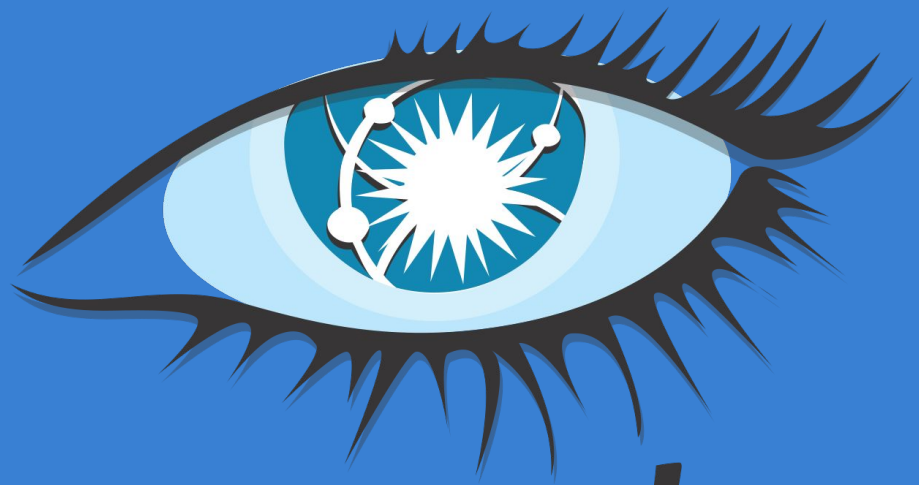


Application Development with the Node.js Drivers

DataStax

Application development

Need a layer between your app and the database

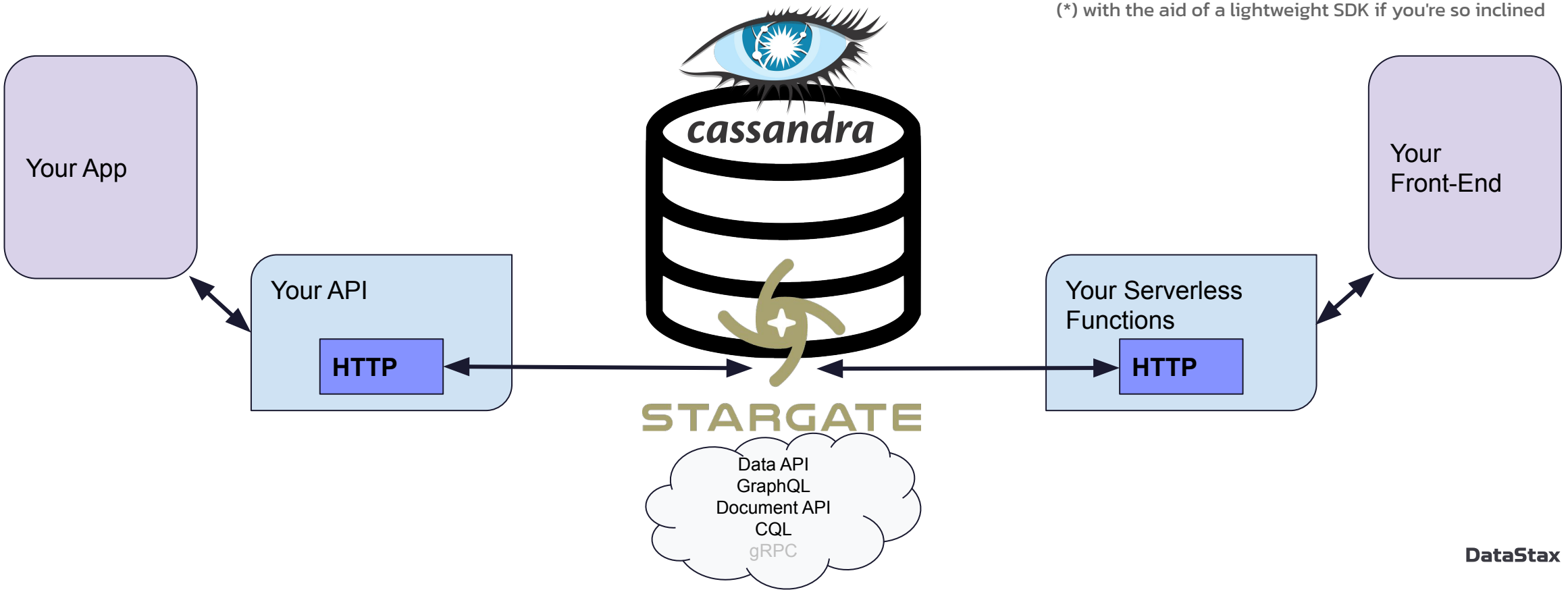
Standard answer: what you need is a **driver**.

