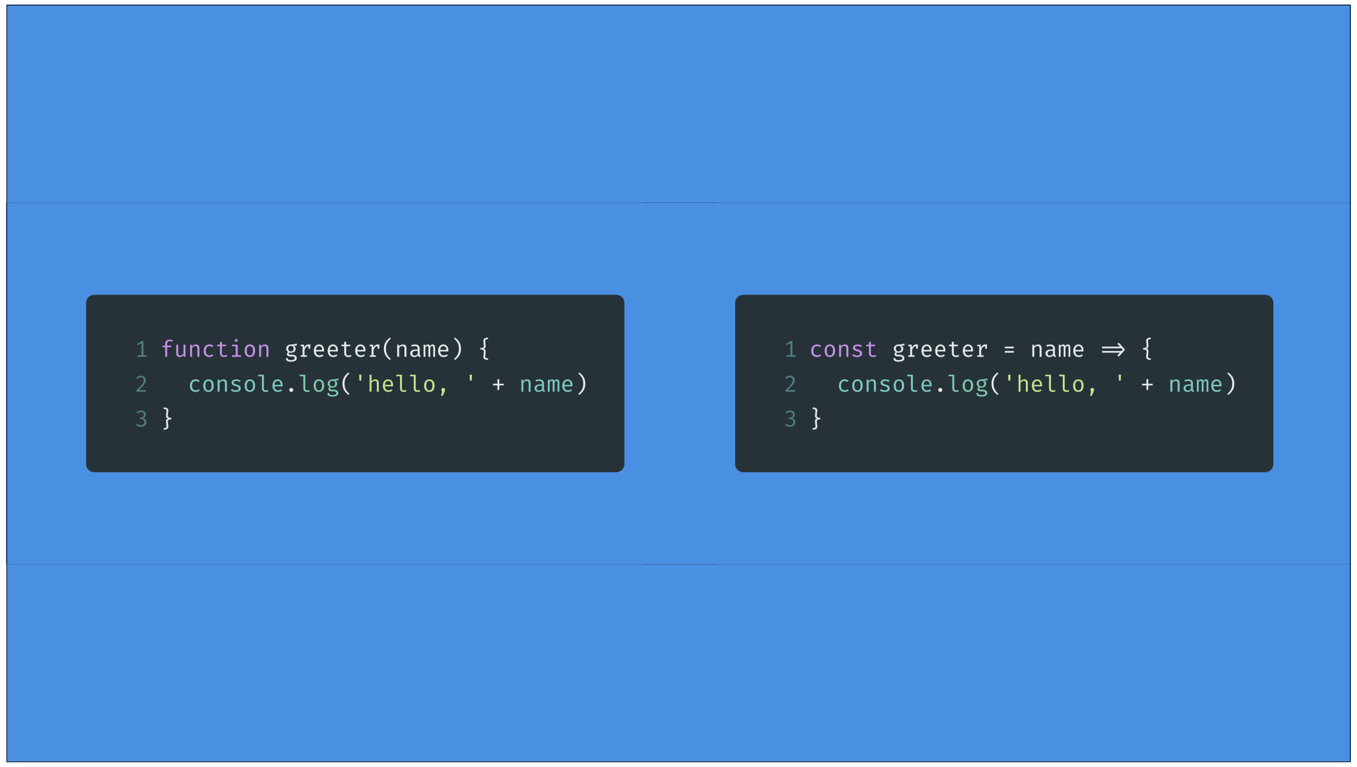
Javascript Promises

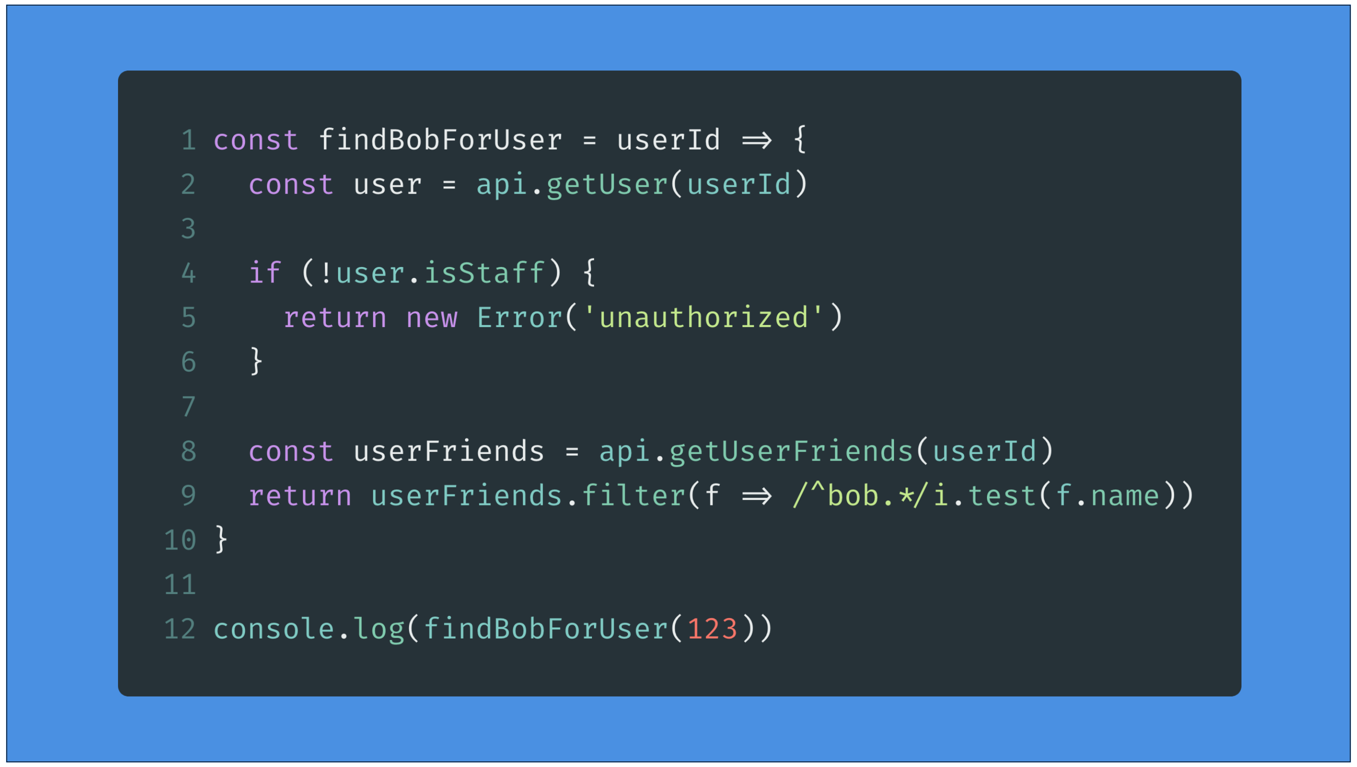
Elijah Wilson
