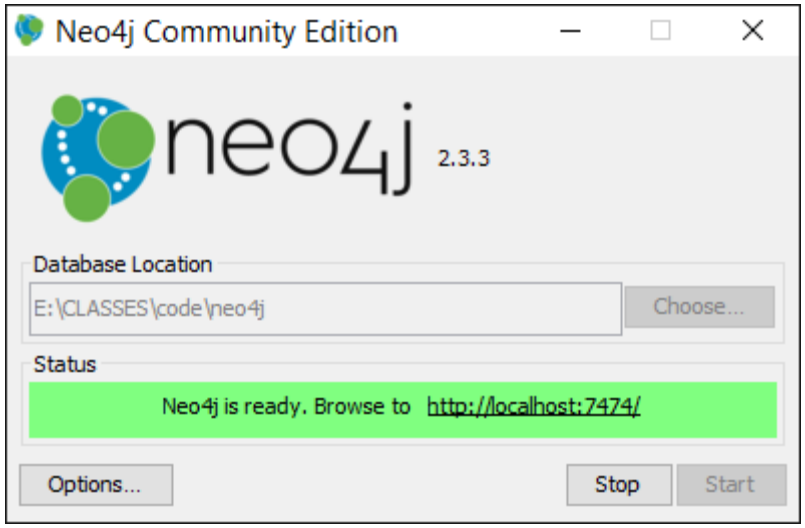
Section 10

REST API for Neo4J

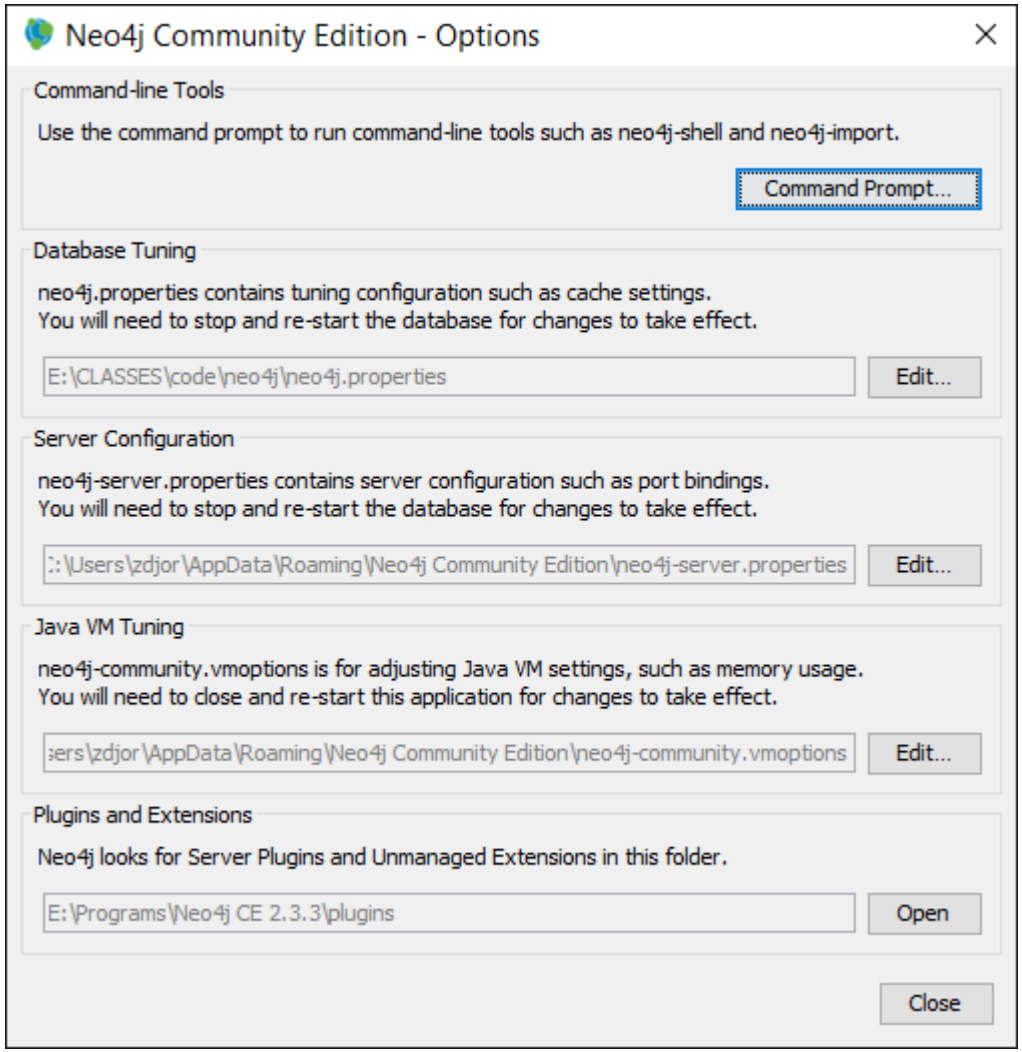
Zoran B. Djordjević

Installing Neo4J Community Edition

- Go to http://neo4j.com/download-thanks/?edition=community
- Select your operating system: Mac OS, Linux, Windows.
- Download neo4j-community_windows-x64_2_3_3.exe or similar
- Run the installation
- Select the directory
- Start

