PROMISES

Vijesh M. Nair
Assistant Professor
Dept. of CSE (AI-ML)

Why do we need Promises?

- Why is this important?
  - Think about a website that loads data from an API then processes and formats the data to display to the user. If we try to process and format our data before the API has even fetched the information, we're either going to get an error or a blank website.
  - By using a Promise, we can ensure that the API data isn't processed/formatted until after our API call has succeeded.

What is a Promise?

- A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it's not resolved (e.g., a network error occurred).

- In JavaScript, a Promise represents the eventual *result* of an asynchronous operation.

What is a Promise?

- Lets think about promise as the following scenario
  - "Imagine you are a **kid**. Your mom **promises** you that she'll get you a **new phone** next week."
  - You *don't know* if you will get that phone until next week. Your mom can either *really buy* you a brand new phone, or *stand you up* and withhold the phone if she is not happy :(.
  - That is a **promise**. A promise has 3 states. They are:
    - Promise is **pending**: You don't know if you will get that phone until next week.
    - Promise is **resolved**: Your mom really buy you a brand new phone.
    - Promise is **rejected**: You don't get a new phone because your mom is not happy.

Promise's States

- A promise may be in one of 3 possible states
  - **Fulfilled**—Operation has completed and the Promise has a value
    - onFulfilled() will be called (e.g., resolve() was called)
  - **Rejected**—Operation has completed with an error or failed.
    - onRejected() will be called (e.g., reject() was called)
  - **Pending**—Asynchronous operation has not completed yet
    - not yet fulfilled or rejected

Settled or Pending?

- A promise is **settled** if it is not pending (it has been resolved or rejected).). Once a Promise has settled, it is settled for good. It cannot transition to any other state.
  - Sometimes people use *resolved* and *settled* to mean the same thing: *not pending*.

- Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.

Working With Promises

Most of the time when working with Promises, you will be consuming already-created promises that have been returned from functions. However, you can also create a promise with it's constructor.

Here's what a simple Promise might look like:

```
runFunction().then(successFunc, failureFunc);
```

In the above example, we first invoke the runFunction() function.

Since runFunction() returns a promise, only when the Promise is settled, successFunc, or failureFunc function runs.

If the Promise is Fulfilled, our sucessFunc is invoked. If the Promise fails, our failureFunc is invoked.

Methods & Constructor in Promise

Method Name	Summary
Promise.resolve(promise)	This method returns promise only if promise.constructor==Promise.
Promise.resolve(thenable)	Makes a new promise from thenable containing then().
Promise.resolve(obj)	Makes a promise resolved for an object.
Promise.reject(obj)	Makes a promise rejection for the object.
Promise.all(array)	Makes a promise resolved when each item in an array is fulfilled or rejects when items in the array are not fulfilled.
Promise.race(array)	If any item in the array is fulfilled as soon, it resolves the promise, or if any item is rejected as soon, it rejects the promise.

Constructor in Promise

new Promise(function(resolve, reject) {});	Here, resolve(thenable) denotes that the promise will be resolved with then(). Resolve(obj) denotes promise will be fulfilled with the object Reject(obj) denotes promise rejected with the object.

Promise Implementation

- In the Promise implementation, the Promise constructor takes an argument that callbacks the function. This callback function takes two arguments, i.e.,
  - **Resolve:** When the promise is executed successfully, the resolve argument is invoked, which provides the result.
  - **Reject:** When the promise is rejected, the reject argument is invoked, which results in an error.
- It means either resolve is called or reject is called.
- Here, then() has taken one argument which will execute, if the promise is resolved. Otherwise, catch() will be called with the rejection of the promise.

```
<html>
<head>
<h2> Javascript Promise</h2>
</br> </head>
<body>
<script>
var p=new Promise(function(resolve, reject){
var x= 2+3;
if(x==5)
   resolve(" executed and resolved successfully");
else
   reject("rejected");
});
 p.then(function(fromResolve){
 document.write("Promise is"+fromResolve);
 }).catch(function(fromReject){
 document.write("Promise is "+fromReject);
 });
</script>
</body>
</html>
```

