



# Async approach

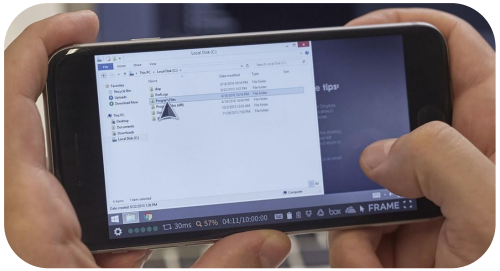
## Callbacks vs Promises

Milos Zikic, Front end Team Lead
zile@fra.me
www.fra.me

FRAME

# Frame - Run any software in a browser

Frame is a revolutionary cloud platform that lets enterprises deliver Windows apps to users on any device, anywhere.



| Founded | Locations | Employees | Investments | Customers |
|---|---|---|---|---|
| 2012<br>Silicon Valley<br>California | Nis<br>Belgrade<br><br>San Mateo<br>Washington<br>Redding | > 70<br>Currently hiring<br>jobs@fra.me | 2017, **$16M** Series A<br>2015, **$10M** Series A<br>2014, **$2.2M** Seed | Microsoft<br>Adobe<br>SIEMENS |

FRAME

# What is synchronous flow?

```
1:  var a = 5;
2:  var b = a + 3;
3:  var c = b + 4;
4:  console.log(c); // 12
```

**Definition:** Each step is executed after previous one was completed. If one of the steps fails -- throws an error, a rest of the steps are skipped.

**Example:** Line #4 will be executed only after completion of lines #1, #2, #3.

FRAME

# What is asynchronous flow?

```
1:  uploadMyPhotoToTW();
2:  console.log('Done');
3:  // Done or not?
```

**Definition:** A step can be executed even if the previous one was not completed (just started).

**Example:** Line #2 will be executed after the upload has been started, but before its completion.

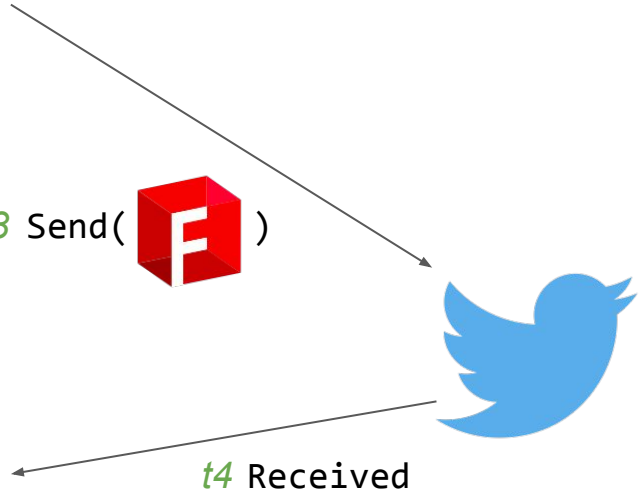
FRAME

# Why?

timeline

*t1* `uploadMyPhotoToTW();`

*t2* `console.log('Done');`



# Callbacks

# Solution 1 - Callbacks

A callback is a function which is passed as a parameter to an async function. The callback will be invoked once the async execution is completed. A good example for async flow is *setTimeout* function.

```
1: setTimeout(function() {
2:  console.log('A');
3: }, 1000); // 1s
4: console.log('B');
5: // AB or BA?
```

**Example:** The anonymous function which prints 'A' is a callback function, whereas *setTimeout* is an async function. The callback will be invoked after the specified timeout -- 1 second in this case.

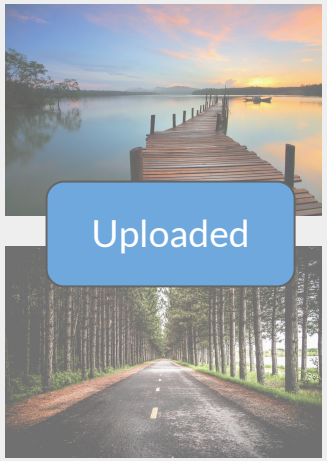
Due to async flow, line #4 will be executed before #2.

FRAME

# Multiple dependent async flows

The challenge is how to handle multiple flows with the callback mechanism. You want to execute some piece of code, once several async operations are completed.

