프로젝트 결과보고서

1. 프로젝트 개요

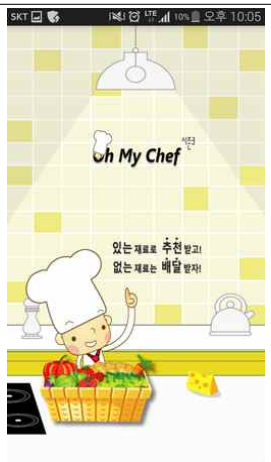
o 기본정보

프로젝트명	개인화 요리 추천 서비스[실제 레시피, 영양/식이 정보를 활용한]		
참여 학생	윤홍석(20103365) 최슬(20115237)	**교수님**	임성수 교수님

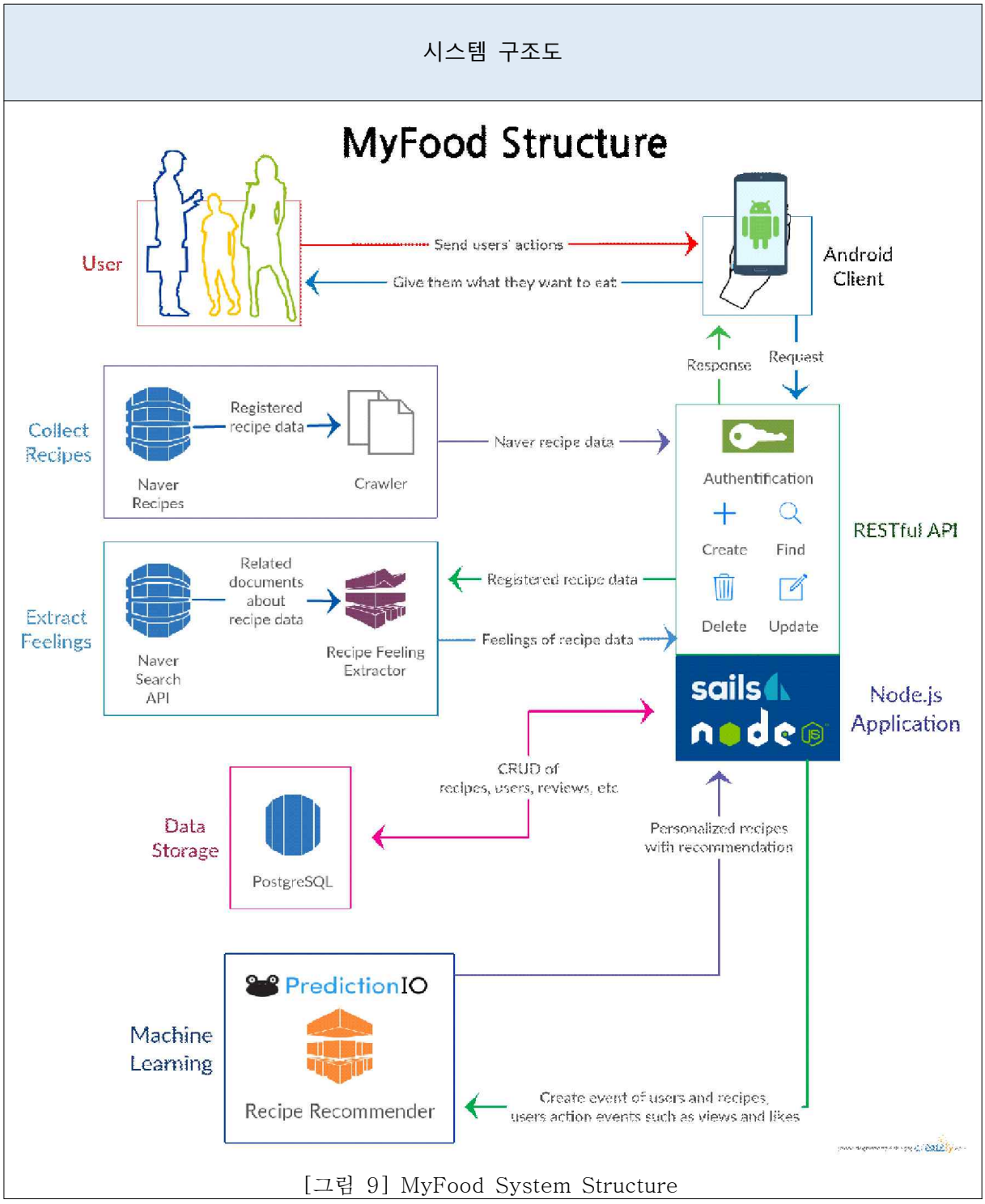
o SW개발 환경

구분	세부 내용
사용언어	Client – Java, Android SDK API 22, Main Server – Node.js 0.12.7, Crawler – Python 2.7, Machine Learing Server – Scala 2.11.7
실습장비	안드로이드 스마트폰 (넥서스5, 갤럭시6, 갤럭시 노트2)

o 기존 유사SW 현황

구분	세부 내용
만개의 레시피	   • 레시피를 종류별, 상황별, 방법별, 재료별로 필터링하여 제공 • 즐겨찾는 레시피 스크랩 기능 • 내가 만든 요리/레시피 공유 기능 • 커뮤니티 기능 • 컬럼수와 스크랩수에 따른 유저 랭킹 시스템 • 레시피 내용이 자세하지 않아 불편을 호소하는 사례 많음 • 필터링된 레시피의 정렬기준이 최신순, 베스트순, 조회순 등으로 이루어져 있어서 개인화 요리추천과는 거리가 있음
오마이셰프	   • 레시피를 분류별, 테마별, 상황별로 필터링하여 제공 • 나의 냉장고에 있는 재료를 세세하게 입력하고 그에 기반한 필터링 제공 • 정렬기준은 입력받은 재료와 레시피의 주재료가 가장 많이 겹치는 순서 • 장보기 메뉴를 통해 인기상품 공동구매 가능 • 사용자의 행동패턴을 기반으로 자동화된 추천을 하지는 않으며, 정확한 결과를 얻으려면 사용자가 직접 상세한 필터를 적용해야 해서 다소 번거롭다는 단점

o 시스템 구조도



[그림 9] MyFood System Structure

o 팀원 역할분담

팀원명	전문분야	주요 역할
1. 윤홍석	알고리즘	• Project Manager역할 - 팀원간 역할 및 일정 조율, 전체적인 프로젝트의 진행 방향 및 계획 운용 • 레시피 추천알고리즘 설계, 구현 - Collaborative Filtering Algorithm - Content-Based Filtering Algorithm • 메인 API서버와 연동 가능한 ML서버 구축 - PredictionIO Template을 이용함
2. 최슬	UI/UX	• 스토리보드 담당 • 앱 전체의 디자인을 담당했으며 디자인 설계와 해당 디자인에 어울리는 UX를 구현함
3. 최슬	클라이언트	• 안드로이드 클라이언트 개발 - 회원가입, 로그인, 개인화된 레시피 조회, 좋아요, 레시피 필터링 등의 기능 구현 - RESTful API를 통한 Main 서버 연동
4. 윤홍석, 최슬	백엔드	• 레시피 RESTful API 개발 - OAuth 2.0 인증 - 회원, 레시피, 좋아요, 리뷰 같은 중요 데이터 항목들을 생성, 수정, 조회, 삭제할 수 있음 - Machine Learning 서버 연동 • 크롤러 개발 - 네이버 음식백과 데이터 수집 • 식감 추출 - 레시피와 관련있는 식감 부여: 짜다, 맵다, 담백하다 등

