구글 API를 이용한 기능 구현

0) 준비 과정

Pom.xml에 Google Calendar API 라이브러리를 쓰기 위한 라이브러리들을 추가함.

com.google.api-client -> client library The Google API Client Library for Java (google-api-client) is designed to be compatible with all supported Java platforms, including Android.

Google APIs give you programmatic access to Google Maps, Google Drive, YouTube, and many other Google products. To make coding against these APIs easier, Google provides client libraries that can reduce the amount of code you need to write and make your code more robust.

The libraries can also make it simpler to set up authorization and authentication. If your website or app requires users to sign in, check out Google's Identity Platform products

Google api를 더 쉽게 사용하기 위해 제공. 인증을 쉽게 만들어 줌. 구글 로그인 기능 제공.

org.springframework.social -> Spring Social extension with connection support and an API binding for Google 소셜 로그인을 해줌.

com.google.oauth-client -> 구글 인증. Jetty extensions to the Google OAuth Client Library for Java

com.google.apis -> Calendar API v3-rev235-1.22.0 Calendar API를 사용하기 위한 라이브러리 제공.

org.apache.httpcomponents -> Apache HttpComponents Client

creating and maintaining a toolset of low level Java components focused on HTTP and associated protocols.

(http client )

HttpClient is NOT a browser. It is a client side HTTP transport library. HttpClient's purpose is to transmit and receive HTTP messages. HttpClient will not attempt to process content, execute javascript embedded in HTML pages, try to guess content type, if not explicitly set, or reformat request / rewrite location URIs, or other functionality unrelated to the HTTP transport.

Google Cloud Platform에 접속.

사용자 인증 정보를 만들어서 Oauth 클라이언트 ID(특정 사용자의 구글 안에서의 정보를 가져오는 경우)를 받음.

프로젝트를 생성하고 사용할 API 설정.

client\_secret.json 파일을 다운 받아 프로젝트에 저장.

Gooogle의 API Explorer를 사용하면 API에서 제공해주는 기능들을 보여줌.

Oauth 클라이언트의 아이디와 비밀번호를 노출하면 안됨.

1) 구글 인증

OAuth 2.0 이용해서 구글 계정으로 로그인을 한 후 사용자에게서 캘린더 수정 권한을 받음.

Oauth-> Resource Server-> 정보가 저장되어 있는 서버. Resource owner-> 정보의 주인

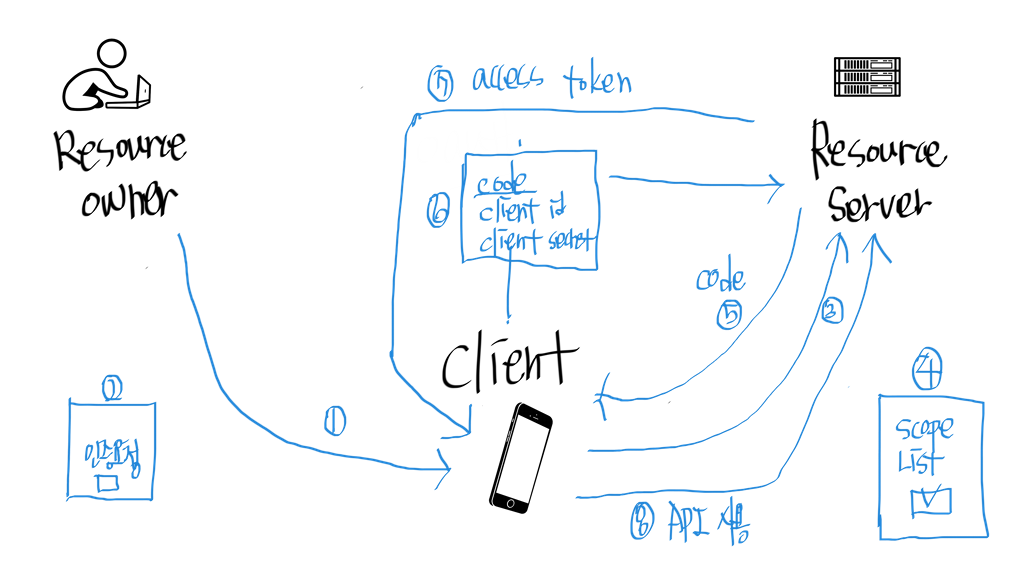
Resource Server에 서비스 등록을 해서 서버에서는 client id, client secret을 발급해 줌. 이 두개의 값을 저장. secret정보는 절대로 노출 되면 안됨.

