@push.rocks/smartuniverse

A messaging service enabling secure, reactive communication between microservices.

readme.md for @push.rocks/smartuniverse

messaging service for your micro services

Install

To install `@push.rocks/smartuniverse`, use the following command with npm:

```
npm install @push.rocks/smartuniverse --save
```

This command adds `@push.rocks/smartuniverse` to your project's dependencies.

Usage

`@push.rocks/smartuniverse` is designed to enable messaging services for microservices, allowing them to communicate in a structured and secure manner. Below are examples and scenarios illustrating how to use `@push.rocks/smartuniverse` for both server (managing messages across services) and client (microservices communicating within the universe) aspects.

Server side setup: Creating your Universe

First, you need to set up the server side of your messaging ecosystem, which involves creating a "universe" where channels reside.

```
import { Universe } from '@push.rocks/smartuniverse';

// Initialize a new Universe
const myUniverse = new Universe({
  messageExpiryInMilliseconds: 60000, // messages expire after 60 seconds
});
```

```
// Create channels for communication within the universe
// These channels can be thought of as topics or queues that services can subscribe to or post
messages to
myUniverse.addChannel('channel-one', 'password1');
myUniverse.addChannel('channel-two', 'password2');

// Start the universe server on a specified port
myUniverse.start(8765);
```

By starting the universe, you've established a messaging hub for your microservices. Ensure that the services know the universe's address and the channels and passwords they should use for communication.

Client side: Microservices joining the Universe

On the client side, your microservices will join the universe, subscribing to channels to listen for messages or post their messages to be consumed by other services.

```
import { ClientUniverse, ClientUniverseChannel } from '@push.rocks/smartuniverse';

// Initialize client that connects to the universe server
const clientUniverse = new ClientUniverse({
  serverAddress: 'http://your-universe-server:8765',
  autoReconnect: true,
});

// Define a channel to subscribe to (the channel must be created in the universe server)
const channel = clientUniverse.addChannel('channel-one', 'password1');

// Start the client to enable communication
clientUniverse.start();

// Posting a message to the channel
channel.postMessage({
  messageText: 'Hello, universe!',
  payload: { some: 'data' },
});
```

```
// Listening for messages from the channel
channel.subscribe((message) => {
  console.log('Received message:', message);
});
```

Reaction Patterns: Request and Response Within the Universe

`@push.rocks/smartuniverse` supports reactive programming. Microservices can emit "reaction requests" and listen for "reaction responses" tied to specific actions or commands.

```
import { ReactionRequest, ReactionResponse } from '@push.rocks/smartuniverse';

// Define a request-response type
interface MyRequestResponse {
  method: 'greet';
  request: { name: string };
  response: { message: string };
};

// Creating a reaction request on client side
const reactionRequest = new ReactionRequest<MyRequestResponse>({ method: 'greet' });

// Emitting a reaction request and handling responses
reactionRequest.fire([channel], { name: 'World' }).then((reactionResult) => {
  reactionResult.getFirstResult().then((response) => {
    console.log(response.message); // Output: Hello, World!
  });
});

// Handling reaction requests on server side or another client
const reactionResponse = new ReactionResponse<MyRequestResponse>({
  channels: [myUniverse.getChannel('channel-one')],
  funcDef: async (requestData) => {
    return { message: `Hello, ${requestData.name}!` };
  },
  method: 'greet',
});
```

This pattern enables a powerful, flexible communication system where services can asynchronously request information or trigger actions across the microservices architecture without direct coupling.

Conclusion

`@push.rocks/smartuniverse` provides a robust platform for facilitating communication between microservices. By setting up a universe and defining channels, your services can securely exchange messages, supporting both direct communications and reactive programming patterns. Whether sharing updates, events, or performing request-response interactions, `@push.rocks/smartuniverse` simplifies the process of building a cohesive microservices ecosystem.

For more advanced use cases and configuration options, refer to the complete documentation.

License and Legal Information

Trademarks

Company Information

Task Venture Capital GmbH
Registered at District court Bremen HRB 35230 HB, Germany

For any legal inquiries or if you require further information, please contact us via email at hello@task.vc.

Capital GmbH of any derivative works.