Application development

Need a layer between your app and the database

Alternative answer: can place an *API Data Layer* and do simple HTTP requests*

(*) with the aid of a lightweight SDK if you're so inclined



Drivers

Cassandra drivers exist for most popular languages

Connectivity

- ★ Token & Datacenter Aware
- ★ Load Balancing Policies
- ★ Retry Policies
- ★ Reconnection Policies
- ★ Connection Pooling
- ★ Health Checks
- ★ Authentication | Authorization
- ★ SSL

Query

- ★ CQL Support
- ★ Schema Management
- ★ Sync/Async/Reactive API
- ★ Query Builder
- ★ Compression
- ★ Paging

Parsing Results

- ★ Lazy Load
- ★ Object Mapper
- ★ Spring Support
- ★ Paging

Drivers

Cassandra drivers exist for most popular languages

Documentation:
docs.datastax.com/en/developer/nodejs-driver/4.6/

Examples today:
github.com/hemidactylus/cassandra-nodejs-drivers-practice

Installing

Same drivers for: Cassandra, DSE, Astra DB

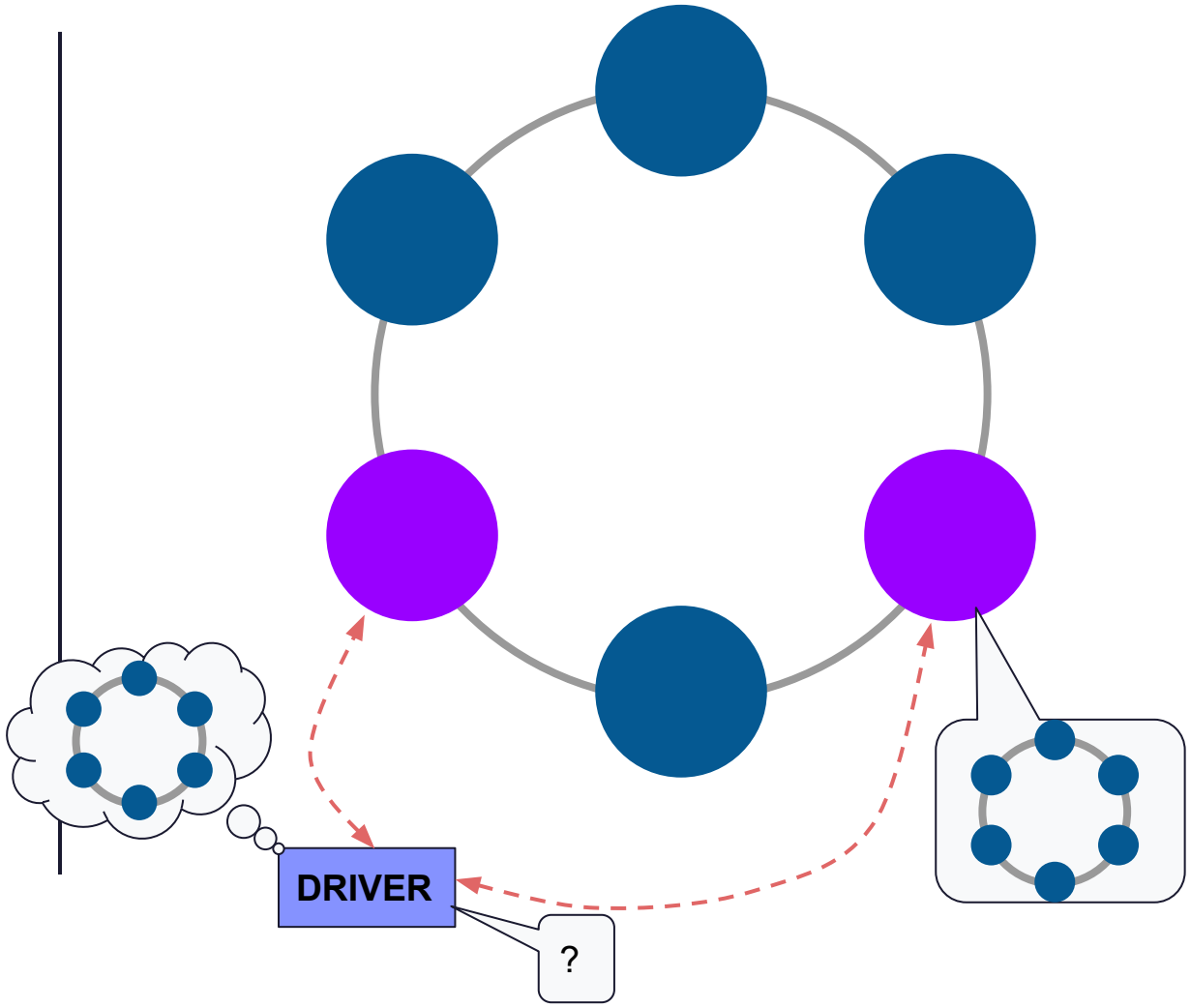
```
npm install cassandra-driver
```

```
{
  "dependencies": {
   "cassandra-driver": "^4.6.3"
  }
}
```

Connecting

Contact Points

- Only one necessary
- Unless that node is down
- Better ~3 nodes for resilience
- From there, drivers discover whole cluster



Connecting

Create a `Client` with connection parameters, then call `.connect()`

