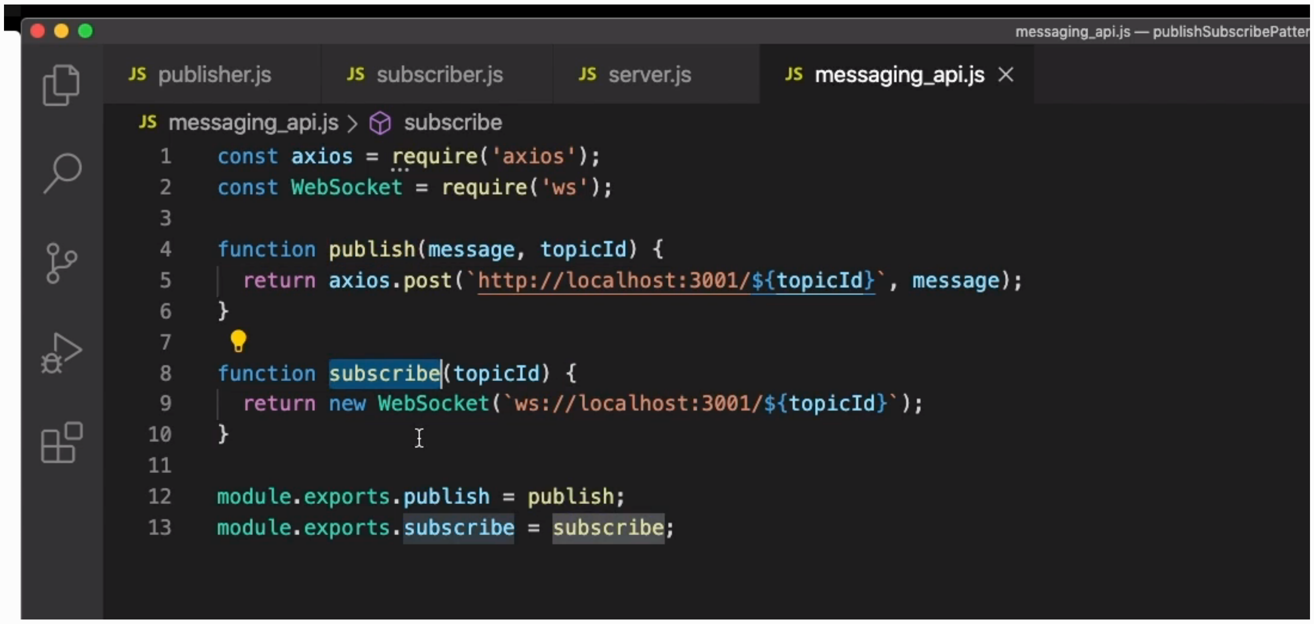
Pub-sub models

Publishers - Servers publishe topics(channels with specific types of data)

