

클레이풀 — AWSKRUG 데이터베이스 소모임 세션

MongoDB, The Good and Bad and 삽질 —

300+개의 API를 가진 API 기반의 쇼핑몰 솔루션 개발 MongoDB 사용기



누구신가요...?

- 디자인과 출신의 풀스택 개발자
- 개발자를 위한 판매 기능 API 솔루션, 클레이풀 창업자
- 다만 데이터베이스 전문가는 아닙니다...
- 그렇다고 AWS 전문도 아닙니다...



그럼 무슨 얘기 하시려구요...?

생각보다 MongoDB 관련된 이야기가 없더라고요!

크게 보면 이렇게 흥미진진하고 중요한 이야기를 진지하게 하게 됩니다.

- ▼ #0. 배경 상황

- ▼ #1. NoSQL의 강자 MongoDB는 왜 사용하게 되었나요?

- ▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요?

- ▼ #3. MongoDB 그 외?

- ▼ #4. Q&A

배경 상황

▼ #0. 배경 상황 —

부제목을 잠시 다시 가져왔습니다...



| **300+개의 API**를 가진 **API 기반의 쇼핑몰 솔루션** 개발 MongoDB 사용기

진짜 광고 타임 아니고 진짜 어디까지나 배경 상황 설명용인데요...

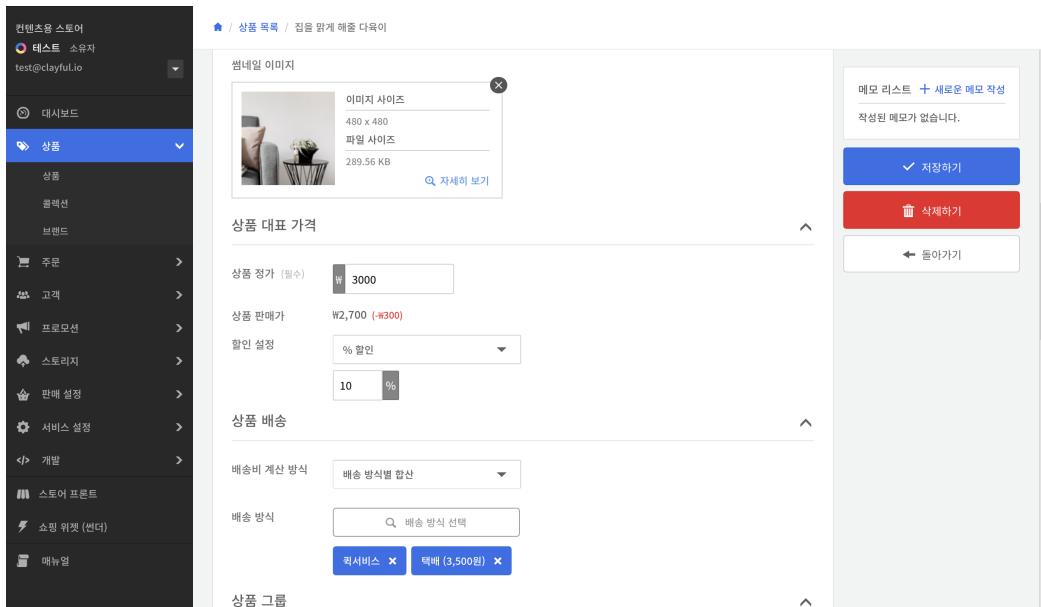
저희 클레이풀은 진짜 간략하게 어떤 서비스냐면요... 😊

- 에이치티티피에스 클레이풀 짹 아이오.
- 에이치티티피에스 클레이풀 짹 아이오.

- 에이치티티피에스 클레이풀 쪽 아이오.

▼ #0. 배경 상황 —

쇼핑몰처럼 관리자 페이지에서 판매용 상품 등록하고, 고객도 관리하고, 주문도 관리하고



▼ #0. 배경 상황 —

개발 언어나 환경에 관계 없이 API로 상품 읽어오고, 카트 담고, 주문하고 등등등

Cart.checkoutForMe

POST /v1/me/cart/checkout/{type}

고객 로그인 필요
공개/비공개 클라이언트 + model.(params.type) - 생성
필수: cart.checkoutForMe

로그인된 고객으로서 본인의 카트 내역을 주문합니다.

