

Restaurant Management System

Problem Statement:

A mid-sized restaurant faces operational challenges due to manual order processing, table booking, and billing procedures. These manual processes lead to delays, errors, and poor customer experience. To modernize operations, you are tasked with designing and developing a Java console-based Restaurant Management System that supports real-time updates and multiple concurrent users.

Solution:

In this system, when a customer enters the restaurant, they first view the list of tables displaying table number, seating capacity, and availability status. The customer selects a single table to book, and since multiple customers may attempt booking the same table concurrently, the booking operation is synchronized using a shared persistent store (file/JSON) to ensure consistency. If two customers try to reserve Table 1 at the same time, only the first successful request locks the table, while the second customer receives a message that the table is already booked and must choose another one. After confirming the table booking, the customer can browse the menu, select multiple items with quantities, and place an order. Each order is treated as a task and submitted into an order queue, where waiter worker threads (managed using a fixed-size ExecutorService) pick up incoming orders and dispatch them to the kitchen queue. Chef worker threads then process these tasks concurrently, simulate preparation time, and upon completion update the order status to PREPARED, enabling smooth parallel execution of restaurant operations from booking to food preparation.

User Story 1: On-Arrival Table Booking

As a customer,

I want to view all tables with their seating capacity and availability status when I arrive,
so that I can choose and book a suitable table immediately.

Acceptance Criteria:

The system displays all tables with table number, seats, and status (Available/Booked).

Customers can select one or more tables for booking.
Booked tables are immediately marked unavailable.

User Story 2: Concurrent Table Booking Conflict Handling

As a customer,

I want the system to prevent double-booking when multiple customers try to book tables at the same time,
so that table allocation remains accurate and fair.

Acceptance Criteria:

Two customers cannot book the same table simultaneously.
If some requested tables are already booked, the customer receives partial allocation.
Customers are asked to confirm or cancel the booking based on available tables.

User Story 3: Menu Browsing and Item Selection

As a customer,

I want to view the restaurant menu with item details such as price and preparation time,
so that I can decide what to order.

Acceptance Criteria:

Menu shows item name, price, estimated cooking time, and available quantity.
Customers can select multiple items with required quantities.

User Story 4: Placing an Order from the Table

As a customer,

I want to place my selected food items as an order after booking a table,
so that the kitchen can start preparing my food.

Acceptance Criteria:

Order is created with table ID and selected menu items.
Order is submitted successfully into the restaurant's processing pipeline.
The customer receives confirmation that the order was placed.

User Story 5: Checkout Request After Order Completion

As a customer,

I want to request checkout only after all my orders are prepared,
so that I can complete my dining session correctly.

Acceptance Criteria:

Bill/checkout option is enabled only when all orders for that table are

completed.

Customers cannot checkout if orders are still pending.

User Story 6: Waiter Thread Order Forwarding

As the restaurant system,

I want waiter worker threads to pick up customer orders and forward them to the kitchen queue,
so that orders are delivered in real time without manual delays.

Acceptance Criteria:

Orders placed by customers are assigned to waiter threads.

Waiter threads push orders into the kitchen queue immediately.

User Story 7: Kitchen Queue Management

As the restaurant system,

I want incoming orders to be stored in a synchronized kitchen queue,
so that chefs can process them efficiently in the correct order.

Acceptance Criteria:

Orders are queued safely using a thread-safe structure (e.g., BlockingQueue).
Multiple chefs can consume orders without conflict.

User Story 8: Concurrent Order Preparation by Chefs

As the restaurant system,

I want chef worker threads to prepare multiple orders simultaneously,
so that service remains efficient during peak hours.

Acceptance Criteria:

Chef threads process orders concurrently using a thread pool.

Order status updates from RECEIVED → IN_PROGRESS → COMPLETED.

User Story 9: Real-Time Order Status Updates

As the restaurant system,

I want order status changes to be reflected immediately across customer and kitchen modules,
so that all users see live updates.

Acceptance Criteria:

When chefs complete an order, customers can see the updated status.
Kitchen view always shows active and completed orders correctly.

User Story 10: Automatic Table Release After Checkout**As the restaurant system,**

I want tables to be automatically marked available after checkout,
so that they can be booked again by new customers.

Acceptance Criteria:

Once checkout is complete, table status resets to Available.
Released tables instantly appear free in the shared table view.

User Story 11: Shared Table Availability Across Multiple Terminals**As the restaurant system,**

I want table booking data to remain consistent across multiple running
instances of the application,
so that customers booking from different terminals see the same live
availability.

Acceptance Criteria:

Table status is stored in a shared file-based persistence layer.
File locking prevents race conditions during concurrent bookings.
Updates made in one terminal are visible immediately in another terminal.

