Full Stack Application Development

Web Protocols

Akshaya Ganesan

Module 3: Web Protocols

- HTTP
  - HTTP Request- Response and its structure
  - HTTP Methods;
  - HTTP Headers
  - Connection management - HTTP/1.1 and HTTP/2
- Synchronous and asynchronous communication
- Communication with Backend
  - AJAX, Fetch API
  - Webhooks
  - Server-Sent Events
- Polling
- Bidirectional communication - Web sockets

Source: Font: Arial, Font Size: 8 pts

Calling API from Frontend

Fetching data from the server

- How do you call a service from a client?
- The browser makes one or more HTTP requests to the server for the files needed to display the page, and the server responds with the requested files.
- The browser reload the page with the new data.
- So, instead of the traditional model, many websites use JavaScript APIs to request data from the server and update the page content without a page load.
- **This is called AJAX**
- The Fetch API enables JavaScript on a page to request an HTTP server to retrieve specific resources.

Fetch API

- The Fetch API provides an interface for fetching resources
- Provides a generic definition of Request and Response objects
- The fetch() method takes one mandatory argument, the path to the resource you want to fetch.
- It returns a promise that resolves to the response of that request (successful or not).
- You can optionally pass an init options object as a second argument (used to configure req headers for other types of HTTP requests such as PUT, POST, DELETE)

Fetch API Example

```
fetch("https://www.boredapi.com/api/activity")
    .then(response => {
        return response.json();
    })
    .then(data => {
        console.log(JSON.stringify(data));
        document.getElementById("h1id").innerText = data.activity;
    }
    )
    .catch(error => {
        alert('There was a problem with the request.');
    });
```

Axios

- Axios is a promise-based HTTP client for JavaScript.
- It allows you to
  - Make XMLHttpRequests from the browser
  - Make http requests from node.js
  - Supports the Promise API
  - Automatic transforms for JSON data

Axios

- Axios provides more functions to make other network requests as well, matching the HTTP verbs that you wish to execute, such as:
  - axios.post(<uri>, <payload>)
  - axios.put(<uri>, <payload>)
  - axios.delete(<uri>, <payload>)

Fetch vs Axios

- Fetch API is built into the window object, and therefore doesn’t need to be installed as a dependency or imported in client-side code.
- Axios needs to be installed as a dependency.
- However, it automatically transforms JSON data.
- If you use .fetch() there is a two-step process when handing JSON data.
- The first is to make the actual request and then the second is to call the .json() method on the response.

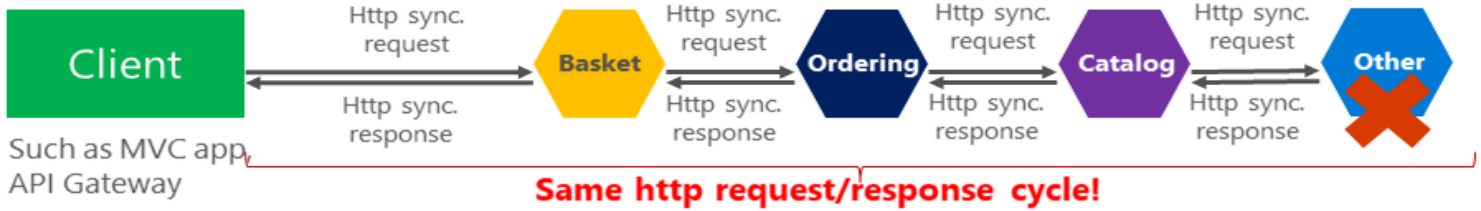
Synchronous vs Asynchronous communication