Node.js로 API 요청 예시

```
const Clayful = require('clayful');
const Cart = Clayful.Cart;
const payload = {
  ...
};

const options = {
  customer: '<customer-token>',
  query: { ... },
  ...
};

Cart.checkoutForMe('type', payload, options, (err, result) => {
  if (err) {
    // Error case
    console.log(err.code);
  }
});
```

Contents

- 1 API 기본 정보
- 2 Node.js로 API 요청 예시
- 3 API 요청 헤더와 인자
- 4 Payload 데이터
- 5 Query 옵션
- 6 반환 데이터
- 7 발생 가능 에러

▼ #0. 배경 상황 —

상황#1 — 쇼핑몰 솔루션



해외 쇼핑몰 솔루션인 Magento의 DB 설계도 (SQL)

— 출처: <https://magento.stackexchange.com/questions/101055/magento-2-diagram-of-database-tables-such-as-orders-products-etc-and-their>

- 쇼핑몰 솔루션 정말 복잡합니다... 상품, 세부 품목, 카테고리, 고객, 카트, 주문, 배송, 환불, 결제 ... (Magento 기준 테이블 300-400개)
- 운용적인 문제지만 데이터 설계 레벨의 관계/복잡도가 높을 수 밖에 없는 분야

- 일반적으로 트랜잭션이 지원되는 비지니스 로직이 들어가게 됨
-

▼ #0. 배경 상황 —

상황#2 — API 기반의 서비스 (SaaS)

- 공개 API만 300개지 사실 고객사에서 사용하는 내부용/비공개 API 포함하면 더 많아요...
- API 기반의 서비스다보니 고객사 서비스의 데이터베이스와 백엔드를 담당
- SaaS 서비스이다보니 데이터 하위 호환성 유지와 필요시 데이터/Schema 레벨에서 유연한 대응이 가능해야함
(설치형 솔루션들 처럼 고객사 별로 버저닝을 따로 할 수는 없으므로)

NoSQL의 강자 MongoDB는 왜 사용하게 되었나요?

▼ #1. NoSQL의 강자 MongoDB는 왜 사용하게 되었나요? —

부제: 나는 전생에 뭘 잘못했길래 다들 쓰는 SQL을 안쓰고 MongoDB를 사용하게 되었나

일단 MongoDB는?



- NoSQL 트렌드를 만들어낸 장본인 — Node.js와 함께 성장 (MEAN 스택, Meteor.js 등)
- 기본적으로 Schemaless한 JSON 구조(BSON)를 지원 — 데이터적으로 굉장히 유연함!
- 4.x 버전 이전에는 트랜잭션을 지원하지 않았음 — ...
- DBaaS 류에는 빅 플레이어로 Atlas(MongoDB에서 운영), MLab이 있었으나Atlas가 MLab 인수 → 현재 거의 독주?

▼ #1. NoSQL의 강자 MongoDB는 왜 사용하게 되었나요? —

부제: 나는 전생에 뭘 잘못했길래 다들 쓰는 SQL을 안쓰고 MongoDB를 사용하게 되었나

그래서 MongoDB는 왜 사용하게 되었나요?

사실 별 이유는 없습니다...



- 프론트엔드로 개발을 시작했던터라 JSON 구조에 훨씬 익숙했다는 점
 - 서버 개발을 Node.js로 해비하게 시작했던터라 JSON 구조에 훨씬 익숙했다는 점
 - 개인적으로는 Node.js와 그 시점에 가장 잘 어울렸던 DB라고 생각합니다.
 - 더불어 팀에서도 Node.js가 익숙했던 상황인지라 MongoDB로 결정
 - 결론: 그냥 JSON 구조가 편하고 좋아서
-

써보니 어떤 특징/장단점이 있나요?

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

우선 MongoDB 단어부터 조금 배워봅시다.

MongoDB

- 데이터베이스 (Database)
- 콜렉션 (Collection)
- 다큐먼트 (Document)
- 임베디드 다큐먼트 (Embedded Document)
- 필드 (Field)
- 룩업 (\$lookup) — 안써도 그만이라는 주의?