2. 추진성과

가. 기술능력

o SW개발능력 : 프로젝트 구현, 오류해결

<백엔드>

- RESTful API
  - Node.js의 Sails 프레임워크로 구현함
  - 구현시 비동기의 Callback Hell 문제 -> async를 통해 해소함
  - 모든 컬렉션들에 대해서 공통된 방법으로 서버 외부에서 CRUD 제어 가능함
- 데이터 수집
  - 네이버 레시피 데이터 수집함
  - 크롤러는 Python의 Scrapy 프레임워크를 이용해 구현함
- 데이터 가공
  - 레시피에 식감을 부여하여 사용자의 레시피 접근성을 향상함
  - Google의 Word2vec으로 시도해보았지만 만족스런 결과 도출 실패함
  - 수기 사전 이용하여 식감 수집 후 부여함

<추천알고리즘>

- 알고리즘 구현
  - PredictionIO의 'E-Commerce Recommendation Engine Template'과 'Similar Product Engine Template'을 결합, 수정, 보완하여 Collaborative Filtering Algorithm을 구현하였음
  - Collaborative Filtering Algorithm은 오픈소스를 접하기 용이하고, 데이터가 쌓일수록 신뢰성 있는 성능을 보이기 때문에 많은 서비스들이 이를 채택하여 사용하고 있음
  - 하지만 우리 서비스의 경우 앱의 특성상 더욱더 개인화된 레시피 추천을 해야할 필요성이 있었고, 시작시 유저의 데이터가 별로 없어 추천 성능이 잘 안나오는 'Cold Start Problem'이 있었음
  - 따라서 비교적 구현하기 쉬우면서도 사용자의 레시피 선호도를 반영하기에 적합한 Content-Based Filtering Algorithm을 PredictionIO에 도입함
  - Content-Based Filtering Algorithm은 사용자가 액션을 취한 아이템의 내용을 기반으로 비슷한 아이템을 추천해주는 알고리즘

- ML서버 구축
  - PredictionIO Template 기반으로 커스터마이징 하였음
  - 하나의 쿼리에 Collaborative Filtering Algorithm과 Content-Based Filtering Algorithm을 동시에 수행하여 결과값을 합산, 반환하도록 함
  - 기존 유저의 최근 행동 패턴을 기반으로 비슷한(선호할만한) 레시피를 추천
  - 새로운 유저에게는 대중적으로 인기있는 레시피를 추천
  - 보지 않은 아이템만 추천하는 기능 (Optional)

- 카테고리, 식감, 화이트리스트, 블랙리스트 필터링 기능 (Optional)
- 페이징 기능 구현

• 오류해결
- PredictionIO가 Scala로 작성되어 있었는데, Scala라는 언어가 매우 생소해서 초반에 개발에 난항을 겪었음
- 또한 Collaborative Filtering과 Content-Based Filtering 두 알고리즘의 프로세스가 매우 상이하여 하나의 Recommender로 합하는 것에 어려움이 있었음
- Scala관련 서적과 레퍼런스 등을 찾아보며 기초부터 확실하게 습득한 뒤 단계적으로 문제들을 차근차근 해결함
- 두 알고리즘을 완전히 유기적으로 합하지는 못하였으나, 대신 하나의 쿼리에 두 알고리즘을 독립적으로 수행하여 결과값을 합산하는 방식을 취하였음

<클라이언트>
• 기능구현
- supportlibrary v22를 사용하여 Android 5.0 Material design을 구현함
- Volley 라이브러리를 통해 RESTful API로 Request를 전달함
- 많은 이미지로 인한 OOM을 피하기 위해 Volley의 NetworkImageView를 사용함
- 메모리 낭비를 막기 위해 이미지를 메모리 캐쉬에 저장하는 방식을 취함

• 오류 해결
- NestedLayout과 CollapsingToolbarLayout을 같이 사용하는 도중 자체 ScrollView인 NestedLayout 때문에 충돌이 일어나 Recycle Layout으로 바꾸어줌
- 원인을 알 수 없는 오류일 때, 라이브러리의 이슈를 확인하여 버전 업데이트로 해결함

<UI/UX>
• 레이아웃 UX 구현
- 다양한 기본 레이아웃들의 가중치와 넓이와 높이를 적절히 조정해 화면 구성을 함
- NavigationView 사용함. 네비게이션 뷰 안에 헤더xml을 두었고, 그 안에 라이브러리를 사용하여 이미지와 텍스트들을 배치하였다. 네비게이션 뷰에는 drawer.xml이 있어 drawer.xml이 메뉴를 구성한다.
- CollapsingToolbarLayout안에 Toolbar를 두어 collapsing되면서 Toolbar가 생기는 레이아웃을 구현함
- Review에 쓰이는 CustomListView와 해당 어댑터를 생성함
- 머티리얼 디자인의 기본요소인 cardView를 사용함

• 효과 구현
- 블러효과 오픈소스를 활용함. 레이아웃 background에 알파값을 주어 해당 view

의 느낌을 조금씩 달리함
- RecycleView를 좌우로 swipe하면 사라지는 기능을 구현함
- Toolbar에 있는 NavigationView를 여는 아이콘을 클릭하면 화살표로 바뀌면서 도는 애니메이션을 적용시킴
- 다양한 레이아웃에 폰트를 적용함. fonthPath와 setTypeface를 이용해서 폰트를 적용한 것 뿐만 아니라 레이아웃의 구조적인 측면으로 들어가 폰트를 적용함.
- 그 외에도 지금은 쓰지 않는 메인에서 pull to regresh기능을 구현하기도 함

- 오류 해결
- floating기능과 collapsingToolbarLayout을 함께 쓰면 margin이 적용되지 않는 버그들이 있었는데 이런 부분들은 수동적으로 고쳐주었음
- collapsingToolbarLayout과 listView를 쓰면 두 스크롤이 충돌하여서 맨 처음에는 중간에 textView를 넣는 방식을 썼음. 하지만 이 방식으로도 리스트가 끝까지 올라가지 않아 리스트의 크기 자체를 변경해 줘서 일부 문제를 해결했으나 후에는 RecycleView를 써서 이 문제를 해결하였음
- Button이 눌렸을 때와 누른 채 유지되는 효과가 제대로 유지되지 않아 버틀 리스너를 달아 flag로 상황을 체크해 배경을 달리해주는 방식을 썼음

o SW개발지식 : 전문지식, 신규습득, 이론의 실무적용

<백엔드>
- RESTful API 사용 제한에 필요한 인증은 다양한 방식이 존재함. 대표적인 인증방식인 OAuth 2.0 이용함. 많은 기업들이 OAuth 2.0을 제공하지만 OAuth 2.0는 문제점이 많이 야기되고 있음