Synchronous vs Asynchronous communication

Synchronous vs. async communication across microservices

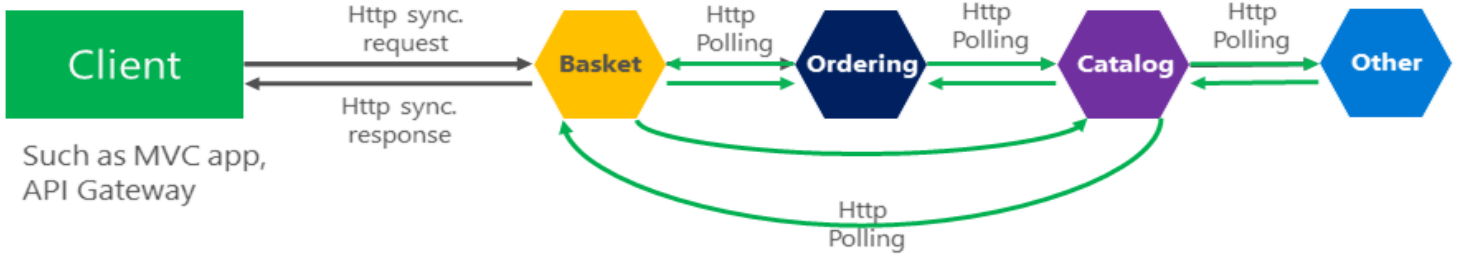
Anti-pattern

Synchronous
all request/response cycle



Asynchronous
Comm. across internal microservices
(EventBus: like **AMQP**)

"Asynchronous"
Comm. across internal microservices
(Polling: **Http**)



Long running Transactions

- API that performs tasks that could run longer than the request timeout limit.
- The server typically replies with a 202 Accepted Response indicating that the request is accepted and is in progress.
- HTTP is a **unidirectional** protocol, which means the client always initiates the communication.
- So client has to periodically check the server , if the work has been completed.

Approaches

- **How Server to Client Real Time Notification Response Works?**

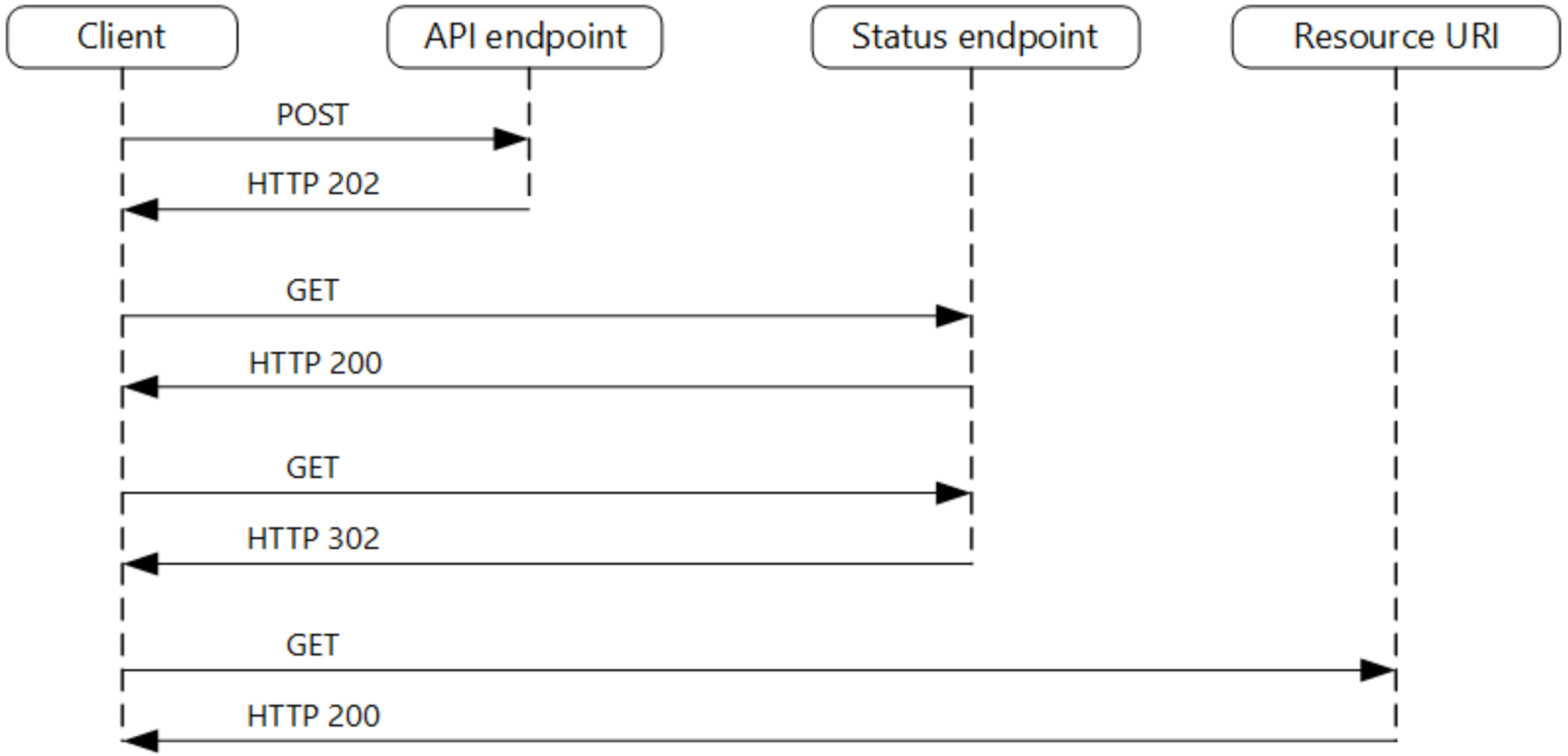
- Polling
- Webhooks
- Server Sent Events
- Websockets
- Message Queues

