Media streaming with IBM cloud video streaming

Innovation

Real-Time Messaging

One of the core features we developed was real-time messaging. Users could join chat rooms and instantly send messages that were synchronized across all participants. Leveraging the power of Socket.IO, we ensured efficient and reliable communication, allowing users to engage in dynamic conversations.

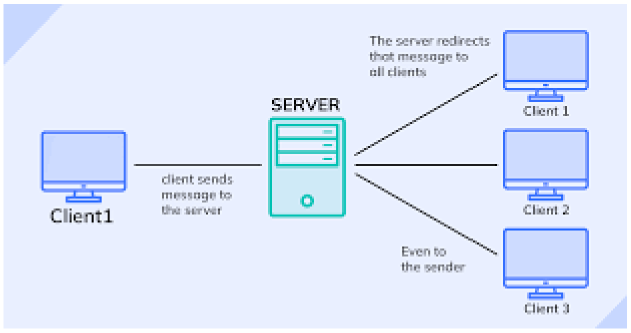
User Authentication and Registration

To enhance security and privacy, we implemented user authentication and registration functionality. Users could create accounts, securely log in, and enjoy personalized chat experiences. We integrated server-side authentication logic, password hashing for secure storage, and password reset functionality, ensuring a robust authentication system.

```
<!DOCTYPE html

<html lang="en">




<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <link rel="icon" href="favicon.png">

    <title>Chat App</title>

    <link rel="stylesheet" href="styles.css">
```

</head>

Architectural Overview


The server redirects that message to all clients
SERVER
Client 1
Client1
client sends message to the server
Client 2
Even to the sender
Client 3

The architecture of our Chat App consists of a client-server model. Clients interact with the frontend interface built with HTML, CSS, and JavaScript, while the server, powered by Node.js and Express.js, handles the communication and storage of messages and user data. Socket.IO facilitates real-time bidirectional communication between the clients and the server, ensuring instant message updates.

The Most Difficult Technical Challenge

One of the most challenging technical aspects we encountered was ensuring synchronized message updates across multiple clients. To overcome this, we implemented an event-driven communication approach using Socket.IO's event-based architecture. We utilized broadcasting techniques to emit messages to all connected clients within a chat room, ensuring real-time updates for all participants. This required meticulous planning, efficient data synchronization techniques, and a deep understanding of websockets.

