

Gilhari®

A RESTful JSON Persistence Microservice Framework

Beta Version 0.8

Gilhari® is a microservice framework to provide persistence for JSON objects in relational databases. **Gilhari makes JSON and relational data interchangeable.** 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.

In the context of Gilhari, a resource for the REST APIs is a conceptual mapping to a set of JSON objects.

Gilhari, meaning squirrel in Hindi, transfers (exchanges) JSON data between an app and a database.

The Gilhari microservice framework utilizes JDX, a lightweight and flexible Java ORM engine, to integrate with relational (JDBC) databases. The framework is configured for an app with its object and relational models to create an app-specific Gilhari microservice.

Gilhari can be used to leverage existing data in legacy relational databases.

Architecturally, an app-specific Gilhari microservice, whose functional focus is to exchange JSON data with a relational database, typically fits behind other app components (perhaps another microservice and/or an API gateway) that take care of business logic, authentication, and authorizations. A Gilhari microservice may be deployed in a container orchestration system like Kubernetes for scalability.

Gilhari RESTful JSON Persistence Microservice Framework Highlights:

- Object model independent
- Database agnostic
- Metadata driven
- Supports complex object modeling including one-to-one, one-to-many, and many-to-many relationships
- No code needs to be written for handling REST APIs, for data exchange (CRUD) operations, or for database schema creation
- Can leverage existing data in legacy relational databases
- Delivered in a Docker image (gilhari)
- Can provide persistence functionality for other microservices

RESTful APIs for JSON persistence:

Here are some examples of the RESTful APIs exposed by Gilhari.

REST Operation	URI	Body	Comments
GET	http://localhost:80/gilhari/v1/Employee		Retrieves all Employee objects
GET	http://localhost:80/gilhari/v1/Employee?filter=exempt=true		Retrieves all Employee objects who are exempt
GET	http://localhost:80/gilhari/v1/Employee/getAggregate?attribute=id&aggregateType=COUNT		Gets the count (aggregateType=COUNT) of the "id" attribute for all the Employee objects
GET	http://localhost:80/gilhari/v1/Employee/getObjectById?filter=id=16		Retrieves an Employee object with the id specified in the filter clause (id=16)
GET	http://localhost:80/gilhari/v1/A/access?attribute=aB	An "A" object in JSON format	Retrieves a contained (referenced) object indicated by the given 'attribute' name (aB) of an entity; The "entity" in the body needs to have only the connecting key value
GET	http://localhost:80/gilhari/v1/Employee/startStream?maxObjects=20		Retrieves the first 20 Employee objects after opening a query in the streaming mode
GET	http://localhost:80/gilhari/v1/Employee/fetchMore?maxObjects=10		Retrieves 10 subsequent Employee objects from a previously

			opened query stream (can be called multiple times until the stream is exhausted)
GET	http://localhost:80/gilhari/v1/Employee/closeStream		Closes the currently open query stream
GET	http://localhost:80/gilhari/v1/getObjectModelSummary/now		Get a summary of the information about the underlying object model, such as classes (types), attributes, primary keys, relationships, etc. May be helpful to a client application like an AI LLM to generate proper API calls.
POST	http://localhost:80/gilhari/v1/Employee/	Employee objects in JSON format	Saves one or more Employee objects specified in the Body of the request
PUT	http://localhost:80/gilhari/v1/Employee/updateEntity	Employee objects in JSON format	Updates one or more Employee objects specified in the Body of the request
PATCH	http://localhost:80/gilhari/v1/Employee?filter=id=39	{"newValues":["exempt","false"]}	Updates the exempt value to "false" for the Employee with id=39
DELETE	http://localhost:80/gilhari/v1/Employee/deleteEntity	Employee objects in JSON format	Deletes one or more Employee objects specified in the Body of the request
DELETE	http://localhost:80/gilhari/v1/Employee?filter=exempt=true		Deletes all the exempt Employee objects

Please note that the filter clause specifies the predicate (equivalent to the standard WHERE clause in a SQL statement) to qualify the objects to be operated on. So you don't have to learn a new query language to specify the filter conditions. Use a + operator to specify a space in a filter clause. You may specify the JSON attribute names in a filter clause without worrying about how the attribute names are mapped to the table column names.

More details on the syntax and semantics of the REST APIs are provided in a subsequent section. You may also find the Gilhari API details and examples in the POSTMAN documentation by importing the "**Gilhari REST API**" collection in your POSTMAN instance from the following exported file in the docs directory of the SDK:

Gilhari_REST_API.postman_collection.json

Role of JDX ORM and Java in Gilhari:

As mentioned earlier, the Gilhari microservice framework utilizes JDX, a lightweight and flexible Java ORM engine. JDX uses JDBC drivers to access relational databases. Consequently, your app-specific Gilhari microservice becomes portable to any JDBC standard compliant database even if the database does not support JSON data types natively. Further, a Gilhari microservice can create a new database schema per your JSON object model. Moreover, your apps can also work with any legacy relational data.

