



Promises

Miti Bhat



Basics



- At their most basic, promises are a bit like event listeners except:
- A promise can only succeed or fail once. It cannot succeed or fail twice, neither can it switch from success to failure or vice versa.
- If a promise has succeeded or failed and you later add a success/failure callback, the correct callback will be called, even though the event took place earlier.
- This is extremely useful for async success/failure, because you're less interested in the exact time something became available, and more interested in reacting to the outcome.



Basics



- A promise can be:
- **fulfilled** - The action relating to the promise succeeded
- **rejected** - The action relating to the promise failed
- **pending** - Hasn't fulfilled or rejected yet
- **settled** - Has fulfilled or rejected



Syntax

```
var promise = new Promise(function(resolve, reject) {  
    // do a thing, possibly async, then...  
  
    if (/* everything turned out fine */) {  
        resolve("Stuff worked!");  
    }  
    else {  
        reject(Error("It broke"));  
    }  
});
```



So a Promise...

- The promise constructor takes one argument:
- a callback with two parameters, resolve and reject.
- Do something within the callback, perhaps async, then call resolve if everything worked, otherwise call reject.



Now... Use it

```
promise.then(function(result) {  
  console.log(result); // "Stuff worked!"  
}, function(err) {  
  console.log(err); // Error: "It broke"  
});
```



DOM Usage

Although they're a JavaScript feature, the DOM isn't afraid to use them. In fact, all new DOM APIs with async success/failure methods will use promises. This is happening already with

- [Quota Management](#),
- [Font Load Events](#), [ServiceWorker](#),
- [Web MIDI](#),
- [Streams](#), and more.

Jquery Deferred & JS Promises

jQuery's Deferreds are a bit ... unhelpful. They can be cast to standard promises, which is worth doing as soon as possible:

```
var jsPromise = Promise.resolve($.ajax('/whatever.json'))
```

Here, jQuery's \$.ajax returns a Deferred.

Since it has a then() method, Promise.resolve() can turn it into a JavaScript promise.

However, sometimes deferreds pass multiple arguments to their callbacks, for example:

```
var jqDeferred = $.ajax('/whatever.json');
```

```
jqDeferred.then(function(response, textStatus, xhrObj) {  
  // ...  
}, function(xhrObj, textStatus, err) {  
  // ...  
})
```

Whereas JS promises ignore all but the first:

```
jsPromise.then(function(response) {  
  // ...  
}, function(xhrObj) {  
  // ...  
})
```





Demo

- ▶ Demo(Use Chrome Web Server from Extensions)



Promise Chaining

- `then()` isn't the end of the story, you can chain them together to transform values or run additional async actions one after another.
- 



Demo



Demo





Angular Observable

- The way to communicate between components is to use an Observable and a Subject (which is a type of observable)
- there are two methods that we're interested in: `Observable.subscribe()` and `Subject.next()`.
- `Observable.subscribe()`
The observable subscribe method is used by angular components to subscribe to messages that are sent to an observable.
- `Subject.next()`
The subject next method is used to send messages to an observable which are then sent to all angular components that are subscribers (a.k.a. observers) of that observable.



Angular Http

- Angular's HTTP method returns an Observable instead of returning a Promise. This Observable then needs to be subscribed to for it to be consumed. Using the pipeable operator on the subscription, we are able to transform the returned subscription, and now we need to use an async pipe to consume the subscription on the template side.
- To make Angular and RxJS better together, Angular has created an async pipe
- Async will automatically subscribe to the Observable for you, and it will automatically unsubscribe for you as well when you navigate out of the page or component.



Angular Promise & Observable

- With a Promise you can only handle one event.
- With an Observable you can handle multiple events.
- `.subscribe()` is similar to `.then()`.
- An Observable can do everything that a Promise can do, plus more.
- Use Angular's `HttpClient` to handle API calls.
- The Angular framework uses a lot of RxJS.
- Many Angular APIs use Observables – we can take advantage of this to build some really cool stuff.



Angular Observable

- Observables are not executed until a consumer subscribes.
- The subscribe() executes the defined behavior once, and it can be called again. Each subscription has its own computation. Resubscription causes recomputation of values.

// declare a publishing operation

```
new Observable((observer) => { subscriber_fn });
```

// initiate execution

```
observable.subscribe(() => {  
    // observer handles notifications  
});
```




Angular Promise

- Promises execute immediately, and just once. The computation of the result is initiated when the promise is created. There is no way to restart work. All then clauses (subscriptions) share the same computation.

// initiate execution

```
new Promise((resolve, reject) => { executer_fn });
```

// handle return value

```
promise.then((value) => {
```

```
    // handle result here
```

```
});
```