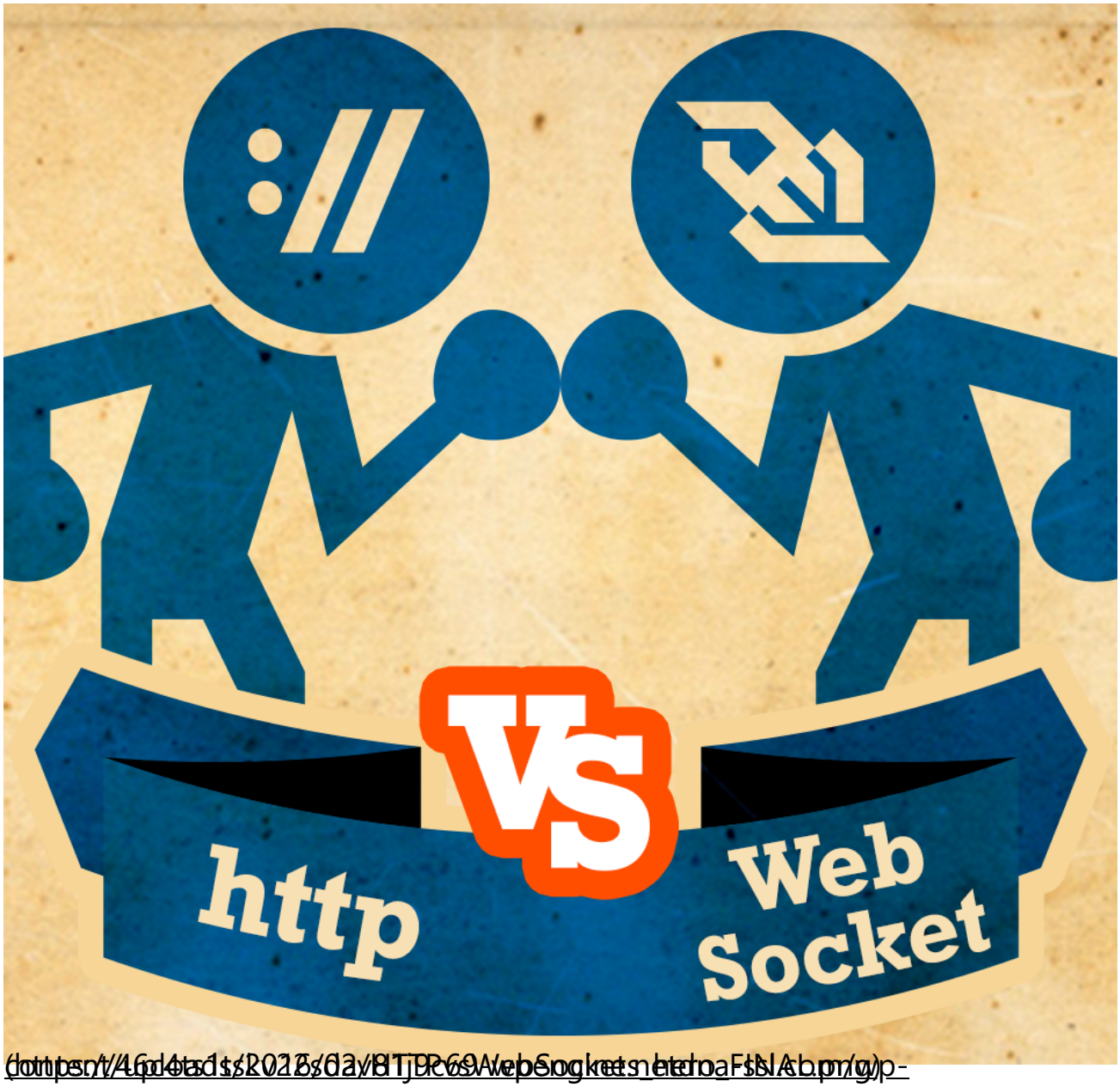
March 14, 2016 Mobile (https://blogs.windows.com/windowsdeveloper/category/mobile/)

When to use a HTTP call instead of a WebSocket (or HTTP 2.0)

Windows Apps Team (https://blogs.windows.com/windowsdeveloper/author/windowsappsteam/)



WebSocket icon by w3.org (CC BY)

It isn't always easy to know when it might be better to use HTTP (http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) request/responses versus WebSockets (http://en.wikipedia.org/wiki/WebSocket) for your project, Universal Windows Platform app or not, especially when you're facing so many other critical decisions for your project/app at the same time. To help provide some clarity (on this decision, at least), today's blog will compare HTTP requests (REST (http://en.wikipedia.org/wiki/Representational_state_transfer)) versus WebSocket communication for you and lay out some general guidance you can follow. This guidance is general and not always black and white, there may be situations where the technology that best fits actual client requirements and scenarios goes against this general guidance. As the developer, we suggest you use your best judgement.