The `client` should be a singleton; shut it down to release resources

```
const { Client } = require("cassandra-driver");

const config = {
   contactPoints: ['12.34.56.78', '78.56.34.12'],
   localDataCenter: 'datacenter1',
   keyspace: 'chemistry'
};
const client = new Client(config);
await client.connect();
// ...
client.shutdown();
```

Connecting

If you have an Astra DB instance, just change **`config`**:

```
const { Client } = require('cassandra-driver');
const config = {
  cloud: {
   secureConnectBundle: process.env.SECURE_CONNECT_BUNDLE
  },
  credentials: {
   username: process.env.ASTRA_DB_CLIENT_ID,
   password: process.env.ASTRA_DB_CLIENT_SECRET
  },
  keyspace: 'chemistry'
};
const client = new Client(config);
await client.connect();
```

Mini-example

Connect, read rows and print them

```
const { Client } = require("cassandra-driver");
const config = {
    contactPoints: ['12.34.56.78', '78.56.34.12'],
    localDataCenter: 'datacenter1',
    keyspace: 'chemistry'
};
const client = new Client(config);

async function main() {
  await client.connect();
  const query = 'SELECT symbol, name FROM elements; ';
  client.execute(query)
    .then(result => {
     const rows = result.rows;
      rows.forEach( row => {
        console.log('Symbol for %s is %s', row['name'], row['symbol']);
     });
     client.shutdown();
    });
}

main();
```

