How does LinkedIn identify its users online?

How does LinkedIn identify its users online? How does it know when a user on its platform is online or has gone offline? How does it show the online status of our connections to us? What is happening on the backend? What are the technologies involved?

1. What is the LinkedIn Real-Time Presence Platform?

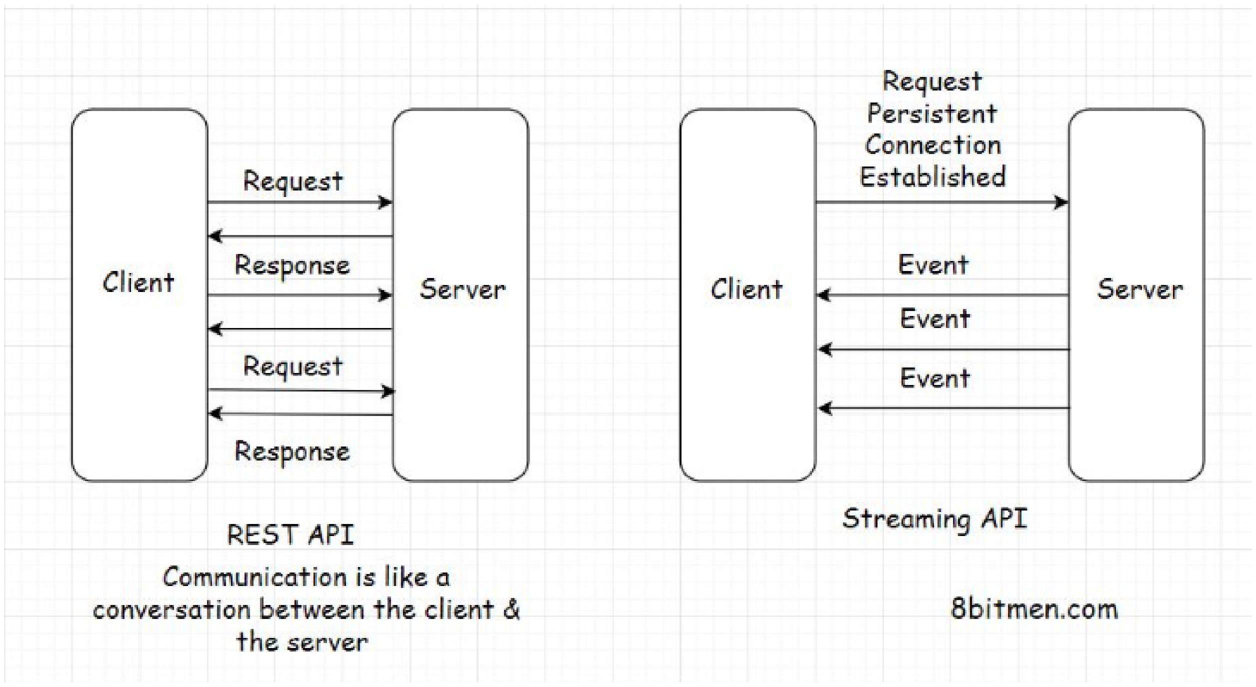
LinkedIn's real-time presence platform enables us to know if a user is online & if offline what was the last time he logged in, technically the *last active timestamp*. We can see a green dot on the user's profile either on his profile or in the messaging feature.

This is pretty convenient & helps a great deal in knowing if it's the appropriate time to ping someone & start a conversation.

How do we write such a system, which runs smooth in a distributed environment & is also resilient to network failures?

2. Establishing an Asynchronous Connection Between Client (User) & the Backend

In general, when we think of writing a feature which involves a ***persistent connection* between the client & the server**, for the need for streaming in information in real-time. There are a few ways which pop up in our minds such as *polling, long polling, server-sent push events, web sockets* etc.


Client
Request
Response
Request
Response
Server
REST API
Communication is like a conversation between the client & the server
Client
Request Persistent Connection Established
Event
Event
Event
Server
Streaming API
8bitmen.com

All of these client-server communication designs have their respective use cases, pros & cons. There no one solution which fits all.

For instance, if we take **polling**, more commonly known as *AJAX*. It is the simplest way to establish an asynchronous connection with the server & fulfils the majority of the use cases. But it has its downsides too.

If we are not sure when an event would occur on the backend. *Polling* the server every now & then within a stipulated time adds on to the traffic & consumes bandwidth.

