**Summary of Key Points for Students**

1. **RESTful Endpoints & Resources**: Understand that APIs like reqres.in have specific URLs for data. You use GET, POST, PUT, etc., to interact with those resources (e.g., “users”).
2. **Promises & jQuery Deferred**: jQuery AJAX returns a “promise-like” object, letting you handle success and error once data arrives from the server.
3. **JSON Handling**: APIs typically return JSON. With jQuery, it’s parsed automatically, so you can directly access properties (like response.data).
4. **Template Literals**: Use **backticks** and ${ } syntax to embed variables or expressions in strings. This makes dynamic content generation (like table rows) easier.

**1. Endpoints and Resources in a RESTful API**

* **Endpoint**: A specific URL that your front-end can call to **access or manipulate data** on a server. For example, https://reqres.in/api/users is an endpoint that returns a list of users when you send a GET request.
* **Resource**: In RESTful terms, a “resource” generally refers to **the data** (like “users,” “posts,” “comments”) that you want to create, read, update, or delete. With reqres.in, the resource is “users.”
* **Example**:
  + GET https://reqres.in/api/users?page=2
    - Endpoint is /api/users with a query parameter page=2.
    - Resource: “users” collection, specifically on page 2.

**Key takeaway**: When you see URLs like https://reqres.in/api/users, you’re looking at a RESTful endpoint. You make **HTTP requests** (GET, POST, PUT, DELETE) to these endpoints to interact with the resource (in this case, user data).

**2. Promises & How They Relate to jQuery AJAX**

**2.1 What Is a Promise?**

* A **Promise** in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.
* Typically, when you use the **Fetch API** or **modern libraries**, you get a “native promise,” allowing you to do .then() and .catch().

**2.2 jQuery $.ajax() and Deferred Objects**

* jQuery’s $.ajax() **predates** native ES6 promises, but it **returns a “deferred” object** which behaves very similarly to a promise.
* You can still do:

js

CopyEdit

$.ajax({ ... })

.done(function(response) { /\* handle success \*/ })

.fail(function(error) { /\* handle error \*/ });

or

js

CopyEdit

$.ajax({ ... })

.then(function(response) { /\* handle success \*/ })

.catch(function(error) { /\* handle error \*/ });

* The success/error callbacks you see are part of jQuery’s own system, but they serve the same purpose as .then() and .catch() in a promise chain.

**Key takeaway**: jQuery uses a system of “deferred” objects which function similarly to promises. They allow asynchronous AJAX calls to return data once the server responds.

In modern JavaScript, **Promises** represent the eventual result of an asynchronous operation. A promise can be in exactly **one** of the following states:

1. **Pending**
   * The initial state: the asynchronous operation has started but is **not yet** completed.
2. **Fulfilled** (or “resolved”)
   * The operation **completed successfully**.
   * The promise now has a **resolved value** (the result of the async operation).
   * Once fulfilled, the promise is **settled** and cannot change to any other state.
3. **Rejected**
   * The operation **failed** (e.g., there was an error, or data could not be fetched).
   * The promise now has a **reason** for the rejection (often an Error object).
   * Once rejected, the promise is **settled** and cannot change to any other state.

**Settled** is a term often used to mean the promise is **either** fulfilled **or** rejected. After settling, the promise’s state is final and does not switch again.

**Example Flow**

1. When you start an AJAX request, you might create a new promise (or jQuery does it internally). It’s **pending** while waiting for the server response.
2. If the response is successful and valid, the promise is **fulfilled**, and you can handle the data in .then(...).
3. If there’s a network error or the server responds with an error, the promise is **rejected**, and you handle that case in .catch(...) (or a .fail(...) in jQuery’s syntax).

**Key Points**

* **A promise can go from “pending” to either “fulfilled” or “rejected.”**
* Once it’s “fulfilled” or “rejected,” it is **settled** and cannot change states again.
* You use **.then()** (or .done()) to handle the **fulfilled** case, and **.catch()** (or .fail()) to handle the **rejected** case.

**3. Handling and Dismantling JSON-Formatted Data**

* Most modern APIs (like reqres.in) return data in **JSON** (JavaScript Object Notation).
* **JSON** is just a text format that looks like JavaScript objects, but it’s actually a string when transferred over the network.
* In jQuery, when dataType: 'json' is used (or when the server sends Content-Type: application/json), jQuery **automatically parses** that JSON string into a JavaScript object for you.
* You can then access its properties using **dot notation**. For example, if the JSON response is:

json

CopyEdit

{

"page": 2,

"per\_page": 6,

"total": 12,

"total\_pages": 2,

"data": [

{ "id": 7, "email": "user7@example.com", ... },

{ "id": 8, "email": "user8@example.com", ... }

]

}

You can do:

js

CopyEdit

console.log(response.total\_pages); // 2

console.log(response.data); // Array of user objects

console.log(response.data[0].email); // "user7@example.com"

* **“Dismantling” JSON** means you’re taking the returned object/array and extracting the specific fields you need (e.g., user ID, user email, etc.).

**Key takeaway**: Understand that once jQuery has done the parsing, you’re dealing with a **plain JavaScript object** or **array**. Access the desired data with standard object/array notation.

**4. Using Backticks (`) for Template Literals**

* In modern JavaScript (ES6+), instead of string concatenation like "Hello " + name + "!", you can use **template literals** with **backticks**:

js

CopyEdit

let message = `Hello ${name}!`;

* They allow you to **embed variables** and **expressions** directly inside strings by using ${ }.
* **Why use them?** They’re cleaner for multi-line strings and for inserting variables into HTML strings or console logs.

**Example From the Code**

js

CopyEdit

const newRow = $(`

<tr>

<td>${user.id}</td>

<td><img src="${user.avatar}" alt="Avatar" class="img-thumbnail" style="width: 50px;"></td>

<td>${user.first\_name}</td>

<td>${user.last\_name}</td>

<td>${user.email}</td>

</tr>

`);

* Notice how we do **${user.id}** and **${user.first\_name}** inside the HTML string. This is possible because of **template literals**.

**Key takeaway**: Template literals (with backticks) simplify **string building**—especially when constructing HTML dynamically. They’re an essential ES6 feature for modern JavaScript coding.