Generate 된 DX 코드를 로컬에서 테스트하기(With SAS StandAlone 모드)

개요

현재 우리가 구현해야 할 (절)대다수의 서비스는 **DX custom service** 기능을 활용하여 작성해야 한다. 그리고 서비스 코드를 생성한 뒤 PX Editor로 gen된 코드를 확인하거나 수정할 수 있는데, 이 과정에서 해당 코드에 대한 **syntax 오류**를 잡거나, **런타임 시점에서의 쿼리 작동에 대한 검증**을 하는 작업을 해야 한다.

하지만 **PX Editor 상에서는 이러한 작업이 불가능**하다. 첫 번째로 CI/CD 시 빌드 실패를 검증할 수 있는데 해당 방법으로는 어떠한 오류로 빌드에 실패했는 지에 대한 로그가 공개되지 않아서 디버깅 하기가 쉽지 않다.

또한 런타임 시점에서의 쿼리 검증 또한 실제로 SAS 환경에 배포를 한 후에 서비스 콜을 한 뒤에야 검증이 가능한데, 이는 매우 생산성이 떨어지고, 또 파드 재구동과 같은 작업이 병행되어야 하기 때문에 매우 비효율적이다.

따라서, 다음 방법을 사용하여 **로컬 환경에서 문법 오류 체크 및 해당 서비스 코드가 잘 작동하는 지를 검증** 한 후, 현재 사용중인 SAS 환경에 앱(서비스)을 배포하도록 한다.

SAS 구동에 필요한 파일 다운로드

1. http://192.168.9.12/binary/super-app-runtime/super-app-runtime-0.3.0.hotfix6 링크에 접속한다.

2. **super-app-runtime-0.3.0.hotfix6.tar** 혹은 **super-app-runtime-0.3.0.hotfix6.zip** 파일을 본인의 OS에 맞추어 다운로드한다.

3. 압축을 푼다. 해당 프로젝트의 구조에 대한 설명은 다음 링크를 참고

커스텀 서비스 작성 및 호출

1. **Java 프로젝트를 하나 생성**한다. (빌드 툴은 gradle을 사용)

2. build.gradle에 다음 코드를 추가한다.

```
repositories {
    maven {
        url "http://192.168.9.12:8081/repository/maven-releases"
```

```
        allowInsecureProtocol true
    }
    mavenCentral()
}

implementation 'com.tmax:super-app-server:0.2.5'
implementation group: 'com.google.code.gson', name: 'gson', version: '2.9.0'
```

3. **서비스 클래스를 하나 생성**한다. 필자의 경우, 예시로 request body를 그대로 response로 돌려주는 Echo 서비스를 작성했다. 이 때, 반드시 **AbstractServiceObject** 클래스를 상속받고, **service**와 **completeService** 메서드를 오버라이딩한다.

```
public class EchoServiceExample extends AbstractServiceObject {

    private static Logger logger = SuperAppDefaultLogger.getLogger(EchoServiceExample.class.getName());
    @Override
    public void service(BodyObject bodyObject) {
        JsonObject request = bodyObject.getAsJsonObject();

        DefaultBodyObject response = new DefaultBodyObject();
        JsonObject responseBody = new JsonObject();
        responseBody.addProperty( property: "response", request.toString());
        response.setJsonObject(responseBody);

        setReply(response);
    }

    @Override
    public void completeService() {}

}
```

4. 추가로, 정적인 DB source를 추가하고 싶다면 다음과 같은 클래스를 작성한다. 필자의 경우, 개발용 로컬 티베로를 설치해두어 해당 정보를 입력하였다. DbcpConfiguration 구현체를 작성하면, 추후 SAS 구동 시 자동으로 해당 구현체를 인식하여 자동으로 DBsource가 등록이 된다.

```
public class DbcpConfig implements DbcpConfiguration {
    @Override
    public DbcpProperties config() {
        return DbcpProperties.builder()
                .user("{DBMS_USER}")
                .pass("{DBMS_USER_PASSWORD}")
                .url("jdbc:tibero:thin:@{DBMS_SERVER_IP}:{DBMS_SERVER_PORT}:{DB_NAME}")
                // other optional values
                .build();
    }
}
```

5. 문법 오류를 검증하고 서비스 코드 작성(수정)을 완료 했다면, 프로젝트 루트 경로에서 다음 명령어로 **프로젝트를 빌드**한다.

```
./gradlew build
```

6. 빌드한 jar 파일을 위에서 다운로드 받은 **super-app-runtime-0.3.0.hotfix6** 디렉토리(**이하 SAS_HOME**)의 **application** 디렉토리에 넣어 준다.

7. 다시 SAS_HOME 디렉토리로 돌아가서, 다음 명령어를 입력하여 SAS를 StandAlone 모드로 구동한다. 기본 포트를 8080포트이며, 옵션을 주어 수정이 가능하다.

```
java -jar super-app-runtime.jar
```

8. WebSocket Test Client(크롬 익스텐션), 혹은 Postman을 활용하여 소켓 커넥션을 열어준 후, 해당 서비스가 정상적으로 작동하는 지를 확인한다. 호출 시 targetServiceName에는 빌드한 Jar파일명과 서비스 path를 명시한다. (참고로 빌드한 Jar파일 이름 및 버전은 build.gradle에서 수정이 가능하니 본인이 편한대로 명시한다)

```
// request

{
    "header": {
        "targetServiceName": "sas-test-1.0-SNAPSHOT/service.EchoServiceExample",
        "messageType": "REQUEST",
        "requestId": 1
    },
    "body": {
        "key_of_service_input": "echo service call"
    }
}

//response

{
    "header": {
        "targetServiceName": "sas-test-1.0-SNAPSHOT/service.EchoServiceExample",
        "messageType": "REQUEST",
        "requestId": 1,
        "traceId": "00000000FFFFFFFF0000018A835DCC7E",
        "spanId": "0000000000000015",
        "statusCode": "OK"
    },
    "body": {
        "response": "{\"key_of_service_input\":\"echo service call\"}"
    }
}
```

해당 방법을 사용하여, **DX로 gen된 자바 코드를 복사/붙여넣기** 하거나, 필요한 경우 본인이 **직접 서비스 코드를 작성**하는 방법 등으로 서비스를 구현하고 로컬에서 테스트를 진행해 볼 수 있다.

추가적인 정보는 다음 링크를 참조 : http://192.168.1.150:10081/superobject/super-object/wikis/home