- 좋아요, 조회수 등 중요하지 않은 정보들을 실시간 반영은 자원소모 초래함. 스케쥴러 이용 일정주기마다 일괄 처리하는 것이 트래픽이 많은 서비스에서 필요함

<추천알고리즘>
- 전문지식, 신규습득
- 실제로 서비스에 어떠한 알고리즘을 적용해야만 성능이 가장 좋을지 고민하면서 많은 추천 알고리즘의 이해도가 자연스럽게 높아질 수 있었음
- 특히 이번 프로젝트에 도입한 Collaborative Filtering Algorithm과 Content-Based Filtering Algorithm의 구체적인 방식과 장단점을 학습하였음
- 결과적으로 각각의 단점을 해소하기 위해 둘을 혼합한 방식을 적용하여 의미있는 결과를 도출해 내었음

- 실무적용

- 오픈소스 템플릿을 커스터마이징하여 우리만의 서비스를 구현해본 것이 처음이라 매우 의미있는 경험이었음
- 서적에서 찾아볼 수 있는 유명한 알고리즘들이 실제 서비스에서 적용되고 있는 형태들을 보면서 자연스럽게 이론적인 지식들이 실무로 적용되는 사례를 학습하고, 응용할 수 있었음

<클라이언트>

- 전문지식, 신규습득
  - supportlibrary v22를 사용하여 보다 쉽게 Material design을 구현하는 방법을 습득함

- 실무적용
  - RESTful API를 통해 서버와 통신하고, OAuth 방식의 토큰 인증을 사용해봄

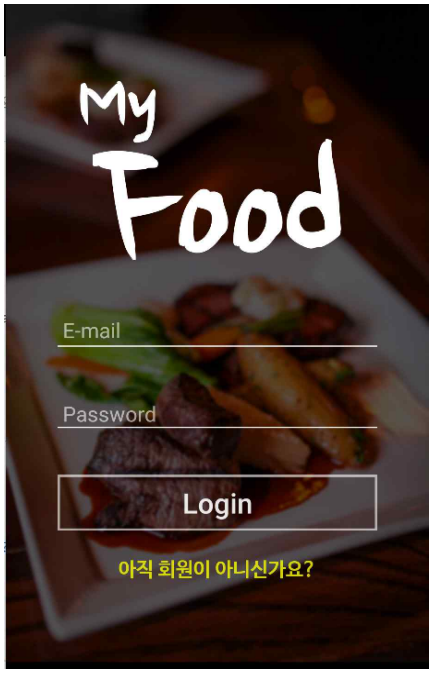
<UI/UX>

- 전문지식 및 신규습득
  - 레이아웃의 가중치를 활용해 구조를 나누는 것은 알았으나 margin과 padding을 적용하는 방법을 Material Design을 통해 새롭게 알게되었음
  - 레이아웃의 배경에 색상과 알파값을 정하는 것을 포토샵의 색상추출기능으로 해왔었음. 하지만 프로젝트 도중에 새롭게 안드로이드 내에서도 색상을 변경할 수 있다는 것을 알게되었음
  - 기존엔 Fragment로 드로어를 만드는 것을 알았다면 이번엔 라이브러리를 적용하여 활용하는 방법을 알게되었음

- 실무적용
  - 기본적으로 Material Design의 가이드라인은 본적이 있었지만, 실제적으로 앱에 적용해본 것은 이번이 처음이었음. 레이아웃과 그 외의 효과등을 효율적으로 적용하는데에 조금 어려움이 따랐음.
  - 레이아웃의 배치와 상하관계등에 대해서 android 표준문서를 많이 참고하였음

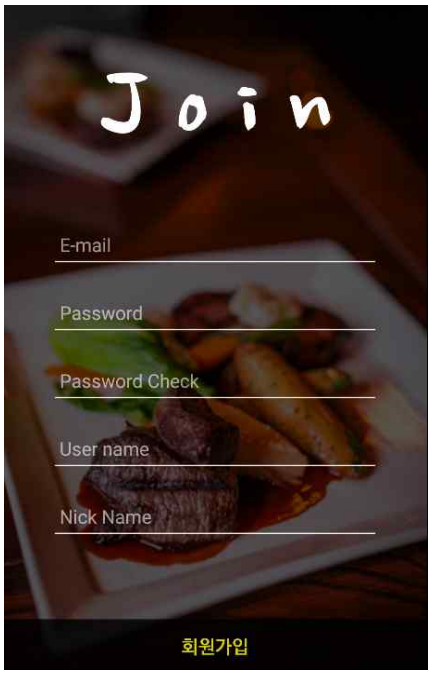
다. 소스코드 설명

o 주요 코딩정보 설명



[그림 10] 로그인 화면

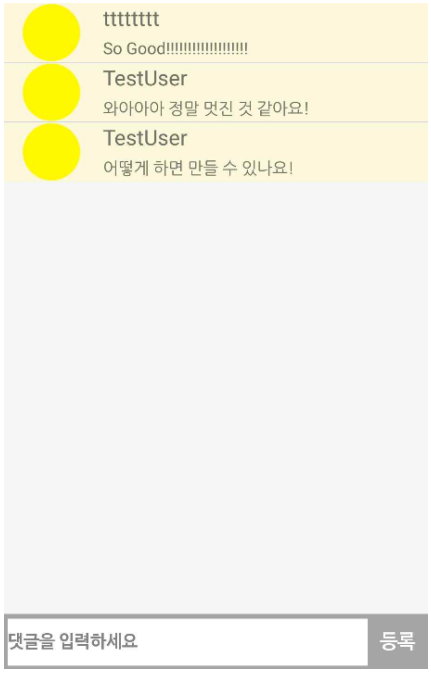
- 프로젝트 로고와 함께 애플리케이션 실행
- E-mail과 Password를 메인서버로 전송하고 AOuth 토큰을 수신



[그림 11] 회원가입 화면

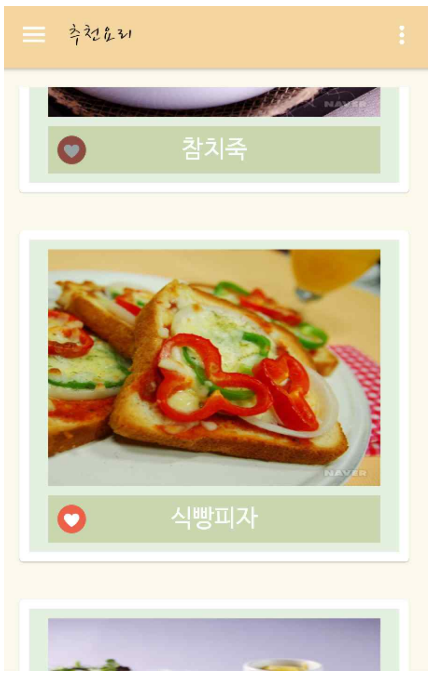
- 회원가입 시 필요한 정보를 받아서 메인서버로 전송
- 정보가 잘못되었을 시 Snackbar를 통한 알림

