Contents

[Task 1: Design Document 1](#_Toc144212953)

[Overview: 1](#_Toc144212954)

[Use Case Diagram: 2](#_Toc144212955)

[UML Class Diagram 3](#_Toc144212956)

[Data Store Specification 3](#_Toc144212957)

[Design Patterns 3](#_Toc144212958)

[Application Architecture 4](#_Toc144212959)

[Appendices 4](#_Toc144212960)

[Appendix A – UML Class Diagram: 4](#_Toc144212961)

# Task 1: Design Document

## Overview:

This is a Command-Line Interface application that allows a software project team to track and manage tickets. Users can display a list of all active tickets filtered by priority, view completed or deleted tickets and search the database for tickets that match their query, they can add create new tickets, update & delete existing tickets & log time to tickets.

## Use Case Diagram:

A diagram of a person's work flow

Description automatically generated

Figure 1 Use Case Diagram

## UML Class Diagram

A diagram of text and words

Description automatically generated with medium confidence

Figure 2 UML Class Diagram - See Appendix A for screenshots of smaller pieces.

## Data Store Specification

## Design Patterns

This application makes use of a few Object-Oriented Design Patterns.

Singleton Pattern:

Ensures that throughout the lifecycle of the application, ‘Singleton’ classes can only be instantiated once. This is useful for ensuring you re-use the same instance of your database connection for multiple users of the application, whilst not showing it’s strengths in a CLI application, this is commonly used in Backend Website-Development to implement the ACID principles.

Generic Interface & Object Relational Mapper pattern:

The application is built in an easily extendable way. The Repositories & Services are not dependent on the specific datatype (Database Table) that they are working with, new tables can easily be integrated because our infrastructure is ***generic***.

Dependency Injection & Compositional patterns:

All logic is broken up into classes with single responsibilities, such as the Repository that handles all communication with the database and maps data to Python Objects or an InputValidator that prompts the user for inputs and ensures no invalid data can be passed into the application. These small building blocks are then *composed* within a class that manages the control flow of the application, our Controller.

## Application Architecture

This application uses a modular, N-tier architecture based on the dependency injection pattern. The application is made up of individual modules, each having a single responsibility and designed to be generic for easy and safe extension.

At the heart is the TicketController, which interfaces with the user and invokes the appropriate services and functions based on user needs. It includes an Output & Prompt Manager, responsible for print, read, and input validation operations, as well as a Formatter module for human-readable data presentation.

The TicketController utilizes a TicketService for data manipulation and database operations, including filtering, sorting, auto-generation of timestamps, and ticket building. To ensure the integrity of data, there's a separate TicketValidationService that handles the generation and validation of ticket fields based on user input.

The TicketService encapsulates a TicketRepository that performs database actions, assured of data integrity due to the upper-layer validators and services.

Both the TicketService and TicketRepository inherit from BaseService and BaseRepository respectively, which are generic base classes accommodating any Data Model. This provides common functionalities like GetById, DeleteById, etc., to all future Services and Repositories. They are designed to work with generics, simplifying data management by accepting Table Name at runtime with no further modification needed.

## Navigation

### Main Context Menu

A screenshot of a computer program

Description automatically generated

Prints a list of all possible options and awaits input.

### View Active Records

A screenshot of a computer program

Description automatically generated

Displays a list of active (not completed or deleted) tickets, ordered by priority descending (lower is greater priority), then remaining time ascending.

### View Record by ID

A screenshot of a computer program

Description automatically generated

Lists a detailed report of the ticket, allows a user to log time.

### Add a new Record

A screen shot of a computer

Description automatically generated

Runs through multiple prompts to generate a new ticket from user input.

### Update by ID

A screenshot of a computer program

Description automatically generated

Allows a user to select a property of an existing ticket to change.

### Delete by ID

A screen shot of a computer

Description automatically generated

Mark a ticket for soft deletion.

### Search for Records

A screenshot of a computer program

Description automatically generated

User enters a string, tickets with any matching fields are displayed.

### Show completed records.

A screen shot of a computer

Description automatically generated

Prints a list of completed records.

### Show deleted records.

A screen shot of a computer

Description automatically generated

Prints a list of deleted records.

### Show all records Raw & Unfiltered

A screenshot of a computer program

Description automatically generated

Prints the entire ticket table from the database.

### Quit

A black screen with white text

Description automatically generated

Allows a user to exit the application.

# Task 2: Develop Application.

Please see Appendix B for application source code and Appendix C for automated testing source code.

# Appendices

## Appendix A – UML Class Diagram Segments:

A screenshot of a computer program

Description automatically generated

Bottom Right Section from UML Class Diagram.

A diagram of a server

Description automatically generated

Center-Right section from UML Class Diagram.

A screenshot of a computer

Description automatically generated

Center-Left Section from UML Class Diagram.

A screenshot of a computer

Description automatically generated

Bottom-Left Section from UML Class Diagram.

A screenshot of a computer

Description automatically generated

Left Section from UML Class Diagram.

A diagram of a program

Description automatically generated

Top-Left & Top-Middle section from UML Class Diagram.

## Appendix B – Application Source Code:

### 1: Main.py

from config.composition\_manager import TicketConfiguration

def main():

app = TicketConfiguration().compose\_ticket\_controller()

app.start()

if \_\_name\_\_ == "\_\_main\_\_":

main()

### 2: config/composition\_manager.py

from environment.env import DB\_TICKET\_STRING

from controllers.ticket\_controller import TicketController

from repositories.database\_connection import DatabaseConnection

from repositories.ticket\_repository import TicketRepository

from services.terminal.formatter import Formatter

from services.terminal.output\_manager import OutputManager, PromptManager

from services.tickets.ticket\_input\_validation\_service import TicketInputValidationService

from services.tickets.ticket\_service import TicketService

# This metaclass ensures a single instance of a class is created and shared.

class Singleton(type):

\_instances = {}

def \_\_call\_\_(cls, \*args, \*\*kwargs):

if cls not in cls.\_instances:

cls.\_instances[cls] = super(Singleton, cls).\_\_call\_\_(\*args, \*\*kwargs)

return cls.\_instances[cls]

# This class uses Singleton metaclass to create a single instance that will be shared for any inheritors of the base Configuration.

# It creates instances of DatabaseConnection and OutputManager

class BaseConfiguration(metaclass=Singleton):

def \_\_init\_\_(self):

# Create a DatabaseConnection instance if not exists

if not hasattr(self, 'app\_db\_conn'):

self.app\_db\_conn = DatabaseConnection()

# Create an OutputManager instance with Formatter and PromptManager if not exists

if not hasattr(self, 'output\_manager'):

self.output\_manager = OutputManager(

Formatter(),

PromptManager(),

)

# TicketConfiguration inherits from Configuration class and additionally composes the TicketController.

class TicketConfiguration(BaseConfiguration):

def \_\_init\_\_(self):

super().\_\_init\_\_()

# This method composes the TicketController with necessary dependencies

def compose\_ticket\_controller(self) -> TicketController:

return TicketController(

TicketService(

TicketRepository(

self.app\_db\_conn,

DB\_TICKET\_STRING,

)

),

TicketInputValidationService(),

self.output\_manager,

)

### 3: controllers/log\_controller.py

from datetime import datetime

from environment.env import LOG\_FILE\_NAME

class LogController:

"""

A controller class for handling operations related to logs.