Although Gilhari utilizes the JDX Java ORM engine to provide persistence for JSON objects, the dependence on JDX or Java from an app-specific Gilhari microservice developer's point of view is minimal. Except for defining the "empty" container classes in Java and compiling them, there is no Java programming involved for the app-specific Gilhari microservice developer. The RESTful APIs deal with only app-specific JSON objects. The declarative mapping specification is defined in a text file and mostly uses virtual attribute mappings for the corresponding JSON objects.

How to use Gilhari?

There are five easy steps to configure and use the Gilhari microservice framework for a particular app. Each of these steps will be explained in detail later in the document.

1. Define and compile empty Java (container) classes corresponding to the conceptual types of your app-specific JSON objects (domain model classes)
2. Define a declarative Object Relational Mapping (ORM) specification on those domain model classes mapping corresponding attributes of the 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 Dockerfile, build a Docker image for the app-specific Gilhari microservice (e.g., my_app_gilhari)
5. Deploy and run the app-specific Gilhari microservice (my_app_gilhari) in a Docker container to provide the RESTful APIs for persistence of your app-specific JSON objects

Here are some details of the above steps using the example of an object model comprising just one class corresponding to the Employee type of JSON objects with the following four attributes: id, name, exempt, and compensation.

1. Define and compile empty Java (container) classes corresponding to the conceptual types of your app-specific JSON objects (domain model classes)

In this example, we define a simple Java container class named `JSON_Employee` (in a file `JSON_Employee.java`) corresponding to the Employee type of JSON objects. Here, Employee is a

conceptual name for a type of JSON objects; Gilhari does not require an actual EC6 style JavaScript class for Employee.

```
package com.mycompany.myproject.model;

public class JSON_Employee extends JDX_JSONObject {

    public JSON_Employee() {
        super();
    }

    public JSON_Employee(JSONObject jsonObject) throws JSONException {
        super(jsonObject);
    }
}
```

It is a good practice to define a container class within a package (in this case `com.mycompany.myproject.model`) to encapsulate and identify it in a modular way. The container class should extend the `JDX_JSONObject` class provided by JDX ORM, and have two constructors as shown above. This (empty) container class does not need to declare any primitive attributes.

Please note that for defining any new container class X, you just have to create an X.java file and change `JSON_Employee` to X at three places in the code above.

The container classes should be compiled (using `javac`) for **JDK version 1.8**. You may use the option “**—release 8**” of the `javac` command if you are using JDK 1.9 or higher for that purpose.

2. Define a declarative Object Relational Mapping (ORM) specification on those domain model classes mapping corresponding attributes of the JSON objects

We define an ORM specification for an app in a mapping file (e.g., `myproject.jdx`) textually. Here is the declarative ORM specification for the attributes of our Employee JSON objects:

```
CLASS com.mycompany.myproject.model.JSON_Employee

    // First declare all the persistent JSON properties
    VIRTUAL_ATTRIB id ATTRIB_TYPE int
    VIRTUAL_ATTRIB name ATTRIB_TYPE java.lang.String
    VIRTUAL_ATTRIB exempt ATTRIB_TYPE boolean
    VIRTUAL_ATTRIB compensation ATTRIB_TYPE double

    // Now provide the rest of the mapping specification for this class
    PRIMARY_KEY id
    SQLMAP FOR compensation COLUMN_NAME salary
;
```

The names and Java types of the persistent properties (`id`, `name`, `exempt`, `compensation`) of the model Employee JSON objects are declared with `VIRTUAL_ATTRIB` specifications. This mapping specification is non-intrusive to the domain model classes or to the JSON objects of your app.

Please note that if we have more types of JSON objects in our app, we define the corresponding container classes and add the ORM specification for those classes in the same mapping file.

The mapping file also contains the JDBC driver name, the JDBC URL of the target database, and the login credentials (for brevity, not shown above in the mapping snippet). The login credentials may be overridden at runtime (see Note 5 below).

Please make sure that the database is accessible from the container using the specified URL. For example, you may need to change “localhost” to an appropriate IP address applicable for your database instance.

The JDX User (Manual JDX-5.0-Manual-v1.pdf) shipped in the docs directory of the Gilhari SDK has full details about specifying the mappings for JSON objects.

3. Create a Dockerfile with commands to combine/configure the base Gilhari image (gilhari) with the app-specific artifacts

We create a text file named Dockerfile that contains commands to combine/configure the base Gilhari microservice framework with our app-specific artifacts like domain model classes, ORM specification, JDBC driver, communication ports, etc.