SQL

- 데이터베이스 (Database)
- 테이블 (Table)
- 로우 (Row)
- 로우 (Row) — 분리된
- 칼럼 (Column)
- 조인 (Join) — 당연히 기본

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Schemaless — 장점?

```
{  
  _id: 'ABCDE',  
  name: '아주 멋진 상품',  
  price: { // Object Nesting - 키 이름 고민을 덜해도 됩니다.  
    original: 15000,  
    sale: 10000  
  },  
  options: [ // 무려 2중 배열을 그냥 넣을 수 있습니다.  
    { name: '사이즈', values: ['M', 'L', 'XL'] },  
    { name: '컬러', values: ['Red', 'Blue', 'XL'] }  
  ],  
  variants: [  
    {  
      _id: '...',  
      name: '...',  
      price: 10000,  
      options:  
        {  
          size: '...',  
          color: '...',  
          ...  
        }  
    }  
  ]  
}
```

▼ 상황에 따라 유연하고 단순합니다.

- SQL이었다면 상품 모델 하나 때문에 테이블과 Join을 벌써 몇 번 을...?
- 필요한 경우에는 콜렉션(SQL의 테이블)을 분리해서 키를 레퍼런스 시키는 방식도 사용할 수 있기 때

```
{
  _id: 'ABCDE-A'
  price: {
    original: 15000,
    sale: 10000
  },
  options: [
    { name: '사이즈', value: 'M' },
    { name: '컬러', value: 'Red' }
  ]
}
]
```

문에, 설계 방식의 선택지가 많습니다.

- 한 Document에 필요한 정보들이 다 들어가 있어서 인지하기가 좀 편리합니다.

▼ 정리하면... 좀 쉽고 간단한 느낌?

→ (설계만 잘되었으면) 거의 대부분의 CRUD 오퍼레이션이 여러번 업데이트 없이 한 Document 안에서 이루어져, 어플리케이션 코드가 간단해집니다.

→ 데이터베이스계의 WYSIWYG 같은 느낌

→ 특히 One-to-many 상관 관계에서 Many가 많지 않다면, 한 Document 에 다 넣을 수 있어 설계/관리 복잡도를 정말 많이 줄일 수 있습니다.

→ 세어보니 약 50개 미만의 콜렉션으로 현재 커머스 서비스를 운영하고 있습니다. (Magento와 동일한 기능을 제공하진 않지만, 최소 3배는 줄인 느낌)

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Schemaless — 단점?

```
{
  _id: 'ABCDE',
  name: '아주 멋진 상품',
  price: {
    original: 15000,
    sale: 10000
  },
}
```

▼ 너무 유연해서 문제일지도?

- 기본적으로 필드에 들어가는 값에는 타입형을 정해서 넣을 수 있지만, 자세한 Validation 룰을 DB 레벨에서 제어하지 않습니다.

```

options: [
  { name: '사이즈', values: ['M', 'L', 'XL'] },
  { name: '컬러', values: ['Red', 'Blue', 'Black'] }
],
variants: [
  {
    _id: 'ABCDE-A'
    price: {
      original: 15000,
      sale: 10000
    },
    options: [
      { name: '사이즈', value: 'M' },
      { name: '컬러', value: 'Red' }
    ]
  }
]
}

```

- 그러다보니 한 필드의 값이 Document에 따라서 키/값이 아예 없거나 객체 배열일 수도 있습니다. (예 — `undefined || [{ hello: 'world' }]`)
- 너무 편하게 생각하다보니 처음 시작하면 설계적인 부분을 등한시하고 데이터를 막 집어넣게 될 수도 있습니다. → MongoDB의 DB적 한계에 뒤늦게 부딪히고 후회하게 됩니다.

▼ 정리하면... 역시 어느 정도는 정량화된 게 편하다!

→ 완전 Schemaless로 사용하면 역으로 관리가 어려워지기 때문에, 주로 어플리케이션 레벨의 ORM/Validation Schema 라이브러리 등을 이용해서 관리합니다.

→ 어차피 Business Logic과 관련된 Validation은 어플리케이션 레벨에서 이루어지니, Validation이 한 곳에서 관리된다는 장점도 있습니다.

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Schemaless — 해결안?

```

// Mongoose Schema
{
  _id: { type: String, required: true },
  name: { type: String, required: true },
  price: {
    original: { type: Number, required: true, min: 0 },
    sale: { type: Number, required: true, min: 0 }
  }
}

```

유연하게 관리할 수 있도록!

