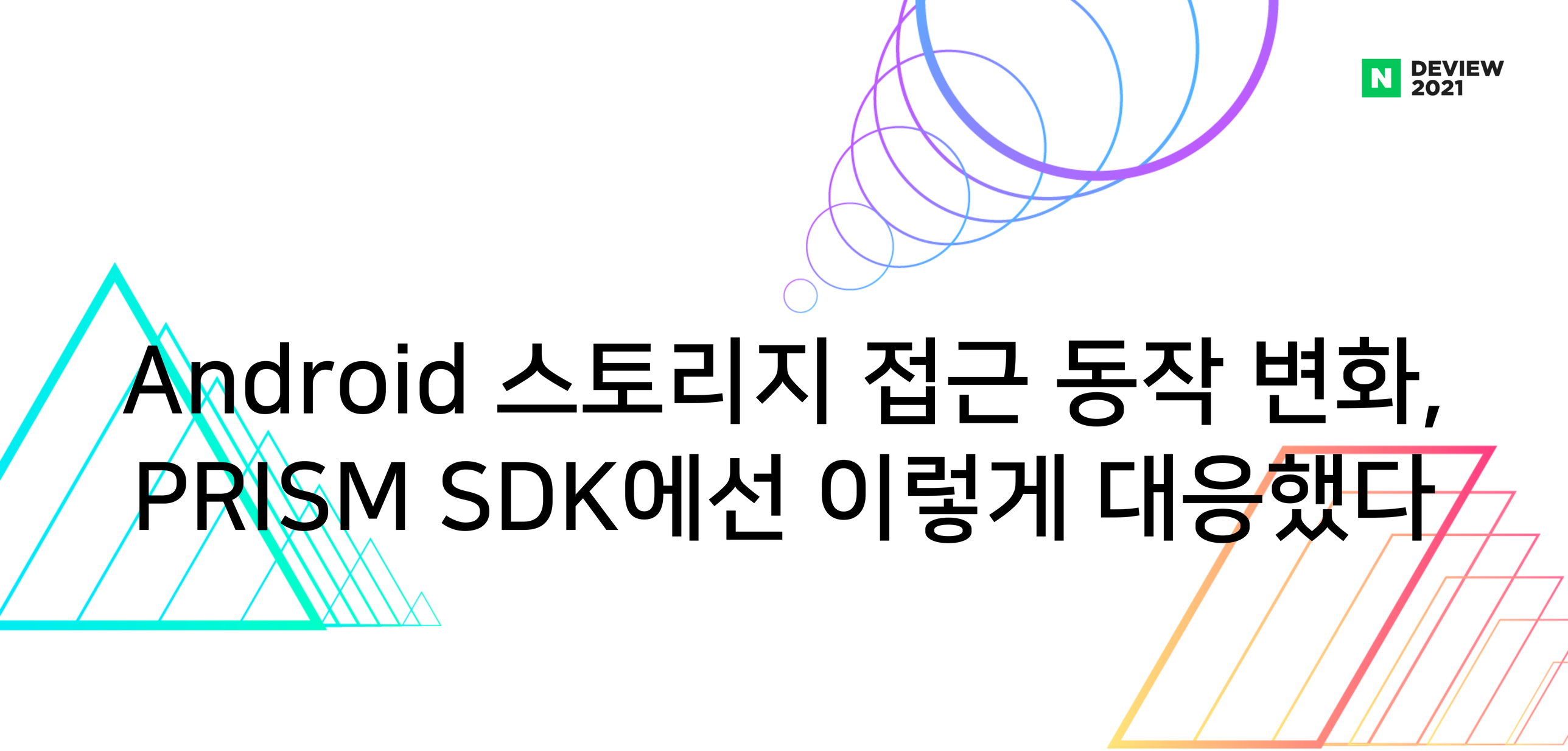


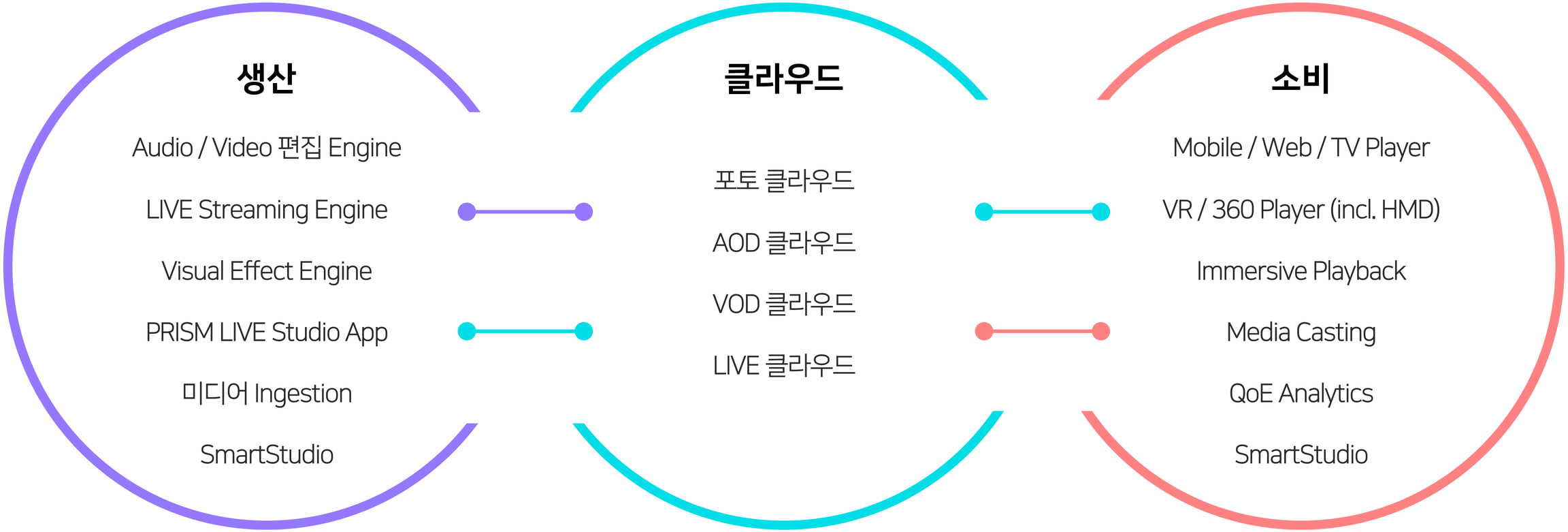
Android 스토리지 접근 동작 변화, PRISM SDK에선 이렇게 대응했다

남상규 NAVER ETECH

NAVER
Emerging
TECHnology

NAVER ETECH.