Let’s assuming that we have the following directory structure where angle brackets denote a directory name:

```
-- <BaseDir>

-- <bin>      (See Note 1)

-- <config>

-- myproject.jdx      (See Note 2)

-- classNamesMapping.json      (See Note 3)

-- mysql-connector-java-5.1.39-bin.jar      (See Note 4)

-- gilhari_service.config      (See Note 5)

-- Dockerfile      (See Note 6)
```

Note 1: This directory contains the compiled .class files for the object model.

Note 2: This text file contains the declarative ORM specification.

Note 3: This optional JSON format file contains the mapping between the conceptual type names of the JSON objects and the corresponding Java class names. This is to potentially simplify the usage of the class names in the URIs of the REST calls. For example, by specifying the following mapping, you may use just “Employee” in your REST APIs instead of a fully qualified Java class name:

```
{ "Employee": "com.mycompany.myproject.model.JSON_Employee" }
```

Note 4: This is a JDBC driver file for your relational database.

Note 5: This file contains various configuration information including the location of mapping files, JDBC driver, compiled .class files for the object model, and ports for the app-specific Gilhari microservice. Here is an example `gilhari_service.config` file:

```
{ "gilhari_microservice_name": "gilhari_myproject",  
  "jdx_orm_spec_file": "./config/myproject.jdx",  
  "jdbc_driver_path": "./config/sqlite-jdbc-3.50.3.0.jar",  
  "db_username": "mySecretUserName",  
  "db_password": "mySecretPassword",  
  "jdx_debug_level": 3,  
  "jdx_force_create_schema": "true",  
  "jdx_persistent_classes_location": "./bin",  
  "classnames_map_file": "config/classNamesMapping.json",  
  "gilhari_rest_server_port": 8081  
}
```

Most of the settings in the above example are self-explanatory.

Here is a brief explanation:

"jdx_microservice_name": Optional name to identify this Gilhari microservice. The name is logged on console during start up.

"jdx_orm_spec_file": The name and location of the ORM specification file (e.g., `myproject.jdx` located in the config directory), which contains the mapping specification for the persistent (container) classes used by JDX ORM.

"jdbc_driver_path": The location of the JDBC driver (.jar) file to be used for accessing the relational data source. A JDBC driver for SQLite database has already been added as a default JDBC driver in the classpath. If you are using a different database, the path to the corresponding JDBC driver (.jar) should be specified here.

"db_username": This optional setting may be used to override the database username specified in the ORM specification file.

"db_password": This optional setting may be used to override the database password specified in the ORM specification file.

"jdx_debug_level": A value in the range from 0 to 5 (e.g., 3) for getting the debug output from the JDX ORM engine. Use 0 for the most verbose output and 5 for the minimal output. A value of 3 will output all the SQL statements used by JDX. Default is 5.

"jdx_force_create_schema": Specifies if the database schema should be freshly created every time the server is run. Default is "false".

A value of "true" means yes => could be useful during development phase when the object model and the mapping are changing frequently.

A value of "false" means no => the schema will be created only the first time the service is run. Appropriate for using existing data in a legacy relational database where you don't want to discard the old data by creating a fresh schema.

"jdx_persistent_classes_location": Specifies the root location (e.g., ./bin directory) for the compiled persistent (container) classes. This could also be the path of a jar file. Used as a java CLASSPATH.

"classnames_map_file": This is an optional JSON format file consisting of the mappings between the conceptual JSON class names and the corresponding persistent (container) class names used by JDX ORM. If this file is not specified or if it does not contain the mapping for a conceptual JSON class specified in a REST URL, the specified class name in the URL is used as is by the JDX ORM engine.

"gilhari_rest_server_port": The port number (e.g., 8081) where the service listens to the RESTful requests. Default is 8081. This port may be mapped to different port number (e.g., 80) by a Docker run command.

Note 6: This (Dockerfile) text file contains the commands to create the app-specific Docker image for the Gilhari microservice starting with a base Gilhari Docker image (gilhari) and then adding the above configuration information. Here is an example of a Dockerfile file:

```
#FROM gilhari      (See Note 7)
FROM softwaretree/gilhari  (See Note 7)

WORKDIR /opt/gilhari_simple_example    (See Note 8)

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    (See Note 9)
```

Note 7: This (FROM) command sets the starting image as `gilhari`, which is the base Docker image for the Gilhari microservice. In the subsequent commands, app-specific configuration information is added on top of this base image.

Note 8: This (WORKDIR) command changes the active directory in your app-specific Docker container to `/opt/gilhari_simple_example`. This directory is used to populate the configuration information for your particular app as you can see in the subsequent ADD commands.

Note 9: This (`node`) command, specified using the CMD keyword, runs an instance of the Gilhari microservice passing it the app-specific configuration information through the command line parameter.