One thing you may be wondering is why not HTTP 2.0? HTTP 2.0 connections could be used in place of a WebSocket depending on how they will be used as they have bi-directional messaging abilities, but they must follow the request/response pattern. Stack Overflow has as great discussion on this HTTP 2.0 versus WebSockets (http://stackoverflow.com/questions/28582935/does-http-2-make-websockets-obsolete). Additionally, while UWP does support HTTP 2.0 (https://channel9.msdn.com/Events/Build/2015/3-88), depending on who else consumes your service, other clients may not support it. Using HTTP 2.0, therefore, could reduce the number of supported clients for many applications.

Too long; didn't read

In general, follow these two principles when coming to a decision:

> Assume your API should be traditional HTTP by default.
>
> Review the guidance below and try to convince yourself your API should be WebSocket.

For a quick, visual comparison of the two, check out our TL;DR chart:

(https://46d4ts1s2l7y3c8j9c69-wpengine.netdna-ssl.com/wp-
content/uploads/2016/02/81_j9tc69-websocket.png)

When HTTP Is better

When evaluating whether HTTP is the better choice, you may find it helpful to think in terms of scenarios. And when it comes to scenarios, these are the ones for which you'll find HTTP is particularly well-suited.

Retrieve Resource

A client wants the current state of a resource and does not want or require ongoing updates.*Example:* A football fan wants to check the result of a game. If the game were from last week, the game result would be stable and additional updates very unlikely. In that case, HTTP would be a sound choice. Not so, however, if the game were currently in progress. For a game in progress, the score will change constantly and updates will be frequent. In that case, a WebSocket would likely be the better choice.

Highly Cacheable Resource

Resources benefit from caching when the representation of a resource changes rarely or multiple clients are expected to retrieve the resource. The word "rarely" is intentionally left vague here, because it must be judged against the anticipated client access pattern and not against a fixed amount of time. Highly volatile resources may still be highly cacheable if they are frequently retrieved, especially if by multiple clients.Note that the WebSocket design does not allow explicit or transparent proxies to cache messages, which can degrade client performance.

Example: Football scores from the previous week's game are highly cacheable because they are stable and unlikely to change, so HTTP would be a good fit. Football scores from a game in progress, however, are likely to change frequently. In that case, the resource is not highly cacheable, so a WebSocket becomes the better fit.

Idempotency and Safety

HTTP methods have well-known idempotency and safety properties. A request is "idempotent" if it can be issued multiple times without resulting in unique outcomes. This property enables clients to accommodate timeouts or transient networking issues in a straightforward manner. A request is "safe" if it does not modify the resource being acted upon. This property is the key to enabling caching (or pre-fetching).Safety and idempotency are key enablers for designs that must be resilient to communication failure. HTTP is ideally suited to this scenario because the HTTP methods have broadly shared safety and idempotency expectations. The WebSocket protocol leaves these issues to the messaging layer design (which means there are no broad industry standards).

Error Scenarios

HTTP is designed around the request-response messaging pattern, which means it has extensive support for error scenarios. The HTTP design allows for responses to describe errors with the request, with the resource, or to provide nuanced status information to differentiate between success scenarios.The WebSocket protocol offers support only for error scenarios affecting the establishment of the connection. Once the connection is established and messages are exchanged, any additional error scenarios must be addressed in the messaging layer design.

Synchronized Events

The request-response pattern is well suited to operations that require synchronization or that must act in a serialized fashion. The HTTP response represents a definitive conclusion to a specific request, allowing subsequent actions to be gated on it. In WebSockets, this detail is left up to the messaging layer design.The WebSocket protocol offers no guarantee a message will be acknowledged in any form. Although, in your design, you should try to avoid assuming that clients will always synchronize their actions perfectly. Do your best to offer a stateless interaction pattern, or allow clients to make requests in parallel, because it will make your application more resilient to unavoidable networking issues or buggy client behavior.