It provides methods to create a new log file, count lines in the log file,

add a new log and read all logs.

Attributes:

LOG\_FILE\_NAME: Default log file name. Needs to be defined elsewhere in the code.

"""

def \_\_create\_log\_file(file\_name: str = LOG\_FILE\_NAME):

"""

Creates a new log file.

Args:

file\_name (str): The name of the file to create. Defaults to LOG\_FILE\_NAME.

Returns:

None

"""

with open(file\_name, 'x') as file:

file.write(f"1: Log File created at: {datetime.now()}")

def \_\_get\_line\_count(file\_name: str = LOG\_FILE\_NAME):

"""

Gets the line count of the log file. If the file doesn't exist, creates a new one.

Args:

file\_name (str): The name of the file to get line count. Defaults to LOG\_FILE\_NAME.

Returns:

int: The number of lines in the log file.

"""

try:

with open(file\_name, 'r') as file:

lines = file.readlines()

return len(lines) + 1

except FileNotFoundError:

LogController.\_\_create\_log\_file(file\_name)

return LogController.\_\_get\_line\_count(file\_name)

@staticmethod

def new\_log(msg: str, file\_name: str = LOG\_FILE\_NAME):

"""

Writes a new log entry to the log file.

Args:

msg (str): The message to be logged.

file\_name (str): The log file where the message will be logged. Defaults to LOG\_FILE\_NAME.

Returns:

None

"""

line\_number = LogController.\_\_get\_line\_count(file\_name)

with open(file\_name, 'a') as file:

file.write(f"\n{line\_number}: {msg} - at {datetime.now()}")

@staticmethod

def read\_logs():

"""

Reads all logs and prints the count of lines that contain the word "User".

Returns:

None

"""

with open(LOG\_FILE\_NAME, 'r') as file:

lines = file.readlines()

print(len([l for l in lines if "User" in l]))

### 4: controllers/ticket\_controller.py

from dataclasses import fields

from controllers.log\_controller import LogController

from helpers.print\_colours import print\_green, print\_red

from services.tickets.ticket\_input\_validation\_service import TicketInputValidationService

from services.terminal.output\_manager import OutputManager

from services.tickets.ticket\_service import TicketService

class TicketController():

"""

Controller class for handling ticket-related operations.

Attributes:

service (TicketService): Service for handling ticket-related business logic.

input\_validator (TicketInputValidationService): Service for validating user input related to tickets.

output\_manager (OutputManager): Service for managing output to the terminal.

commands (dict): Mapping of user commands to corresponding methods.

"""

def \_\_init\_\_(self, ticket\_service: TicketService, ticket\_input\_validator: TicketInputValidationService, output\_manager: OutputManager):

"""

Initialize a new instance of the TicketController class.

Args:

ticket\_service (TicketService): The ticket service.

ticket\_input\_validator (TicketInputValidationService): The ticket input validator.

output\_manager (OutputManager): The output manager.

"""

self.service = ticket\_service

self.input\_validator = ticket\_input\_validator

self.output\_manager = output\_manager

self.commands = {

1: self.preview\_all,

2: self.view\_by\_id,

3: self.new\_ticket,

4: self.update\_ticket,

5: self.delete\_ticket,

6: self.search\_by\_name,

7: self.show\_completed,

8: self.show\_deleted,

9: self.show\_all\_raw,

0: self.quit

}

def start(self):

"""

Start the ticket controller, logging the start and end of the application session.

Handles command execution and continues until the quit command is given.

"""

LogController.new\_log("Application Started")

run = True

while run:

user\_command = self.output\_manager.startup\_message()

LogController.new\_log(f"User attempted: {str(self.commands[user\_command].\_\_name\_\_)}")

run = self.commands[user\_command]()

if run:

self.output\_manager.prompt\_manager.continue\_message()

if not run:

LogController.new\_log("Successfully Quit Application")

def preview\_all(self):

"""

Preview all non-deleted, non-completed tickets in order of priority.

"""

non\_deleted\_or\_completed = self.service.filter\_complete(False, self.service.filter\_deleted(False, self.service.get\_all()))

ordered\_tickets = self.service.order\_by\_priority(non\_deleted\_or\_completed)

print\_green(f"There are {len(ordered\_tickets)} outstanding tickets, and {len(non\_deleted\_or\_completed)} in total!\n")

preview = self.service.create\_preview(ordered\_tickets)

self.output\_manager.single\_line(preview)

LogController.new\_log(f"Preview All Succeeded. ")

return True

def view\_by\_id(self):

"""

View a ticket by its ID, and optionally log time to it if it's not completed.

"""

id = self.output\_manager.get\_id\_input(self.service.get\_count())

ticket = self.service.get\_by\_id(id)

if (self.service.check\_if\_deleted(ticket)):

self.output\_manager.no\_records\_found(f"Record deleted. {ticket.updated\_date}")

LogController.new\_log(f"User searched for deleted record: {ticket.id}: {ticket.title}")

return True

self.output\_manager.multi\_line(ticket)

if (self.service.check\_if\_completed(ticket)):

LogController.new\_log(f"Record: {ticket.id} - {ticket.title} is complete. Not prompting user to log time")

return True

if self.output\_manager.prompt\_manager.get\_user\_confirmation("log time to ticket: ", id):

LogController.new\_log(f"User logging time to {ticket.id}: {ticket.title}. ")

time\_to\_log = self.output\_manager.display\_time\_logging\_information(ticket.remaining\_time)

LogController.new\_log(f"Attempting to log {time\_to\_log} to {ticket.id}: {ticket.title}")

updated\_ticket = self.service.log\_time(ticket, time\_to\_log)

LogController.new\_log(f"Successfully logged time. Total time logged: {updated\_ticket.logged\_time}. Remaining time: {updated\_ticket.remaining\_time}")

self.output\_manager.show\_time\_logged(updated\_ticket.logged\_time, updated\_ticket.remaining\_time)

return True

def new\_ticket(self):

"""

Create a new ticket based on user input.

"""

input\_props = self.input\_validator.generate\_validated\_user\_ticket\_inputs()

LogController.new\_log(f"Attempting to create new ticket from inputs: {' '.join(f'{input.name}: {getattr(input\_props, input.name)}' for input in fields(input\_props))}")

ticket = self.service.create\_ticket(input\_props)

LogController.new\_log(f"Successfully created new ticket - {ticket.id}: {ticket.title}")

self.output\_manager.single\_line(ticket)

return True

def update\_ticket(self):

"""

Update a selected ticket based on user input. If the ticket is deleted, no update is performed.