4. Using the Dockerfile, build a Docker image for the app-specific Gilhari microservice (e.g., `my_app_gilhari`)

Run a `docker build` command with the Dockerfile file described above, to create the Docker image for the app-specific Gilhari microservice (e.g., `my_app_gilhari`). For example,

```
docker build -t my_app_gilhari:1.0 -f ./Dockerfile
```

Note: The `docker build` command, by default, works on a local Dockerfile, if available. So the above command may be simplified as:

```
docker build -t my_app_gilhari:1.0
```

5. Deploy and run the app-specific Gilhari microservice (`my_app_gilhari`) in a Docker container to provide the RESTful APIs for persistence of your app-specific JSON objects

You may register the previously built app-specific Gilhari microservice image (`my_app_gilhari`) in a container registry for subsequent deployment in a Docker container. You may use a Kubernetes service to orchestrate the deployment and scaling of your app-specific Gilhari microservice.

You may also run the app-specific Gilhari microservice image (`my_app_gilhari`) in a Docker container from the command line as follows:

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

After the app-specific Gilhari microservice (`my_app_gilhari`) image starts running in a Docker container, you can start using the REST APIs to persist app-specific JSON objects. The REST APIs exposed by the Gilhari microservice are explained later.

Rebuilding App-specific Gilhari Docker Image:

Whenever you make any changes to the configuration of your app-specific Gilhari microservice (e.g., mapping specification, JDBC URL, new compiled classes, etc.), you must rebuild your app-specific Gilhari Docker image for those changes to be effective in the microservice. You should also stop any Docker containers running the old image of your app-specific Gilhari microservice before running the new image.

Object Caching with Gilhari:

The underlying JDX ORM provides support for caching objects to improve query performance. The current support is for read-only objects. This can be useful in situations where the objects have relatively static information and are accessed very frequently. Finding objects in the local memory cache avoids time-consuming trips to the database resulting in improved application performance as well as better throughput.

With JDX, caching can be configured at an individual class level. Caching is specified declaratively with the mapping information of a class. A cache can be pre-populated such that Gilhari (through JDX) will fetch the qualifying objects based on the given predicate to populate the cache at the startup time.

JDX looks up the cache when a singleton object needs to be retrieved. For top-level objects, cache is checked only when getObjectById () call is made.

Please see the details on the caching specification in the JDX USER manual shipped with the SDK. (**Note:** Secondary Cache is not supported in Gilhari).

Using Gilhari on Mac or Unix:

Most of the scripts (e.g., .bat or .cmd files) and documentation in the Gilhari SDK example directories are written for the Windows platform. If you are using this SDK on Mac or Unix, you should modify the scripts and other settings (e.g., PATH, CLASSPATH, JX_HOME, etc.) according to your platform. Here is a nice article about setting CLASSPATH on various platforms:

<https://howtodoinjava.com/java/basics/java-classpath/>

Mac arm 64 platform:

When using Gilhari on Mac arm 64 (mac silicon) platform, you need to specify an extra argument while running the Docker image of your app-specific Gilhari microservice as follows:

```
docker run -p 80:8081 --platform linux/amd64 my_app_gilhari
```

"--platform linux/amd64" emulates the amd64 platform on arm 64 enabling you to use it seamlessly for Gilhari.

Gilhari SDK:

The Gilhari SDK (Software Development Kit) comes with Gilhari API documentation as well as some self-contained sample apps and scripts for configuring, building, and using a Gilhari microservice. You may find the following folders useful in terms of defining model classes for different types of JSON objects (including their relationships) and defining their object relational mapping (ORM) specifications:

- gilhari_autoincrement_example
- gilhari_ecommerce_example
- gilhari_onetomany_example
- gilhari_manytomany_example
- gilhari_relationships_example
- gilhari_relationships_implicit_attribs_example
- gilhari_relationships_inline_attribs_example
- gilhari_simple_example
- gilhari_streaming_example

Each of the above folders has a comprehensive README file explaining the object model and how to configure and run the particular Gilhari microservice. Each folder also has a Dockerfile which has commands to build the Docker image for that microservice.

We highly recommend you following a similar directory structure (e.g., using the src, bin, and config directories) when developing your app-specific Gilhari microservice.

Syntax and semantics of the RESTful APIs provided by the Gilhari microservice framework:

The file **Gilhari_APIs.pdf** (located in the docs directory of the Gilhari SDK) provides details on the syntax and semantics of various APIs exposed by the Gilhari RESTful JSON Persistence Service microservice framework. It also has some sample APIs for an app-specific Gilhari microservice that deals with Employee objects.

Please note that, in response to a POST request, only the mapped attributes specified for a class in the ORM specification are saved in the database. So, if an original JSON object in the body of

a POST request has more attributes, they will be ignored - the object will essentially be trimmed when saved.

You may also find the Gilhari API details and examples in the POSTMAN documentation by importing the "**Gilhari REST API**" collection in your POSTMAN instance from the following exported file in the docs directory of the SDK: *Gilhari_REST_API.postman_collection.json*

The Postman collection could be useful to understand the syntax and semantics of the Gilhari REST API calls. However, if you want to invoke any APIs of the Postman collection, you should have an appropriate app-specific Gilhari microservice running to respond to the REST calls. Also, please make sure that the port number is correct in the URL.