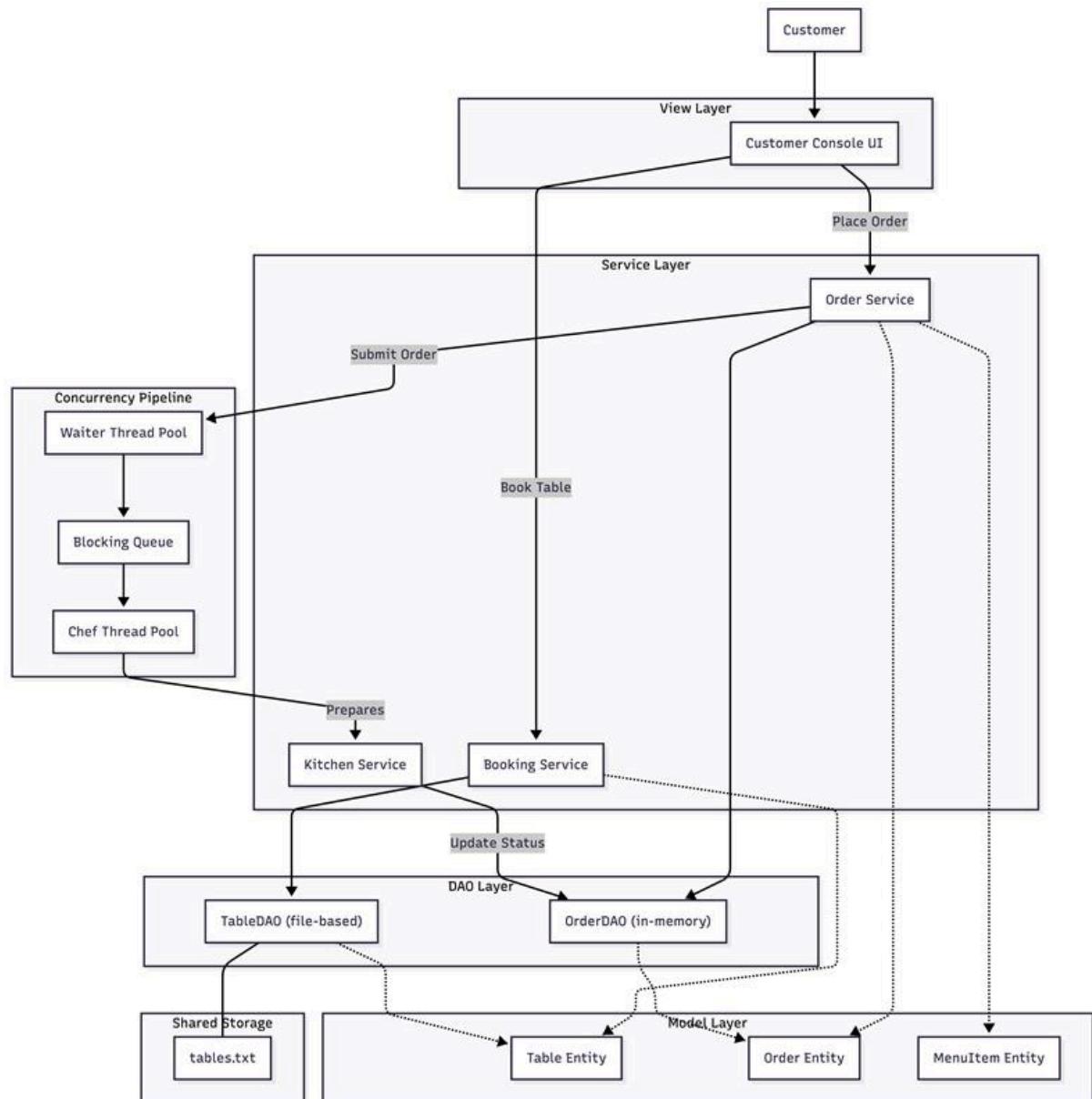


Fig-1:High Level Design

Fig-1 represents a **layered architecture** designed for a restaurant management system. The system follows clear separation of concerns across View, Service, DAO, and Model layers, with asynchronous order processing.

Flow of the project

- Customers interact with the system.

- Requests go to the View Layer (Customer Console).
- The View delegates actions to the Service Layer.
- Services interact with:
 - DAO layer (data access)
 - Concurrency pipeline (for order processing)
- Data is stored in:
 - File-based storage (tables)
 - In-memory storage (orders)