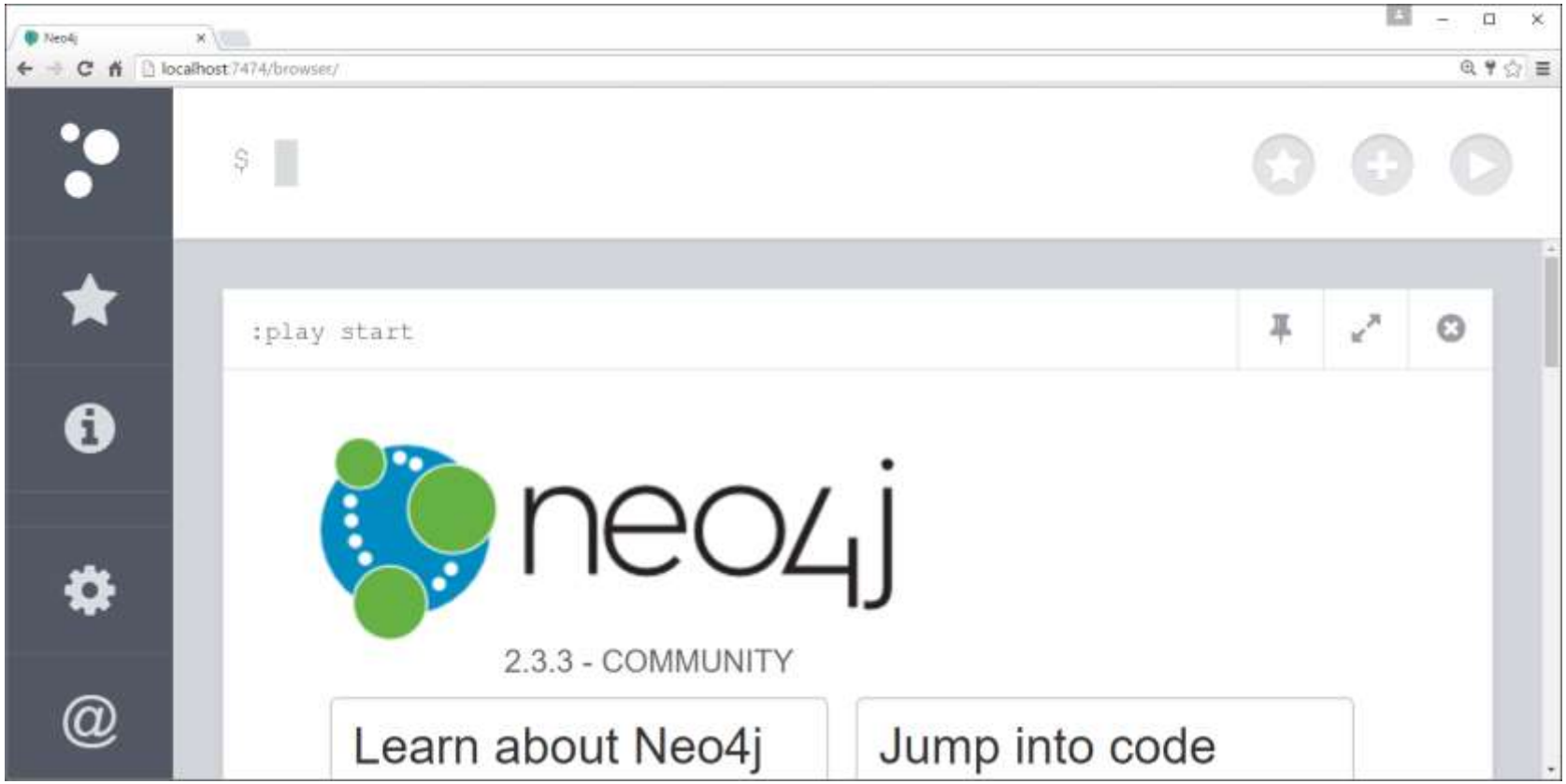


Properties files



Port 7474, neo4j/admin

- The first time you login, you might be asked to change your password.
- Username is neo4j.
- Record your password. You might need it.



Java APIs

- There are several ways of using Neo4j from Java and other languages.
- You can see a list of options on this page:

http://neo4j.com/developer/java/#_neo4j_for_java_developers

- The standalone Neo4j Server can be installed on any machine and then accessed via its HTTP API from any language. There appear to be REST libraries for many languages: Java, JavaScript, PHP, Ruby, Scala, .Net, …
- The dedicated Neo4j drivers go beyond that by offering comprehensive APIs for integrating with graph-based applications.
- One can also run Neo4j embedded in your JVM process, much like HSQL or Derby. This is great for unit testing, but also for high performance / no-network setups.
- If you use Neo4j Embedded, you can use the Neo4j Core-Java-API directly.
- Besides an object-oriented approach to the graph database, working with Nodes, Relationships, and Paths, it also offers highly customizable high-speed traversal- and graph-algorithm implementations.
- You can also choose from any useful drivers wrapping that API, which exist either for specific programming languages or that add interesting functionality.

Use REST API to communicate with the Server

- The Neo4j REST API is designed to be used with any client that can send HTTP requests and receive HTTP responses.
- Existing Neo4J REST API is designed with discoverability in mind, so that you can start with a GET and from there discover URIs to perform other requests.
- The existing APIs are subject to change in the future, so for future-proofness discover URIs where possible, instead of relying on the current layout.
- The default representation is json, both for responses and for data sent with POST/PUT requests.

- To interact with the JSON interface you must explicitly set the request header Accept:application/json for those requests that responds with data.
- You should also set the header Content-Type:application/json if your request sends data, for example when you're creating a relationship.

- The server supports streaming results, with better performance and lower memory overhead.

Neo4J REST API

- The default way to interact with Neo4j is by using REST endpoint.
- If you go to Cypher Browser and elect </> Code screen you will see that Cypher Browser is submitting its queries as REST requests.
- The Neo4j transactional HTTP endpoint allows you to execute a series of Cypher statements within the scope of a transaction. The transaction may be kept open across multiple HTTP requests, until the client chooses to commit or roll back. Each HTTP request can include a list of statements, and for convenience you can include statements along with a request to begin or commit a transaction.

- The server guards against orphaned transactions by using a timeout. If there are no requests for a given transaction within the timeout period, the server will roll it back.

Transaction Endpoint

- If there is no need to keep a transaction open across multiple HTTP requests, you can begin a transaction, execute statements, and commit with just a single HTTP request.