Database and JDBC Driver considerations:

Gilhari microservice framework integrates with a relational database using a JDBC driver. For an app-specific Gilhari microservice running in a container, the JDBC driver will reside in the container image and the database will typically be running outside the container.

For this version of Gilhari, please use a JDBC driver that is compatible with JDK 1.8 (JDK 8). Also, use a JDBC driver that is compatible with the version of your target database. Otherwise, Gilhari may experience database connection problems.

For quick prototyping purposes, you may also use a SQLite database that can reside and run in the file system of the container. A SQLite JDBC driver is already packaged in the Docker image of the Gilhari framework and has been added as a default JDBC driver in the classpath. **Please note that the SQLite database and its JDBC driver are not appropriate for serious work or testing as the error handling and multi-threading support in them are not very robust.**

Database Specification:

The database URL is specified using the JDX_DATABASE tag in the ORM specification (.jdx) file. Please make sure that you are using the correct URI for your target database.

Sometimes, the "localhost" specification would not work from inside a container as the database is running outside the container "host"; you have to specify the IP address of the system where your database is running. Alternatively, you may try using "host.docker.internal" to access "localhost". For example,

JDX_DATABASE

JDX:jdbc:mysql://**host.docker.internal**:3306/JDXTestDB?useSSL=false;allowPublicKeyRetrieval=

```
true;USER=username;PASSWORD=myPassword;JDX_DBTYPE=MYSQL;DEBUG_LEVEL=5
```

The above specification of **JDX_DATABASE** in the mapping (.jdx) file should be in one line (without any line breaks).

For using a remote database (e.g., a Supabase database connected through a Postgres JDBC driver), you may have to specify an IPv4 network compatible JDBC URL. For example,

JDX_DATABASE

```
JDX:jdbc:postgresql://aws-0-us-west-1.pooler.supabase.com:5432/postgres?user=postgres....
```

Please make sure that you use the delimiter of a semicolon (“;”) between various elements of the JDX_DATABASE specification. So, for example, don’t use an ampersand (“&”) or comma (“,”) between “USER=username” and “PASSWORD=myPassword”.

If any firewall or remote connection issues are indicated in an error message, please configure your systems to take care of that.

Also, please make sure that the correct JDBC driver name corresponding to your database has been specified with the JDBC_DRIVER tag in the ORM specification (.jdx) file.

The JDX_DATABASE tag in the ORM specification (.jdx) file also contains the login credentials (name/password) for the database. You should change them as per your setup in the ORM specification file itself or you may override those credentials in the Gilhari microservice configuration (e.g. gilhari_service.config) file. See Note 5 above.

Please make sure that the appropriate JDX_DBTYPE is specified in the JDX_DATABASE tag. For example, **JDX_DBTYPE=MYSQL** or **JDX_DBTYPE=POSTGRES**, etc.

JDBC Driver:

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-42.7.5.jar (for Postgres JDBC version 4.2)

postgresql-42.2.2.jre6.jar (for Postgres JDBC version 4.0)

sqlite-jdbc-3.50.3.0.jar (for most SQLite versions higher than 3.8)

sqlite-jdbc-3.8.11.2.jar (for SQLite 3.8 or lower)

So, if any of these pre-packaged JDBC drivers is suitable for your target database, you may specify the appropriate JDBC driver in the configuration file (e.g., `gilhari_service.config`) for your app-specific Gilhari microservice.

For example, if you are using a MySQL database (version 5.7 or lower), and want to use a pre-packaged MySQL JDBC driver, you may specify:

```
"jdbc_driver_path": "/node/node_modules/jdxnode/external_libs/mysql-connector-  
java-5.1.39-bin.jar",
```

However, if you want to use a different (or a more updated) database or a JDBC driver, you should package the corresponding .jar file in your app-specific Gilhari image (e.g., in the config directory of your app) and then specify that location for the JDBC driver in the configuration file (e.g., `gilhari_service.config`). For example,

```
"jdbc_driver_path": "config/myDatabaseJDBCDriver.jar",
```

if you want to use another kind of database (e.g., Oracle or SQL Server), you would need to download and use a JDBC driver appropriate for that database. Please use the correct name and the location of the JDBC driver in various configuration or script files.

