API Documentation

InsTax! is a standalone JavaFX taxi service application designed for local execution with no internet connectivity requirements. The system implements a complete taxi service solution with robust data management, sophisticated algorithms, and an intuitive user interface. All data persists locally through CSV files with ACID-compliant transaction handling. The app has Linux and Windows support and all dependencies are included in distribution.

model (package)

Contains the domain model classes.

Driver (public class)

The `Driver` class models a taxi driver within the InsTax! system, encapsulating core attributes and behaviors required for trip assignment and management. It represents an active driver entity that can be assigned to passenger trips, tracks real-time location/availability status, and provides geographical distance calculations.

id (private String)

- *Data Type*: Immutable unique identifier (`String`)
- *Purpose*: Uniquely identifies a driver across the system, ensuring consistent referencing in trip assignments, repository lookups, and equality checks.
- *Usage*: Serves as the primary key when storing/retrieving drivers from `DriverRepository` . Critical for avoiding collisions during concurrent trip assignments.

currentLocation (private Location)

- *Data Type*: Compositional `Location` object.
- *Purpose*: Stores real-time geographical coordinates for proximity-based trip assignment.
- *Usage*: Accessed by `TripService` during `requestTrip()` to compute nearest drivers via `calculateDistanceTo()` .

available (private boolean)

- *Data Type*: Mutable boolean state flag
- *Purpose*: Indicates driver's readiness to accept new trips (`true` = available, `false` = mid-trip/unavailable).
- *Usage*: Guards trip assignments in `TripService` ; flipped via `setAvailable()` when starting/completing trips. Critical for preventing over-assignment.

getter methods

Simple accessors returning field references.

setCurrentLocation (setter method)

- *Purpose*: Updates driver's geographical position for real-time tracking.
- *Behavior*: Replaces existing `currentLocation` reference.

calculateDistanceTo (public method)

- *Purpose*: Computes distance between the driver's current location and a target location.
- *Mechanics*: Delegates to `DistanceCalculator.calculateDistance()`, passing `this.currentLocation` and the parameterized `location`.
- *Code Flow*:
  1. Accepts `Location` parameter (non-null validated implicitly by utility).
  2. Invokes static calculation method with current driver coordinates.
  3. Returns primitive `double` representing straight-line distance in meters.
- *Error Handling*: Propagates `NullPointerException` if `currentLocation` is uninitialized.

equals (public method)

- *Purpose*: Implements identity-based equality using the driver's unique `id` field.
- *Algorithm*:
  1. Reference check: Returns `true` for same object instance.
  2. Type check: Rejects `null` or non-`Driver` objects via `getClass()` comparison.
  3. ID comparison: Compares `this.id` and `driver.id` using `String.equals()`.

toString (public method)

- *Purpose*: Generates debug-friendly string representation including critical state variables.
- *Format*: `"Driver[<id> - <name> | <vehicle> | <currentLocation> | Available: <available>]"`.
- *Technical Notes*:
  - Uses `String.format()` for structured output.
  - Implicitly calls `Location.toString()` via concatenation.

Location (public class)

The `Location` class represents immutable geographical coordinates within the InsTax! system, serving as the foundational unit for all distance calculations and proximity-based operations. It models two-dimensional Cartesian or geographic points (with `x` typically representing longitude and `y` latitude) using IEEE 754 double-precision floating-point values. The class enforces immutability through final fields and constructor initialization, Critical for calculating driver-passenger distances and fare estimations.

x (private final double)

- *Data Type*: Immutable 64-bit floating-point value (`double`)
- *Purpose*: Represents the horizontal coordinate in a Cartesian/geographic system (typically longitude in geographic contexts). Precision is critical for accurate distance calculations using the Haversine formula.
- *Usage*: Passed to `DistanceCalculator` ; accessed via `getX()` .

y (private final double)

- *Data Type*: Immutable 64-bit floating-point value (`double`)
- *Purpose*: Represents the vertical coordinate (typically latitude in geographic contexts).
- *Usage*: Combined with `x` in `equals()` for coordinate equality checks and in `DistanceCalculator` operations.