- 어차피 아주 Hotfix성 해결이 아닌 이상, DB에 디렉트하게 접근하여 다큐먼트가 생성/업데이트되는건 이상적이지 않은 상황!

```
  },
  options: [
    {
      name: { type: String, required: true },
      values: [ { type: String } ]
    }
  ]
}
```

- 항상 데이터가 어플리케이션 서버를 거치는게 더 이상적이므로 ODM 혹은 Schema Validation 라이브러리를 이용해 저장 전에 항상 확인.
- 어플리케이션 코드는 항상 Git에 의해 버저닝되고, 비지니스 로직의 업데이트와 함께 패칭되므로 DB 변화와 로직의 변화를 한번에 볼 수 있다는 장점도 생김.

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Schemaless — 결론

- 아무래도 웹 개발에서는 거의 무조건 사용되는 JSON 형태를 차용하다보니, DB 지식이 없는 초심자도 아주 쉽게 사용할 수 있다고 생각합니다.
- Schema가 자유롭다는건 "설계만 잘하면" 정말 정말 편합니다.
- 돌려말하면 너무 자유롭기 때문에 어플리케이션 레벨에서 버저닝과 데이터 관리를 잘 해줘야 합니다.
- 설계 레벨에서 One-to-Many, Many-to-Many 관계를 설계하는 방법론이 꽤 많고, 상황에 따른 설계 방안에 대한 튜토리얼이 많습니다.
- 일부 MongoDB만의 특징들을 모르고 데이터를 설계하면, 추후에 쿼리문이나 업데이트문에서 깔끔하게 처리를 못해 이슈가 생길 수 있습니다.
(너무 NoSQL적인 생각일 수도 있지만, 큰 문제만 아니면 대부분 어플리케이션 레벨에서 처리가 됩니다.)

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Syntax — 간단한 쿼리/업데이트

아, 참고로 MongoDB CLI 문법은 모두 JavaScript와 동일한 문법을 사용합니다!

```
db.products.update({
  // Query Condition
  'brand': 'Clayful',
  'price.original': { $gte: 10000 }
}, {
  // Update Commands
  '$set': { 'brand': '클레이풀' },
  '$inc': { 'price.original': 5000 },
  '$push': { 'tags': '10,000원 이상' }
});
```

일반적인 필드 조건/업데이트

- 기본적으로 `field: { $command: value }` 문법으로 매칭하게 됩니다.
- 업데이트는 `$command: { field: value }` 문법을 사용합니다.
- 네스팅된 객체, 배열 내 필드에 접근할 때는 기본적으로 `.` 을 이용할 수 있습니다. (매칭, 업데이트)
- `$inc`, `$push` 와 같은 특정 업데이트 커맨드는 Atomic하게 작동합니다.

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Syntax — 배열 쿼리/업데이트

그럼 배열에 대한 쿼리나 업데이트는 어떻게 이루어 질까요?

```
db.products.update({
  'options.0.name': '컬러'
}, {
  '$set': { 'options.0.name': '색상' }
});
```

찾으려는 요소의 위치(Index)를 알고 있다면?

- 이런 경우에는 객체와 동일하게 `.` 을 이용해서 N번째 요소에ダイ렉트하게 접근할 수 있습니다.
- 다중 배열이라도 `a.0.b.1.c.2` 와 같이 접근할 수 있습니다.
- 다만, 요소의 순차가 자주 바뀔 가능성 이 있는 경우에는 조금 위험할 수 있습니다.

```
db.products.update({
  'options.name': '컬러',
}, {
  '$set': { 'options.$.name': '색상' }
});
```

찾으려는 요소의 위치를 모른다면?

- 한 배열에서 조건에 의해 매칭된 N번 째 요소를 지칭할 때는 `$` 을 이용해 편리하게 업데이트!
- 다만, 끔찍하게도 3.6 버전 이전에는 배열 내 매칭된 복수개 요소를 한번에 업데이트하지는 못했습니다... (무조건 배열 내 첫번째 매칭된 요소만 업데이트)
- 더 끔찍하게도 2중 배열 이상의 다중 배열의 경우 `a.$.b.$.c` 처럼 업데이트 할 수 없습니다 → 너무 덥스를 깊게는 안하는게 좋아요.