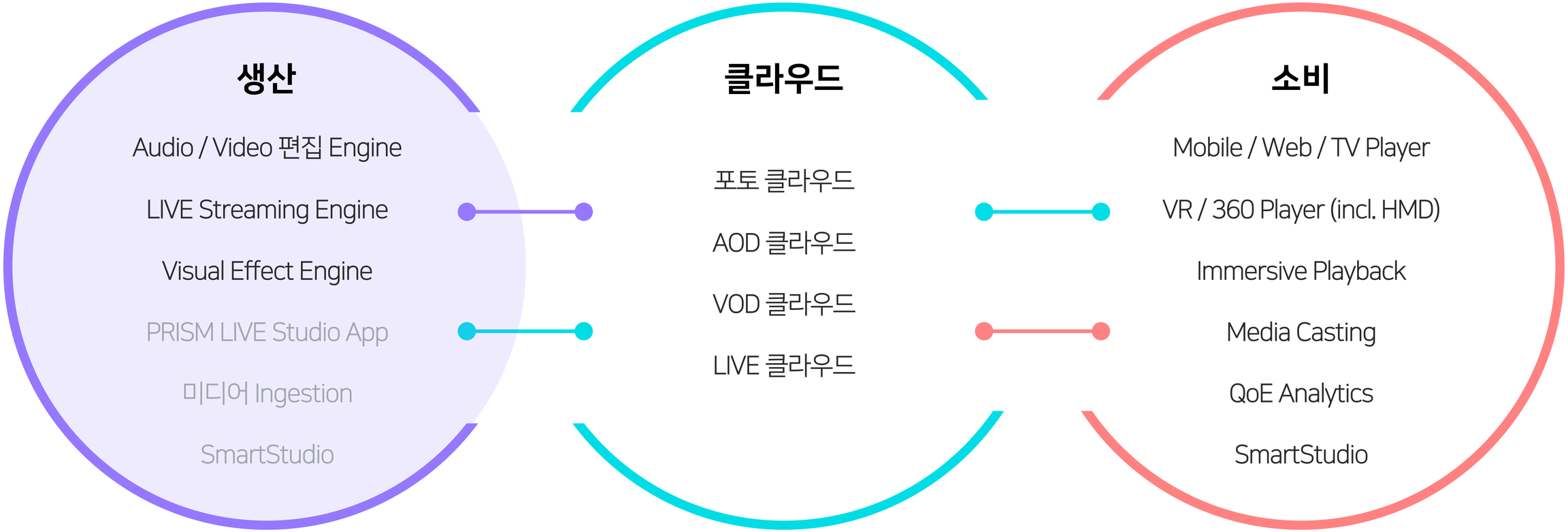
포토 / 오디오 / 비디오의 <생산 – 클라우드 – 소비> 워크플로의 전구간 기술 연구와 개발을 담당합니다.
글로벌 환경에서 시간 / 공간 / 용량의 제약 사항을 극복하고 생생한 현장 느낌과 안정적인 지원을 위해
이머징 기술 연구와 개발을 통한 원격의 시대를 준비하고 있습니다.



NAVER
Emerging
TECHnology

NAVER ETECH.

포토 / 오디오 / 비디오의 <생산 – 클라우드 – 소비> 워크플로의 전구간 기술 연구와 개발을 담당합니다.
글로벌 환경에서 시간 / 공간 / 용량의 제약 사항을 극복하고 생생한 현장 느낌과 안정적인 지원을 위해
이머징 기술 연구와 개발을 통한 원격의 시대를 준비하고 있습니다.



DEVIEW 2021

Android 스토리지 접근 동작 변화, 우리 모두의 대응

Android 9 이하

```
/sdcard/
├── Android/
│   ├── data/
│   │   ├── com.naver.alfa/
│   │   └── com.naver.bravo/
│   └── media/
│       ├── com.naver.alfa/
│       └── com.naver.bravo/
├── user-owned-file.mp4
└── ...
```

com.naver.alfa 실행 중
WRITE_EXTERNAL_STORAGE 권한 부여

com.naver.bravo

Android 10 출시, scoped storage 도입

```
/sdcard/
├── Android/
│   ├── data/
│   │   ├── com.naver.alfa/
│   │   └── ~~com.naver.bravo/~~
│   └── media/
│       ├── com.naver.alfa/
│       └── com.naver.bravo/
├── ~~user-owned-file.mp4~~
└── ~~...~~
```

com.naver.alfa 실행 중
WRITE_EXTERNAL_STORAGE 권한 부여

com.naver.bravo

특정 API 통해 사용자로부터
권한 획득 후 접근 가능

Android 10 + requestLegacyExternalStorage

```
/sdcard/
├── Android/
│   ├── data/
│   │   ├── com.naver.alfa/
│   │   └── com.naver.bravo/
│   └── media/
│       ├── com.naver.alfa/
│       └── com.naver.bravo/
├── user-owned-file.mp4
└── ...
```

com.naver.alfa 실행 중
WRITE_EXTERNAL_STORAGE 권한 부여

com.naver.bravo

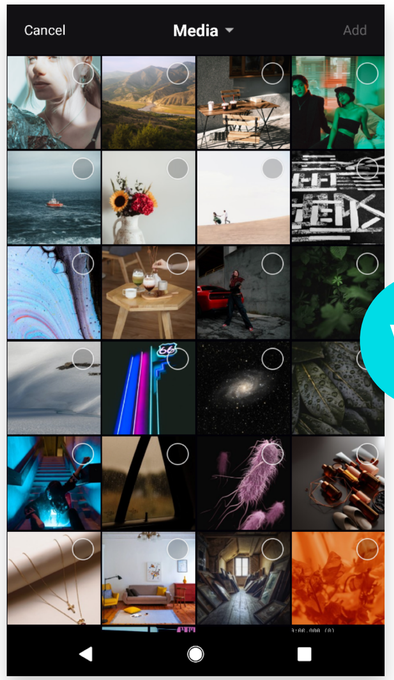
Scoped storage 사용 강제

DEVIEW 2021

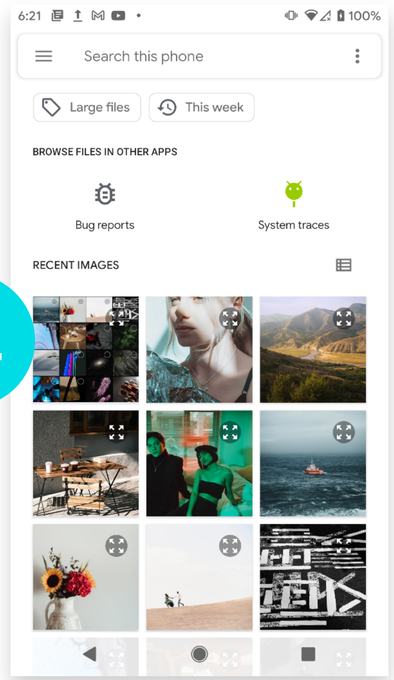
Android 11 출시, 더 이상 미룰 수 없어졌다

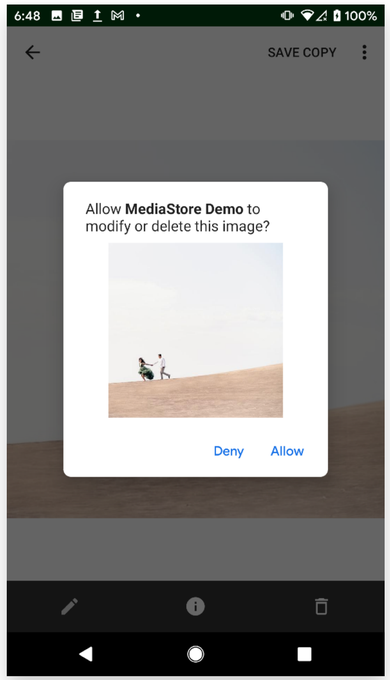
- 기기의 `SDK_INT`, 앱의 `targetSdkVersion` 모두 30 이상이면
  scoped storage 무조건 사용 (`MANAGE_EXTERNAL_STORAGE` 권한 부여 시 제외)
- Google Play 게시 신규 앱 (8월), 업데이트 (11월)
  `targetSdkVersion` 30으로 준수
- 이미 Android 11이 널리 쓰이고 있어 **기한 내 반드시 대응해야 했던 이슈**

