Real-time Applications

Software System Design

Spring 2024 @ Ali Madooei

Learning Outcomes

By the end of this lecture, you should be able to:

- Understand the significance and definition of real-time systems in modern software.
- Differentiate between traditional synchronous systems and real-time asynchronous systems.
- Explore the various approaches to real-time data including polling, long polling, websockets, and SSE.
- Delve into the role of webhooks, push notifications, real-time databases, and GraphQL in the context of real-time systems.

Introduction to Real-time Systems

Real-time systems provide live, immediate interaction and data updates.

- **Comparison**: Traditional systems rely on synchronous, periodic updates, while real-time systems offer dynamic, asynchronous updates.
- **Shift in Paradigm**: Transitioning from static content delivery to real-time, dynamic user experiences.
- **Real-world Scenario**: A news website updating instantly during significant events, enhancing user engagement and experience.

Activity: Real-time Applications

Think of some applications that would benefit from being implemented as real-time systems or having real-time features.

- List 5 such applications and briefly describe the real-time features they would require.

Solution: Real-time Applications

These applications demand real-time features to enhance user experience, ensure accurate data representation, and facilitate instant decision-making.

1. **Traffic Navigation App**: Real-time traffic updates, accident notifications, and route recalculations based on current road conditions.
2. **E-commerce Platform**: Real-time inventory updates, live customer support chat, and flash sale notifications.
3. **Health Monitoring System**: Continuous tracking and alerting based on patient vitals, real-time data visualization.
4. **Online Auction Platform**: Live bid updates, countdown timers, and instant notifications when outbid.
5. **Remote Team Collaboration Tool**: Real-time document editing, live video conferencing, instant messaging, and task status updates.

Approaches to Real-time Data

- **Historical Methods**: Polling and long polling were initial solutions to check for data updates.
- **Modern Techniques**: Websockets and server-sent events revolutionized real-time interactions.
- **Pub/Sub Techniques**: Webhooks, push notifications, real-time databases, and GraphQL/tRPC subscriptions, etc.
- **Evolving Needs**: As applications became more interactive, the demand for instant data updates grew, necessitating newer techniques.

Polling: An Initial Approach

- **Basic Concept**: Periodically check the server for data updates at fixed intervals.

```
setInterval(() => {
  fetch('/checkUpdates').then(response => {
    if (response.newData) {
      updateUI(response.data);
    }
  });
}, 300000);  // Check every 5 minutes
```

- **Inefficiencies**: Unnecessary requests when no updates, potential delays in reflecting new data.

- **Scenario**: A dashboard that refreshes data every 5 minutes to check for new updates.

Long Polling: A Refined Approach

- **Distinguishing Factor**: Client requests data, server holds the request until new data is available.
- **Efficiency**: Reduces unnecessary requests; server responds only when there’s new data.
- **Advantage over Basic Polling**: Less network overhead and quicker data updates without frequent queries.
- **Scenario**: A chat application where the client waits for new messages from other users without repeatedly querying the server.

Client-side (JavaScript):

```javascript
function longPoll() {
  fetch('/waitForMessages').then(response => {
    if (response.newMessage) {
      displayMessage(response.message);
    }
    longPoll();  // Recurse to establish the next long poll
  });
}
longPoll();  // Initiate long polling
```

Server-side (ExpressJS):

```javascript
let messages = [];
app.get('/waitForMessages', (req, res) => {
  if (messages.length) {
    res.json({ newMessage: true, message: messages.pop() });
  } else {
    // Wait for 10 seconds before responding
    setTimeout(() => res.json({ newMessage: false }), 10000);
  }
});
```

Websockets: Bi-directional Communication

Websockets enable full-duplex communication channels over a single, long-lived connection.

