ASSE Assignment 7 & 8 – Sub-Task 1 & 4

Group 5, St. Gallen, 25/11/2023

Sub-Task 1: *Critically reflect the (collective) decisions on the uniform HTTP API for the Auction House, discussing the advantages and disadvantages of the individual choices when creating this API.*

Generally, we had a good experience with the Interoperability team. The interoperability team pre-defined the requests for bids (placing/receiving) as well as the Auction representation and how the task is delegated to the winning bidder.

Bidding

The primary focus for the bidding process was on standardizing endpoints for bid placement and data representations. Bids are received at the /auction/auctionId path with a HTTP 204 response for accepted requests. The bidding process was effective due to well communicated data type representations. Furthermore, using the standard media type “application/json” allowed us to be more flexible due to the API being more versatile.

The use of JSON (JavaScript Object Notation) as the media type offers several advantages that align well with our goals for a robust and adaptable API design. Firstly, JSON's lightweight data-interchange format is both easy to read and write for humans, and simple to parse and generate for machines. This dual benefit enhances the API's accessibility and efficiency. The readability of JSON aids in easier debugging and maintenance. Secondly, JSON's flexibility allows for the representation of complex data structures in a concise and organized manner. This capability enables our API to handle various types of data without necessitating multiple formats. Such versatility is crucial when dealing with diverse data requirements as the system functionality expands. Thirdly, JSON's ubiquity in web development ecosystems means that it is well-supported across a wide range of programming languages and frameworks. This widespread support facilitates integration with other systems and applications, reducing compatibility issues and accelerating development cycles.

The bid receiving endpoint should, as per the specification, return an HTTP 204 status. However, this standard doesn't cater to other scenarios like closed auctions or duplicate bids. It would be helpful to ensure that varied responses are returned based on the result of the request (e.g., HTTP 202 for open auctions).

Auction

There was confusion on the Auction JSON Object representation as other teams included additional attributes to the Auction object that were not clearly discussed beforehand. During the PlugFest, this led to the majority of our debugging issues where we had to include new attributes to the Auction object, leading to a large number of changes throughout the auction house microservice. One of the additonal attributes that the interoperability team decided to add to an auction was the task inputData, which we generally think is not a good architectural decision. The auction house is not supposed to communicate any data about the task itself, as its only responsibility is handling the auctioning of the task. We believe that it would be better design to communicate the task information from the internal task list to the external

system, without adding any of the task information to the auction, apart from metadata such as the taskUri and taskType.

Task Delegation

Another decision was to directly create a shadow task in the winner's task list, essentially bypassing the notification to the winning auction house. While this led to some confusion in task creation across different implementations, the implementation was well pre-defined by the Interoperability team with the pre-existing POST new task request and therefore did not lead to issues. We believe that this choice allowed us to delegate the task in an easy and efficient way, though it comes with drawbacks such as increased complexity due to the use of resources for the shadow task as well as possible data integrity issues due to the duplicate task and additional communication overhead.

Sub-Task 4 - *Comparing W3C WebSub and MQTT for event-based interaction*

W3C WebSub

W3C WebSub enables communication (via HTTP) between publishers and subscribers through a Hub. First, a subscriber needs to discover the hub advertised by the publisher's topic and then the subscriber can create a POST request to the hub to subscribe to updates. Whenever, the publisher notifies the hub of new content, the hub delivers the content of that topic to each subscriber (s. Figure 1 for the original WebSub diagram).