"""

selected\_ticket = self.service.get\_by\_id(

self.output\_manager.get\_id\_input(self.service.get\_count()))

if (self.service.check\_if\_deleted(selected\_ticket)):

LogController.new\_log( f"Updating record {selected\_ticket.id}: {selected\_ticket.title} failed. Deleted.")

self.output\_manager.no\_records\_found(f"Record deleted. {selected\_ticket.updated\_date}")

else:

field\_to\_update = self.output\_manager.display\_update\_information(self.service.populate\_update\_fields(selected\_ticket))

updated\_value = self.input\_validator.call\_chosen\_update\_function(field\_to\_update)

self.service.update\_record(selected\_ticket, field\_to\_update, updated\_value)

print\_green(f"Successfully updated record {selected\_ticket.id} : {selected\_ticket.title}.")

LogController.new\_log(f"Successfully updated record {selected\_ticket.id} : {selected\_ticket.title}.")

return True

def delete\_ticket(self):

"""

Delete a ticket based on user input. If the ticket is already deleted, no action is performed.

"""

id = self.output\_manager.get\_id\_input(self.service.get\_count())

record = self.service.get\_by\_id(id)

if (record.deleted):

print\_red("Record already deleted. ")

LogController.new\_log(f"Deleting record {record.id}: {record.title} failed. Already deleted.")

return

self.output\_manager.single\_line(record)

if (self.output\_manager.prompt\_manager.get\_user\_confirmation("delete record ", id)):

LogController.new\_log(f"Deleted record {record.id}: {record.title} successfully.")

self.output\_manager.multi\_line(self.service.delete\_record(id))

return True

def search\_by\_name(self):

"""

Search for tickets by a string query input by the user.

"""

tickets = self.service.get\_all()

query = self.output\_manager.prompt\_manager.get\_raw\_string\_from\_user()

LogController.new\_log(f"User searched for: {query}")

filtered = self.service.search\_string\_values(tickets, query)

LogController.new\_log(f"Found: {len(filtered)} records.")

self.output\_manager.multi\_line(filtered)

return True

def show\_completed(self):

"""

Show all non-deleted, completed tickets.

"""

non\_deleted = self.service.filter\_deleted(False, self.service.get\_all())

completed = self.service.filter\_complete(True, non\_deleted)

LogController.new\_log(f"Found: {len(completed)} completed records.")

self.output\_manager.single\_line(completed)

return True

def show\_deleted(self):

"""

Show all deleted tickets.

"""

deleted = self.service.filter\_deleted(True, self.service.get\_all())

LogController.new\_log(f"Found: {len(deleted)} deleted records.")

self.output\_manager.single\_line(deleted)

return True

def show\_all\_raw(self):

"""

Show all tickets in raw form.

"""

self.output\_manager.multi\_line(self.service.get\_all())

return True

def quit(self):

"""

Quit the application. Before quitting, the user is asked for confirmation.

"""

self.service.ticket\_repository.db.close()

return not self.output\_manager.prompt\_manager.get\_user\_confirmation("quit the application")

### 5: enums/ticket\_type.py

from enum import Enum

"""

This module contains the TicketType enumeration for representing the valid ticket types.

"""

# I opted for this as a method of statically stored Constants

# due to the limitations of a single table Database

class TicketType(Enum):

"""

An enumeration to represent our three valid Ticket Types

1: DEVELOPMENT

2: TESTING

3: DEPLOYMENT

"""

PLANNING = 'PLANNING'

DESIGN = 'DESIGN'

DEVELOPMENT = 'DEVELOPMENT'

TESTING = 'TESTING'

DEPLOYMENT = 'DEPLOYMENT'

DOCUMENTATION = 'DOCUMENTATION'

SUPPORT = 'SUPPORT'

### 6: environment/colours.py

#Referencing as Static. Defined this as a Class to use Dot notation from JS in code e.g: Terminal.OKGREEN

class Terminal:

HEADER = '\033[95m'

OKBLUE = '\033[94m'

OKCYAN = '\033[96m'

OKGREEN = '\033[92m'

WARNING = '\033[93m'

FAIL = '\033[91m'

ENDC = '\033[0m'

BOLD = '\033[1m'

UNDERLINE = '\033[4m'

### 7: environment/env.py

"""

Typically we would want to use a .env (dotenv) file to store our environment variables, with our secrets

hidden away from the rest of the application. However since we're making a simple console application,

we won't have any cloud, server, OS or even containerisation available to us, so I'm going to use

magic strings in this module to replicate a .env or secrets management platform.

"""

DB\_CONNECTION = "kanban\_db.sqlite"

DB\_TICKET\_STRING = "tickets"

LOG\_FILE\_NAME = "app\_logs.txt"

DB\_ADM\_USER = "very\_cool\_user"

DB\_ADM\_PASS = "very\_encrypted\_password123!"

### 8: helpers/extension\_helpers.py

def convert\_float\_to\_time(time\_float):

hours = int(time\_float)

minutes = (time\_float - hours) \* 60

return hours, round(minutes)

def validate\_as\_int(input: str):

try:

int(input)

return True

except ValueError:

return False

def validate\_as\_float(input: str):

try:

float(input)

return True

except ValueError:

return False

def find\_option\_by\_number(options: list[str], user\_input: int):

#Search our pre-defined List of outputs with with a user input. Format example: "3: User Option..."

return next((opt for opt in options if (int(opt[0]) == user\_input)), "Unknown Option")

def check\_option\_in\_range(options: int | float, opt: int | float):

return True if opt <= options else False

### 9: helpers/print\_colours.py

from environment.colours import Terminal

#Wrapper for our pre-defined console output colours.

#Could be generic enough to take in the colour as an arg, but cleaner to work with in important areas by having it all handled by a function.

def print\_green(message: str):

print(Terminal.OKGREEN + message + Terminal.ENDC)

def print\_red(message: str):

print(Terminal.FAIL + message + Terminal.ENDC)

def print\_blue(message: str):

print(Terminal.OKBLUE + message + Terminal.ENDC)

def print\_yellow(message: str):

print(Terminal.WARNING + message + Terminal.ENDC)

def print\_beige(message: str):

print(Terminal.HEADER + message + Terminal.ENDC)

def prompt\_underline(message: str):

return input(Terminal.UNDERLINE + message + Terminal.ENDC)

### 10: misc/example\_new\_setup.py

from dataclasses import dataclass

from datetime import datetime

from models.base\_model import BaseModel

from repositories.base\_repository import BaseRepository

from repositories.database\_connection import DatabaseConnection

from services.base\_service import BaseService

#NOTE: Not intended to actually be used. Bootstrapped setup to demonstrate creating a new application.

def main():

#Define a new dataclass with any desired properties.