Founder/CEO @ HookActions



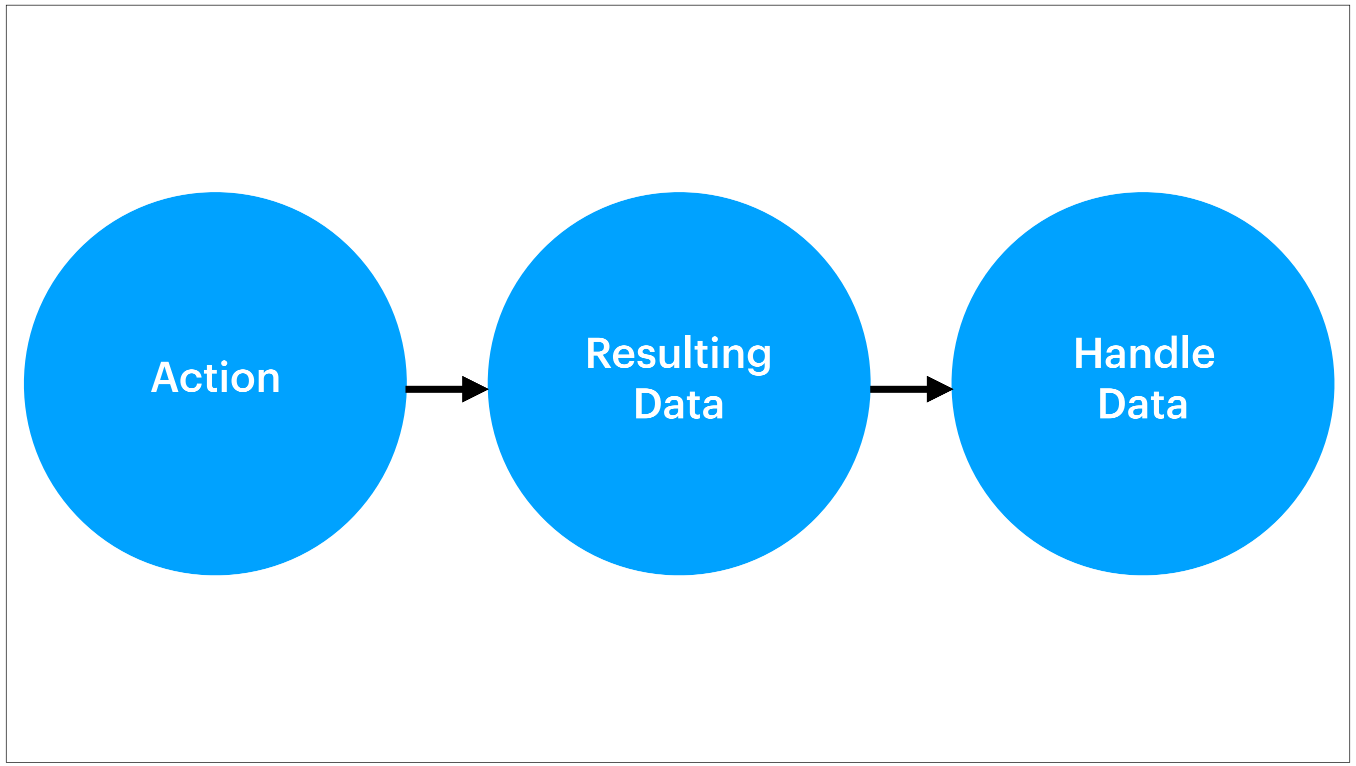
```js
function greeter(name) {
  console.log('hello, ' + name)
}
```