Resource Server에게 어떤 정보를 쓰겠다고 허락 받아 달라고 요청해서 허락 받아옴. 허락 되면 resource 자동으로 접근이 됨. 로그인이 안되어 있으면 로그인 요청. 이 승인했다는 정보를 code(비밀번호 같은 것, 캘린더 목록을 사용할 수 있는 권한)로 전달해 줌. Code, id, secret을 담아서 리소스 서버에 다시 보냄. 리소스 서버는 클라이언트에게 발급했던 코드 값, 자신을 입증하는 id, secret 값을 비교해서 리소스 오너가 승인해준 클라이언트인지 확인 함. 유효하면 리소스 서버는 클라이언트에게 access token이라는 진짜 비밀 번호를 알려줌. 이 것은 파일에 저장. 이 토큰을 제출해서 서버에 접속해서 요청함. 보안적으로 안전하니깐 sdk (도구)를 사용해야 함.

Scope-> 전체 데이터에서 일부를 가져오는 것이 Scope. 구글 캘린더에서 리스트만 가져오는 것.

구글의 전체 데이터에서 일부 데이터만 사용하고 싶기 때문에 resource owner와 resource server에게 일부 데이터가 뭔지 알려줘서 허락 받아야 함.

클라이언트(내가 만든 프로젝트)의 id와 패스워드(클라이언트에 대한 구글의 인증 정보)는 일부 api에 대한 인증. Scopes은 그 일부 api안에서 필요한 일부 데이터를 요청하는 것. Resource server는 로그인 안되어 있으면 로그인 요청, 되어있으면 현재 클라이언트가 필요로 하는 scope가 무엇인지 리스트를 보여줌.-> 이런 일을 클라이언트가 하도록 허용할 것인지 resource owner에게 물어봄. -> 허락하면 resource server는 client에게 authorization code(리소스 오너가 스콥에 대한 허락에 대한 비밀번호)를 전송.-> 이 code를 직접 사용하지 않고 code, client id, client secret을 resource server에게 다시 보냄-> resource server는 이 client가 code에 해당하는 행위(owner의 scope)가 맞는지 확인하고 맞으면 access token을 보냄-> client는 access token을 이용해 API 사용.



(출처 : <https://opentutorials.org/course/2473/16571>)

3번에서 버튼을 눌렀을 때 이동할 url 생성이 필요

* Scope(resource owner에게 승낙 받아야 하는 것),
* Access type : access token을 client가 계속 가지고 있기엔 위험해서 수명이 있음. 수명이 끊기면 다시 허락 받음.
  + Online, offline이 있음.
    - Offline : access token을 리소스 서버가 줄 때 refresh token도 같이 줌. 클라이언트는 두 token을 저장. Access token이 만료되서 못 쓰면 리소스 서버에 refresh token(사용자가 처음 허락 할 때만 생김)으로 다시 요청해 access token을 받음.
* Redirect uri: 사용자가 인증이 끝나 resource server에서 code값을 줄 때 어디에 줘야 클라이언트가 받을 수 있을지
  + 5번 일 때와 6번 일 때 redirect uri 값을 같게 써야함.
* Response\_type=code 지금 resource server에 요청하는게 authorization code
* Client\_id

Access token-> 오너에 대한 인증.

