Real-Time API Connectivity Scenarios

Table of Contents

- Connections
  - Plain PTL over TCP
  - WebSocket
- Pseudo connections
- Request / response HTTPS
- Considerations on software design
- Connectivity use cases
  - Websites
  - Standalone Applications for anonymous access
  - Standalone Applications for access by known/registered users

Connections

The Real-Time API Platform primarily uses stateful, long-lived connections where both sides are allowed to spontaneously send messages to communicate with clients. This is a prerequisite to allow an efficient implementation of features such as push subscriptions or cache invalidations.

Plain PTL over TCP

This connection type may exchange API messages in a serialized, binary form directly over a TCP socket. It is the most efficient method and is used by most client libraries provided by FactSet.

WebSocket

This connection type may exchange all API messages in several different serialization forms:

- Plain PTL
- JSON
- JavaScript compatible JSON

These forms are also available as compressed variants.

Pseudo connections

If a client opts to contact the Real-Time API Platform using a wire protocol that does not allow such statefulness, longevity or the spontaneous sending of messages from both sides - such as HTTP (without the WebSocket protocol extension) - a "Pseudo Connection" has to be established as a (somewhat less efficient) substitute. This is also known as “HTTP Long Polling”.

This connection type may exchange all API messages in several different serialization forms:

- JSON
- JavaScript compatible JSON

These forms are also available as compressed variants and base64 encoded variants.

Request / response HTTPS

A request / response interface for exchanging API endpoint request messages exists for lightweight applications that do not need the capability of connections like push functionality.

This interface requires authentication via high-level-token, also sometimes referred to as “high level request token”.

The request and response format is specified in the *Request and Response Specification*.

Considerations on software design

A number of design considerations should be discussed when developing in the context of the Real-Time API.

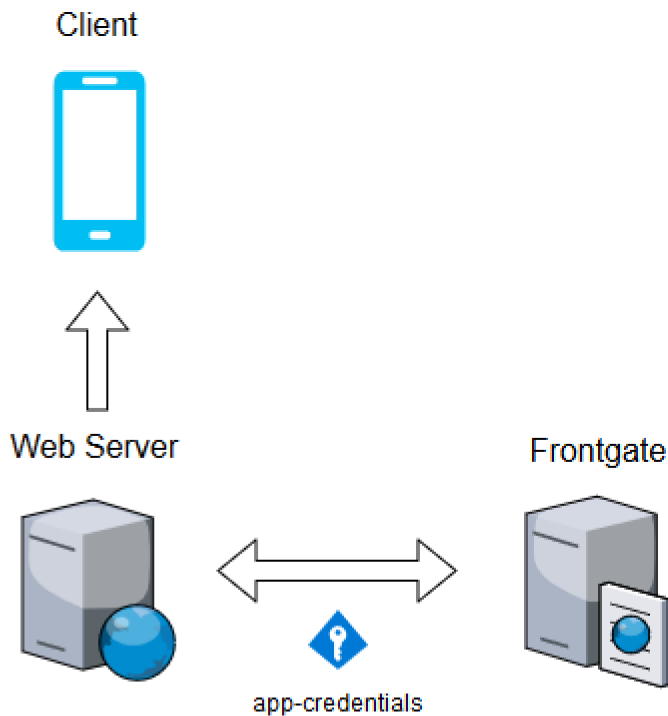
1. Web server based or standalone application
    1. Is your software a web server-based application?
    2. How is data to be delivered to the client - through the web server or directly to the client application?
2. User management
    1. Does the application make use of user accounts?
    2. Where are users managed: on the customer’s side or on the FactSet platform?
3. Push requirements
    1. Is push required? It is required if you want to subscribe to any updates.
    2. Will you have push requirements in the future?

Connectivity use cases

Websites

Use case 1: Website, FactSet-hosted, no push

FactSet provides a website on behalf of a customer, for use by any (also anonymous) users.



Credentials involved

The FactSet web server knows the app-credentials and keeps a persistent connection to a FactSet-hosted Frontgate.