Connectivity, options

Load balancing, retry policy, reconnection policies (you can also write your own)

```
const { Client, policies } = require('cassandra-driver');

const config = {
     contactPoints: ['12.34.56.78', '78.56.34.12'],
     localDataCenter: 'datacenter1',
     keyspace: 'chemistry'
};

const client = new Client({
  ...config,
  ...{
    'policies': {
      'loadBalancing': new policies.loadBalancing.TokenAwarePolicy(
        new policies.loadBalancing.DCAwareRoundRobinPolicy('datacenter1')
      ),
      'reconnection': new policies.reconnection. ExponentialReconnectionPolicy(
        2000,           // base value
        10 * 60 * 1000, // max value
        false,          // start with no delay?
      ),
     retry: new policies.retry.FallthroughRetryPolicy()
    }
  }
});
```

Querying

Running a query

CQL statements are passed to the client's `execute()` method

```
const creationCommand = `CREATE TABLE IF NOT EXISTS metals
  (kind TEXT,
  name TEXT,
  density FLOAT,
  PRIMARY KEY ((kind), name));`;

await client.execute(creationCommand);
```

Notable features:

- *concurrent execution primitives*
- *speculative query execution*
- **Promise-based API**
- **Callback-based API**

Write queries – 1

A single CQL command string with everything in it

```
await client.execute(`INSERT INTO metals (kind, name, density)
    VALUES ('regular', 'copper', 0.01);`);
```

We can do better (trouble with: serialization, escaping, injections)

Write queries – 2

Command / parameters, bound to one another

```
await client.execute(
  "INSERT INTO metals (kind, name, density) VALUES (?, ?, ?);",
  ['regular', 'palladium', 12.02],
  {
    hints: [
    'text',
    'text',
    'float'
    ]
  }
);
```

Must provide hints for types to be properly serialized (expect errors otherwise)

Write queries – 3

Enter **prepared statements**: schema info comes from server

```
const metalInsertionStatement = "INSERT INTO metals (kind, name, density) VALUES
(?, ?, ?);";

await client.execute(
  metalInsertionStatement,
  ['regular', 'zinc', 7.14],
  {prepare: true}
);

await client.execute(
  metalInsertionStatement,
  {
    kind: 'regular',
    name: 'mercury',
    density: 13.534
  },
  {prepare: true}
);
```

mapping to Cassandra types

statements are on nodes, waiting to be re-used

Write queries – 4

Promise-based insertion

```
const metalInsertionStatement = `INSERT INTO metals
    (kind, name, density) VALUES (?, ?, ?);`;

client.execute(
  metalInsertionStatement,
  ['band!', 'AC/DC', 9999.99],
  {prepare: true}
).then(
  r => console.log('AC/DC inserted!')
).catch(
  err => console.error('Did something go wrong?', err)
);
```

Write queries – 5

Callback-based insertion

```
const metalInsertionStatement = `INSERT INTO metals
    (kind, name, density) VALUES (?, ?, ?);`;

client.execute(
  metalInsertionStatement,
  ['band!', 'Iron Maiden', 9999.99],
  {prepare: true},
  (err, res) => {
    if (! err){
      console.log('Iron Maiden inserted!');
    }else{
      console.error('ERROR inserting Iron Maiden:', err);
    }
  }
)
```

Reading queries – 1

Promise-based (parameters passed as for insertions)

```
client.execute(
  'SELECT name, density FROM metals WHERE kind = ?;',
  ['band!'],
  {prepare: true}
).then(result => {
  const rows = result.rows;
  rows.forEach( row => {
    console.log('Band %s has density %s', row['name'], row['density']);
  });
});
```

Reading queries – 2

Callback-based (just pass a fourth argument to `execute()`)

```
client.execute(
  'SELECT density FROM metals WHERE kind = ? AND name = ?;',
  ['regular', 'copper'],
  {prepare: true},
  (err, res) => {
    const row = res.first();
    console.log('Copper density = %s', row['density']);
  }
);
```