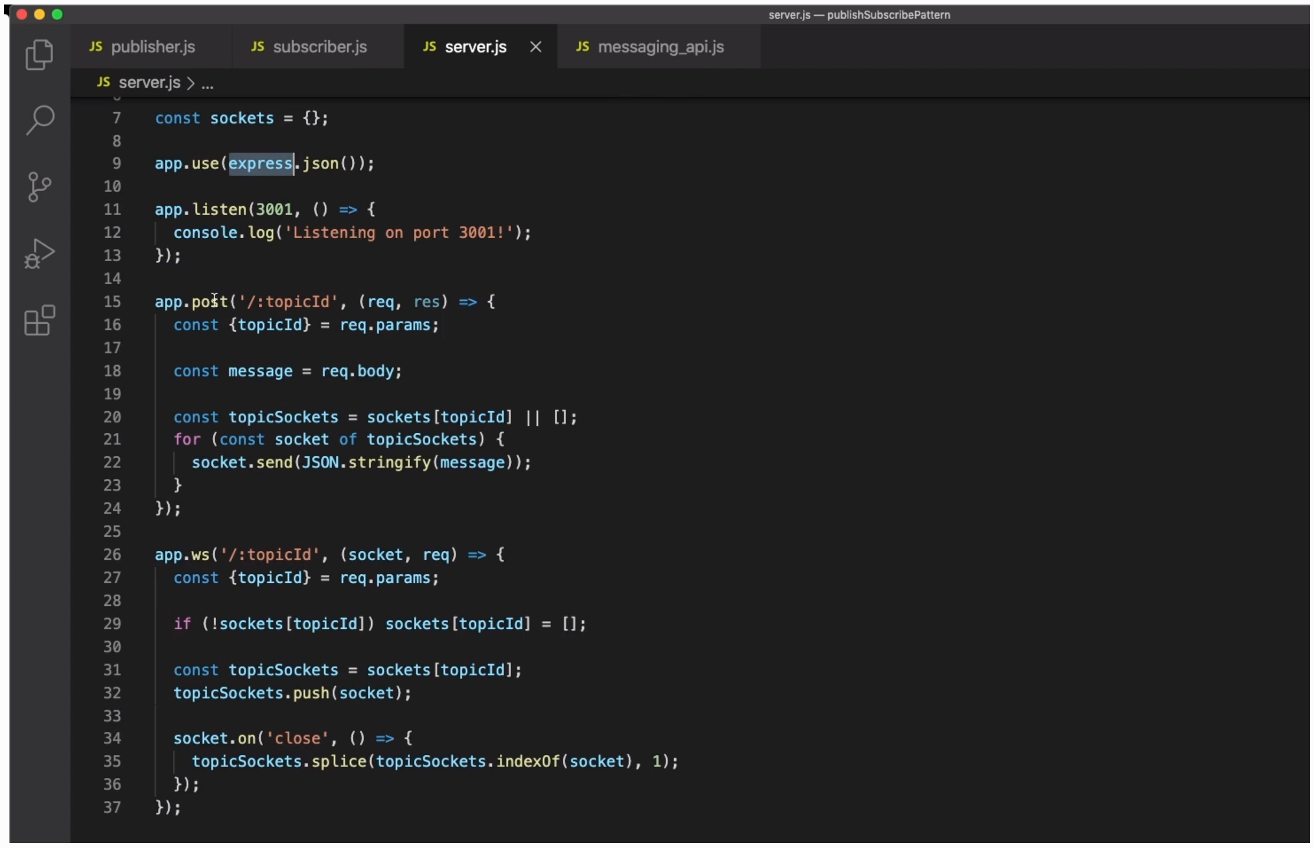
Subscribers - Clients subscribe topics

Topics-

Messages-


messaging_api.js — publishSubscribePatter
publisher.js
subscriber.js
server.js
messaging_api.js
messaging_api.js > subscribe

```javascript
const axios = require('axios');
const WebSocket = require('ws');

function publish(message, topicId) {
  return axios.post(`http://localhost:3001/${topicId}`, message);
}

function subscribe(topicId) {
  return new WebSocket(`ws://localhost:3001/${topicId}`);
}

module.exports.publish = publish;
module.exports.subscribe = subscribe;
```

