Week 9 - JQuery

Created by Dakshil Gorasiya

Callback

A **callback** is a function passed as an argument that jQuery calls **after an operation completes**.

In animations/effects:

- The callback runs **after the animation finishes**
- Ensures correct execution order
- Prevents race conditions caused by async animations

```
$("#box").toggle(1000, function () {
  alert("Toggle completed");
});
```

Here, the function runs **only after** the toggle animation ends.

Animation & Effect Functions

Visibility & Toggle Effects

Method	Purpose
`hide(duration, callback)`	Hide element with animation
`show(duration, callback)`	Show element with animation
`toggle(duration, callback)`	Toggle visibility
`fadeTo(duration, opacity, easing?, callback)`	Fade to specific opacity
`fadeToggle(duration, easing?, callback)`	Toggle fade effect

Sliding Effects

Method	Purpose
`slideUp(duration, callback)`	Slide element up (hide)
`slideDown(duration, callback)`	Slide element down (show)
`slideToggle(duration, callback)`	Toggle slide

Custom Animations

Method	Purpose
`animate(properties, duration, easing?, callback)`	Animate CSS properties
`delay(time)`	Delay next animation in queue

- `delay()` pauses the animation queue, not JavaScript execution

Animation Control Functions

`stop()`

```
$("#box").stop();
```

- Stops the **current animation immediately**
- Keeps element in its current visual state
- Does NOT jump to final values

`finish()`

```
$("#box").finish();
```

- Instantly completes all queued animations
- Jumps element to final state
- Executes animation callbacks

Difference from `stop()`:

- `stop()` halts
- `finish()` completes

Event Callback Functions

All event handlers here are callbacks too:

```
$("#btn").on("click", function () { ... });
```

- Executed when the event occurs
- Passed a jQuery event object
- Run asynchronously via event loop

Execution Order

Without callbacks:

```
$("#box").hide(1000);
alert("Done"); // runs immediately
```

With callbacks:

```
$("#box").hide(1000, function () {
  alert("Done"); // runs after animation
});
```

Deferred and Promise object

- **Deferred** → object that *controls* an async task (Producer)
- **Promise (jQuery)** → object that *observes* an async task (Consumer)

Deferred

A `Deferred` represents a task that will **finish later**.

```
const dfd = $.Deferred();
```

Deferred States

- `pending` → initial state
- `resolved` → success
- `rejected` → failure

State can be checked using:

```
dfd.state();
```

Resolving, Rejecting, Notifying

- `resolve(value)` → marks task successful
- `reject(error)` → marks task failed
- `notify(data)` → sends progress updates

Deferred Callbacks

Method	When it runs
`done(fn)`	On resolve
`fail(fn)`	On reject
`progress(fn)`	On notify
`always(fn)`	On resolve or reject

`then()`

```
dfd.then(successFn, errorFn, progressFn);
```

- Combines `done`, `fail`, and `progress`
- Executes based on final outcome
- Returns a **new promise**, allowing chaining

Promise (`promise()`)

Calling `.promise()` creates a **read-only view**.

```
const promise = dfd.promise();
```

Why `promise()`

- Hides `resolve`, `reject`, `notify`
- Prevents external code from changing state
- Enforces separation of concerns

Deferred vs Promise

Deferred	Promise
Can resolve / reject	Read-only
Controls async flow	Observes async flow
Used by task creator	Used by consumer
Mutable state	Immutable state

Progress Notifications

- jQuery Deferred supports **progress updates**
- Native ES6 Promises do **not**

```
dfd.notify("Loading...");
```

Handled via:

```
progress(fn)
```

Manual Resolution

Deferred can be resolved or rejected **anytime**:

```
dfd.reject("Manually rejected");
```

This flexibility is powerful but dangerous if misused.

AJAX

AJAX (Asynchronous JavaScript and XML) allows web pages to communicate with a server **without reloading the page**.

In jQuery, AJAX is built on top of **XMLHttpRequest** and tightly integrated with **Deferred/Promise**.

How it works

- Sends HTTP requests in the background
- Receives data asynchronously
- Updates UI dynamically
- Abstracts browser differences

jQuery AJAX returns a **jqXHR object**, which is:

- An `XMLHttpRequest` wrapper
- A **Deferred/Promise-like** object

`$.ajax()`

`$.ajax()` is the most flexible and low-level method.

```
$.ajax({
  url,
  type | method,
  data,
  contentType,
  success,
  error
});
```

Key options:

- `url` → endpoint
- `type` / `method` → HTTP verb
- `data` → request payload
- `contentType` → format of sent data

HTTP Methods Usage

Method	Purpose
GET	Fetch data
POST	Create new data
PUT	Replace existing data
PATCH	Update partial data
DELETE	Remove data

Shorthand Methods

jQuery provides simplified helpers for common cases.