HTTP Polling

HTTP Polling

- HTTP Polling is a mechanism where the client requests the resource regularly at intervals.
- If the resource is available, the server sends the resource as part of the response.
- If the resource is not available, the server returns an empty response to the client.

HTTP Polling



HTTP Polling

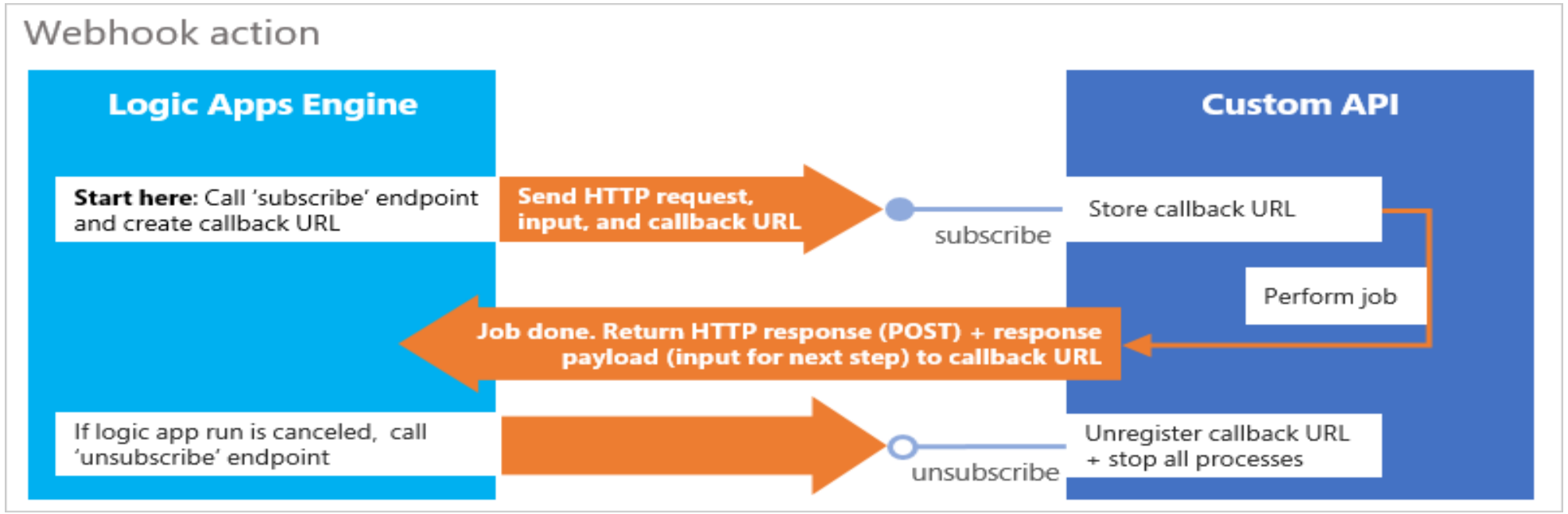
- An HTTP 202 response should indicate the location and frequency that the client should poll for the response.
- It should have the following additional headers:
  - Location: A URL the client should poll for a response status.
  - Retry-After: This header is designed to prevent polling clients from overwhelming the back-end with retries.

