

# Building a Realtime Chat Application Using Angular and Appwrite 🧐

Brandon Roberts

[Appwrite](#) is an open-source backend-as-a-service that provides developers with a core set of functionality needed to build any application with any stack. From database interactions to authentication, real-time updates, and more.

When building web applications with Angular, it's common practice to connect to different APIs to manage data, authenticate users, and possibly listen to live updates to data. The APIs to connect to these different services could be done through multiple providers. With Appwrite, you can do all of these things using a single backend. This post shows you how to get up and running with Appwrite, authenticate users, manage data, and listen to realtime events using a chat application.

## Prerequisites

To get started with Appwrite, you need to have Docker installed on your local machine or server. After you have Docker running, use the following command to install and run Appwrite.

```
docker run -it --rm \
  --volume /var/run/docker.sock:/var/run/docker.sock \
  --volume "$(pwd)"/appwrite:/usr/src/code/appwrite:rw \
  --entrypoint="install" \
  appwrite/appwrite:0.15.0
```

❏ ❏ ❏

Also check out the complete [installation guide](#) for more information about the process. If everything went smoothly, you can visit the Appwrite Console and register your root account.

Next, let's set up the first project.

## Creating a Project

You can host many different applications in Appwrite using projects. To create a project:

- Click on Create Project
- Click on the pencil icon and enter **ngchat** as the custom Project ID
- Enter **Angular Chat** as the name
- Click Create

Next, let's setup the database and collection for the chat application.

## Creating a Database and Collection

A database in Appwrite is group of collections for managing data. To create a database,

visit the Database section:

- Click on Create Database
- Enter **chat** as the custom Database ID
- Enter **Chat** as the name
- Click Create

For the collection:

- Click on Create Collection
- Enter **messages** as the custom Collection ID
- Enter **Chat Messages** as the name
- Click Create

We also want to configure permissions for the collection for read/write access. For messages, you'll choose **Document Level** permissions. You can choose more granular permissions depending on your use case. The [permissions page](#) has more details on permissions so the user keeps ownership of their message.

## Creating Collection Attributes

Each collection in an Appwrite database consists of attributes that model the structure for the document you want to store. For the chat application, you'll store the user's name and message.

### Creating Document Attributes

Attributes can be defined as strings, numbers, emails, and more. To create an attribute:

- Click on Create Attribute.
- Select the type of Attribute to create.

Use the table below to create the necessary attributes for chat.

key	size	required	array
user	32	true	false
message	10000	true	false

When a document is created in the collection, it also has extra metadata for the when the document is created and updated, named `$createdAt` and `$updatedAt` respectively. You can use this metadata for querying, syncing, and other use cases.

You can do other things like toggle services, choose which OAuth provider to use and more, but for this chat application, anonymous authentication is used, which is also enabled by default.

Next, let's put the Angular application together.

## Building with Angular

To start, clone an existing repository already running Angular version 14 with a couple of routes setup for login and chat. Use the command below to clone the GitHub

repository.

```
git clone git@github.com:brandonroberts/appwrite-angular-chat.git
```

```
[[[
```

Install the dependencies:

```
yarn
```

```
[[[
```

And start the application to get the development server running

```
yarn start
```

```
[[[
```

Navigate to <http://localhost:4200> in the browser to view the login page.

## Setting up the Appwrite Config

To configure Appwrite in our Angular project, configure some environment variables first for the Appwrite endpoint, project, and collection values.

Update the `src/environments/environment.ts`

```
export const environment = {  
  endpoint: 'http://localhost/v1',  
  projectId: 'ngchat',  
  databaseId: 'chat',  
  chatCollectionId: 'messages',  
  production: false  
};
```

```
[[[
```

After the environment variables are set, move on to setting up the Appwrite Web SDK.

To initialize the Appwrite Web SDK, use the `appwrite` package installed earlier, along with setting up some Injection Tokens in Angular to be able to inject the SDK into services created later.

Let's create 2 tokens, one for the Appwrite Environment variables, and one for the SDK instance itself.