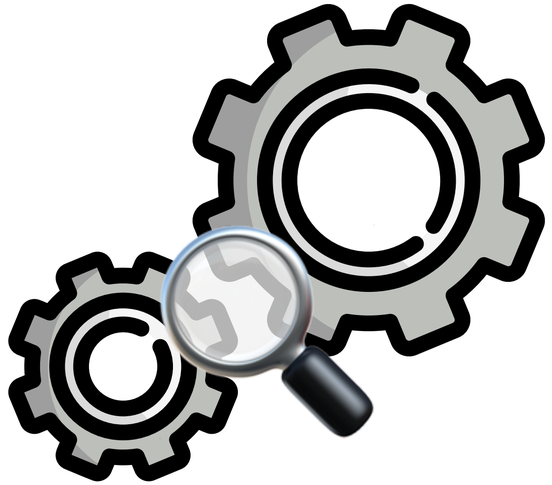
앱 개발 프로젝트의 고민거리



vs.

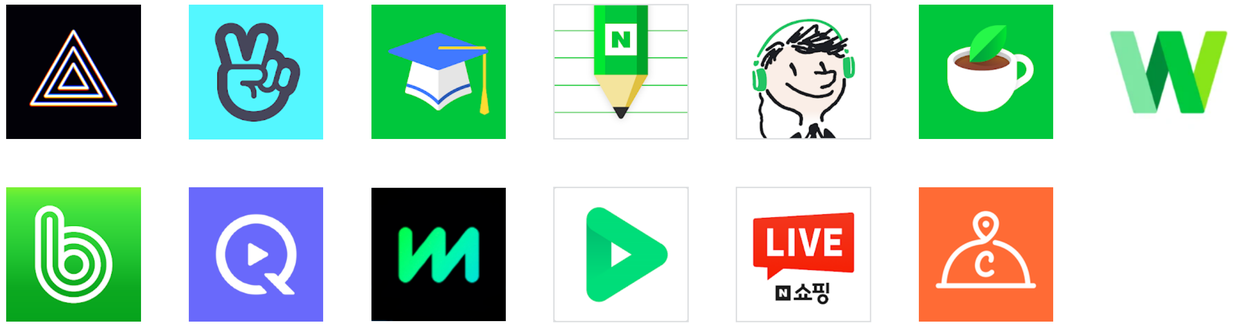






PRISM SDK

앱



PRISM SDK

Live streaming library

Video editing library

+

스토리지 접근이 빈번한 PRISM SDK 라이브러리

```kotlin
interface Control : OverlayFilter.Control {
    /**
     * 입력 영상 위 지정한 표시 영역에 나타낼 미디어 파일을 선택한다.
     */
    fun changeVideoFile(context: Context, videoFilePath: String)
}
```

```java
public class ColorFilterModel extends BaseModel
        implements Serializable {
    @SerializedName("ResourcePath")
    @Expose
    @Comparable
    private String mResourcePath;
}
```

```kotlin
data class FileConfig(
    val dir: String,
    val filename: String,
    val minStartFreeSpace: Long,
    val minFreeSpace: Long,     // in MB
    /* ... */
)
```

PRISM SDK 프로젝트에서의 대응

DEVIEW 2021

임의의 앱과 상호작용 하기에 고려했던 것

- 가능한 한 컨텐트 접근의 모든 방법 지원
- 공개 API 중 파일 I/O 관련 항목 변경 (+ 하위 호환성)
- 처리 불가능한 컨텐트에 대한 예외 처리
- 공개 API 변경을 알리는 문서 작성 (migration guide, release notes 등)

앞으로 살펴볼 내용

PRISM SDK 프로젝트에서의 스토리지 접근 동작 변화 대응 과정

- 계획 수립
- ‘파일 I/O 라이브러리’ 개발
- 기존 라이브러리 - 파일 I/O 라이브러리 연동

스토리지 접근 관련 팁

- 네이티브 라이브러리에서 컨텐트 접근
- `MediaStore` API

대응 계획 수립

계획 수립 과정

공식 문서
훑어보기

사전 지식
정리하기

수정할 부분
파악하기

계획 수립 과정

공식 문서
훑어보기

사전 지식
정리하기

수정할 부분
파악하기

왜 '훑어보기'인가?

DEVIEW 2021

계획 수립 진행을 방해하지 않는 수준의 정보만 취해야 했다

- 앱/라이브러리가 어떻게 동작해야 하는가
- 코드에서 '무슨' API를 사용해야 하는가

더욱 상세한 정보는 우리의 코드를 들여다보는 시점부터 하나씩 취했다

- 코드에서 '어떻게' API를 사용해야 하는가 - 메서드, 예외 처리, 제약사항 등

우리가 참고한 Android 공식 문서

DEVIEW 2021

릴리즈 문서

- Storage updates in Android 11
- Privacy changes in Android 10

개발자 가이드 중 ‘App data & files’ 토픽

- Data and file storage overview
- Android storage use cases and best practices

Storage updates in Android 11

Scoped storage enforcement

Apps that run on Android 11 but target Android 10 (API level 29) can still request the `requestLegacyExternalStorage` attribute. This flag allows apps to temporarily opt out of the changes associated with scoped storage, such as granting access to different directories and different types of media files. After you update your app to target Android 11, the system ignores the `requestLegacyExternalStorage` flag.

Privacy changes in Android 10

External storage access scoped to app files and media

By default, apps targeting Android 10 and higher are given scoped access into external storage, or *scoped storage*. Such apps can see the following types of files within an external storage device without needing to request any storage-related user permissions:

- Files in the app-specific directory, accessed using `getExternalFilesDir()`.
- Photos, videos, and audio clips that the app created from the media store.

To learn more about scoped storage, as well as how to share, access, and modify files that are saved on external storage devices, see the guides on how to manage files in external storage and access and modify media files.