WebHooks

Webhooks

- A webhook is an HTTP-based callback function that allows lightweight, event-driven communication between 2 application programming interfaces (APIs).
- This callback is an HTTP POST that sends a message to a URL when an event happens.
- To set up a webhook, the client gives a unique URL to the server API and specifies which event it wants to know about.
- Once the webhook is set up, the client no longer needs to poll the server;
- When the specified event occurs, the server will automatically send the relevant payload to the client's webhook URL.
- Webhooks are often called reverse APIs or push APIs because they put communication responsibility on the server rather than the client.

Webhooks


Webhook action
Logic Apps Engine
Custom API
Start here: Call 'subscribe' endpoint and create callback URL
Send HTTP request, input, and callback URL
subscribe
Store callback URL
Perform job
Job done. Return HTTP response (POST) + response payload (input for next step) to callback URL
If logic app run is canceled, call 'unsubscribe' endpoint
unsubscribe
Unregister callback URL + stop all processes

Webhooks

- Eliminate the need for polling
- Are quick to set up.
- Automate data transfer.
- Are good for lightweight, specific payloads.

Webhooks

- client-side” application is the one making the request to the API on the “server-side”.
- The “client-side” must be running a server, and the “server-side” must be running a server.
- The “client-side” application makes an API request to the “server-side” server, and sends the “server-side” server a “webhook” to call once the “server-side” wants to notify the “client-side” application of some “event”.
- Once the “event” occurs, and the “server-side” application calls the “webhook” url, the server that is running on the “client-side” application will “receive” that “webhook” notification.
- **Front end applications eg. pure React JS, AngularJS, Mobile Apps, cannot use webhooks directly.**

- Webhooks basically are APIs implemented by API consumers but defined and used by API providers to send notifications of events.

Server Sent Events

Server-Sent Events

- Server-sent events (SSE) represent a unidirectional communication channel from the server to the client, allowing servers to push updates in real time.
- Unlike other technologies like WebSockets, SSE operates over a single HTTP connection.

Key Features of SSE

- **Simplicity**: Easy to implement and use.
- **Unidirectional Flow**: Data flows from server to client only.
- **Automatic Reconnection**: SSE connections automatically attempt to reconnect in case of interruptions.

Server-Sent Events

- Establishing the SSE Connection
- To initiate an SSE connection, the server sends a special text/event-stream MIME type response.
- On the client side, the EventSource API is used to handle incoming events.

Client Side

- To begin receiving events from the server , create a new EventSource object with the URL of a script that generates the events.
- Create an EventSource object to establish the SSE connection

```
const eventSource = new EventSource('http://localhost:3000');
```

- To receive message events, attach a handler for the message event:

```
eventSource.onmessage = (event) => {
    const data = JSON.parse(event.data);
 document.getElementById('output').innerHTML = `Received data: ${data.message}`;
  };
```

- This code listens for incoming message events and appends the message text to a list in the document's HTML.

Server Side

- The server-side script that sends events must respond using the MIME type text/event-stream.
- Each notification is sent as a block of text terminated by a pair of newlines.
- Event stream format
- data: {"message":"Server time: Thu Jan 18 2024 00:37:43 GMT+0530 (India Standard Time)"}
- A pair of newline characters separate messages in the event stream.