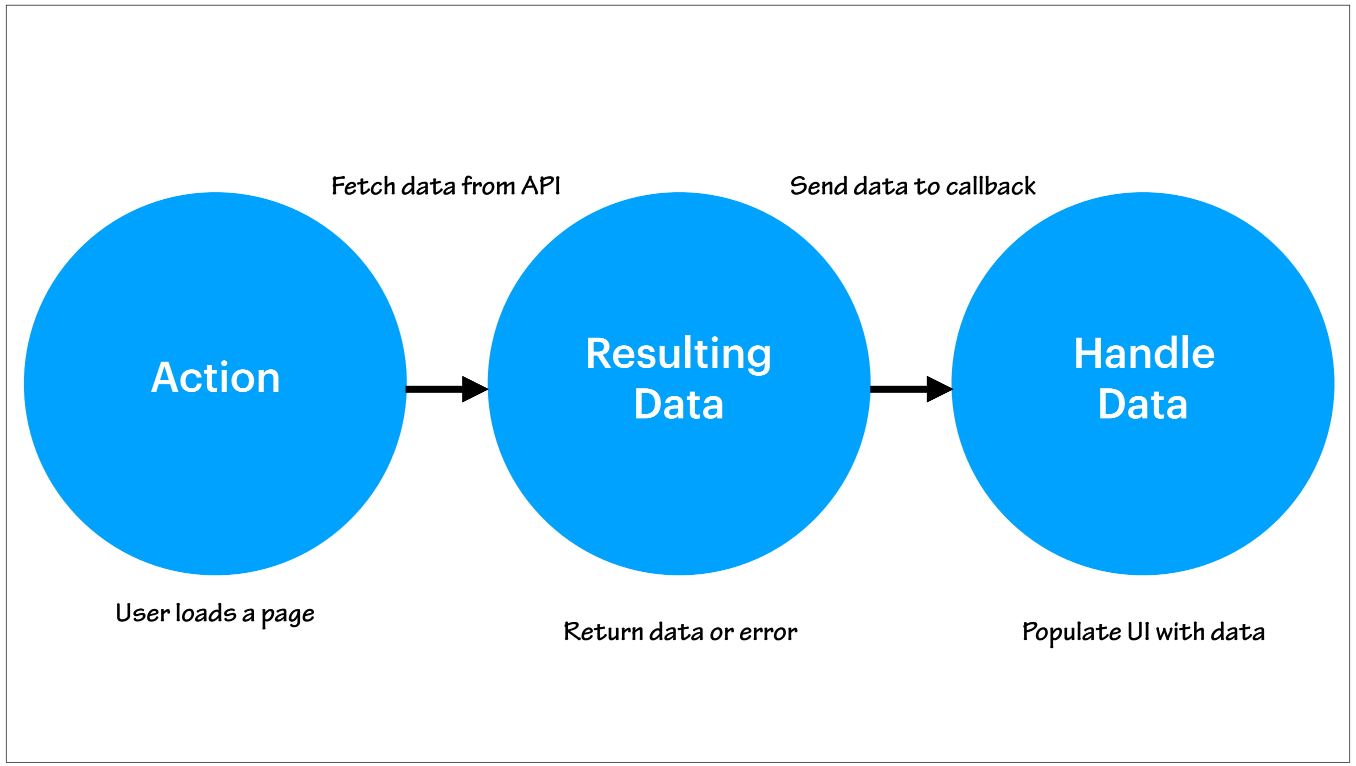
```js
const greeter = name => {
  console.log('hello, ' + name)
}
```

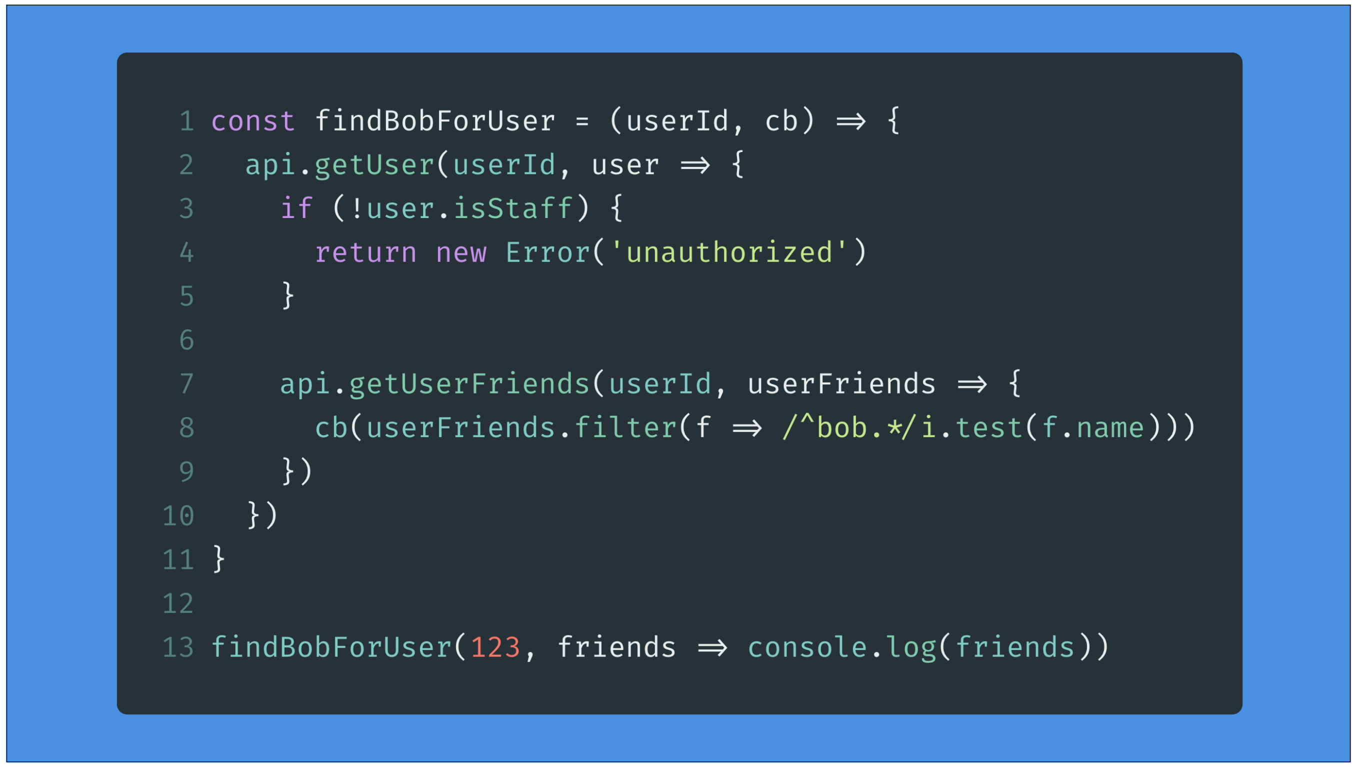
Quick ES6 primer, these two functions are equal. The left is the old way and the right is the new ES6 way of writing the same function. I'll be using the new ES6 way of writing functions in the code examples.


```javascript
const findBobForUser = userId => {
  const user = api.getUser(userId)

  if (!user.isStaff) {
    return new Error('unauthorized')
  }

  const userFriends = api.getUserFriends(userId)
  return userFriends.filter(f => /^bob.*/i.test(f.name))
}

console.log(findBobForUser(123))
```

How we might find a friend of a given user who's name is "bob". If we think of the "api" object as a traditional synchronous interface that calls an external api, the usage might look like this. And there's nothing wrong with this, it's how we're taught to code. The problem becomes that with Javascript, we won't be able to get code that looks like this. With some new keywords (async/await) we can get pretty close but not 100%.