Data and file storage overview

On this page
Categories of storage locations
Permissions and access to external storage
Scoped storage
View files on a device
Additional resources

Android uses a file system that's similar to disk-based file systems on other platforms. The system provides several options for you to save your app data:

- **App-specific storage:** Store files that are meant for your app's use only, either in dedicated directories within an internal storage volume or different dedicated directories within external storage. Use the directories within internal storage to save sensitive information that other apps shouldn't access.
- **Shared storage:** Store files that your app intends to share with other apps, including media, documents, and other files.
- **Preferences:** Store private, primitive data in key-value pairs.
- **Databases:** Store structured data in a private database using the Room persistence library.

The characteristics of these options are summarized in the following table:

	Type of content	Access method	Permissions needed	Can other apps access?	Files removed on app uninstall?
App-specific files	Files meant for your app's use only	From internal storage, `getFilesDir()` or `getCacheDir()` From external storage, `getExternalFilesDir()` or `getExternalCacheDir()`	Never needed for internal storage Not needed for external storage when your app is used on devices that run Android 4.4 (API level 19) or higher	No	Yes
Media	Shareable media files (images,	`MediaStore` API	`READ_EXTERNAL_STORAGE` when accessing other apps'	Yes, though the other app needs the	No

Android storage use cases and best practices

On this page
Handle media files
Show image or video files from multiple folders
Show images or videos from a particular folder
Access location information from photos
Modify or delete multiple media files in a single operation

To give users more control over their files and limit file clutter, Android 10 introduced a new storage paradigm for apps called scoped storage. Scoped storage changes the way apps store and access files on a device's external storage. To help you migrate your app to support scoped storage, follow the best practices for common storage use cases that are outlined in this guide. The use cases are organized into two categories: handling media files and handling non-media files.

To learn more about how to store and access files on Android, see the storage training guides.

Handle media files

This section describes some of the common use cases for handling media files (video, image, and audio files) and explains the high-level approach that your app can use. The following table summarizes each of these use cases, and links to the each of sections that contain further details.

Use case	Summary
Show all image or video files	Use the same approach for all versions of Android.
Show images or videos from a particular folder	Use the same approach for all versions of Android.
Access location information from photos	Use one approach if your app uses scoped storage. Use a different approach if your app opts out of scoped storage.
Modify or delete multiple media files in a single operation	Use one approach for Android 11. For Android 10, opt out of scoped storage and use the approach for Android 9 and lower instead.

계획 수립 과정

공식 문서
훑어보기

사전 지식
정리하기

수정할 부분
파악하기

습득한 사전 지식 (1/2)

앱에서 접근 가능한 파일 형태의 컨텐트 유형 세 가지

- `java.io.File` 등의 API로 접근 가능한 앱 소속의 app-specific files
- `MediaStore` API로 접근 가능한 media
- Storage Access Framework (SAF) 통해 접근 가능한 documents

URI - 또 하나의 컨텐트 위치 표현 방식

- 더 이상 파일 경로만으로 표현 불가

습득한 사전 지식 (2/2)

DEVIEW 2021

앱에 필요한 권한이 제각각

- OS 버전 따라, 컨텐트 유형 따라
- 하지만 권한 획득은 보통 앱에서 처리

파일 I/O 방법이 많~이 바뀐다 😱

계획 수립 과정

공식 문서
훑어보기

사전 지식
정리하기

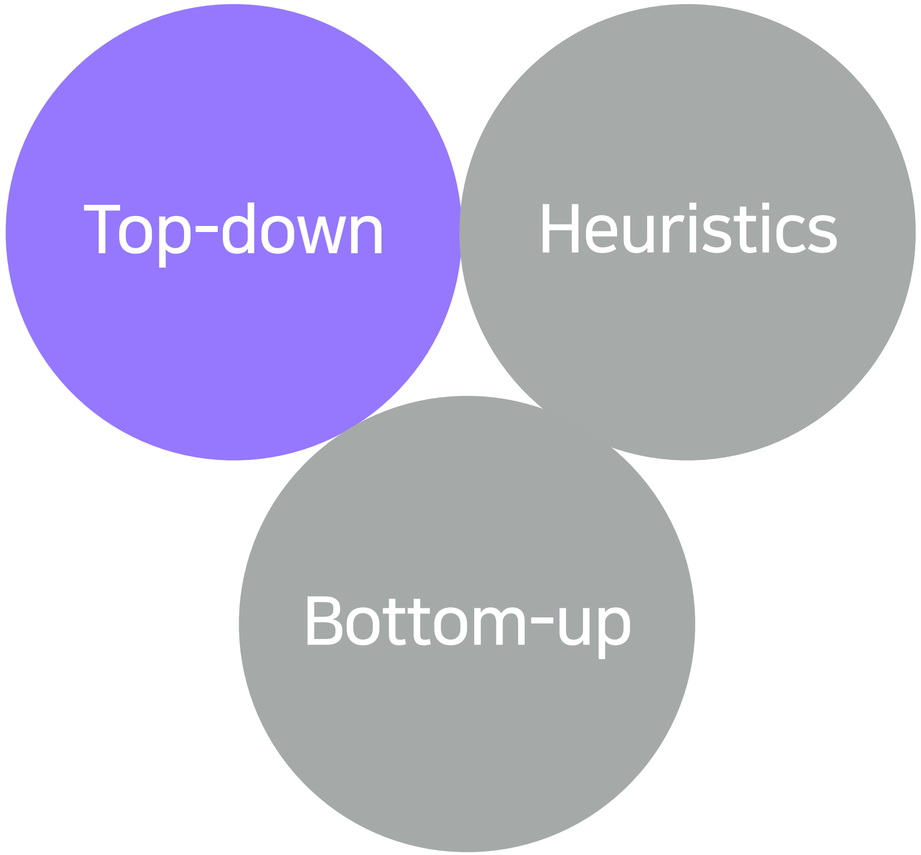
수정할 부분
파악하기

라이브러리의 파일 I/O 지점 찾기 (1/3)

DEVIEW 2021

Top-down

- API 중 파일 경로가 쓰이는 것 나열
- IDE의 ‘jump to definition’ 기능 사용
- 함수 호출 깊이가 꽤 돼서 이것만으로 전수조사는 어려움

