バックエンド第2回 検索機能の実装

検索機能仕様

RESTful API(Controller)仕様

Class名：ItemController

機能	メソッド	URL	method名	レスポンスボディ
全件検索	GET	/api/items/	selectAll()	Json配列
1件取得	GET	/api/items/{id}	selectByKey(id)	Jsonデータ(1件)

ENTITY仕様

Class名：Item

No.	フィールド名	型	説明
1	id	int	レコードID(自動採番)
2	title	String	アイテムタイトル
3	content	String	アイテム内容
4	status	String	状態

REPOSITORY仕様

Interface名：ItemDao

No.	戻り値型	メソッド名	引数
1	List<Item>	selectAll	なし
2	Item	selectByKey	int id

Mapper名：ItemDao

No.	ディレクティブ	id	結果タイプ	SQL
1	select	selectAll	com.example.webapi.entity.Item	select id, title, content, status from items order by id
2	select	selectByKey	com.example.webapi.entity.Item	select id, title, content, status from items where id = #{id}

SERVICE仕様

Class名：ItemService

No.	戻り値型	メソッド名	引数
1	List<Item>	selectAll	なし
2	Item	selectByKey	int id

ENTITY実装

`src/main/java/com/example/webapi/entity`に`Item.java`を作成します。

`Item.java`:

```
package com.example.webapi.entity;

import java.io.Serializable;

import lombok.Data;
```

```
@Data
public class Item implements Serializable{
    private static final long serialVersionUID = 1L;
    private int id;
    private String title;
    private String content;
    private String status;
}
```

@Dataアノテーションについて

通常、JavaBeansは次のルールを満たしている必要があります。
(JavaBeansについては、こちらが参考になります。=> https://engineer-club.jp/java-beans)

1. プロパティへのgetter/setterメソッドを持つ
2. 引数のないコンストラクタを持つ
3. java.io.Serializableを実装している

```
// コンストラクタ
public Item() {
}
// 各フィールドのSetter/Getter
public int getId() {
    return this.id;
}
public void setId(int id) {
    this.id = id;
}
:
これらを省略可能になるのです
```

@Dataアノテーションを利用することにより、Lombokの機能でコンパイル時に自動で`setter, getter, equals, toString`などのメソッドを自動で生成してくれるようになります。これにより、決まりきったコードを書く手間を省けます。

REPOSITORY(java)実装

`src/main/java/com/example/webapi/repository`に`ItemDao.java`を作成します。

`ItemDao.java:`

```
package com.example.webapi.repository;

import java.util.List;

import com.example.webapi.entity.Item;
```

```
import org.apache.ibatis.annotations.Mapper;

@Mapper
public interface ItemDao {

    List<Item> selectAll();

    Item selectByKey(int id);

}
```

@Mapperアノテーションについて

MyBATISのXMLにSQLを記述して、それを実行する機能を使う場合に必要です。

`Item`に対するCURDの操作を提供するインターフェースを定義します。
実際のSQLは`resource`の同じ階層でクラス名と同じXMLに記載します。

REPOSITORY(xml)実装

`src/main/resources/com/example/webapi/repository`に`ItemDao.xml`を作成します。

`ItemDao.xml:`

```
<!--?xml version="1.0" encoding="UTF-8" ?-->
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.example.webapi.repository.ItemDao">

    <select id="selectAll" resultType="com.example.webapi.entity.Item">
        select id, title, content, status from items order by id
    </select>

    <select id="selectByKey" resultType="com.example.webapi.entity.Item">
        select id, title, content, status from items where id = #{id}
    </select>

</mapper>
```

namespace

namespaceには対応するJavaのクラス(`com.example.webapi.repository.ItemDao`)を設定します。

resultType

resultTypeにはItemのクラス(`com.example.webapi.entity.Item`)を設定します。

これにより、クエリの結果が、`List<Item>`や`Item`にセットされます。

SERVICE実装

`src/main/java/com/example/webapi/service`に`ItemService.java`を作成します。

`ItemService.java`:

```
package com.example.webapi.service;

import java.util.List;

import com.example.webapi.entity.Item;
import com.example.webapi.repository.ItemDao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
@Transactional(readOnly = true)
public class ItemService {

    @Autowired
    private ItemDao itemMapper;

    @Transactional
    public List<Item> selectAll() {
        return itemMapper.selectAll();
    }

    @Transactional
    public Item selectByKey(int id) {
        return itemMapper.selectByKey( id );
    }

}
```

