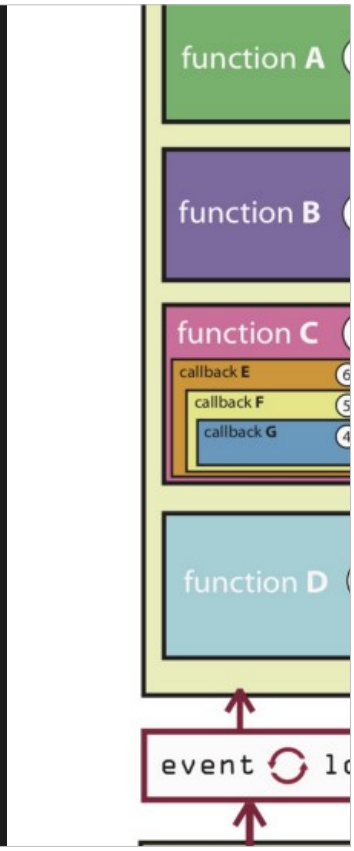
Contemporary Web Development
Lesson 5

https://www.youtube.

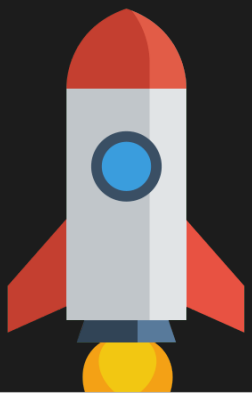
Asynchronous Programming

Back to the runtime model



When we offload IO to a separate thread with the browser API, we need to somehow get the reply back. That's what the event queue is for. The difference is how we do it conveniently and also handle errors.

Spaceship Launch



Launch Sequence

1) Start engines → Power must be above 0.8
2) Deploy Spring
3) Release Spring

*** Every stage could fail on its own with its own reason**

The Synchronous Way

```
function launchSync() {
      log("Attempt " + attempts)

      let result = Spaceship.startEnginesSync()
      If (result.status != "success") {
          error(result.status)
      }
      else If (result.power < 0.5) {
            error("Not enough engine power")
      }  else {
           log("Engines started!");
           let result = Spaceship.deploySpringSync();
             If (result.status != "success") {
                  error(result.status)
           }   else {
                ……. // Move on
      }
}
```

- Not very maintainable.

It's easy to lose track of where an error happens and where it is handled, there is no standard for failure.

Exception handling

Errors bubble up and you can catch them in one dedicated location!

Also good for handling unwanted crashes.

```
function launchSync() {
   try {
      log("Attempt " + attempts)
      let power = Spaceship.startEnginesSync()
      if (power < 0.5)  {
         throw new Error("Not enough engine power! " + power.toString())
      }
      log("Engines started!");
      Spaceship.deploySpringSync();
     // .. Go on
    }
   catch(err) {
      log("Launch failed :( (" + err + ")")
   }
   finally {
      attempts += 1;
   }
}
```

Custom Errors

./space-error.js

```
export default class SpaceError extends Error {
  constructor(...args) {
    super(...args)
  }
}
```

./spaceship.js

```
if (Math.random() < 0.3) {
  throw new SpaceError("Too slippery");
}
```

./index.js

```
function launchSync() {
  try {
    ...
  }
  catch(err) {
    if (err instanceof SpaceError) {
      log("Launch failed :( (" + err + ")")
    } else {
      log("Unknown error " + err);
    }
  }
  finally {
    attempts += 1;
  }
}
```

The Async way

```
export function startEngines() {
   setTimeout(()=> {
         if (Math.random() >= 0.2) {
            let power = Math.random();
                  ...
         } else {
                  ….
         }
      },2000);
   })
}
```

So how do we notify the result back to the client?

Callbacks

./spaceship.js

```
export function startEnginesCB(callback) {
  if (output) {
    output("WRRROOOOOMMM....");
  }
  setTimeout(()=> {
    if (Math.random() >= 0.2) {
      let power = Math.random();
      callback(null, power)
    } else {
      callback("loose screw", 0)
    }
  },2000);
}
```

./index.js

```
Spaceship.startEnginesCB((err, power) => {
  if (err) {
    log("Launch failed :( (" + err + ")")
  } else {
    if (power >= 0.8) {
      Spaceship.deploySpringCB((err) => {
                if (err) {
                }
                else {
                    // .. Go on
                }
        });
    }
});
```