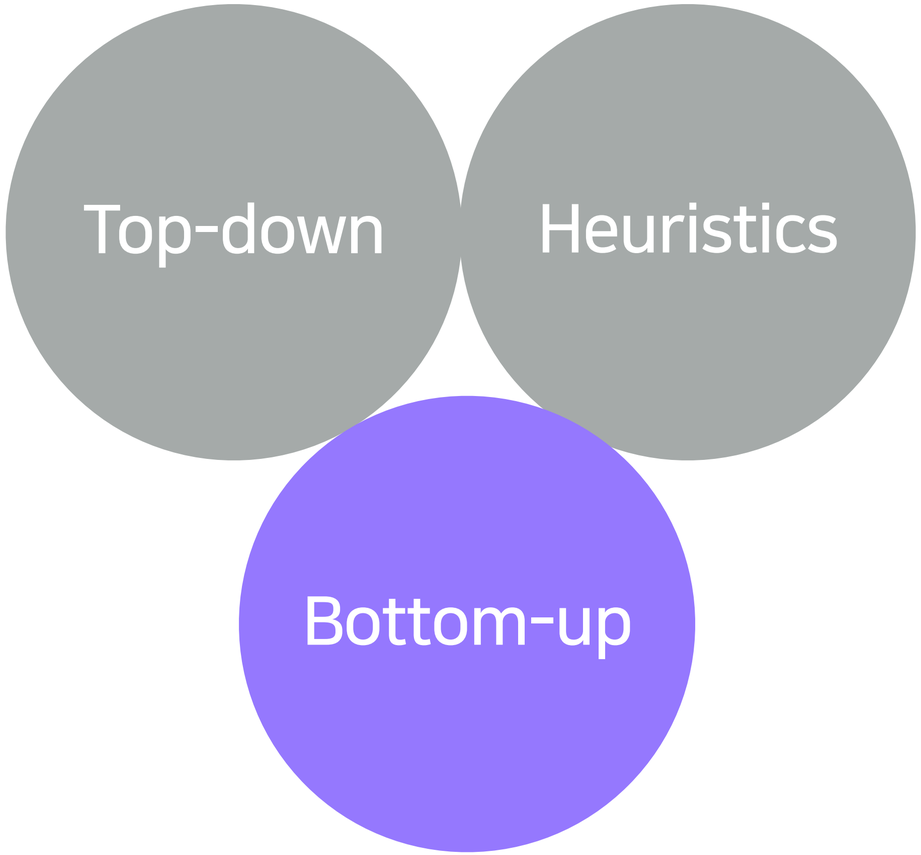


라이브러리의 파일 I/O 지점 찾기 (2/3)

DEVIEW 2021

Bottom-up

- 심볼 이름에 `file` 혹은 `path`를 포함
  ➡️ 파일 I/O와 관련된 것으로 간주
- IDE의 ‘find usages’ 기능 사용
- Top-down 방식의 결과를 보완했으나 전수조사는 어려움

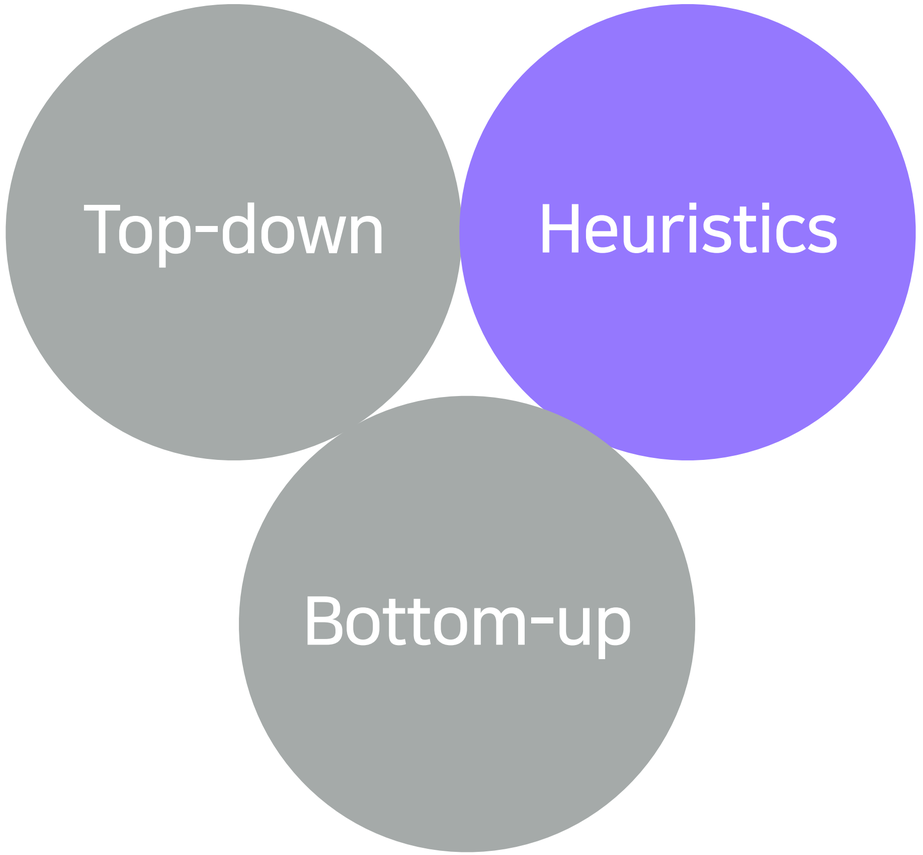


라이브러리의 파일 I/O 지점 찾기 (3/3)

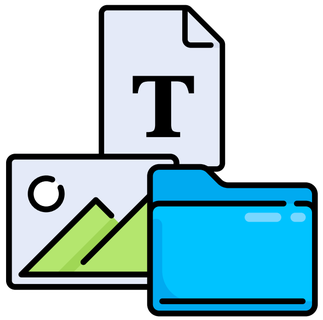
DEVIEW 2021

Heuristics

- 어느 기능에서 파일 I/O가 쓰이는지 감이 잡혔으니 경험을 활용할 차례
- Android, 네이티브 라이브러리 API 중 무엇을 파일 I/O에 쓰는지 가늠
- 소스 코드 검색을 통해 실제로 쓰이는지 확인



예상과 크게 다르지 않았다



```
File(...)
BitmapFactory.decodeStream(...)
MediaExtractor(...)
MediaMuxer(...)
ExifInterface(...)
```

```
fun fromFile(...)
fun fromAsset(...)
```

아무래도 파일 I/O 라이브러리를 짜야겠어요

파일 I/O 라이브러리 개발 동기 (1/2)

URI에 대한 파일 I/O 구현이 필요하다

- 한 곳이 아니라 여러 곳에서
- 이왕이면 앱 애셋 경로도 URI로 표현하면 좋겠다

`ContentResolver`로는 충분하지 않다

- 파일 열어 읽고 쓰는 것 외의 작업은 컨텐트 유형에 따라 다른 API를 사용해야 함
  (컨텐트 삭제, 디렉토리의 자식 나열, MIME type 획득 등)
- 앱 애셋에 접근할 수 없음

파일 I/O 라이브러리 개발 동기 (2/2)

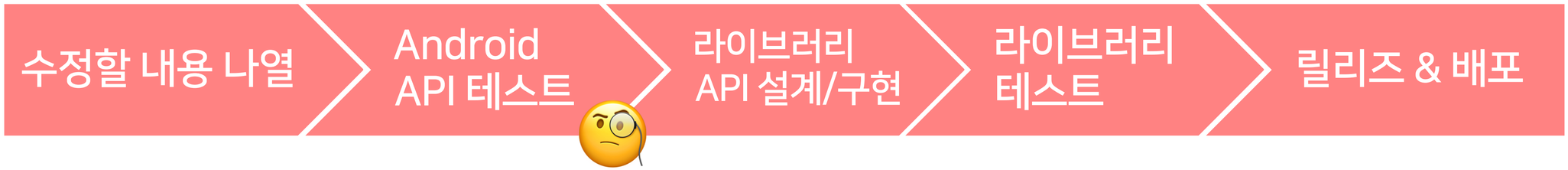
DEVIEW 2021

기존 라이브러리에 포함시키지 않는 이유는?