Location (constructor method)

- *Purpose*: Initializes immutable coordinate state with strict one-time assignment semantics.
- Absence of setters preserves immutability.

getter methods

- *Purpose*: Provide controlled read-only access to private coordinate fields.
- *Implementation*: Simple accessor methods returning primitive values.

equals (public method)

- *Purpose*: Implements value-based equality using IEEE 754-compliant coordinate comparisons.
- *Algorithm*:
  1. Reference equality short-circuit (`this == obj`).
  2. Type check via `getClass()` (rejects subclasses and `null`).
  3. Precision-aware comparison using `Double.compare()` for both coordinates.
- *Critical Details*:
  - Uses `Double.compare(a,b)` instead of `a == b` to handle NaN/±infinity consistently.
  - Strict class checking prevents subclass equality violations.
  - ~O(1) complexity with no heap allocations.

toString (public method)

- *Purpose*: Generates human-readable coordinate representation for debugging and serialization.
- *Format*: Uses `String.format("(%.1f, %.1f)", x, y)` to output rounded values with one decimal place.
- *Technical Behavior*:
  - Formatting rounds coordinates to 0.1 precision (e.g., 12.34 → 12.3).
  - Locale-neutral formatting ensures consistent parsing.

Passenger (public class)

The `Passenger` class models a taxi service user within the InsTax! system, encapsulating core identity attributes and authentication mechanisms required for trip requests and account management. It serves as the principal actor in trip initiation workflows, storing immutable credentials (`id` , `password`) and personal identifiers (`name`). The class enforces strict validation through `isValid()` and provides secure password verification via `authenticate()` , forming the foundation of user session management. Instances are created during registration, persisted via `UserRepository` , and referenced in `Trip` objects during lifecycle operations.

id (private final String)

- *Data Type*: Immutable unique identifier (`String`)
- *Purpose*: Primary key for passenger identification across repositories, trip assignments, and session tracking.
- *Usage*: Accessed by `UserRepository` ; referenced in `Trip` objects to link passengers to trips. Null/empty checks in `isValid()` preventing invalid states.

name (private final String)

- *Data Type*: Immutable display identifier (`String`)
- *Purpose*: Human-readable passenger identifier for UI display and reporting. Not used in business logic beyond validation.
- *Constraints*: Enforced non-empty in `isValid()` ;

password (private final String)

- *Data Type*: Immutable credential string (`String`)
- *Purpose*: Stores authentication secret in plaintext. Used exclusively for credential verification during login.

Passenger (constructor method)

- *Purpose*: Initializes immutable passenger state with one-time assignment of core attributes.

- *Lifecycle*: Typically instantiated during registration; objects reused throughout session via `Session.java`.

getter methods

Simple accessors returning field references.

authenticate (public method)

- *Purpose*: Verifies credential match without exposing raw password.
- *Algorithm*:
  1. Accepts `inputPassword` string (plaintext).
  2. Performs exact string comparison via `this.password.equals(inputPassword)`.
  3. Returns `true` only on exact character sequence match (case-sensitive).
     - *Usage*: Invoked by `UserService.login()` during authentication.

isValid (public method)

- *Purpose*: Domain integrity check for mandatory attribute constraints.
- *Validation Logic*:
  1. `id != null && !id.isEmpty()`: Rejects null/empty primary keys.
  2. `name != null && !name.isEmpty()`: Ensures display name exists.
  3. `password != null && !password.isEmpty()`: Prevents blank passwords.
- *Return Semantics*: `true` only if ALL checks pass; fails fast on first violation.

equals (public method)

- *Purpose*: Implements identity equality based solely on `id` field.
- *Mechanics*:
  1. Reference check (`this == obj`).
  2. Type/`null` check via `getClass()`.
  3. Delegates to `Objects.equals(id, passenger.id)` for null-safe comparison.

toString (public method)

- *Purpose*: Generates debug-safe representation excluding sensitive credentials.
- *Format*: `"Passenger[<id> - <name>]"` using `String.format()`.
- *Security*: Deliberately omits `password` field to prevent accidental logging exposure.