- Example request. Note that endpoint contains /transaction/commit, instructing the server to commit whatever it receives with this statement.

```
POST http://localhost:7474/db/data/transaction/commit
Accept: application/json; charset=UTF-8
Content-Type: application/json
{
  "statements" : [ {
    "statement" : "CREATE (n:Apple) RETURN id(n)"
  } ]
}
```

- Example response

```
201: OK
Content-Type: application/json
{
  "results" : [ {
    "columns" : [ "id(n)" ],
    "data" : [ {
      "row" : [ 18 ]
    } ]
  } ],
  "errors" : [ ]
}
```

Use `curl` to submit REST Statements

- We need to find a way to send POST request to neo4j server. Curl is one such tool. You can run curl on any OS. You can install it on Cygwin. It comes with many Python distributions, like Anaconda. Some of many curl options are

```
C:\> curl –help
Usage: curl [options...] <url>
Options: (H) means HTTP/HTTPS only, (F) means FTP only
-K,  --config FILE   Read config from FILE
-d,  --data DATA     HTTP POST data (H)
     --data-raw DATA  HTTP POST data, '@' allowed (H)
     --data-ascii DATA  HTTP POST ASCII data (H)
     --data-binary DATA  HTTP POST binary data (H)
     --data-urlencode DATA  HTTP POST data url encoded (H)
-f,  --fail          Fail silently (no output at all) on HTTP errors (H)
     --false-start   Enable TLS False Start.
 -F, --form CONTENT  Specify HTTP multipart POST data (H)
     --form-string STRING  Specify HTTP multipart POST data (H)
     --ftp-account DATA  Account data string (F)
     --ftp-create-dirs  Create the remote dirs if not present (F)
     --ftp-method [MULTICWD/NOCWD/SINGLECWD]  Control CWD usage (F)
-P,  --ftp-port ADR  Use PORT with given address instead of PASV (F)
-G,  --get           Send the -d data with a HTTP GET (H)
-H,  --header LINE   Pass custom header LINE to server (H)
-i,  --include       Include protocol headers in the output (H/F)
-o,  --output FILE   Write to FILE instead of stdout
-X,  --request COMMAND  Specify request command to use
```

Creating a Node with curl

- Let us try running the following on a single line:

```
curl -i -H accept:application/json -H content-type:application/json -XPOST
http://localhost:7474/db/data/transaction/commit -d
'{"statements":[{"statement":"CREATE (p:Strawberry) RETURN p"}]}'
```

- We are passing 2 headers accept and content-type (-H options), submitting a POST request (-XPOST), providing auto-commit URL endpoint and submitting data (-d option).
- With curl I installed on Windows I am getting an error.
- With curl that came with my Anaconda Python 3.4 visible to my Cygwin, the command runs with response.

```
$ which curl
/cygdrive/e/Programs/Anaconda3/Library/bin/curl
$ curl -i -H accept:application/json -H content-type:application/json -XPOST
http://localhost:7474/db/data/transaction/commit -d
'{"statements":[{"statement":"CREATE (p:Peach) RETURN p"}]}'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   123  100    65  100    58    245    218 --:--:-- --:--:-- --:--:--
245HTTP/1.1 200 OK
Date: Sat, 09 Apr 2016 12:52:37 GMT
Content-Type: application/json
Access-Control-Allow-Origin: *
Content-Length: 65
Server: Jetty(9.2.z-SNAPSHOT)

{"results":[{"columns":["p"],"data":[{"row":[{}]}]}],"errors":[]}
```

Verifying the result

- We can verify that our Peach node is created in Cypher Browser or run a curl command:

```
$ curl -i -H accept:application/json -H content-type:application/json -
XPOST http://localhost:7474/db/data/transaction/commit -d
'{"statements":[{"statement":"MATCH (p:Strawberry) RETURN p"}]}'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100   122  100    65  100    57    260    228 --:--:-- --:--:-- --:--:--
260HTTP/1.1 200 OK
Date: Sat, 09 Apr 2016 12:58:35 GMT
Content-Type: application/json
Access-Control-Allow-Origin: *
Content-Length: 65
Server: Jetty(9.2.z-SNAPSHOT)

{"results":[{"columns":["p"],"data":[{"row":[{}]}]}],"errors":[]}
```

“Prepare” your statements

- When passing many identical Cypher commands you should treat them as “prepared statements” and use “parameters” option. That will allow server engine to avoid discovering optimal execution plan every time and commands would run faster.
- Creation of a node in that format should read:

```
POST http://localhost:7474/db/data/transaction
Accept: application/json; charset=UTF-8
Content-Type: application/json
{
  "statements" : [ {
    "statement" : "CREATE (n:Albartos {props}) RETURN n",
    "parameters" : {
      "props" : {
        "name" : "My Bird"
      }
    }
  } ]
}
```

- We placed the JSON string in file `node01.json` in the directory where we are running curl.

Running on a single line

- Previous JSON is still manageable even on single line:

```
$ curl -i -H accept:application/json -H content-type:application/json -
XPOST http://localhost:7474/db/data/transaction/commit -d
'{"statements":[{"statement":"CREATE (p:Person {props}) RETURN
p","parameters" : { "props" : {"name":"Clint Eastwood","born":1930}}}]}'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100   233  100   100  100   133    319    424 --:--:-- --:--:-- --:--:--
424HTTP/1.1 200 OK
Date: Sat, 09 Apr 2016 15:33:28 GMT
Content-Type: application/json
Access-Control-Allow-Origin: *
Content-Length: 100
Server: Jetty(9.2.z-SNAPSHOT)

{"results":[{"columns":["p"],"data":[{"row":[{"born":1930,"name":"Clint
Eastwood"}]}]}],"errors":[]}
```

- Node is created what you could confirm by going to Cypher browser, but you need a better way to pass long statements. Use files to store your JSON statements!

Passing a file to `curl`

- To pass complicated JSON statements to curl you better do not try to type them on a single line. Place your JSON statements document into a file and then pass the file name with the following instruction:

```
curl --data "@/path/to/filename" http://...
```

- If you want to be real fancy you can do:

```
cat file.txt | curl --data "@-" `(< url.txt )`
```

- `@-` tells curl to read from `stdin`. You could also just use the redirect (< x.txt) to put in whatever you want. If you're using `bash` shell.
- curl will strip all newlines from the file. If you want to send the file with newlines intact, use `--data-binary` in place of `- - data`

`curl` reading from a JSON File

- When we run curl command with a JSON file we get the following :