Also, if the name of the class that implements `java.sql.Driver` in the updated JDBC driver of your database has changed, please use the specified name for the updated JDBC driver. For example, suppose the name of the class that implements `java.sql.Driver` in the newer MySQL Connector/J has changed from `com.mysql.jdbc.Driver` to **`com.mysql.cj.jdbc.Driver`**. So, if you are using the newer JDBC driver of MySQL, you should specify the `JDBC_DRIVER` in the mapping specification (.jdx) file as follows:

```
JDBC_DRIVER com.mysql.cj.jdbc.Driver
```

Database Connection Errors:

If you encounter database connection errors even after making the suggested changes for the Database Specification and JDBC Driver described above, the following might help:

- **IMPORTANT:** Don't forget to update the Docker image of your app by rebuilding the image after making the suggested changes. Then deploy and run the new image.
- Start (or restart) the database server (service) if it is not running.
- **netstat command**
The `netstat` command (available on all the platforms including Windows, Mac, and Unix) displays active TCP connections, ports on which the computer is listening. It can help you check which database servers are listening on which port on your machine.

- Before running the updated image of your app-specific Gilhari microservice, remove the old containers with the earlier images as they may be blocking some communication ports. A good resource on the topic: <https://spacelift.io/blog/docker-stop-container>
- Sometimes, running Gilhari with `jdx_debug_level` of 3 can be very helpful in diagnosing many problems as that would emit all the SQL statements generated and used by Gilhari. See Note 5.
- Any error messages emitted by the Gilhari microservice may point to a specific area to investigate further.

Using Legacy Data in an Existing Database:

You may use Gilhari with existing data in a legacy relational database where the schema (tables) is already created and the tables might contain useful data.

Normally, when JDX, the underlying ORM for Gilhari, does not find a `JDXMetadata` table during the first connection to the database, it assumes that it needs to create the schema (tables) for the mapped classes in the mapping specification and, as part of that process, it drops the old tables, if any, and creates them fresh again. This behavior may be reasonable if JDX is being used for a brand new database schema or when the existing data is not important enough to be retained.

However, if the legacy tables and data should be retained and be used by Gilhari through JDX, you should just pre-create (only once) the `JDXMetadata` table in the database using the following SQL command:

```
CREATE TABLE IF NOT EXISTS JDXMetadata (jdxORMId TEXT, jdxTimestamp
TEXT, jdxMetaVersionId TEXT, jdxMetaFileName TEXT, jdxMetaInfo TEXT)
```

The existence of the `JDXMetadata` table in the database will prevent JDX from creating a new schema for the mapped classes. **However, beware that if you use a “true” value for the `jdx_force_create_schema` property (`"jdx_force_create_schema": "true"`) in the Gilhari microservice configuration file (e.g., `gilhari_service.config`), any existing tables for the mapped classes will be dropped and recreated.**

Reverse-engineering an existing relational schema to use with Gilhari:

You may use the reverse-engineering feature of JDX ORM such that given a template file identifying some existing tables in a database, JDX can generate the corresponding class definitions and the mapping specification. You may find the details about the reverse-engineering feature in the JDX User (Manual JDX-5.0-Manual-v1.pdf) shipped in the docs

directory of the Gilhari SDK. Please set the JX_HOME environment variable to the root directory of the Gilhari SDK installation to use this feature.

Gilhari SDK also ships with an example directory (JDX_ReverseEngineeringJSONExample) to demonstrate reverse engineering of an existing relational schema for a JSON object data model. You may follow this example to get a jump start on using Gilhari for your legacy data in a relational database.

More details on Reverse-engineering:

If a database (schema) already exists and you want to use some or all of its tables in a new Gilhari microservice powered application, you would need to define the corresponding container classes (for JSON object model) and the mapping specification for those container classes. If you want to use a substantial number of database tables (e.g., more than 10) with a non-trivial average number of columns (e.g., more than 10) and the tables have a few 1-to-1 and/or 1-to-many relationships, it would be quite tedious and error-prone task to manually define the JSON model classes and their mapping specifications. Imagine there being typos in the table/columns names or imagine missing on some relationship structures implicit in the referential integrity constraints in the database schema, etc. The reverse-engineering tool simplifies the process of generating the appropriate container classes and the mapping specification for those container classes based on an existing schema and the chosen tables specified declaratively in a reverse-engineering template file. However, you may still need to fine-tune the reverse engineered classes and the mapping specification based on your semantic knowledge of the data and the application need.

For example, if your application would rather use the JSON attribute name of "compensation" than the default column name of "salary", you should add an appropriate SQLMAP specification in the generated mapping file. For another example, if the reverse-engineering tool inferred a relationship between two JSON model classes (e.g., Employee and Address) as 1-to-many, but your application semantics says that it should be a 1-to-1 relationship, you should modify the generated Employee container class and the associated mappings in the mapping file for Employee and (a list of) Address. For yet another example, if you do not want all the columns of a table to be exposed in your JSON object model (to achieve object trimming), you may remove the mappings for those columns (the corresponding VIRTUAL_ATTRIB specifications) from the mapping specification. That's it. However, in that case, any new JSON objects of the corresponding model class inserted in the database through that app-specific Gilhari microservice would have the values of the missing (non-mapped) columns set to null or some default value. That may not be a concern though if Gilhari is used to just query the data from those tables.