@dataclass

class ExampleNewModel(BaseModel):

example\_new\_int: int

example\_new\_string: str

example\_new\_date: datetime

example\_new\_bool: bool

#Define a new Repository, passing the desired Model.

class ExampleNewRepository(BaseRepository[ExampleNewModel]):

def \_\_init\_\_(self, db\_conn: DatabaseConnection, table\_name: str):

super().\_\_init\_\_(db\_conn, table\_name, ExampleNewModel)

#Define a new Service, passing the desired Model.

class ExampleNewService(BaseService[ExampleNewModel]):

def \_\_init\_\_(self, example\_new\_repository: ExampleNewRepository):

super().\_\_init\_\_(example\_new\_repository)

self.example\_new\_repository = example\_new\_repository

#Instantiate, triggering creation of new table.

new\_service = ExampleNewService(

ExampleNewRepository(

DatabaseConnection(), "Example\_New\_Table"

)

)

new\_service.get\_all() #Generic Implementation

### 11: models/base\_model.py

from dataclasses import dataclass

from datetime import datetime

from typing import Optional

from controllers.log\_controller import LogController

"""

This module contains our Base database Model.

Every dataclass we instantiate will inherit from this.

"""

@dataclass

class BaseModel:

"""

A dataclass representing common properties every Model in the application will inherit.

id: an integer Unique Identifier that will be auto-generated on record creation.

created\_date: a datetime that will be auto-generated on record creation.

updated\_date: a datetime that will be auto-generated on record creation & record updates.

completed: a bool to mark completion of the ticket for querying.

deleted: a bool to flag whether a record has been deleted. (we're using soft-deletion throughout the application)

"""

id: Optional[int]

created\_date: datetime

updated\_date: datetime

completed: bool

deleted: bool

def \_\_post\_init\_\_(self):

"""

Convert our stored dates to actual datetime implementations.

SQLite is very limiting in how we store our data here, this ensures consistency of our data types between reads/writes.

"""

if isinstance(self.created\_date, str):

self.created\_date = datetime.strptime(self.created\_date, "%Y-%m-%d %H:%M:%S.%f")

if isinstance(self.updated\_date, str):

self.updated\_date = datetime.strptime(self.updated\_date, "%Y-%m-%d %H:%M:%S.%f")

if isinstance(self.completed, int):

self.completed = bool(self.completed)

if isinstance(self.deleted, int):

self.deleted = bool(self.deleted)

### 12: models/preview\_model.py

from dataclasses import dataclass

from typing import Optional

@dataclass

class TicketPreviewModel():

"""

A data model for representing a preview of a ticket in a ticketing system.

Attributes

----------

id : int, optional

The unique identifier of the ticket, by default None

priority : int

The priority level of the ticket, where a lower value indicates a more urgent ticket

title : str

The title or brief description of the ticket

remaining\_time : float

The estimated remaining time to resolve the ticket.

"""

id: Optional[int]

priority: int

title: str

remaining\_time: float

### 13: models/ticket\_model.py

from dataclasses import dataclass

from controllers.log\_controller import LogController

from models.base\_model import BaseModel

from enums.ticket\_type import TicketType

"""

This module contains our main dataclass for Object Relational Mapping

"""

@dataclass

class ValidatedUserTicketInputs():

priority: int

ticket\_type: TicketType

title: str

description: str

initial\_estimate: float

"""

A dataclass representing the user input driven content of the Ticket model.

It contains the following properties:

- ticket\_type: a TicketType enumeration value

- name: a string representing the ticket name

- description: a string describing the ticket

- initial\_estimate: a float representing the initial time estimate for the ticket

- remaining\_time: a float representing the remaining time to complete the ticket

- logged\_time: a float representing the time already spent on the ticket

- priority: an integer between 1 and 5

"""

@dataclass

class TicketModel(ValidatedUserTicketInputs, BaseModel):

# Combines BaseModel properties that all Models inherit with User Input fields from ValidatedUserTicketInputs and stores auto-calculated properties.

remaining\_time: float

logged\_time: float

def \_\_post\_init\_\_(self):

# Currently disabled, but was used during Development for tracking Object creation order & garbage collection

# LogController.new\_log(f"Created Object in Memory: {hex(id(self))} - TicketID - {self.id} - TicketName - {self.title}", "memory.txt")

return super().\_\_post\_init\_\_()

# def \_\_del\_\_(self):

# LogController.new\_log(f"Garbage Collecting: {hex(id(self))} - TicketID - {self.id} - TicketName - {self.title}", "memory.txt")

@dataclass

class ValidUpdateFields():

"""

A dataclass representing the valid fields that can be updated for a ticket.

Attributes:

priority (int): The priority level of the ticket.

ticket\_type (TicketType): The type of the ticket.

title (str): The title of the ticket.

description (str): The description of the ticket.

"""

priority: int

ticket\_type: TicketType

title: str

description: str

### 14: repositories/base\_repository.py

from dataclasses import fields

from datetime import datetime

from enum import Enum, EnumMeta

from typing import List, Optional

from models.base\_model import BaseModel

from repositories.database\_connection import DatabaseConnection

from typing import Type, TypeVar, Generic

T = TypeVar('T', bound=BaseModel)

class BaseRepository(Generic[T]):

"""

BaseRepository is a generic class to handle database operations for any model that extends the BaseModel.

Attributes

----------

type\_map : dict

A mapping from Python types to SQLite types used for table creation.

db : DatabaseConnection

The database connection object used to execute queries.

table\_name : str

The name of the table associated with the repository.

model\_class : Type[T]

The class of the model that this repository manages.

Methods

-------

create\_table(model: T)

Creates a table if it does not exist based on the fields in the model.

add\_to\_db(item: T) -> T

Adds a new item to the database and returns the newly created item with the assigned ID.

get\_by\_id(id: int) -> T

Fetches an item from the database by ID.

get\_all() -> List[T]

Fetches all items from the database.

get\_count\_from\_table()

Fetches the total count of records in the database.

delete\_record(id: int, date: datetime) -> T

Marks a record as deleted in the database.

complete\_record(id: int, date: datetime) -> T

Marks a record as completed in the database.

"""

type\_map = {

int: "INTEGER",

Optional[int]: "INTEGER PRIMARY KEY",

float: "REAL",

bool: "BOOLEAN",

str: "TEXT",

datetime: "TEXT"

}

def \_\_init\_\_(self, db\_connection: DatabaseConnection, table\_name: str, model\_class: Type[T]):

self.db = db\_connection

self.table\_name = table\_name

self.model\_class = model\_class

self.create\_table(self.model\_class)

def create\_table(self, model: T):

column\_definitions = []

#Map generic fields to a predefined SQL Type

for field in fields(model):

sql\_type = self.type\_map.get(field.type)

#Fringe condition for Enums.

if isinstance(field.type, EnumMeta):

sql\_type = "TEXT"

if not sql\_type:

raise ValueError(f"Unsupported field type: {field.type} for field {field.name}")

column\_definitions.append(f"{field.name} {sql\_type}")

self.db.cursor.execute(f'''

CREATE TABLE IF NOT EXISTS {self.table\_name} (

{', '.join(column\_definitions)}

)

''')

self.db.connection.commit()

def add\_to\_db(self, item: T) -> T:

column\_names = [f.name for f in fields(item) if f.name != 'id']

query = f"""