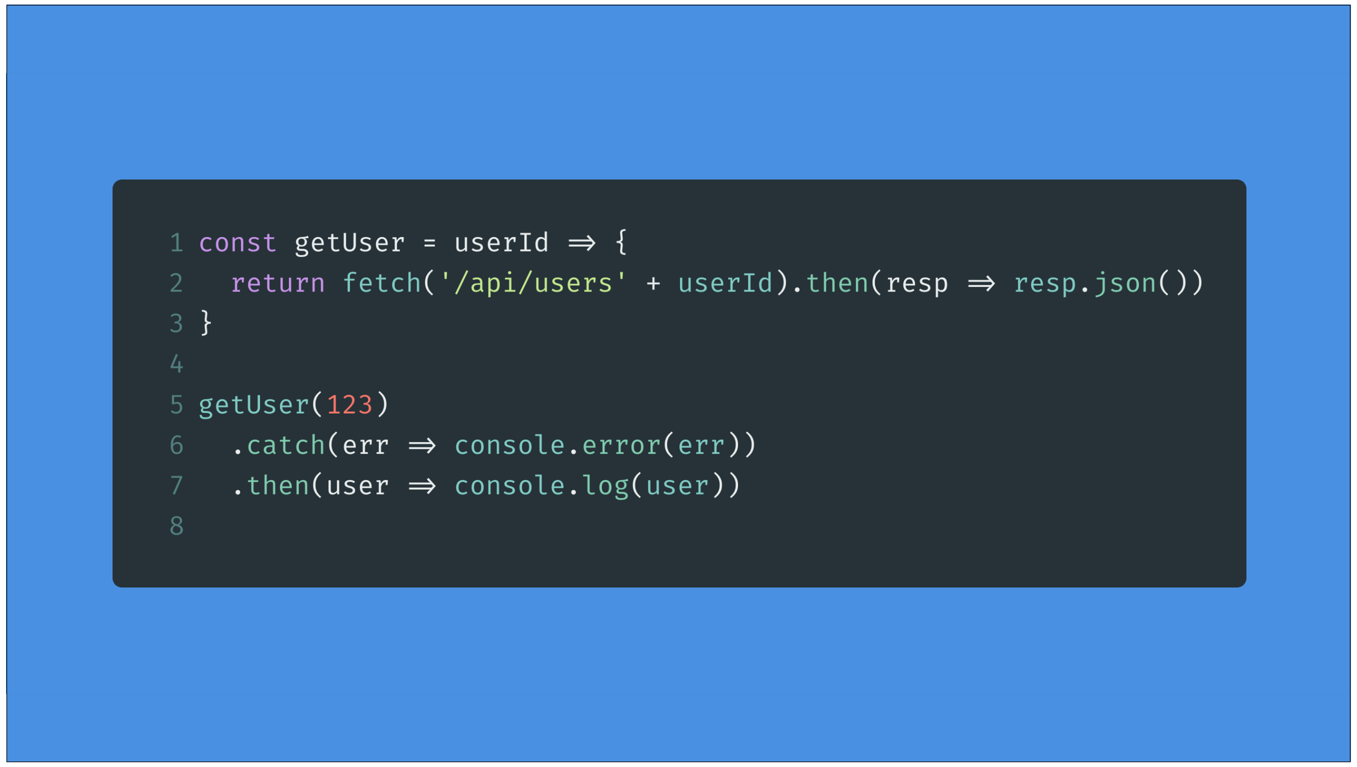

Action
Resulting Data
Handle Data


Fetch data from API
Send data to callback
Action
Resulting Data
Handle Data
User loads a page
Return data or error
Populate UI with data