리소스 오너가 클라이언트에 접속하면 화면에 인증 요청 페이지가 나옴-> 확인 버튼을 누르면 버튼의 링크는 resource server로 향함

*HttpTransport HTTP\_TRANSPORT* = GoogleNetHttpTransport.*newTrustedTransport*();

*DATA\_STORE\_FACTORY* = **new** FileDataStoreFactory(***DATA\_STORE\_DIR***);

->httptransport는 delete, get, post put만 지원해 줌.

//인증

**public** **static** Credential authorize() **throws** IOException {

// Load client secrets.

InputStream in =

GoogleCalendarService.**class**.getResourceAsStream("/client\_secret\_1.json");

GoogleClientSecrets clientSecrets =

GoogleClientSecrets.*load*(***JSON\_FACTORY***, **new** InputStreamReader(in));

// Build flow and trigger user authorization request.

GoogleAuthorizationCodeFlow flow =

**new** GoogleAuthorizationCodeFlow.Builder(

*HTTP\_TRANSPORT*, ***JSON\_FACTORY***, clientSecrets, ***SCOPES***)

.setDataStoreFactory(*DATA\_STORE\_FACTORY*)

.setAccessType("offline")

.build();

Credential credential = **new** AuthorizationCodeInstalledApp(

flow, **new** LocalServerReceiver()).authorize("user");

System.***out***.println(

"Credentials saved to " + ***DATA\_STORE\_DIR***.getAbsolutePath());

**return** credential;

}

->LocalServerReceiver에서 redirecUri 생성. Host(localhost), port는 -1로 해서 사용하지 않은 port 선정 verification code receiver

->authorize(“user”) 이 부분에서 브라우저를 열어서 승인을 받고 code를 받아서 access token을 받음. 그리고 credential 저장.

//연결

**public** **static** com.google.api.services.calendar.Calendar

getCalendarService() **throws** IOException {

Credential credential = *authorize*();

**return** **new** com.google.api.services.calendar.Calendar.Builder(

*HTTP\_TRANSPORT*, ***JSON\_FACTORY***, credential)

.setApplicationName(***APPLICATION\_NAME***)

.build();

}

-> JsonFactory -> JSON 파싱 기능 제공.(input stream에서 파싱)

->builder안에 캘린더 정보 요청할 때 사용하는 url 가지고 있음.

2) 캘린더 리스트 받아오기

**public** **static** ArrayList<CalendarDTO> getCalendarList() **throws** IOException{

ArrayList<CalendarDTO> result = **new** ArrayList<CalendarDTO>();

com.google.api.services.calendar.Calendar service = *getCalendarService*();

String pageToken = **null**;

**do** {

CalendarList calendarList = service.calendarList().list().setPageToken(pageToken).execute();

List<CalendarListEntry> items = calendarList.getItems();

**for** (CalendarListEntry calendarListEntry : items) {

CalendarDTO tempDTO = **new** CalendarDTO();

tempDTO.setId(calendarListEntry.getId());

tempDTO.setSummary(calendarListEntry.getSummary());

tempDTO.setCheck(**true**);

tempDTO.setColorId(calendarListEntry.getColorId());

**boolean** primary = **true**;

**if**(calendarListEntry.getPrimary() == **null**) {

primary = **false**;

}

tempDTO.setPrimary(primary);

tempDTO.setAccessRole(calendarListEntry.getAccessRole());

result.add(tempDTO);

}

pageToken = calendarList.getNextPageToken();

} **while** (pageToken != **null**);

**return** result;

}

->서버에서 받아올 자원들의 개수를 max로 지정해두어도 보장받지 못할 수 있음. 그럴 때 page token을 사용해서 page token이 null이 나올 때 까지 받아오면 모두 받을 수 있음.

한 페이지 안에 서버가 보낼 수 있는 최대의 자원들이 들어가 있음.

* 한번 요청에 최대 250개를 받아 올 수 있음. Default = 100
* 캘린더 개수가 많지 않으므로 빼도 상관없지만 page token 사용법을 익히기 위해 기록.

-> CalendarList calendarList = service.calendarList().list().setPageToken(pageToken).execute();

이 방법으로 리스트를 얻어 옴.

