모델 - logic을 어떻게 세울지
table끼리 어떻게 엮을지
api형식을 어떻게 만들것인가?

컨트롤러 - action 받아서 던져주는것

repo → db에서 받아오는 것.

service → controller + model + repo

기본구조

code -
message -
data -

controller - 앞전에 받은 응답을 분석.
판단해서
Response 만들어서 보냄?

model - business 데이터베이스랑 매핑
최적화?

repo - model = db.
이렇게 복잡한 수도 있도록 최대한 mapping 해줄

service - Service - impl에서 쓸 interface 작성

serviceImpl - service에 있는 클래스 작성.

Business - Controller.

- controller

@RestController – HTTP 요청을 처리하고 JSON 데이터 반환하는 역할.
이게 적용된 클래스의 메서드들은 각각의 엔드포인트에 매핑되어 요청을 처리하고, 데이터를 반환.
REST API를 제공하기 위한 컨트롤러 역할을 수행, 요청을 처리하고 응답을 반환

@RequiredArgsConstructor – 해당 클래스의 모든 필수 인자를 가진 생성자를 자동으로 생성해주는 기능 제공
생성자 주입

@GetMapping("/api/business") → GET은 사업목록을 받아와서 보여줘야하기 때문에?
public ResponseEntity<?> myBusinessList(@AuthenticationPrincipal CustomUserDetails user)
Spring Security에서 제공하는 어노테이션
현재 사용자의 인증정보를 쉽게 가져오기 위해 사용

user/{num} @PathVariable

return new ResponseEntity<>

HTTP GET 요청에 대한 처리를 지원하는 Spring Framework의 어노테이션에는 다음과 같은 종류가 있습니다:

1. `@GetMapping`: HTTP GET 요청에 대한 매핑을 지원하는 어노테이션으로, 특정 경로에 대한 GET 요청을 처리합니다.
2. `@RequestMapping(method = RequestMethod.GET)`: 모든 HTTP 메서드에 대한 매핑을 지원하는 범용적인 어노테이션 중 하나입니다. `method` 속성을 `RequestMethod.GET`으로 설정하여 GET 요청에 대한 처리를 지정할 수 있습니다.
3. `@PathVariable`: 경로 변수를 사용하여 동적인 경로 매개변수를 추출합니다. `@GetMapping` 또는 `@RequestMapping`과 함께 사용하여 경로 변수를 활용할 수 있습니다.
4. `@RequestParam`: 쿼리 파라미터를 추출합니다. 요청 URL의 쿼리 문자열에서 파라미터를 가져올 때 사용됩니다.
5. `@RequestHeader`: 요청 헤더에서 값을 추출합니다. 특정 헤더 필드의 값을 가져올 때 사용됩니다.

이 외에도 Spring Framework는 다양한 HTTP 요청 메서드를 처리하기 위한 어노테이션을 제공합니다. 예를 들면, `@PostMapping`, `@PutMapping`, `@DeleteMapping` 등이 있으며, 각각 POST, PUT, DELETE 메서드에 대한 매핑을 지원합니다. 이렇게 HTTP 요청 메서드별로 특화된 어노테이션을 사용하여 각각의 요청을 처리할 수 있습니다.

Spring Framework에서는 HTTP 요청에 대한 응답을 처리하기 위해 여러 가지 방법을 제공합니다. 몇 가지 주요한 방법은 다음과 같습니다:

1. `ResponseEntity`: 이미 이전에 언급한 `ResponseEntity<?>`를 사용하여 응답을 처리할 수 있습니다. 이 클래스를 사용하면 상태 코드, 응답 본문 데이터, 응답 헤더 등을 포함한 HTTP 응답을 구성할 수 있습니다.
2. `@ResponseBody`: 컨트롤러의 메서드에 `@ResponseBody` 어노테이션을 사용하면 메서드가 반환하는 객체가 HTTP 응답 본문으로 직접 전송됩니다. 이 방식은 객체를 자동으로 직렬화하여 응답 본문으로 변환하고, 적절한 Content-Type 헤더를 설정합니다.
3. `@RestController`: `@RestController` 어노테이션이 적용된 클래스는 JSON 또는 XML과 같은 형식의 데이터를 반환하는 RESTful 웹 서비스 엔드포인트로 사용됩니다. 이 어노테이션은 `@Controller`와 `@ResponseBody`를 결합한 것으로, 컨트롤러 클래스 내의 모든 메서드가 응답 본문으로 직접 전송됩니다.
4. `HttpServletResponse`: 서블릿 환경에서 직접 `HttpServletResponse` 객체를 활용하여 응답을 처리할 수 있습니다. 이 객체를 사용하면 상태 코드, 응답 헤더 및 응답 본문을 설정할 수 있습니다.
5. `StreamingResponseBody`: 대용량 데이터나 실시간 스트리밍 데이터를 처리해야 할 때 사용됩니다. `StreamingResponseBody`는 응답 본문을 스트리밍으로 처리할 수 있는 기능을 제공합니다.

위의 방법들은 주로 Spring Framework에서 HTTP 요청에 대한 응답을 처리하는 방법 중 일부입니다. 선택할 방법은 요구사항과 프로젝트의 특성에 따라 다를 수 있으며, 각 방법은 각자의 장단점과 사용 시나리오가 있습니다.

질문.

질문 1. business controller부분에서 line 24을 작성하게 되면 line 26은 자동으로 생성이 되는 건가요??

(생성자주입) = 의존성주입.

lombok(롬복) 사용.

@RequiredArgsConstructor 을 작성해주면

public class ~ {

클래스에서 생성자를 따로 작성해주지 않고

private final BusinessService service;

}

다른 클래스 선언하고 사용할수 있다.

의존성주입의 종류로는 `Constructor(생성자)`, `Setter`, `Field` 타입이 있다.

1. Constructor(생성자)

```
public  class  ExampleCase {

    private  final  ChocolateService  chocolateService;
    private  final  DrinkService  drinkService;

    @Autowired
    public ExampleCase(ChocolateService  chocolateService, DrinkService  drinkService)
    this.chocolateService = chocolateService;
    this.drinkService = drinkService;
    }
}
```

2. Setter

```
public  class  ExampleCase{

        private  ChocolateService  chocolateService;
        private  DrinkService  drinkService;

        @Autowired
        public void  setChocolateService(ChocolateService  chocolateService){
            this.chocolateService = chocolateService;
        }

        @Autowired
        public void  setDrinkService(DrinkService  drinkService){
            this.drinkService = drinkService;
        }

    }
```

3.Field

```
public  class  ExampleCase{

        @Autowired
        private ChocolateService  chocolateService;

        @Autowired
        private DrinkService  drinkService;
    }
```

@RequiredArgsConstructor 어노테이션을 사용한 생성자 주입 방법

생성자주입의 단점은 위의 Constructor(생성자) 코드처럼 생성자를 만들기 번거롭다는 것이다. 하지만 이를 보완하기위해 `롬복`을 사용하여 간단한 방법으로 생성자 주입 방식의 코딩을 할 수 있다.

@RequiredArgsConstructor

`final`이 붙거나 `@NotNull` 이 붙은 필드의 생성자를 자동 생성해주는 롬복 어노테이션

필드 주입방식을 사용한 기존 Service

```
@Service
    public class BannerServiceImpl implements BannerService {

        @Autowired
        private BannerRepository bannerRepository;

        @Autowired
        private CommonFileUtils commonFileUtils;
```

@RequiredArgsConstructor 를 활용한 생성자 주입

```
    @Service
    @RequiredArgsConstructor
    public class BannerServiceImpl implements BannerService {

        private final BannerRepository bannerRepository;

        private final CommonFileUtils commonFileUtils;
        ...
```

@RequiredArgsConstructor를 사용하지 않으면 원래는 이렇게 생성자 주입을 해야한다

```
@Service
public class BannerServiceImpl implements BannerService {

    private BannerRepository bannerRepository;

    private CommonFileUtils commonFileUtils;

    @Autowired
    public BannerServiceImpl(BannerRepository bannerRepository, CommonFileUtils co
        this.bannerRepository = bannerRepository;
        this.commonFileUtils = commonFileUtils;
    }
    ...
```

질문 2. GET /user/1 을 코드로 작성하면 @GetMapping("/user/1") 이렇게 쓰는 건가요??