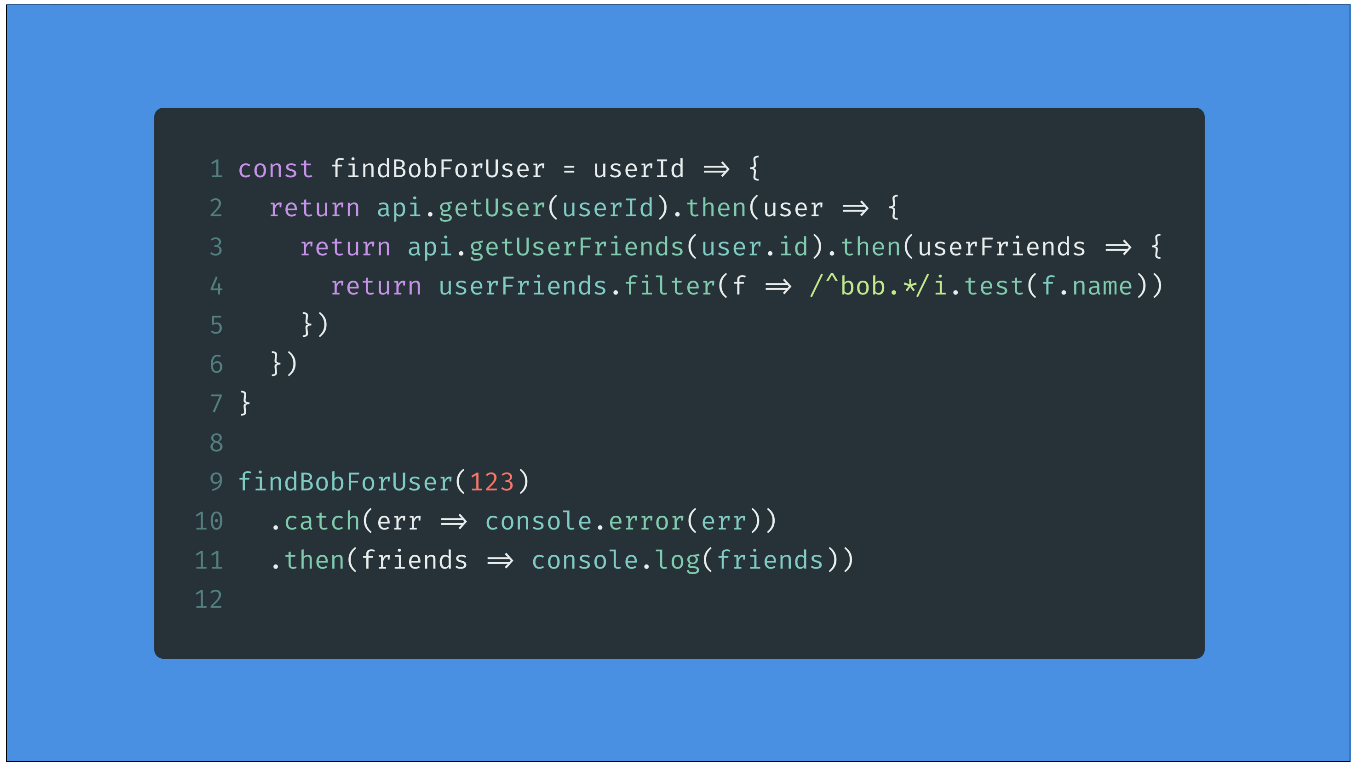
```
const findBobForUser = (userId, cb) ⇒ {
  api.getUser(userId, user ⇒ {
    if (!user.isStaff) {
      return new Error('unauthorized')
    }

    api.getUserFriends(userId, userFriends ⇒ {
      cb(userFriends.filter(f ⇒ /^bob.*/i.test(f.name)))
    })
  })
}

findBobForUser(123, friends ⇒ console.log(friends))
```

Callback hell, without Promises, maybe using XHR. This is actually not horrible, but imagine if instead of the simple console.log we needed to do something else with the resulting friends!



```
const getUser = userId => {
  return fetch('/api/users' + userId).then(resp => resp.json())
}

getUser(123)
  .catch(err => console.error(err))
  .then(user => console.log(user))

```

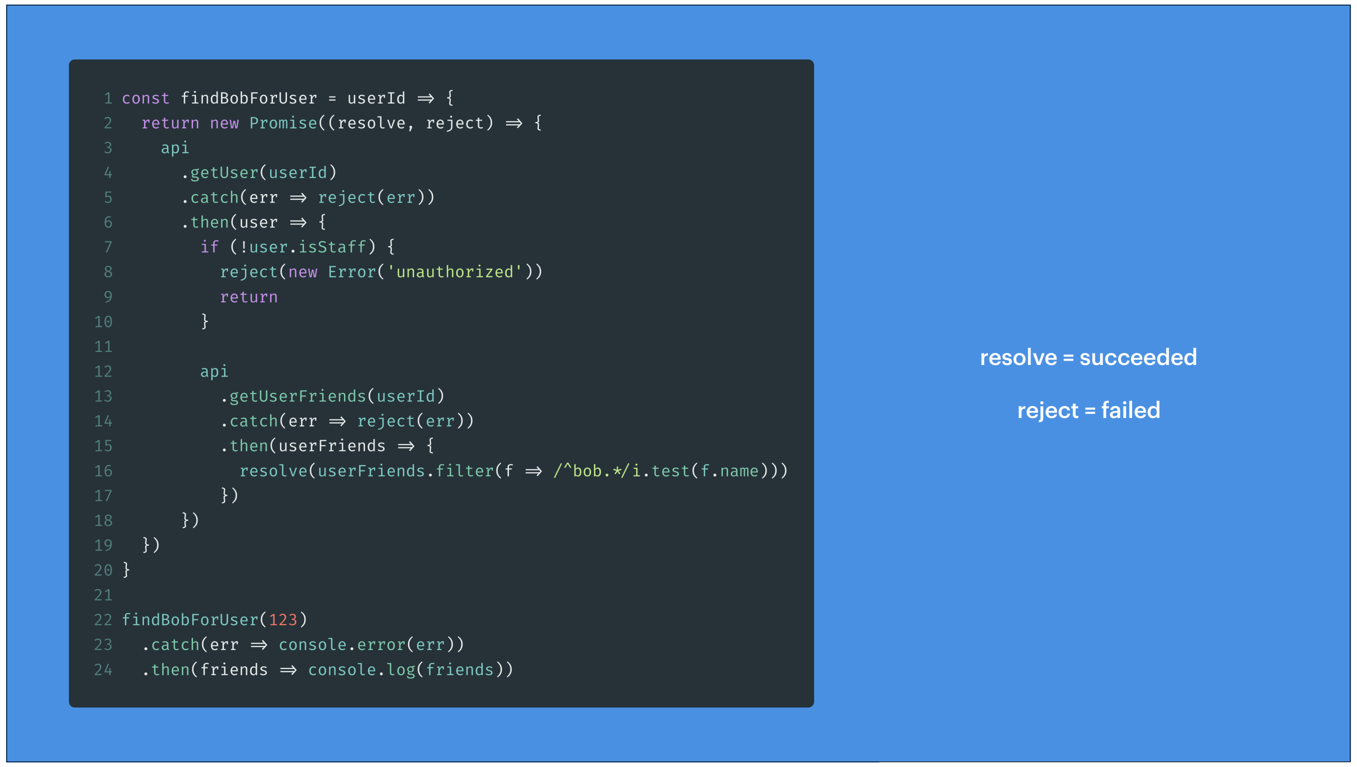
Let's take a simple promise example. Maybe we're writing the api module and are defining the getUser method. We can use the fetch method (a newer replacement to xhr) which itself uses promises. resp.json() is also a promise! If you return a promise in the "then" callback, your outer function is then working with that promise's result.



```
const findBobForUser = userId => {
  return api.getUser(userId).then(user => {
    return api.getUserFriends(user.id).then(userFriends => {
      return userFriends.filter(f => /^bob.*/i.test(f.name))
    })
  })
}

findBobForUser(123)
  .catch(err => console.error(err))
  .then(friends => console.log(friends))
```

Let's say we didn't need the custom logic around checking if the user is staff, we could write something like this. This actually isn't too bad! It's getting close to callback hell, but it's pretty manageable.