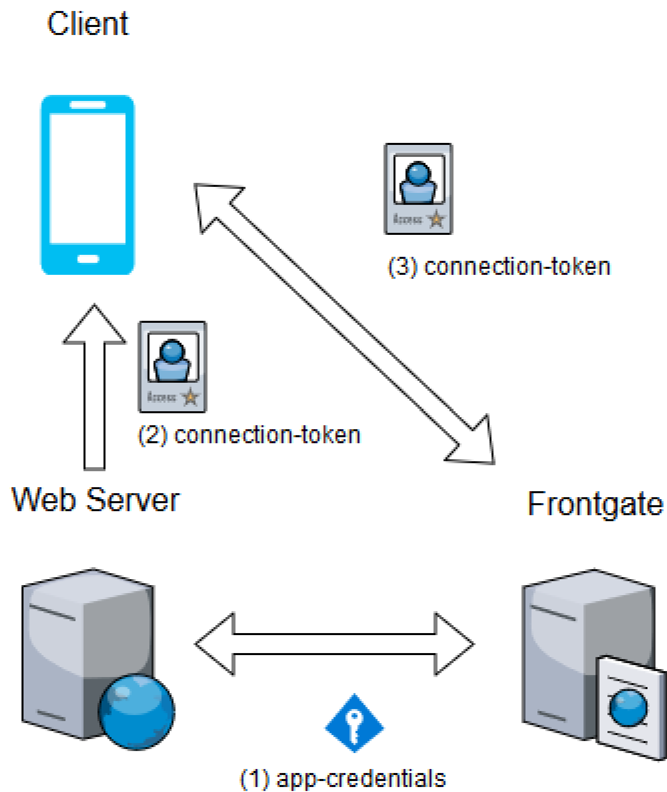
Recommended software

- Server part: API client libraries (PHP, Java, C#, Node JavaScript)

Use case 2: Website, FactSet-hosted, push to anonymous or registered users

FactSet provides a website on behalf of a customer, for use by any (also anonymous) users.

JavaScript code running inside the users browsers require the ability to push-subscribe certain data from the Real-Time Platform API.



Credentials involved

The FactSet web server knows the app-credentials and keeps a persistent connection to a FactSet-hosted Frontgate.

When the web server is asked to deliver a page using push-subscriptions, the application code creates a connection-token that is sent to the user's browser as part of the JavaScript code contained in the delivered HTML page.

If the user is a known individual, registered as a certain user with Real-Time API Platform, the connection-token is created for his specific id_user.

If the user is an unknown, anonymous user, the connection-token is created for the id_user of a technical user the application has created for that very purpose.

Recommended software

- Server part: API client libraries (PHP, Java, C#, Node JavaScript)
- Client part: API client libraries (JavaScript)

Use case 3: Website, hosted by 3rd party, push to anonymous or registered users

Customer hosts a website, for use by any (also anonymous) users.

JavaScript code running inside the users browsers provides the ability to push-subscribe certain data from the API.

Credentials involved

Same as (1) or (2).

For optimization reasons, a deployed Frontgate server instance might be used instead of a FactSet-hosted Frontgate. In that case, push-subscribing JavaScript code in the user's browser needs to contact the same deployed Frontgate that was used to create the connection-token.

Recommended software

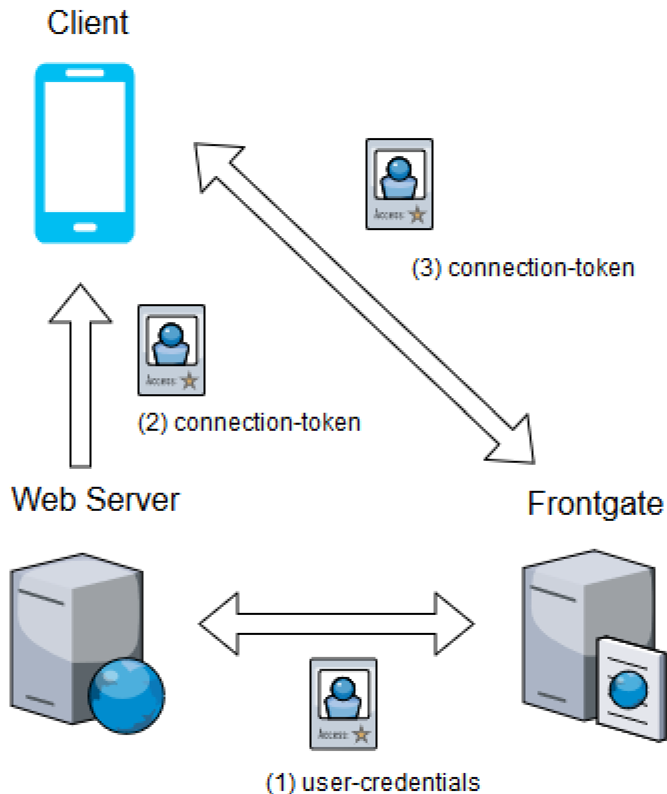
- Server part: API client libraries (PHP, Java, C#, Node JavaScript)
- Client part: API client libraries (JavaScript)

Standalone Applications for anonymous access

A standalone "App" is a software application that is not accompanied by some application-specific web-service but meant to directly retrieve data from a Frontgate server.

Use case 4: “App”, only requiring request/response retrieval of data, any API message type

In this use case, anyone is allowed to use the “App”, without any registration. This use case supports all API message types but requires a connection from the client to Frontgate.



Credentials involved

FactSet sets up the application but does not disclose the app-credentials to the “App” programmer(s).

FactSet creates a technical user associated with the application and makes the user-credentials of this user known to the "App" programmer(s) for inclusion into the “App” code.