Trip (public class)

The `Trip` class is the central entity in the InsTax! system, modeling the complete lifecycle of a taxi journey from request to completion. It encapsulates immutable trip metadata (ID, passenger, driver, locations, fare) alongside mutable state transitions managed through status codes. The class implements strict finite state machine logic via action methods (`startTrip()`, `completeTrip()`, `cancelTrip()`), ensuring only valid status progressions. Business rules enforce that fare calculation occurs at creation (immutable fare) while status changes update driver availability in downstream workflows. Integrates with `TripRepository` for persistence and `TripService` for real-time coordination.

id (private final String)

- *Data Type*: Immutable unique identifier (`String`)
- *Purpose*: Primary key for trip identification across repositories and UI components. Ensures traceability.
- *Usage*: Set at construction via `TripService`; used in `equals()` for identity checks and repository lookups.

start/end (private final Locations)

- *Data Type*: Immutable geographical coordinates (`Location`)
- *Purpose*: Defines trip origin (`start`) and destination (`end`). Used for fare calculation, driver assignment.
- *Immutability*: Coordinates fixed at creation; prevents mid-trip destination changes.

fare (private final double)

- *Data Type*: Immutable 64-bit floating-point value (`double`)
- *Purpose*: Stores pre-calculated trip cost in monetary units. Immutability prevents post-creation tampering.
- *Calculation*: Set during construction by `FareCalculator` based on `start`/`end` locations.

status (private int)

- *Data Type*: Mutable state flag constrained to `REQUESTED` (0), `IN_PROGRESS` (1), `COMPLETED` (2), `CANCELLED` (3)
- *Purpose*: Tracks lifecycle progression via explicit state transitions. Guarded by action methods that enforce valid transitions.
- *Invariant*: Validated at construction and during state changes to prevent illegal values.

Trip (constructor method)

- *Purpose*: Initializes immutable trip attributes and validates initial status.
- *Validation*: Delegates to `validateStatus(status)` to ensure status falls within 0-3 range. Defaults to `REQUESTED` for invalid inputs.

getter methods

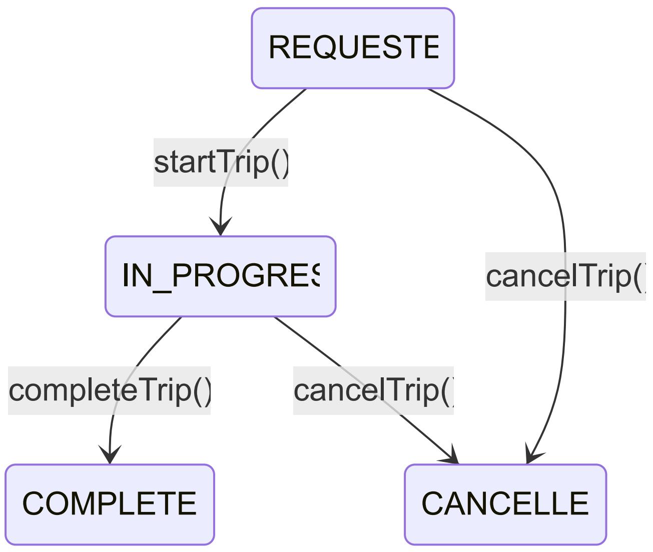
Simple accessors returning field references.

validateStatus (private method)

- *Purpose*: Enforces valid status range (0-3) at construction.
- *Algorithm*:
  1. Checks `if (status >= REQUESTED && status <= CANCELLED)`.
  2. Returns original status if valid.
  3. Returns `REQUESTED` as fallback for invalid inputs (e.g., -1, 4).
- *Design*: Private scope restricts validation to construction phase only.

startTrip/completeTrip/cancelTrip (public method)

- *State Machine Logic*:
  - `startTrip()`: Allows transition *only* from `REQUESTED` → `IN_PROGRESS` via `if (status == REQUESTED)`.
  - `completeTrip()`: Allows *only* `IN_PROGRESS` → `COMPLETED`.
  - `cancelTrip()`: Allows `REQUESTED` or `IN_PROGRESS` → `CANCELLED`.
- **Legal Transitions**:



- **Illegal Actions**:
  - Starting completed/cancelled trips
  - Completing non-active trips
  - Cancelling completed trips
- *Idempotency*: No-op if called in invalid states (e.g., `completeTrip()` on `CANCELLED`).
- *Downstream Effects*: Triggering driver availability updates in `TripService`.

validation methods

- `isActive()`: Returns `true` for `REQUESTED` / `IN_PROGRESS`.
- `canBeCancelled()`: Delegates to `isActive()` for cancellation eligibility.