@Serviceアノテーション

Spring Bootにこのクラスがserviceであることを知らせます。
このアノテーションが設定されると、クラスがDIコンテナへの登録対象としてマークされます。

@Transactionalアノテーション

クラスに設定するとクラス全体でトランザクション管理が可能になります。
通常はServiceレベルでトランザクション管理を行います。

`readOnly = true`に関して

クラス全体を読み込み専用として設定します。
個別のメソッドで`readOnly = false`に設定することで更新系に対応します。

@Autowiredアノテーション

DIコンテナへ登録されたインスタンスを注入したい箇所に設定します。
サンプルの場合、`ItemDao`のMapperを注入することで、そのメソッドを利用可能にします。

(参考)　DIコンテナについて
https://qiita.com/gksdyd88/items/7886f54ee8a22d311400
https://qiita.com/park-jh/items/4df5c67d895b2ea219d6

CONTROLLER実装

`src/main/java/com/example/webapi/controller`に`ItemController.java`を作成します。

`ItemController.java`:

```
package com.example.webapi.controller;

import java.util.List;

import com.example.webapi.entity.Item;
import com.example.webapi.service.ItemService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("api")
@CrossOrigin
public class ItemController {

    @Autowired
    private ItemService itemService;

    @GetMapping(value = "/items")
    public List<Item> selectAll() {
        return itemService.selectAll();
    }

    @GetMapping(value = "/items/{id}")
    public Item selectByKey(@PathVariable int id) {
        return itemService.selectByKey( id );
    }

}
```

@RestControllerアノテーション

SpringBootでRESTful APIを作るときのお約束です。

@RequestMappingアノテーション

URLが`/api`で始まるものが評価されます。

@CrossOriginアノテーション

別のサーバー(例えばフロントエンドが別ポートや別サーバー)からAPIコールされる場合、これを付けないと不正なアクセスとして扱われる。

@Autowiredアノテーション

ここでは、`ItemService`クラスのメソッドを使いたいので、インスタンスを注入する。

@GetMappingアノテーション

メソッドが`GET`の場合に評価される。
`value = "xxx"`で/api以降のURLを取り出し、合致したものが実行される。

- /api/items -> selectAll がコールされる
- /api/items/1 -> selectByKey(1)がコールされる

`@PathVariable`はPathに含まれる変数を取得する。
サンプルの場合URLの`{id}`に設定された値を`int id`として取り出す。

ビルド・動作確認

ビルド

gradlew buildでビルドします。

```
PS C:\home\webapi> ./gradlew build

> Task :test
2020-07-11 22:49:08.994 DEBUG 20096 --- [extShutdownHook]
o.s.w.c.s.GenericWebApplicationContext   : Closing
org.springframework.web.context.support.GenericWebApplicationContext@39401536,
started on Sat Jul 11 22:49:05 JST 2020
2020-07-11 22:49:09.000  INFO 20096 --- [extShutdownHook]
o.s.s.concurrent.ThreadPoolTaskExecutor  : Shutting down ExecutorService
'applicationTaskExecutor'

BUILD SUCCESSFUL in 17s
5 actionable tasks: 5 executed
```