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Syntax — 배열 내 복수개 조건 매칭

다음 두 예시는 유사하지만 다릅니다. 처음 할 때는 생각보다 헛갈립니다...

```
db.products.update({
  'options.name': '컬러',
  'options.values': '레드',
}, { /* ... */ });
```

배열 내 요소 관계 없이 여러 조건

- `options` 배열 내에서 `name`, `values` 에 매칭되는 조건을 모두 포함하는지 확인하는 쿼리
- 만약 0번째 요소가 `name`에 매칭되고, 3번째 요소가 `values`에 매칭되더라도 작동합니다.

```
db.products.update({
  'options': { '$elemMatch': {
    'name': '컬러',
    'values': '레드',
  } }
}, { /* ... */ });
```

배열 내 한 요소에 대한 여러 조건

(`$elemMatch`)

- 하나의 `options` 요소가 조건에 모두 맞는지 확인합니다.
- 배열 내 복수개 조건을 만족하는 한가의 요소를 확인하는 경우 무조건 `$elemMatch` 를 사용하세요!

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Syntax — 더 복잡한 쿼리 조건

```
db.products.update({
  '$where': function() {
    return (
      this.brand === 'Clayful' &&
      this.tags.indexOf('API') >= 0 &&
      this.tags.indexOf('eCommerce') >= 0
    );
  },
  { /* ... */ });
}
```

아주 아주 복잡한 조건 (\$where)

- `$where` 조건문을 사용하면 JavaScript를 그냥 사용할 수 있습니다.
- 당연하지만 인덱스를 전혀 사용하지 않기 때문에, 지엽적인 인덱스가 작동하는 쿼리문과 함께 사용하면 편리할 수 있습니다.
- 사실 생각보다 MongoDB 쿼리가 리치하기 때문에 그렇게 자주 사용할 일은 없었습니다.
(그래도 있으면 분명 써먹는 시점이 오는 가능?)

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Syntax — 결론

- NoSQL에서 메이저급 DB다보니 기본적인건 다 커버됩니다.
- 배열(Embedded Document) 내부 검색이나 업데이트도 안될 것 같아도 웬만한 건 다 됩니다.
- 그래도 Schema 레벨에서 중첩된 배열을 너무 남발하면, 업데이트 구문 처리하기가 어려워집니다.
- (설명은 않았지만) Aggregation을 활용하면 데이터 추합 등의 통계적인 일도 리치하게 처리할 수 있습니다.

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Indexing — 기본적인 인덱스

```
// Indexing
db.products.createIndex({ 'brand': 1 });

// Query
db.products.find({ 'brand': 'Clayful' });
```

Single Field Index

- 일반적인 DB에서의 한 필드 인덱스 설정입니다.
- 값을 `1`로 설정하느냐 `-1`로 설정하느냐에 따라 인덱스의 ASC, DESC를 설정하게 됩니다.

```
// Indexing
db.products.createIndex({ 'categories': 1 });

// Query
db.products.find({ 'categories': 'Pants' });
```

Multi Key Index

- 배열형 정보의 인덱스 설정입니다.
- 배열 내 각 요소 정보에 대해서 각각 모두 인덱스를 설정하게 됩니다.
- 예를 들면 상품과 카테고리와 관계와 같은 색인이 필요한 경우 사용합니다.

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Indexing — 비정형 데이터에 대한 색인#1

```
{
  ...
  meta: {
    hello: 'world',
    myField: 'myValue'
  }
}
```

Schemaless와 인덱스

Schemaless한 특징 때문에 유저가 임의로 생성하는 커스텀 필드 등에서도 독특한 인덱스 전략도 사용할 수 있습니다.

예를 들어 좌측과 같은 정보가 있는데, 커스텀 정보를 인덱싱할 필요가 있다면...

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Indexing — 비정형 데이터에 대한 색인#2

```
{  
  ...,  
  meta: {  
    hello: 'world',  
    myField: 'myValue'  
  }  
}
```

객체 구조를 그대로 유지하려는 경우

- 만약 중첩된 객체 구조를 그대로 유지하면서 색인하려는 경우, 현 시점에는 한가지 방법밖에 없습니다.
- 그냥 이런 인덱싱을 할 수 있다는 사실에 감사합니다.

```
// Indexing  
db.products.createIndex({ 'meta.$**' : 1 });  
  
// Query  
db.products.find({ 'meta.hello': 'world' });
```

Wildcard Index

- `meta` 내부에 있는 모든 자식 필드들의 값이 인덱싱 되게 됩니다.
- 하지만 가장 최근에 나온 스펙이라 아직 못쓰실 수도 있습니다... (4.2버전부터 지원)

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Indexing — 비정형 데이터에 대한 색인#3

그럼 가장 최신 버전인 4.2 이전 버전에서는 방법이 없을까요?

```
{  
  meta: [  
    { key: 'hello', value: 'world' },  
    { key: 'myField', value: 'myValue' }  
  ]  
}
```

설계를 약간 바꿔서 처리할 수 있습니다!