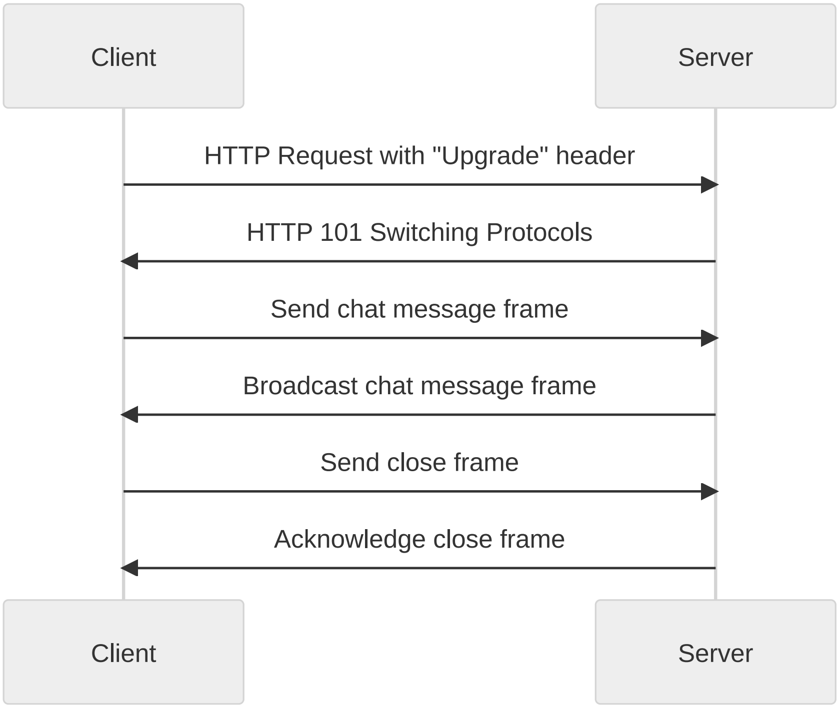
- **Difference from HTTP**: Websockets maintain a persistent connection, allowing for real-time data flow in both directions.
- **Advantages**: Minimized latency, reduced overhead, and immediate data updates.

Websockets vs. HTTP

- **Protocol**:
  - Websockets: `` `ws://` `` or `` `wss://` `` (secure)
  - HTTP: `` `http://` `` or `` `https://` ``
- **Connection**:
  - Websockets: Continuous two-way communication
  - HTTP: New connection for each interaction (unless using keep-alive)
- **Overhead**:
  - Websockets: Minimal after initial handshake
  - HTTP: Repeated header metadata for each request

Websockets: Life Cycle (Chat Application)

- **Handshake:** Upgrade from HTTP to the Websockets protocol.
- **Data Frames:** Continuous data exchange in the form of frames.
- **Closing:** Either end can initiate the termination of the connection.



Websockets: Use Cases

Websockets are particularly valuable in applications that require instantaneous data exchange, like chats, live score updates, or stock trading platforms.

- **Scenario**: A stock trading platform where price fluctuations occur in milliseconds, requiring instant updates to traders.

Websockets: Libraries and Frameworks

Many libraries and frameworks exist for the development of real-time web applications with Websockets.

- **Socket.io:** Simplifies real-time web app development.
- **WebSockets API:** Native browser support for Websockets.
- **Others:** Pusher, SignalR, and ActionCable offer more options.

Server-side code with Express.js

```
const app = express();
const server = http.createServer(app);
const io = socketIo(server);

io.on('connection', (socket) => {
    console.log('a user connected');

    // Handle chat messages
    socket.on('chat message', (msg) => {
        io.emit('chat message', msg);
    });

    socket.on('disconnect', () => {
        console.log('user disconnected');
    });
});

server.listen(3000, () => console.log('server running!'));
```

Client-side (Vanilla JavaScript)

```javascript
const socket = io('http://localhost:3000');

// Listening for chat messages from the server
socket.on('chat message', (msg) => {
    console.log('Message received:', msg);
});

// Sending a chat message to the server
function sendMessage(text) {
    socket.emit('chat message', text);
}
```

Websockets: Key Points & Challenges

- Revolutionized real-time web communication.
- Persistent, full-duplex channel over a single connection.
- Suited for low overhead, high-performance scenarios.
- **Challenges**:
  - **Reconnection**: Managing dropped connections.
  - **Scaling**: Complexity increases with user growth.
  - **Security**: Encryption using `` `wss://` ``.
  - **Solutions**: Libraries like Socket.io abstract away many challenges.