However, the problem is that we do actually need to check if the user is staff to search for Bob. We could write our own Promise that would fail if the user is not staff.



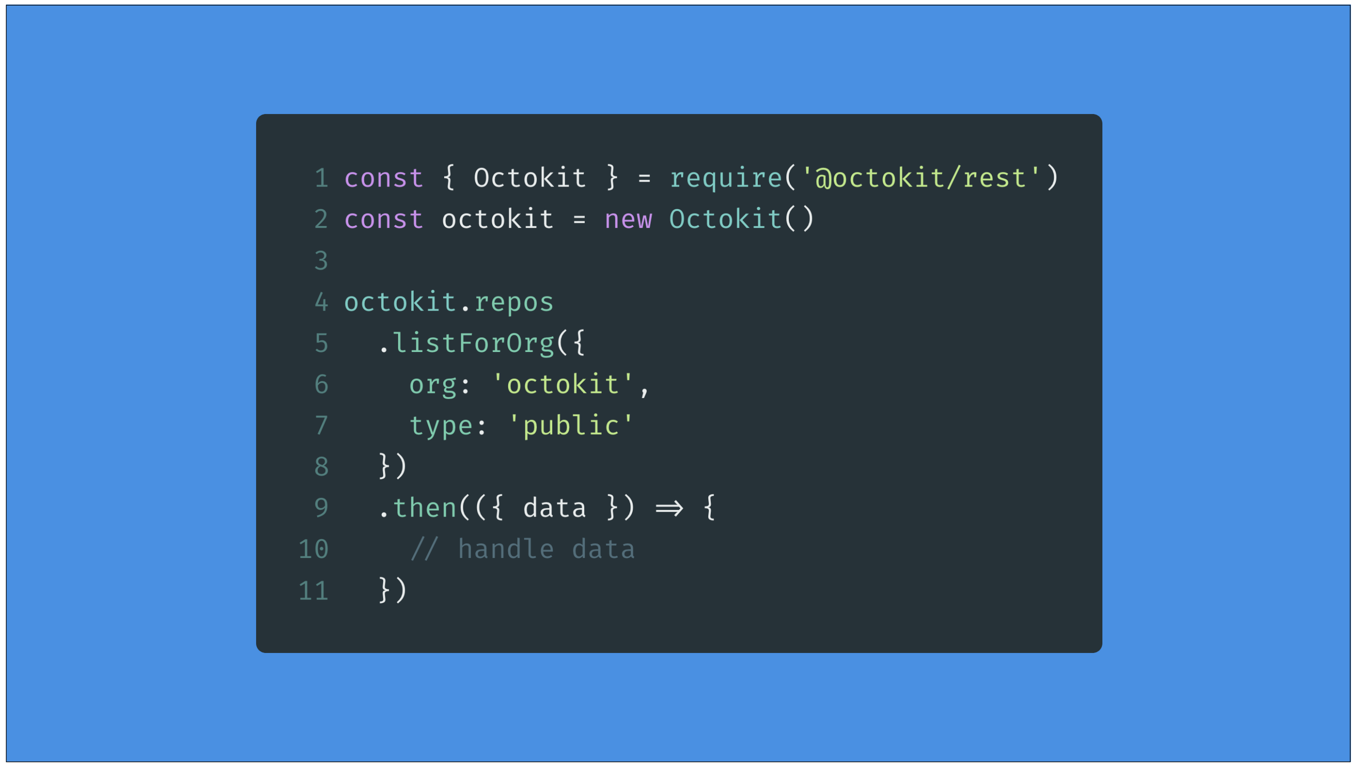
```javascript
const findBobForUser = userId => {
  return new Promise((resolve, reject) => {
    api
      .getUser(userId)
      .catch(err => reject(err))
      .then(user => {
        if (!user.isStaff) {
          reject(new Error('unauthorized'))
          return
        }

        api
          .getUserFriends(userId)
          .catch(err => reject(err))
          .then(userFriends => {
            resolve(userFriends.filter(f => /^bob.*/i.test(f.name)))
          })
      })
  })
}

findBobForUser(123)
  .catch(err => console.error(err))
  .then(friends => console.log(friends))
```