Create a new file named `src/appwrite.ts` and configure the 2 tokens as root providers.

```
import { inject, InjectionToken } from '@angular/core';  
import {  
  Account,  
  Client as Appwrite,  
  Databases  
} from 'appwrite';  
import { environment } from 'src/environments/environment';  
  
interface AppwriteConfig {  
  endpoint: string;  
  projectId: string;  
  databaseId: string;
```

```

    chatCollectionId: string;
  }
}

export const AppwriteEnvironment = new InjectionToken<AppwriteConfig>(
  'Appwrite Config',
  {
    providedIn: 'root',
    factory() {
      const { endpoint, projectId, databaseId, chatCollectionId } = environment;
      return {
        endpoint,
        databaseId,
        projectId,
        chatCollectionId,
      };
    },
  },
);

```

❏ ❏ ❏ ❏

The first token sets up the environment variables so they can be injected to one or more services.

```

export const AppwriteApi = new InjectionToken<{
  database: Databases;
  account: Account;
}>('Appwrite SDK', {
  providedIn: 'root',
  factory() {
    const env = inject(AppwriteEnvironment);
    const appwrite = new Appwrite();
    appwrite.setEndpoint(env.endpoint);
    appwrite.setProject(env.projectId);

    const database = new Databases(appwrite, env.databaseId);
    const account = new Account(appwrite);

    return { database, account };
  },
});

```

❏ ❏ ❏ ❏

The second token creates an instance of the Appwrite Web SDK, sets the endpoint to point to the running Appwrite instance, and the project ID configured earlier.

After the Appwrite SDK is setup, let's create some services for authentication and accessing chat messages.

First, let's create an **src/auth.service.ts** that allows you to login, check auth status, and logout

```

import { inject, Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { Models } from 'appwrite';
import {
  BehaviorSubject,
  concatMap,
  from,
  tap,
  mergeMap
} from 'rxjs';
import { AppwriteApi } from '../appwrite';

```

```

@Injectable({
  providedIn: 'root',
})
export class AuthService {
  private appwriteAPI = inject(AppwriteApi);
  private _user = new BehaviorSubject<Models.User<Models.Preferences> | null>(
    null
  );
  readonly user$ = this._user.asObservable();

  constructor(private router: Router) {}

  login(name: string) {
    const authReq = this.appwriteAPI.account.createAnonymousSession();

    return from(authReq).pipe(
      mergeMap(() => this.appwriteAPI.account.updateName(name)),
      concatMap(() => this.appwriteAPI.account.get()),
      tap((user) => this._user.next(user))
    );
  }

  async isLoggedIn() {
    try {
      const user = await this.appwriteAPI.account.get();
      this._user.next(user);
      return true;
    } catch (e) {
      return false;
    }
  }

  async logout() {
    try {
      await this.appwriteAPI.account.deleteSession('current');
    } catch (e) {
      console.log(`${e}`);
    } finally {
      this.router.navigate(['/']);
      this._user.next(null);
    }
  }
}

```

❏ ❏ ❏

The **AuthService** injects the Appwrite SDK to:

- Authenticate the user with the **login** method, update the name, and store the current user in an observable.
- Checks to see if the user is logged in and returns a boolean
- Logs the user out by clearing the current session

With the Appwrite SDK, all of this is done without using Angular's **HttpClient** service. You can always access Appwrite's REST APIs directly, but it's not required as the SDK handles this for you.

Next, let's create the **src/chat.service.ts** to load and send chat messages.

```

import { inject, Injectable } from '@angular/core';
import { Models, RealtimeResponseEvent } from 'appwrite';
import { BehaviorSubject, take, concatMap, filter } from 'rxjs';

import { AppwriteApi, AppwriteEnvironment } from '../appwrite';
import { AuthService } from '../auth.service';

```

```

export type Message = Models.Document & {
  user: string;
  message: string;
};

@Injectable({
  providedIn: 'root',
})
export class ChatService {
  private appwriteAPI = inject(AppwriteApi);
  private appwriteEnvironment = inject(AppwriteEnvironment);

  private _messages$ = new BehaviorSubject<Message[]>([]);
  readonly messages$ = this._messages$.asObservable();

  constructor(private authService: AuthService) {}

  loadMessages() {
    this.appwriteAPI.database
      .listDocuments<Message>(
        this.appwriteEnvironment.chatCollectionId,
        [],
        100,
        0,
        undefined,
        undefined,
        [],
        ['ASC']
      )
      .then((response) => {
        this._messages$.next(response.documents);
      });
  }

  sendMessage(message: string) {
    return this.authService.user$.pipe(
      filter((user) => !!user),
      take(1),
      concatMap((user) => {
        const data = {
          user: user!.name,
          message,
        };

        return this.appwriteAPI.database.createDocument(this.appwriteEnvironment.chatCol
          'unique()',
          data,
          ['role:all'],
          [`user:${user!.$id}`]
        );
      })
    );
  }
}