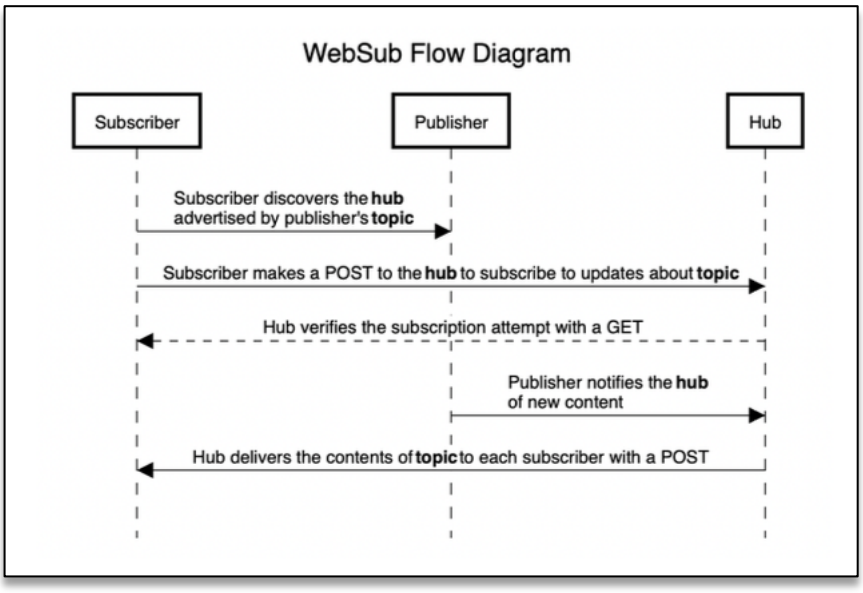

Figure 1 - WebSub Diagram[1]

In this context, the Hub serves as the distributor of content. Subscribers interact with the Hub using HTTP POST and GET requests, facilitating asynchronous communication. This setup enables ongoing, continuous communication between the parties.

[1] https://www.w3.org/TR/websub/

The main advantage of WebSub is that it is well designed for the web & integrates seamlessly with HTTP/HTTPS protocols, making it ideal for web-based applications. Disadvantages include dependency on HTTP (high overhead), limited support of varying QoS levels, high-power usage / communication overhead.

W3C WebSub is particularly well-suited for web applications where the primary goal is to deliver real-time content updates efficiently and seamlessly within the existing HTTP/HTTPS infrastructure. Its ideal context includes scenarios where the simplicity and familiarity of HTTP are prioritized, and the overhead associated with this protocol is not a significant concern. Moreover, WebSub is ideal for scenarios where the publisher/subscriber model needs to be integrated into web services without the need for additional protocol support. It is also beneficial in situations where we are looking for a straightforward pub/sub implementation without the complexities of managing MQTT brokers or dealing with connection persistence. WebSub is best utilized in scenarios that demand real-time, web-based communication with minimal setup complexity. It suits applications that can afford the higher HTTP overhead in exchange for the ease of use and integration with standard web technologies.

MQTT

MQTT is a lightweight messaging protocol that also operates asynchronously. Distinct from W3C WebSub's reliance on the HTTP request/response model, MQTT utilizes its own publish/subscribe protocol. In this system, a central MQTT broker manages connections, linking the MQTT clients of publishers and subscribers. This architecture allows MQTT to attain space decoupling, time decoupling, and synchronization decoupling by separating the messages exchanged between publishers and subscribers.

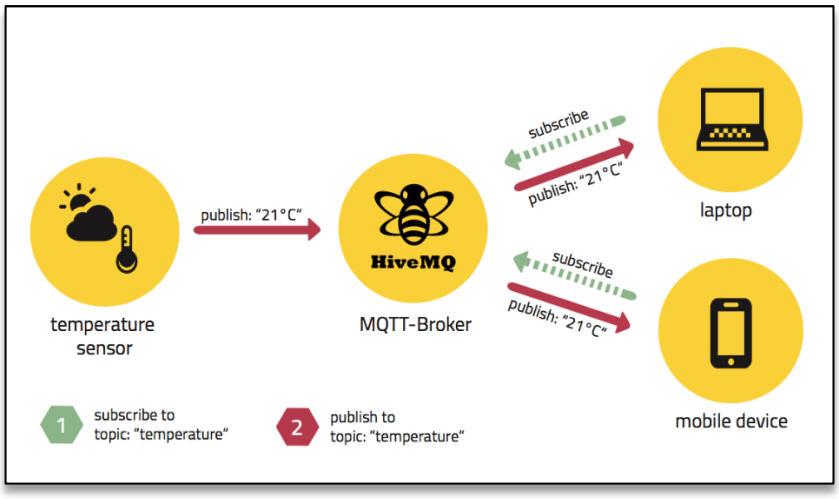

Figure 2 - MQTT Diagram[2]

