

# A Developer Diary

{about:"code learn and share"}

[Home](#)[Data Science](#)[Java](#)[JavaScript](#)[jBPM](#)[Tools](#)[Tips](#)[About](#)

January 23, 2018 By [Abhisek Jana](#) — [Leave a Comment \(Edit\)](#)

## How to connect to TitanDB using Gremlin API



We will be looking into **How to connect to TitanDB using Gremlin API** in this tutorial. Interacting with TitanDB is bit different than other databases, since you can only use a WebSocket or HTTP Connection. First we will discuss on the challenges and explore a future proof way of interacting with TitanDB from Spark using Scala using Gremlin API. Then we will see how to Configure TitanDB and then use Play framework to connect to TitanDB over HTTP ( RESTful way ). We also may need to submit multiple queries together. we will learn how to combine queries in different ways.

## Challenges :

TitanDB supports two types of connections:

- WebSocket
- HTTP Connection

Most of the projects uses the HTTP Connection option, since it would be easy at times just to get the data using a REST call from the Browser itself. However Scala or Java does not have an API yet to use the JSON Object. You have Python library for the HTTP Connection and Java/Scala Library for the WebSocket. Using the RESTful option from scala is difficult however you would need good Scala REST Client which is future proof and supports SSL as well.

## Approach:

After brief research, trial and error; I found that The WS API available as part of the Play framework is not only robust but it comes with a wide variety of support. You can also use the Play REST WS API in a standalone mode without starting the entire play framework.

## How to setup Titan DB:

Setting up TitanDB is very straight forward. Just follow the instructions. Once the installation has been completed, open `...installed_dir/conf/gremlin-server/gremlin-server.yaml`, then replace the channelizer with `org.apache.tinkerpop.gremlin.server.channel.HttpChannelizer`.

```
scriptEvaluationTimeout: 30000
serializedResponseTimeout: 30000
#channelizer:
org.apache.tinkerpop.gremlin.server.channel.WebSocketChannelizer
channelizer:
org.apache.tinkerpop.gremlin.server.channel.HttpChannelizer
graphs: {
  graph: conf/gremlin-server/titan-berkeleyje-
server.properties
}
```

Once you have made the change, start the TitanDB. I am running the `titan-1.0.0-hadoop1/bin/gremlin-server.sh`

## Configure Scala:

We will use SBT to to setup the scala project. Add the following dependency to the your `build.sbt` file.

```
libraryDependencies += "com.typesafe.play" % "play-  
ws_2.11" % "2.5.10"
```

## Scala Code (Skeleton):

Now lets look at the skeleton of the Scala Code, before jumping into specifics. The `ActorMaterializer` and `AhcWClient` will initialize the components needed for the Play WS API.

We will call `submitQuery()` to invoke the REST Service. Since REST is asynchronous by nature, we need to return a `Future` interface which is part of `scala.concurrent` package.

The `Future` trait will allow us to take action after the response has been received. In this case `Future` returns a `WSResponse` object.

Once the REST Service has responded back, we need to close the client ( if we dont have any more calls to make ) and also process the response.

The `onComplete()` method will be called upon completion, we need to process the response for both success and failure scenerios.

```
import akka.actor.ActorSystem  
import akka.stream.ActorMaterializer  
import play.api.libs.ws._  
import play.api.libs.ws.ahc.AhcWClient  
import scala.concurrent.Future  
import scala.util.{Failure, Success}
```

```
object TitanDBConnector {
```

```
    import scala.concurrent.ExecutionContext.Implicits._
```

```

def submitQuery(wsClient:
WSClient, query:String, URL:String): Future[WSResponse] = {
    println(URL)
    wsClient.url(URL+query).get()
}

def main(args: Array[String]): Unit = {
    implicit val system = ActorSystem()
    implicit val materializer = ActorMaterializer()
    val wsClient = AhcWSClient()

    submitQuery(wsClient, "", "http://localhost:8182/?
gremlin=")
        .andThen { case _ => wsClient.close() }
        .andThen { case _ => system.terminate() }
        .onComplete(
            {
                case Success(response) => {

                }
                case Failure(exception) => {
                    println(exception)
                }
            }
        )
    }
}

```

## Different ways to use Gremlin:

We can invoke the REST Services using either GET or POST methods. The above example of `submitQuery` shows how to use GET to submit a single query. We can also use POST method and pass a Gremlin script.

This `submitPostQuery` would be very helpful when we have to submit multiple update or insert queries at one.

Here is an example of that.

```
def submitPostQuery(wsClient:
WSClient, query:String, URL:String):Future[WSResponse]={
    println(query)
    //The URL should be just http://localhost:8182.
    //So the below code removes the ?gremlin= from the
URL.
    wsClient.url(URL.substring(0,URL.indexOf("?")-1))
        .post("{    \"gremlin\": \"\"+query+\"\"}")
}
```

## Parse The Response:

Before talking about parsing the response, lets see example of both successfull and unsuccessful response messages.