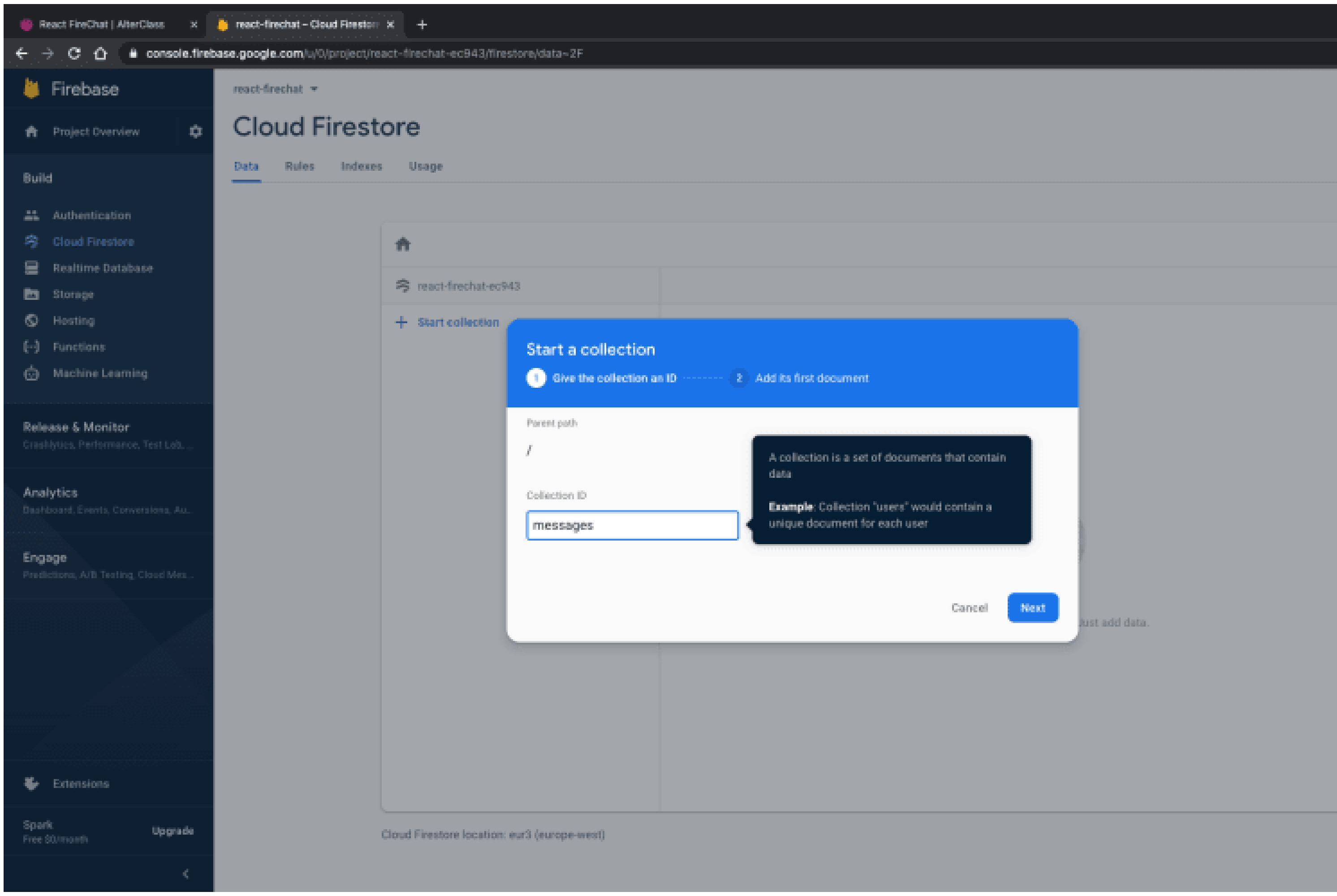
Read data in realtime from Firestore

Now, let's jump into creating the `Channel` component.

This one is responsible for displaying the chat messages and an input field to allow the user to send new messages to the chat.

But first, we need to set up our Firestore database from the Firebase console.

Cloud Firestore stores data within "documents," which are contained within "collections." In our case, we'll store the chat messages of our users within a "Messages" collection.



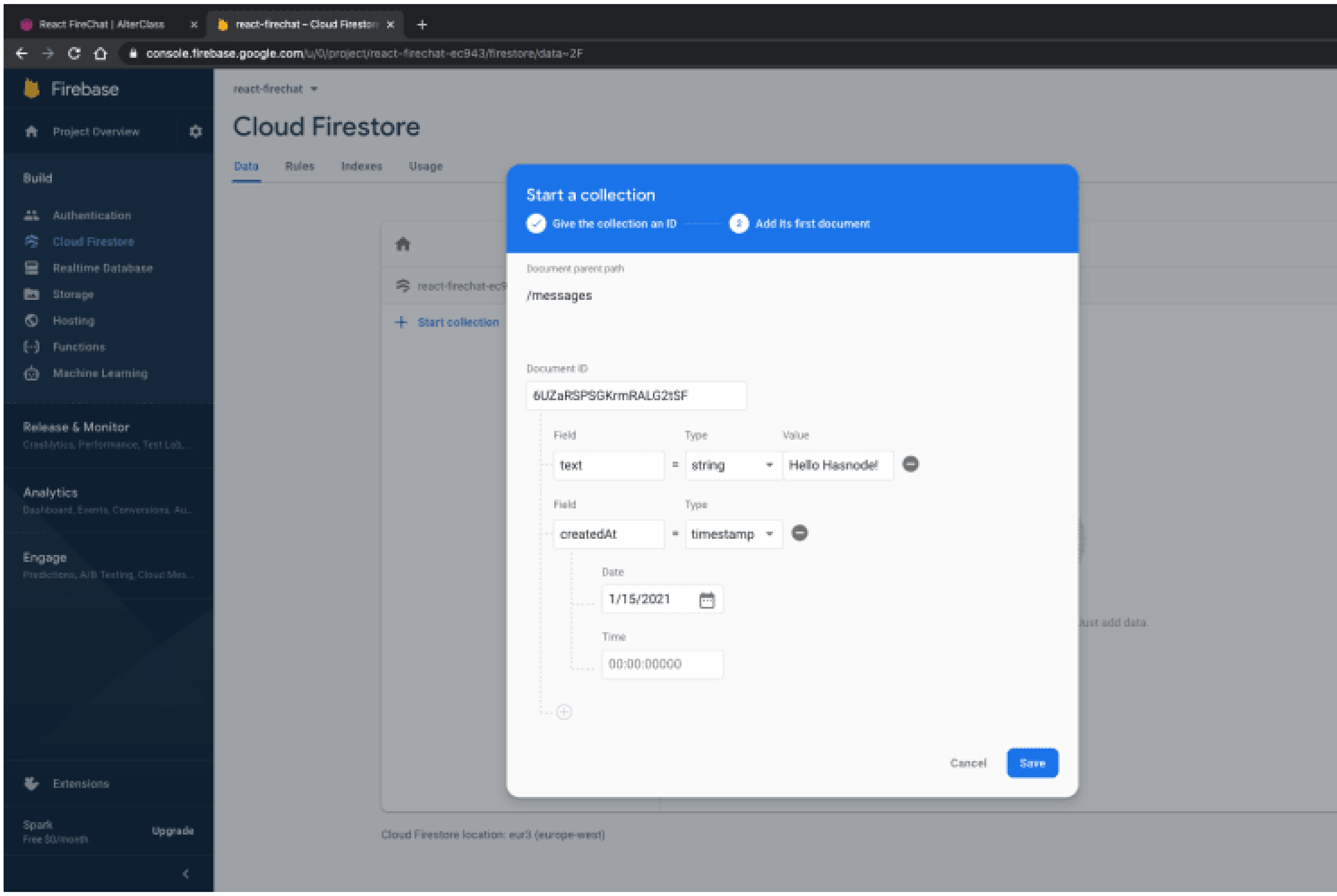
Collection ID: messages

A collection is a set of documents that contain data

Example: Collection "users" would contain a unique document for each user

Let's go ahead and also create our first document within this collection.

For now, we just add two fields to this new document. The text of the message itself and a timestamp representing the date and time when the message has been created.



Document parent path
/messages

Document ID
6UZaRSPSGKrmRALG2tSF

Field	Type	Value
text	string	Hello Hasnode!
createdAt	timestamp	

Date
1/15/2021

Time
00:00:00000