* 그 후 아래의 DTO에 맞게 저장함.
* DTO를 사용하는 이유는 전송하는 데이터의 사이즈를 작게 해서 전송 시간을 줄이기 위함.
  + API에 요청하고 받아오는 시간이 오래 걸림.

CalendarDTO

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 자료명 | 자료형 | 자료형 크기 | 설명 | 값 |
| id | String |  | 사용자의 캘린더 아이디 | CalendarListEntry.getId() |
| summary | String |  | 사용자의 캘린더 제목 | CalendarListEntry.getSummary() |
| check | boolean |  | 사용자가 캘린더의 체크박스 체크 여부 | Calendar의 id를 name으로 갖는 세션의 value  (초기는 true) |
| colorId | String |  | 사용자의 캘린더 색 아이디 | CalendarListEntry.getColorId() |
| primary | Boolean |  | 캘린더가 사용자의 메인 캘린더인지에 대한 여부 | calendarListEntry.getPrimary() |
| accessRole | String |  | 캘린더의 Access Role(일정에 대한 접근 권한 정의) | calendarListEntry.getAccessRole() |

3) 이벤트 리스트 받아오기

**public** **static** ArrayList<EventDTO> getEvent\_Month(ArrayList<CalendarDTO> calendarList, **int** year, **int** month) **throws** IOException{

com.google.api.services.calendar.Calendar service =

*getCalendarService*();

// List the next 10 events from the primary calendar.

//DateTime now = new DateTime(System.currentTimeMillis());

Date cur = **new** ~~Date~~(year-1900, month-1, 1);

ArrayList<EventDTO> dtoList = **new** ArrayList<EventDTO>();

Date nextDate;

**if**(month == 12) {

nextDate = **new** ~~Date~~(year-1900 + 1,0,1);

}**else** {

nextDate = **new** ~~Date~~(year-1900,month,1);

}

DateTime now = **new** DateTime(cur);

DateTime next = **new** DateTime(nextDate);

**int** size = calendarList.size();

**for**(**int** i=0;i<size;i++) {

//System.out.println(checkedCalId.get(i));

String id = calendarList.get(i).getId();

Events events = service.events().list(id)

.setTimeMin(now)

.setTimeMax(next)

.setOrderBy("startTime")

.setSingleEvents(**true**)

.execute();

List<Event> items = events.getItems();

**if** (items.size() == 0) {

System.***out***.println("No upcoming events found.");

} **else** {

// System.out.println(now.toString());

System.***out***.println("Upcoming events");

**for** (Event event : items) {

DateTime start = event.getStart().getDateTime();

**if** (start == **null**) {

start = event.getStart().getDate();

}

DateTime end = event.getEnd().getDateTime();

**if**(end == **null**) {

end = event.getEnd().getDate();

}

// System.out.printf("%s (%s)\n", event.getSummary(), start.toString());

EventDTO tempDTO = **new** EventDTO();

tempDTO.setCalendarID(id);

tempDTO.setSummary(event.getSummary());

tempDTO.setStart(start.getValue(),start.isDateOnly());

tempDTO.setEnd(end.getValue(),end.isDateOnly());

tempDTO.setEventID(event.getId());

dtoList.add(tempDTO);

}

}

}

Collections.*sort*(dtoList,**new** comparator());

dtoList = **new** EventProcessing().arrangeOrder(dtoList, year, month);

**return** dtoList;

}

* calendarList에는 현재 체크 된 캘린더 만 입력 되어 있음.