- PRISM SDK의 두 라이브러리 모두에서 필요함
- 두 라이브러리를 모두 쓰는 앱을 빌드 시 바이너리 크기를 줄일 수 있음

파일 I/O 라이브러리 개발

파일 I/O 라이브러리 개발 이터레이션



파일 I/O 라이브러리 개발 이터레이션

수정할 내용 나열하기

DEVIEW 2021

추가할 기능

- 입력/출력 스트림 열기
- File descriptor 열기
- 자식 컨텐트 나열
- 스토리지 여유 공간 확인

취급할 컨텐트 유형

- App-specific files
- Media
- Documents

- 앱 애셋
- 파일처럼 다룰 수 있는 앱 리소스

파일 I/O 라이브러리 개발 이터레이션

Android API 테스트..?

DEVIEW 2021

직접 써보고, 돌려봐야 한다

- Android 개발자 문서 내용에서 사용법 파악이 어려운 API 존재
- 파편화에 대비하여 instrumented test로 실제 기기 동작 파악

Android Compatibility Test Suite(CTS)이 있어도 안심할 수 없다

- CTS에서 다루지 않는 시나리오는 동작을 보장할 수 없음
- 공식 문서에 나온 시나리오가 구 버전의 CTS에 없는 경우가 실제로 존재

테스트 케이스 정의

DEVIEW 2021

(기능) × (컨텐트 유형 및 조건) → (테스트 케이스)

```
class ScopedStorageTest {
    @Test
    fun openOutputStream_writeableDocument_succeed() { /* ... */ }

    @Test
    fun getFreeSpace_appSpecificDirectory_succeed() { /* ... */ }

    @Test
    fun list_treeDocument_returnExistingDocuments() { /* ... */ }

    /* ... */
}
```

그런데 테스트를 어떻게 자동화하지..? 🤔

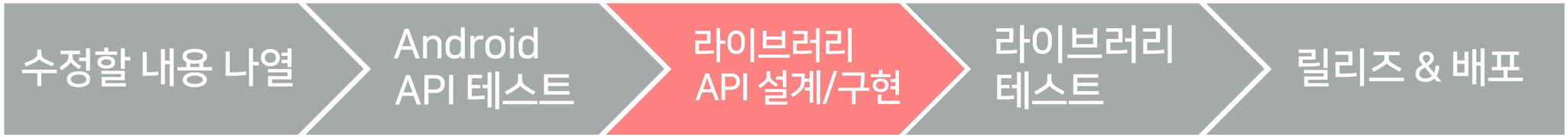
Test fixture

테스트 앱이 스스로 컨텐트를 준비/제공하도록 구현

- 텍스트 파일, 미디어 파일을 `androidTest` source set의 애셋, 리소스로 수록
- 테스트 실행 전후에 앱 애셋을 app-specific files, media로 복사, 제거
  (JUnit4 `@Before`, `@After` 사용)
- 복사한 app-specific files를 document로 제공하는 `DocumentsProvider` 구현,
  `androidTest` source set의 `AndroidManifest.xml`에 등록

자동화된 테스트에서 사용 가능한 컨텐트와 그 URI를 갖춤 👍

파일 I/O 라이브러리 개발 이터레이션



파일 I/O 라이브러리 API 설계 (1/2)

DEVIEW 2021

ContentResolver, java.io.File과 유사한 함수 선언

```
interface FileManager {
    fun openInputStream(uri: Uri): InputStream
    fun openOutputStream(uri: Uri): OutputStream
    fun openAssetFileDescriptor(uri: Uri): AssetFileDescriptor
    fun openParcelFileDescriptor(uri: Uri, mode: String): ParcelFileDescriptor
    fun listChildren(uri: Uri): List<Uri>
    fun getFreeSpace(uri: Uri): Long

    /* 뒷장에서 계속... */
```

파일 I/O 라이브러리 API 설계 (2/2)

DEVIEW 2021

Factory 함수 정의 포함

- 유일한 인터페이스 구현의 인스턴스를 생성
- 라이브러리에서 앱 애셋을 가리키는 데 쓰는 형식의 URI를 생성

```
    companion object {
        @JvmStatic
        fun create(context: Context): FileManager = /* ... */

        @JvmStatic
        fun uriForAsset(path: String): Uri = /* ... */
    }
}
```

파일 I/O 라이브러리 API 함수 동작 규칙

어느 I/O에 해당하는 함수든 이런 규칙에 따라 동작

- URI 파싱하여 컨텐트 유형 식별, 유형에 따라 적절한 Android API 사용
- 입력된 URI에 대해 수행할 수 없는 동작 ➡️ `UnsupportedOperationException`
- I/O 오류 발생 ➡️ `IOException`
  (Android API가 예외 발생이 아닌 방법으로 오류를 알리더라도)

앱 애셋 I/O 구현

AssetManager API 사용

FileManager 함수	AssetManager 함수
openInputStream	open
openOutputStream	❌
openAssetFileDescriptor	openFd
openParcelFileDescriptor	❌
listChildren	list (호출 결과를 URI로 변환해야 함)
getFreeSpace	❌

❌: 미지원

앱 리소스 I/O 구현

DEVIEW 2021

ContentResolver API 사용

FileManager 함수	ContentResolver 함수
openInputStream	openInputStream
openOutputStream	❌
openAssetFileDescriptor	openAssetFileDescriptor
openParcelFileDescriptor	❌
listChildren	❌
getFreeSpace	❌

❌: 미지원

Media와 documents 기본 I/O 구현

DEVIEW 2021

ContentResolver API 사용

FileManager 함수	ContentResolver 함수
openInputStream	openInputStream
openOutputStream	openOutputStream
openAssetFileDescriptor	❌
openParcelFileDescriptor	openParcelFileDescriptor

❌: 미지원

Media의 스토리지 여유 공간 확인 기능 구현

DEVIEW 2021

같은 스토리지에 위치한 디렉토리 탐색 후 `getFreeSpace()`

- API level 30 이상에서는 스토리지 볼륨 디렉토리 직접 획득
  (`StorageManager`, `StorageVolume`)
- 그 미만에서는 미디어 저장 경로(`DATA` column 값)의 부모 디렉토리 사용

```
val fileInSameStorage = if (Build.VERSION.SDK_INT ⩾ 30) {
    storageManager?.getStorageVolume(mediaUri)?.directory
} else {
    // getAbsolutePathOfMedia는 ContentResolver를 사용해
    // MediaStore.MediaColumns.DATA column 값을 획득한다.
    File(getAbsolutePathOfMedia(mediaUri)).parentFile
} ?: throw IllegalArgumentException("Cannot find a file in the same storage")
```

Document의 자식 컨텐트 나열 기능 (1/2)

