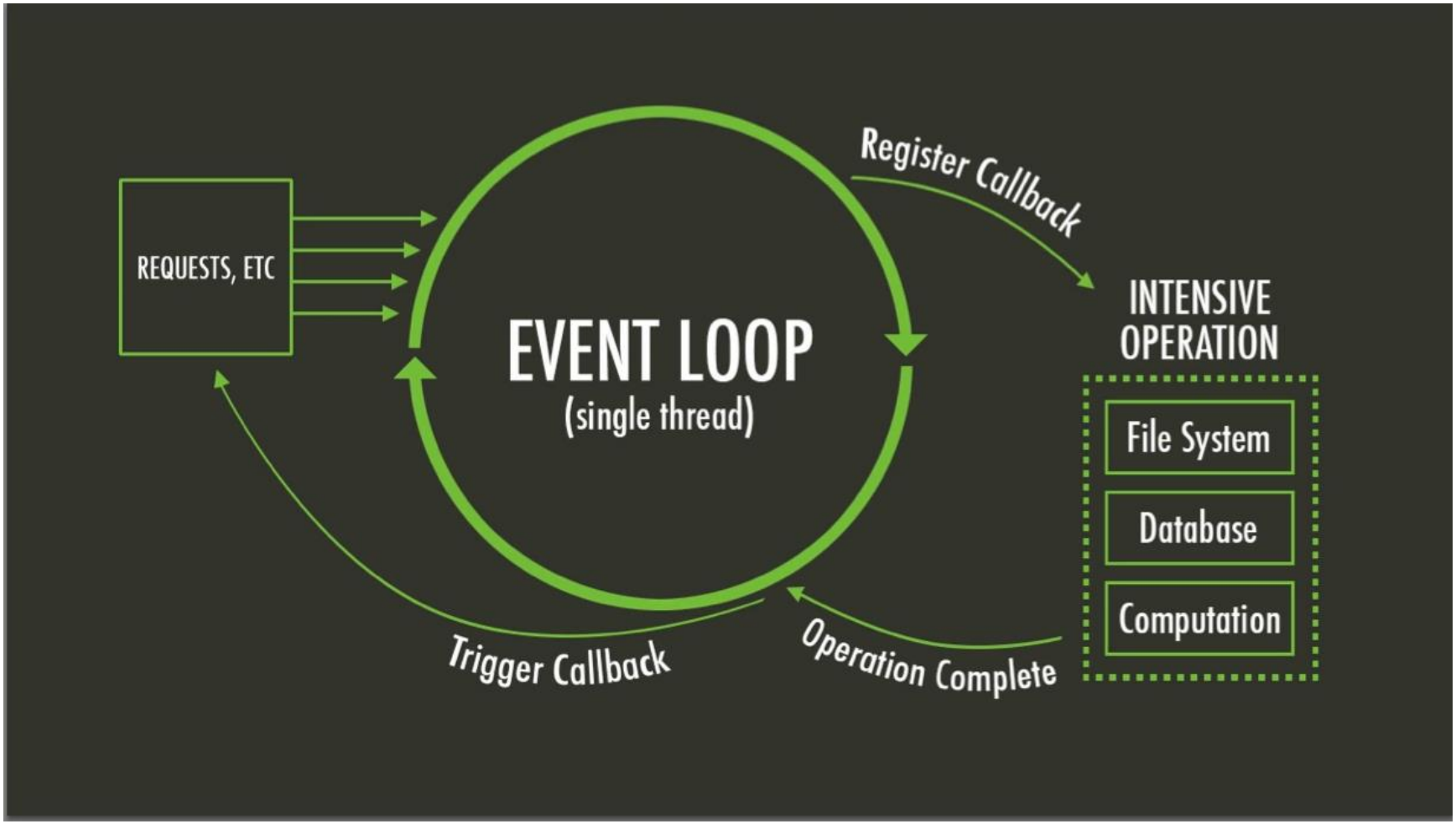
ECMAScript 2015

Les promesses

Programmation asynchrone en JS

Comment marche le JS ?