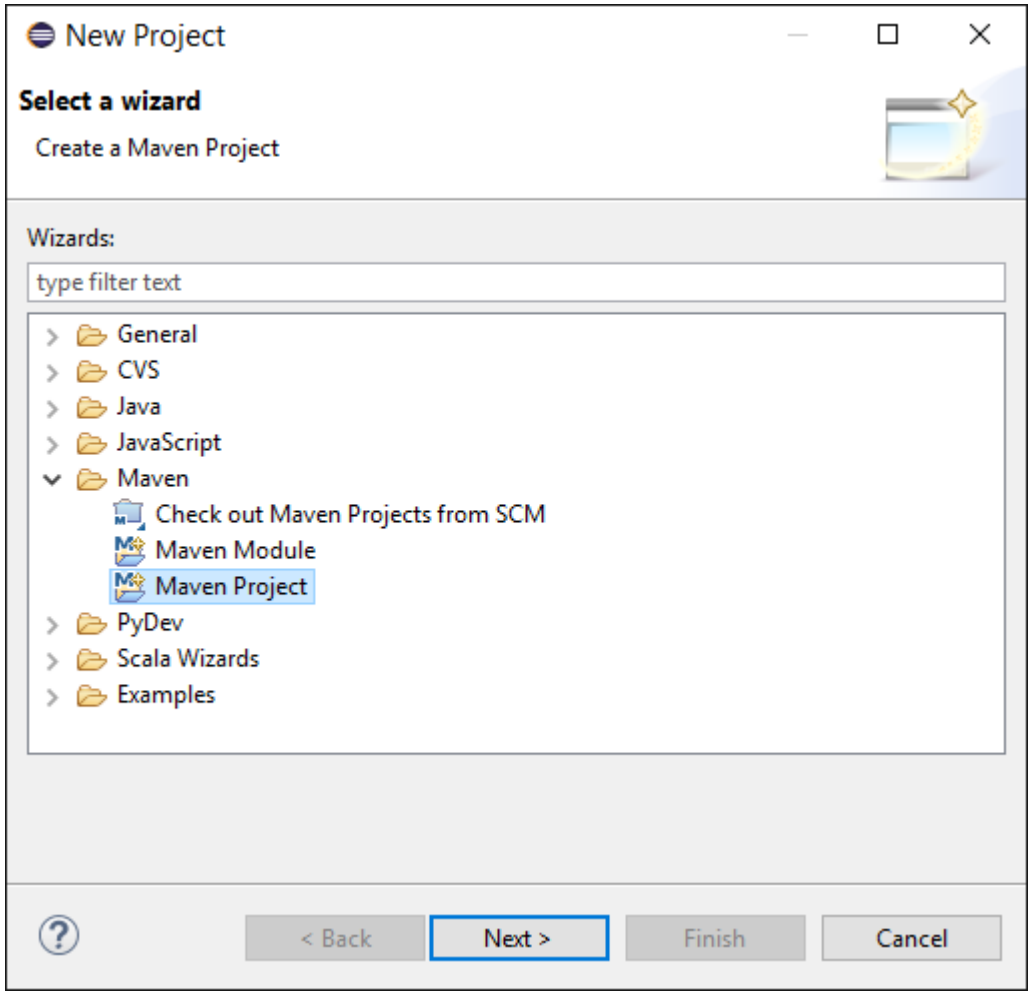
```
$ curl -i -H accept:application/json -H content-type:application/json
-XPOST http://localhost:7474/db/data/transaction --data "@node01.json"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100   354  100   204  100   150    421    309 --:--:-- --:--:-- --:--
:--   421HTTP/1.1 201 Created
Date: Sat, 09 Apr 2016 13:18:06 GMT
Location: http://localhost:7474/db/data/transaction/63
Content-Type: application/json
Access-Control-Allow-Origin: *
Content-Length: 204
Server: Jetty(9.2.z-SNAPSHOT)

{"commit":"http://localhost:7474/db/data/transaction/63/commit","resul
ts":[{"columns":["n"],"data":[{"row":[{"name":"My
Node"}]}]}],"transaction":{"expires":"Sat, 09 Apr 2016 13:19:06
+0000"},"errors":[]}
```