INSERT INTO {self.table\_name} ({', '.join(column\_names)})

VALUES ({', '.join('?' for \_ in column\_names)})

"""

values = []

#Use of Enums require us to test each property before returning the value.

for column in column\_names:

value = getattr(item, column)

if isinstance(value, Enum):

value = str(value.value)

values.append(value)

self.db.cursor.execute(query, values)

self.db.connection.commit()

return self.get\_by\_id(self.db.cursor.lastrowid)

def get\_by\_id(self, id: int) -> T:

self.db.cursor.execute(f'SELECT \* FROM {self.table\_name} WHERE id = ?', (id,))

result = self.db.cursor.fetchone()

return self.model\_class(\*result) if result else None

def get\_all(self) -> List[T]:

self.db.cursor.execute(f'SELECT \* FROM {self.table\_name}')

return [self.model\_class(\*row) for row in self.db.cursor.fetchall()]

def get\_count\_from\_table(self):

self.db.cursor.execute(f"SELECT COUNT(\*) FROM {self.table\_name}")

return self.db.cursor.fetchone()[0]

def delete\_record(self, id: int, date: datetime) -> T:

query = f"""

UPDATE {self.table\_name}

SET deleted = ?, updated\_date = ?

WHERE id = ?

"""

self.db.cursor.execute(query, (1, date, id))

self.db.connection.commit()

return self.get\_by\_id(id)

def complete\_record(self, id: int, date: datetime) -> T:

query = f"""

UPDATE {self.table\_name}

SET completed = ?, updated\_date = ?

WHERE id = ?

"""

self.db.cursor.execute(query, (1, date, id))

self.db.connection.commit()

return self.get\_by\_id(id)

### 15: repositories/ticket\_repository.py

from datetime import datetime

from models.ticket\_model import TicketModel

from repositories.base\_repository import BaseRepository

from repositories.database\_connection import DatabaseConnection

from repositories.seed\_database\_tickets import generate\_dummy\_tickets

class TicketRepository(BaseRepository[TicketModel]):

"""

TicketRepository is a class that handles database operations for the TicketModel.

It extends the BaseRepository class.

Attributes

----------

Inherits all attributes from the BaseRepository class.

Methods

-------

\_\_init\_\_(db\_conn: DatabaseConnection, table\_name: str)

Initializes a new instance of the TicketRepository, and seeds the database if it's empty.

seed\_database()

Seeds the database with dummy ticket records.

update\_record(id: int, updated\_record: TicketModel, date: datetime)

Updates an existing ticket record in the database.

log\_time(id: int, remaining\_time: float, logged\_time: float, date: datetime) -> TicketModel

Logs the time for a ticket and updates the remaining and logged time in the database.

"""

def \_\_init\_\_(self, db\_conn: DatabaseConnection, table\_name: str):

super().\_\_init\_\_(db\_conn, table\_name, TicketModel)

if (self.get\_count\_from\_table() == 0):

print("First time run detected, beginning database seeding...")

self.seed\_database()

print("Database seed complete!")

def seed\_database(self):

for record in generate\_dummy\_tickets():

self.add\_to\_db(record)

def update\_record(self, id: int, updated\_record: TicketModel, date: datetime):

query = f"""

UPDATE {self.table\_name}

SET updated\_date = ?,

priority = ?,

ticket\_type = ?,

title = ?,

description = ?

WHERE id = ?

"""

values = (

date,

updated\_record.priority,

updated\_record.ticket\_type,

updated\_record.title,

updated\_record.description,

id

)

self.db.cursor.execute(query, values)

self.db.connection.commit()

return self.get\_by\_id(id)

def log\_time(self, id: int, remaining\_time: float, logged\_time: float, date: datetime) -> TicketModel:

query = f"""

UPDATE {self.table\_name}

SET remaining\_time = ?, logged\_time = ?, updated\_date = ?

WHERE id = ?

"""

self.db.cursor.execute(query, (remaining\_time, logged\_time, date, id))

self.db.connection.commit()

return self.get\_by\_id(id)

### 16: repositories/database\_connection.py

import sqlite3

from environment.env import DB\_CONNECTION

class DatabaseConnection:

"""

Database Instance that will be passed to our Application

Methods

-------

close(self)

Closes the database connection.

"""

def \_\_init\_\_(self):

self.connection = sqlite3.connect(DB\_CONNECTION)

self.cursor = self.connection.cursor()

def close(self):

self.connection.close()

### 17: repositories/seed\_database\_tickets.py

from datetime import datetime, timedelta

from enums.ticket\_type import TicketType

from models.ticket\_model import TicketModel

def generate\_dummy\_tickets() -> list[TicketModel]:

"""

Generates the initial seed data for our application.

"""

return [

TicketModel(

id=None,

created\_date=datetime.now() - timedelta(days = 5, hours = 2, minutes = 35),

updated\_date=datetime.now() - timedelta(days = 5, hours = 2, minutes=35),

completed=False,

deleted=False,

priority=3,

ticket\_type=TicketType.DESIGN,

title="Project Uno Requirements Analysis",

description="Time allocated for meetings with Client Uno to discuss a new CMS based eCommerce online store, and documentation design.",

initial\_estimate=16.0,

remaining\_time=12.0,

logged\_time=4.0,

),

TicketModel(

id=None,

created\_date=datetime.now() - timedelta(days = 4, hours = 12, minutes = 50),

updated\_date=datetime.now() - timedelta(days = 4, hours = 12, minutes = 50),

completed=False,

deleted=False,

priority=4,

ticket\_type=TicketType.DOCUMENTATION,

title="Document the requirements for Project Uno.",

description="Identify any areas of development that may not be considered in the initial requirements provided by the Client.",

initial\_estimate=7.5,

remaining\_time=7.5,

logged\_time=0,

),

TicketModel(

id=None,

created\_date=datetime.now() - timedelta(days = 4, hours = 19, minutes = 50),

updated\_date=datetime.now() - timedelta(days = 4, hours = 19, minutes = 50),

completed=False,

deleted=False,

priority=3,

ticket\_type=TicketType.DEVELOPMENT,

title="Build a proof of Concept system for Project Uno",

description="Client Uno wants a small proof of concept to be built to showcase the system before we fully commit to building their Website from the ground up.",

initial\_estimate=37.5,

remaining\_time=37.5,

logged\_time=0,

),

TicketModel(

id=None,

created\_date=datetime.now() - timedelta(days = 3, hours = 6, minutes = 20),

updated\_date=datetime.now() - timedelta(days = 1),

completed=True,

deleted=False,

priority=1,

ticket\_type=TicketType.SUPPORT,

title="Printer Service for Application Xeta is down.",

description="This ticket went over. Client Xata needs their Microsoft Print Manager Service to be urgently fixed as shipping labels for products aren't printing",