Promise Implementation

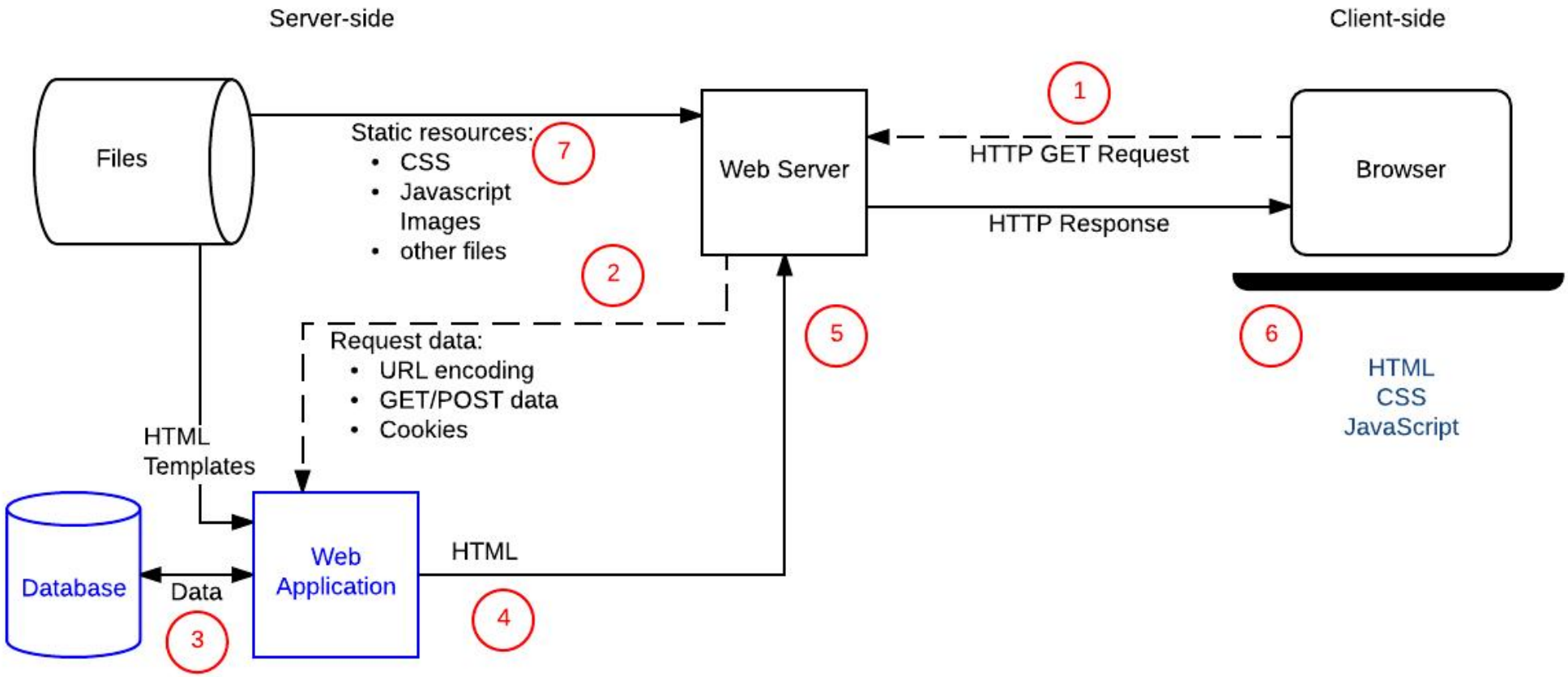
```
<html>
<head>
<h2> Javascript Promise</h2>
</br> </head>
<body>
<script>
var p=new Promise(function(resolve, reject){
var x= 2+3;
if(x==5)
    resolve(" executed and resolved successfully");
else
    reject("rejected");
});
 p.then(function(fromResolve){
 document.write("Promise is"+fromResolve);
 }).catch(function(fromReject){
 document.write("Promise is "+fromReject);
 });
</script>
</body>
</html>
```

Output:

Promise is executed and resolved successfully

CLIENT-SERVER COMMUNICATION

Client-Server Model



Communication channel between a client and a server using JavaScript

- The communication channel between a client and a server is full-duplex: either side can send a message to the other at any time.
- This can be implemented over various protocols, such as HTTP, WebSocket, or WebRTC.
- HTTP stands for Hypertext Transfer Protocol, and it is the basis of the web.
- HTTP defines how messages are formatted and transmitted between clients and servers.
- HTTP messages consist of two parts: headers and body.
- Headers contain metadata about the message, such as the method, the URL, the content type, etc.
- Body contains the actual data of the message, such as HTML, JSON, XML, etc.

HTTP REQest & RESponse Methods/Verbs

- A URL identifying the target server and resource (e.g. an HTML file, a particular data point on the server, or a tool to run).
- A method that defines the required action (for example, to get a file or to save or update some data). The different methods/verbs and their associated actions are listed below:
  - `GET`: Get a specific resource (e.g. an HTML file containing information about a product, or a list of products).
  - `POST`: Create a new resource (e.g. add a new article to a wiki, add a new contact to a database).
  - `HEAD`: Get the metadata information about a specific resource without getting the body like `GET` would. You might for example use a `HEAD` request to find out the last time a resource was updated, and then only use the (more "expensive") `GET` request to download the resource if it has changed.
  - `PUT`: Update an existing resource (or create a new one if it doesn't exist).
  - `DELETE`: Delete the specified resource.
  - `TRACE`, `OPTIONS`, `CONNECT`, `PATCH`: These verbs are for less common/advanced tasks, so we won't cover them here.