### Unsuccessfull Response:

In case of any script error we will get following error. So we need to validate whether the message has been returned.

```
{
    "message":"Error encountered evaluating script:
g.test()"
}
```

### Successfull Response:

A successful response will contain the status and the result. The result will have the data element which will have the output of the query passed.

```
{
    requestId: "f59c4057-1cc0-46dd-b4f9-12d8d6013a51",
    status: {
        message: "",
        code: 200,
        attributes: { }
    },
    result: {
        data: [ ],
        meta: { }
    }
}
```

We can have a simple function to validate the response and return the `data` array. The function would return `None` if there are script errors.

```
def getResult(response:WSResponse):Option[JsArray]={
    println("=====")
    println(response.body)
    println("=====")

    if(response.json!=None){
        val json=response.json
        if(!(json \ "message").asInstanceOf[JsUndefined]){
            println("Error:"+ (json \ "message").get)
        }else{
            val data=(json \ "result" \
"data").get.asInstanceOf[JsArray]
            return Option(data)
        }
    }
}
```

```
None  
}
```

## Conclusion:

We can use the below boilerplate code to submit queries to TitanDB. Please refer this link to find out different ways to create the queries.

## Boilerplate Code:

```
import akka.actor.ActorSystem  
import akka.stream.ActorMaterializer  
import play.api.libs.json.{JsArray, JsUndefined}  
import play.api.libs.ws._  
import play.api.libs.ws.ahc.AhcWSSClient  
  
import scala.concurrent.Future  
import scala.util.{Failure, Success}  
  
object TitanDBConnector {  
  
    import scala.concurrent.ExecutionContext.Implicits._  
  
    def submitQuery(wsClient:  
WSClient, query:String, URL:String): Future[WSResponse] = {  
        println(URL)  
        wsClient.url(URL+query).get()  
    }  
  
    def submitPostQuery(wsClient:  
WSClient, query:String, URL:String): Future[WSResponse] = {  
        println(query)
```



```

wsClient.url(URL.substring(0,URL.indexOf("?")-1))
    .post("{      \"gremlin\": \"\"+query+\"\"}")
}

def getResult(response:WSResponse):Option[JsArray]={
    println("=====")
    println(response.body)
    println("=====")

    if(response.json!=None){
        val json=response.json
        if(!(json \ "message").asInstanceOf[JsUndefined]){
            println("Error:"+ (json \ "message").get)
        }else{
            val data=(json \ "result" \
"data").get.asInstanceOf[JsArray]
            return Option(data)
        }
    }
    None
}

def main(args: Array[String]): Unit = {
    implicit val system = ActorSystem()
    implicit val materializer = ActorMaterializer()
    val wsClient = AhcWSCClient()

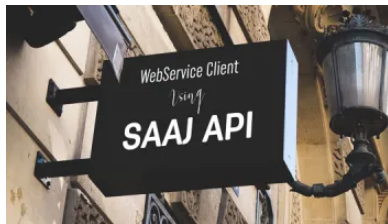
    submitQuery(wsClient,"","http://localhost:8182/?
gremlin=")
        .andThen { case _ => wsClient.close() }
        .andThen { case _ => system.terminate() }
        .onComplete(
            {
                case Success(response) => {

```

```
}  
    case Failure(exception) => {  
        println(exception)  
    }  
}  
)  
}  
}
```

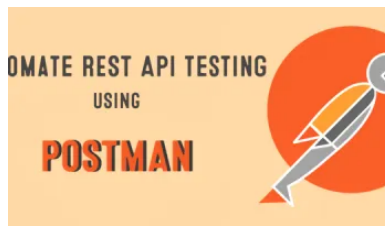
---

## Related



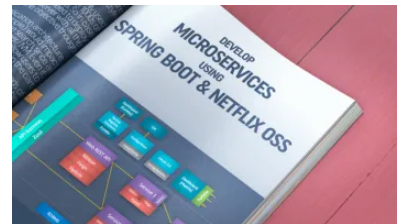
How to write Java  
WebService Client using  
SAAJ API?

In "Java"



How to Automate REST API  
JSON Schema Validation  
testing using Postman

In "REST"



Develop Microservices using  
Netflix OSS and Spring Boot

In "Microservice"

---

Filed Under: [DataBase](#) | Tagged With: [bigdata](#), [gremlin](#), [play](#), [scala](#), [titandb](#)

## Subscribe to stay in loop

\* indicates required


Email Address \*

[Subscribe](#)

## Leave a Reply

Logged in as Abhisek Jana. [Edit your profile](#). [Log out?](#) Required fields are marked \*

Comment \*



Post Comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

Copyright © 2024 A Developer Diary