Server-Sent Events (SSE)

A mechanism enabling servers to push information to web clients over a standard HTTP connection.

- **Unidirectional Nature**: Specifically designed for server-to-client updates without the necessity for the client to send data back.
- **Protocol Usage**: Operates over standard `` `http://` `` or `` `https://` ``, leveraging the existing web infrastructure.

SSE vs. Websockets

- **Communication Direction**:
  - SSE: Unidirectional (server-to-client)
  - Websockets: Bidirectional (server-to-client and client-to-server)
- **Protocol**:
  - SSE: Standard HTTP (`` `http://` `` or `` `https://` ``)
  - Websockets: Specialized (`` `ws://` `` or `` `wss://` ``)
- **Overhead**:
  - SSE: Minimal, especially after the initial connection
  - Websockets: Low overhead post-handshake, but requires a separate protocol setup

Applications of SSE

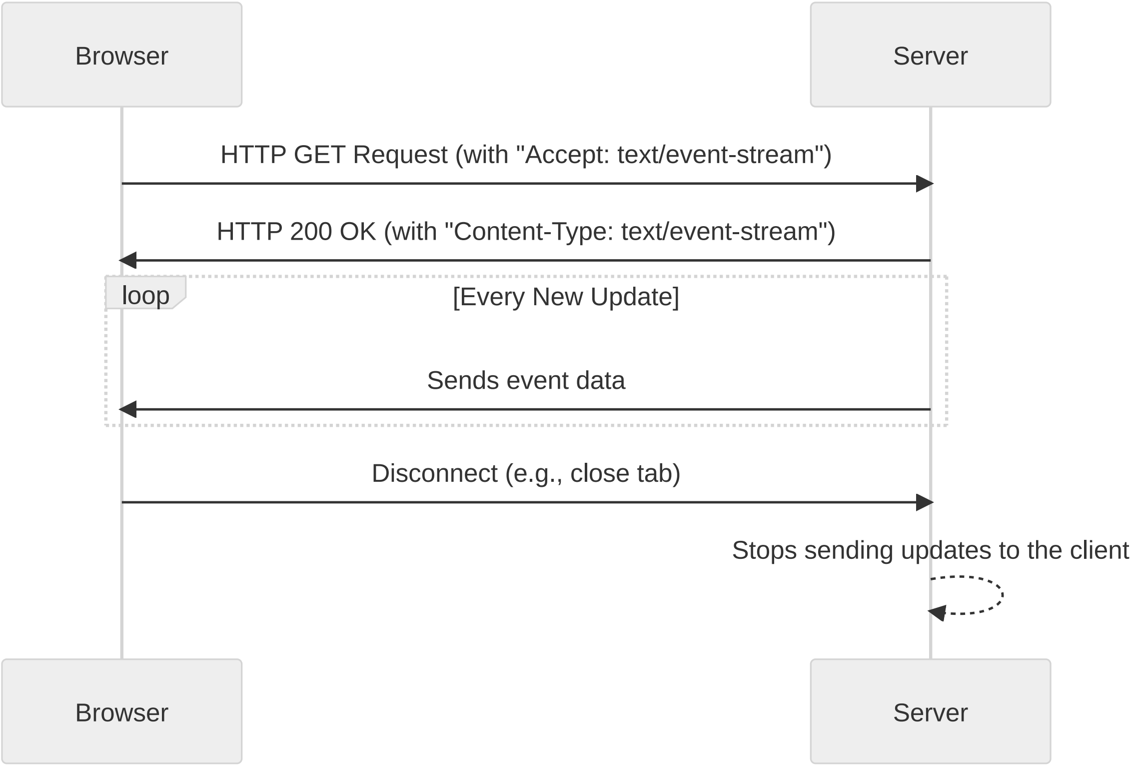
- **Live Blogs**: Continuous updates for events like news happenings, sports games, or award shows.
- **Real-time Notifications**: Dashboard updates, stock price changes, or system alerts.
- **Monitoring**: Systems where the client continuously receives updates, such as server health or environmental sensor data.

