Promises: More Async Programming in Javascript

Prof. Fabio Ciravegna
Dipartimento di Informatica
Università di Torino
fabio.ciravegna@unito.it

UNIVERSITÀ DI TORINO

Promises

- Promises are a way to create asynchronous code that allows easy sequencing of async processes
  - Following a very familiar structure:
    - if x
      - do Y
      - then do W
      - then do Z

Promise Declaration

- Step 1: declare the promise
  - the promise declares some code that is to be run asynchronously
    - for example uploading an image may take a lot of time so it is executed asynchronously
  - it also contains the strategy to decide if the promise was successfully executed or not
    - for example: it is successful if the image file was found; it is unsuccessful otherwise
  - it has two parameters:
    - function to call if code is executed successful
    - function to call if code returns an error

UNIVERSITÀ DI TORINO

Consuming the promise

- Here is when we run the promise so that the async code is executed
- As parameter to the call, we pass the functions to call in case of success/error
  - the success function is passed in a branch called .then, the failure function is passed in the branch called .catch
  - by passing the success/rejection functions as parameters we can use the same async code in different contexts and obtain a different behaviour
    - for example in one context the success in downloading an image will display it to the user, in another case it may compute some metadata for the image (e.g. size)
    - so when the resolve function is called, we execute the function passed in the .then branch (or in the .catch branch otherwise)

In summary

- So the promise
  - at declaration
    - it declares a long running computation
    - it declares placeholders for the behaviour to adopt in case of success and in case of error (resolve and reject represent functions passed as parameters)
  - at consumption time
    - it declares the functions actual functions to be used for success/error
  - at execution time
    - it runs the promise code and calls the actual success/reject function

Declaration and Consumption

Declaration

```
let myPromise=  new Promise(function(resolve, reject) {
  // long running operation
  Math.asyncRandom(20, function(random)){
    // condition of success evaluated when the long running
    // operation is finished
      if (random%2===0)
            // success, so we are calling resolve
         resolve('it is even!!!');
    //failure: calling reject
    else
        reject('whoops odd number');
    }
  }
```

Consumption

```
function consumeMyPromise(){
  // calling this function will run the promise
  //(async code)
      myPromise()
          // the function to execute in
          // case of success (resolve parameter)
          .then(function(okMessage) {
                console.log(okMessage);
            })
            // the function to execute in case of
            // error (reject parameter)
      .catch(function(failureMessage)
              alert(failureMessage);
          })
}
```

This is just an exemplification.
Obviously the random function is not asynchronous

The Declaration Better Annotated

```
let myPromise=  new Promise(function(resolve, reject) {

  let random= Math.random(20);

  if (random%2===0)

          resolve(‘it is even!!!’);

  else
          reject(‘whoops odd number’);
```

Declaration

Code that takes a long time

Condition of success/error

call to the success function

call to the success function

Example

or - if you prefer using the fat arrow notation…

```
myPromise()
.then(function(okMessage) {
     console.log(okMessage);
     })
 .catch(function(failureMessage)
    alert(failureMessage);
    })
```

```
myPromise()
.then(okMessage =>
     console.log(okMessage);
     )
 .catch(failureMessage =>
    alert(failureMessage);
    )
```

- Note: if you want to use the fat arrow notation (ES6) in IntelliJ you must
- enable ES6
  - https://hackernoon.com/quickstart-guide-to-using-es6-with-babel-node-and-intellij-a83670afbc49
  - read until the math example, the rest is irrelevant

Fat Arrow Notation

- The "fat arrow" notation in JavaScript refers to the use of the arrow function syntax
  - also known as "arrow functions" or "lambda functions")
- Arrow functions are a more concise way to write anonymous functions in JavaScript,
  - making your code shorter and often easier to read
- Basic syntax:

```
(parameter1, parameter2, ..., parameterN) => expression
```

  - parameter1, parameter2, etc.: These are the function parameters, similar to parameters in regular function declarations.
  - =>: The fat arrow (=>) separates the parameters from the function's body.
  - expression: This is the code that gets executed when the arrow function is called. It is also the value that is implicitly returned from the function when there is no explicit return statement

Examples

```
// with no parameter
const greet = () => {
  console.log("Hello, world!");
};
//with one param
const square = (x) => {
  return x * x;
};

// with two params
const add = (a, b) => {
  return a + b;
};
```

Concatenating Promises

- Promises gets you out of the callback hell
- Instead of nesting callbacks, you just concatenate .then branches

```
function orderFood(){
  chooseToppings(function(err, toppings){
    if (err) {
        alert (err);
              return;
              }
      placeOrder(toppings, function(err, order){
           if (err) {
             alert (err);
                return;
           }
                collectOrder(order,function(err, pizza) {
                 if (err) {
                   alert (err);
                        return;
                 }
                         eatPizza(pizza, function()
                return;);})})})})
```

Vs

```
chooseToppings()
.then(toppings =>
  placeOrder(toppings))
.then(order =>
  collectOrder(order))
.then(pizza =>
  eatPizza(pizza))
.catch(err =>
           alert (err);
);
```

UNIVERSITÀ DI TORINO

Passing values from the environment

- you can pass values to a promise in this way
  - you include the promise in a function with parameters
  - use the name of the parameters within the promise

```
var loginF = function(username, password) {
  return new Promise(function(resolve, reject) {
    if (correct(username, password) {
      resolve("login accepted");
    } else {
      reject(Error(“Login failed"));
    }
  });
}
```

https://stackoverflow.com/questions/35318442/how-to-pass-parameter-to-a-promise-function

Consuming Promises with parameters

- So when you consume the promise:

```
loginF(username, password)
  .then(function(uid) {
      // show login page
    }
    .catch (error){
     // show failure message
    })
```

What you should know

- You should be confident in writing async programmes in Javascript using promises
  - you should avoid having nested callbacks when possible
- You should be able to:
  - Declare a promise
  - Consume a promise
  - Pass parameters to a promise

Questions?

© Prof. Fabio Cravegna, Università di Torino