John Conrad  
Dr. Yeung  
CS 5300  
3 November 2023

Programming Project 01 Code Description

Contents

* Overview
* Launching the Application with Docker
* Testing with Swagger
* Project File Structure
* Code Flow

Overview

This is an overview of the Database Normalization API I built using Python. It is built on the Fast Api library and with support for a Docker container.

**Fast Api:** I wrote my project as a Python Fast API. I thought this was the easiest way to get a functional application rolling with an easy to use and understand user interface since the Fast API library generates a Swagger page from the code. If you have never seen a Swagger docs page for an API, don’t panic. I will walk you through it step by step with screenshots below.

**Docker:** To simplify the deployment process for other users (namely graders), I built the API inside of a Docker container which contains all the necessary packages and libraries needed to run the application. If you do not have Docker installed on your machine, it’s simple to set up and I’ll outline the basics of that in another file named “InstallingDocker.md”. It’s meant to run on Linux and as such can be a little clunky on Windows and MacOS, but it is a very powerful tool this app only brushes the surface of.

Launching the Application with Docker

1. **Open a Terminal** - If you have Visual Studio Code, you can do this by opening the database-normalizer project folder in Visual Studio Code, going to Terminal, then click New Terminal.  
   A screenshot of a computer

   Description automatically generated
2. **Run Docker** - If you’re on Windows, ensure the Docker Desktop application is running. This runs WSL and the Docker Daemon in the background, allowing you to build and run containers. I’m unsure how Docker works on Linux and Mac, but I imagine it is very similar.
3. **Build the Docker Image** – within your Terminal, run the command:  
   docker build --no-cache -t database-normalizer .  
   A black screen with white text

   Description automatically generated  
   This may take a moment as it will compile the Python code and install packages outlined in the requirements.txt file to the image via pip.
4. **Run the Docker Image inside a Container** – Once the build has finished, run the following command in your terminal to launch the application on your localhost port 80:

docker run -d -p 80:80 database-normalizer  
A black screen with white text

Description automatically generated

1. That’s it! You should see your container running in Docker desktop with a random name, but the image should say database-normalizer.  
   A screenshot of a computer

   Description automatically generated
2. **Troubleshooting** – if the container fails to deploy, you should get a status of “Exited(1)” or something similar in your list of containers. If you click on that status, it should open up the container’s logs and provide more context regarding the crash.  
   A screenshot of a computer

   Description automatically generated
3. **Teardown** – I have had some rather mixed results regarding Docker’s ability to tear itself down. Stopping containers is as easy as hitting the stop button under actions in the containers view on Docker Desktop. However, you may want to perform some additional cleanup.
   1. Stop the Containers
   2. Manually delete any old containers to free up memory by clicking the checkbox at the top of the list of containers and then hitting the trash bin under actions.
   3. Manually delete old images from the images panel in the same manner as deleting the containers.
   4. Manually ensure all background instances of the application have shut down after you close the Docker Desktop application via Task Manager.

Testing with Swagger

1. **Open a Web Browser and Navigate to the Swagger** Page - After you run the container with Docker, the command you ran should make the application’s swagger page available at: <http://localhost/docs>  
   A screenshot of a computer

   Description automatically generated
2. **Expand the POST /normalize-database** **Endpoint** – click the dropdown arrow at the right end of the green bar to expand the endpoint for the database.  
   A screenshot of a computer

   Description automatically generated
3. **Try it out** – in the top right of the screenshot above, you’ll see there’s a button to “Try it out”. This will open a very simple user interface which will allow you to configure your test circumstances. I have included dropdowns for the Target Normal Form (target\_normal\_form), Detect Current Normal Form (detect\_current\_normal\_form), the CSV file import (sample\_data\_csv), a text file containing each of the keys you would like set for the input relation (keys\_txt), and a text file list of dependencies (dependencies\_txt) you would like to include.  
   A screenshot of a computer

   Description automatically generated  
   Simply hit the big blue “Execute” button once your inputs are selected to run the database normalizer.
4. **Sample Inputs** – if you need any sample input files to reference for formatting, I have included this in a folder aptly named “sampleInput” within the main working directory.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. **Output Queries** – following a successful execution of the endpoint, the queries and current normal form of the input relation (if requested) are provided as part of a JSON payload if you scroll down below the Execute button.

A screenshot of a computer

Description automatically generated

Please excuse the current screenshot. I am still polishing the SQL Query generator, but you can see the requested information in the response body as well. If you’d like to extract this to Notepad++ for better formatting, you can hit the copy to clip board or download buttons in the bottom right of the Response body window.  
A screenshot of a computer

Description automatically generated

1. **Troubleshooting** – if you run into issues inputting testing data, the parsers should warn you via the response body if your data is not properly formatted. For larger issues, I would recommend looking into the Docker Container logs within Docker Desktop.