Reading queries – 3

Using the `eachRow()` method

```
client.eachRow(
  'SELECT name, density FROM metals WHERE kind = ?;',
  ['regular'],
  {prepare: true},
  (n, row) => {
    console.log('Metal %s is %s with density %s', n, row.name, row.density)
  },
  err => {
    // if err is null, it just means 'no more rows left'
    if(err){
      console.error('Error reading regular metals:', err)
    }
  }
);
```

Batches

`client.batch()` - can be promise- or callback-based as well

```
const densityUpdateStatement = 'UPDATE metals SET density = ? WHERE kind = ? AND name = ?;';
const batchForMetals = [
  {
    query: metalInsertionStatement,
    params: {
      kind: 'regular',
      name: 'silver',
      density: 10.49
    }
  },
  {
    query: densityUpdateStatement,
    params: [
      8.95,
      'regular',
      'copper'
    ]
  }
];
client.batch(batchForMetals, { prepare: true })
  .then(function() {
    console.log('The whole batch succeeded');
  })
  .catch(function(err) {
    console.error('Batch has failed: ', err);
  });
```

Execution profiles – 1

Define several *execution profiles* when instantiating the client …

```
const { ExecutionProfile, types, policies } = require('cassandra-driver');

const defaultProfile = new ExecutionProfile('default', {
  readTimeout: 4000,
  consistency: types.consistencies.localQuorum
})
const pedanticProfile = new ExecutionProfile('pedantic', {
  consistency: types.consistencies.all,
  retry: new policies.retry. IdempotenceAwareRetryPolicy()
})
const localizedProfile = new ExecutionProfile('europe', {
  // let's pretend there's a datacenter we want to direct *some* queries to...
  loadBalancing: new policies.loadBalancing.DCAwareRoundRobinPolicy('overseas_dc')
})

const client = new Client({
  ...config,
  ...{
    profiles: [
     defaultProfile,
      pedanticProfile,
      localizedProfile
    ]
  }
});
```

Execution profiles – 2

… and use one or the other in your mixed-workload queries

```
const qry = 'SELECT density FROM metals WHERE kind = ? AND name = ?;';
const arg = ['regular', 'copper']

const row1 = (await client.execute(qry, arg,
  {
    executionProfile: 'pedantic'
  }
)).first();
console.log('Copper density = %s', row1['density']);

const row2 = (await client.execute(qry, arg,
  {
    executionProfile: 'europe'
  }
)).first();
console.log('Copper density = %s', row2['density']);

const row3 = (await client.execute(qry, arg)).first();    // will use the default profile
console.log('Copper density = %s', row3['density']);
```

Other query options

Consistency level, timeouts, page size, ...

```
const { types, policies } = require('cassandra-driver');
client.eachRow(
  'SELECT name, density FROM metals WHERE kind = ?;',
  ['regular'],
  {
    prepare: true,
    autoPage: true,   // this option relevant for eachRow() only
    consistency: types.consistencies.localQuorum,
    fetchSize: 2,
    readTimeout: 1500, // milliseconds
    policies: {
      retry: new policies.retry.FallthroughRetryPolicy(),
    }
  },
  (n, row) => {
    // 'n' will be an in-page index!
    console.log('Metal %s is %s with density %s', n, row.name, row.density)
  },
  err => {
    if(err){
      console.error('Error reading regular metals:', err)
    }
  }
);
```