[그림 12] 레시피 뷰



[그림 13] 리뷰 뷰

- 레시피 뷰에서는 레시피 사진을 가로 스크롤 형태로 보여줌
- 리뷰 목록을 보면 타 사용자들이 남긴 리뷰를 확인 할 수 있음,
- 레시피를 조회하면 조회기록이 서버로 전송되어 추후 개인화 추천에 반영됨



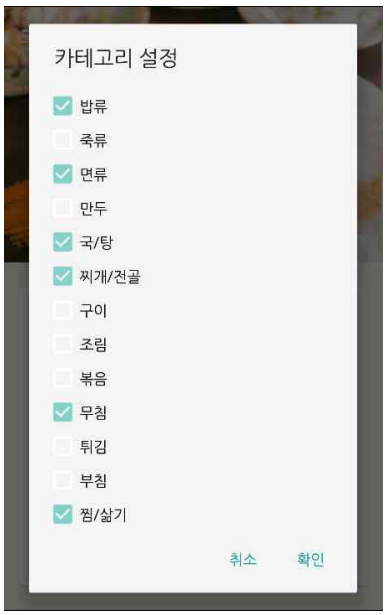
[그림 14] Like 버튼

- Like 버튼을 누르면 하트가 활성화됨
- Like 이벤트는 서버에 전송되어 추후 개인화 추천에 반영됨

[그림 15] 추천받은 목록

- 피자와 파스타류를 View하고 Like했을 때, 개인화 추천 레시피 최상단에 “올리브 스파게티”를 추천받음

[그림 16] 식감설정

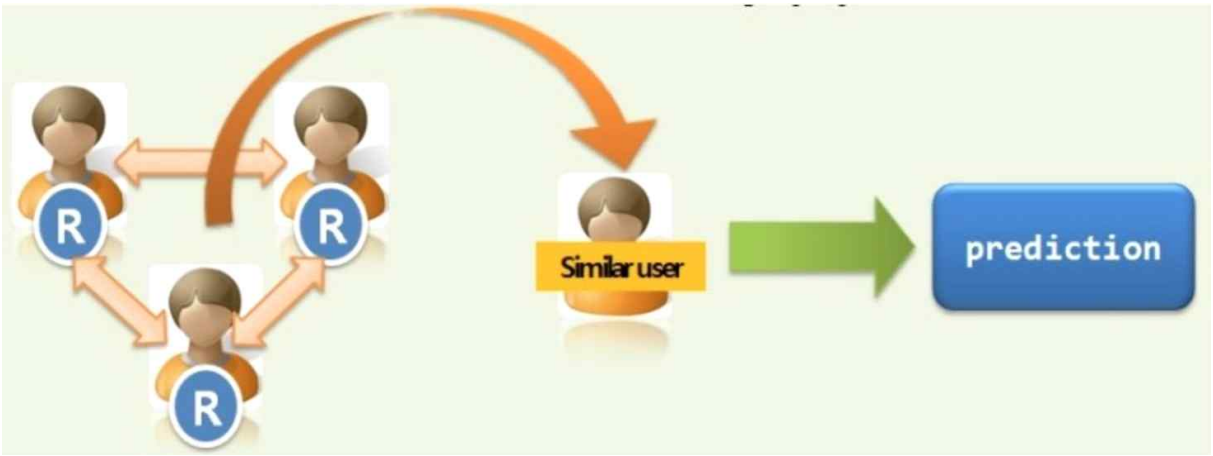


[그림 17] 카테고리 설정

- 식감설정과 카테고리 설정을 통해 보다 사용자가 원하는 레시피를 필터링 할 수 있는 기능을 제공

<ML서버>

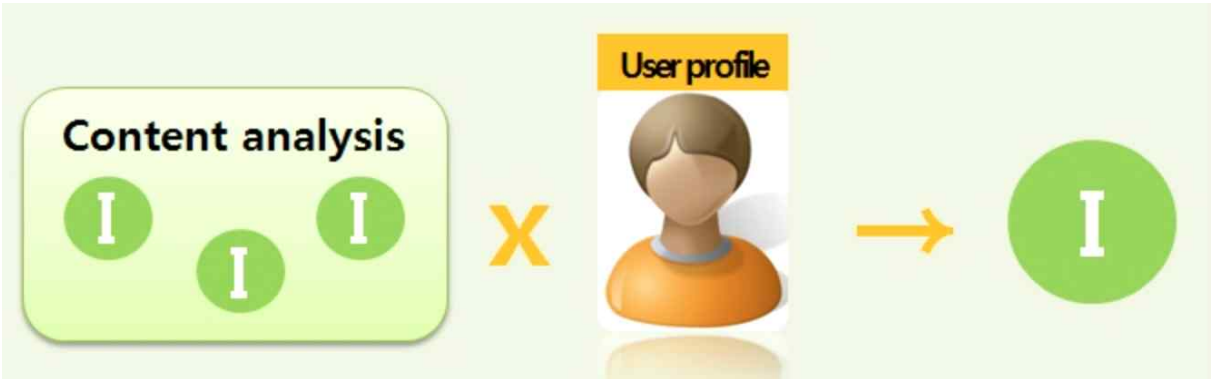
- **Collaborative Filtering Algorithm과 Content-Based Algorithm을 이용하였음**
- **Collaborative Filtering Algorithm:** 사회적 추천 알고리즘
  - 사용자와 유사한 취향을 가지고 있는 사용자의 과거 아이템 rating을 기반으로 추천
  - 필터링을 하기 위해서는 rating이 높은 item과 neighborhood creation과 ratings prediction 이 필요함



1) 사용자들 사이의 rating을 비교
2) 현 사용자와 유사한 사용자를 user-user similarity를 이용하여 선택
   (이 유사한 사용자들이 neighborhood creation)
3) 현 사용자에게 추천해주기 위한 몇 개의 item에 대해 rating을 예측
   (neighborhood user의 rating을 기반으로)
4) 그 다음으로 best로 rank된 아이템을 사용자에게 추천해줌

  - 단점: sparseness & cold start
    - CF는 이전 사용자들에 대한 정보를 이용해 계산하기 때문에 충분한 양의 rating 정보가 없을 시 성능이 현저히 떨어짐
    - Cold-start problem: 시스템 초기나 새로운 사용자들에게 item을 추천할 경우 정확도가 매우 낮은 문제