Send HTTP REQ from Client to a Server

- To send an HTTP request from a client to a server using JavaScript, we can use the built-in `XMLHttpRequest` object or the newer **fetch** API.
- Both methods allow us to specify the URL, the method, the headers, and the body of the request.

For example:

```
// Using XMLHttpRequest
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://example.com/api/users");
xhr.setRequestHeader("Content-Type", "application/json");
xhr.send();

// Using fetch
fetch("https://example.com/api/users", {
  method: "GET",
  headers: {
    "Content-Type": "application/json",
  },
});
```

Receive HTTP RES from Server to a Client

- To receive an HTTP response from a server to a client using JavaScript, we can use the `onload event handler` of the `XMLHttpRequest object` or the then method of the **fetch** API.

- Both methods allow us to access the status code, the headers, and the body of the response. For example:

```
// Using XMLHttpRequest
xhr.onload = function () {
  if (xhr.status == 200) {
    // Success
    var data = JSON.parse(xhr.responseText);
    console.log(data);
  } else {
    // Error
    console.error(xhr.statusText);
  }
};

// Using fetch
fetch("https://example.com/api/users")
  .then(function (response) {
    if (response.ok) {
      // Success
      return response.json();
    } else {
      // Error
      throw new Error(response.statusText);
    }
  })
  .then(function (data) {
    console.log(data);
  })
  .catch(function (error) {
    console.error(error);
  });
```

Best practices to handle requests and responses

- Use HTTPS instead of HTTP to ensure secure and encrypted communication.
- Use RESTful principles to design your API endpoints, such as using meaningful URLs, standard HTTP methods, and consistent data formats.
- Use JSON as the default data format for both requests and responses, as it is easy to parse and manipulate in JavaScript.
- Use appropriate status codes to indicate the outcome of each request, such as 200 for success, 404 for not found, 500 for internal server error, etc.
- Use error handling techniques to catch and handle any errors that may occur during the communication process, such as using `try-catch` blocks or `catch` methods.
- Use asynchronous programming techniques to avoid blocking the main thread and improve performance, such as using callbacks, promises, or async-await.

FETCH()

Syntax

- The **fetch()** method in JavaScript is used to request data from a server.
- The request can be of any type of API that returns the data in JSON or XML.
- The **fetch()** method requires one parameter, the URL to request, and returns a promise.

Syntax:

```
fetch('url')           //api for the get request
  .then(response => response.json())
  .then(data => console.log(data));
```

Parameters: This method requires one parameter and accepts two parameters:

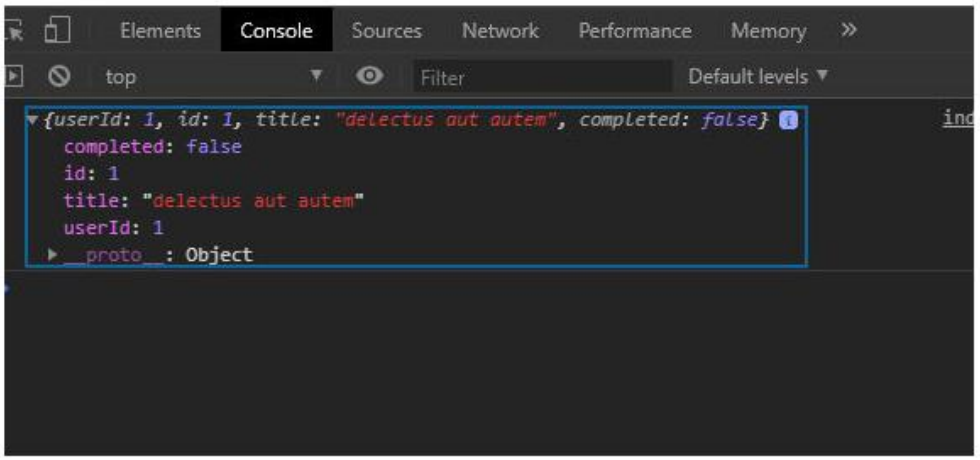
- **URL:** It is the URL to which the request is to be made.
- **Options:** It is an array of properties. It is an **optional** parameter.

Return Value: It returns a promise whether it is resolved or not. The return data can be of the format JSON or XML. It can be an array of objects or simply a single object.

fetch() as GET Req

- Without options, Fetch will always act as a get request.

```
// API for get requests
let fetchRes = fetch(
"https://jsonplaceholder.typicode.com/todos/1");
// fetchRes is the promise to resolve
// it by using.then() method
fetchRes.then(res =>
    res.json()).then(d => {
        console.log(d)
    })
```



fetch() as POST Req

Making Post Request using Fetch: Post requests can be made using fetch by giving options as given below:

```
let options = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(data)
}
```

Thank You!