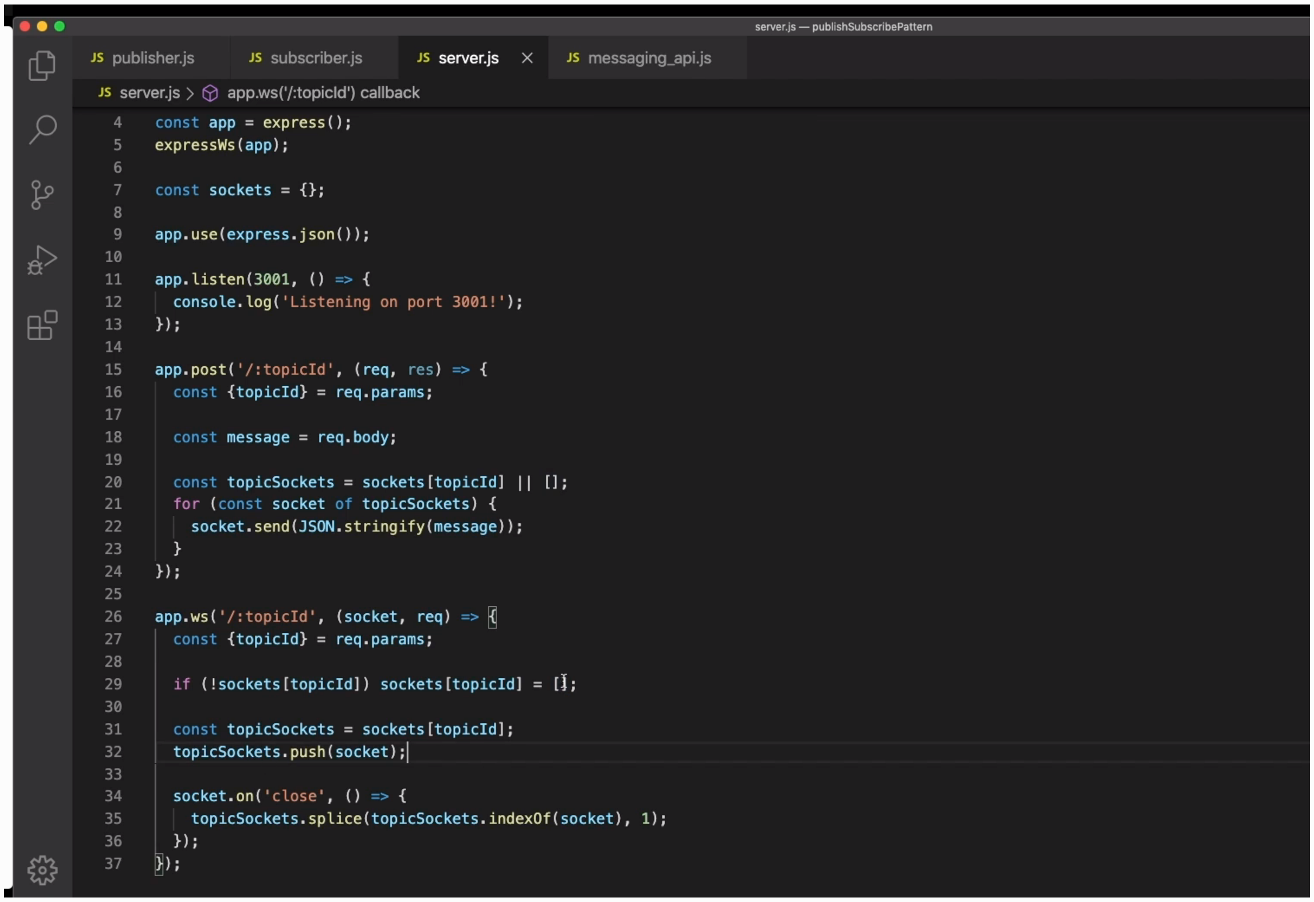


server.js — publishSubscribePattern

publisher.js | subscriber.js | server.js | messaging_api.js

server.js > ...

```
const sockets = {};

app.use(express.json());

app.listen(3001, () => {
  console.log('Listening on port 3001!');
});

app.post('/:topicId', (req, res) => {
  const {topicId} = req.params;

  const message = req.body;

  const topicSockets = sockets[topicId] || [];
  for (const socket of topicSockets) {
    socket.send(JSON.stringify(message));
  }
});

app.ws('/:topicId', (socket, req) => {
  const {topicId} = req.params;

  if (!sockets[topicId]) sockets[topicId] = [];

  const topicSockets = sockets[topicId];
  topicSockets.push(socket);

  socket.on('close', () => {
    topicSockets.splice(topicSockets.indexOf(socket), 1);
  });
});
```