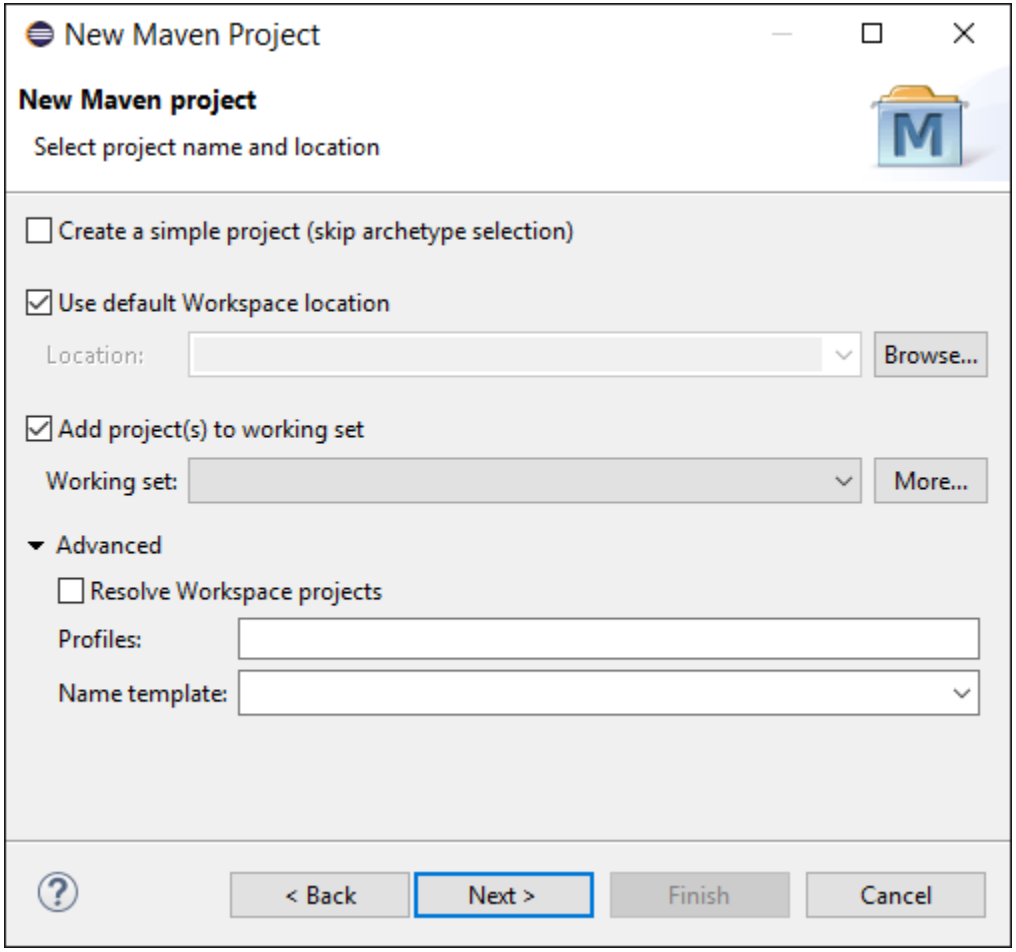
RESTful Requests from Java

Create Maven Project

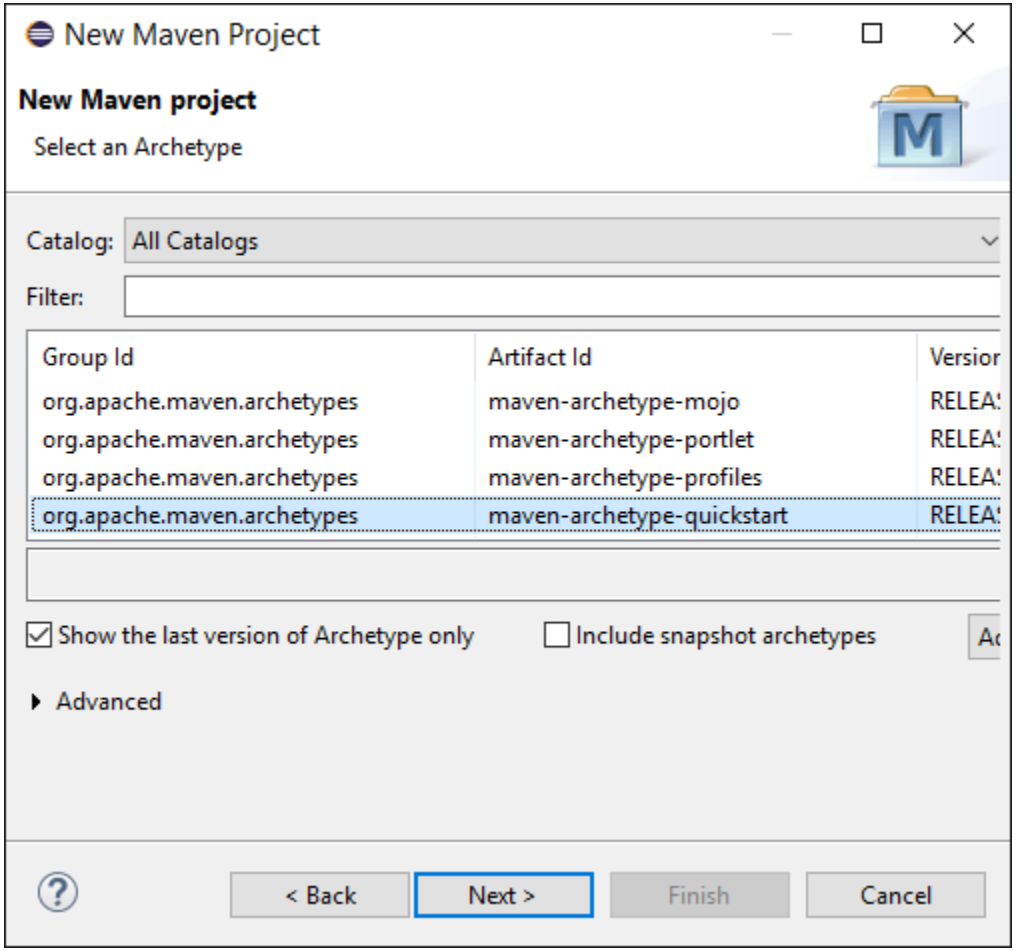
- The fastest way to prepare the IDE for Neo4j is using **Maven**. Maven is a dependency management as well as an automated building tool. In the following procedure, we will use **Eclipse**, but it works in a very similar way with the other IDEs (for Eclipse, you will need the **m2eclipse** plugin).
- When you open Eclipse, go to File > New > Project > Maven Project



Leave as is

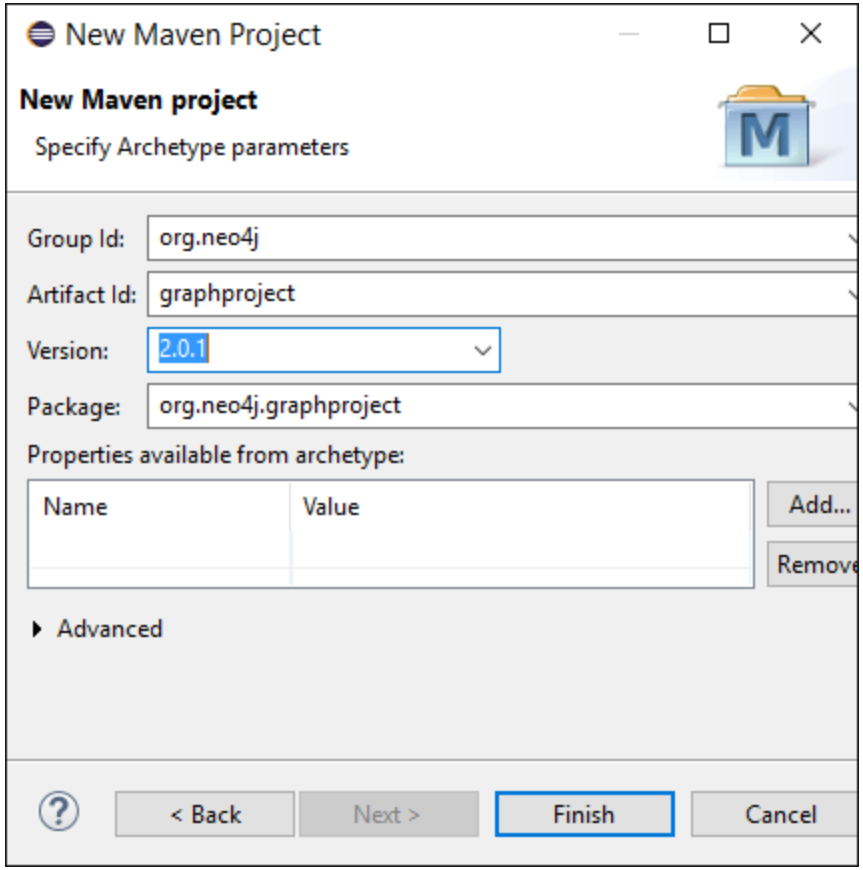

New Maven Project
New Maven project
Select project name and location
Create a simple project (skip archetype selection)
Use default Workspace location
Location:
Browse...
Add project(s) to working set
Working set:
More...
Advanced
Resolve Workspace projects
Profiles:
Name template:
< Back
Next >
Finish
Cancel