NO.

@GetMapping ("/user/{num}") 이렇게 특정한게 아닌 전체가 사용할수있게 만들고 를 가져와야한다.

그리고

1. public ResponseEntity<?> myBusinessList (@PathVariable(value="~") ~)
String id.

2. @RequestParam(value="param1", required=true) String param1,

RequestParam

먼저 requestParam을 보면 4가지 파라미터를 가지고있습니다.

- **defaultValue** - 값이 없을때 기본으로 전달할 값
- **name** - uri에서 바인딩할 파라미터의 이름
- **value** - uri에서 바인딩하여 별칭으로 정할 값
- **required** - 필수적으로 값이 전달되어져야 하는 파라미터. 이게 없다면 에러가 뜰것.

```
public String get(
    @RequestParam(value="param1", required=true) String param1,
    @RequestParam(value="param2", required=false) String param2){
...
}
```

PathVariable

PathVariable 에서 알아야 할것 은 딱 하나입니다. 어떤 요청이든간에 하나밖에 못쓰는겁니다. 아래와 같이 id 하나만 설정할 수 있겠습니다. 이것을 인지한 상태로 작업을 해야합니다.

```
@RequestMapping("/hello/{id}")
public String getDetails(@PathVariable(value="id") String id){
.....
}
```

굳이 PathVarible 방식으로 여러 파라미터를 보내고 싶다면 비슷한 방법으로 **@MatrixVariable** 라는 메서드가 있으니 이것도 한번 참고해보시길 바랍니다. (여기)
이는 uri에 맵으로 데이터를 보내는 방식이다.

RequestParam & PathVariable

그리고 두가지를 혼합하여 사용하는 경우도 있는데

```
@RequestMapping("/hello/{id}")
public String getDetails(@PathVariable(value="id") String id,
    @RequestParam(value="param1", required=true) String param1,
    @RequestParam(value="param2", required=false) String param2){
.....
}
```

이와 같이 코드를 작성하면 id값에 따라 여러가지 파라미터를 주고받을수 있으며 이것을 통해 다양한 시도를 할수 있을것 같습니다.

질문 3. line 32에서 @AuthenticationPrincipal 이 부분이 인증된 사용자의 정보를 주입받을 수 있도록 도와준다고 하는데 인증된 사용자인지는 어떻게 아는건가요?

여기서 로그인이 성공하면 token을 확인해 이걸로 인하여 @AuthenticationPrincipal 이 인증된 사용자라 판단.
돌아가능하면

질문 4. line 40에서 ResponseEntity<> 여기서 악어입 사이에 아무것도 없는 이유는 line32에서 ResponseEntity<?> '?'이기 때문인가요?
4-1. 그럼 line 32에서 ResponseEntity<T>이면 line 40에서 ResponseEntity<T>로 작성하는게 맞나요?

return 값에서는 무조건 ResponseEntity<> 입니다.

Business - model

@Builder 쓰는 이유

1. 생성자 파라미터가 많을 경우 가독성이 좋지 않다.

위에 예시에서 Bag 클래스는 생성자 파라미터를 3개만 받는다. 3개까지는 괜찮다. 하지만 생성자 파라미터로 받아야하는 값이 수업이 많아진다면? 각 값들이 어떤 값을 의미하는지 이해하기 힘들 것이다.

```
Bag bag = new Bag("name", 1000, "memo", "abc", "what", "is", "it", "?");
```

하지만 이를 빌더 패턴으로 구현하면 각 값들은 빌더의 각 값들의 이름 함수로 셋팅이 되지 각각 무슨 값을 의미하는지 파악하기 쉽다. 따라서 생성자로 설정해야하는 값이 많을 경우 빌더를 쓰는 것이 생성자보다 가독성이 좋다. 이는 같은 타입의 다른 변수의 값을 서로 바꿔 넣는 것을 방지할 수도 있다.

```
Bag bag = Bag.builder()
                .name("name")
                .money(1000)
                .memo("memo")
                .letter("This is the letter")
                .box("This is the box")
                .build();
```