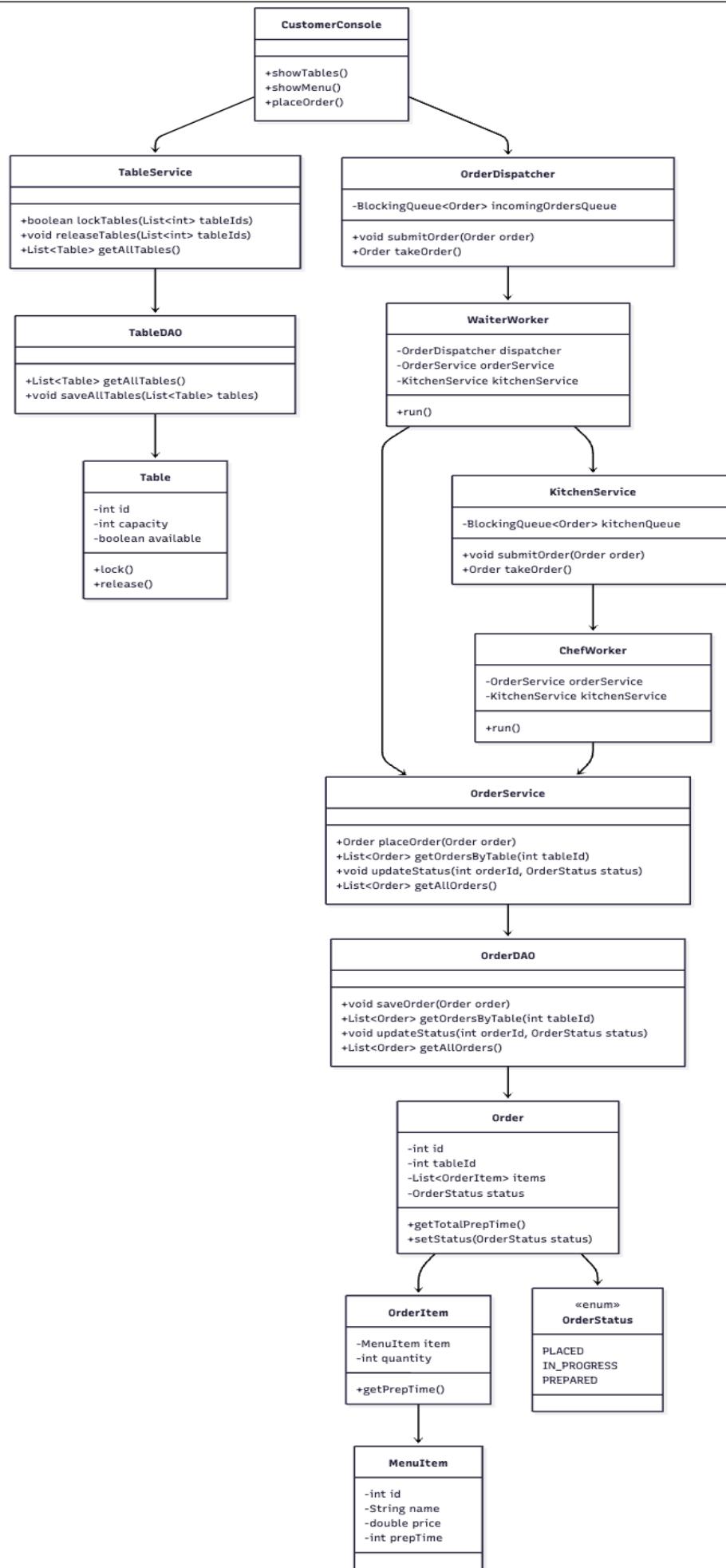


Fig-2: Low Level design

1. Model Layer (Domain Layer)

The Model Layer contains the core business entities of the restaurant system. These classes represent real-world objects such as tables, menu items, and orders. This layer contains pure domain logic and remains independent of upper layers like services or views.

The MenuItem class represents a dish available in the restaurant menu. It stores static information such as a unique identifier (id), the name of the dish (name), its price (price), and the preparation time required (prepTime). Its main responsibility is to describe a menu dish and provide basic details for ordering.

The OrderItem class represents a single line item inside a customer order. It consists of a MenuItem along with the quantity ordered. This class helps calculate preparation time based on the selected dish and quantity, making it an essential building block for complete orders.

The OrderStatus enum defines the lifecycle of an order in a type-safe manner. It includes the states PLACED, IN_PROGRESS, and PREPARED. Using an enum prevents invalid string-based status values and ensures consistent order state transitions.

The Order class represents a complete customer order. It contains a unique order ID, the associated table ID, a list of OrderItem objects, and the current status of the order. It provides methods such as getTotalPrepTime() to compute the overall preparation time and lifecycle-related methods like markPrepared() to update the order state. This class encapsulates order behavior and manages the order lifecycle.

The Table class represents a restaurant table. It stores information such as table ID, seating capacity, and availability status. It provides methods like lock() to mark a table as unavailable and release() to make it available again. This class encapsulates state changes related to table booking.

2. DAO Layer (Data Access Layer)

The DAO Layer is responsible for data storage and retrieval. It isolates persistence logic from business logic, ensuring that the service layer does not directly handle file operations or storage details.

The TableDAO component handles persistent storage of table data using a file-based approach. It provides methods such as getAllTables() to retrieve all tables from storage, lockTables() to reserve specified tables, releaseTables() to free them, and saveToFile() to persist updated table information. Its responsibility is to manage table availability consistently and ensure correct booking behavior even under concurrent access.

The OrderDAO component manages order storage using an in-memory approach. It supports operations like saving a new order, retrieving orders for a specific table, and updating the status of an order. This DAO maintains active order data and supports lifecycle state updates during order preparation.

3. Service Layer (Business Logic Layer)

The Service Layer contains the core business logic and orchestration of the system. It coordinates DAO operations, concurrency workers, and domain-level rules.

The BookingService is responsible for table reservation operations. It provides methods such as bookTables() which internally calls TableDAO.lockTables() and returns whether the booking was successful. It also supports releasing tables through TableDAO.releaseTables(). This service ensures that reservation logic is handled cleanly outside the persistence layer.

The OrderService manages the complete lifecycle of an order. When a customer places an order, this service creates a new Order object, saves it through OrderDAO, submits it into the waiter workflow, and returns the created order. It acts as the central coordinator between storage and concurrency components.

The KitchenService manages kitchen processing using a producer-consumer mechanism. It maintains a blocking queue of incoming orders and provides methods to start chef worker threads, submit orders into the kitchen queue, and mark orders as prepared. It updates order status by calling DAO methods and ensures asynchronous kitchen workflow execution.

4. Concurrency Workers

The concurrency layer introduces multithreaded workers to simulate real restaurant operations where multiple orders are processed simultaneously.

The OrderDispatcher acts as the central incoming order queue. Customers submit their orders into this dispatcher, and waiter threads consume orders from

it. It works as a buffer between customer input and waiter processing, ensuring that orders are safely handled in a thread-safe queue.

The WaiterWorker represents waiter threads in the system. Waiters continuously take orders from the OrderDispatcher, save them through OrderService, update their status to IN_PROGRESS, and forward them into the kitchen queue managed by KitchenService. This worker acts as a producer for the kitchen workflow.

The ChefWorker represents chef threads. Chefs continuously consume orders from the kitchen queue, simulate preparation using preparation time, and once completed, update the order status to PREPARED. This worker acts as the final consumer in the producer-consumer pipeline.

5. View Layer (Presentation Layer)

The View Layer provides the user interaction entry point of the system. It does not contain business rules but delegates actions to the service layer.

The CustomerConsole class handles customer interaction through a console interface. It provides methods such as showTables() to display table availability, showMenu() to list menu items, and placeOrder() to collect customer order input. Once input is collected, it calls OrderService.placeOrder() to trigger the complete ordering workflow.