**Example:** Imagine that you want to upload 2 photos of a user, and once that is done you show the message ‘Uploaded’.



# Multiple dependent async flows - solution

Drawbacks?

```
var firstImg = new Image(...);
var secondImg = new Image(...);

var firstUploaded = false;
var secondUploaded = false;
```

Initialization

```
uploadImage(firstImg, function() {
 if (secondUploaded) {
  showMessage();
 } else {
  firstUploaded = true;
 }
});
```

Flow A

```
uploadImage(secondImg, function() {
 if (firstUploaded) {
  showMessage();
 } else {
  secondUploaded = true;
 }
});
```

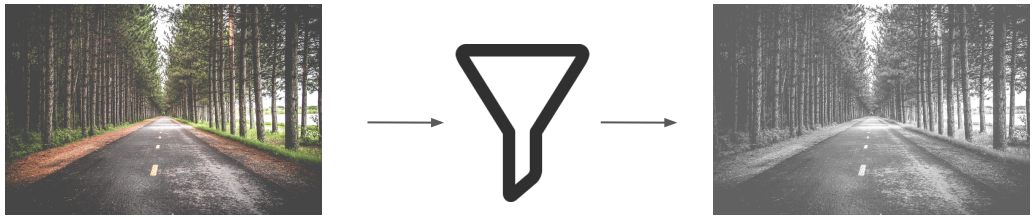
Flow B

FRAME

# Sequential async operations

The challenge is how to do several sequential operations in row. The principle is simple, once operation #n is completed, start operation #n+1. We stop when the last operation is completed.

**Example:** Simple image editing web app. Downloads an image, applies the filter and uploads back.



# Sequential async operations - solution

Nested callbacks -- a state known as ‘callback hell’.

```
downloadPhoto('photos.com/photo1.jpg', function(original) {

    applyFilter(original, function(blackAndWhite) {

        uploadPhoto(blackAndWhite, function() {

            console.log('Uploaded')

        });

    });

});
```

**Drawbacks:**

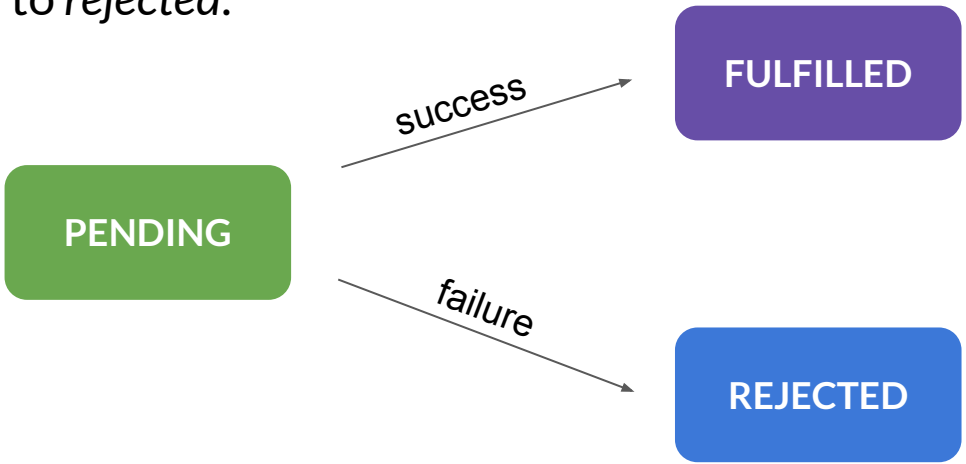
- not readable
- hard to maintain
- code tied together

# Promises

# Solution 2 - Promises

*Promise* is a class which encapsulates an asynchronous operation. It is a placeholder in which a successful result or a reason for failure will be eventually stored.

*Promise* instance is a simple state machine, which can change its state only once. The initial state is *pending*. If async operation is completed, it transitions to *fulfilled*, otherwise it goes to *rejected*.