initial\_estimate=4,

remaining\_time=0,

logged\_time=6,

),

TicketModel(

id=None,

created\_date=datetime.now(),

updated\_date=datetime.now(),

completed=False,

deleted=True,

priority=3,

ticket\_type=TicketType.SUPPORT,

title="Faulty Deleted",

description="Deleted Description that requires extra text for validation",

initial\_estimate=0,

remaining\_time=0,

logged\_time=0,

),

TicketModel(

id=None,

created\_date=datetime.now() - timedelta(days = 2, hours = 18, minutes = 33),

updated\_date=datetime.now() - timedelta(days = 2, hours = 18, minutes = 33),

completed=False,

deleted=False,

priority=3,

ticket\_type=TicketType.TESTING,

title="Testing of Client Uno's Online Shop",

description="Time allocated for testing the functionality of the website against the defined requirements",

initial\_estimate=22.5,

remaining\_time=22.25,

logged\_time=0,

),

TicketModel(

id=None,

created\_date=datetime.now() - timedelta(days = 2, hours = 2, minutes = 5),

updated\_date=datetime.now(),

completed=True,

deleted=False,

priority=1,

ticket\_type=TicketType.SUPPORT,

title="Emergency database restore.",

description="This ticket will go under time. Client Alpha has botched their database and need their database to be recovered.",

initial\_estimate=4,

remaining\_time=2,

logged\_time=2,

),

TicketModel(

id=None,

created\_date=datetime.now() - timedelta(days = 1, hours = 20, minutes = 55),

updated\_date=datetime.now() - timedelta(days = 1, hours = 20, minutes = 55),

completed=False,

deleted=False,

priority=3,

ticket\_type=TicketType.DEVELOPMENT,

title="Primary development allocated for Client Uno's Web Shop",

description="Time quoted for development of Client Uno's Web Shop, pending signoff after POC work is completed.",

initial\_estimate=150,

remaining\_time=150,

logged\_time=0,

),

TicketModel(

id=None,

created\_date=datetime.now() - timedelta(days = 1, hours = 4, minutes = 30),

updated\_date=datetime.now() - timedelta(hours = 4, minutes = 30),

completed=False,

deleted=False,

priority=4,

ticket\_type=TicketType.DEPLOYMENT,

title="Set up a CI/CD Pipeline for rapid deployment of features.",

description="Client will require continued support after the initial release of their product, we need a solid CI/CD pipeline to be able to deliver support and future development",

initial\_estimate=50,

remaining\_time=25,

logged\_time=25,

),

TicketModel(

id=None,

created\_date=datetime.now() - timedelta(hours = 12, minutes = 30),

updated\_date=datetime.now() - timedelta(hours = 6, minutes = 30),

completed=True,

deleted=False,

priority=5,

ticket\_type=TicketType.DEVELOPMENT,

title="AWS Cloud environments",

description="This ticket is massively over. Client Uno requires a Dev, Staging & Prod environment on AWS that integrates with our CI/CD pipeline. Load balancers need to be configured for horizontal scaling during promotions",

initial\_estimate=75,

remaining\_time=0,

logged\_time=70,

)

]

### 18: services/terminal/formatter.py

from datetime import datetime

class Formatter():

"""

Formatter is a utility class for formatting various types of data into a more human-readable format.

Methods

-------

make\_readable(key: str) -> str

Transforms a snake\_case string into a properly spaced and capitalized string.

day\_suffix(day: int)

Returns the correct ordinal suffix for a day of the month.

format\_date(date: datetime)

Formats a datetime object into a readable string with the correct day suffix.

format\_object(obj) -> str

Takes an object and formats each of its properties into a list of readable strings.

"""

def make\_readable(self, key: str) -> str:

return ' '.join(map(str.capitalize, key.split('\_')))

def day\_suffix(self, day: int) -> str:

return ("th" if 4 <= day <= 20 else {1: "st", 2: "nd", 3: "rd"}.get(day % 10, "th"))

def format\_date(self, date: datetime) -> str:

return date.strftime(f"%H:%M %p %d{self.day\_suffix(date.day)} %B %y")

def format\_object(self, obj) -> str:

formatted\_entries = []

for (key, value) in vars(obj).items():

if isinstance(value, datetime):

formatted\_value = self.format\_date(value)

else:

formatted\_value = str(value)

formatted\_entries.append(f"{self.make\_readable(key)}: {formatted\_value} ")

formatted\_entries.append(" ")

return formatted\_entries

### 19: services/terminal/output\_manager.py

import dataclasses

from helpers.extension\_helpers import check\_option\_in\_range, find\_option\_by\_number, validate\_as\_int

from helpers.print\_colours import print\_beige, print\_blue, print\_green, print\_red, print\_yellow, prompt\_underline

from services.terminal.formatter import Formatter

from services.terminal.prompt\_manager import PromptManager

class OutputManager:

"""

OutputManager class is responsible for managing and formatting the output presented to the user in the terminal.

It also handles user inputs prompted on the terminal.

Methods

-------

\_print\_single\_line(obj, colour\_func)

Print a single line representation of an object using specified color function.

\_print\_multi\_line(obj, colour\_func)

Print a multi-line representation of an object using specified color function.

\_handle\_print(objects: list, single\_line: bool)

Handles the print operation based on the 'single\_line' flag.

multi\_line(objects: list)

Executes a multi-line print for a list of objects.

single\_line(objects: list)

Executes a single-line print for a list of objects.

startup\_message()

Prints the start options and prompts the user for choice.

no\_records\_found(optional: str | None = None)

Prints a message indicating that no records were found.

print\_options(options)

Prints the provided options to the console.

get\_id\_input(range: int)

Prompts the user for an id input within a given range.

display\_update\_information(items)

Displays update information and options for provided items.

display\_time\_logging\_information(remaining\_time: float)

Displays information about the time logging and prompts the user for the time to log.

show\_time\_logged(time\_logged: float, time\_remaining: float)

Displays a message indicating the successful logging of time.

"""

def \_\_init\_\_(self, formatter: Formatter, prompt\_manager: PromptManager):

self.formatter = formatter

self.prompt\_manager = prompt\_manager

start\_options = [

"1: View Active Records",

"2: View Record by ID",

"3: Add a new Record",

"4: Update by ID",

"5: Delete by ID",

"6: Search for Records",

"7: Show completed records.",

"8: Show deleted records.",

"9: Show all Records Raw & Unfiltered.",

"0: Quit"

]

update\_options = [

"1: Priority",

"2: Ticket Type",

"3: Title",

"4: Description"

]

def \_print\_single\_line(self, obj, colour\_func):

colour\_func(' '.join(self.formatter.format\_object(obj)))

def \_print\_multi\_line(self, obj, colour\_func):

for entry in self.formatter.format\_object(obj):

colour\_func(entry)

def \_handle\_print(self, objects: list, single\_line: bool):

#This is designed to let you call 'multi\_line' or 'single\_line' in our controller without having to care what format the data looks like.

#We can pass a single object or a list of any length without having to validate it at the caller level.