2. 어떤 값을 먼저 설정하던 상관 없다

생성자의 경우는 정해진 파라미터 순서대로 꼭 값을 넣어줘야한다. 순서를 무시하고 값을 넣으면 에러가 발생하거나 엉뚱한데 값이 들어갈 수 있다.

```
public Bag(String name, int money, String memo) {
        this.name = name;
        this.money = money;
        this.memo = memo;
}
```

하지만 빌더 패턴은 빌더의 필드 이름으로 값을 설정하기 때문에 순서에 종속적이지 않다.
그냥 쓰이는 곳에서 어떤 필드를 먼저 설정해야하는지 굳이 순서를 생각할 필요 없이 편하게 설정하면 된다.

```
Bag bag = Bag.builder()
                .name("name")
                .memo("memo")   // memo를 money 대신 먼저!
                .money(1000)
                .build();
```

@Builder 와 @Getter를 함께 사용하면 객체 생성과 필드 접근을 간편하게 처리할 수 있으며
코드의 가독성과 유지보수성을 향상시킬 수 있습니다.

특히 데이터 객체(Data Object)나 불변 객체(Immutable Object)를 구현할 때 많이 사용되는 패턴.

@Getter. ※ 사용하는 클래스가 아닌 사용되는 클래스에 작성되어 있어야함.

ex) public class Person {
    private String name;
    private int age;
}

가 있을 때 'name'과 'age'와 같은 Getter 메소드 'getName()' 'getAge()' 같은 것을
자동으로 생성해줌.

@NoArgsConstructor

보통 객체 생성할 때 Person person = new Person(승완, 25);
이렇게 해야하는데 Person person = new Person(); 이게 가능해진다.
변수들은 null, 0 등 기본 값으로 자동 초기화. 생성자 없이.

@AllArgsConstructor

생성자없이 Person person = new Person(승완, 25); 가능.

@RequiredArgsConstructor

private final int height.
이런 final 필드에 대한 생성자 자동 생성

@Entity

데이터베이스 테이블과 일대일로 매핑되는 객체단위
Entity 객체의 인스턴스 하나가 테이블에서 하나의 레코드값을 의미.
키값은 @Id 어노테이션을 표시. (객체의 인스턴스를 구분하기 위한 유일한 키값)
테이블상 Primary Key와 같은의미.

@Table (name="~", schema="~", ~등등)

Entity 클래스와 데이터베이스 테이블과 매핑을 명시할 때 사용.

@Where

Entity가 조회되는 시점에 추가적인 조건을 적용하는데 사용.
쿼리 연산
조건이 여러개면 SQL 논리연산자를 사용해 조건을 조합.
ex) clause = "~ AND ~"

@Column (name="~")

Entity의 변수나 클래스를 db의 컬럼과 매핑.

@JsonProperty

Json 직렬화 및 역직렬화 시에 사용되는 필드 또는 메소드 이름을 명시하는데 사용.
한번 설정하면 변경 불가!
다시 설정해줘야 함.

@Id, @GeneratedValue.

테이블의 Primary key값 설정하는 어노테이션.

@GeneratedValue는 키값을 설정하는 것이고
3가지 방식이 있다. Identity, sequence, table // DB종류에 따라서 1차선택

@JsonIgnore

Json 직렬화 or 역직렬화 시 무시하도록 하는 것.
민감하거나 불필요한 정보. Json 변환시 해당 부분 무시함.

Business - repo

@Repository

스프링 프레임워크에서 사용되는 어노테이션 중 하나로, 데이터 엑세스 계층의 구성요소에 적용.
해당 클래스가 데이터베이스와의 상호작용을 담당하는 Repository 컴포넌트임을 나타낸다.
이 어노테이션이 클래스에 적용되면 스프링 컨테이너는 해당 클래스를 빈(Bean)으로 등록하고
데이터 엑세스 계층에서 사용할 수 있게 된다.
데이터베이스와의 상호작용을 위한 CRUD 기능을 제공하는 메소드를 가짐.