- State-specific checkers (`isRequested()` , `isInProgress()` , etc.): Direct status comparisons.

getStatusString (public method)

- *Purpose*: Maps numeric status to human-readable string for UI/logging.
- *Mechanics*:
  1. `switch` statement over `status` field.
  2. Returns predefined strings for known states.
  3. Returns `"UNKNOWN"` for illegal values (though `validateStatus()` prevents this).

equals (public method)

operates similarly to the previous equals() methods.

toString (public method)

- *Format*: `"Trip[<id> | <start> --> <end> | Fare: <fare> | Status: <status>]"` .
- *Details*:
  - Uses `String.format()` with `%.2f` for fare to limit decimal places.
  - Implicitly calls `Location.toString()` and `getStatusString()` .
  - Excludes passenger/driver details for brevity.

repository (package)

Handles data persistence.

DriverRepository (public class)

The `DriverRepository` class implements a static in-memory data store for driver entities within the InsTax! system, providing initialization, persistence, and retrieval capabilities for driver objects. It simulates a database layer by maintaining a pre-configured collection of drivers with randomized locations, serving as the authoritative source for driver data used in trip assignment workflows. The class enforces singleton-like initialization via static blocks and provides immutable access patterns to prevent unintended state mutations. Critical for `TripService` operations that require nearest-driver lookups and availability checks.

drivers (private static final List<Driver>)

- *Data Type*: Static immutable `ArrayList<Driver>`
- *Purpose*: Centralized in-memory store for all driver entities. Populated once during `initializeDrivers()` and exposed via defensive copies.

random (private static final Random)

- *Data Type*: Static `java.util.Random` instance
- *Purpose*: Generates pseudorandom coordinates during driver initialization. Seeded once at class loading for deterministic testing.
- *Algorithm*: Uses `nextDouble()` for uniform distribution across coordinate ranges.

initialized (private static boolean)

- *Data Type*: Static boolean flag
- *Purpose*: Idempotency guard preventing duplicate initialization. Set to `true` after first successful driver population.
- *Invariant*: Checked in `initializeDrivers()` to ensure one-time setup.

DRIVER_NAMES/VEHICLES (private static final String[])

- *Data Type*: Static string arrays
- *Purpose*: Hardcoded datasets for driver attributes (name and vehicle type). Index-correlated during object creation.

initializeDrivers (public static method)

- *Purpose*: Populates the driver repository with preconfigured entities using randomized locations.
- *Algorithm*:
  1. Checks `initialized` flag to skip redundant execution.
  2. Iterates over `DRIVER_NAMES` array using index `i`.
  3. Constructs driver ID as `"DRIV-"+(i+1)` (e.g., `DRIV-1`).
  4. Generates random location via `generateRandomLocation()`.
  5. Instantiates `Driver` with `available=true` and adds to `drivers` list.
  6. Sets `initialized=true` to block re-initialization.

generateRandomLocation (private static method)

- *Purpose*: Creates randomized geographic coordinates within a 1000x1000 unit grid.
- *Coordinate Generation*:
  1. `x = 1 + random.nextDouble() * 999` → Range: [1, 1000)
  2. `y = 1 + random.nextDouble() * 999` → Same range
  3. Returns new `Location(x, y)`

getInitialDrivers (public static method)

- *Purpose*: Provides immutable snapshot of current drivers via defensive copy.
- *Mechanics*: Returns `new ArrayList<>(drivers)` to prevent external modification of internal state.

FileDataHandler (public class)

The `FileDataHandler` class implements the persistence layer for InsTax! using CSV-based file storage, providing serialization/deserialization for core domain objects (passengers, drivers, trips). The class employs a columnar CSV format with strict schema enforcement, leveraging functional interfaces (`FileWriterHandler` , `DataHandler`) for polymorphic data processing. Critical workflows include startup data loading (`loadInitialData`), shutdown persistence (`saveAllData`), and driver state updates (`saveDriver`). File operations include automatic file creation, error handling, and relational resolution between entities during deserialization.