Events events = service.events().list(id)

.setTimeMin(now)

.setTimeMax(next)

.setOrderBy("startTime")

.setSingleEvents(**true**)

.execute();

* + id는 calendar Id.
  + setTimeMin과 Max로 이벤트를 받아올 날짜의 범위를 지정. 그 안에 포함된 이벤트들을 모두 가져온다.
    - setTimeMin은 이벤트의 끝 날짜를 기준으로 파라미터 날짜 보다 늦은 이벤트들만 뽑는다. inclusive
    - setTimeMax는 이벤트의 시작 날짜를 기준으로 파라미터 날짜 보다 빠른 이벤트들만 뽑는다. exclusive
    - 둘다 RFC3339 형태. 밀리초 단위는 무시.
  + setOrderBy(“startTime)을 사용하려면 setSingleEvents(true)로 해야함.
    - Default = false
    - 반복 이벤트를 각각 하나의 이벤트로 볼지에 대한 여부
* 이벤트의 시작과 끝 날짜가 종일 일정인 경우(Date)와 아닌 경우(DateTime)로 나뉘므로 다르게 값을 받음.
* 아래의 DTO 형태로 저장.
* 선택된 모든 캘린더들의 이벤트들을 저장하고 우선순위에 맞게 정렬하여 리턴.
* 이벤트 한번 요청 시 받아올 수 있는 최대 개수는 250개. 만약 더 가져오고 싶으면 pageToken 사용.
* EventDTO

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 자료명 | 자료형 | 자료형 크기 | 설명 | 값 |
| calendarID | String |  | 사용자의 캘린더 아이디 | CalendarListEntry.getID() |
| eventID | String |  | 일정 아이디 | Event.getId() |
| summary | String |  | 일정 제목 | Event.getSummary() |
| start | long |  | 일정 시작 날짜 | Event.getStart().getDateTime().getValue() |
| startTime | Int | 5 | 일정 시작 날짜를 년, 월, 일, 시, 분으로 나눔. | start를 년, 월, 일, 시, 분으로 나눔. |
| end | long |  | 일정 끝 날짜 | Event.getEnd().getDateTime().getValue() 여기서 종일 일정인 경우는 하루 빼줘야 함. |
| endTime | Int | 5 | 일정 끝 날짜를 년, 월, 일, 시, 분으로 나눔. | end를 년, 월, 일, 시, 분으로 나눔. |

->start나 end를 저장할 때 startTime, endTime의 값을 함수 내에서 파싱해서 세팅.

->종일 일정의 경우 end는 하루 뒤의 값을 가지므로 하루치 값을 뺌.

->종일 일정의 경우 start는 현재 timezone에 의해 그날 9시의 값을 가지게 됨.

-> 추후 이벤트들을 정렬하기 위해서 종일 일정인 경우 start의 값을 그날 오전 12시에서 1밀리초를 뺀 값을 저장.(추후 날짜 정보는 startTime으로만 다루기 때문에 값이 달라져도 상관 없음)

4) 이벤트 하나 정보 요청

**public** **static** EventDetailDTO getEventDetail(String calendarId, String eventId) **throws** IOException {

EventDetailDTO result = **new** EventDetailDTO();

com.google.api.services.calendar.Calendar service = *getCalendarService*();

Event event = service.events().get(calendarId, eventId).execute();

result.setSummary(event.getSummary());

DateTime start = event.getStart().getDateTime();

**if** (start == **null**) {

start = event.getStart().getDate();

}

DateTime end = event.getEnd().getDateTime();

**if**(end == **null**) {

end = event.getEnd().getDate();

}

result.setStart(start.getValue(), start.isDateOnly());

result.setEnd(end.getValue(), end.isDateOnly());

result.setLocation(event.getLocation());

result.setDescription(event.getDescription());

result.setRecurrence(event.getRecurrence());

**return** result;

}

* Event event = service.events().get(calendarId, eventId).execute();
  + 이벤트의 calendarId와 eventId를 가지고 요청 후 아래의 DTO 형태로 저장.