Also, it wouldn't work that smooth in an app where we need to see the status of a vehicle moving around in a map in real-time. In this scenario, a **continual open connection between the client & the server would fit best like WebSockets**.

Why I am telling you all this? Just to give you a context of the things, common client-server communication solutions for writing a real-time communication platform.

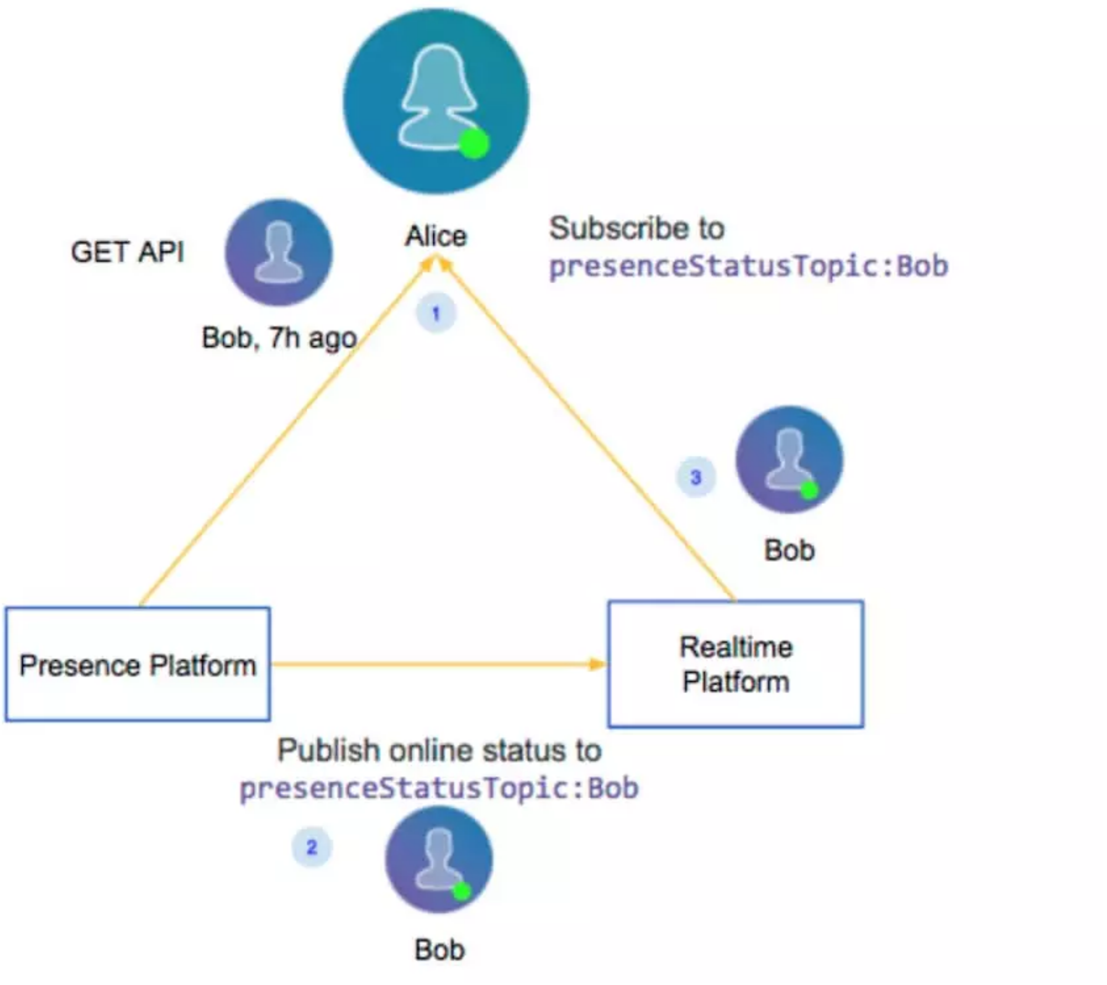
Speaking of LinkedIn's Real-Time User Presence Platform, there are two modules. *The Real-Time Delivery & the User Presence Platform*

3. LinkedIn's Real Time Delivery & the Presence Platform

The real-time delivery platform is the core component of the real-time user presence architecture.

It's a *publish-subscribe system* which streams data to the client be it web or a mobile client over a persistent connection as the events occur.