The screenshot shows the IntelliJ IDEA interface with the project 'rms' open. The main editor window displays the 'Main.java' file, which contains the following code:

```
public class Main {    public static void main(String[] args) {        PiyushHMehta        OrderDispatcher orderDispatcher = new OrderDispatcher();        ExecutorService waiterPool = Executors.newFixedThreadPool(POOL_SIZE);        ExecutorService chefPool = Executors.newFixedThreadPool(POOL_SIZE);        for(int threads=0;threads<POOL_SIZE;threads++) {            waiterPool.submit(new WaiterWorker(orderDispatcher, orderService, kitchenService));            chefPool.submit(new ChefWorker(orderService, kitchenService));        }    }}
```

The 'Run' tool bar at the bottom has 'Main' selected. The run configuration dropdown shows the command: /Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=59777 -Dfile.encoding=UTF-8. The status bar at the bottom right indicates the time is 19:14, LF, UTF-8, 4 spaces.

Fig 3 : Customer Console for booking and to view tables

The screenshot shows the IntelliJ IDEA interface with the project 'rms' open. The main editor window displays the 'Main.java' file, which contains the same code as in Fig 3. The 'Run' tool bar at the bottom has 'Main' selected. The run configuration dropdown shows the command: /Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=59777 -Dfile.encoding=UTF-8. The status bar at the bottom right indicates the time is 19:14, LF, UTF-8, 4 spaces.

The output window shows the following log entries:

```
Feb 18, 2026 11:00:10 PM com.zeta.logger.AppLogger.info
INFO: Tables view:
Table{id=1, capacity=4, available=false}
Table{id=2, capacity=4, available=false}
Table{id=3, capacity=4, available=false}
Table{id=4, capacity=4, available=false}
Table{id=5, capacity=4, available=true}
```

Fig 4 : view of tables with availability and capacity of table

The screenshot shows the IntelliJ IDEA interface. The left sidebar displays the project structure under 'Project'. The main editor window shows the code for 'Main.java'.

```
public class Main {    public static void main(String[] args) {        OrderDispatcher orderDispatcher = new OrderDispatcher();        ExecutorService waiterPool = Executors.newFixedThreadPool(POOL_SIZE);        ExecutorService chefPool = Executors.newFixedThreadPool(POOL_SIZE);        for(int threads=0;threads<POOL_SIZE;threads++) {            waiterPool.submit(new WaiterWorker(orderDispatcher, orderService, kitchenService));            chefPool.submit(new ChefWorker(orderService, kitchenService));        }    }}
```

The terminal window at the bottom shows the following output:

```
/Library/Java/JavaVirtualMachines/amazon-corretto-17.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=59777 -Dfile.encoding=UTF-8
1. View Tables
2. Book Table
3. Simulation
4. Exit
1
Feb 18, 2026 11:00:10 PM com.zeta.logger.AppLogger info
INFO: Tables view:
Table{id=1, capacity=4, available=false}
Table{id=2, capacity=4, available=false}
Table{id=3, capacity=4, available=false}
Table{id=4, capacity=4, available=false}
Table{id=5, capacity=4, available=true}
1. View Tables
2. Book Table
3. Simulation
4. Exit
a
Feb 18, 2026 11:00:52 PM com.zeta.logger.AppLogger warning
WARNING: Invalid input. Please enter a number.
```

Fig 5 : Validation for Integer Input

The screenshot shows the IntelliJ IDEA interface. The left sidebar displays the project structure under 'Project'. The main editor window shows the code for 'Main.java'.

```
public class Main {    public static void main(String[] args) {        OrderDispatcher orderDispatcher = new OrderDispatcher();        ExecutorService waiterPool = Executors.newFixedThreadPool(POOL_SIZE);        ExecutorService chefPool = Executors.newFixedThreadPool(POOL_SIZE);        for(int threads=0;threads<POOL_SIZE;threads++) {            waiterPool.submit(new WaiterWorker(orderDispatcher, orderService, kitchenService));            chefPool.submit(new ChefWorker(orderService, kitchenService));        }    }}
```

The terminal window at the bottom shows the following output:

```
1
Feb 18, 2026 11:00:10 PM com.zeta.logger.AppLogger info
INFO: Tables view:
Table{id=1, capacity=4, available=false}
Table{id=2, capacity=4, available=false}
Table{id=3, capacity=4, available=false}
Table{id=4, capacity=4, available=false}
Table{id=5, capacity=4, available=true}
1. View Tables
2. Book Table
3. Simulation
4. Exit
a
Feb 18, 2026 11:00:52 PM com.zeta.logger.AppLogger warning
WARNING: Invalid input. Please enter a number.
2
Enter table id to book:
1
Feb 18, 2026 11:01:23 PM com.zeta.logger.AppLogger info
INFO: Sorry, table already booked: main
Enter table id to book:
```

Fig 6 : Booking is not allowed for already booked table

The screenshot shows an IDE interface with a dark theme. The left sidebar displays a project structure under 'Project' with 'src' containing 'main' which has 'java' and 'com.zeta'. The 'com.zeta' folder contains 'concurrency', 'dao', 'logger', and 'model'. The right pane shows the code editor for 'Main.java'.

```
1 package com.zeta;
2
3 import com.zeta.concurrency.ChefWorker;
4 import com.zeta.concurrency.OrderDispatcher;
5 import com.zeta.concurrency.WaiterWorker;
6 import com.zeta.dao.FileTableDAO;
7 import com.zeta.dao.InMemoryOrderDAO;
8 import com.zeta.dao.TableDAO;
9 import com.zeta.service.KitchenService;
10 import com.zeta.service.OrderService;
11 import com.zeta.service.TableService;
12 import com.zeta.view.CustomerConsole;
13 import com.zeta.view.MenuCatalog;
14
15 import java.util.concurrent.ExecutorService;
16 import java.util.concurrent.Executors;
17
18 public class Main {
19     private static final int POOL_SIZE = 2; 3 usages
20
21     public static void main(String[] args) { 4 PiyushHMehtha
22         TableDAO tableDAO = new FileTableDAO();
23         OrderDAO orderDAO = new InMemoryOrderDAO();
```

The 'Run' tab is selected, showing a dropdown menu with options: 1. View Tables, 2. Book Table, 3. Simulation, 4. Exit. The number '10' is highlighted in blue. The bottom status bar shows the path 'rms > src > main > java > com > zeta > Main' and the time '11:38'.