**public** **static** Event getEvent(String calendarId, String eventId) {

Event event = **new** Event();

com.google.api.services.calendar.Calendar service;

**try** {

service = *getCalendarService*();

event = service.events().get(calendarId, eventId).execute();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** event;

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 자료명 | 자료형 | 자료형 크기 | 설명 | 값 |
| summary | String |  | 일정 제목 | Event.getSummary() |
| startTime | Int | 5 | 일정 시작 날짜를 년, 월, 일, 시, 분으로 나눔. | 시작 날짜를 년, 월, 일, 시, 분으로 나눔. |
| endTime | Int | 5 | 일정 끝 날짜를 년, 월, 일, 시, 분으로 나눔. | 끝 날짜를 년, 월, 일, 시, 분으로 나눔. |
| description | String |  | 일정 상세 내용 | Event.getDescription() |
| Location | String |  | 일정 장소 | Event.getLocation() |
| Recurrence | List<String> |  | 일정 반복 규칙 | Event.getRecurrence() |

+ 구글의 이벤트 객체 정리

5) 이벤트 생성

Event event = **new** Event() .setSummary(request.getParameter("summary"))

.setLocation(request.getParameter("location"))

.setDescription(request.getParameter("description"))

.setStart(start)

.setEnd(end);

String newCalendarId = request.getParameter("calendars");

service.events().insert(newCalendarId, event).execute();

* setStart, setEnd의 파라미터 데이터 타입은 EventDateTime을 가짐.
  + EventDateTime을 생성하기 위해서는 DateTime이 필요.
  + 사용자의 인풋을 받아 Date->DateTime->EventDateTime으로 저장.
  + 종일 일정인 경우 DateTime으로 저장할 때 DateOnly값을 true로 지정.
    - end값에 하루를 더 더해줘야 함.(exclusive)
    - 종일 일정을 Date에서 DateTime으로 변경할 때는 long value값을 사용하는데 timezone 때문에 현재 +9시간이므로 시간을 0으로 세팅하지 않고 9로 세팅해야 함.
  + 월 값은 0~11을 가지므로 1을 빼줘야 함.
* 나머지는 String
* Insert의 파라미터로는 이벤트를 넣을 calendarId와 이벤트 객체.

6) 이벤트 수정

**public** String updateEvent(HttpServletRequest request, Model model) {

GoogleCalendarService gcs = **new** GoogleCalendarService();

Calendar service;

String calendarId = request.getParameter("calendarId");

String eventId = request.getParameter("eventId");

EventDateTime start = **new** EventDateTime();

EventDateTime end = **new** EventDateTime();

String[] strStartDate = request.getParameter("startDate").split("-");

String[] strEndDate = request.getParameter("endDate").split("-");

System.***out***.println(request.getParameter("allDay"));

**if**(request.getParameter("allDay") != **null**) {

Date startD;

startD = **new** ~~Date~~(Integer.*parseInt*(strStartDate[0])-1900, Integer.*parseInt*(strStartDate[1])-1, Integer.*parseInt*(strStartDate[2]),9,0); //timezone만큼 시간 설정해야함.

System.***out***.println(startD.toString());

start.setDate(**new** DateTime(**true**,startD.getTime(),startD.~~getTimezoneOffset~~())).setTimeZone("Asia/Seoul");

}**else** {

String[] strStartDateTime = request.getParameter("startDateTime").split(":");

System.***out***.println(request.getParameter("startDateTime"));

Date startD = **new** ~~Date~~(Integer.*parseInt*(strStartDate[0])-1900, Integer.*parseInt*(strStartDate[1])-1, Integer.*parseInt*(strStartDate[2]),

Integer.*parseInt*(strStartDateTime[0]), Integer.*parseInt*(strStartDateTime[1]));

start.setDateTime(**new** DateTime(startD)).setTimeZone("Asia/Seoul");

}

**if**(request.getParameter("allDay") != **null**) {

Date endD;

endD = **new** ~~Date~~(Integer.*parseInt*(strEndDate[0])-1900, Integer.*parseInt*(strEndDate[1])-1, Integer.*parseInt*(strEndDate[2])+1,9,0);

DateTime endDate = **new** DateTime(endD);

end.setDate(**new** DateTime(**true**,endD.getTime(),endD.~~getTimezoneOffset~~())).setTimeZone("Asia/Seoul");

}**else** {

String[] strEndDateTime = request.getParameter("endDateTime").split(":");

Date endD = **new** ~~Date~~(Integer.*parseInt*(strEndDate[0])-1900, Integer.*parseInt*(strEndDate[1])-1, Integer.*parseInt*(strEndDate[2]),