```

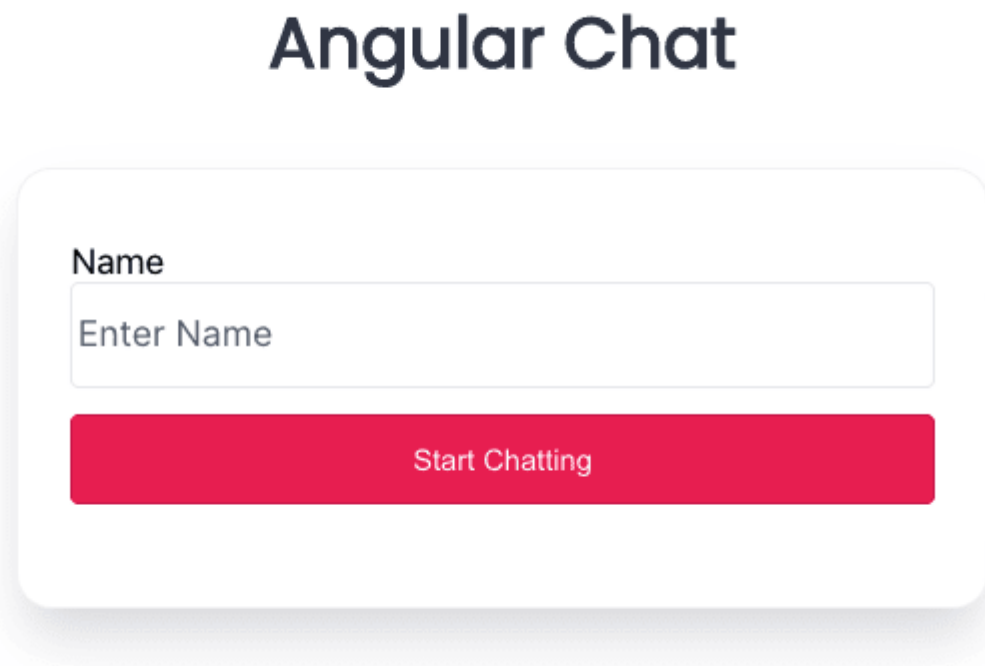
❏ ❏ ❏

### The **ChatService**:

- Injects the Appwrite Environment variables
- Sets up an observable of chat messages
- Uses the Appwrite SDK to load chat messages from the **messages** collection
- Gets the currently logged in user to add chat messages to the **messages** collection.
- Assigns permissions to the document so anyone can read, but only the specific user can update/delete.

With the services set up, we can move on to the components for login and chat.

## Building the Login page



For the login component, use the **AuthService** to login using anonymous authentication with the provided name.

```
import { Component } from '@angular/core';
import {
  FormControl,
  FormGroup,
  ReactiveFormsModule
} from '@angular/forms';
import { Router } from '@angular/router';
import { tap } from 'rxjs';

import { AuthService } from '../auth.service';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [ReactiveFormsModule],
  template: `
    <div class="app-container">
      <div class="content">
        <span class="appwrite-chat">Angular Chat</span>

        <div class="login-container">
          <form [formGroup]="form" class="login-form" (ngSubmit)="login()">
            <p class="login-name">
              <label for="name">Name</label>

              <input
                type="text"
                id="name"
                formControlName="name"
              >
            </p>
          </form>
        </div>
      </div>
    </div>
  `
})
export class LoginComponent {
  form: FormGroup;

  constructor(
    private router: Router,
    private authService: AuthService
  ) {
    this.form = new FormGroup({
      name: new FormControl()
    });
  }

  login() {
    this.authService.login(this.form.get('name').value).subscribe(
      () => {
        this.router.navigate(['/chat']);
      }
    );
  }
}
```

```
        placeholder="Enter Name"
      />
    </p>

    <button type="submit">Start Chatting</button>
  </form>
</div>
</div>
</div>
,
})
export class LoginComponent {
  form = new FormGroup({
    name: new FormControl('', { nullable: true }),
  });

  constructor(
    private authService: AuthService,
    private router: Router
  ) {}

  login() {
    const name = this.form.controls.name.value;

    this.authService
      .login(name)
      .pipe(
        tap(() => {
          this.router.navigate(['/chat']);
        })
      )
      .subscribe();
  }
}
```

❏ ❏ ❏ ❏

After the authentication is successful, we redirect to the chat page.

## Displaying Chat Messages



## Let's Chat

Brandon:  
Hello Angular!

Type a message...