- 기존 중첩 객체와 같은 구조를 배열로 저장하면서 해결할 수 있습니다.

Multi Key Index

```
// Indexing
db.products.createIndex({ 'meta.value' : 1 });

// Query
db.products.find({
  'meta.key': 'hello',
  'meta.value': 'world' // Indexed
});
```

- 이전에 알아보았던 Multi Key 인덱스를 사용해서 커스텀 정보 내 값들을 인덱싱합니다.
- `meta.value` 값에 대한 쿼리가 인덱싱되어 있으므로 `meta.key` 값에 대한 쿼리를 추가적으로 넣어도 효율적으로 작동합니다.

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

Indexing — 결론

- 일반적인 Key, Value에 대한 인덱스는 당연하지만 잘 됩니다.
- 배열 내 요소에 대한 인덱스도 지원되기 때문에 DB/쿼리 설계시 크게 걱정하실 필요는 없습니다.
- 커스텀 정보와 같은 비정형 데이터에 대한 인덱스가 지원되는 점은 SaaS 솔루션 입장에서 너무 좋습니다.
- 복수개 조건을 통해 매칭하려는 경우, 효율성을 위해 Index Intersection, Compound Index와 같은 조금 더 복잡한 개념들이 있습니다.
- → 하지만 어차피 쿼리 효율성까지 들어가면 쿼리 퍼포먼스 분석이 동반되어야하니 키워드만 던지고 갑니다. 😅

... 지금까지는 다 좋아보이시죠? 후후후후후 :)

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

트랜잭션 — ...하

MongoDB는 4.0 버전 이전에는 트랜잭션을 지원하지 않았습니다... (~~쇼핑몰 서비스인데 어 떠하지...~~)



▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

트랜잭션 — ...해결은 해야지

마음을 고쳐먹고 왜 잘못된 선택이 아니었는지 여러가지 변명을 해봅니다.

#1. 어차피 커머스 서비스 특성상, 일부 기능들이 3rd 파티 서비스(외부 시스템/DB)랑 연관 관계가 있을 수 밖에 없습니다.

결제 API 서비스, 물류 API 서비스, 상품 추천 API 서비스 기타 등등...

#2. 어차피 마이크로 서비스 환경에서 모든 DB를 이원화해서 줘는건 어렵다.

#1번과 더불어서 마이크로 서비스 구성이 보편화되면 어차피 하나의 DB가 모든걸 해결할 수는 없다.

→ 이참에 분산 데이터베이스 환경에서 트랜잭션을 최대한 보장하는 법을 배워보자!

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

트랜잭션 — Two Phase Commit

시작 전(initial) → 대기(pending) → 적용됨(applied) → 완료(done)

- 한 작업에 대한 로그를 남길 수 있는 트랜잭션 객체와 이의 적용 상태(위)를 이용한 방식입니다.
- MongoDB에서 트랜잭션을 지원하기 전에 권장하던 방법입니다.
- 개념적으로 보면 MongoDB 뿐만 아니라, 멀티 데이터베이스/서비스 환경에 다 적용할 수 있습니다.
- 예시로 고객이 주문 → 주문된 상품에 대해서 재고가 차감되어야하는 상황...