A mini-API

A mini-API

```
const express = require('express');
const api = express();
api.use(express.json());
api.listen(5000, () => {
  console.log('API ready on port 5000');
});

const { Client } = require('cassandra-driver');
const config = ... // FILL ME!
const client = new Client(config);

api.get('/metal', (req, res) => {
  // get all metals
  client.execute(
    'SELECT name, density FROM metals WHERE kind = ?;',
    ['regular'],
    {prepare: true}
  ).then( qres => {
    const rows = qres.rows;
    res.send(rows);
  })
});
```

```
curl localhost:5000/metal | jq

curl localhost:5000/metal/silver | jq

curl -XPOST localhost:5000/metal \
  --data '{"density": 101.11, "name": "armonium"}' \
  -H "Content-Type: application/json" | jq

curl localhost:5000/metal | jq
```

```
api.get('/metal/:name', (req, res) => {
  // get a specific metal
  client.execute(
    'SELECT name, density FROM metals WHERE kind = ? AND name = ?',
    ['regular', req.params.name],
    {prepare: true}
  ).then( qres => {
    if (qres.rowLength > 0){
      const row = qres.first();
      res.send(row);
    } else {
      res.status(404).send('Not found');
    }
  });
});

api.post('/metal', (req, res) => {
  // upsert a metal
  const newMetal = {
    ...req.body,
    ...{kind: 'regular'}
  };
  console.log(newMetal);
  client.execute(
    'INSERT INTO metals (kind, name, density) VALUES (?, ?, ?);',
    newMetal,
    {prepare: true}
  ).then( () => {
    res.send({inserted: true});
  })
});
```

LIVE DEMO

Pagination

Pagination

Read queries return the first "page" of results (default **`fetchSize = 5000`**)

Automatic paging available (with *async iterator*):

```
const qry = 'SELECT name, density FROM metals WHERE kind = ?;';
const arg = ['regular'];
const fetchSize = 2;
const qOpts =  {
  prepare: true,
  fetchSize: fetchSize
};

const result = await client.execute(qry, arg, qOpts);
for await (const row of result) {
  console.log('Metal %s has density %s', row.name, row.density)
}
```

But sometimes manual paging preferrable …

Manual pagination - 1

Passing `autoPage: true` to `eachRow()`

```
const qry = 'SELECT name, density FROM metals WHERE kind = ?;';
const arg = ['regular'];
const fetchSize = 2;
const qOpts =  {
  prepare: true,
  fetchSize: fetchSize
};
client.eachRow(
  qry,
  arg,
  {
    ...{autoPage: true},
    ...qOpts
  },
  (n, row) => {
    // 'n' will be an **in-page** index
    console.log('Metal %s is %s with density %s', n, row.name, row.density)
  },
  err => {
    if(err){
      console.error('Error reading regular metals:', err)
    }
  }
);
```

Manual pagination – 2

Passing an explicit `pageState` (string) to next call

```
const qry = 'SELECT name, density FROM metals WHERE kind = ?;';
const arg = ['regular'];
const fetchSize = 2;
const qOpts =  {
  prepare: true,
  fetchSize: fetchSize
};

let resSet = await client.execute(qry, arg, qOpts);
console.log('first page:');
for(var row of resSet.rows){
  console.log('Metal %s has density %s', row.name, row.density)
}
while(resSet.pageState){
  console.log('another page:');
  resSet = await client.execute(qry, arg, {...qOpts, ...{pageState: resSet.pageState}});
  for(var row of resSet.rows){
    console.log('Metal %s has density %s', row.name, row.density)
  }
}
```

Manual pagination – 3

Invoking `nextPage()` within `eachRow()`

```
const qry = 'SELECT name, density FROM metals WHERE kind = ?;';
const arg = ['regular'];
const fetchSize = 2;
const qOpts =    {
  prepare: true,
  fetchSize: fetchSize
};
client.eachRow(
  qry,
  arg,
  qOpts,
  function (n, row) {
    // Invoked per each row in all the pages
    console.log('Metal %s has density %s', row.name, row.density)
  },
  function (err, result) {
    if (typeof result !== undefined) {
      pageState = result.pageState;
      console.log("   [found pageState: ", pageState, "]");
      if(pageState !== null) {
        console.log("   [Requesting next page]");
        result.nextPage();
      } else {
        console.log("   [End of results]");
      }
    } else {
      // some error to handle here
    }
  });
```