In comparison, Websockets are used in real-time applications requiring **two-way communication** (e.g., chat apps, online gaming, collaborative tools such as whiteboarding, etc.).

SSE Life Cycle

- **1. Connection Initiation**: Browser sends an HTTP GET request indicating it wants to receive Server-Sent Events.
- **2. Server Acknowledgment**: Server responds, confirming the setup and the intention to send SSE.
- **3. Continuous Updates**: Server sends event data to the browser whenever there’s a new update.
- **4. Disconnection**: If the client (browser) disconnects, the server stops sending updates.

SSE Life Cycle: Sequence Diagram



Server (ExpressJS)

```javascript
app.get('/events', (req, res) => {
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');
  res.setHeader('Connection', 'keep-alive');

  // Send a message every second
  setInterval(() => {
    res.write(`data: ${new Date().toISOString()}\n\n`);
  }, 1000);
});
```

Client (Vanilla JS using Browser API):

```javascript
const eventSource = new EventSource('/events');

eventSource.onmessage = (event) => {
  console.log('New message:', event.data);
};
```

Challenges and Considerations with SSE

- **Unidirectional Limitation**: SSE only supports server-to-client communication, not the other way around.
- **Connection Constraints**: Servers might struggle with a large number of open SSE connections. How many can it effectively handle?
- **Reconnection**: Browsers generally attempt to reconnect automatically if an SSE connection is lost. But how often and when should reconnection attempts be made?
- **Browser Support**: Not all browsers support SSE. How do you handle compatibility issues or fallback mechanisms?

Activity: Websockets vs. SSE

- Identify the benefits and potential challenges of using websockets over SSE for a stock trading platform.

Solution: Websockets vs. SSE

In stock trading, real-time bid updates and instant trade executions are crucial, making websockets suitable despite potential challenges.

- **Benefits**:

  1. **Two-Way Communication**: Allows traders to send trade orders while receiving market updates.
  2. **Real-time Interactivity**: Instant execution of trades reflecting real-time market conditions.
  3. **Persistent Connection**: Continuous data flow without needing to reestablish connections.

- **Challenges**:

  1. **Complexity**: Websockets can be more complex due to two-way communication.
  2. **Scaling**: Managing numerous simultaneous websocket connections can be challenging.
  3. **Overhead**: Websockets can introduce overhead, especially with many users.

Real-time with Pub/Sub Pattern

There are a few other ways to achieve real-time data flow which fall under the Pub/Sub pattern. This pattern separates concerns between publishers (who emit events) and subscribers (who listen to them).

Understanding the Pub/Sub Pattern

The Pub/Sub pattern separates concerns between publishers (who emit events) and subscribers (who listen to them):

- **Publisher**: Emits events or messages.
- **Subscriber**: Listens for specific events or messages.
- **Delivery Mechanism (e.g., Event Bus)**: Facilitates communication between publishers and subscribers, ensuring the right events are delivered.

Pub/Sub Pattern: Real-world Analogy

Consider a real-world analogy of email subscriptions to news:

- **Publisher**: The news organization that creates and sends out news updates via email.
- **Subscribers**: Individuals who subscribe to the news organization’s email updates.
- **Delivery Mechanism**: The email delivery system that ensures the news updates are delivered to the subscribers’ inboxes.

Pub/Sub vs. Polling

- **Traditional Polling**:
  - Clients periodically check (poll) the server for updates.
  - Inefficient: Consumes resources even when there are no updates.
  - Latency: Delays due to fixed poll intervals.
- **Pub/Sub**:
  - Server (publisher) notifies the client (subscriber) immediately when an event occurs.
  - Efficient: No unnecessary resource consumption.
  - Reduced Latency: Immediate notifications reduce latency.