DEVIEW 2021

입력으로 ‘tree URI’만 허용

- 한 document tree 내 모든 컨텐트에 접근 권한을 부여함
- 유형 1: document tree만 표현
- 유형 2: document tree + tree 내 document 표현

```
treeRoot/
├── dirDoc/
│   └── doc1
├── doc2
└── ...
```

`content://.../tree/treeRoot` (유형 1)

`content://.../tree/treeRoot/document/doc1` (유형 2)

Document의 자식 컨텐트 나열 기능 (2/2)

DEVIEW 2021

`DocumentsContract` API, `ContentResolver` API 사용

- Tree URI의 ‘child documents URI’에 질의하여 자식 document ID 획득
  (`DocumentsContract.buildChildDocumentsUriUsingTree`, `ContentResolver.query`)
- Tree URI와 획득한 자식 document ID를 사용하여 자식 document URI 생성
  (`DocumentsContract.buildDocumentUriUsingTree`)

```
private fun listChildDocumentUris(treeUri: Uri): List<Uri> {
    val docId = if (DocumentsContract.isDocumentUri(context, treeUri)) {
        DocumentsContract.getDocumentId(treeUri)         // 유형 2
    } else {
        DocumentsContract.getTreeDocumentId(treeUri)     // 유형 1
    }
    val childrenUri =
            DocumentsContract.buildChildDocumentsUriUsingTree(treeUri, docId)
    // queryStringColumn은 주어진 URI에 문자열 타입 컬럼을 질의하여 값을 획득하는 확장 함수.
    return contentResolver.queryStringColumn(
        childrenUri,
        DocumentsContract.Document.COLUMN_DOCUMENT_ID,
    ).map { DocumentsContract.buildDocumentUriUsingTree(treeUri, it) }
}
```

파일 I/O 라이브러리 개발 이터레이션

파일 I/O 라이브러리 테스트

DEVIEW 2021

Android API 테스트와 비슷한 구성

- (기능) × (컨텐트 유형 및 조건) → (테스트 케이스)
- 자동화를 위한 test fixture

파일 I/O 라이브러리 테스트만의 특징

- 성립 불가능한 조합에 대해 `UnsupportedOperationException` 발생하는지 확인함
- 나중에 라이브러리의 기본 동작과 예외 처리를 빠르게 파악하는 데 도움을 줌

파일 I/O 라이브러리 개발 이터레이션

기존 라이브러리 -
파일 I/O 라이브러리 연동

Fail-fast 원칙으로 설계된 PRISM SDK

이런 경우에 작업을 수행해도 괜찮을까?

- 디렉토리 접근이 필요한데 디렉토리가 아닌 형태의 컨텐트가 입력됨
- 출력 위치 설정하는데 앱 애셋 혹은 앱 리소스 위치가 입력됨
- File seeking이 필요한데 seeking 불가능한 컨텐트가 입력됨

전제 조건 검사를 위해 파일 I/O 라이브러리에 기능 추가 필요

네이티브 라이브러리 파일 I/O 수정

파일 경로 대신 file descriptor를 사용

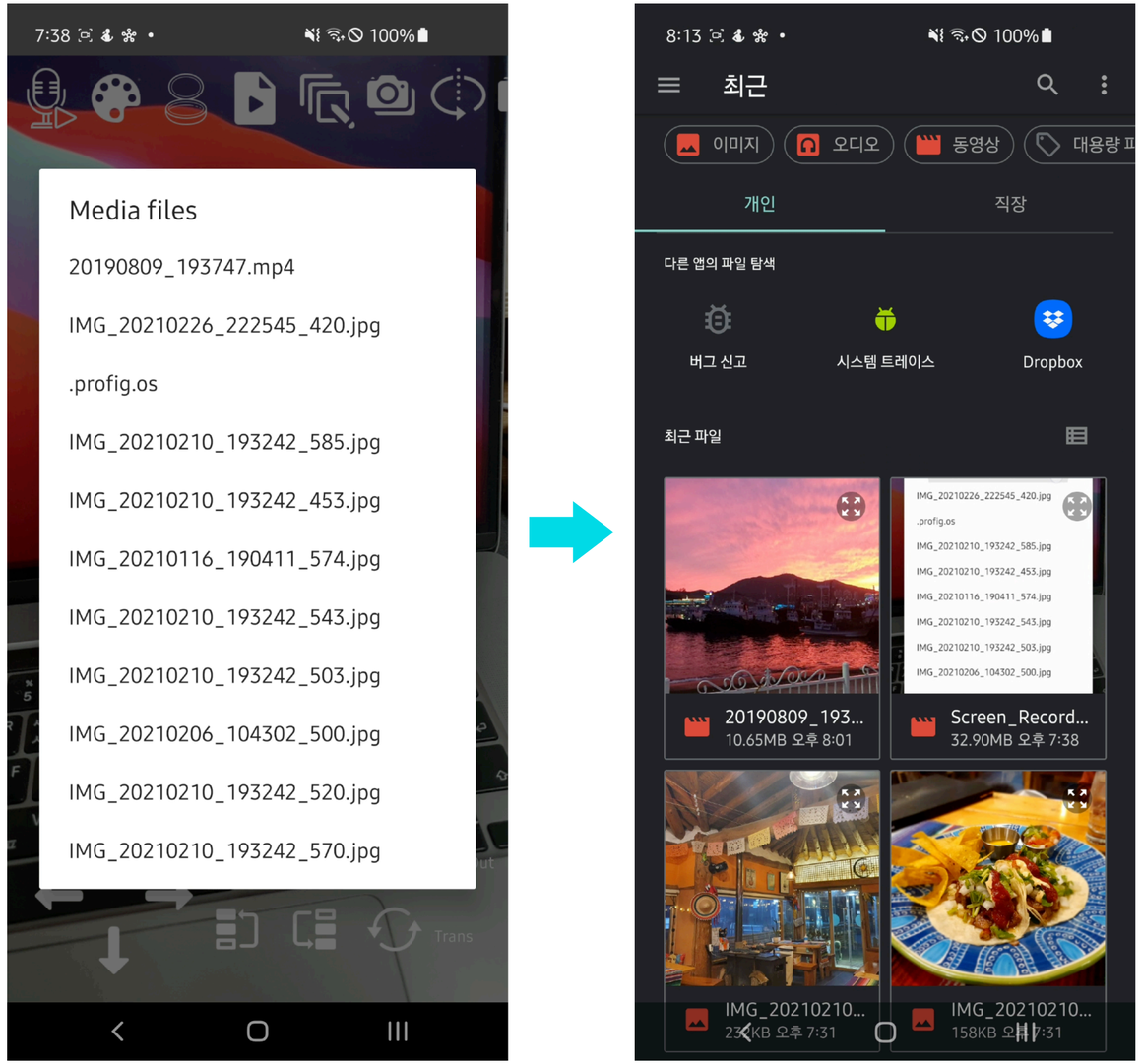
- File descriptor 미지원 API 사용 시 저수준 I/O 구현이 필요할 수 있음