With the Chat component, start with displaying chat messages using the **ChatService**:

```
import { CommonModule } from '@angular/common';
import { Component, OnInit } from '@angular/core';
import {
  FormControl,
  FormGroup,
  ReactiveFormsModule
} from '@angular/forms';
import { tap } from 'rxjs';

import { ChatService } from '../chat.service';
import { AuthService } from '../auth.service';

@Component({
  selector: 'app-chat',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
  template: `
    <div class="chat-container" *ngIf="user$ | async as vm; else loading">
      <div class="chat-header">
        <div class="title">Let's Chat</div>
        <div class="leave" (click)="logout()">Leave Room</div>
      </div>

      <div class="chat-body">
        <div
          id="{{ message.$id }}"
          *ngFor="let message of messages$ | async"
          class="message"
        >

```

```

        <span class="name">{{ message.user }}:</span>
        {{ message.message }}
      </div>
    </div>

    <div class="chat-message">
      <form [formGroup]="form" (ngSubmit)="sendMessage()">
        <input
          type="text"
          FormControlName="message"
          placeholder="Type a message..."
        />
        <button type="submit" class="send-message">
          <svg
            class="arrow"
            width="24"
            height="24"
            viewBox="0 0 24 24"
            fill="none"
            xmlns="http://www.w3.org/2000/svg"
          >
            <path
              d="M13.0737 3.06325C12.8704 2.65671 12.4549 2.3999 12.0004 2.3999C11.5459
              fill="#373B4D"
            />
          </svg>
        </button>
      </form>
    </div>
  </div>

  <ng-template #loading>Loading...</ng-template>
  ,
  styles: [...]
})
export class ChatComponent implements OnInit {
  form = new FormGroup({
    message: new FormControl('', { nullable: true }),
  });
  user$ = this.authService.user$;
  messages$ = this.chatService.messages$;

  constructor(
    private authService: AuthService,
    private chatService: ChatService
  ) {}

  ngOnInit() {
    this.chatService.loadMessages();
  }

  sendMessage() {
    const message = this.form.controls.message.value;

    this.chatService
      .sendMessage(message)
      .pipe(
        tap(() => {
          this.form.reset();
        })
      )
      .subscribe();
  }

  async logout() {
    await this.authService.logout();
  }
}

```



The **ChatComponent** makes use of the **AuthService** and **ChatService** to:

- Use the current logged in user
- Listen to the observable of chat messages
- Load the chat messages in the **ngOnInit** of the component
- Use the input field to send the message using the **ChatService**
- Logout from the chat page

We're able to load chat messages, but let's add the interesting part and integrate some realtime chat messages.

## Connecting to Realtime Events

Appwrite provides realtime updates from practically every event that happens in the Appwrite system, such as database records being inserted, updated or deleted. These events are provided through a WebSocket. To subscribe to realtime, update the **ChatService** with a **listenToMessages** method to subscribe to events from the **messages** collection.

```
export class ChatService {
  ...

  listenToMessages() {
    return this.appwriteAPI.database.client.subscribe(
      `databases.chat.collections.messages.documents`,
      (res: RealtimeResponseEvent<Message>) => {
        if (res.events.includes('databases.chat.collections.messages.documents.*.create'))
          const messages: Message[] = [...this._messages$.value, res.payload];

        this._messages$.next(messages);
      }
    );
  }
}
```



Whenever a new message is created, the new message is pushed into the observable of users so we have realtime updates wired up. To start listening to realtime events:

- Update the **ngOnInit** of the **ChatComponent** to call the method.
- Store the live connection for unsubscribing
- Destroy the live connection when the component is destroyed

```
export class ChatComponent implements OnInit, OnDestroy {
  messageunSubscribe!: () => void;
  form = new FormGroup({
    message: new FormControl('', { nullable: true }),
  });
  user$ = this.authService.user$;
  messages$ = this.chatService.messages$;

  constructor(
    private authService: AuthService,
    private chatService: ChatService
  ) {}

  ngOnInit() {
```

```
    this.chatService.loadMessages();
    this.messageunSubscribe = this.chatService.listenToMessages();
  }

  ngOnDestroy() {
    this.messageunSubscribe();
  }
}
```

👉👉👉

## Summary

And that's it! We now have a functioning Angular application with:

- Authentication
- Database management
- Realtime events

There's more we could do here, but as you can see, you can build just about anything with the core functionality already taken care of. And [cloud functions](#) help you extend the functionality of Appwrite even further.





To view the working example:

<https://appwrite-angular-chat.netlify.app>

GitHub Repo: <https://github.com/brandonroberts/appwrite-angular-chat>

---

## Learn More

-  [Getting Started Tutorial](#)
-  [Appwrite GitHub](#)
-  [Appwrite Docs](#)
-  [Discord Community](#)

If you liked this, click the ❤️ so other people will see it. Follow [Brandon Roberts](#) and [Appwrite](#) on Twitter for more updates!