- Les moteurs JavaScripts sont monothread
- Afin de ne pas bloquer l'exécution (quasiment) tout est gérer en asynchrone
- Le moteur utilise une event loop pour traiter les évènements qui sont déclenchés
- Les évènements sont des actions utilisateurs, le résultat de calculs asynchrones, des accès réseaux, accès disques, etc.
- Il est possible de reporter un traitement à plus tard (setTimeout, requestAnimationFrame)
- Il est important de bloquer le moins possible l'event loop


REQUESTS, ETC
EVENT LOOP
(single thread)
Register Callback
INTENSIVE OPERATION
File System
Database
Computation
Trigger Callback
Operation Complete

Comment gérer l'asynchrone

- Traditionnellement on utilise des « callbacks »
- On passe en paramètre à une fonction asynchrone les différentes méthodes qu'elle pourra exécutée plus tard

```
document.addEventListener("click", function(e) {
  // Do something when someone click
});


$.ajax('url', {
  success: function() {},
  error: function()
});
```

Les problèmes

- Les callbacks posent plusieurs problèmes :
  - Il est difficile de s'y retrouver
  - Les erreurs ne sont pas propagées

- Lorsque notre app/site possède de nombreux appels asynchrone on s'y perd rapidement
- Si on enchaîne les appels asynchrones c'est encore pire

Callback Hell

```
onClick(function(event) {
  getSomething(function(data) {
    var newData = process(data);
    uploadData(newData,function() {
      // are we done yet ?
    })
  });
});
```

Callback Hell avec erreurs

```
onClick(function(errorCb, event) {
  getSomething(function(errorCb, data) {
    var newData = process(data);
    uploadData(newData,function() {
      // are we done yet ?
    }, function(err) {
      errorCb(err);
    })
  }, function(err) {
    errorCb(err);
  });
},function(err) {
    // manage errors...
});
```

Les promesses

Qu’est ce qu’une promesse ?

- Les promesses sont des objets qui encapsulent un appel asynchrone
- Une promesse possède plusieurs états possibles :
  - En attente : l’opération est en cours
  - Tenue : l’opération s’est finie avec succès
  - Rompue : l’opération s’est finie mais a échouée
  - Acquittée : l’opération est terminée
- Elles offrent une alternative plus élégante aux callbacks
- Il est possible d’enchainer les promesses avec la fonction **then()**

Un exemple de promesse

```
onClick(function(event) {
  getSomething(function(data) {
    var newData = process(data);
    uploadData(newData,function() {
      // are we done yet ?
    })
  });
});
```

devient

```
onClick()
  .then(getSomething)
  .then((data) => processData(data))
  .then((data) => upload(data))
  .then(() => {
     // are we done yet ?
  });
```

La méthode then

- Cette méthode permet d'enchainer des promesses
- Elle prend en paramètre une promesse ou une fonction
- Elle retourne une nouvelle Promesse
- Si on passe une fonction, celle-ci est encapsulée dans une promesse automatiquement
- La valeur retournée par une promesse est passée à la promesse suivante
- La méthode peut prendre un deuxième callback qui sera appelé en cas de rejet de la promesse précédente

```
then(onFulfilled, onRejected)
```

Exemples

```
myPromise.then(anotherPromise);

myPromise.then(anotherPromise,errorCb);

myPromise.then(() => {/*will become a promise*/});

myPromise
   .then(() => 3)
   .then( val => console.log(val)); // "3"
```

Créer une promesse

- Il est possible de créer une promesse en instanciant un objet **Promise** et en lui passant un callback correspondant au traitement asynchrone
- Le callback possède deux paramètres : **resolve** et **reject**
- **resolve** et **reject** sont deux fonctions que vous appelez pour changer l'état de la promesse
- Le contenu d'une promesse est toujours exécuté de manière asynchrone

Exemple

```
let myPromise = new Promise((resolve, reject) => {
    if(Math.random() > 0.5)
      resolve();
    else
      reject();
});

myPromise.then(
   () => { /* Yay */},
   () => { /* Bleh */}
);
```

Exemple 2

```
let myPromise = new Promise((resolve, reject) => {
    if(Math.random() > 0.5)
      resolve("Yay");
    else
      reject("Bleh");
});

myPromise.then(
  (val) => { console.log(val) }, // "Yay"
  (val) => { console.log(val) }  // "Bleh
);
```