#Using the strategy pattern throughout to decide the behaviour at runtime.

print\_func = self.\_print\_single\_line if single\_line else self.\_print\_multi\_line

if not isinstance(objects, list):

objects = [objects]

if len(objects) <= 0:

self.no\_records\_found()

for obj in objects:

#Print in yellow for even, beige for odd indexes.

colour\_func = print\_yellow if (objects.index(obj) % 2 == 0) else print\_beige

print\_func(obj, colour\_func)

def multi\_line(self, objects: list):

self.\_handle\_print(objects, single\_line=False)

def single\_line(self, objects: list):

self.\_handle\_print(objects, single\_line=True)

def startup\_message(self):

self.print\_options(self.start\_options)

return self.prompt\_manager.get\_user\_choice(self.start\_options)

def no\_records\_found(self, optional: str | None = None):

print\_red("No records found.")

if (optional): print\_red(optional)

def print\_options(self, options: list[str]):

for opt in options:

print\_blue(opt)

def get\_id\_input(self, range: int):

return self.prompt\_manager.get\_number\_from\_user\_in\_range(range)

def display\_update\_information(self, items: list):

self.multi\_line(items)

print\_green("Please note, once a ticket has been created, the initial estimate cannot be changed.\nPlease open a record by ID to log time. ")

self.print\_options(self.update\_options)

return self.prompt\_manager.get\_number\_from\_user\_in\_range(len(self.update\_options))

def display\_time\_logging\_information(self, remaining\_time: float):

print\_green(f"Available time: {remaining\_time}")

time\_to\_log = self.prompt\_manager.get\_number\_from\_user\_in\_range(remaining\_time)

if time\_to\_log == remaining\_time:

print\_green("Ticket now complete. ")

return time\_to\_log

def show\_time\_logged(self, time\_logged: float, time\_remaining: float):

print\_green(f"Time logged successfully. {time\_logged} hours logged in total. {time\_remaining} hours remaining. ")

### 20: services/terminal/prompt\_manager.py

from helpers.extension\_helpers import check\_option\_in\_range, find\_option\_by\_number, validate\_as\_float, validate\_as\_int

from helpers.print\_colours import print\_green, print\_red, prompt\_underline

class PromptManager:

"""

PromptManager class is responsible for managing user inputs from the terminal. It provides various functions to take user input and perform validation checks.

Methods

-------

get\_user\_choice(options)

Prompts the user to make a choice from a list of options and returns the chosen option.

get\_number\_from\_user(as\_float: bool)

Prompts the user to enter a number, validates the input, and returns the number as a float if 'as\_float' is true, or as an int otherwise.

get\_number\_from\_user\_in\_range(top\_end: int | float)

Prompts the user to enter a number within a specified range and returns the number.

get\_raw\_string\_from\_user() -> str

Prompts the user to enter a string and returns the input string.

get\_user\_confirmation(opt\_string: str, id: int = 0) -> bool

Prompts the user for a Yes or No confirmation and returns True if 'Yes', or False otherwise.

continue\_message()

Displays a message prompting the user to press any key to continue.

"""

def get\_user\_choice(self, options: list[str]) -> int | float:

choice = self.get\_number\_from\_user\_in\_range(len(options))

print\_green(f"\n{find\_option\_by\_number(options, choice)}\n")

return choice

def get\_number\_from\_user(self, as\_float: bool) -> int | float:

while True:

user\_input = prompt\_underline("\nPlease enter your choice (Number): ")

if (not as\_float and validate\_as\_int(user\_input) == False):

print\_red(f"\n'{user\_input}' is not a number.")

continue

elif (as\_float and validate\_as\_float(user\_input) == False):

print\_red(f"\n'{user\_input}' is not a number.")

continue

return int(user\_input) if not as\_float else float(user\_input)

def get\_number\_from\_user\_in\_range(self, top\_end: int | float) -> int | float:

while True:

choice = self.get\_number\_from\_user(as\_float = True if isinstance(top\_end, float) else False)

if (check\_option\_in\_range(top\_end, choice) == False):

print\_red(f"\n{choice} was not within range 1 to {top\_end}. ")

continue

return choice

def get\_raw\_string\_from\_user(self) -> str:

return prompt\_underline("\nPlease enter your search query: ")

def get\_user\_confirmation(self, opt\_string: str, id: int = 0) -> bool:

while True:

print\_green(f"Do you want to {opt\_string}{id if id > 0 else ''}?\n")

response = prompt\_underline(f"Enter Y for Yes or N for No: ").strip()

if response.lower() in ['y', 'n']:

return response.lower() == 'y'

else:

print\_red("\nInvalid input, please enter Y or N.")

def continue\_message(self):

prompt\_underline("\nPress any key to continue.\n ")

### 21: services/tickets/ticket\_input\_validation\_service.py

from helpers.extension\_helpers import check\_option\_in\_range, find\_option\_by\_number, validate\_as\_float, validate\_as\_int

from helpers.print\_colours import print\_green, print\_red, prompt\_underline

class PromptManager:

"""

PromptManager class is responsible for managing user inputs from the terminal. It provides various functions to take user input and perform validation checks.

Methods

-------

get\_user\_choice(options)

Prompts the user to make a choice from a list of options and returns the chosen option.

get\_number\_from\_user(as\_float: bool)

Prompts the user to enter a number, validates the input, and returns the number as a float if 'as\_float' is true, or as an int otherwise.

get\_number\_from\_user\_in\_range(top\_end: int | float)

Prompts the user to enter a number within a specified range and returns the number.

get\_raw\_string\_from\_user() -> str

Prompts the user to enter a string and returns the input string.

get\_user\_confirmation(opt\_string: str, id: int = 0) -> bool

Prompts the user for a Yes or No confirmation and returns True if 'Yes', or False otherwise.

continue\_message()

Displays a message prompting the user to press any key to continue.

"""

def get\_user\_choice(self, options: list[str]) -> int | float:

choice = self.get\_number\_from\_user\_in\_range(len(options))

print\_green(f"\n{find\_option\_by\_number(options, choice)}\n")

return choice

def get\_number\_from\_user(self, as\_float: bool) -> int | float:

while True:

user\_input = prompt\_underline("\nPlease enter your choice (Number): ")

if (not as\_float and validate\_as\_int(user\_input) == False):

print\_red(f"\n'{user\_input}' is not a number.")

continue

elif (as\_float and validate\_as\_float(user\_input) == False):

print\_red(f"\n'{user\_input}' is not a number.")

continue

return int(user\_input) if not as\_float else float(user\_input)

def get\_number\_from\_user\_in\_range(self, top\_end: int | float) -> int | float:

while True:

choice = self.get\_number\_from\_user(as\_float = True if isinstance(top\_end, float) else False)

if (check\_option\_in\_range(top\_end, choice) == False):

print\_red(f"\n{choice} was not within range 1 to {top\_end}. ")

continue

return choice

def get\_raw\_string\_from\_user(self) -> str:

return prompt\_underline("\nPlease enter your search query: ")

def get\_user\_confirmation(self, opt\_string: str, id: int = 0) -> bool:

while True:

print\_green(f"Do you want to {opt\_string}{id if id > 0 else ''}?\n")

response = prompt\_underline(f"Enter Y for Yes or N for No: ").strip()

if response.lower() in ['y', 'n']:

return response.lower() == 'y'

else:

print\_red("\nInvalid input, please enter Y or N.")