Fig 7 : Booking is not allowed for Invalid table Id

The screenshot shows an IDE interface with a dark theme. The left sidebar displays a project structure under 'Project' with 'src' containing 'com.zeta'. The 'com.zeta' folder contains 'concurrency', 'dao', and 'logger'. The 'concurrency' folder contains 'ChefWorker', 'OrderDispatcher', and 'WaiterWorker'. The 'dao' folder contains 'FileTableDAO', 'InMemoryOrderDAO', 'OrderDAO', and 'TableDAO'. The right pane shows the code editor for 'Main.java'.

```
19 public class Main { 4 PiyushHMehtha
20     public static void main(String[] args) { 4 PiyushHMehtha
21         OrderDispatcher orderDispatcher = new OrderDispatcher();
22
23         ExecutorService waiterPool = Executors.newFixedThreadPool(POOL_SIZE);
24         ExecutorService chefPool = Executors.newFixedThreadPool(POOL_SIZE);
25
26         for(int threads=0;threads<POOL_SIZE;threads++) {
27             waiterPool.submit(new WaiterWorker(orderDispatcher, orderService, kitchenService));
28             chefPool.submit(new ChefWorker(orderService, kitchenService));
29         }
30     }
31
32     OrderDispatcher orderDispatcher = new OrderDispatcher();
33
34     ExecutorService waiterPool = Executors.newFixedThreadPool(POOL_SIZE);
35     ExecutorService chefPool = Executors.newFixedThreadPool(POOL_SIZE);
36
37 }
```

The 'Run' tab is selected, showing a dropdown menu with options: 1. View Tables, 2. Book Table, 3. Simulation, 4. Exit. The number '5' is highlighted in blue. The bottom status bar shows the path 'rms > src > main > java > com > zeta > Main' and the time '19:14'.

Fig 8 : Table Booking for valid input

```

public class Main {
    public static void main(String[] args) {
        PiyushHMehtha
        OrderDispatcher orderDispatcher = new OrderDispatcher();
        ExecutorService waiterPool = Executors.newFixedThreadPool(POOL_SIZE);
        ExecutorService chefPool = Executors.newFixedThreadPool(POOL_SIZE);

        for(int threads=0;threads<POOL_SIZE;threads++) {
            waiterPool.submit(new WaiterWorker(orderDispatcher, orderService, kitchenService));
            chefPool.submit(new ChefWorker(orderService, kitchenService));
        }
    }
}

Run Main
1. View Tables
2. Book Table
3. Simulation
4. Exit
a
Feb 18, 2026 11:00:52 PM com.zeta.logger.AppLogger warning
WARNING: Invalid input. Please enter a number.
2
Enter table id to book:
1
Feb 18, 2026 11:01:23 PM com.zeta.logger.AppLogger info
INFO: Sorry, table already booked: main
Enter table id to book:
5
Feb 18, 2026 11:01:38 PM com.zeta.logger.AppLogger info
INFO: Table booked successfully: main
1. Show menu
2. Add item
3. Show Cart
4. Place order
5. Checkout

```

Fig 9 : Customer Console for order Giving

```

package com.zeta;
import com.zeta.concurrency.ChefWorker;
import com.zeta.concurrency.OrderDispatcher;
import com.zeta.concurrency.WaiterWorker;
import com.zeta.dao.FileTableDAO;
import com.zeta.dao.InMemoryOrderDAO;
import com.zeta.dao.OrderDAO;
import com.zeta.dao.TableDAO;
import com.zeta.service.KitchenService;
import com.zeta.service.OrderService;
import com.zeta.service.TableService;
import com.zeta.view.CustomerConsole;
import com.zeta.view.MenuCatalog;

Run Main
3. Simulation
4. Exit
2
Enter table id to book:
1
Feb 19, 2026 11:27:42 AM com.zeta.logger.AppLogger info
INFO: Table booked successfully: main
1. Show menu
2. Add item
3. Show Cart
4. Place order
5. Checkout
2
Enter menu item id, or enter 0 to exit
1
Enter quantity:
c
Feb 19, 2026 11:27:57 AM com.zeta.logger.AppLogger warning
WARNING: Invalid input. Please enter a number.

```

Fig 10 : validation for quantity

The screenshot shows an IDE interface with the following details:

- Project View:** Shows the project structure with packages like service, view, target, and others.
- Main.java Content:**

```
1 package com.zeta;
2
3 import com.zeta.concurrency.ChefWorker;
4 import com.zeta.concurrency.OrderDispatcher;
5 import com.zeta.concurrency.WaiterWorker;
6 import com.zeta.dao.FileTableDAO;
7 import com.zeta.dao.InMemoryOrderDAO;
8 import com.zeta.dao.OrderDAO;
9 import com.zeta.dao.TableDAO;
10 import com.zeta.service.KitchenService;
11 import com.zeta.service.OrderService;
12 import com.zeta.service.TableService;
13 import com.zeta.view.CustomerConsole;
14 import com.zeta.view.MenuCatalog;
```
- Run Tab:**
 - Shows the command: `java Main`.
 - Output log:

```
Feb 19, 2026 11:33:51 AM com.zeta.logger.AppLogger info
INFO: Table booked successfully: main
1. Show menu
2. Add item
3. Show Cart
4. Place order
5. Checkout
6
```
 - Message: `WARNING: Invalid choice`
 - Output log:

```
Feb 19, 2026 11:33:57 AM com.zeta.logger.AppLogger warning
WARNING: Invalid choice
1. Show menu
2. Add item
3. Show Cart
4. Place order
5. Checkout
```
- Bottom Status Bar:** Shows the path `rms > src > main > java > com > zeta > Main`, file encoding `UTF-8`, and line numbers `19:14`.