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

트랜잭션 — Two Phase Commit#1

먼저 트랜잭션 작업할 내역을 남기고 (`initial`), 실제 트랜잭션 시작시 대기(`pending`) 상태로 바꿉니다. (단순 구분용)

트랜잭션 콜렉션

```
{  
  '_id': 'T001',  
  'status': 'pending', // < 'initial'  
  'product': 'P001',  
  'order': '0001',  
  'item': 'I001'  
}
```

상품 다큐먼트

```
{  
  '_id': 'P001',  
  'name': '상품 A',  
  'quantity': 10, // 차감 후 7개  
  'transactions': []  
}
```

주문 다큐먼트

```
{  
  '_id': '0001',  
  'status': 'pending',  
  'items': [  
    {  
      '_id': 'I001',  
      'product': '상품 A',  
      'quantity': 3  
    }  
  ]  
}
```

```
        }
    ],
    'transactions': []
}
```

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

트랜잭션 — Two Phase Commit#2

트랜잭션을 각각 반영합니다. 업데이트가 따로 2회 발생하고 중간에 실패하더라도, 반영 내역이 남아 있으니 다른 다큐먼트에만 반영해도 됩니다.

트랜잭션 콜렉션

```
{  
  '_id': 'T001',  
  'status': 'pending',  
  'product': 'P001',  
  'order': '0001',  
  'item': 'I001'  
}
```

상품 다큐먼트

```
{  
  '_id': 'P001',  
  'name': '상품 A',  
  'quantity': 7,  
  'transactions': ['T001']  
}
```

주문 다큐먼트

```
{  
  '_id': '0001',  
  'status': 'pending',  
  'items': [  
    {  
      '_id': 'I001',  
      'product': '상품 A',  
      'quantity': 3  
    }  
  ],  
  'transactions': ['T001']  
}
```

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

트랜잭션 — Two Phase Commit#3

각 다큐먼트에 반영이 완료 되었으니, 트랜잭션 상태를 적용 완료([applied](#))로 바꿉니다.

트랜잭션 콜렉션

```
{  
  '_id': 'T001',  
  'status': 'applied',  
  'product': 'P001',  
  'order': '0001',  
  'item': 'I001'  
}
```

상품 다큐먼트

```
{  
  '_id': 'P001',  
  'name': '상품 A',  
  'quantity': 7,  
  'transactions': ['T001']  
}
```

주문 다큐먼트

```
{  
  '_id': '0001',  
  'status': 'pending',  
  'items': [  
    {  
      '_id': 'I001',  
      'product': '상품 A',  
      'quantity': 3  
    }  
  ],  
  'transactions': ['T001']  
}
```

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

트랜잭션 — Two Phase Commit#4

이제 반영 되었으니 적용된 트랜잭션을 각 다큐먼트에서 빼줍니다.

트랜잭션 콜렉션

```
{  
  '_id': 'T001',  
  'status': 'applied',  
  'product': 'P001',  
  'order': '0001',  
  'item': 'I001'  
}
```

상품 다큐먼트

```
{  
  '_id': 'P001',  
  'name': '상품 A',  
  'quantity': 7,  
  'transactions': []  
}
```

주문 다큐먼트

```
{  
  '_id': '0001',  
  'status': 'pending',  
  'items': [  
    {  
      '_id': 'I001',  
      'product': '상품 A',  
      'quantity': 3  
    }  
  ],  
  'transactions': []  
}
```

트랜잭션 — Two Phase Commit#5

모든 작업이 완료 되었으니, 트랜잭션의 상태를 완료(`done`)으로 변경해줍니다 → 끝!

트랜잭션 콜렉션

```
{  
  '_id': 'T001',  
  'status': 'done',  
  'product': 'P001',  
  'order': '0001',  
  'item': 'I001'  
}
```

상품 다큐먼트

```
{  
  '_id': 'P001',  
  'name': '상품 A',  
  'quantity': 7,  
  'transactions': []  
}
```

주문 다큐먼트

```
{  
  '_id': '0001',  
  'status': 'pending',  
  'items': [  
    {  
      '_id': 'I001',  
      'product': '상품 A',  
      'quantity': 3  
    }  
  ],  
  'transactions': []  
}
```