It's built using the *Play framework*, specifically written for modern web applications, based on *Akka*. What is Play? What is Akka? Why use Play instead of Spring? Which is naturally the most used Java framework in the industry.



The architectural diagram above shows the flow between the user, LinkedIn Presence Platform & the Realtime platform.

*When the user logs into LinkedIn, he establishes a persistent connection with the LinkedIn's Real-Time Platform***.** As, soon as the connection is established, the platform knows that the user is online & puts a green icon on his profile.

Now, after a user logs in he would also want to know if his connections are online.

To achieve this the Real-Time Platform makes the user subscribe to a topic for his connection’s presence status. The Real-Time Platform is a publish-subscribe platform remember? ?

Now when the user’s connections come online, the platform publishes an event to the topic & notifies all of the subscribers of that particular topic. So now all the subscribers, including the user, would know that a particular connection is online.

3.1 What’s the role of LinkedIn’s Real-Time Presence Platform?

The real-time presence platform keeps a check on the persistent connection with the client. It monitors if the user is online or has gone offline. Eventually, it publishes the presence status to the topic of that user, to which he has subscribed, so that the real-time delivery platform can send the updates to the subscribers of that topic.

3.2 How Does the Presence Platform Handle Network Disconnections?

Network disconnections are pretty common scenarios especially if the user is logged in from the mobile. He might be moving through a subway or a place with a low network reach. Within a certain stipulated time, the user might get disconnected several times. This would make his online status fluctuate often. The more he disconnects the more he fluctuates.

Imagine this how it would look on the UI? More like a fun disco light.

To manage this the presence platform doesn’t immediately change the user status, as soon as he disconnects. Rather the platform sends periodic heartbeats to the connected members. If the user connects again within a certain time say x seconds. The platform doesn’t show him offline. On the UI the user is online, all along.

If the disconnected user doesn’t reconnect within x seconds. The presence platform sends the update to the real-time delivery platform & in turn the delivery platform updates all the subscribers of that user/topic & persists the user’s presence state in a distributed Key/Value database with the last active timestamp.

Also, the common state of the user’s presence in stored in the database, which allows any node to do the same processing for any member.

An end user is connected to the Presence platform via the *REST API*. The entire flow is really fast, it hardly takes less than 200ms.

The Real-Time Delivery Platform is built for massive scale, can handle approx. 1.8K queries per second & powers the LinkedIn Feed, Messaging, Notifications etc.

Now let's talk about the underlying tech involved in building the real-time system.

4. Why Play Framework? Why not Spring MVC?

Play framework is really lightweight, stateless & has a web-friendly architecture. Being **stateless** makes it more conducive to write applications to run on distributed environments.

Play is specifically written for writing modern web applications. It has a **fully asynchronous model,** is built on Akka & has minimal resource consumption like CPU, memory & threads for highly scalable applications.

Due to the reactive model of Akka apps built on Play can scale seamlessly both vertically & horizontally. Being built for real-time web applications **Play inherently supports *WebSockets***, *Comet, EventSource* etc.

Akka enables the developers to focus on the business logic instead of writing low-level code to provide a reliable application behavior, fault tolerance & high performance.

The framework provides multi-threaded behavior without the need for concurrency constructs like locks or atomic. A clustered high availability architecture to scale out horizontally without any additional overhead.

Most importantly **Play supports a non-blocking IO inherently** unlike Spring MVC.

What is Non-Blocking?

Remember that *@async* annotation in Spring MVC? Which would put the flow, for instance calling the database, in a separate thread & would enable the main flow to move on even if the results are not returned from the database.

You can also think of the Executor framework, which runs different processes in separate threads.

Play framework supports that inherently.

Well, when I can split the processes into different threads with Spring annotations & the Executor framework. Why do I use Play?

Well, the primary difference is when the application requirements are truly asynchronous, there are so many threads generated for every micro process that it becomes impossible to manage things.

It’s a spaghetti of threads. This is where Play really shines. It has a completely event-driven architecture, as opposed to the traditional servlet architecture, keeping the number of threads to a bare minimum.

Akka’s use of the actor model provides a level of abstraction which makes it easier to write concurrent, parallel & distributed systems.

The traditional server architectures find it really difficult to manage, high throughput asynchronous requirements. Imagine how would the Spring MVC code look littered with @*async* annotations everywhere. It would be a nightmare to debug & manage.

Though Spring too has released a framework for the Reactive architecture called the Reactor.