Fig 11 : Validating invalid choice in menu

The screenshot shows an IDE interface with the following details:

- Project View:** Shows the project structure with packages like META-INF, src, and main.
- Main.java Content:**

```
1 package com.zeta;
2
3 import com.zeta.concurrency.ChefWorker;
4 import com.zeta.concurrency.OrderDispatcher;
5 import com.zeta.concurrency.WaiterWorker;
6 import com.zeta.dao.FileTableDAO;
7 import com.zeta.dao.InMemoryOrderDAO;
8 import com.zeta.dao.OrderDAO;
9 import com.zeta.dao.TableDAO;
10 import com.zeta.service.KitchenService;
11 import com.zeta.service.OrderService;
12 import com.zeta.service.TableService;
13 import com.zeta.view.CustomerConsole;
14 import com.zeta.view.MenuCatalog;
15
16 import java.util.concurrent.ExecutorService;
17 import java.util.concurrent.Executors;
18
19 public class Main { private static final int POOL_SIZE = 2; 3 usages
20     public static void main(String[] args) { 3 usages
21         TableDAO tableDAO = new FileTableDAO();
22         OrderDAO orderDAO = new InMemoryOrderDAO();
```
- Run Tab:**
 - Shows the command: `java Main`.
 - Output log:

```
Feb 19, 2026 7:59:26 AM com.zeta.logger.AppLogger info
INFO: Menu
MenuItem{id=1, name='Pizza', price=200.0, preparationTime=3}
MenuItem{id=2, name='Burger', price=150.0, preparationTime=2}
MenuItem{id=3, name='Coke', price=50.0, preparationTime=1}
MenuItem{id=4, name='Pasta', price=180.0, preparationTime=4}
```
 - Message: `INFO: Menu`
 - Output log:

```
1. Show menu
2. Add item
3. Show Cart
4. Place order
5. Checkout
```
- Bottom Status Bar:** Shows the path `rms > src > main > java > com > zeta > Main`, file encoding `UTF-8`, and line numbers `11:38`.

Fig 12 : Printing Menu

```

rms - Main.java
public class Main {
    public static void main(String[] args) {
        PiyushHMehra
        CustomerConsole customerConsole = new CustomerConsole(tableService, orderDispatcher, menuCatalog);
        customerConsole.start();
        waiterPool.shutdown();
        chefPool.shutdown();
    }
}

```

The screenshot shows an IDE interface with the Main.java file open. The code implements a simple command-line application for booking a table. It includes imports for various DAOs, logger, model, and service classes. The main method creates a CustomerConsole instance and starts it. It then attempts to shutdown waiter and chef pools.

The terminal output shows the application running and accepting user input. It successfully books a table (id 5) and handles invalid input by prompting the user to enter a number. It also handles an invalid choice by displaying a warning message.

Fig 13 : Verifying Invalid input in menu

```

rms - Main.java
package com.zeta;
import com.zeta.concurrency.ChefWorker;
import com.zeta.concurrency.OrderDispatcher;
import com.zeta.concurrency.WaiterWorker;
import com.zeta.dao.FileTableDAO;
import com.zeta.dao.InMemoryOrderDAO;
import com.zeta.dao.OrderDAO;
import com.zeta.dao.TableDAO;
import com.zeta.service.KitchenService;
import com.zeta.service.OrderService;
import com.zeta.service.TableService;
import com.zeta.view.CustomerConsole;
import com.zeta.view.MenuCatalog;

```

The screenshot shows an IDE interface with the Main.java file open. The code defines a Main class with a main method that imports various DAOs, workers, and services from the com.zeta package. The main method then imports the CustomerConsole and MenuCatalog classes.

The terminal output shows the application running and accepting user input. It successfully books a table (id 1) and handles an empty cart by displaying an info message stating "INFO: Cart is empty".

Fig 14 : show cart , cart is empty items not added to cart

The screenshot shows a Java IDE interface with the following details:

- Project View:** Shows the package structure under com.zeta, including concurrency, dao, logger, model, and service.
- Main.java Content:**

```
1 package com.zeta;
2
3 import com.zeta.concurrency.ChefWorker;
4 import com.zeta.concurrency.OrderDispatcher;
5 import com.zeta.concurrency.WaiterWorker;
6 import com.zeta.dao.FileTableDAO;
7 import com.zeta.dao.InMemoryOrderDAO;
8 import com.zeta.dao.OrderDAO;
9 import com.zeta.dao.TableDAO;
10 import com.zeta.service.KitchenService;
11 import com.zeta.service.OrderService;
12 import com.zeta.service.TableService;
13 import com.zeta.view.CustomerConsole;
14 import com.zeta.view.MenuCatalog;
```
- Run Output:**

```
3. Show Cart
4. Place order
5. Checkout
  3
    1. Show menu
    2. Add item
    3. Show Cart
    4. Place order
    5. Checkout
Feb 19, 2026 10:53:58 AM com.zeta.logger.AppLogger info
INFO: Cart is empty
  4
    1. Show menu
    2. Add item
    3. Show Cart
    4. Place order
    5. Checkout
Feb 19, 2026 10:54:09 AM com.zeta.logger.AppLogger info
INFO: Cart is empty
```
- Bottom Status Bar:** rms > src > main > java > com > zeta > Main | 11:38 LF UTF-8 4 spaces

Fig 15 : Place order , order will not get placed cart is empty

The screenshot shows a Java IDE interface with the following details:

- Project View:** Shows the package structure under com.zeta, including concurrency, dao, and model.
- Main.java Content:**

```
19 public class Main {  ↳ PiyushMehta
20     public static void main(String[] args) {  ↳ PiyushMehta
21         OrderDispatcher orderDispatcher = new OrderDispatcher();
22
23         ExecutorService waiterPool = Executors.newFixedThreadPool(POOL_SIZE);
24         ExecutorService chefPool = Executors.newFixedThreadPool(POOL_SIZE);
25
26         for(int threads=0;threads<POOL_SIZE;threads++) {
27             waiterPool.submit(new WaiterWorker(orderDispatcher, orderService, kitchenService));
28             chefPool.submit(new ChefWorker(orderService, kitchenService));
29     }
30 }
```
- Run Output:**

```
INFO: Item added: Coke, quantity: 2
  3
Enter quantity:
  2
  ↳ Enter menu item id, or enter 0 to exit
Feb 18, 2026 11:05:21 PM com.zeta.logger.AppLogger info
INFO: Item added: Coke, quantity: 2
  0
  1. Show menu
  2. Add item
  3. Show Cart
  4. Place order
  5. Checkout
  3
  ↳ MenuItem{id=3, name='Coke', price=50.0, preparationTime=1} 2
  ↳ MenuItem{id=3, name='Coke', price=50.0, preparationTime=1} 2
  1. Show menu
  2. Add item
  3. Show Cart
  4. Place order
  5. Checkout
```
- Bottom Status Bar:** rms > src > main > java > com > zeta > Main | 19:14 LF UTF-8 4 spaces

Fig 16 : Adding items along with Quantity to Cart

The screenshot shows an IDE interface with the following details:

- Project:** rms
- Current File:** Main.java
- Code View:** The Main.java code is displayed, showing the logic for creating thread pools and submitting tasks to them.
- Terminal View:** The terminal shows the following log output:

```
Feb 18, 2026 11:05:43 PM com.zeta.logger.AppLogger info
INFO: Waiter {pool-1-thread-1} picked order: 1
Feb 18, 2026 11:05:43 PM com.zeta.logger.AppLogger info
INFO: Customer placed order: 1
Feb 18, 2026 11:05:43 PM com.zeta.logger.AppLogger info
INFO: Chef {pool-2-thread-1} started preparing order: 1
Feb 18, 2026 11:05:43 PM com.zeta.logger.AppLogger info
INFO: Order submitted to kitchen: 1
Feb 18, 2026 11:05:43 PM com.zeta.logger.AppLogger info
INFO: Order placed successfully: 1
1. Show menu
2. Add item
3. Show Cart
4. Place order
5. Checkout
Feb 18, 2026 11:05:47 PM com.zeta.logger.AppLogger info
INFO: Order completed: 1
```
- Status Bar:** Shows the current time as 19:14, file encoding as LF, and code style as 4 spaces.

Fig 17 : The customer clicks on Place Order.

The screenshot shows an IDE interface with the following details:

- Project:** rms
- Current File:** Main.java
- Code View:** The Main.java code is displayed, showing the logic for booking a table.
- Terminal View:** The terminal shows the following log output:

```
Feb 19, 2026 11:13:55 AM com.zeta.logger.AppLogger info
INFO: Table booked successfully: main
1. View Tables
2. Book Table
3. Simulation
4. Exit
2
Enter table id to book:
1
Feb 19, 2026 11:14:00 AM com.zeta.logger.AppLogger info
INFO: Thanks for your visit
```
- Status Bar:** Shows the current time as 19:14, file encoding as LF, and code style as 4 spaces.

Fig 18 : Checkout

The screenshot shows an IDE interface with a dark theme. The top bar includes tabs for 'Current File' and various icons. The left sidebar shows a project structure with 'Project' expanded, containing 'model', 'service', 'view', 'test', and 'target'. A file named 'Main.java' is open in the center editor area. The code implements a main method that creates two threads to book tables simultaneously. The log window at the bottom shows the execution of the program and the resulting logger output.

```
import com.zeta.view.MenuCatalog;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class Main { ± PiyushHMehtha
    private static final int POOL_SIZE = 2; 3 usages
    public static void main(String[] args) { ± PiyushHMehtha
        TableDAO tableDAO = new FileTableDAO();
        OrderDAO orderDAO = new InMemoryOrderDAO();

        TableService tableService = new TableService(tableDAO);
        OrderService orderService = new OrderService(orderDAO);
        KitchenService kitchenService = new KitchenService();
```

Run Main

- 3. Simulation
- 4. Exit

1. Concurrent table booking(2 people trying to book same table)

2. Concurrent order giving

3. Exit simulation

1

1. Concurrent table booking(2 people trying to book same table)

2. Concurrent order giving

3. Exit simulation

Feb 19, 2026 11:17:27 AM com.zeta.logger.AppLogger info
INFO: Customer 1 trying to book table: 1

Feb 19, 2026 11:17:27 AM com.zeta.logger.AppLogger info
INFO: Customer 2 trying to book table: 1

Feb 19, 2026 11:17:27 AM com.zeta.logger.AppLogger info
INFO: Table booked successfully: pool-3-thread-1

Feb 19, 2026 11:17:27 AM com.zeta.logger.AppLogger info
INFO: Sorry, table already booked: pool-3-thread-2

Fig 19 : Simulating Concurrent table Booking on available table

This screenshot is nearly identical to Fig 19, showing the same Java code for concurrent table booking. However, the log output indicates that both threads attempt to book the same table simultaneously, leading to a conflict where one thread succeeds and the other fails.

```
import com.zeta.view.MenuCatalog;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class Main { ± PiyushHMehtha
    private static final int POOL_SIZE = 2; 3 usages
    public static void main(String[] args) { ± PiyushHMehtha
        TableDAO tableDAO = new FileTableDAO();
        OrderDAO orderDAO = new InMemoryOrderDAO();

        TableService tableService = new TableService(tableDAO);
        OrderService orderService = new OrderService(orderDAO);
        KitchenService kitchenService = new KitchenService();
```