Please see gilhari_relationships_example directory to see how the structure of the container classes and the corresponding mappings for a 1-to-1 or 1-to-many relationships may be defined. By the way, if you want to remove some clutter from the reverse-engineering

generated mapping file, the following statements, which you might have needed in the reverse-engineering template file, may safely be deleted.

```
JDX_SUPERCLASS_NAME <superClassName>  
JDX_GENERATE_ACCESSOR_METHODS <TRUE | FALSE>  
JDX_GENERATE_JSON_MAPPINGS <TRUE | FALSE>
```

Important Note:

If you are using Gilhari with an existing database and you don't want to lose the existing data when Gilhari is run, please make sure that in the [gilhari service.config](#) (or equivalent) file, you specify:

```
"jdx_force_create_schema": "false",
```

The above comment applies whether you reverse-engineer the mapping specification or define it manually.

Using Streaming Operations:

You may use Gilhari to get the qualifying JSON objects from the database in a streaming mode such that the data is retrieved for few objects at a time instead of in one fell swoop. This could be useful when a large number of objects could result in a GET operation (query) but the client does not want to wait for the data of all the objects to be returned in one call because of, for example, memory and performance reasons. Instead, the client can get the data using the streaming variations of the GET operation as follows:

- Start a streaming operation with an initial number of objects using the **startStream** directive in the URI:

<http://localhost:80/gilhari/v1/Employee/startStream?maxObjects=20>

Notes:

- The maxObjects parameter specifies the initial number of objects to be retrieved from the stream.
 - If the value of maxObjects (x) exceeds the total number of qualifying objects (y) of the streaming operation, only the total number of the qualifying objects (y) will be returned.
- Fetch the subsequent objects iteratively from the previously opened stream using the **fetchMore** directive in the URI:

<http://localhost:80/gilhari/v1/Employee/fetchMore?maxObjects=10>

Notes:

- If the value of maxObjects (x) exceeds the remaining number of objects (y) in the stream, only the remaining number of objects (y) will be returned.
- To retrieve all the remaining objects in the stream at any time, specify a value of -1 for maxObjects. For example, if there is a total of 100 qualifying objects in a query, here is how many objects Gilhari will return in response to various calls:

Directive	maxObjects	Returned objects
startStream	20	20
fetchMore	10	10
fetchMore	25	25
fetchMore	30	30
fetchMore	-1	15

- A stream must have been opened using the **startStream** directive prior to a **fetchMore** operation. Otherwise an error will be returned.

- A stream must not have been closed using the **closeStream** directive (see below) prior to a **fetchMore** operation. Otherwise an error will be returned.
- Close the streaming operation using the **closeStream** directive in the URI:

`http://localhost:80/gilhari/v1/Employee/closeStream`

Notes:

- You must use the **closeStream** directive to close a previously opened stream. This is important for Gilhari to release any resources (transaction, cursor, etc.) held for the streaming operation.
- You may use the **closeStream** directive before exhausting the stream fully. This may be useful when there is no need to consume any remaining objects of a potentially large query.

Qualifying and ordering the streaming objects:

You may specify filter conditions and/or sort order with the **startStream** directive. The filter clause is equivalent to SQL WHERE clause in a SELECT statement. For example,

- The following URI specifies to start a streaming query on Employee objects that are “exempt” and to retrieve the first 20 such objects in the ascending (default) order of their names. The order remains effective for the subsequent **fetchMore** operations.

`http://localhost:80/gilhari/v1/Employee/startStream?filter=exempt=1+ORDER+BY+name&maxObjects=20`

- The following URI specifies to start a streaming query on all the Employee objects and to retrieve the first 20 such objects in the descending order of their names. The order remains effective for the subsequent **fetchMore** operations.

`http://localhost:80/gilhari/v1/Employee/startStream?filter=ORDER+BY+name+DESC&maxObjects=20`

- **Note:** A + operator is used above to incorporate a space character in the URI.

Using streaming for transferring data among multiple data repositories:

You may open a query stream on a data repository using one instance of Gilhari and iteratively transfer the streaming data to another data repository using another instance of Gilhari. The mapping for the JSON object attributes may be different (e.g., different table names, different column names, pruning attributes, etc.) for the two data repositories.

This can simplify the development of a data replication, a data synchronization, or a data pipeline functionality.

Fine-tuning REST API calls with operational directives:

Gilhari provides some operational directives (like "projections", "ignore", "follow", and "filters") for fine-tuning the REST API calls. These directives that are specified in the URL of a GET request or body of a POST request can help in refining the shape of the target JSON objects. This can potentially be useful in implementing GraphQL Queries and Mutations.

Please also see an accompanying OperationDetailsDoc.pdf file in the docs directory of the SDK for more details.