FRAME

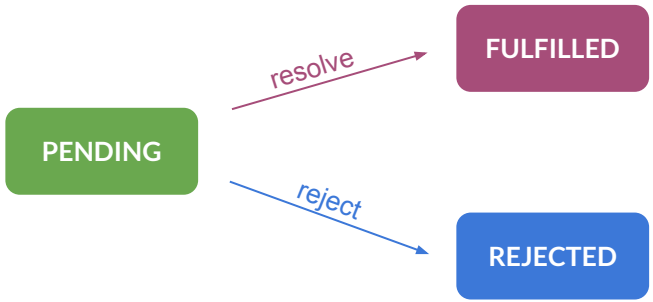
# Promise - Constructor

- It accepts a executor function, which is the code we want to execute
- Once our code  is completed, we invoke *resolve*
- If something fails, we invoke *reject*

```
var p = new Promise(function(resolve, reject) {

     setTimeout(resolve, 1000); // 1 second

});
```



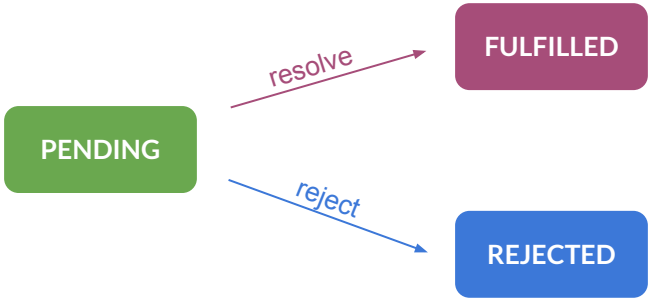
# Promise - Handlers

- Once promise changes its state it invokes registered handlers (listeners)
- Using *then* function, you can register a handler for success
- Using *catch* function, you can register a handler for failure

```
var p = new Promise(...);

p.then(function(result) {
  console.log("Success");
});

p.catch(function(error) {
  console.error("Error:" + error);
});
```



FRAME

# Promises - deep dive

## Multiple handlers of a promise

```
var uploadPromise = new Promise(...);

p.then(function(result) {
  console.log("Here I hide spinner");
});

p.then(function(result) {
  console.log("Here I show a message");
});
```

## Supported try / catch flow

```
var p = new Promise(...);

p.then(function() {
  // do something
})
.catch(function(error) {
  // handle error
})
.then(function() {
  // continue work
});
```

FRAME

# What *then* returns?

```
var a = new Promise(...);

a
.then(function() {
  console.log('A completed!');
  return new Promise(...);
})
.then(function() {
  console.log('B completed!');
});
```

# Chaining of promises

Promises support sequential chaining, which allows one async operation to be executed after another. Chaining solves the problem of callback hell.

```
var downloadPromise = downloadImg();

var filterPromise = downloadPromise
  .then(function(image) {
    return applyFilter(image);
  });

var uploadPromise = filterPromise
  .then(function(image) {
    return uploadImg(image);
  });

uploadPromise.then(function() {
  console.log('Success');
});
```

```
downloadImg()
  .then(function(image) {
    return applyFilter(image);
  })
  .then(function(image) {
    return uploadImg(image);
  })
  .then(function() {
    console.log('Success');
  })
  .catch(function() {
    console.log('Error');
  });
```

FRAME

# Parallel execution

Promise class offers static methods which can combine several promises into one.

## Promise.all

```
var uploadImg1 = new Promise(...);
var uploadImg2 = new Promise(...);
var uploadImg3 = new Promise(...);

var joinpPromise = Promise.all([
 uploadImg1, uploadImg2, uploadImg3
]);
joinedpPromise.then(function() {
 console.log("Uploaded!");
});
```

## Promise.race

```
var uploadImg1 = new Promise(...);
var uploadImg2 = new Promise(...);
var uploadImg3 = new Promise(...);

var racePromise = Promise.race([
 uploadImg1, uploadImg2, uploadImg3
]);
racePromise.then(function() {
 console.log("Progress 33%");
});
```

FRAME

# Quiz

```
var p = new Promise(function(resolve, reject) {
     setTimeout(resolve, 1000); // 1 second
});
```

```
p.then(function() {
  console.log('A');
  p.then(function() {
    console.log('B');
  });
});
```

?

```
p.then(function() {
  console.log('A');
});
p.then(function() {
  console.log('B');
});
```

?

FRAME

# Where JS is going with ES7

- Make async and sync code the same
- Remove all boilerplate code, which just creates a noise
- Simplify error handling

```
async function processImage(imagePath) {
    try {
        let img = await loadImage(imagePath);
        let filteredImg = await applyFilter(img);
        await uploadImage(filteredImg);
    } catch (e) {
        console.error(e);
    }
}
```



# Questions?

zile@fra.me
www.fra.me

15 min

Next topic: Extreme JS performances

zile@fra.me
www.fra.me