# Design Specification

## Use Case Diagram:

## Data Store Specification

## Application Structure & Architecture

I have used the dependency injection pattern to implement an N-tier architecture for this application.

The application is composed of individual modules that each have a single responsibility within the programme. Each core module is designed to be generic in implementation, so the application can be quickly and safely extended for new requirements and use cases.

A TicketController is responsible for interfacing directly with the user, allowing them to navigate through the use cases of the application without doing any of the heavy lifting itself. It is responsible for calling the appropriate services, modules and functions depending on the needs of the user.

The controller contains an Output & Prompt Manager the handles print & read actions as well as validating user inputs. It’s role is to ensure that no corrupt data can reach our service layer, alleviating errors before they have a chance to happen and finally displaying information to the user in a clear and concise manner. It has it’s own Formatter module that is responsible for presentation of our data, making it as human readable as possible.

The controller has a TicketService that is responsible the manipulation of application data as well as diverting database Create, Read, Update and Delete operations to it’s own internal repository. This includes filtering and sorting, calculation of automatically generated data such as date timestamps and calculations of time in our tickets. It builds our tickets for us too.

The controller has a TicketValidationService that handles the generation and validation of Ticket fields that require user input. This is a separate module from the TicketService as we never want to send any non-conformant data to our service. Validation is not a part of the TicketService’s responsibility so the we leverage this to ensure the TicketService can complete it’s own responsibilities.

Our TicketService contains a TicketRepository which is solely responsible for database actions. No validation is necessary at this level as our higher level Validators and Services promise that we have no erroneous data.

Working with generics, our TicketService & TicketRepository both inherit from a BaseService & BaseRepository respectively. All future Services & Repositories will extend these base classes as they contain functionality that doesn’t depend on specific implementation logic. They simply need to be passed a Data Model, Dataclass in Python and they will function as desired. Examples of common functionality are methods like GetById, DeleteById etc, every database table created will always have an ID PK so we can pass in the Table Name at runtime with no modification needed.

When the application is built, a Configuration Builder module is instantiated to compose our Ticket application, any implementation of this will share a Database Connection to ensure that the state of our database is concurrent even if read/write actions come from separate implementations of our base configuration. This abstraction means that our “main.py” application is devoid of any implementation logic.

Our TicketModel is composed three things, a BaseModel that contains generic properties every Model & database table will contain across the application, such as an unique Identifier, date of creation, date of last update and whether they have been deleted or not. Then a sub-class for any fields that are reliant on user input. This is so when a user creates a new ticket, inputs will first be stored in a ValidatedUserInputs class while they’re being generated & validated. Then this can safely be passed to the TicketService to handle the creation of the actual TicketModel. Then the specific properties to the TicketModel that do not need validation.

We use soft delete. This is because true delete operations are considered bad practice outside of extreme circumstances related to data science and engineering.

With all of this in mind, if a developer wished to add new features, such as Customer management, they would only need to create a new CustomerModel and CustomerController, inheriting from our Base classes & composed in our Configuration Builder and the generic functionality will already be available. Then they can extend it for their own requirements.

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.