High-level grammar for specifying operation details in the body of a Gilhari REST request:

```
<operationDetails> ::= { "operationDetails": [ comma-separated  
list of <operationDetail> ] }
```

```
<operationDetail> ::= <projectionsDetail> | <ignoreDetail> |  
<followDetail> | <filtersDetail>
```

```
<projectionsDetail> ::= { "opType": "projections",  
"projectionsDetails": [ comma-separated list of  
<projectionDetail> ] }
```

```
<projectionDetail> ::= { "type": "<typeName>", "attrs": [  
<comma-separated list of "<attributeName>"> ] }
```

```
<ignoreDetail> ::= { "opType": "ignore", "references": [ comma-  
separated list of <reference> ] }
```

```
<reference> ::= "<typeName>", "<referencedAttribName>"
```

```
<followDetail> ::= { "opType": "follow", "references": [ comma-  
separated list of <reference> ] }
```

```
<filtersDetail> ::= { "opType": "filters", "predicates" :  
[comma-separated list of <predicate> ] }
```

```
<predicate> ::= { "type": "<typeName>", "predicate": "<predicate  
string>" }
```

Explanation for the grammar and terms used in specifying operation details

The ``operationDetails`` parameter allows you to fine-tune query operations with operational directives similar to GraphQL capabilities. It accepts a JSON array containing one or more operation directives that refine the shape and scope of returned objects.

With the "operationDetails" clause in the query URL of a REST GET request (e.g., query, getObjectById, access, startStream, fetchMore), you may specify one or more directives (using the "opType" tag) to further refine the operation. You may combine multiple directives within one "operationDetails" specification. The "operationDetails" clause is added to the query part of the GET URL similar to how a "filter" clause is added.

With the "operationDetails" tag in the body of a REST POST request, you may specify one or more directives (using the "opType" tag) to further refine the operation. You may combine multiple directives within one "operationDetails" specification.

Some example directories of the Gilhari SDK provide examples of specifying "operationDetails" in curl commands.

<projectionsDetail>

"projections": This opType specifies that the operation should consider only a subset of the attributes of particular classes (types). You may specify "projectionDetails" for multiple classes (types). For example, the following specification means that only attributes "ald" and "aString" of the type (class) "A" should be queried.

```
{
  "operationDetails": [
    {
      "opType": "projections",
      "projectionsDetails": [
        {
          "type": "A",
          "attrs": [
            "ald",
            "aString"
          ]
        }
      ]
    }
  ]
}
```

```

    }
  ]
}
]
}

```

<ignoreDetail>

"ignore": This opType specifies that the operation should ignore (not follow or consider) the referenced branch of the data specified by the class name and the attribute name pair. Useful if the overall operation is of "deep" kind where, by default, the data of the whole object graph is in play but you want to ignore some branch(es). You may specify "references" for multiple branches of multiple types (classes). For example, the following specification means that the object branch referenced by the "aCs" attribute in the type (class) "A" should be ignored in this operation.

```

{
  "operationDetails": [
    {
      "opType": "ignore",
      "references": ["A", "aCs"]
    }
  ]
}

```

<followDetail>

"follow": This opType specifies that the operation should follow or consider the referenced branch of the data specified by the class name and the attribute name pair. Useful if the overall operation is of "shallow" kind where, by default, the data of only the top level object is in play. You may specify "references" for multiple branches of multiple types (classes). For example, the following specification means that the object branch referenced by the "aB" attribute in the type (class) "A" should be followed in this operation.

```

{
  "operationDetails": [
    {
      "opType": "follow",

```

```

        "references": ["A", "aB"]
    }
]
}

```

<filtersDetail>

"filters": This opType specifies that the operation should consider only that subset of the objects for a specified type (class) which satisfy a given predicate (filter criterion). You may specify multiple predicates for multiple types (classes). For example, the following specification means that the a predicate of "aString > 'aString_0'" should be applied for objects of type "A" and a predicate of "bInt > 100" should be applied for objects of type "B".

```

{
  "operationDetails": [
    {
      "opType": "filters",
      "predicates": [
        {
          "type": "A",
          "predicate": "aString > 'aString_0'"
        },
        {
          "type": "B",
          "predicate": "bInt > 100"
        }
      ]
    }
  ]
}

```

Some more examples of specifying operation details in the body of a Gilhari REST request

// Here is an example of three operation details for "projections", "ignore", and "filters" combined together.

```

{ "operationDetails":
  [
    {
      "opType": "projections",
      "projectionsDetails": [
        {
          "type": "B",
          "attrs": [
            "ald",
            "bld",
            "bInt"
          ]
        },
        {
          "type": "A",
          "attrs": [
            "ald",
            "aString"
          ]
        }
      ]
    },
    {
      "opType": "ignore",
      "references": ["A", "aCs"]
    },
    {
      "opType": "filters",
      "predicates": [

```

```
{
  "type": "A",
  "predicate": "aString > 'aString_0'"
},
{
  "type": "B",sqlite

  "predicate": "bInt > 100"
}
]
}
]
}
```

10 June 2026