resolve = succeeded

reject = failed

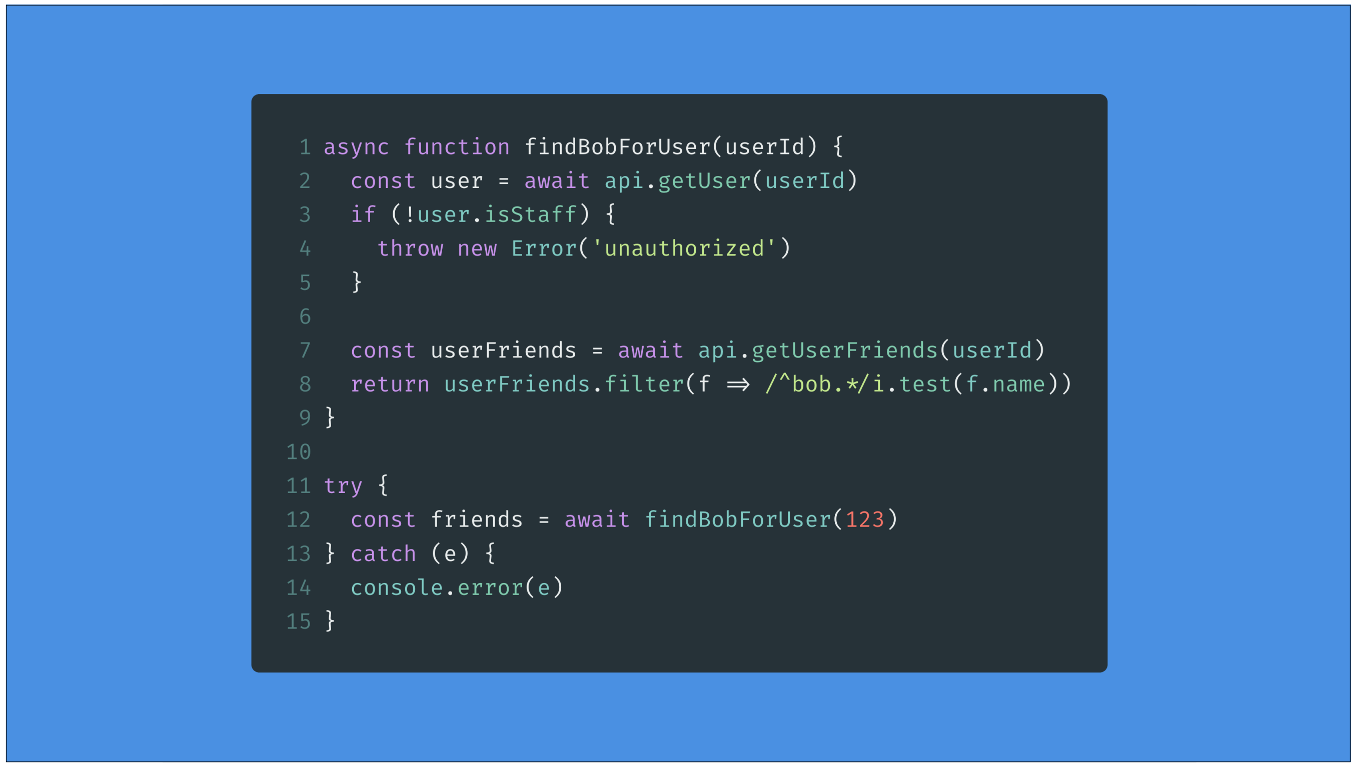
Now let's go back to the more complex example of finding the users named bob. If we want to write our own promise with custom resolve/reject logic, this is what it might look like.

The resolve function says, this promise succeeded, here's the result. The reject function is the opposite. It says this promise failed, here's an error.

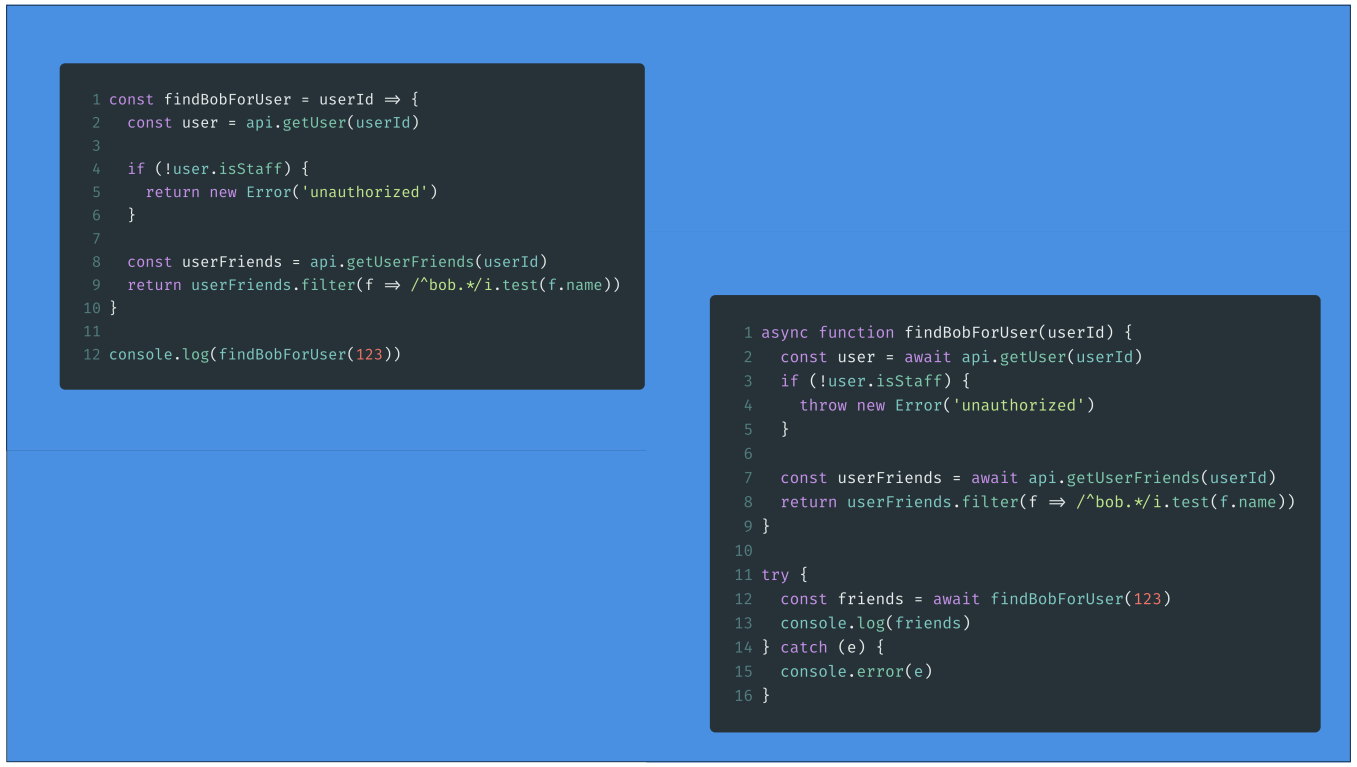


```
const { Octokit } = require('@octokit/rest')
const octokit = new Octokit()

octokit.repos
  .listForOrg({
    org: 'octokit',
    type: 'public'
  })
  .then(({ data }) => {
    // handle data
  })
```