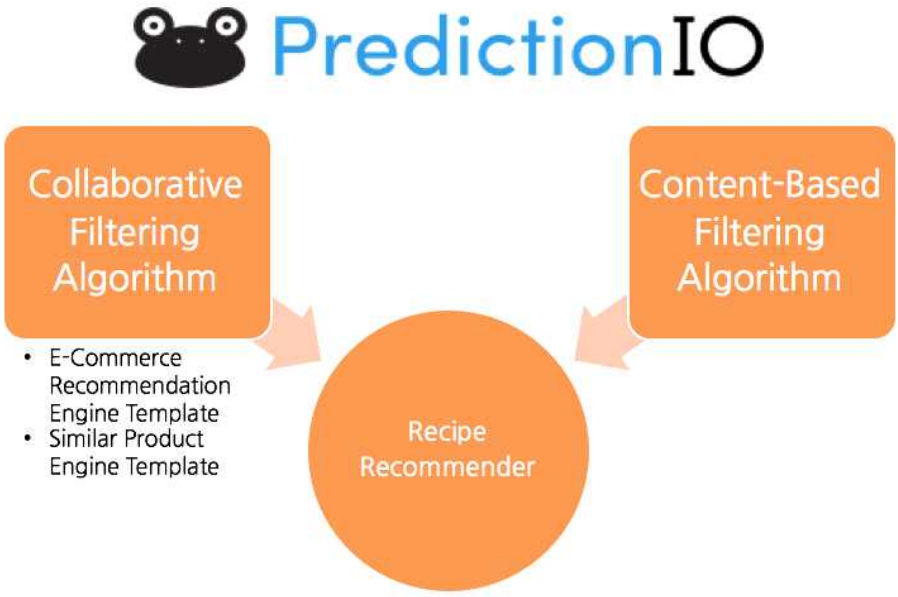
- **Content-Based Filtering Algorithm:** 내용 기반 추천 알고리즘
  - 각 item에 대한 속성 등의 item content 정보를 이용하는 방법
  - 사용자의 과거 경험에서 선호도, 취향 등의 사용자 정보를 어떻게 찾아내는가가 핵심



1) 먼저 알려진 선호 item들의 집합에 대해 내용을 분석
2) 사용자의 선호도를 나타내는 user-profile 생성
3) item과 user-profile 사이의 유사도를 계산
4) user가 선호할 item을 예측

-장점: 구현이 간단하고, 사용자의 선호정보를 직접적으로 반영 가능
-단점: 아이템의 content-information을 구하기 어렵고, 사용자의 명시적 profile을 얻기 힘듬

- **Recipe Recommender: PredictionIO의 'E-Commerce Recommendation Engine Template'과 'Similar Product Engine Template'을 결합, 수정, 보완하여 구현한 Collaborative Filtering Algorithm과 이를 보완하기 위해 Scala기반으로 구현한 Content-Based Filtering Algorithm을 합친 Recommender**
  - 하나의 쿼리에 두 알고리즘이 동시에 수행되어 나온 각각의 결과값을 자동으로 합산, 반환
  - Collaborative Filtering Recommender
    - 기존 유저의 최근 행동 패턴을 기반으로 비슷한(선호할만한) 아이템을 추천
    - 기존 유저의 행동 패턴과 비슷한 아이템이 없을 경우 대중적으로 인기있는 아이템을 추천
    - 새로운 유저에게는 대중적으로 인기있는 아이템을 추천
    - 보지 않은 아이템만 추천하는 기능 (Optional)
    - 카테고리, 식감, 화이트리스트, 블랙리스트 필터링 기능 (Optional)
    - 페이징 기능
  - Content-Based Filtering Recommender
    - 기존 유저가 최근 'view' 또는 'like'를 한 아이템의 attributes와 비슷한 아이템을 추천
    - 보지 않은 아이템만 추천하는 기능 (Optional)
    - 카테고리, 식감, 화이트리스트, 블랙리스트 필터링 기능 (Optional)
    - 페이징 기능



- 소스코드 주요부분 설명

  - Engine.scala

```
case class Query(
  user: String,
  limit: Int,
  skip: Int,
  categories: Option[Set[String]],
  feelings: Option[Set[String]],
  whiteList: Option[Set[String]],
  blackList: Option[Set[String]]
) extends Serializable
```

  - 쿼리: 추천받을 User id와 레시피 추천받을 시작 num(skip), 끝 num(limit) 기본, categories, whitelist, blacklist를 통한 필터링은 옵션

```
case class PredictedResult(
  itemScores: Array[ItemScore]
) extends Serializable

case class ItemScore(
  item: String,
  score: Double
  ) extends Serializable with Ordered[ItemScore] {
  def compare(that: ItemScore) = this.score.compare(that.score)
}
```

  - 각 레시피(item)별로 score를 저장한 뒤 점수순으로 나열되어 반환

- DataSource.scala

```
case class User()

case class Item(
  item: String,
  title: String,
  categories: Array[String],
  categories2: Option[List[String]],
  feelings: Array[String],
  feelings2: Option[List[String]],
  cooktime: Int,
  calories: Int,
  expire: Int
)

case class ViewEvent(
  user: String,
  item: String,
  t: Long
)

case class LikeEvent(
  user: String,
  item: String,
  t: Long,
  like: Boolean // true: like. false: cancel_like
)

class TrainingData(
  val users: RDD[(String, User)],
  val items: RDD[(String, Item)],
  val viewEvents: RDD[ViewEvent],
  val likeEvents: RDD[LikeEvent]
) extends Serializable {
  override def toString = {
    s"users: [${users.count()} (${users.take(2).toList}...)]" +
    s"items: [${items.count()} (${items.take(2).toList}...)]" +
    s"viewEvents: [${viewEvents.count()}] (${viewEvents.take(2).toList}...)" +
    s"likeEvents: [${likeEvents.count()}] (${likeEvents.take(2).toList}...)"
  }
}
```

  - users, items, viewEvents, likeEvents RDD 생성
  - title: 레시피이름
    categories: 음식 카테고리
    feelings: 맛, 식감
    cooktime: 조리시간
    calories: 칼로리/1인분
    expire: 보관기간
  - viewEvents: 레시피 클릭
  - likeEvents: 레시피 좋아요 클릭 혹은 클릭 해제

- RecipeAlgorithm.scala

```
case class RecipeAlgorithmParams(
  appName: String,
  unseenOnly: Boolean,
  seenEvents: List[String],
  similarEvents: List[String],
  rank: Int,
  numIterations: Int,
  lambda: Double,
  seed: Option[Long],
  dimensions: Int,
  cooktimeWeight: Double,
  caloriesWeight: Double,
  expireWeight: Double,
  normalizeProjection: Boolean
) extends Params
```