Select an Archetype



Add project name and package name

- Artifact Id will turn into the name of your Maven project.
- Group id will expand into your package name



Modify pom.xml

- Open generated pom.xml file and in `<dependencie> …</dependencies>` element add elementd:

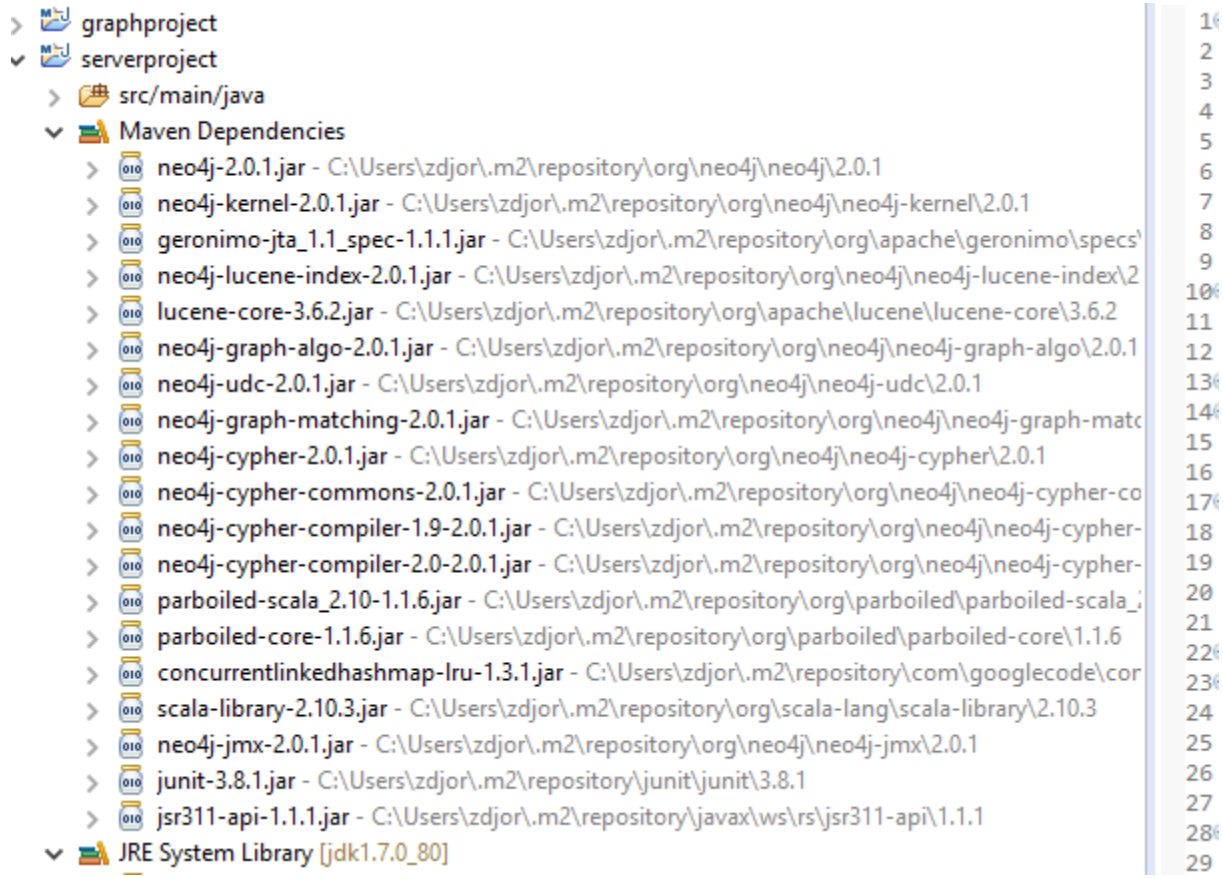
```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j</artifactId>
    <version>2.0.1</version>
</dependency>
 <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
   <groupId>javax.ws.rs</groupId>
   <artifactId>jsr311-api</artifactId>
   <version>1.1.1</version>
    </dependency>
```

- Go to `Project > Clean>` select your project name, `e.g serverproject`
- Maven dependencies should expand to include several neo4j libraries

