Asynchronous Javascript

When did that happen?

IT Akademija

Javascript

Synchronous, blocking, asynchronous and non-blocking.

When running usual synchronous methods - blocking:

```
console.log("One");
console.log("Two");
console.log("Three");
//One
//Two
//Three
```

When running async operations - non-blocking:

```
console.log("One");
console.log(fetchSomethingFromBackend()) //returns 1.5
console.log("Two");
console.log("Three");
//One
//Two
//Three
//1.5
```

Callbacks to the rescue

Callback - a function passed to another function, that is executed after some work is done.

```
function fetchSomethingFromBackend(callback) {
  const req = doTheActualFetch(); //returns 1.5
  req.on("complete", callback);
}
function thisIsCallback() {
  console.log("Two");
  console.log("Three");
}
console.log("One");
console.log(fetchSomethingFromBackend(thisIsCallback));
//One
//1.5
//Two
//Three
```

Don’t call me

Christmas tree from hell.

Passing functions to other function is hard to manage.

```
firstFunction(args, function() {
  secondFunction(args, function() {
    thirdFunction(args, function() {
      // And so on…
    });
  });
});
```

Async #2: Promises

Async #2: Promises

- Value containers for asynchronous actions.
- Can be one of three states: pending (in progress), resolved (success) or rejected (error).
- Created using a simple Promise constructor:

```
const promise = new Promise((resolve, reject) => {
    if (success()) {
        resolve();
    } else {
        reject();
    }
});
```

Chainable promises

Promises expose convenience methods - .then and .catch - which allow us to provide callbacks (huh?) via a tidy, chainable, interface.

.then - callback to be run on resolution.

.catch - callback to be run on rejection

```
function loadPosts() {
  this.httpClient
    .get("https://backend-url")
    .toPromise()
    .then(resp => {
      console.log("This means good", resp);
      return resp;
    })
    .then(resp => {
        //.then() returns a Promise
        //It can be chained.
    })
    .catch(error => {
      console.log("This is bad", error);
    });
}
```

Synchronous Promises
Async/Await

- Special Async/Await syntax lets us handle Asynchronous Promises just like synchronous code.
- Easier to use (just like synchronous code), but harder to handle errors.
- Await can only be used in async funtions.

```
function promise() {
    return new Promise((resolve) ⇒ resolve("result"))
}

async function callPromise() {
    const result = await promise(); //wait for promise resolution
    console.log(result);
}

callPromise();
//Output: 'result'
```


Milk
Skim
Butter Milk
Lite White

Async #3: Observables

Observables

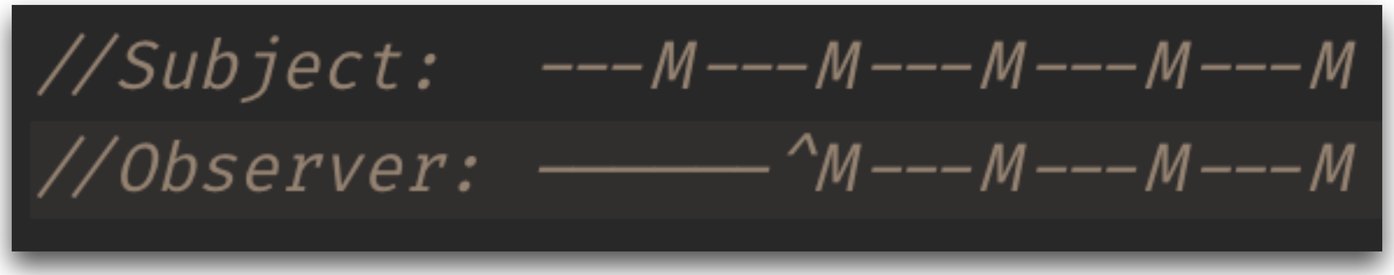
Sometimes, you want to provide a continuous stream of updated data to your components and have the components act on the data as they get it - show it, format it or validate it.

Observables are functions, that return values to their subscribers (more on that later) over time.

They allow developers to use the observer pattern easily when programming Javascript.

Observer pattern in Javascript

Subject (source of messages) passes messages to a created observable. A subscriber (also known as an observer) can start reading the messages once it is subscribed to that Observable. The messages will only be visible once the subscription has happened.



Observable subscriptions

For Observables to provide values - an observer has to be subscribed:

```
import { Observable } from "rxjs";

//Observable
const obs = new Observable((subject) ⇒ {
  //Push messages to the observer
  subject.next(1);
  subject.next(2);
  subject.complete();
});

//Define an observer
function observer(message) {
  console.log(message);
}

//Attach the observer to the Observable
obs.subscribe({
  next: observer,
  error: (err) ⇒ console.error(err),
  complete: () ⇒ console.log("Observable completed"),
});
/** Output:
1
2
Observable completed
*/
```

Subjects

- Multicasted observables - observables, that can have multiple observers at once.
- Different types of subjects are available:
  - Subject - barebones subject. Observe values when subscribed.
  - ReplaySubject - Subject that emits the previous values when subscribed.
  - BehaviorSubject - ReplaySubject that also gets an initial value.

```
import {Subject} from 'rxjs';

const subject = new Subject();

function observerOne(message) {
    console.log(`Observer1: ${message}`);
};

function observerTwo(message) {
    console.log(`Observer2: ${message}`);
};

subject.subscribe(observerOne);

subject.next("This");
subject.next("Is");
console.log("Not");

subject.subscribe(observerTwo);

subject.next("Sparta");
/** Output:
Observer1: This
Observer1: Is
Not
Observer1: Sparta
Observer2: Sparta
 */
```

Observable operators

- Simple functions, that provide utility when working with observables.
- Two types:
  - Creation: creates observables from provided sources or combines several observables to make one.
  - Pipeable: Used in the ‘.pipe’ function to modify or transform observables and return new observables.
- Full list of operators can be found at https://www.learnrxjs.io/

Operator examples

- Creation -
  - of([1,2,3]) - creates observable with the array as a value.
  - from([1,2,3]) - creates observable with each element in the array as a value.
  - fromEvent(document, ‘click’) - creates observable from document click events.
- Pipeable -
  - filter - blocks specific observable values from being passed to subscriber.
  - map - modifies an observable value as it is being passed.

Fin.

Observe caution.