When a WebSocket is typically better

Just as with HTTP, you'll find that a WebSocket has its own set of scenarios that illustrate when it may be the best choice for your project. Remember our caution at the start of this blog, however, as the following guidance does not take any special messaging protocol into account.

Fast Reaction Time

When a client needs to react quickly to a change (especially one it cannot predict), a WebSocket may be best. Consider a chat application that allows multiple users to chat in real-time. If WebSockets are used, each user can both send and receive messages in real-time. WebSockets allow for a higher amount of efficiency compared to REST because they do not require the HTTP request/response overhead for each message sent and received.

Ongoing Updates

When a client wants ongoing updates about the state of the resource, WebSockets are generally a good fit. WebSockets are a particularly good fit when the client cannot anticipate when a change will occur and changes are likely to happen in the short term.HTTP, on the other hand, may be a better fit if the client can predict when

changes occur or if they occur infrequently—for example, a resource that changes hourly or changes only after it knows that a related resource is modified. If the client doesn't know that the related resource is modified (e.g. because some other client modified it, or the service modified it), then WebSockets are better.

Ad-hoc Messaging
The WebSocket protocol is not designed around request-response. Messages may be sent from either end of the connection at any time, and there is no native support for one message to indicate it is related to another. This makes the protocol well suited to "fire and forget" messaging scenarios and poorly suited for transactional requirements. The messaging layer must address your transactional needs if that's needed in your application.

High-Frequency Messaging with Small Payloads
The WebSocket protocol offers a persistent connection to exchange messages. This means that individual messages don't incur any additional tax to establish the transport. Taxes such as establishing SSL, content negotiation, and exchange of bulky headers are imposed only once when the connection is established. There is virtually no tax per message. On the other hand, while HTTP v1.1 may allow multiple requests to reuse a single connection, there will generally be small timeout periods intended to control resource consumption. Since WebSockets were designed specifically for long-lived connection scenarios, they avoid the overhead of establishing connections and sending HTTP request/response headers, resulting in a significant performance boost.However, this should not be taken to extremes. Avoid using WebSockets if only a small number of messages will be sent or if the messaging is very infrequent. Unless the client must quickly receive or act upon updates, maintaining the open connection may be an unnecessary waste of resources.

You might be doing it wrong if…

You *might* be using HTTP incorrectly if…

Your design relies on a client polling the service often, without the user taking action.

Your design requires frequent service calls to send small messages.

The client needs to quickly react to a change to a resource, and it cannot predict when the change will occur.

The resulting design is cost-prohibitive. Ask yourself: Is a WebSocket solution substantially less effort to design, implement, test, and operate?

You *might* be using WebSockets incorrectly if:

> The connection is used only for a very small number of events, or a very small amount of time, and the client does not need to quickly react to the events.
>
> Your feature requires multiple WebSockets to be open to the same service at once.
>
> Your feature opens a WebSocket, sends messages, then closes it—then repeats the process later.
>
> You're re-implementing a request/response pattern within the messaging layer.
>
> The resulting design is cost-prohibitive. Ask yourself: Is a HTTP solution substantially less effort to design, implement, test, and operate?

Conclusion

As a starting place, we suggest assuming your API should be traditional HTTP by default unless the guidance in this blog can convince you that WebSockets would truly be the better choice. That said, this guidance is general and not written in stone. As with so many other aspects of a given project, the technology that ultimately works best for you may not always be what the general guidance would suggest.

For additional resources on UWP documentation with HTTP and Websockets:

> MSDN documentation on WebSocket (https://msdn.microsoft.com/en-us/library/windows/apps/mt186447.aspx)
>
> UWP Websockets sample on GitHub (https://github.com/Microsoft/Windows-universal-samples/tree/master/Samples/WebSocket)
>
> MSDN documentation on HttpClient (https://msdn.microsoft.com/en-us/library/windows/apps/mt187345.aspx)
>
> UWP HttpClient sample on Github (https://github.com/Microsoft/Windows-universal-samples/tree/master/Samples/HttpClient)

Post written by Jeff Carnahan, a Program Manager at 343 Industries

Tags: http (https://blogs.windows.com/windowsdeveloper/tag/http/) http vs websocket (https://blogs.windows.com/windowsdeveloper/tag/http-vs-websocket/) REST (https://blogs.windows.com/windowsdeveloper/tag/rest/)

websocket (https://blogs.windows.com/windowsdeveloper/tag/websocket/)