Pub/Sub Technologies

We will cover the following Pub/Sub technologies in this lecture:

- **GraphQL Subscriptions**: Real-time data interactions with GraphQL.
- **RPC APIs**: Real-time capabilities in Remote Procedure Call (RPC) APIs.
- **Real-time Databases**: Databases that support real-time data updates.
- **Webhooks**: HTTP callbacks for event notifications.
- **Push Notifications**: Real-time notifications on mobile devices.

GraphQL Subscriptions

GraphQL is a query language for APIs and runtime for executing those queries that enables clients to request exactly the data they need.

- **Subscriptions in GraphQL**:
  - A real-time mechanism to "listen" to data changes.
  - Allows clients to receive live updates without re-querying.
- **Key Distinction**: Unlike traditional queries (request-response) and mutations (modify data), subscriptions maintain an active connection and push (publish) updates.

Recall: GraphQL Operations

- **Queries** for fetching data; <u>Request-response cycle:</u> Client requests specific data; server responds with matching data.
- **Mutations** for modifying data (create, update, delete); <u>Request-response cycle:</u> Client sends a request to change data; server responds with the result of the change.
- **Subscriptions** for listening to real-time data changes; <u>Persistent, real-time connection:</u> Client subscribes to specific events; server pushes updates when events occur.

GraphQL Subscriptions: Pub/Sub Pattern

GraphQL’s real-time functionality is enabled through the Pub/Sub pattern.

- **Publisher**: The server, acting as the publisher, emits the event.
- **Subscriber**: Clients with active subscriptions for that event (the subscribers) receive the associated data in real-time.
- **Delivery Mechanism**: Under the hood, GraphQL uses Websockets. Some implementations may use SSE.

GraphQL Subscriptions: Delivery Mechanism

Under the hood, GraphQL uses Websockets. Some implementations may use SSE.

- **WebSockets**: The primary transport mechanism for GraphQL subscriptions, offering persistent two-way communication.
- **HTTP2/Server-Sent Events (SSE)**: Some implementations may use these as alternative methods, but WebSockets are the most common choice.

GraphQL Subscriptions: Code Example

```
// 1. Set up a PubSub instance for your GraphQL server
const { PubSub } =  require('graphql-subscriptions');
const pubsub = new PubSub();

// 2. Define a subscription type in your schema
const typeDefs = `
  type Subscription {
    postAdded: Post
  }
`;
```

GraphQL Subscriptions: Code Example

```
// 3. Resolve the subscription
const resolvers = {
  Subscription: {
    postAdded: {
      subscribe: () =>
        pubsub.asyncIterator(['POST_ADDED'])
    },
  },
};

// 4. When a post is added elsewhere in your app
pubsub.publish('POST_ADDED', {
  postAdded: { message: "Hello World" }
});
```

```
// 5. Set up a subscription on the client side
import { gql, useSubscription } from '@apollo/client';

const SUBSCRIBE_TO_NEW_POSTS = gql`
  subscription OnPostAdded {
    postAdded {
      message
    }
  }
`;

function NewPostNotification() {
  const { data, loading, error } = useSubscription(SUBSCRIBE_TO_NEW_POSTS);

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error :(</p>;

  return <div>New post: {data.postAdded.message}</div>;
}
```

Activity: Real-time Social Media Feed

Imagine you are building a social media application similar to Facebook or Twitter:

- **Objective**: Users should see real-time updates in their feed when their friends post new statuses.
- **Your Task**: How would you implement this feature using GraphQL so that the feed updates in real-time without manual refresh?

Solution: Real-time Social Media Feed

- **Server Preparation:** The GraphQL server should be configured to handle subscriptions, and define a `newStatus` subscription type.
- **Subscription Setup**: Users initiate a subscription to `newStatus` events, specifically targeting updates from their friends' list.
- **Event Emission**: When a friend publishes a new post, the backend service triggers a `newStatus` event. This event carries the new post's data, ensuring that only relevant updates are transmitted.
- **Real-time Delivery**: The GraphQL server efficiently dispatches the update to all active subscribers. Each user's feed dynamically refreshes, displaying the latest post without any need for manual reloading.