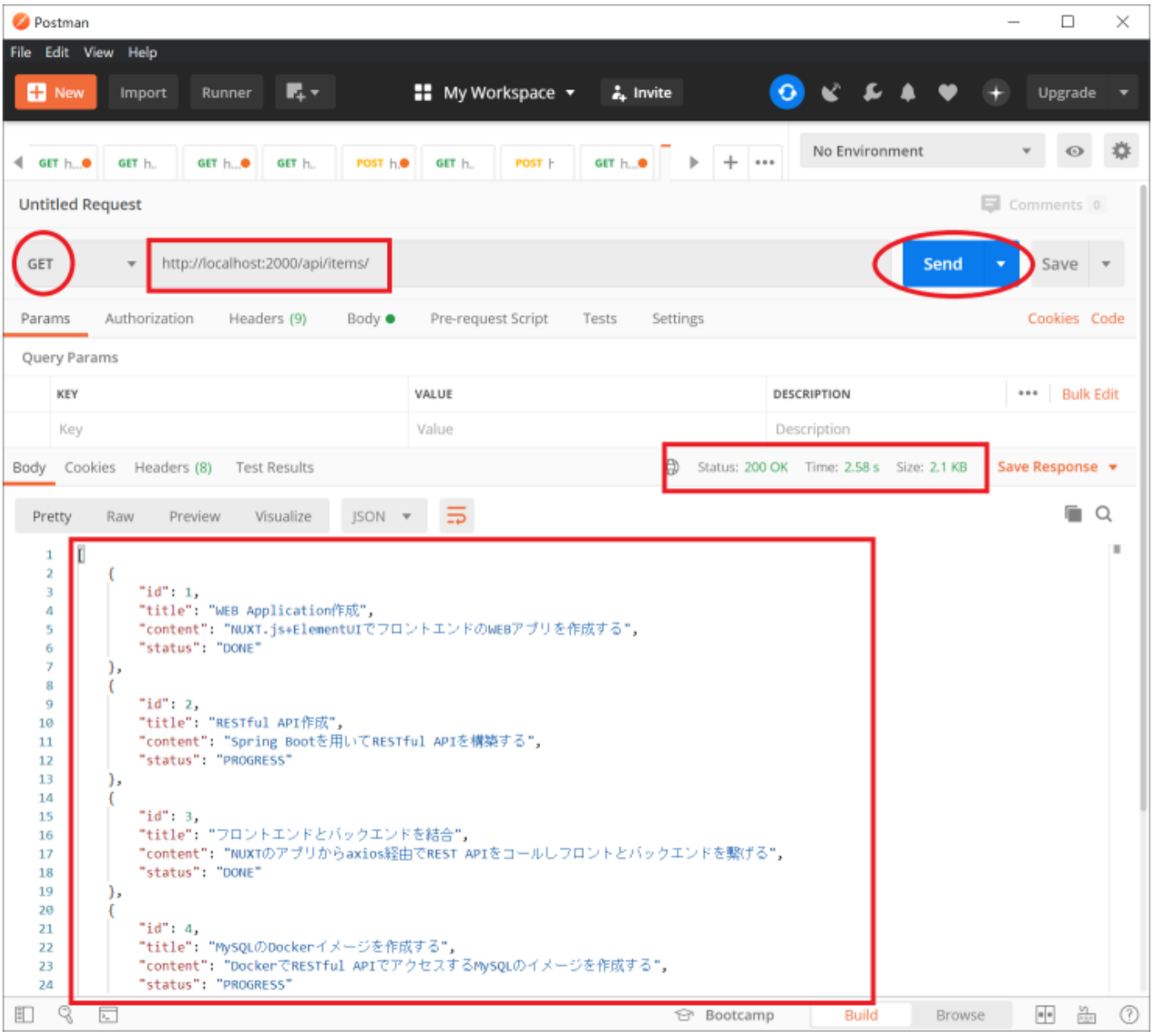
起動

SPRING-BOOT DASHBOARDでwebapiを起動します。

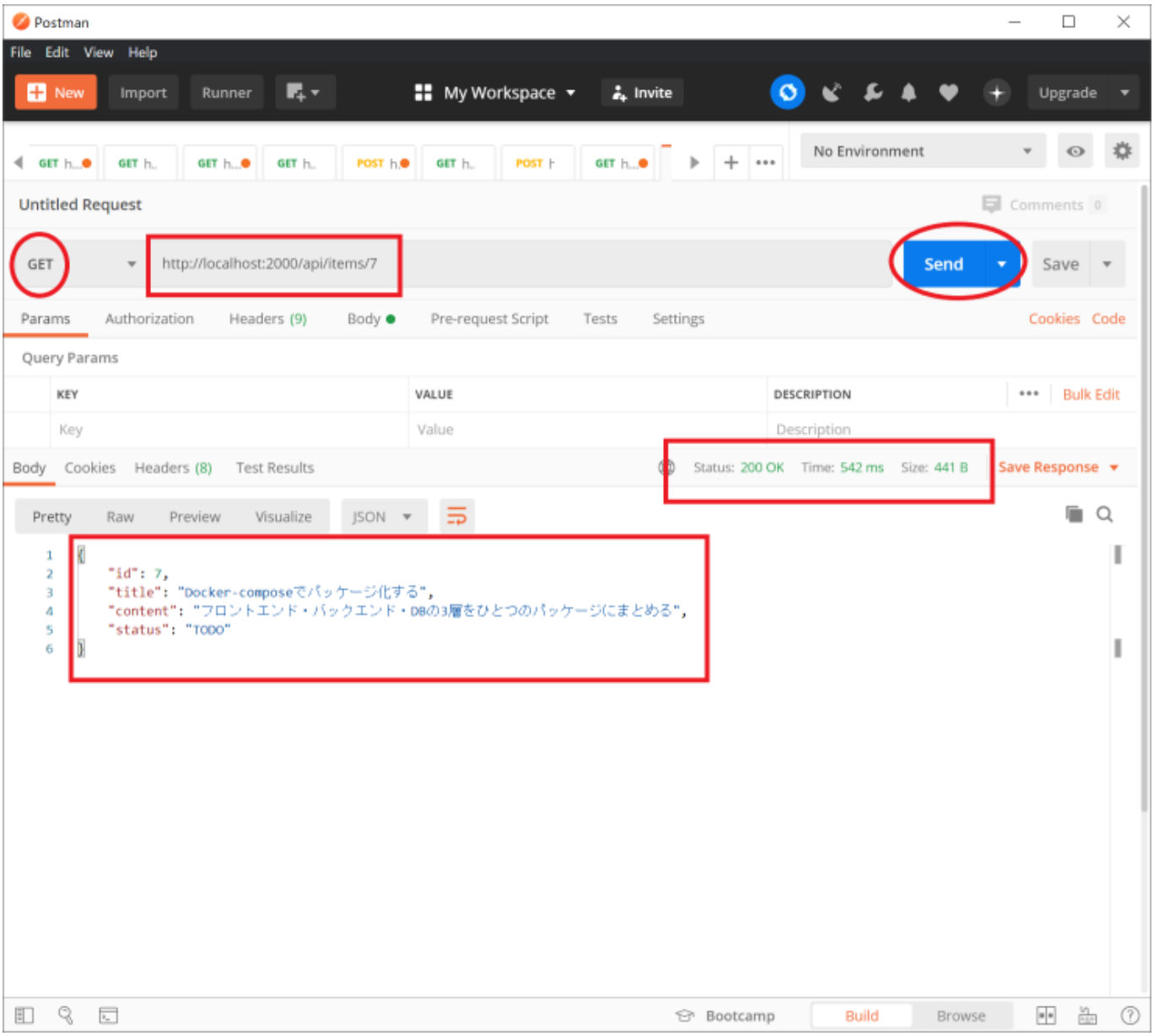
動作確認

Postmanを起動し、下記URLにアクセス(GET)して動作結果を確認します。

```
http://localhost:2000/api/items/
```




```
http://localhost:2000/api/items/7/
```



APIからは応答が返ってきたでしょうか？

CH2-2はこれで終了します。

次回は編集・削除・新規登録を一気にやりたいと思います。