Figure 2 shows an example of an MQTT framework. In this example, a temperature sensor acts as an MQTT publisher. The sensor detects the temperature and sends this data ("21°C")

[2] https://www.eclipse.org/community/eclipse_newsletter/2014/october/article2.php

to an MQTT broker—represented here by HiveMQ, which is a popular MQTT platform. The role of the MQTT broker is central to the MQTT protocol; it receives messages published to it and then distributes these messages to the clients that have subscribed to the corresponding topic. The diagram shows two subscribers: a laptop and a mobile device. Both have subscribed to the "temperature" topic with the MQTT broker. When the temperature sensor publishes the temperature reading to this topic, the broker forwards the message to all subscribers of that topic. This means the laptop and mobile device both receive the temperature data ("21°C").

The main advantage of MQTT is that is well designed for IoT use case (or generally in environments with low bandwidth, high latency, or unreliable networks) due to its lightweight protocol Another advantage is that MQTT supports multiple QoS levels, ensuring message delivey as per the requirements of the application. However, disadvantages are higher complexity and the requirement of a broker (that can be a single point of failure if not managed properly).

MQTT is an excellent fit for IoT ecosystems that include devices with limited computational power and scenarios where network bandwidth is at a premium. Its lightweight nature and efficient communication protocol make it highly suitable for such constrained environments. The protocol's support for multiple levels of Quality of Service (QoS) caters to the diverse requirements of message delivery reliability, from scenarios where it's acceptable to miss a message, to others where message delivery must be guaranteed under all circumstances. This flexibility is particularly crucial in applications where the consistent and reliable transmission of data is non-negotiable, such as in monitoring critical infrastructure or real-time control systems. Furthermore, MQTT's design is centered around minimizing the overhead of messages, which preserves both bandwidth and energy, extending the life of battery-powered IoT devices. It is the protocol of choice for situations that not only require efficient and reliable communication but also need to ensure the data is exchanged effectively despite potential network disruptions.

Choosing Between WebSub and MQTT:

When deciding between WebSub and MQTT, the application's context plays a pivotal role. For standard web applications, especially for content distribution and updates, WebSub is the preferred choice. Its integration with standard web technologies makes it ideal for scenarios where MQTT's complexity isn't necessary. Conversely, MQTT is the go-to for IoT applications or in environments with constrained devices and unreliable networks, where its ability to provide different levels of message delivery assurance is invaluable. In cases requiring real-time interactivity within a web context, WebSub's efficiency and ease of use make it more suitable. However, for IoT scenarios involving extensive cross-device communication, MQTT's robustness and scalability make it the superior option. Our choice to use WebSub was influenced by these considerations, alongside our practical experiences and the specific requirements of our application environment, such as robust network conditions and ample computing resources.

After careful consideration and drawing from our practical experiences, we would prefer to proceed with WebSub for our system. This decision was grounded in the specific needs and

conditions of our application environment. Given that our network conditions are robust and each application runs on VMs with 8 GB of RAM, providing ample computing resources, we are not constrained to prioritize technologies geared towards limited conditions.

Furthermore, our development experience with WebSub has been notably smoother and more intuitive compared to MQTT. We found WebSub to be more straightforward in terms of development, debugging, and integration, particularly when interfacing with other teams' applications. This ease of use and the ability to work seamlessly in a well-resourced environment played a significant role in our decision.

WebSub's HTTP-based communication model also aligns well with our system's architecture, offering a more natural fit for web-centric applications. This compatibility, along with its efficient real-time update mechanism, makes WebSub an ideal choice for our current setup. Its simplicity and familiarity within the web development paradigm have allowed us to develop and maintain our application more effectively, leading to a more streamlined and cohesive team workflow.