Real-time Capabilities in RPC APIs

Originally designed for request-response communication. Client sends a procedure call request, and the server sends back a response.

- Newer RPC frameworks have adapted to support real-time communication. They extend the traditional model with features like subscriptions.
- **tRPC**: Provides a subscription model for real-time updates. It uses **WebSockets** under the hood for real-time communication.
- **gRPC**: Supports bi-directional streaming. It employs **HTTP/2** for transport, which inherently supports streaming, making it suitable for real-time.

Real-time Databases

A database designed to process requests and deliver data in real-time, allowing multiple users to read and write simultaneously.

- **Instant Updates**: Changes in the database are immediately pushed to all subscribed clients.
- **Event-driven**: Operations are executed based on events rather than static queries.
- **Data Synchronization**: Consistent and coordinated data across multiple devices or clients.

Subscription Mechanism in Real-time DB

- **Listen to Changes**: Clients can "subscribe" to specific data points or collections in the database.
- **Automatic Updates**: When the subscribed data changes, updates are automatically pushed to the client without the need for polling.
- **Event-based Interaction**:
  - Clients listen for specific events (e.g., data addition, update, deletion).
  - Actions or callbacks are executed in response to these events.

Popular Real-time Databases

- **Google's Firebase Realtime Database**:
  - NoSQL cloud-hosted database.
  - Data is stored as JSON.
  - Provides SDKs for various platforms to simplify integration.
- **RethinkDB**:
  - Open-source database.
  - Supports JSON documents.
  - Real-time change feeds to push updates to apps.
- **Others**: Apache Kafka, AWS Kinesis, TinyBird, ClickHouse, etc.

Supabase Database Code Example

```
// Initialize Supabase (after adding SDK and configuration)
const supabase = createClient(supabaseUrl, supabaseKey);

// Reference to the data we want to listen to
const myDataTable = 'my_data';

// Subscribing to changes in data
const myDataSubscription = supabase
  .from(myDataTable)
  .on('INSERT', (payload) => {
    // Update UI or perform actions based on data changes
    console.log('New data:', payload.new);
  })
  .subscribe();

// Make sure to handle the error and data as needed
```

Activity: Collaborative Document Editor

Consider creating a platform like Google Docs where multiple users can collaboratively edit a document in real-time.

- **Your Task**: How would you use a real-time database to ensure that changes made by one user are instantly visible to others?

Solution: Collaborative Document Editor

- **Real-time Synchronization**:

  1. Users A and B open the same document.
  2. As User A types, changes are pushed to the real-time database.
  3. User B’s app "listens" to these changes and updates the document view in real-time, and vice-versa.

- **Addressing Challenges**: Just having a real-time database will not suffice. You need to address challenges such as concurrent edits, version history, and low latency. More on that in future lessons.

Activity: Real-time Q&A Decision Making

Imagine a platform where users can post questions and others can answer in real-time, similar to a live Q&A session:

- Instantaneous display of new questions and answers.
- Scalability to accommodate many users and interactions.
- Efficient querying for retrieving past Q&A sessions.

Given the above, justify your choice between:

1. Using a real-time database.
2. Implementing GraphQL subscriptions.
3. Opting for another method.

Solution: Real-time Q&A Decision Making

- **Real-time Database**:
  - **Pros:** Instantaneous updates, simpler setup for real-time features.
  - **Cons:** Might lack advanced querying capabilities.
- **GraphQL Subscriptions**:
  - **Pros:** Provides the flexibility of GraphQL, allowing for complex querying and integration with other services.
  - **Cons:** Might be overkill if only simple real-time features are needed.
- **Other Methods**: Polling might not offer the instantaneous updates required.

Introduction to Webhooks

- **Definition**: Webhooks are "user-defined HTTP callbacks".
- **Basic Explanation**: They allow external systems to notify you when an event occurs.
- **Advantages**:
  - Real-time notifications without continuous polling.
  - Efficient use of server resources.
  - Enables responsive, event-driven architectures.