Integer.*parseInt*(strEndDateTime[0]), Integer.*parseInt*(strEndDateTime[1]));

end.setDateTime(**new** DateTime(endD)).setTimeZone("Asia/Seoul");

}

System.***out***.println(start.toString());

System.***out***.println(end.toString());

//System.out.println(startDate.toString());

//String[] strStartDateTime = request.getParameter("startDateTime").split(":");

**if**(calendarId.equals(request.getParameter("calendars"))) {

**try** {

service = gcs.*getCalendarService*();

Event event = service.events().get(calendarId, eventId).execute();

event.setSummary(request.getParameter("summary"))

.setLocation(request.getParameter("location"))

.setDescription(request.getParameter("description"))

.setStart(start)

.setEnd(end)

;

Event updatedEvent = service.events().update(calendarId, event.getId(), event).execute();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}**else** **if**(!calendarId.equals(request.getParameter("calendars"))){//calendar Id를 수정한 경우나 일정을 입력한 경우

**try** {

service = gcs.*getCalendarService*();

**if**(!calendarId.equals("addEvent")) {

service.events().delete(calendarId, eventId).execute();

}

Event event = **new** Event()

.setSummary(request.getParameter("summary"))

.setLocation(request.getParameter("location"))

.setDescription(request.getParameter("description"))

.setStart(start)

.setEnd(end)

;

String newCalendarId = request.getParameter("calendars");

service.events().insert(newCalendarId, event).execute();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**return** "redirect:/m/"+request.getParameter("startDate");

}

service = gcs.*getCalendarService*();

Event event = service.events().get(calendarId, eventId).execute();

event.setSummary(request.getParameter("summary"))

.setLocation(request.getParameter("location"))

.setDescription(request.getParameter("description"))

.setStart(start)

.setEnd(end);

Event updatedEvent = service.events().update(calendarId, event.getId(), event).execute();

* 생성과 거의 동일. 함수만 다름.
* 이벤트 객체를 새로 생성하는 것이 아니고 calendarId, eventId로 수정할 이벤트의 객체를 얻어와야 함.
* Update의 파라미터는 현재 이벤트의 calendarId, eventId, 이벤트 객체

7) 이벤트 삭제

@RequestMapping(value = "/deleteEvent", method = RequestMethod.***GET***)

**public** @ResponseBody **boolean** deleteEvent(CalendarAndEventIdDTO dto){

**boolean** result = **false**;

GoogleCalendarService gcs = **new** GoogleCalendarService();

com.google.api.services.calendar.Calendar service;

**try** {

service = gcs.*getCalendarService*();

service.events().delete(dto.getCalendarId(), dto.getEventId()).execute();

result = **true**;

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** result;

}

service.events().delete(dto.getCalendarId(), dto.getEventId()).execute();

* Delete를 할 때는 지우고자 하는 이벤트의 calendarId, eventId를 파라미터로 넣어 실행시키면 됨.

8) 동작 방식

이벤트나 캘린더를 가져오거나 이벤트를 삽입, 수정, 삭제를 하는 요청을 하면

* http 전송을 위해 인증 정보와 json factory, http\_transport를 가지고 전송 준비.
* 요구하는 파라미터에 null이 들어가 있는지 아닌지 확인하고
* 정보를 가져오는 경우에는 GET 메소드 방식, 삽입은 POST, 수정은 PUT, 삭제는 DELETE
  + 이 때 정보로 리턴 되는 정보를 담을 객체 클래스도 지정.
  + 정보를 수정 삽입할 때는 이벤트 객체도 같이 보냄. Content가 있는 경우는 JsonHttpContent를 사용하여 Json 형식으로 바꿈.

Calendar client, String method, String uriTemplate, Object content, Class<T> responseClass) {

**super**(

client, method, uriTemplate, content, responseClass);

}

* Execute()를 하면 http request를 수행하고 response를 받음.