Use Cases and Applications

- **Real-Time Notifications**
- **Live Feeds and Dashboards**
- **Collaborative Editing and Chat Applications**

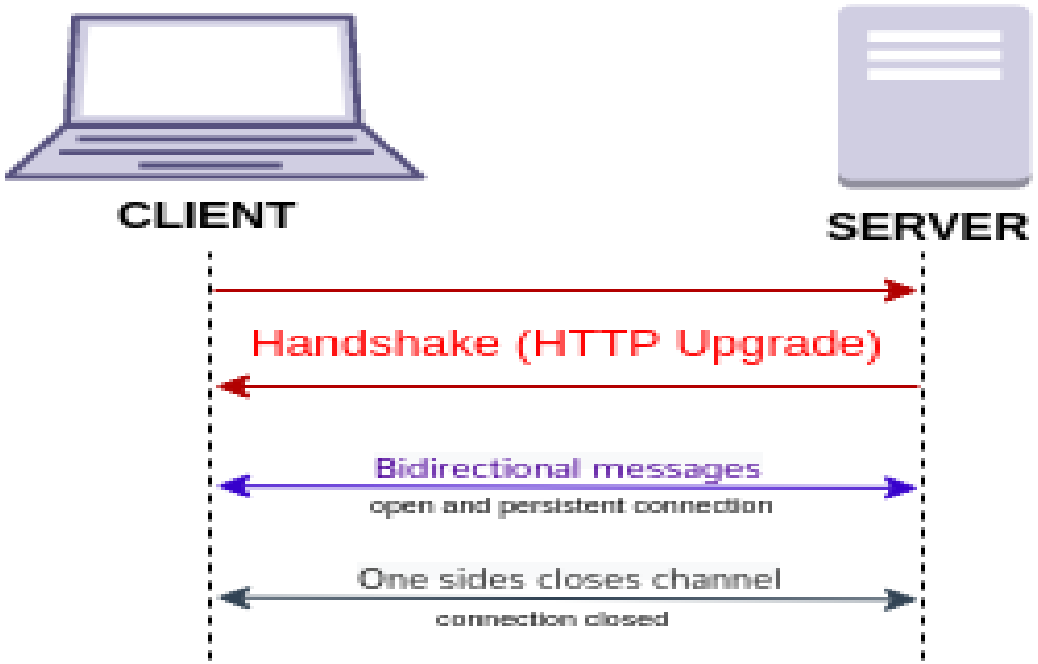
WebSockets

WebSockets

- WebSocket is a communications protocol that supports bidirectional communication over a single TCP connection.
- It is a living standard maintained by the WHATWG and a successor to The WebSocket API from the W3C.
- The WebSocket protocol enables full-duplex interaction between a web browser (or other client application) and a web server

WebSockets Handshake

- The WebSocket protocol specification defines ws (WebSocket) and wss (WebSocket Secure) as two new uniform resource identifier (URI) schemes.
- ws : [// authority] path [? query]



https://commons.wikimedia.org/wiki/File:Websocket_connection.png#/media/File:Websocket_connection.png

WebSocket - Upgrade

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==

Sec-WebSocket-Protocol: chat, superchat

Sec-WebSocket-Version: 13

Origin: http://example.com

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=

Sec-WebSocket-Protocol: chat

Headers

- The Sec-WebSocket-Key header is calculated by base64 encoding a random string of 16 characters, each with an ASCII value between 32 and 127.
- A different key must be randomly generated for each connection.
- The Sec-Websocket-Accept header is included with the initial response from the server.
- The server reads the Sec-WebSocket-Key, appends UUID 258EAFA5-E914-47DA-95CA- to it, re-encodes it using base64, and returns it in the response as the parameter for Sec-Websocket-Accept
- The client sends the Sec-WebSocket-Protocol header to ask the server to use a specific subprotocol.

Summary

- HTTP Polling
- Webhooks
- Server-Sent Events
- Websockets

Thank You!