Webhooks in Real-time Systems

- **Instant Notifications**: Instead of polling, get notified the moment an event occurs.
- **Third-party Integrations**: Seamlessly integrate with other systems, services, and platforms.
- **Automation**: Trigger workflows or processes in other systems automatically.
- **Custom Responses**: Define custom logic to execute when the webhook is triggered.
- **Examples**:
  - E-commerce: Notify when an order is placed.
  - CRM systems: Update when a customer’s details change.
  - Monitoring: Send alerts based on system health metrics.

Server-side (ExpressJS):

```javascript
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.json());

// Endpoint to receive webhooks
app.post('/webhook-endpoint', (req, res) => {
    console.log('Received Webhook:', req.body);
    // Do something with the webhook data, e.g., store it in a database
    res.status(200).send('Webhook received!');
});

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Scenario: GitHub and Slack Integration

Imagine you're managing a software development project on GitHub and want to keep your team updated on project activities through Slack.

- **User Interaction**:
  - A developer commits code to a GitHub repository.
  - The team wants to receive notifications in a Slack channel for every commit.
- **Behind the Scenes**:
  1. The GitHub repository is configured with a webhook pointing to a Slack app.
  2. When a commit is pushed to the repository, GitHub triggers the webhook.
  3. The webhook sends a payload to the Slack app, which then formats a message and posts it to the specified Slack channel.

Push Notifications

Push notifications are messages that pop up on a user’s mobile or web device.

- **User Engagement**: Enhance user experience by providing timely updates, reminders, and alerts.
- **Relevance**: Deliver tailored content directly to users, increasing the chance of immediate interaction.
- **Platforms**: Ubiquitous across both web and mobile platforms, becoming a vital tool for real-time information delivery.

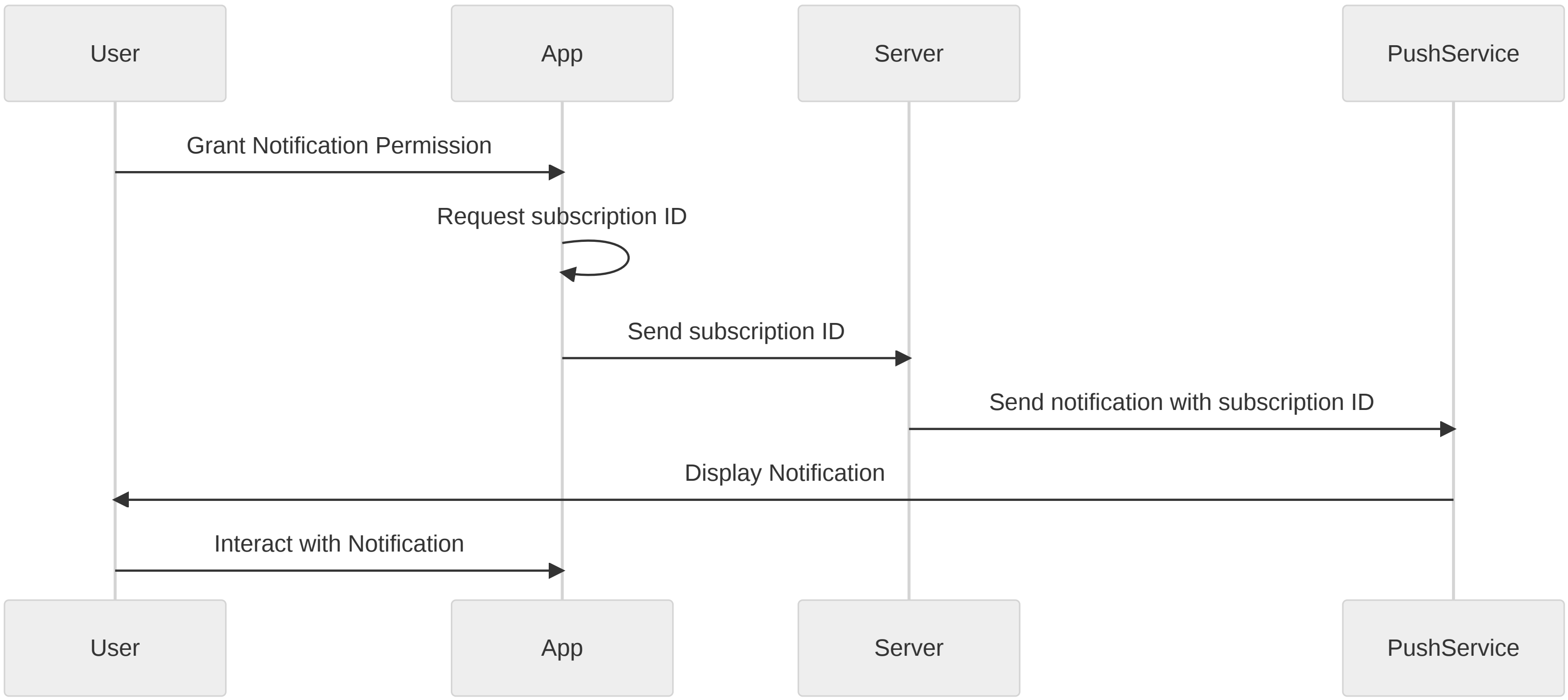
Platforms for Push Notifications

- **Firebase Cloud Messaging (FCM)**: Google’s cloud solution for messages on iOS, Android, and web applications.
- **Apple Push Notification Service (APNS)**: Apple’s platform for sending notifications to iOS devices.
- **Web Push APIs**: Allows sending notifications to users’ browsers.
- **Benefits**:
  - **Scalability**: Handle millions of notifications efficiently.
  - **Flexibility**: Customize notifications based on user preferences and behavior.
  - **Reliability**: Ensure delivery even if the user’s device is offline.

Push Notifications Flow

1. **Subscription**:
   - The application requests permission from the user.
   - On approval, the application receives a unique subscription ID.
2. **Storing Subscription Details**:
   - The application sends the subscription ID to the server.
   - The server stores this ID for future communication.
3. **Sending a Notification**:
   - The server sends a message to the push service (like FCM or APNS) with the subscription ID.
4. **Delivering to the Device**:
   - The push service communicates with the target device.
   - The notification is displayed even if the application isn’t actively running.
5. **User Interaction**:
   - The user can click on the notification to launch the application or perform a specific action.

Push Notifications Flow



Subscribing a User