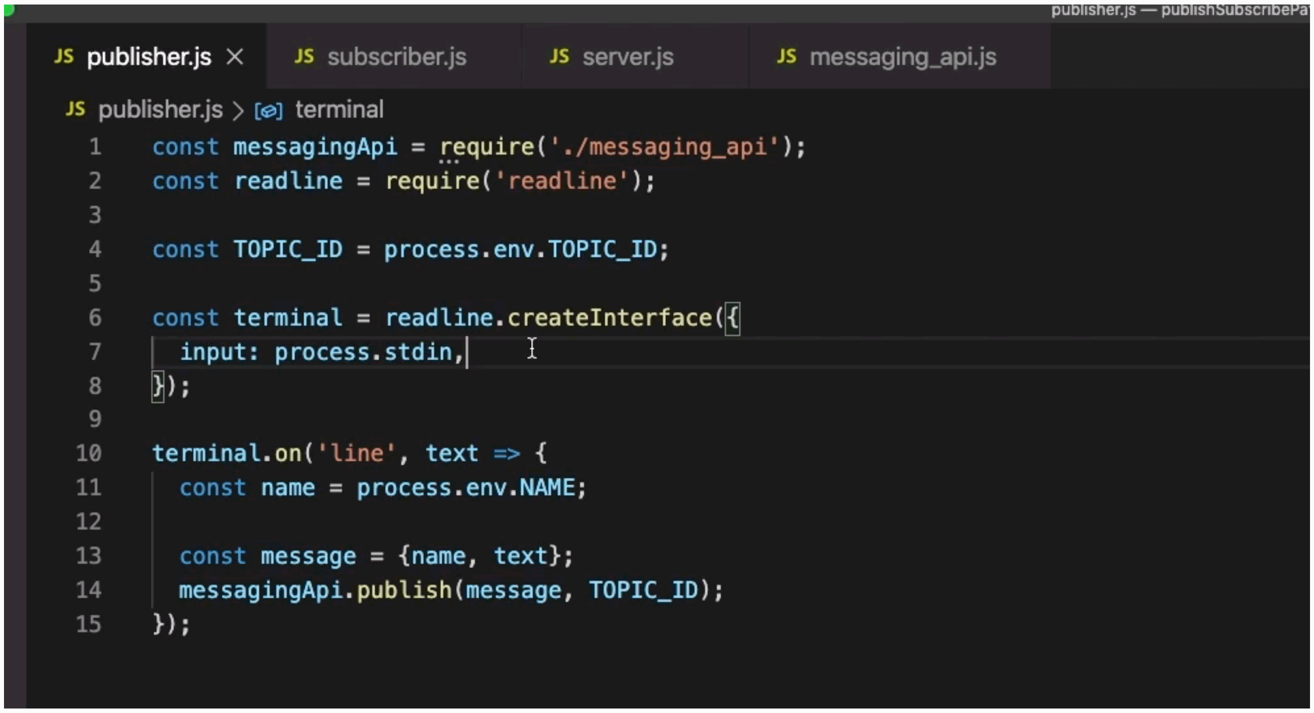


server.js — publishSubscribePattern

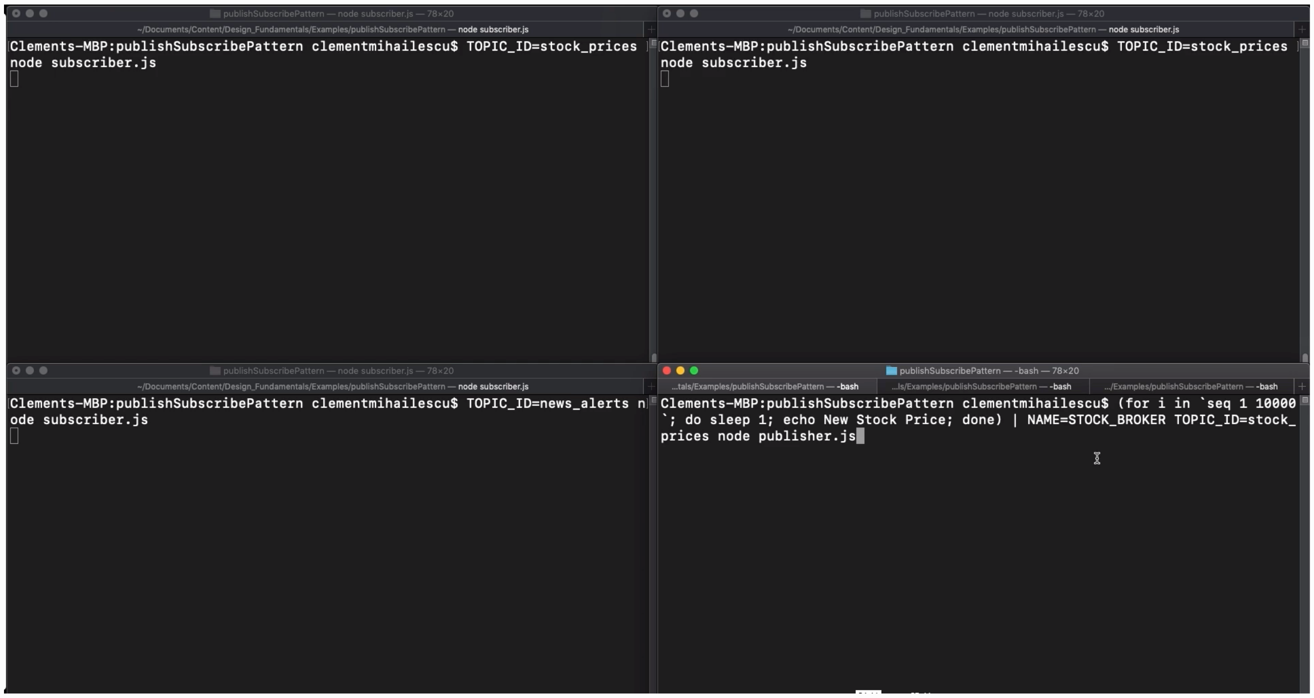
publisher.js | subscriber.js | server.js | messaging_api.js