9)우선순위

**public** **class** comparator **implements** Comparator<EventDTO>{

**public** **int** compare(EventDTO dto1, EventDTO dto2) {

**int** ret = 0;

**if**(dto1.getStart() < dto2.getStart()) {

ret = -1;

}**else** **if**(dto1.getStart() == dto2.getStart()){

**if**(dto1.getEnd() > dto2.getEnd()) {

ret = -1;

}**else** **if**(dto1.getEnd() == dto2.getEnd()) {

ret = 0;

}

**else** **if**(dto1.getEnd() < dto2.getEnd()){

ret = 1;

}

}**else** {

ret = 1;

}

**return** ret;

}

}

* 각 캘린더에서 받아온 이벤트들을 시작 날짜가 빠른 순으로 정렬.
* 시작 날짜가 같은 경우에는 끝 날짜가 늦은 순으로 정렬

**class** comparatorEndDesc **implements** Comparator<EventDTO>{

**public** **int** compare(EventDTO dto1, EventDTO dto2) {

**int** ret = 0;

**if**(dto1.getEnd() > dto2.getEnd()) {

ret = -1;

}**else** **if**(dto1.getEnd() < dto2.getEnd()) {

ret = 1;

}

**return** ret;

}

}

* 현재 사용자가 보고 있는 월보다 시작 날짜가 전인 이벤트들을 모아
* 끝 날짜가 늦은 순으로 정렬

**class** comparatorSameDate **implements** Comparator<EventDTO>{

**public** **int** compare(EventDTO dto1, EventDTO dto2) {

**int** ret = 0;

**if**(dto1.getStartTime()[0] == dto2.getStartTime()[0] && dto1.getStartTime()[1] == dto2.getStartTime()[1] && dto1.getStartTime()[2] == dto2.getStartTime()[2]) {

**if**(dto1.getEndTime()[0] == dto2.getEndTime()[0] && dto1.getEndTime()[1] == dto2.getEndTime()[1] && dto1.getEndTime()[2] == dto2.getEndTime()[2]) {//같은날

**if**(dto1.getStart() < dto2.getStart()) {

ret = -1;

}**else** **if**( dto1.getStart() > dto2.getStart()) {

ret = 1;

}**else** {//같은 경우

**if**(dto1.getEnd() > dto2.getEnd()) {

ret = -1;

}**else** {

ret = 1;

}

}

}**else** {

**if**(dto1.getEnd() > dto2.getEnd()) {

ret = -1;

}**else** **if**(dto1.getEnd() < dto2.getEnd()) {

ret = 1;

}

}

}

**return** ret;

}

}

* 위의 경우를 제외하고
* 시작 날짜가 같은 경우
  + 끝 날짜도 같으면 시작 시간이 빠른 순으로 정렬(종일 > 시간)
    - 시작 시간이 일치 하면 끝 시간이 늦은 순으로 정렬.
  + 끝 날짜가 같지 않은 경우
    - 끝 날짜가 늦은 순으로 정렬.
* 이렇게 정렬을 하면 그릴 때 끝 날짜가 늦은 순으로 정렬되어 있기 때문에 즉 칸을 많이 차지하는 순서대로 시작 날짜 순서에 맞게 정렬 되어 있어 미리 자리를 차지하게 만들 수 있음.

우선순위

* 일정이 여러 날에 거쳐서 나타날 경우
  + End Date가 늦은 순서 > 시간이 있는 일정 > 종일 일정
* 시작 날짜와 끝 날짜가 같은 일정인 경우
  + 종일 일정 > 시간이 있는 일정.
  + 시간이 있는 일정인 경우는 시작 시간이 빠른 순으로 정렬.
* 현재 보고 있는 월보다 시작 날짜가 전인 경우
  + 끝 날짜가 늦은 순서
* 그 외의 경우
  + 시작 날짜와 끝 날짜가 같은 일정인 경우
    - 종일 일정 > 시간이 있는 일정 순서
    - 시간이 있는 일정인 경우에는 시작 시간이 빠른 순으로 정렬.