Run Main

- 3. Simulation
- 4. Exit

1. Concurrent table booking(2 people trying to book same table)

2. Concurrent order giving

3. Exit simulation

1

1. Concurrent table booking(2 people trying to book same table)

2. Concurrent order giving

3. Exit simulation

Feb 19, 2026 11:18:40 AM com.zeta.logger.AppLogger info
INFO: Customer 2 trying to book table: 4

Feb 19, 2026 11:18:40 AM com.zeta.logger.AppLogger info
INFO: Customer 1 trying to book table: 4

Feb 19, 2026 11:18:40 AM com.zeta.logger.AppLogger info
INFO: Sorry, table already booked: pool-3-thread-2

Feb 19, 2026 11:18:40 AM com.zeta.logger.AppLogger info
INFO: Sorry, table already booked: pool-3-thread-1

Fig 20 : Simulating Concurrent table Booking on not available table

The screenshot shows a Java development environment with the following details:

- Project Structure:** The project structure includes a `model`, `service`, `view`, and `test` package. The `Main` class is located in the `com.zeta` package under `view`.
- Main.java Content:** The code defines a `Main` class with a static final variable `POOL_SIZE = 2`. The `main` method creates `TableDAO`, `OrderDAO`, `TableService`, `OrderService`, and `KitchenService` instances.
- Run Output:** The terminal window shows the application's log output. It starts with simulation instructions (1. Concurrent table booking, 2. Concurrent order giving, 3. Exit simulation) and then logs customer orders being placed and processed by the system.
- Log Output:**

```
Feb 19, 2026 11:15:38 AM com.zeta.logger.AppLogger info
INFO: Customer placed order: 101
Feb 19, 2026 11:15:38 AM com.zeta.logger.AppLogger info
INFO: Waiter {pool-1-thread-1} picked order: 101
Feb 19, 2026 11:15:38 AM com.zeta.logger.AppLogger info
INFO: Waiter {pool-1-thread-2} picked order: 102
Feb 19, 2026 11:15:38 AM com.zeta.logger.AppLogger info
INFO: Customer placed order: 102
Feb 19, 2026 11:15:38 AM com.zeta.logger.AppLogger info
INFO: Chef {pool-2-thread-2} started preparing order: 102
Feb 19, 2026 11:15:38 AM com.zeta.logger.AppLogger info
INFO: Order submitted to kitchen: 102
Feb 19, 2026 11:15:38 AM com.zeta.logger.AppLogger info
INFO: Chef {pool-2-thread-1} started preparing order: 101
Feb 19, 2026 11:15:38 AM com.zeta.logger.AppLogger info
```
- Bottom Status Bar:** Shows the current time (19:14), line separator (LF), encoding (UTF-8), and code space count (4 spaces).

Fig 21 : Simulating Concurrent orders

Code coverage:

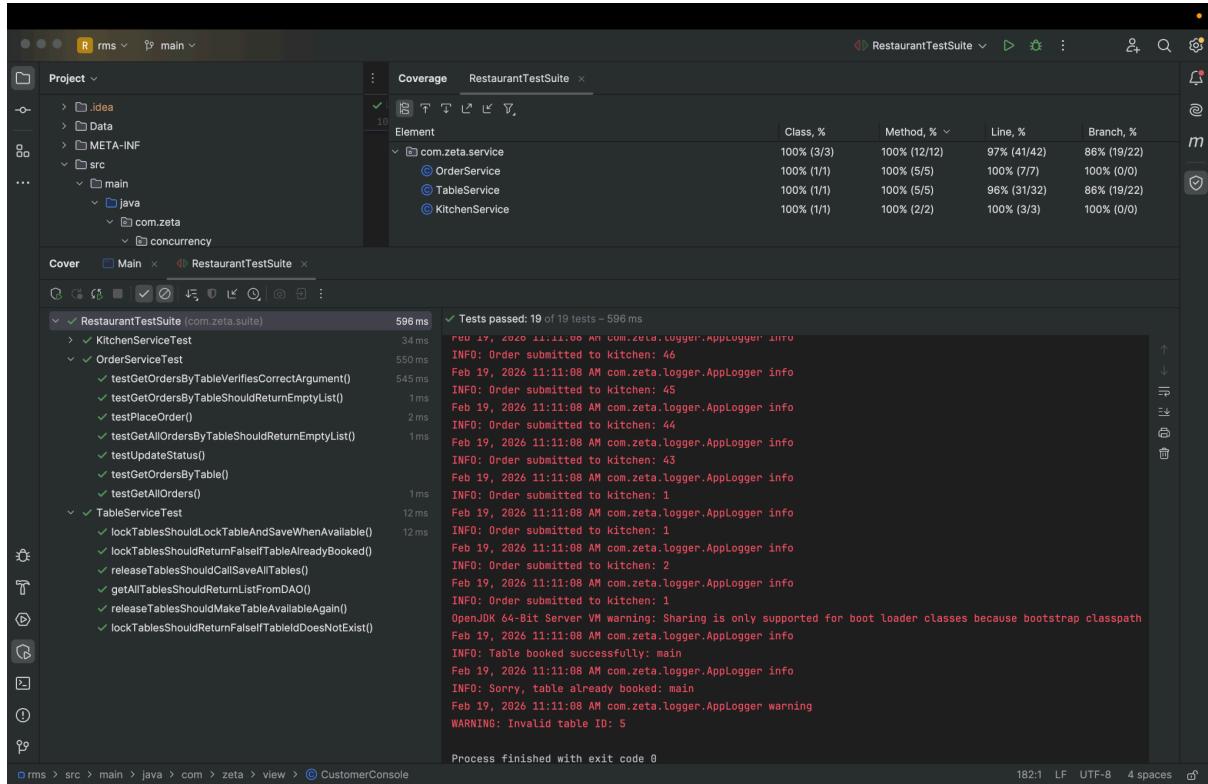


Fig 22