pom.xml

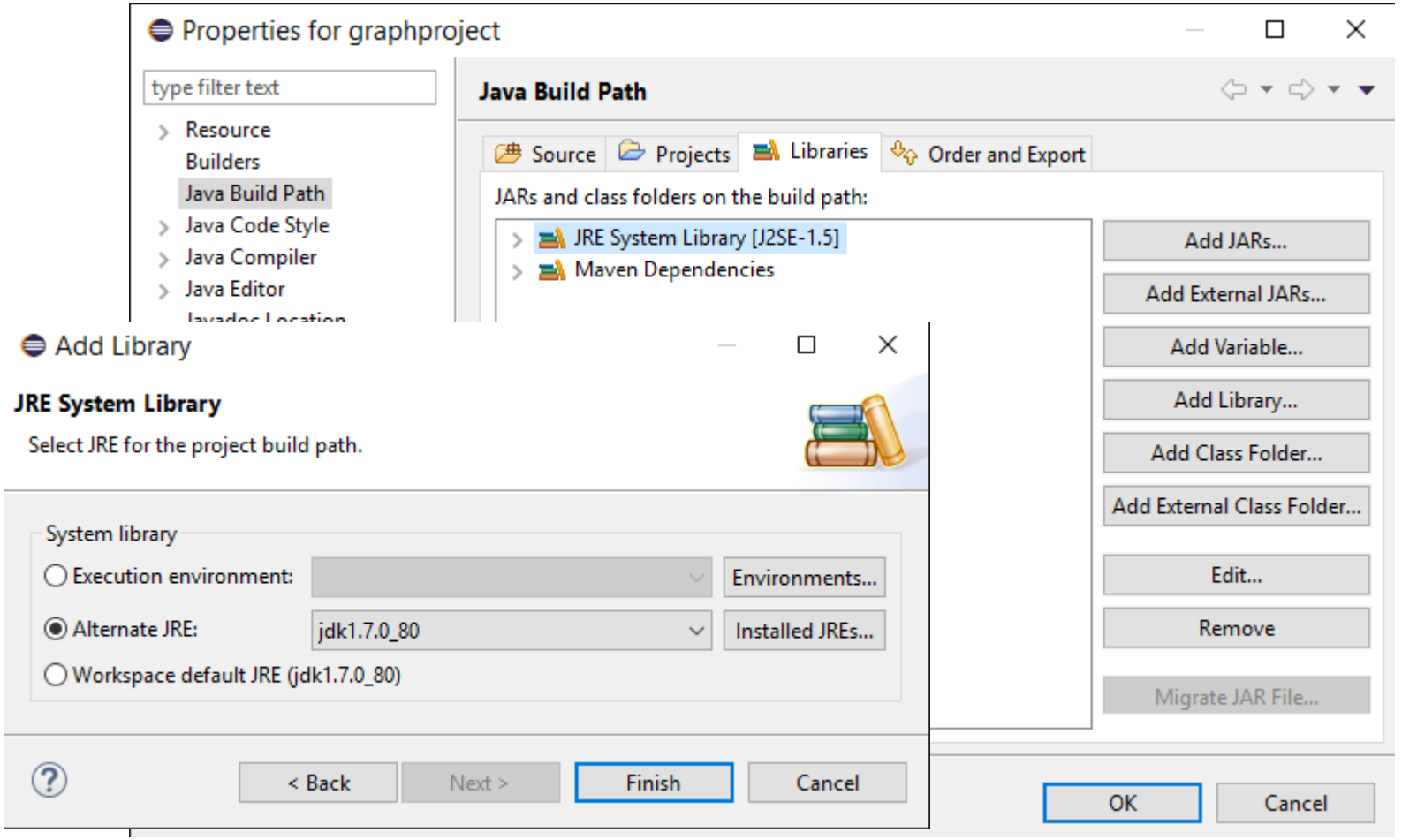
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>org.neo4j</groupId>  <artifactId>serverproject</artifactId>
 <version>2.0.1</version>
    <packaging>jar</packaging>  <name>serverproject</name>  <url>http://maven.apache.org</url>
 <properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 </properties>
 <repositories>
   <repository>
     <id>neo4j</id>  <url>http://m2.neo4j.org/content/repositories/releases/</url>
     <releases>
        <enabled>true</enabled>
      </releases>
    </repository>
 </repositories>
 <dependencies>
 <dependency>
  <groupId>org.neo4j</groupId>  <artifactId>neo4j</artifactId>    <version>2.0.1</version>
  </dependency>
  <dependency>
   <groupId>junit</groupId>   <artifactId>junit</artifactId>   <version>3.8.1</version>
   <scope>test</scope>
  </dependency>
  <dependency>
<groupId>javax.ws.rs</groupId>
<artifactId>jsr311-api</artifactId>
<version>1.1.1</version>
</dependency>
```

Maven Dependencies



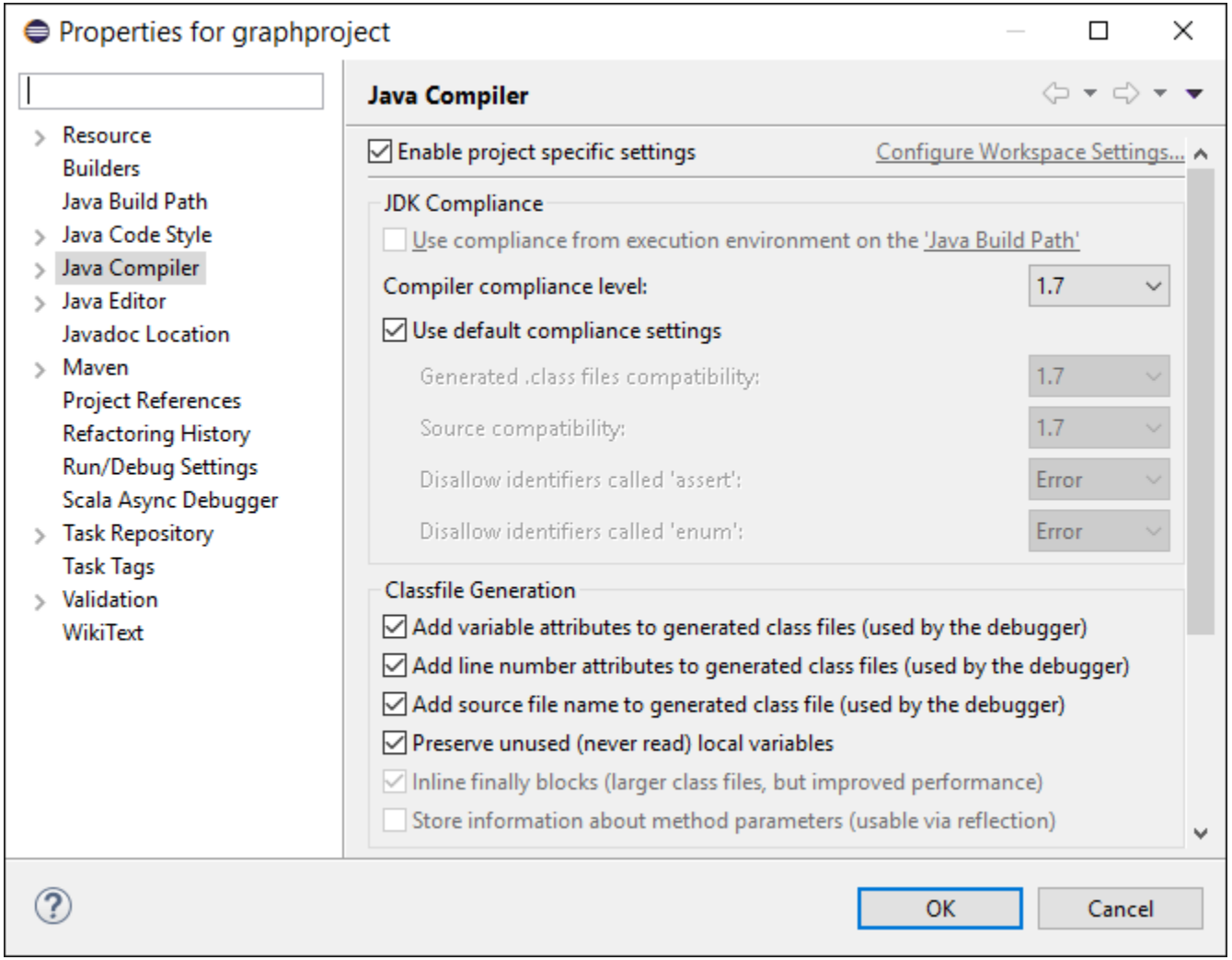
Clean BuildPath

- Right click on project, select `BuildPath > Configure BuildPath >Libraries`
- `Remove` JRE System Library [J2SE-1.5] and then `Add Library` > JRE System Library



Java Compiler

- Project > Properties > Java Compiler. Set Compiler compliance level to 1.7



REST API

- The REST API uses HTTP and JSON, so that it can be used from many languages and platforms.
- You create and manipulate a simple graph through the REST API and also how to query it.
- For these examples, we have chosen the Jersey client components.
- From https://jersey.java.net/ we downloaded Jersey bundle which contains all (most) of dependency packages so you do not have to fetch them on your own.
- We downloaded file `jersey-bundle-1.19.1.jar` and added to our project as an external library.

ConnectToServer.java

