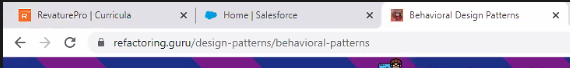
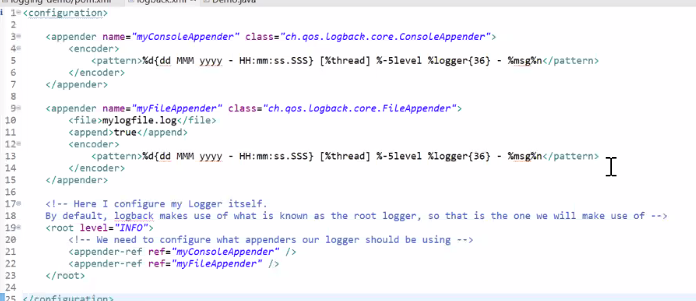
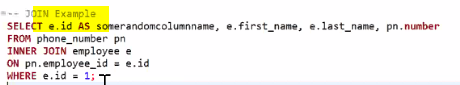
|  |
| --- |
| <!-- Logback library slf4j also part of Project 0 separate logging requirement-->  Issue when trying to use the Javalin version.  Appears that local logback.xml is not used.  <root level=*"trace"*> is overridden and only INFO and above are shown  <!-- Javalin and its dependencies [slf4j, jackson] -->  <!-- Javalin https://javalin.io -->  <dependency>  <groupId>io.javalin</groupId>  <artifactId>javalin</artifactId>  <version>3.13.10</version>  </dependency>  <!-- Javalin http://logback.qos.ch/ - replaces log4j -->  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-simple</artifactId>  <version>1.7.30</version>  </dependency>  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>2.10.5</version>  </dependency> |
| Runtime error if two logback definition are in the pom.xml  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-classic</artifactId>  <version>1.2.3</version>  </dependency> |
| Commenting out the Javalin logback allowed controlled of logging level in Project 0, however using levels below INFO activates a lot of Javalin logging.  <!-- Logback library slf4j also part of Project 0 separate logging requirement -->  <!-- Javalin http://logback.qos.ch/ - replaces log4j -->    <!-- Try commenting out the Javalin logback library  Doing this allows logging level to be controlled within our project  before it appeared that the logging level was controlled by Javalin    <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-simple</artifactId>  <version>1.7.30</version>  </dependency>  --> |
| Are we strictly using Postman for testing HTTP requests (post, get, put, …)?  Would using Javalin in src/test/java Classes work for test automation?  Doing a get <http://localhost:3005/clients> with Javalin in an ApplicationTest class?      Or is this where Mockito comes into play? |