Project File Structure

I loosely attempted to follow a Domain Driven Design Architecture with this application. This approach helps break pieces of code up into logical chunks and separate concerns in large applications. I honestly just needed a bit of practice in it having just switched primary coding languages for my job. Please do not use this as a clean-cut example for how DDD is supposed to look, there were some architectural liberties taken. However, it can still be useful for loose navigation through the application.

There are a variety of names for the layers, but DDD applications are typically broken up into 5 layers:

1. Presentation/UI – the code responsible for presenting data to the user. In this case, this is simply the Fast Api code held in main.py.
2. Application/Business Logic – the code responsible for performing the heavy lifting of the application – manipulating data in the highly specific way that merited building a standalone application. In this API, this code is contained within the “application” directory. This includes the functions and classes responsible for:
   1. Parsing CSV’s and Dependencies
   2. Determining Normal Form
   3. Building the SQL Queries
   4. Manipulating the Relation Objects in Specific and Reusable Ways
   5. Normalizing Relations
3. Core/Domain – these are the models and entities the application builds and works on. In large enterprise applications, this layer helps prevent leaky abstractions from making their way into the application from external entities which can make the application substantially more difficult to maintain. In this application, the “core” folder simply houses our standard models:
   1. Relation
   2. Attribute (as well as a factory class to simplify construction, though this would traditionally fall under an application layer concern)
   3. Dependency
4. Service – This layer is not present in this application due to lack of need. However, this is typically code responsible for contacting external applications and services. It is a vital piece in preventing leaky abstractions as any external entity is sanitized to a domain specific model.
5. Infrastructure/Persistence – This is also not part of this application, but it is simply the code responsible for persisting data inside of various database contexts.

Outside of the DDD folder structure, there are a few items worth mentioning on their own.

1. tests – This directory contains a series of unit tests I wrote to verify the logic of the normalization related functions.
2. .vscode – This directory simply contains a settigns.json file used to assist Visual Studio Code with discovering and running unit tests.
3. sampleInput – I wrote about this above, but this directory contains sample files to illustrate formatting and perform a end to end test of the system.
4. Dockerfile – This file is what controls logical flow when you build the Docker Image. It sets the working directory, installs external packages, exposes ports, and determines where to direct the binary files.
5. requirements.txt – this simply contains a list that the Dockerfile uses to determine which external libraries to install via pip on image construction.
6. .gitignore – This file determines which items to ignore when performing source control operations with Git.
7. LICENSE – an auto generated copy of the GNU public license supplied by GitHub when I constructed the app.