- 알고리즘 초기설정 파라미터
- appName: PredictionIO 앱 이름
  unseenOnly: unsee 이벤트만 보여줌
  seenEvents: 유저가 본 이벤트의 user-to-item 리스트, unseenOnly가 true일 때 쓰임
  similarEvents: 비슷한 이벤트의 user-item-item 리스트, 유저가 최근에 본 item과 비슷한 item을 찾을 때 쓰임
  rank: MLlib ALS 알고리즘의 파라미터. Number of latent feature.
  numIterations: MLlib ALS 알고리즘의 파라미터. Number of iterations.
  lambda: MLlib ALS 알고리즘의 정규화 파라미터
  seed: MLlib ALS 알고리즘의 random seed. (Optional)
  dimension: 벡터화 된 아이템의 차원 수
  cooktimeWeight: 조리시간의 가중치
  caloriesWeight: 칼로리의 가중치
  expireWeight: 보관기간의 가중치
  normalizeProjection: projection 표준화

```
case class RecipeModel(
  item: Item,
  features: Option[Array[Double]], // features by ALS
  count: Int // popular count for default score
)
```

- 레시피 모델
- item: 레시피
  features: ALS 알고리즘으로 계산된 score
  count: similar product가 없을 때 trainDefault()에 의해 반환된 popular count score

```
class RecipeAlgorithmModel(
  val rank: Int,
  val userFeatures: Map[Int, Array[Double]],
  val recipeModels: Map[Int, RecipeModel],
  val userStringIntMap: BiMap[String, Int],
  val itemStringIntMap: BiMap[String, Int],
  val itemIds: BiMap[String, Int],
  val projection: DenseMatrix
) extends Serializable {
```

- 레시피 알고리즘 모델

  - rank: MLlib ALS 알고리즘의 파라미터. Number of latent feature.
    userFeatures: 유저의 최근 행동 기록
    recipeModels: 레시피 모델(item, features, count)
    userStringIntMap: 유저String을 Int로 Mapping
    itemStringIntMap: 아이템String을 Int로 Mapping
    itemIds: 아이템id
    projection: projection 매트릭스

```
def predictSimilar(
  recentFeatures: Set[Array[Double]],
  recipeModels: Map[Int, RecipeModel],
  query: Query,
  whiteList: Option[Set[Int]],
  blackList: Set[Int]
): Array[(Int, Double)] = {
  val indexScores: Map[Int, Double] = recipeModels.par // parallel collection 으로 변환
    .filter { case (i, pm) =>
      pm.features.isDefined &&
      isCandidateItem(
        i = i,
        item = pm.item,
        categories = query.categories,
        feelings = query.feelings,
        whiteList = whiteList,
        blackList = blackList
      )
    }
    .map { case (i, pm) =>
      val s = recentFeatures.map{ rf =>
        // 위의 filter logic을 위해 pm.features가 정의되어 있어야 함
        cosine(rf, pm.features.get)
      }.reduce(_ + _)

      (i, s)
    }
    .filter(_._2 > 0) // score > 0 인 item들만 keep
    .seq // sequential collection으로 다시 변환

  val ord = Ordering.by[(Int, Double), Double](_._2).reverse
  val topScores = getTopN(indexScores, query.limit)(ord).toArray

  topScores
}
```

- Collaborative Filtering Algorithm Prediction
- User에 대한 정보가 있을 때 수행함
- User가 최근 10개의 action을 취한 item들을 기반으로 top similar item을 prediction

```
def predictDefault(
  recipeModels: Map[Int, RecipeModel],
  query: Query,
  whiteList: Option[Set[Int]],
  blackList: Set[Int]
): Array[(Int, Double)] = {
  val indexScores: Map[Int, Double] = recipeModels.par // sequential collection으로 다시 변환
    .filter { case (i, pm) =>
      isCandidateItem(
        i = i,
        item = pm.item,
        categories = query.categories,
        feelings = query.feelings,
        whiteList = whiteList,
        blackList = blackList
      )
    }
    .map { case (i, pm) =>

      (i, pm.count.toDouble)
    }
    .seq

  val ord = Ordering.by[(Int, Double), Double](_._2).reverse
  val topScores = getTopN(indexScores, query.limit)(ord).toArray

  topScores
}
```

- Default Prediction
- User에 대한 정보가 없을 때 수행함
- 다른 user들의 ‘view’와 ‘like’ event를 기준으로 high score item을 계산하여 반환

```
def predictContentBased(
  recentItems: Set[String],
  model: RecipeAlgorithmModel,
  recipeModels: Map[Int, RecipeModel],
  query: Query,
  whiteList: Option[Set[Int]],
  blackList: Set[Int]
): Array[ItemScore] = {
  val result = recentItems.flatMap { itemId =>
    model.itemIds.get(itemId).map { j =>
      val d = for(i <- 0 until model.projection.numRows) yield model.projection(i, j)
      val col = model.projection.transpose.multiply(new DenseVector(d.toArray))
      for(k <- 0 until col.size) yield new ItemScore(model.itemIds.inverse
        .getOrElse(k, default="NA"), col(k))
    }.getOrElse(Seq())
  }.groupBy {
    case(ItemScore(itemId, _)) => itemId
  }.map(_._2.max).filter {
    case(ItemScore(itemId, _)) => !recentItems.contains(itemId)
  }.filter { case(ItemScore(itemId, _)) =>
    isCandidateItem(
      i = model.itemStringIntMap(itemId),
      item = recipeModels.get(model.itemStringIntMap(itemId)).get.item,
      categories = query.categories,
      feelings = query.feelings,
      whiteList = whiteList,
      blackList = blackList)
  }
  .toArray.sorted.reverse.take(query.limit)

  result
}
```

- Content-Based Filtering Algorithm Prediction
- 모든 item에 대해 user의 recentItems와의 유사도를 측정하여 높은 score를 가진 item 순으로 나열함

- Serving.scala

```
class Serving
  extends LServing[Query, PredictedResult] {

  override
  def serve(query: Query,
    predictedResults: Seq[PredictedResult]): PredictedResult = {

    @transient lazy val logger = Logger[this.type]

    // 같은 item에 대해 Collaborative Filtering Recommender의 결과값과
    // Content Based Filtering Recommender의 결과값을 합산, 정렬함
    val combined = predictedResults.map(_.itemScores).flatten // ItemScore 배열
      .groupBy(_.item) // 같은 item id끼리 묶음
      .mapValues(itemScores => itemScores.map(_.score).reduce(_ + _))
      .toArray // (item id, score) 배열
      .sortBy(_._2)(Ordering.Double.reverse)
      .slice(query.skip, query.limit)
      .map { case (k,v) => ItemScore(k, v) }

    if (!combined.isEmpty) logger.info(s"Recommendation result for user ${query.user} is successfully sent to the user.")

    new PredictedResult(combined)
  }
```

- 같은 item에 대해 Collaborative Filtering Algorithm의 결과값과 Content-Based Filtering Algorithm의 결과값을 합산, 정렬함
- 이렇게 합산하여 나온 추천 리스트 결과값을 최종 반환