Business - service

BusinessServiceImpl

@Log4j2

Lombok 라이브러리에서 제공하는 어노테이션 중 하나로 로깅을 위한 코드를 간편하게 작성할 수 있도록 도와주는 역할.
로깅코드 자동생성

ex) log.info("~");

@Service

비즈니스 로직을 처리하는 서비스(Service) 계층의 구성 요소에 적용됩니다.
이 어노테이션은 해당 클래스가 비즈니스 로직을 담당하는 서비스 역할을 수행함을 나타냄.
캡슐화 및 모듈 내 단독 인터페이스로써 제공되는 등각을 의미.

@Transactional

- 한꺼번에 많은 하나 일련의 코드들을 한 단위로 묶어서 작업을 처리하는 방법
- 중간에 오류가 생겼을 경우에 작업 이전으로 되돌려주는 역할.

일단 service 클래스만의 함수에는 다 붙여라.
한 작업 한 세트로 끝나야함.

ex) @Transactional(value="~")

4. @Transactional 설정

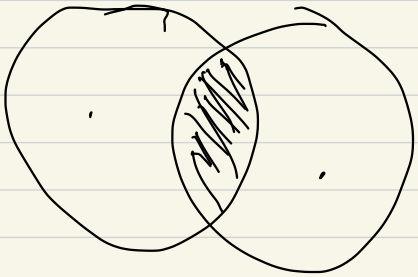
속성	타입	설명
value	String	사용할 트랜잭션 관리자
propagation	enum: Propagation	선택적 전파 설정
isolation	enum: Isolation	선택적 격리 수준
readOnly	boolean	읽기/쓰기 vs 읽기 전용 트랜잭션
timeout	int (초)	트랜잭션 타임 아웃
rollbackFor	Throwable 로부터 얻을 수 있는 Class 객체 배열	롤백이 수행되어야 하는, 선택적인 예외 클래스의 배열
rollbackForClassName	Throwable 로부터 얻을 수 있는 클래스 이름 배열	롤백이 수행되어야 하는, 선택적인 예외 클래스 이름의 배열
noRollbackFor	Throwable 로부터 얻을 수 있는 Class 객체 배열	롤백이 수행되지 않아야 하는, 선택적인 예외 클래스의 배열
noRollbackForClassName	Throwable 로부터 얻을 수 있는 클래스 이름 배열	롤백이 수행되지 않아야 하는, 선택적인 예외 클래스 이름의 배열

@ Override

- 해당 메소드가 부모 클래스에 있는 메소드를 Override 했다는 것을 명시적으로 선언.
- 컴파일러에게 문법체크를 하도록 안내함.