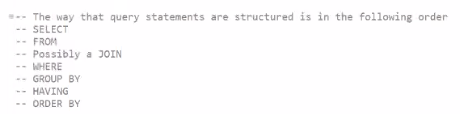
|  |
| --- |
| **# Project 0: RESTful API**  Tentative Due Date: August 11th |
| **## Description**  You are developing a simple API for a bank.  Leveraging Java 8 and following test driven development (TDD), create a RESTful API application that supports two main resources: clients and accounts. |
| ### Technologies  1. **Java**  2. **Javalin**  3. JDBC  4. Logback / **SLF4J**  5. **JUnit**  6. Mockito  7. MariaDB |
| **## Requirements** |
| **### Endpoint Requirements**  The following endpoints and their respective verbs should support the operations as described. In the HTTP response, use appropriate status codes (as specific as possible) and response bodies where applicable.  - `POST /clients`: Creates a new client  - `GET /clients`: Gets all clients  - `GET /clients/{id}`: Get client with an id of X (if the client exists)  - `PUT /clients/{id}`: Update client with an id of X (if the client exists)  - `DELETE /clients/{id}`: Delete client with an id of X (if the client exists)  - `POST /clients/{client\_id}/accounts`: Create a new account for a client with id of X (if client exists)  - `GET /clients/{client\_id}/accounts`: Get all accounts for client with id of X (if client exists)  - `GET /clients/{client\_id}/accounts?amountLessThan=2000&amountGreaterThan=400`: Get all accounts for client id of X with balances between 400 and 2000 (if client exists)  - `GET /clients/{client\_id}/accounts/{account\_id}`: Get account with id of Y belonging to client with id of X (if client and account exist AND if account belongs to client)  - `PUT /clients/{client\_id}/accounts/{account\_id}`: Update account with id of Y belonging to client with id of X (if client and account exist AND if account belongs to client)  - `DELETE /clients/{client\_id}/accounts/{account\_id}`: Delete account with id of Y belonging to client with id of X (if client and account exist AND if account belongs to client) |
| **### General Requirements**  - Java 8  - Users of the application should be able to interact with it through a RESTful API utilizing HTTP  - 3 layered architecture  - Controller (presentation) layer  - Service (business logic) layer  - Data Access layer  - 80-90% test coverage for the \*\*service layer\*\* using **JUnit is required**  - Utilize Mockito in order to mock DAO dependencies  - Logging should be accomplished using Logback  - Have a mental model of what should ideally be logged in an application  - Http Requests  - Methods being executed  - Exceptions  - etc.  - Create a SQL script that will create a table schema and populate some data for your application  - Utilize JDBC in the application for data persistence  - Must utilize the DAO design pattern  - MariaDB |
| **### Stretch Goals**  Remember that the above requirements are the minimum viable product (MVP). You can always look into adding more  Here are some ideas:  - Postman tests to verify API functionality  - Run the application as a standalone JAR  - Run the application on a docker image  - Deploy the application to an EC2 instance on AWS  - Use at least 1 stored procedure w/ MariaDB  - Login Functionality and Roles?  - Research about Session management (HttpSession, Javalin's Access Manager, etc)  - Additional resources and RESTful endpoints (in addition to the clients and accounts resources in the main requirements) |
|  |
|  |
| **RESTful API**  https://en.wikipedia.org/wiki/Representational\_state\_transfer  Representational state transfer (REST) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of an Internet-scale distributed hypermedia system, such as the Web, should behave. The REST architectural style emphasizes the scalability of interactions between components, uniform interfaces, independent deployment of components, and the creation of a layered architecture to facilitate caching components to reduce user-perceived latency, enforce security, and encapsulate legacy systems.[1] REST has been employed throughout the software industry and is a widely accepted set of guidelines for creating stateless, reliable web services.  Any web service that obeys the REST constraints is informally described as RESTful. Such a web service must provide its Web resources in a textual representation and allow them to be read and modified with a stateless protocol and a predefined set of operations. This approach allows the greatest interoperability between clients and servers in a long-lived Internet-scale environment which crosses organizational (trust) boundaries |
| **HashCode**  A hash code is **an integer value that is associated with each object in Java**. Its main purpose is to facilitate hashing in hash tables, which are used by data structures like HashMap.  The purpose of the hashCode() method is to provide a numeric representation of an object's contents so as to provide an alternate mechanism to loosely identify it.  By default the hashCode() returns an integer that represents the **internal memory address of the object**. Where this comes in handy is in the creation and use of an important computer science data structure called a hash table. Hash tables map keys, which are values that result from a hash function (aka, hashCode() method), to a value of interest (i.e., the object the hashCode() method was executed on). This becomes a very useful feature when dealing with moderate-to-large collections of items, because it is **usually a lot faster to compute a hash value compared to linearly searching a collection**, or having to resize and copy items in an array backing a collection when it's limit is reached.  The driving feature behind an efficient hash table is the ability to create a hash that is adequately unique for each object. Buried in that last sentence is the reason why I emphasized the need to override both equals(Object) and hashCode() in the prior article.  If an object has implementation characteristics that require it to be logically distinct from others based on its content then it needs to produce as distinct a hash as reasonably possible. So two objects that are logically equivalent should produce the same hash, but it is sometimes unavoidable to have two logically different objects that may produce the same hash which is known as a collision. When collisions happen the colliding objects are placed in a metaphorical bucket and a secondary algorithm is used to differentiate them within their hash bucket. |
| **Logback**  **http://logback.qos.ch/**  Overview. **Logback** is one of the most widely **used** logging frameworks in the Java Community. It's a replacement for its predecessor, Log4j. **Logback** offers a faster implementation than Log4j, provides more options for configuration, and more flexibility in archiving old log files.  Logback is intended as a successor to the popular log4j project, [picking up where log4j leaves off](http://logback.qos.ch/reasonsToSwitch.html).  Logback's architecture is sufficiently generic so as to apply under different circumstances. At present time, logback is divided into three modules, logback-core, logback-classic and logback-access.  The logback-core module lays the groundwork for the other two modules. The logback-classic module can be assimilated to a significantly improved version of log4j. Moreover, logback-classic natively implements the [SLF4J API](http://www.slf4j.org/) so that you can readily switch back and forth between logback and other logging frameworks such as log4j or java.util.logging (JUL).  The logback-access module integrates with Servlet containers, such as Tomcat and Jetty, to provide HTTP-access log functionality. Note that you could easily build your own module on top of logback-core. |
| **Jackson databind**  Data Binding API is **used to convert JSON to and from POJO (Plain Old Java Object) using property accessor or using annotations**. It is of two type. Simple Data Binding - Converts JSON to and from Java Maps, Lists, Strings, Numbers, Booleans and null objects. |
|  |
| <build>  <plugins>  <plugin>  <groupId>org.appache.maven.plugins</groupId>  <artifactId>maven-assembly-plugin</artifactId>  </plugin>  </plugins>  </build> |
|  |