Transformer un appel en Promise

```
let processDataPromise = new Promise(
   (resolve,reject) => {
     processData(
        (err) => reject(err),
        (data) => { resolve(data) }
     );
});

let processDataPromise = new Promise(
   (resolve,reject) => {
     processData((err, data) => {
        if(err) reject(err);
        else resolve(data);
     });
});
```

Gérer les erreurs

- Si une erreur arrive dans n'importe quelle promesse de la chaine, elle est propagée tant que personne ne l'intercepte
- On intercepte les erreurs avec la méthode **catch()**
- Les erreurs sont toutes exceptions levées par une promesse
- Si la fonction passée à **catch()** retourne une valeur, celle-ci est transmise à la promesse suivante comme si la promesse courant avait été tenue

Exemples

```
myPromise().catch(() => {/* arg */} );

myPromise()
   .catch(() => {/* arg */} )
   .then(anotherPromise)
   .catch(() => {/* arg */} );

myPromise()
   .then(anotherPromise)
   .then(andAother)
   .then(andAThirdOne)
   .catch(() => {/* errors from all 4 promises */});


myPromise
   .catch(() => 3)
   .then((val) => { console.log(val); }); // "3"
```

Executer plusieurs promesses 1/2

```
Promise.all([promise1,promise2])
   .then(([val1,val2]) => {});
```

- Prend en paramètre un tableau de promesses
- La promesse suivante obtiendra un tableau avec les paramètres dans l'ordre
- Les promesses sont exécutées en parallèle
- La promesse est résolue quand toutes les promesses sont résolues
- La promesse est rejetée quand une des promesse est rejetée

Executer plusieurs promesses 2/2

```
Promise.race([promise1, promise2])
   .then((val) => {});
```

- Prend en paramètre un tableau de promesses
- La promesse suivante obtiendra la valeur de la première promesse résolue
- Les promesses sont exécutées en parallèle
- La promesse est résolue quand la promesse la plus rapide est résolue
- La promesse est rejetée quand une des promesse est rejetée

Antipattern 1

- Perdre la chaîne d'appel :

```
function foo() {
  const promise = asyncFunc();
  promise.then(result => {
    ...
  });

  return promise;
}
```

Antipattern 1

- Perdre la chaîne d'appel :

```
function foo() {
  const promise = asyncFunc();
  promise.then(result => {
    ...
  });

  return promise;
}
```

```
function foo() {
  const promise = asyncFunc();
  return promise.then(result => {
    ...
  });
}
```

Antipattern 2

- Imbriquer des promesses :

```
asyncFunc1()
  .then(result1 => {
    asyncFunc2()
      .then(result2 => {
        ...
      });
  });
```

Antipattern 2

- Imbriquer des promesses :

```
asyncFunc1()
  .then(result1 => {
    asyncFunc2()
      .then(result2 => {
        ...
      });
  });

asyncFunc1()
  .then(result1 => asyncFunc2())
  .then(result2 => { ··· });
```

Antipattern 3

- Créer une promesse au lieux d'en retourner une :

```
insertInto = (db) => {
  return new Promise((resolve, reject) => {
    db.insert(fields)
      .then(resultCode => {
        resolve(resultCode);
      })
      .catch(err => { reject(err); })
  });
};
```

Antipattern 3

- Créer une promesse au lieux d'en retourner une :

```
insertInto = (db) => {
  return new Promise((resolve, reject) => {
    db.insert(fields)
      .then(resultCode => {
        resolve(resultCode);
      })
      .catch(err => { reject(err); })
  });
};
```

```
insertInto = (db) => {
  return db.insert(this.fields)
    .then(resultCode => {
      return resultCode;
    });
};
```

Allez plus loin

- L'API de base est simple mais un peu limitée
- Il existe des librairies tierces bien plus puissantes :
  - Bluebird
  - Q

- Attention : jQuery possède une syntaxe proche mais ce ne sont pas des promesses « Promise/A+ »

https://abdulapopoola.com/2014/12/12/the-differences-between-jquery-deferreds-and-the-promisesa-spec/