▼ #2. MongoDB는 써보니 어떤 특징/장단점이 있나요? —

트랜잭션 — Two Phase Commit#결론

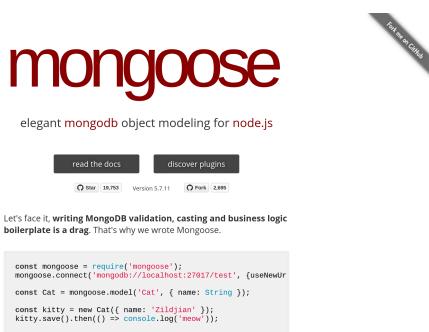
- (!) 어찌 되었던지 간에 어플리케이션 레벨에서 트랜잭션 처리를 할 수 있습니다.
 - 절차가 많고 어플리케이션 레벨에서 처리하기 때문에 조금 느리다는 단점이 있습니다.
 - **리커버리**: 작업이 끊어진 지점부터 다시 절차를 밟으면 최종적으로 반영이 완료되게 됩니다.
 - **롤백**: 작업이 중단된 상태를 체크하고, 트랜잭션 상태를 `cancelling` 등으로 바꾸면서 위 절차를 유사하게 진행하면 됩니다.
 - ~~근데 막상 적용하면 드럽게 복잡합니다... 되도록 하지 마세요... 푸푸~~
-

MongoDB 그 외?

-
- ▼ #3. MongoDB 에코시스템은 얼마나 잘 되어 있나요?

ODM — Mongoose

Node.js 쪽에서 많이 사용되는 MongoDB용 ODM 라이브러리

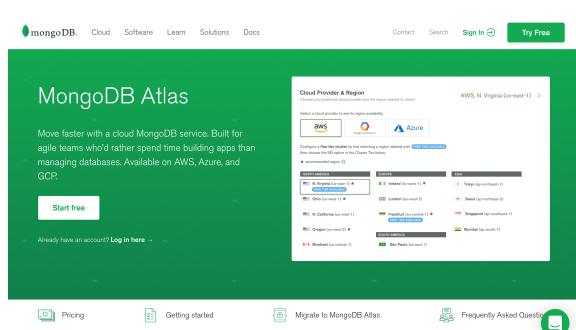


- 사용하면 Schema, Index 관리 등이 많이 간단해지고, 간단한 CRUD 오퍼레이션이 편리해집니다.
- Schema, Index에 대한 관리가 파일 레벨에서 일어날 수 밖에 없기 때문에, Git과 함께 이용시 데이터베이스 버저닝 관리가 자동으로 이루어집니다.
- 일반적으로 단순한 CRUD는 Mongoose를 이용하고...
- 특수한 쿼리/업데이트 조건의 경우에는 MongoDB 고유 문법을 최대한 사용합니다.

▼ #3. MongoDB 에코시스템은 얼마나 잘 되어 있나요?

MongoDB Atlas

MongoDB를 개발한 회사에서 직접 운영하는 DBaaS



- 장애 상황에 대한 서포트 대응 빠른편 (현재까지는)
- AWS 환경에 배포 가능하며 VPC Peering 등을 지원
- Slow Query 분석, Real-time Monitoring, 자동 백업 등 다양한 기능 제공
- 빅 플레이어인 MLab 인수 후 거의 독보적인 선두 주자라고 개인적으로 생각

- (AWS도 비슷하지만) 영미권에서 운영 중이기 때문에 개발자 문서, 서포트 등은 영어를 하면 좀 편리합니다.
 - MongoDB 한국 지사 생겼으니 관심 있으시면 알아보시는 것도...
 - 전반적으로 데이터베이스 쪽 매니징을 덜해도 되서 만족스럽게 사용하고 있습니다.
-

결론 —

- Schema가 유연한 건 좋지만, 네스팅은 그래도 줄이는게 좋습니다.
 - 어플리케이션 레벨에서 Schema 관리 하는게 생각보다 편합니다.
 - 쓰다보면 매니지먼트 코스트가 적다는 느낌이 들고, 왜 스타트업 개발자들이 많이 쓰는지 이해할 수 있습니다.
 - (DB특이지만) 그래도 설계를 잘하려면 공부는 많~이 필요합니다.
 - 주요 이슈인 트랜잭션도 4.2 버전에서 많이 해결된 것 같습니다.
-

끝 + Q&A —

~~이상 개발 언어/환경에 관계 없이 어디에나 연동해서 사용할 수 있는 API 기반의 쇼핑몰 솔루션인 클레이풀의 김대익이었습니다.~~ 😊

발표 자료나 내용에 대해 궁금하신 점이 있으시다면 → daeik.kim@clayful.io

클레이풀 서비스에 대해서 궁금하신 점이 있으시다면 → <https://clayful.io>

아몰랑 전 개발자니 서비스 개발자 문서나 볼래요 → <https://dev.clayful.io>