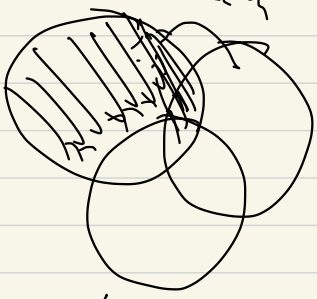
Join

Inner



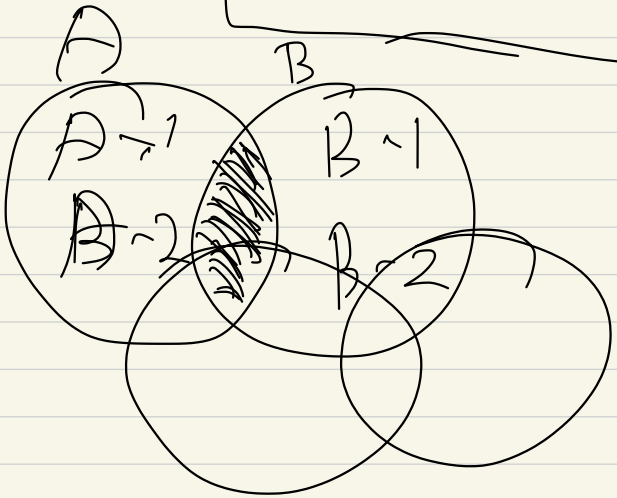
Left

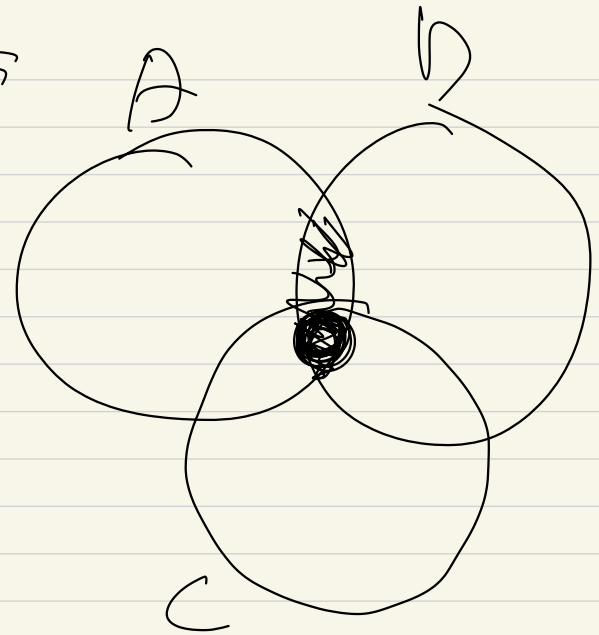
Left

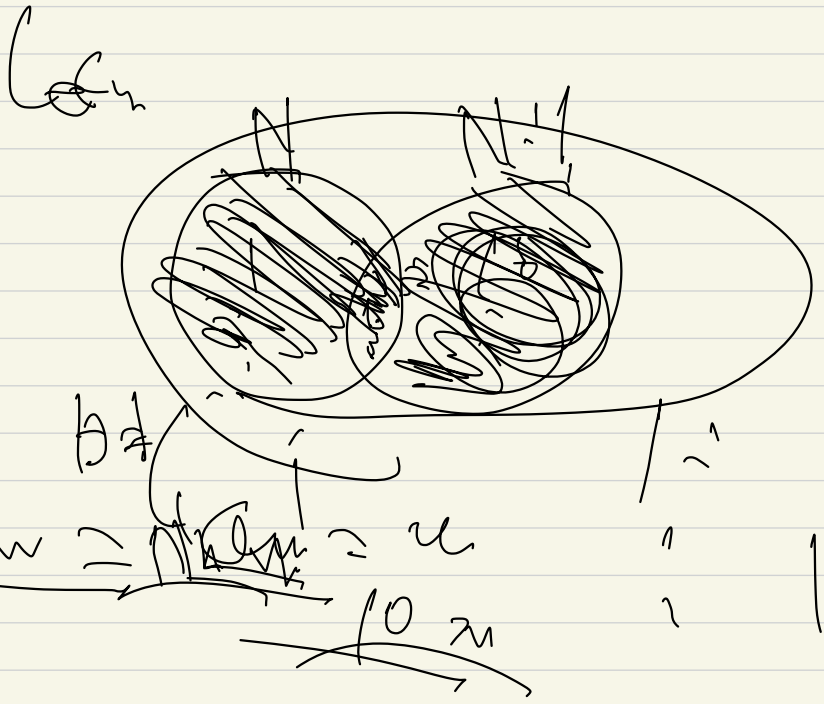


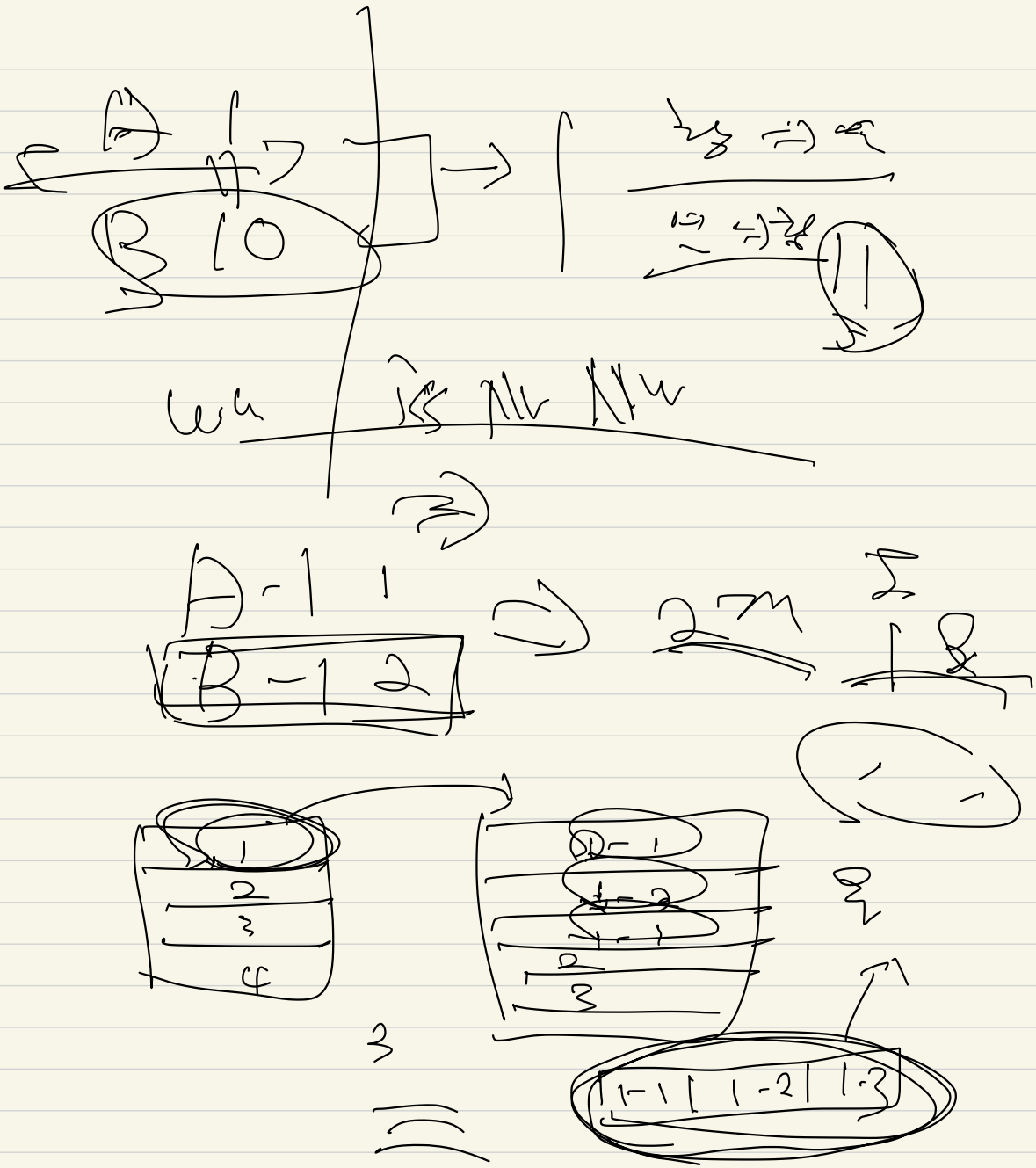