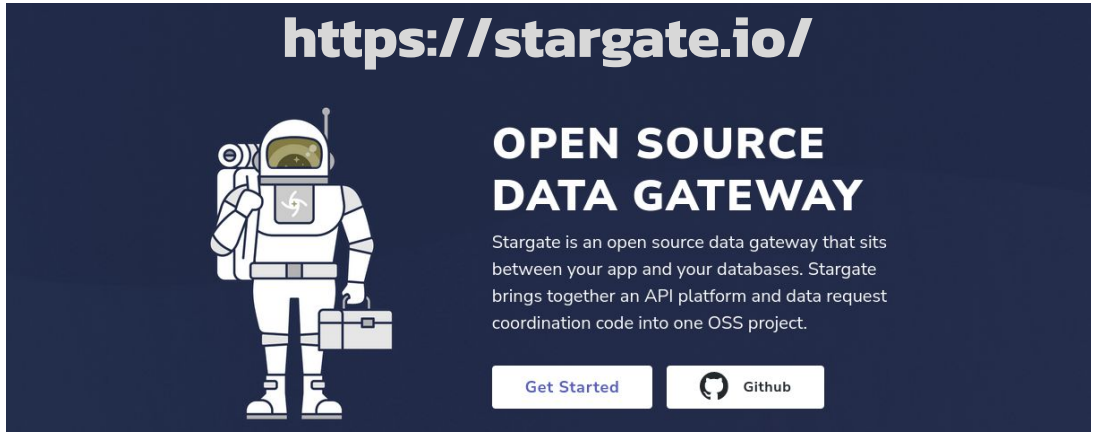
Manual pagination – 4

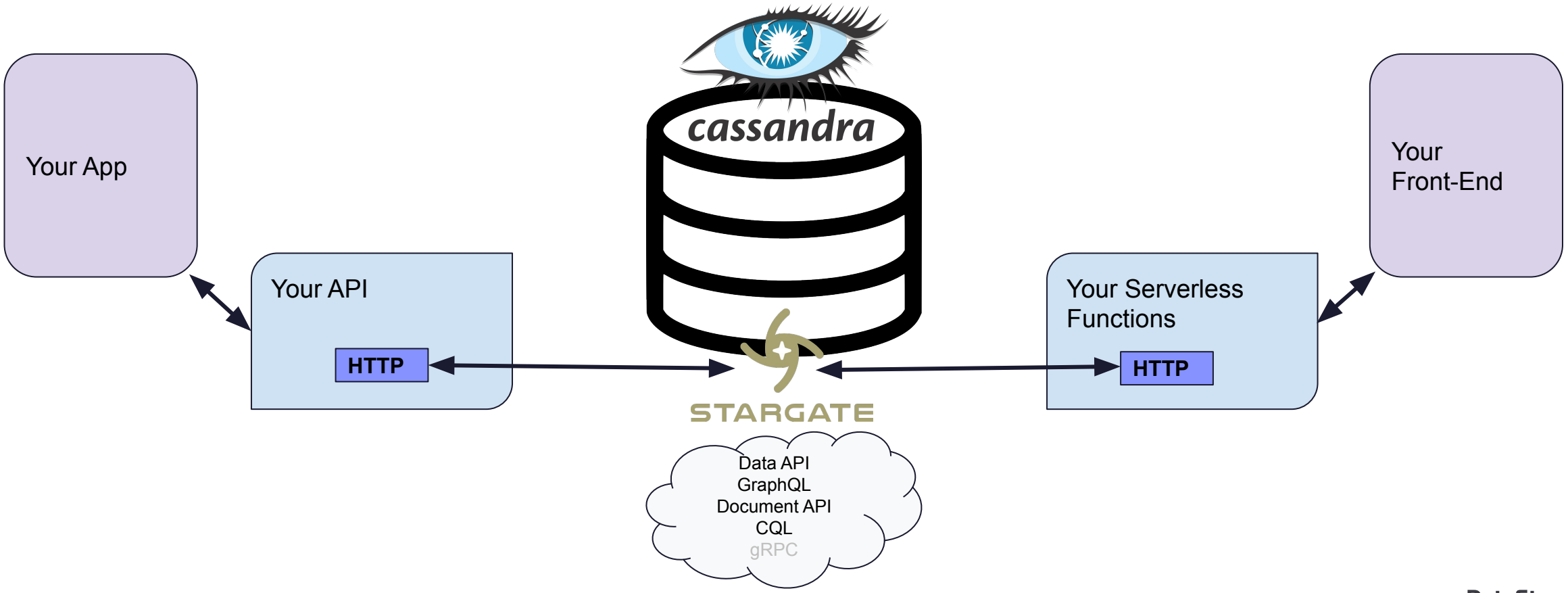
Using `client.stream()` to avoid clogging the local buffer

```
const qry = 'SELECT name, density FROM metals WHERE kind = ?;';
const arg = ['regular'];
const fetchSize = 2;
const qOpts =     {
  prepare: true,
  fetchSize: fetchSize
};

client.stream(qry, arg, qOpts)
  .on('readable', function () {
    console.log("   [Readable rows]");
    // there are emitted rows to be read
    var row;
    while (row = this.read()) {
      console.log('Metal %s has density %s', row.name, row.density)
    }
  })
  .on('end', function () {
    // no more rows to be consumed in the stream
    console.log("   [End of results]");
  });
```