Method	Description
`$.get()`	GET request
`$.post()`	POST request
`$.getJSON()`	GET JSON data

Callbacks vs Promise Style

Callback-based

```
success: function (data) { }
error: function (err) { }
```

Promise-based

```
$.ajax(url).done(fn).fail(fn).always(fn);
```

jqXHR Object

jqXHR combines:

- Native XHR
- jQuery Deferred

Provides:

- `.done()` → success
- `.fail()` → error

- `.always()` → cleanup
- `.abort()` → cancel request
- `.state()` → request state

Data Handling

- GET requests send data as **query parameters**
- POST/PUT/PATCH usually send data in request body
- Use `JSON.stringify()` for JSON payloads
- Set `contentType: "application/json"`

jQuery can auto-parse JSON responses.

Error Handling

- `error` / `fail` runs for:
  - Network errors
  - HTTP status outside 2xx

AJAX Lifecycle (Behind the Scenes)

1. Request created
2. Sent via XHR
3. Server processes request
4. Response received
5. Callbacks / Deferred resolved or rejected

HTTP Request Methods

These HTTP methods define **how a client communicates with a server**.

They describe the **intent of the request**, not the implementation.

GET

Used to **retrieve data** from the server.

Key characteristics:

- Does **not modify** server data
- Data is sent via **URL query parameters**
- Can be **cached** and **bookmarked**
- Should be **idempotent** (same request → same result)

Typical use:

- Fetch list or details
- Load page or API data

POST

Used to **create new data** on the server.

Key characteristics:

- Data is sent in the **request body**
- Changes server state
- Not idempotent
- Not cached by default

Typical use:

- Create new records
- Submit forms
- Trigger server-side actions

PUT

Used to **replace an existing resource completely**.

Key characteristics:

- Sends the **entire updated object**
- Replaces old data with new data
- Idempotent

Typical use:

- Update all fields of a resource
- Save full object state

PATCH

Used to **partially update a resource**.

Key characteristics:

- Sends **only changed fields**
- More efficient than PUT

Typical use:

- Update one or few fields
- Status or property updates

DELETE

Used to **remove a resource**.

Key characteristics:

- Deletes server data
- Idempotent
- May or may not return data

Typical use:

- Remove records
- Clear server-side resources

PUT vs PATCH

PUT:

- Replaces entire resource
- Missing fields may be reset or removed

PATCH:

- Updates only specified fields
- Unchanged fields remain intact

Working with JSON Data

Serialization & Deserialization

JSON (**JavaScript Object Notation**) is the most common format for **data exchange between client and server**.

It is text-based, lightweight, and language-independent.

What Is Serialization

Serialization means converting **JavaScript data** into a **string format** so it can be:

- Sent over the network
- Stored (localStorage, DB)
- Logged or transmitted safely

Common serialized formats:

- URL-encoded string
- JSON string

What Is Deserialization

Deserialization is the reverse process:

- Convert string data back into **usable JavaScript objects**

JSON in JavaScript

JavaScript provides a built-in `JSON` object.

- `JSON.stringify()` → serialization
- `JSON.parse()` → deserialization

Serializing JavaScript Data to JSON

```
JSON.stringify(obj);
```

- Converts objects/arrays into JSON string
- Functions and `undefined` are ignored
- Dates become ISO strings

Use case:

- Sending data in AJAX requests
- Storing structured data

Deserializing JSON to JavaScript

```
JSON.parse(jsonString);
```

- Converts JSON string into JS object
- Throws error if JSON is invalid

Always validate or wrap in `try...catch` when parsing external data.

Working with Form Data (jQuery)

serialize()

```
$("form").serialize();
```

Produces:

- URL-encoded string
- Format: `key=value&key=value`

Characteristics:

- Suitable for GET/POST form submission
- Skips unchecked checkboxes and radios

serializeArray()

```
$("form").serializeArray();
```

Produces:

- Array of objects
- Each object has `name` and `value`

Example structure:

```
[
  { name: "username", value: "abc" },
  { name: "password", value: "123" }
]
```

Converting Form Data to JSON

```
JSON.stringify($("form").serializeArray());
```

Serialization Formats Compared

- `serialize()` → URL-encoded string
- `serializeArray()` → JS-friendly structure
- `JSON.stringify()` → network/storage-ready JSON

Important Notes

- JSON supports only:
  - string, number, boolean, null, object, array
- No functions, symbols, or circular references
- JSON keys must be **double-quoted**
- Order of keys is not guaranteed