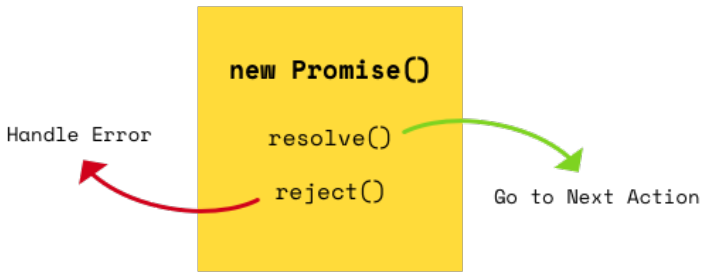
Quickly becomes messy

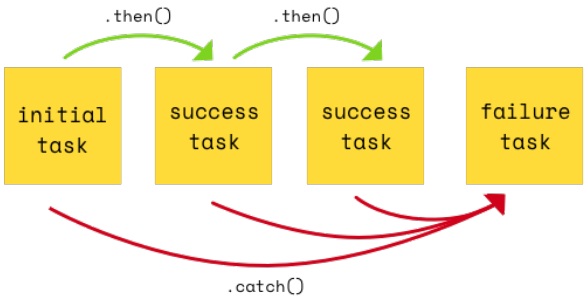
And no exception handling whatsoever

```
function launchCB() {
  try {
    Spaceship.startEnginesCB((err, power) => {
      if (err) {
        log("Launch failed :( (" + err + ")")
      } else {
        if (power < 0.8) {
          throw new SpaceError("Not enough engine power!")
        } else {
          log("Engines started!");
          Spaceship.deploySpringCB((err) => {
            if (err) {
              log("Launch failed :( (" + err + ")")
            } else {
              log("Spring deployed!");
              Spaceship.releaseSpringCB((err) => {
                if (err) {
                  log("Launch failed :( (" + err + ")")
                } else {
                  log("Spring released!");
                  log("LAUNCH SUCCESSFUL!");
                  fly();
                }
              })
            }
          })
        }
      }
    })
  } catch (err) {
    console.log("Spaceship Exception", err);
  }
}
```

With async callbacks you cannot use try/catch

Promises
A standard for Asynchronous tasks





Promises

./spaceship.js

```
export function startEngines() {
  return new Promise(function(resolve, reject) {
    if (output) {
      output("WRRROOOOOMMM....");
    }
    setTimeout(()=> {
      if (Math.random() >= 0.2) {
        let power = Math.random();
        resolve(power);
      } else {
        reject(new SpaceError("loose screw"));
      }
    },2000);
  })
}
```

./index.js

```
function launchPromise() {
  log("Attempt " + attempts)
  Spaceship.startEngines()
  .then((power) => {
    if (power < 0.8) {
      throw new SpaceError("Not enough engine po
    } else {
      log("Engines started!");
      return Spaceship.deploySpring();
    }
  })// … Go on
  .catch((err) => {
    log("Launch failed :( (" + err + ")")
  })
  .finally(() => {
    attempts += 1;
  })
}
```

The goal: Never use callbacks again.
Always Promisify

You can chain promises by returning a new promise in then() {}

(Pseudo) Parallel tasks

./spaceship.js

```
export function startEnginesMulti() {
  return Promise.all([
      startLeftEngine(),
      startRightEngine()
  ])
}
```

./index.js

```
Spaceship.startEnginesMulti()
.then((powers) => {
  console.log(powers);
  if (powers[0] < 0.8 || powers[1] < 0.8)  {
    throw new SpaceError(
            "Not enough engine power! " +
             powers.toString()
     )
  } else {
    log("Engines started!");
    return Spaceship.deploySpring();
  }
 })
 .then(() => {
  ..
})
```

Pseudo because it's still single threaded

Super Contemporary ES8: Async/Await

```
# npm install --save-dev @babel/plugin-transform-runtime
# npm install --save @babel/runtime
```

webpack.config.js

```
use: {
  loader: 'babel-loader',
  options: {
    presets: ['@babel/env'],
    plugins: ["@babel/plugin-transform-runtime"]
  }
}
```

Async/await

./index.js

```
async function launch() {
  try {
    log("Attempt " + attempts)
    let powers = await Spaceship.startEnginesMulti()
    await Spaceship.deploySpring();
    await Spaceship.releaseSpring();
  }
  catch(err) {
  }
  finally {
    attempts += 1;
  }
}
```

Any function calling await must be declared async

Still uses promises, so the spaceship modules remains the same! Can combine with Promise.all()

Python 3.0 has a similar interface (asyncio), so it's good to know.

If you want to catch an error from a specific stage, just add another try/catch

Exercise

- The spaceship now has another function – fixSpring()
- It can be activated only in case deployString() failed, then the fix can either fail or succeed (Promise resolves or rejects).
- Add an attempt to fix the string in case the deployment fails.
- If the fix succeeds – output a "Fix was successful!" message and continues down the initiation sequence.
- If the deployment fails – It should display the regular "Launch failed" message with a reason "Spring fix failed".
- **The code doesn't have to run, you can use pseudo code.**