An alternative to drivers





Stargate: "just HTTP requests"

- Document API
- GraphQL
- REST
- ...

```
const { createClient } = require("@astrajs/collections");
// (
cons
// (
cons
// (
// t
cons
  na
  gr
});
// (
cons
// (
cons
  na
  group: "u
});
// now user
// create a
const userS
  title: "t
  url: "htt
  metrics:
    posts:
    rating:
  }
});
// now user
// get a su
const userB
// now user
// search a
const users
```

```
curl . request PATCH \
  --ur                                                    wicklow.com/blog" \
  -H
  -H
  -H
  -d
    "t
    "u
    "m
    }
  }'
```

```
const fetch = require('node-fetch')

exports.handler = async function (event) {
  const body = JSON.parse(event.body)
  const genre = body.genre
  const pageState = body.pageState
  const url = process.env.ASTRA_GRAPHQL_ENDPOINT
  const query = `
  query {
    movies_by_genre (
      value: { genre: ${JSON.stringify(genre)}},
      orderBy: [year_DESC],
      options: { pageSize: 6, pageState: ${JSON.stringify(pageState)} }
    ) {
      values {
        year,
        title,
        duration,
        synopsis,
        thumbnail
      }
      pageState
    }
  }
  `
  const response = await fetch(url, {
    method: 'POST',
    headers: {
      "Content-Type": "application/json",
      "x-cassandra-token": process.env.ASTRA_DB_APPLICATION_TOKEN
    },
    body: JSON.stringify({ query })
  })
```

Conclusion

Resources for today's practice

CODE SAMPLE github.com/hemidactylus/cassandra-nodejs-drivers-practice

ASTRA DB astra.datastax.com

STARGATE stargate.io

Stargate-related references

WORKSHOPS

www.datastax.com/workshops

Past workshops at: www.youtube.com/datastaxdevs

SELF-SERVICE

github.com/datastaxdevs

Self-contained material from workshops

Resources for developers

LEARN

academy.datastax.com

Free online courses – Cassandra certifications

ASK/SHARE

community.datastax.com

Ask/answer community user questions – share your expertise

CONNECT

Follow us @DataStaxDevs

We are on YouTube – Twitter – Twitch!

CONTACT

dtsx.io/discord

Discord Community Server

Now go and create your wonderful Cassandra-backed application!

Thank You!

DataStax