<RESTful API 구축 과정>

Node.js의 Sails 프레임워크 이용하여 구현함.

```
module.exports = {
  /**
   * Recipe RESTful API Get
   * API Route: GET /recipes?skip=0&limit=30&where={}&sort=id ASC
   */
  find: find,

  /**
   * Recipe RESTful API Get One
   * API Route: GET /recipes/:recipeId
   */
  findOne: findOne,

  /**
   * Recipe RESTful API Get associations
   * API Route: GET /recipes/:recipeId/reviews
   */
  findReviews: findReviews,
};
```

```
///
/// Recipe
///
'post /recipes/:recipe/feelings/:id': 'FeelingController.addRecipe',
'delete /recipes/:recipe/feelings/:id': 'FeelingController.removeRecipe',

///
/// Like
///
'post /recipes/:id/likes': 'LikeController.create',
'delete /recipes/:id/likes': 'LikeController.destroy',

///
/// View
///
'post /recipes/:id/views': 'ViewController.create',
'delete /recipes/:id/views': 'ViewController.destroy',

///
/// Review
///
'get /recipes/:id/reviews': 'RecipeController.findReviews',

///
/// Feeling
///
'post /feelings/:id/recipes/:recipe': 'FeelingController.addRecipe',
'delete /feelings/:id/recipes/:recipe': 'FeelingController.removeRecipe',

///
/// Prediction
///
'get /predictions': 'PredictionController.find',
```

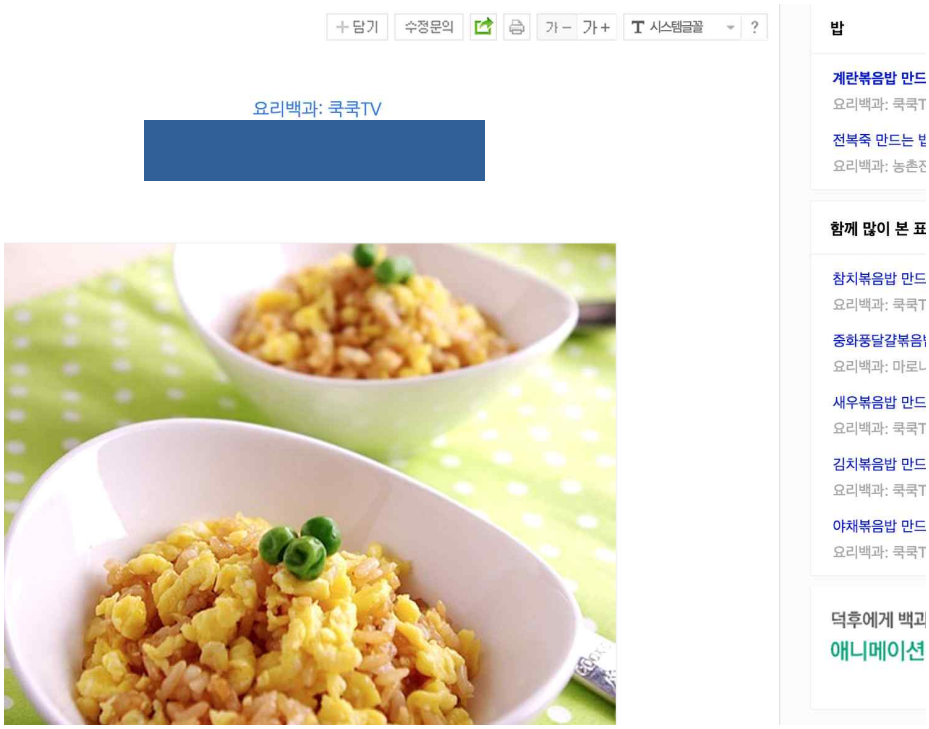
라우트에 컨트롤러 함수를 매핑시키는 방식임. 또 특별히 구현하지 않았을 경우 기본 모델을 참조하여 RESTful API를 제공함.

```
function find(req, res) {
    async.waterfall([
        // 추천 정보를 가져옴
        bringRecommendation,

        // 추천 정보와 레시피를 연결함
        matchRecipe,
    ], serviceUtil.response(req, res));

    function bringRecommendation(cb) {
        var user = req.user;
        if (!user) {
            return res.forbidden();
        }

        var pioRecipe = Pio.getClient('myRecipe');

        // display options
        var criteria = {
            skip    : parseInt(req.param('skip'))   || 0,
            limit   : parseInt(req.param('limit'))  || 15,
            user    : user.id,
        };
        criteria.limit += criteria.skip;

        // conditonal filters
        var where = req.param('where');
        if (where) {
            try {
                where = JSON.parse(where);
            }
            catch (e) {
                return cb(e);
            }

            if (where.feelings) {
                criteria.feelings = where.feelings;
            }

            if (where.categories) {
                criteria.categories = where.categories;
```

RESTful API 인증은 OAuth 2.0(Bearer) 표준을 준수하는 Passport 모듈을 이용해 해결함. Node.js에서 발생하는 Callback Hell은 async 모듈을 이용하여 최대한 해소하려 노력함.

<레시피 크롤링 과정>

웹 문서를 분석하기 위해서 Python의 Scrapy라는 프레임워크를 통해 구현함. 웹 페이지는 모두 Node로 이루어져 있음. 따라서 원하는 정보가 존재하는 Node의 위치를 알면 해당 정보만 조회하는 것이 가능함. Scrapy의 경우 XPath와 CSS Selector를 지원함. XPath는 노드를 윈도우 탐색기를 사용하듯이 조회할 수 있음.



위 사진에서 나오는 제목을 가져온 코드는 다음과 같음.

```
# 제목
recipe['title'] = response.xpath('//*[@id="content"]/div[2]/div[1]/h2/text()').extract()
```

제목 노드를 찾아낸 뒤 해당 노드의 텍스트를 추출함. 같은 방식으로 다른 내용들도 추출이 가능함. 그런데 여기에 문제가 있음. 문서마다 들어가는 내용이 다름. 노드 위치가 항상 일정하지가 않아서 레시피마다 다른 경로를 가질 수 있음. 해결하기 위해서 살짝 고급기법을 적용해야 함.

```
# 요리재료
recipe['ingredient'] = response.xpath((
    u"//h4[contains(text(), '요리재료')]"
    u'/following-sibling::h4[1]'
    u"/preceding-sibling::p[preceding-sibling::h4[contains(text(), '요리재료')]]"
)).extract()
```

단순히 노드의 경로를 적는 것이 아니라 노드가 존재하는 상대적 위치를 이용하여 그 사이의 내용을 전부 가져올 수 있음. 이렇게 레시피에 필요한 모든 정보를 추출할 수 있음. 그러나 모든 레시피의 주소를 사용자가 입력하여 사용자가 크롤러를 돌리기는 힘들 것임. 해결 방법으로 레시피의 목록을 순회하여 레시피를 분석하는 방법이 있음.

레시피 목록에는 레시피로 접속하는 링크와 레시피의 카테고리들이 있음. 따라서 해당 레시피의 링크를 통해 문서를 다운로드 한 뒤 분석하면 되는 것임.

페이지를 쉽게 이동할 수 있도록 제공하는 UI를 이용해 마찬가지로 계속해서 순회하면 결국에는 모든 레시피를 조회할 수 있게 됨.