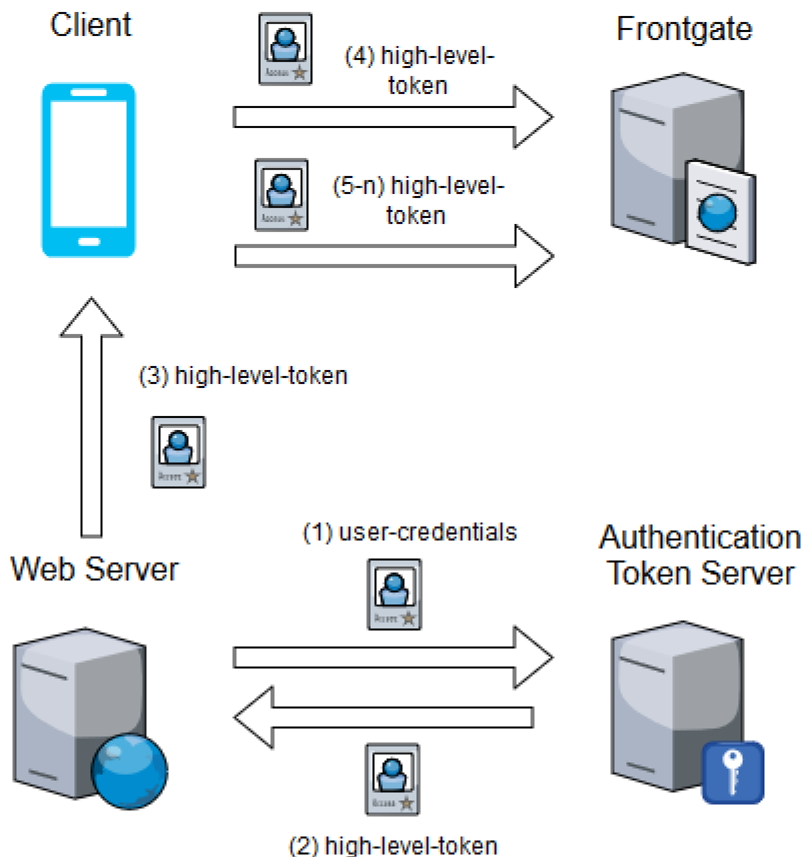
This does, of course, imply that anyone looking into the “App” code can make use of API services also outside of the “App”, subject to the quantity limit restrictions configured for the technical user, which should be chosen appropriately restrictive.

Recommended software

- Server part: API client libraries (PHP, Java, C#, Node JavaScript)
- Client part: API client libraries (JavaScript)

Use case 5: "App", only requiring request/response retrieval of data, only Endpoint messages

In this use case, anyone is allowed to use the "App", without any registration. This use case supports only API Endpoint messages, but may use Frontgate's https request / response interface.



Credentials involved

Same as (4), but if the "App" programmer(s) feel incapable of using the (more efficient) persistent connection mechanism that Frontgate provides, another option is to use the API Authentication Token Server web-service that returns a high-level-token for the technical user when requested with the correct user-credentials by the "App", so the "App" can subsequently resort to only place individual HTTPS requests using that token, without establishing a persistent connection.

Recommended software

- Server part: any https client
- Server part: any https client

Use case 6: "App", any API message type, also requiring push-subscriptions

Same as (4), but the "App" also needs to push-subscribe for certain data provided by the API.

Credentials involved

Same as (4).

Recommended software

- Server part: API client libraries (PHP, Java, C#, Node JavaScript)
- Client part: API client libraries (JavaScript)

Standalone Applications for access by known/registered users

Use case 7: "App" for use by known/registered users, only requiring request/response retrieval of data

In this use case, only known/registered users are allowed to use the "App".

Credentials involved

A newly installed "App" can either:

(a) send an API request asking to create a new user, receiving an id_user and user-client-secret as a result, which is then stored locally by the "App" for logging in henceforth

(b) ask the human user to enter some username / passphrase combination to identify himself as a certain known, already registered user

(b2) same as (b), but have some indirection over a 3rd-party service, where the 3rd-party service authenticates the username / passphrase and returns a pair of id_user and user-client-secret to the "App" that is then used to connect to the Frontgate.

For (b) and (b2) there would obviously the requirement to maintain users to use the "App", then some application-specific web-service (knowing the app-client-secret and app-shared-secret) to administrate users should be implemented, which would also take care of delivering the initial user-client-secret to the user. The "App" is then still "standalone" for all its normal users.

The theoretical possibility of having the "App" itself contain an interface for user administration is not recommended, as such would require the "App" to be supplied with app-client-secret and app-shared-secret, which would be a security issue if hard-coded, and inconvenient for a human user to enter interactively.

Recommended software

- Server part: API client libraries (PHP, Java, Node JavaScript)
- Client part: API client libraries (JavaScript)

Use case 8: "App" for use by known/registered users, also requiring push-subscriptions

Same as (7), but the "App" also needs to push-subscribe for certain data provided by the API.

Credentials involved

Same as (7), but there is no more option to spare the "App" programmer(s) from being able to establish a persistent connection to the Frontgate, so it's also no longer an option to only use a high-level-token, which isn't usable for push subscriptions.

Recommended software

- Server part: API client libraries (PHP, Java, C#, Node JavaScript)
- Client part: API client libraries (JavaScript)