Right

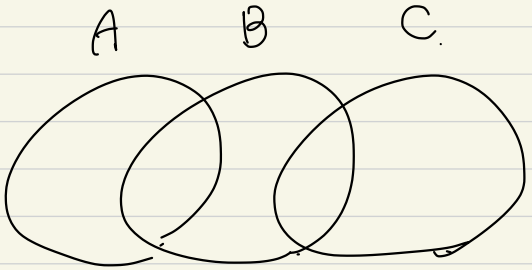
Select * from A join B on A.1 = B.1
left join C on A.1 = C.1
right join D on B.1 = C.1

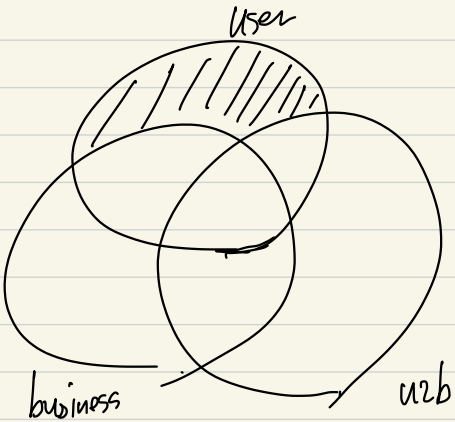



A
B
C






A
B
C.


User
business
u2b


business