추출한 데이터는 로컬에 저장하고 서버에 저장하지 않았기 때문에 별도로 서버에 저장하는 작업을 해야함. RESTful API를 이용해 서버에 저장했음.

```
function createRecipes(recipes) {
    async.eachLimit(recipes, 10, function (recipe, cb) {
        var thumb = recipe.thumb,
            method = recipe.method,
            methodThumb = recipe.methodThumb;

        delete recipe.thumb;
        delete recipe.methodThumbs;

        async.waterfall([
            createThumb,
            createRecipe,
            createMethodThumbs,
        ], done);
```

동시에 너무 많은 커넥션이 발생하면 에러가 발생함. 따라서 동시에 실행되는 작업을 제한할 필요가 있음.

<식감 추출 과정>

레시피의 식감을 추출하기 위해 레시피에 대한 사람들의 생각을 조사할 필요가 있음. 모든 사람들에게 설문조사하는 방법은 무리수임. 네이버 검색 API를 통해 해당 레시피 제목으로 검색했을 때 사용자들의 반응을 검사할 것임.

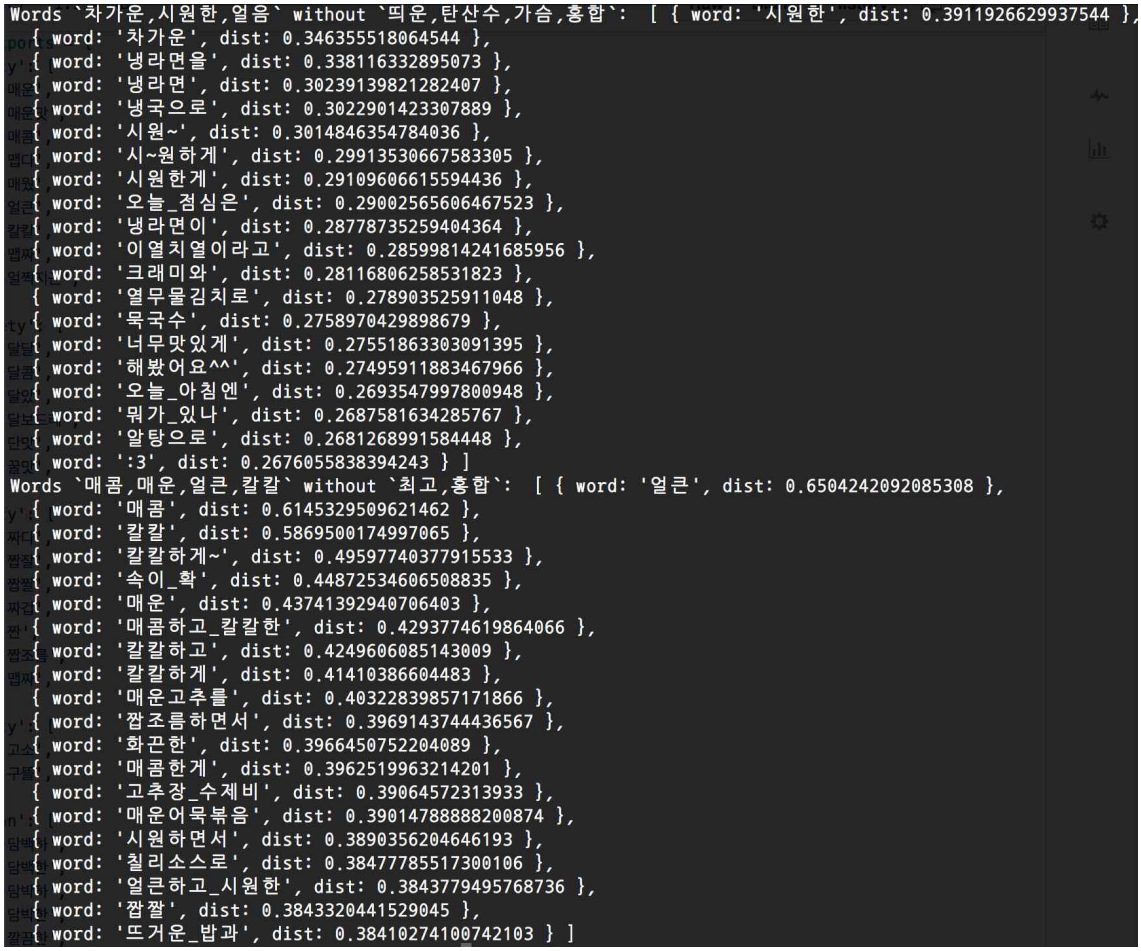
위는 '돼지등뼈찜'의 검색결과임. 내용 중 아래처럼 식감에 대한 언급이 있음.

'매콤'이라는 단어를 발견하면 아래 코드처럼 식감 단어들을 카운트하고 각각 점수를 부여함. 해당 개념의 식감 단어 / 전체 발견된 식감 단어 라는 단순한 식임. 부정어구에 대한 처리는 해두지 않았지만 사람들은 웬만하면 긍정문으로 말함. 해당 문제에 대한 노이즈는 배제함.

```
// 사전에 등록된 단어를 검색하고 저장한다.
for (var feeling in feelingList) {
    var feelingDict = feelingList[feeling];
    foundWords[feeling] = 0;

    feelingDict.forEach(findWord(feeling));
}

// 레시피 식감 저장 배열
feelings[item.id] = [];

for (feeling in foundWords) {
    var importance = foundWords[feeling] / allWords;

    // 우선 순위 저장
    if (!highest || highest < importance) {
        highest = importance;
        highestIdx = feeling;
    }

    // 기준치 이상의 중요도를 가진다면 등록
    if (importance >= importanceHurdle) {
        feelings[item.id].push(feeling);
    }
}

// 만약 추출된 식감이 없다면 가장 높은 항목을 식감으로 지정
if (!feelings[item.id].length) {
    feelings[item.id].push(highestIdx);
}
```

```
module.exports = {
    'spicy': [
        '매운',
        '매운맛',
        '매콤',
        '맵다',
        '매웠',
        '얼큰',
        '칼칼',
        '맵짜',
        '얼쩍지근',
    ],
    'sweety': [
        '달달',
        '달콤',
        '달았',
        '달보드레',
        '단맛',
        '꿀맛',
    ],
    'salty': [
        '짜다',
        '짭잘',
        '짭짤',
        '짜겁',
        '짠',
        '짭조름',
        '맵짜',
    ],
    'fatty': [
        '고소',
        '구뜰',
    ],
    'clean': [
        '담백하',
        '담백한',
```

식감 사전을 구축하는 것에 귀찮음을 자각. Word2vec이라고 불리는 기계 학습 알고리즘을 통해 사전을 구축하는 것을 시도해보았음.



그러나 생각보다 결과가 좋지 않아 실제 식감 추출에는 적용할 수 없었음.

추출된 식감 또한 앞에 레시피 크롤링 과정처럼 RESTful API를 이용해 서버에 저장함.