Code Flow

1. **POST /normalize-database -** Starting at main.py, you’ll find the main flow of the system laid out in the normalize\_database function defined there. This function, it’s input models, and the tag above it are what Fast Api looks at to generate the Swagger page.
2. **Parse Text -** The first step in the process is to parse the input data from the text into lists of strings. The code is very straightforward in application/parse\_text.py.
3. **Parse CSV -** Once we have the list of input keys, we can pass that along with the uploaded CSV file into the parse\_csv method to build the initial relation model the application will work with. The code can be found in application/parse\_csv.py. At a high-level, it performs the following logic:
   1. Perform basic input validation, raising exceptions if parsing errors are encountered.
   2. Instantiate an empty Relation object (found in core/relation.py).
   3. Read the first line of the CSV into a list of attribute\_names.
   4. Line by line, it inserts each of the data records into a list of tuples (functionally, it’s simply a List[List[str]]).
   5. The attribute\_names are parsed into a list of Attribute objects (found in core/attribute.py) via an AttributeFactory (core/attribute\_factory.py) which examines the first tuple to infer basic data types.
   6. The key\_list is parsed and its corresponding Attribute object is loaded into the Relation’s list of primary\_keys. If no corresponding Attribute is found, an exception is returned to the user to inform them to check their CSV and text files to ensure that the attribute names are consistent.
   7. The assembled Relation object is returned to the main function.
4. **Parse Dependencies -** Once the bulk of the relation is built, the parse\_dependencies function (found in application/parse\_dependencies.py) takes the input list of dependencies and constructs them into instances of the Dependency object (core/Dependency.py).
   1. Once again, there is some input validation o ensure that the dependency strings have been provided in the proper format. If an issue is found, it’s reported to the user within Swagger. This includes formatting at first, but at the end of the method, the attribute names are checked against the Relation to ensure they are valid.
   2. The dependency object works in terms of a parent/child relationship. In the standard X determines Y dependency, X would be the parent, and Y would be the child. One dependency can contain multiple children. However, there is only one parent per dependency. I found this structure helpful for thinking about the dependency chains, and it assisted in the recursive functions for examining lineage later.
   3. Once the list is parsed, it’s returned from the function to be set at the main level onto the relation for visibility.
5. **Determine Input Relation’s Current Normal Form -** Now that the full Relation has been parsed, normalize\_database checks if the user selected the option to retrieve the current normal form. If so, the determine\_normal\_form function (found in application/determine\_normal\_form.py is run on the relation.
   1. This function is rather straightforward as it simply runs a series of Boolean functions to determine the current normal form by working up the normal forms in ascending order and returning the previous normal form whenever a normal form’s validation is failed.
   2. Each of the Boolean functions works in a similar way, iterating through attributes (and in some cases tuples) of the relation to check for a failure condition. All of the functions are within the same file, and several of the functions have helper methods.
      1. **isRelationIn1NF** checks if all values are atomic, if there are duplicate attribute names, if there are duplicate tuples, if all the data in each column is a consistent data type, and if there is at least one primary key in the relation.
      2. **isRelationIn2NF** iterates through each key to ensure each of its children (found by iterating through the dependencies) have the full set of keys as parents.
      3. **isRelationIn3NF** looks for transitive functional dependencies by first getting a list of non-key parents for each of its attributes. Then the function iterates through that subset of non-key parents, and looks recursively up the lineage to determine if there is a key ancestor (specifically through the isNonKeyParentDeterminedByKey helper function).
      4. **isRelationInBCNF** looks for non-key parents in functional dependencies by cross-checking each attributes parents against the list of keys.
      5. **isRelationIn4NF** iterates over each parent attribute in the database, and iteratively looking at each child’s data values, constructs a dictionary of parent-value, child-value pairs to determine if there is a multi-value-dependency at play.
      6. **isRelationIn5NF** was the trickiest to sort out since we’re working in the limited context of a single relation. This function works in an opposite manner to the other functions. Instead of ruling out relations, it checks for known valid 5NF states by evaluating if there are only two items in the relation, if every attribute is part of the keys, and if the relation is a single dependency with only attributes in that one dependency. If no circumstance is found, it returns False.
6. **Normalize to the Specified Normal Form –** application/normalize.py is the largest file in the project at a little over 450 lines, but the root method operates in a similar method as the normal form detection logic. The logic flow itself is very similar to the determine\_normal\_form with the large caveat that almost every function in this file is built to be recursive.
   1. **normalize** – this is the entry-point method which takes in a relation, target normal form, and current normal form (if already gathered in main). The relation is immediately loaded into a list of relations which will be added to at each stage in normalization (except for 1NF). At each stage, each relation in the list of relations is checked against the current form using the same methods from determine\_normal\_form. If it is determined not to be in that form, the relation is sent through the specific normalization method for that level. The list of relations returned from the normalization method are appended to the end of the list of relations which will be returned when the set of relations has been normalized to the target level.
   2. **normalize\_to\_1NF** – this method operates through the same logic as its isRelationIn1NF counterpart except when it finds a breaking case, it attempts to fix it. If there’s an non-atomic value, I assumed a parsing error on my part because there are too many unrestricted inputs to consider with a problem like this. If a duplicate attribute name is found, it is prefixed with a “Duplicate\_” string. The list of tuples is deduped. A primary key is selected if the primary key is empty.
   3. **normalize\_to\_#NF** – each of the methods above 1NF follows the same flow except with different conditions to split on. The high level flow is this:
      1. Ensure relations are normalized to the level below it by calling the normalize method with the target normal form as the level below.
      2. The return list of relations is initialized to empty.
      3. The method iteratively moves through the lower\_normalized\_relations list it retrieved in step [i] to normalize and split the relations one at a time.
         1. The equivalent determine\_normal\_form method is called on the relation to short circuit the logic if normalization at that level is not necessary. If it is not, the relation is appended to the return list and the for loop continues.
         2. Following that, the condition that caused the isRelationIn#NF Boolean function to fail is identified.
         3. The relation is split on the dependency which triggered the condition by means of helper methods to getAllDescendants of the problematic dependency, split\_the\_attributes\_by\_parent\_and\_descendants to determine which attributes will be in which of the split tables (using the parent as the link between the two tables), and finally split\_relation to break up the relation into two sub-relations.
         4. Each of the sub-relations is then recursively normalized to the current normal form via the normalize function.
         5. The normalized relations are appended to the return list.
7. **Generate SQL** – Once the input relation has been normalized to the appropriate level, get\_table\_creation\_queries (found in sql\_builder.py) is called to construct the SQL query strings for each table using the metadata stored in each relation’s key and attributes lists.