If we're building a Github integration, we could use their javascript SDK (https://github.com/octokit/rest.js) and list all the public repos associated with the "octokit" organization. When working with HTTP APIs, you'll most likely be working with promises.



```
async function findBobForUser(userId) {
  const user = await api.getUser(userId)
  if (!user.isStaff) {
    throw new Error('unauthorized')
  }

  const userFriends = await api.getUserFriends(userId)
  return userFriends.filter(f => /^bob.*/i.test(f.name))
}

try {
  const friends = await findBobForUser(123)
} catch (e) {
  console.error(e)
}
```

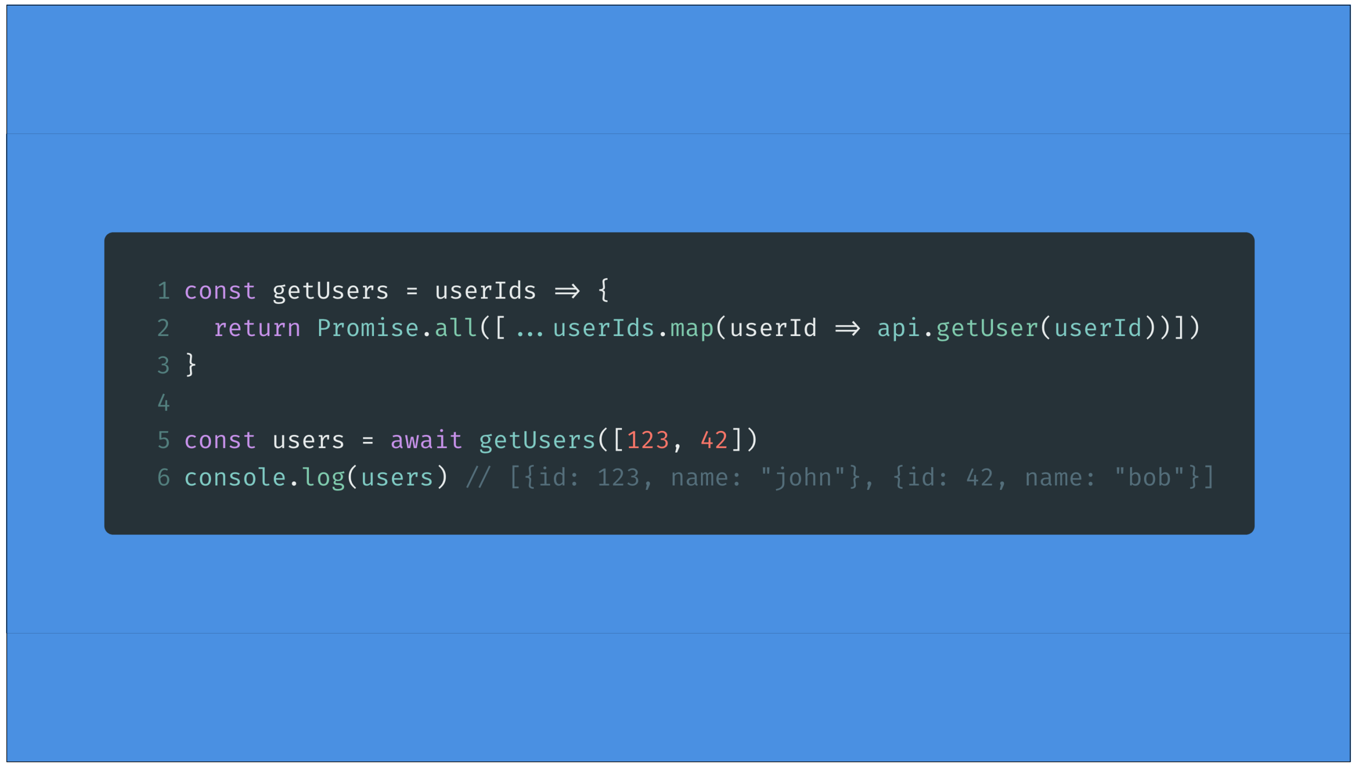


```
const findBobForUser = userId => {
  const user = api.getUser(userId)

  if (!user.isStaff) {
    return new Error('unauthorized')
  }

  const userFriends = api.getUserFriends(userId)
  return userFriends.filter(f => /^bob.*/i.test(f.name))
}

console.log(findBobForUser(123))
```

```
async function findBobForUser(userId) {
  const user = await api.getUser(userId)
  if (!user.isStaff) {
    throw new Error('unauthorized')
  }

  const userFriends = await api.getUserFriends(userId)
  return userFriends.filter(f => /^bob.*/i.test(f.name))
}

try {
  const friends = await findBobForUser(123)
  console.log(friends)
} catch (e) {
  console.error(e)
}
```

If we look at the first example again, and compare it to the async/await version. It's almost identical. These keywords introduced in ES2017 allow us to write asynchronous code in a way that feels synchronous.

With async/await,
why have Promises?



```
const getUsers = userIds => {
  return Promise.all([...userIds.map(userId => api.getUser(userId))])
}

const users = await getUsers([123, 42])
console.log(users) // [{id: 123, name: "john"}, {id: 42, name: "bob"}]
```

Promise.all will resolve a list of the return values of each of the promises or reject on the first promise that rejects. This means that if the first api.getUser promise fails, the getUsers promise will fail with that first error. Async/await help us write easy to read code but in the end we're still using promises.

https://git.io/JvKKZ