```
package edu.harvard.neo4j.server;
import java.net.URI;
import java.net.URISyntaxException;
import javax.ws.rs.core.MediaType;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;

public class ConnectToServer
{
    private static final String SERVER_ROOT_URI = "http://localhost:7474/db/data/";

    public static void main( String[] args ) throws URISyntaxException
    {
        checkDatabaseIsRunning();
    }
    private static void checkDatabaseIsRunning()
    {
        // START SNIPPET: checkServer
        WebResource resource = Client.create()
                .resource( SERVER_ROOT_URI );
        ClientResponse response = resource.get( ClientResponse.class );
        System.out.println( String.format( "GET on [%s], status code [%d]",
                SERVER_ROOT_URI, response.getStatus() ) );
        response.close();
        // END SNIPPET: checkServer
    }
}
GET on [http://localhost:7474/db/data/], status code [200]
```

Send Cypher Queries

- Using the REST API, we can send Cypher (cypher-query-lang.html) queries to the server. This is the main way to use Neo4j. It allows control of the transactional boundaries as needed. Let's try to use this to list all the nodes in the database which have a name property.
- Add method to our class

```
private static void sendTransactionalCypherQuery(String query) {
        // START SNIPPET: queryAllNodes
        final String txUri = SERVER_ROOT_URI + "transaction/commit";
        WebResource resource = Client.create().resource( txUri );
        String payload = "{\"statements\" : [ {\"statement\" : \"" +query + "\"} 
]}";
        ClientResponse response = resource
                .accept( MediaType.APPLICATION_JSON )
                .type( MediaType.APPLICATION_JSON )
                .entity( payload )
                .post( ClientResponse.class );
        System.out.println( String.format(
                "POST [%s] to [%s], status code [%d], returned data: "
                        + System.lineSeparator() + "%s",
                payload, txUri, response.getStatus(),
                response.getEntity( String.class ) ) );
        response.close();
        // END SNIPPET: queryAllNodes
    }
```

- Make a call from the main() method:

```
sendTransactionalCypherQuery( "MATCH (n) WHERE has(n.name) RETURN n.name AS name" );
```

Finding names of all Nodes

- Result of REST API Query

```
POST [{"statements" : [ {"statement" : "MATCH (n) WHERE has(n.name) RETURN
n.name AS name"} ]}] to [http://localhost:7474/db/data/transaction/commit],
status code [200], returned data:
{"results":[{"columns":["name"],"data":[
{"row":["Charlie Sheen"]},
{"row":["Keanu Reeves"]},
{"row":["Tom Hanks"]},
{"row":["Oliver Stone"]},
{"row":["Robert Zemeckis"]},
{"row":["Michael Douglas"]},
{"row":["Martin Sheen"]},
{"row":["Morgan Freeman"]},
{"row":["Keanu Reeves"]},
{"row":["Robert Zemeckis"]},
{"row":["Tom Hanks"]},
{"row":["Keanu Reeves"]},
{"row":["Robert Zemeckis"]},
{"row":["Tom Hanks"]},
{"row":["Tom Hanks"]},
{"row":["Michael J. Fox"]},
{"row":["Christopher Lloyd"]}]}],"errors":[]}
```

Creating a Node with REST API

- The REST API uses POST to create nodes. Encapsulating that in Java is straightforward using the Jersey client:

```
private static URI createNode()
    {
        // START SNIPPET: createNode
         // final String txUri = SERVER_ROOT_URI + "transaction/commit";
        final String nodeEntryPointUri = SERVER_ROOT_URI + "node";
        // http://localhost:7474/db/data/node

        WebResource resource = Client.create()
                .resource( nodeEntryPointUri );
        // POST {} to the node entry point URI
        ClientResponse response = resource.accept( MediaType.APPLICATION_JSON )
                .type( MediaType.APPLICATION_JSON )
                .entity( "{}" )
                .post( ClientResponse.class );

         final URI location = response.getLocation();
        System.out.println( String.format(
                "POST to [%s], status code [%d], location header [%s]",
                nodeEntryPointUri, response.getStatus(), location.toString()));
        response.close();

        return location;
        // END SNIPPET: createNode
    }
```

- We call this method from the main() with: `URI firstNode = createNode();`

Add Property to a Node

```
private static void addProperty( URI nodeUri, String propertyName,
            String propertyValue )
    {
        // START SNIPPET: addProp
        String propertyUri = nodeUri.toString() + "/properties/" + propertyName;
        // http://localhost:7474/db/data/node/{node_id}/properties/{property_name}

        WebResource resource = Client.create()
                .resource( propertyUri );
        ClientResponse response = resource.accept( MediaType.APPLICATION_JSON )
                .type( MediaType.APPLICATION_JSON )
                .entity( "\"" + propertyValue + "\"" )
                .put( ClientResponse.class );

        System.out.println( String.format( "PUT to [%s], status code [%d]",
                propertyUri, response.getStatus() ) );
        response.close();
        // END SNIPPET: addProp
    }
```

- We call this method from the main() with:

```
addProperty( firstNode, "name", "Joe Strummer" );
URI secondNode = createNode();
addProperty( secondNode, "band", "The Clash" );
```