refactoring.guru/design-patterns

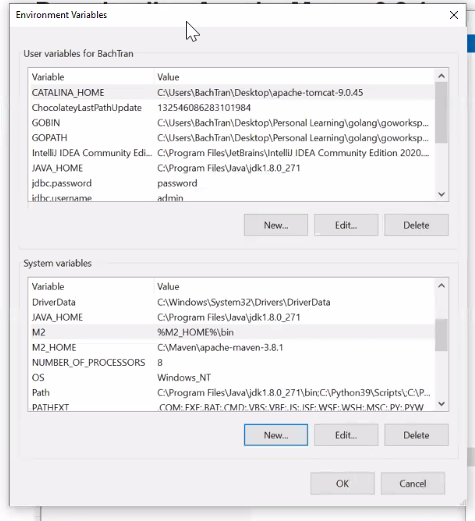








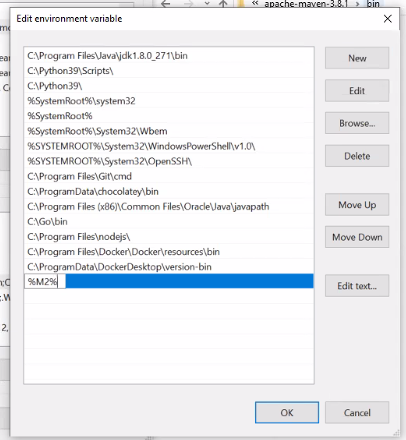
Add system variable



M2 %M2\_HOME%\bin

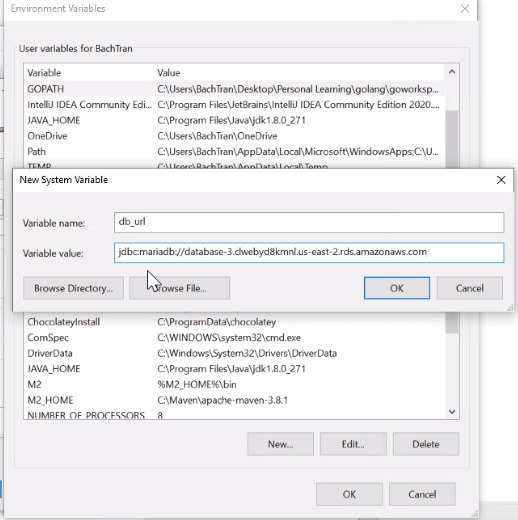
M2\_HOME C:\Maven\apache-maven-3.8.1

Edit path



Add %M2%

mvn --version



localhost:3306

jdbc:mariadb://database-3.clwebyd8kmnl.us-east-2.rds.amazonaws.com:3306/jdbc\_demo

jdbc:mariadb://localhost:3306

my aws connection string

jdbc:mariadb://database-1.cdigirn7b0ro.us-east-2.rds.amazonaws.com:3306/jdbc\_demo

jdbc:mariadb://database-1.cdigirn7b0ro.us-east-2.rds.amazonaws.com

database-1.cdigirn7b0ro.us-east-2.rds.amazonaws.com

Could not connect to address=(host=database-1.cdigirn7b0ro.us-east-2.rds.amazonaws.com)(port=3306)(type=master) : Socket fail to connect to host:database-1.cdigirn7b0ro.us-east-2.rds.amazonaws.com, port:3306. Connection timed out: connect

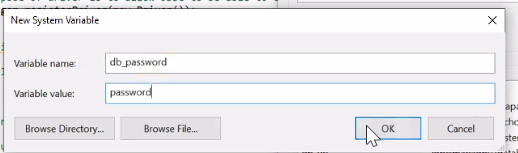
Socket fail to connect to host:database-1.cdigirn7b0ro.us-east-2.rds.amazonaws.com, port:3306. Connection timed out: connect

Socket fail to connect to host:database-1.cdigirn7b0ro.us-east-2.rds.amazonaws.com, port:3306. Connection timed out: connect

Connection timed out: connect

Connection timed out: connect

191.96.67.203



|  |
| --- |
| To create a new MariaDB user, type the following command: **CREATE USER 'user1'@localhost IDENTIFIED BY 'password1'**; In this case, we use the 'localhost' host-name and not the server's IP. This practice is commonplace if you plan to SSH in to your server, or when using the local client to connect to a local MySQL server. |

**CREATE** **USER** 'project0'@localhost IDENTIFIED **BY** 'project0';

172.31.0.0/16

192.168.0.15