FFmpeg `AVFormatContext` 객체 생성 예시

- 이전엔 URL을 인자로 받는 유틸리티 함수를 쓸 수 있었음
- `pipe` 프로토콜이 fd를 처리하지만 seeking 필요한 포맷은 처리 불가 (예: MP4)
- 내장 프로토콜 URL 대신 custom I/O 구현이 담긴 `AVIOContext` 객체 구현 필요

PRISM SDK 샘플 앱 수정

라이브러리 API 변경에 맞춰
샘플 앱의 구현과 UI도 변경



마이그레이션 가이드 작성

DEVIEW 2021

바쁜 SDK 사용자들을 위한 친절한 가이드

- 앱에 새로 추가해야 하는 코드
- API 기존 항목 - 신규 항목 간 대응 표
- API 항목 - 컨텐트 유형 간 호환성 표
- 기타 주의사항

App asset URI를 생성하는 `PrismFileManager.uriForAsset`은 이렇게 사용합니다:

```
val assetUri: Uri = PrismFileManager.uriForAsset("path/to/asset")
```

한편 URI를 인자로 받는 함수마다 처리할 수 있는 URI 종류가 다르니 함수 호출 시 주의해야 합니다. 함수마다 처리할 수 있는 URI 종류는 다음과 같습니다:

함수	File	App asset	App resource	`MediaStore` media	SAF document
`AVCaptureMgr.Builder` `.setActivationKeyUri` `.setFaceTrackModelUri` `.setSegmentModelUri`	✅	✅	❌	❌	❌
`FileConfig.Builder.setUri`	✅	❌	❌	❌ [1]	❌
`RTMPBandwidthEstimator` 생성자	✅	✅	✅	✅	✅
`FileConfig.Builder.setUri`	✅	❌	❌	❌ [1]	❌
`ColorFilter` 생성자	✅	✅	✅	✅	✅
`RadioFilter` 생성자	✅	✅	✅	✅	✅
`GifOverlayFilter` 생성자	✅	✅	✅	✅	✅
`VideoOverlayFilter.Control.changeVideoUri`	✅	✅	✅	✅	✅
`AnimationInfoParser.parse`	✅	✅	❌	❌	❌
`BeautyInfoParser.parse`	✅	✅	❌	❌	❌
`RandomStampInfoParser.parse`	✅	✅	❌	❌	❌
`StampInfoParser.parse`	✅	✅	❌	❌	❌
`StickerInfoParser.parse`	✅	✅	❌	❌	❌

- [1] 향후 지원 예정입니다. 그 전에 file URI를 사용하여 `MediaStore` API로 접근 가능한 미디어 컨텐트를 생성하려면 `Context.getExternalMediaDirs`가 반환하는 경로 아래에 파일을 만들고, 파일 쓰기가 끝난 뒤 `MediaScannerConnection.scanFile`을 호출하여 이를 스캔하는 것을 고려해 보십시오.

각 함수에서 처리 불가능한 종류의 URI, 혹은 존재하지 않는 컨텐트를 가리키는 URI를 인자로 전달하면 `IllegalArgumentException` 혹은 `IllegalStateException`이 발생하거나 SDK가 정의되지 않은 동작을 수행하므로 주의하시기 바랍니다.

스토리지 접근 관련 팁

네이티브 라이브러리에서 컨텐트 접근 (1/2)

파일 경로는 이제 놓아주고 file descriptor를 씁시다

- Media의 경우 `DATA` column 값인 파일 경로를 쓰면 성능 하락
- Document나 새로 만든 media에 대해서는 파일 경로 정의되지 않음
- Linux proc filesystem의 symbolic link (`/proc/self/fd/N`) 또한 사용 불가
- 결국 fd 받는 API를 쓰거나 low-level 구현을 추가해야 함

네이티브 라이브러리에서 컨텐트 접근 (2/2)

DEVIEW 2021

동일한 컨텐트의 fd가 여러 개 필요하다면?

- `dup` 쓰기 전에 한 번 더 생각 - file offset, file status flags 공유됨!
- 매번 새로 여는 것이 안전함
- NDK에 `ContentResolver` API가 없음 ➡️ JNI 구현 필요

MediaStore API (1/2)

Android 공식 문서의 최신 내용 중 일부는 SDK_INT별 분기 필요

- IS_PENDING column 사용 시 오류 발생 (SDK_INT < 29)
- insert 시 지정한 DISPLAY_NAME 값을 update 시에도 지정해야 값이 유지됨 (SDK_INT < 29 일부 환경)
- 파일 경로 지정 없이 삽입한 video의 스트림 혹은 file descriptor를 열 수 없음 (SDK_INT < 24)

MediaStore API (2/2)

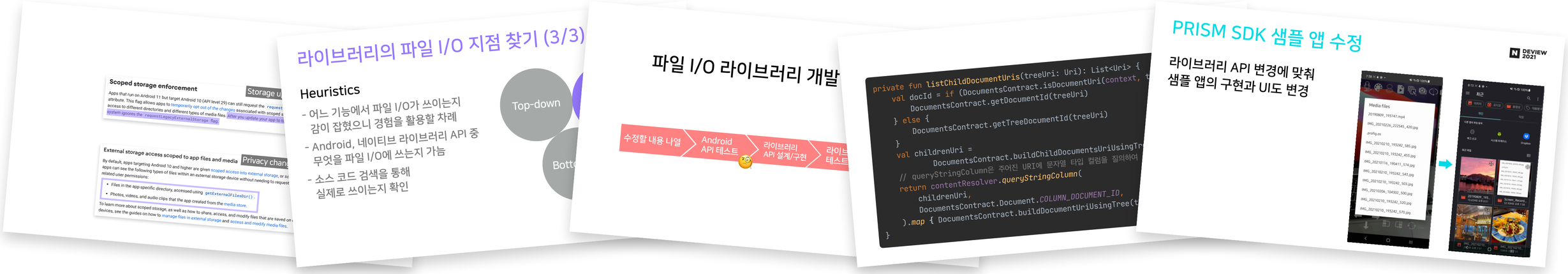
DEVIEW 2021

구 버전에서는 Android CTS 내용처럼 쓰는 게 안전하다

- Environment.getExternalStoragePublicDirectory 리턴 값을 사용하여 미디어 파일 경로 결정
- 결정된 파일 경로에 미디어 출력
- 출력을 마치고 이 경로를 DATA column 값으로 하여 media store에 삽입

DEVIEW 2021

개인정보 보호 강화를 향한 기나긴 여정 🚩



NAVER
Emerging
TECHnology

함께 성장하실 동료분을 모십니다

FE / BE / 앱 / SDK 개발 기술과 함께 멀티미디어의 핵심 기술을 배우고,

포토 / 오디오 / 비디오 / UGC 기술 도메인 전문가로 성장할 수 있도록 적극 지원하겠습니다.

ETECH 직무 소개

- ETECH 직무 소개 : https://naver-career.gitbook.io/kr/service/etech
- ETECH 기술 문의 : etech@navercorp.com
- ETECH 채용 문의 : etech-recruit@navercorp.com

Thank You