server.js > app.ws('/:topicId') callback

```
const app = express();
expressWs(app);

const sockets = {};

app.use(express.json());

app.listen(3001, () => {
  console.log('Listening on port 3001!');
});

app.post('/:topicId', (req, res) => {
  const {topicId} = req.params;

  const message = req.body;

  const topicSockets = sockets[topicId] || [];
  for (const socket of topicSockets) {
    socket.send(JSON.stringify(message));
  }
});

app.ws('/:topicId', (socket, req) => {
  const {topicId} = req.params;

  if (!sockets[topicId]) sockets[topicId] = [];

  const topicSockets = sockets[topicId];
  topicSockets.push(socket);

  socket.on('close', () => {
    topicSockets.splice(topicSockets.indexOf(socket), 1);
  });
});
```



```javascript
const messagingApi = require('./messaging_api');
const readline = require('readline');

const TOPIC_ID = process.env.TOPIC_ID;

const terminal = readline.createInterface({
  input: process.stdin,
});

terminal.on('line', text => {
  const name = process.env.NAME;

  const message = {name, text};
  messagingApi.publish(message, TOPIC_ID);
});
```

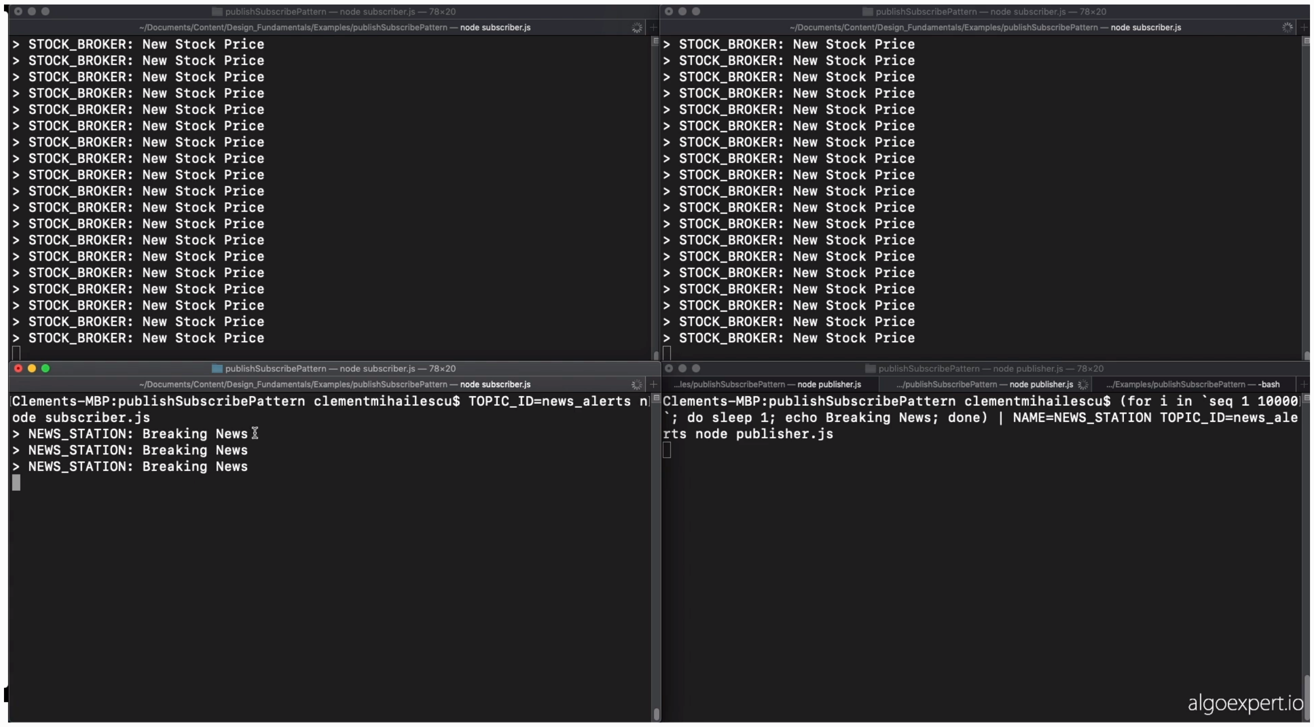

publishSubscribePattern — node subscriber.js — 78×20
~/Documents/Content/Design_Fundamentals/Examples/publishSubscribePattern — node subscriber.js
publishSubscribePattern — node subscriber.js — 78×20
~/Documents/Content/Design_Fundamentals/Examples/publishSubscribePattern — node subscriber.js
publishSubscribePattern — node subscriber.js — 78×20
~/Documents/Content/Design_Fundamentals/Examples/publishSubscribePattern — node subscriber.js
publishSubscribePattern — -bash — 78×20
...tals/Examples/publishSubscribePattern — -bash
...ls/Examples/publishSubscribePattern — -bash
.../Examples/publishSubscribePattern — -bash

```
Clements-MBP:publishSubscribePattern clementmihailescu$ TOPIC_ID=stock_prices
node subscriber.js
```

```
Clements-MBP:publishSubscribePattern clementmihailescu$ TOPIC_ID=stock_prices
node subscriber.js
```

```
Clements-MBP:publishSubscribePattern clementmihailescu$ TOPIC_ID=news_alerts n
ode subscriber.js
```

```
Clements-MBP:publishSubscribePattern clementmihailescu$ (for i in `seq 1 10000
`; do sleep 1; echo New Stock Price; done) | NAME=STOCK_BROKER TOPIC_ID=stock_
prices node publisher.js
```


publishSubscribePattern — node subscriber.js — 78×20
~/Documents/Content/Design_Fundamentals/Examples/publishSubscribePattern — node subscriber.js
publishSubscribePattern — node publisher.js — 78×20
algoexpert.io

```
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
```

```