```
Notification.requestPermission().then(function(permission) {
  if (permission === "granted") { // Ask user for permission
    navigator.serviceWorker.ready.then(function(registration) {
      registration.pushManager.subscribe({
        userVisibleOnly: true
      }).then(function(subscription) {
        // Send subscription object to server
      });
    });
  }
});
```

Receiving a Notification

```
self.addEventListener('push', function(event) {
  var options = {
    body: event.data.text(),
    icon: 'icon.png',
  };

  event.waitUntil(
    self.registration.showNotification('Push Notification', options)
  );
});
```

Scenario: To-Do List App with Reminders

Imagine a web-based to-do list application that allows users to set tasks with specific deadlines. To enhance user experience, the app sends push notifications as reminders before a task's deadline.

- **Task Creation**: Users can create tasks, set deadlines, and choose when they want to be reminded (e.g., 10 minutes, 1 hour, or 1 day before the deadline).
- **Notification Subscription**: Upon registration, users are prompted to allow push notifications.
- **Server-Side Processing**: The server keeps track of tasks and their deadlines. When it's time to send a reminder, a push notification is dispatched to the relevant user.
- **Notification Interaction**: When users receive a reminder, they can mark the task as done, snooze the reminder, or open the app to view details.

Conclusion

- Explored the essence and applications of real-time systems in the digital age.
- Differentiated between traditional synchronous communication and dynamic real-time methods.
- Investigated various mechanisms like polling, websockets, SSE, and the role of webhooks in real-time updates.
- Delved into real-time databases, push notifications, and the evolving role of GraphQL in real-time data handling.
- Engaged in practical scenarios and activities, reinforcing the application of real-time systems in diverse settings.