Gilhari

July 2, 2024

Gilhari is a microservice framework to provide persistence of JSON objects in relational databases. This microservice framework, available in a Docker image, is configurable as per an app-specific object and relational models. Gilhari exposes a REST (REpresentational State Transfer) interface to provide APIs (POST, GET, PUT, DELETE...) for CRUD (Create, Retrieve, Update, and Delete) operations on the app-specific JSON objects.

Gilhari exchanges or transfers JSON data between an application and a database. It utilizes a light and flexible Java ORM engine called JDX to integrate with relational databases. It can be configured per app-specific requirements and the type of database used.

Notable features of Gilhari include object model independence, support for complex object modeling, including one-to-one, many-to-one, and many-to-many relationships, no requirement of code, and high portability through docker images.

1 How to use Gilhari to post data

There are a few simple steps to get the microservice working, enabling users to expose REST APIs for data manipulation.

1. Define and compile empty Java classes corresponding to types of JSON objects to be transferred.
2. Define an Object Relational Mapping specification on the model classes, mapping corresponding attributes of JSON objects.
3. Create a Dockerfile with commands to combine/configure the base Gilhari image (gilhari) with the app-specific artifacts (e.g., domain model classes, ORM specification, JDBC driver, communication ports).

4. Using the docker file, build the image of the configured Gilhari microservice.
5. Deploy and run the microservice in a container to expose the APIs for data transfer.
6. Run the curlCommands file containing commands to transfer the data between the local directory and the cloud database.

2 Some examples to get started

For starters, we have created a sample_data.json file, which contains 50 JSON objects that represent data of students in a college. It has the following attributes:

- ID: it is a unique identifier for every student
- Name: the name of the student
- Height: height of the student
- Weight: weight of the student

```
"id": 1, "name": "Aarav Kumar", "height": 172, "weight": 68,
```

2.1 Tutorial

We have a detailed explanation of the steps involved in using the Gilhari.

1. **Define and compile empty Java classes corresponding to types of JSON objects to be transferred.**
   In our example, we define a container class named JSON_student.java corresponding to the student type of JSON objects.

```
package com.mycompany.gilhari7.cloud.model.Student;
import org.json.JSONException;
import org.json.JSONObject;
import com.mycompany.jdx.JDX_JSONObject;
public class Student extends JDX_JSONObject
public Student()
super();

public Student(JSONObject jsonObject) throws JSONException
```

```
super(jsonObject);
```

2. **Define an Object Relational Mapping specification on the model classes, mapping corresponding attributes of JSON objects.**
   We define the ORM specification for the data in a mapping file (map.jdx) textually. Here is the mapping that we have used for this example:

```
CLASS com.mycompany.gilhari7.cloud.model.Student TABLE Employee
VIRTUAL_ATTRIB id ATTRIB_TYPE int
VIRTUAL_ATTRIB name ATTRIB_TYPE java.lang.String
VIRTUAL_ATTRIB height ATTRIB_TYPE float
PRIMARY_KEY id
;
```

   The names and Java types of the persistent properties (id, name, grade, section, scores) of the model Student JSON objects are shown in the above code. It also contains the JDBC driver name, database URL, target details, and login credentials.

3. **Create a Dockerfile with commands to combine/configure the base Gilhari image (gilhari) with the app-specific artifacts.**
   This file contains the commands to create an app-specific docker image of the microservice, starting with the base image and then adding configuration information.
   The following Docker file picks up the base image of Gilhari, modifies it as indicated in the config file, and creates a new image with modifications.

```
FROM dperiwal/st_repo:gilhari
#FROM gilhari:0.8
WORKDIR /opt/gilhari_simple_example


ADD bin ./bin
ADD config ./config
ADD gilhari_service.config .


EXPOSE 8081
```

```
CMD node /node/node_modules/gilhari_rest_server/
gilhari_rest_server.js gilhari_service.config
```

P.S. We haven't shown the config file it will be present in *"gilhari_simple_example\gilhari_service.config"*

4. **Using the docker file, build the image of the configured Gilhari microservice.**
   Run a docker build command with the Dockerfile described above to create the docker image for the microservice. For example:

   ```
   docker build -t StudentGilhari:1.0
   ```

   Note: the docker build command by default works on a docker file

5. **Deploy and run the microservice in a container to expose the APIs for data transfer.** Run the microservice image in a docker container using the following command to expose APIs to interact with the database.

   ```
   docker run -p 80:8081 StudentGilhari:1.0
   ```

6. **Run the curlCommands file containing commands to transfer the data between the local directory and the cloud database.**
   We have created a curlCommands file containing all the commands necessary to post the data in the sample_data.json file to Oracle 19c database.

   ```
   ./curlCommands.cmd
   ```

A previously built docker image of Gilhari can be used for subsequent deployments to expose APIs. A Kubernetes service may also be used for larger scaling and deployment. Alternatively, the Postman platform may also be used to interact with applications and databases through the Gilhari microservice.

2.2 Database specification

The Gilhari microservice can be used with any database. Care should be taken that the appropriate JDBC driver is linked in the container image corresponding to the database being used. The Gilhari framework ships with the following JDBC drivers (of the three commonly used databases) in the directory /node/node_modules/jdxnode/external_libs:

- mysql-connector-java-5.1.39-bin.jar (for MySQL version 5.7 or lower)

- postgresql-42.2.2.jre6.jar (for Postgres)
- sqlite-jdbc-3.8.11.2.jar (for SQLite 3.8 or lower)

If a different or updated database or JDBC driver is used, it has to be packaged into a corresponding .jar file in the app-specific Gilhari image in the config directory, and the location of the JDBC driver has to be specified in the configuration file.
The database URL is specified using the JDX_DATABASE tag in the ORM specification (.jdx) file. Also, ensure the correct JDBC driver name is specified in the ORM specification file.
The Oracle JDBC driver compatible with JDK8 can be downloaded from the following link:
https://download.oracle.com/otn-pub/otn_software/jdbc/234/ojdbc8.jar
The driver path has to be updated in the "jdbc_driver_path" field in "gilhari_service.config" file.

Note: The JX_HOME environment variable has to be set to the Gilhari SDK folder. The appropriate jars required to compile the empty classes will be mapped using the JX_HOME field to libs and external_libs. The field is present in "compile.cmd" and "compile.sh".

3 Concluding

This specific Gilhari configuration is meant to post data from a local JSON file to a selected database. The functionality of the Gilhari microservice is not restricted to posting data but is also capable of modifying and fetching data from databases. The complete Gilhari package can be downloaded from the official Software Tree website here. The complete documentation is available with the package, and a few sections, which are skipped in this README file, are available in the full documentation.