1. **POST** http://localhost:8080/transactionservice **(create API):**
    a. **Description :** Stores request body in table ‘transaction’. If parentId is given then adds (id , parentId) in ‘transactionancestor’ table
    b. **Analysis :**
        i. **Best case:**
        ii. **Worst case:** O(log(number of rows)) , B-Tree insertion
        iii. **Average case:** O(log(number of rows)) , B-Tree insertion
2. **GET** http://localhost:8080/transactionservice/type/car **(getByType API):**
    a. **Description :** Read from table ‘transaction’ and query using the type field
    b. **Analysis :**
        i. **Best case:** O(1) , insert in table/tables , search by non index parameter
        ii. **Worst case:** O(number of rows) , search by non index parameter
        iii. **Average case:** O(number of rows) , search by non index parameter
3. **GET** http://localhost:8080/transactionservice/sum/1 **(SUM API):**
    a. **Description :** calculates a sum of all transactions that are transitively linked by their parentId.
        i. Uses sequelize-hierarchy to get the n-aray-tree
        ii. Function ‘calRecSum’ (calculate recursive sum) to go through all nodes of n-aray-tree recursively and add the sum. Time complexity is equal to number of nodes in n-aray-tree
        iii. Time Complexity = T(sequelize-hierarchy for n-aray-tree) + T(calRecSum) . Time complexity of calRecSum is number of nodes in tree say nNodes .

        To calculate time taken by sequelize-hierarchy, we need to first understand how it works. Principle is similar to recursive common table expressions in sql. Which finds a row by anchor query, in our case it’s id. Then uses this anchor row to recursively find children of the next hierarchy, unitil no child rows are there.

        Assume max width of tree = W
        Height of the tree is = H
        Number of nodes = nNodes
        Number of rows = nRows

        So T(H) = W*T(H-1) + Constant

        T(H) = POW(W, H) + C

        Now every node runs a find query to find it’s children nodes using parentId, which is indexed and cost of each query is log(nRows)

        So T(find) = nNodes* log(nRows)

        Hence total cost will be = T(H) * T(find)

        Total cost = POW(W, H) * nNodes * log(nRows)
    b. **Analysis :**
        i. **Worst case:** O(POW(W, H) * nNodes * log(nRows))
4. **GET** http://localhost:8080/transactionservice/5 **(getById API):**
    a. **Description :** Read from table ‘transaction’ by id field
    b. **Analysis :**
        i. **Best case:**
        ii. **Worst case:** O(log(number of rows)) , B-Tree search using index id

iii. **Average case:** O(log(number of rows)) , B-Tree search using index id

5. **PUT http://localhost:8080/transactionservice/5 (updateById API):**
    a. **Description :** Update table ‘transaction’ by id field
    b. **Analysis :**
        i. **Best case:**
        ii. **Worst case:** O(log(number of rows)) , B-Tree search using index id
        iii. **Average case:** O(log(number of rows)) , B-Tree search using index id