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
```

```
Clements-MBP:publishSubscribePattern clementmihailescu$ TOPIC_ID=news_alerts n
ode subscriber.js
> NEWS_STATION: Breaking News
> NEWS_STATION: Breaking News
> NEWS_STATION: Breaking News
```

```
Clements-MBP:publishSubscribePattern clementmihailescu$ (for i in `seq 1 10000
`; do sleep 1; echo Breaking News; done) | NAME=NEWS_STATION TOPIC_ID=news_ale
rts node publisher.js
```

3 Prerequisites

Polling
The act of fetching a resource or piece of data regularly at an interval to make sure your data is not too stale.

Streaming
In networking, it usually refers to the act of continuously getting a feed of information from a server by keeping an open connection between the two machines or processes.

Persistent Storage
Usually refers to disk, but in general it is any form of storage that persists if the process in charge of managing it dies.

4 Key Terms

Publish/Subscribe Pattern
Often shortened as **Pub/Sub**, the Publish/Subscribe pattern is a popular messaging model that consists of **publishers** and **subscribers**. Publishers publish messages to special **topics** (sometimes called **channels**) without caring about or even knowing who will read those messages, and subscribers subscribe to topics and read messages coming through those topics.

Pub/Sub systems often come with very powerful guarantees like **at-least-once delivery**, **persistent storage**, **ordering** of messages, and **replayability** of messages.

Idempotent Operation
An operation that has the same ultimate outcome regardless of how many times it's performed. If an operation can be performed multiple times without changing its overall effect, it's idempotent. Operations performed through a **Pub/Sub** messaging system typically have to be idempotent, since Pub/Sub systems tend to allow the same messages to be consumed multiple times.

For example, increasing an integer value in a database is *not* an idempotent operation, since repeating this operation will not have the same effect as if it had been performed only once. Conversely, setting a value to "COMPLETE" *is* an idempotent operation, since repeating this operation will always yield the same result: the value will be "COMPLETE".

Apache Kafka
A distributed messaging system created by LinkedIn. Very useful when using the **streaming** paradigm as opposed to **polling**.

Cloud Pub/Sub
A highly-scalable Pub/Sub messaging service created by Google. Guarantees **at-least-once delivery** of messages and supports "rewinding" in order to reprocess messages.