def continue\_message(self):

prompt\_underline("\nPress any key to continue.\n ")

### 22: services/tickets/ticket\_service.py

from dataclasses import fields

from controllers.log\_controller import LogController

from models.preview\_model import TicketPreviewModel

from repositories.ticket\_repository import TicketRepository

from models.ticket\_model import TicketModel, ValidUpdateFields, ValidatedUserTicketInputs

from datetime import datetime

from services.base\_service import BaseService

class TicketService(BaseService[TicketModel]):

def \_\_init\_\_(self, ticket\_repository: TicketRepository):

super().\_\_init\_\_(ticket\_repository)

self.ticket\_repository = ticket\_repository

def build\_ticket(self, validated\_inputs: ValidatedUserTicketInputs) -> TicketModel:

"""

Create a new ticket based on validated user inputs and automatically calculated properties.

Args:

validated\_inputs (ValidatedUserTicketInputs): The validated inputs for the new ticket.

Returns:

TicketModel: The new ticket.

"""

return TicketModel(

id=None,

created\_date=datetime.now(),

updated\_date=datetime.now(),

completed=False,

deleted=False,

priority=validated\_inputs.priority,

ticket\_type=validated\_inputs.ticket\_type,

title=validated\_inputs.title,

description=validated\_inputs.description,

initial\_estimate=validated\_inputs.initial\_estimate,

remaining\_time=validated\_inputs.initial\_estimate,

logged\_time=0.0

)

def create\_ticket(self, validated\_inputs) -> TicketModel:

return self.ticket\_repository.add\_to\_db(self.build\_ticket(validated\_inputs))

def create\_preview(self, tickets: list[TicketModel]) -> list[TicketPreviewModel]:

return [TicketPreviewModel(t.id, t.priority, t.title, t.remaining\_time) for t in tickets]

def order\_by\_priority(self, tickets: list[TicketModel]) -> list[TicketModel]:

"""

Went for a count sort here as we already know the values we are sorting (1-5)

Create a list for each priority number, iterate over tickets store by priority as index, re-join the list.

"""

LogController.new\_log("Bucket Sorting...")

buckets = [[] for \_ in range(5)]

for t in tickets:

buckets[t.priority - 1].append(t)

LogController.new\_log("Finished BucketSort")

LogController.new\_log("QuickSorting buckets...")

sorted\_buckets = [self.sort\_remaining\_time(bucket) for bucket in buckets]

LogController.new\_log("Finished QuickSort")

return [ticket for bucket in sorted\_buckets for ticket in bucket]

def sort\_remaining\_time(self, tickets: list[TicketModel]):

"""

Very Basic Quick Sort Algorithm to order by remaining time.

QS is recursive, so we have our exit case immediately.

Split our list into a left/right where remaining time is less/greater than the middle element.

Recursively call on each list until sorted.

TODO: This is a basic, memory inefficient implementation, revisit later.

"""

if len(tickets) <= 1:

return tickets

pivot = tickets[len(tickets) // 2]

left = [lower for lower in tickets if (lower.remaining\_time < pivot.remaining\_time)]

middle = [ticket for ticket in tickets if ticket.remaining\_time == pivot.remaining\_time]

right = [greater for greater in tickets if (greater.remaining\_time > pivot.remaining\_time)]

return self.sort\_remaining\_time(left) + middle + self.sort\_remaining\_time(right)

def search\_string\_values(self, tickets: list[TicketModel], query: str):

return [t for t in tickets if query.upper() in (t.title.upper() or t.description.upper())]

def delete\_record(self, id: int) -> TicketModel:

return self.ticket\_repository.delete\_record(id, datetime.now())

def populate\_update\_fields(self, ticket: TicketModel) -> ValidUpdateFields:

data = {f.name: getattr(ticket, f.name) for f in fields(ValidUpdateFields)}

return ValidUpdateFields(\*\*data)

def update\_record(self, ticket: TicketModel, field\_to\_update: int, updated\_value) -> TicketModel:

setattr(ticket, [f.name for f in fields(ValidUpdateFields)][field\_to\_update - 1], updated\_value)

return self.ticket\_repository.update\_record(ticket.id, ticket, datetime.now())

def update\_ticket\_time(self, ticket: TicketModel, time):

"""Update the logged time and remaining time for a ticket."""

updated\_remaining\_time = ticket.remaining\_time - time

updated\_logged\_time = ticket.logged\_time + time

updated\_ticket = self.ticket\_repository.log\_time(ticket.id, updated\_remaining\_time, updated\_logged\_time, datetime.now())

return updated\_ticket

def check\_and\_complete\_ticket(self, ticket: TicketModel):

"""Check if a ticket is completed and mark it as such."""

if ticket.remaining\_time <= 0:

LogController.new\_log(f"Marking ticket: {ticket.id} complete. ")

self.ticket\_repository.complete\_record(ticket.id, datetime.now())

return ticket

def log\_time(self, ticket: TicketModel, time):

"""Log time for a ticket and check its completion status."""

LogController.new\_log(f"Logging {time} to ticket: {ticket.id} with {ticket.remaining\_time} remaining. ")

updated\_ticket = self.update\_ticket\_time(ticket, time)

LogController.new\_log(f"Successfully logged {time} to ticket: {ticket.id} with {ticket.remaining\_time} remaining. ")

completed\_ticket = self.check\_and\_complete\_ticket(updated\_ticket)

return completed\_ticket

### 23: services/base\_service.py

from models.base\_model import BaseModel

from repositories.base\_repository import BaseRepository

from typing import List, TypeVar, Generic

#Enforce that any Class passed to this service must implement our BaseModel

#This is necessary to generically work with properties found on our BaseModel such as completed & deleted whilst ensuring type safety.

T = TypeVar('T', bound=BaseModel)

class BaseService(Generic[T]):

def \_\_init\_\_(self, repository: BaseRepository[T]):

self.repository = repository

self.model\_class = repository.model\_class

def get\_by\_id(self, id) -> T | None:

return self.repository.get\_by\_id(id)

def get\_all(self) -> List[T | None]:

return self.repository.get\_all()

def get\_count(self) -> int:

return self.repository.get\_count\_from\_table()

def filter\_deleted(self, bool: bool, items: list[T]) -> list[T | None]:

return list(filter(lambda t: t.deleted == bool, items))

def filter\_complete(self, bool: bool, items: list[T]) -> list[T | None]:

return list(filter(lambda t: t.completed == bool, items))

def check\_if\_deleted(self, item: T) -> bool:

return item.deleted

def check\_if\_completed(self, item: T) -> bool:

return item.completed

## Appendix C – Application Automated Testing Source Code