PASSENGERS_FILE/DRIVERS_FILE/TRIPS_FILE (private static final String)

- *Data Type*: Static final `String` constants
- *Purpose*: Define canonical paths (`resources/*.csv`) for persistent storage.
- *Schema Implications*:
  - Passengers: `id,name,password`
  - Drivers: `id,name,vehicle,locX,locY,available`
  - Trips: `id,passengerId,driverId,startX,startY,endX,endY,fare,status`

saveAllData/loadInitialData (public static method)

- *Orchestration*:
  - `saveAllData()` : Sequentially invokes `saveUserData()` , `saveDriverData()` , `saveTripData()` .
  - `loadInitialData()` : Loads drivers first (prerequisite for trip assignments), then passengers and trips.

saveUserData/saveDriverData/saveTripData (public static method)

- *Pattern*: Delegates to generic `saveToFile()` with lambda-defined CSV formatting.
- *Data Sourcing*: Pulls live data directly from service layers (`UserService` , `TripService`).

saveDriver (public static method)

- *Purpose*: Updates single driver record without full repository rewrite.
- *Algorithm*:
  1. Fetches all drivers via `UserService.getAllDrivers()` .
  2. Finds target driver by ID using linear search (`O(n)`).
  3. Replaces old driver reference in cloned list.
  4. Rewrites entire driver file with updated list.

loadPassengers/loadDrivers/loadTrips (private static method)

- *Workflow*:
  1. Delegates to `loadFromFile()` with type-specific parsing lambdas.
  2. Splits CSV lines into columns.
  3. Validates column count before processing.

FileWriterHandler/DataHandler (private interface)

- *Purpose*: Enable strategy pattern for polymorphic CSV handling without code duplication.
- *Mechanics*:
  - `FileWriterHandler`: Accepts `PrintWriter` for writing entity-specific CSV lines.
  - `DataHandler`: Processes split `String[]` during file reading for object reconstruction.

saveToFile/loadFromFile (private static method)

- **`saveToFile()` Workflow**:
  1. Checks file existence; creates via `createFile()` if missing.
  2. Opens `PrintWriter` with try-with-resources.
  3. Invokes handler lambda for data writing.
  4. Catches `IOException` with error logging.
- **`loadFromFile()` Workflow**:
  1. Returns early if file doesn't exist.
  2. Uses `BufferedReader` for line-by-line processing.
  3. Splits lines using `line.split(",")` (vulnerable to commas in data).
  4. Delegates parsing to handler lambda per line.

createFile (private static method)

- *Purpose*: Recursively creates missing directories and files.
- *Filesystem Operations*:
  1. `file.getParentFile().mkdirs()`: Creates parent directories.
  2. `file.createNewFile()`: Atomic file creation.

TripRepository (public class)

The `TripRepository` class serves as the in-memory data access layer for trip entities within the InsTax! system. It maintains a static list of `Trip` objects with access patterns, offering query capabilities by ID, passenger, and activity status. The class enforces business logic during state transitions via `updateTripStatus()` and provides ID generation using UUIDs. Tightly integrated with `TripService` for core operations and `FileDataHandler` for persistence, it acts as the system of record for all trip-related data.

trips (private static final List<Trip>)

- *Data Type*: Static `ArrayList<Trip>`
- *Purpose*: Volatile in-memory store for all trip entities. Initialized empty at class loading and populated via `addTrip()` during runtime.

addTrip (public static method)

- *Purpose*: Safely appends valid trips to the repository.
- *Validation*:
  1. Rejects `null` trips via `trip != null`.
  2. Rejects trips with null IDs via `trip.getId() != null`.
- *Performance*: O(1) `ArrayList.add()` amortized constant time.

findTripById (public static method)

- *Purpose*: Retrieves trip by exact ID match using sequential search.
- *Algorithm*:
  1. Streams `trips` collection.
  2. Applies `filter(t -> t.getId().equals(id))` for exact match.
  3. Returns first match via `findFirst()` or `null` if absent.

getTripsByPassengerId (public static method)

- *Purpose*: Returns all trips associated with a passenger ID.
- *Filter Logic*: Uses `t.getPassenger().getId().equals(passengerId)` to resolve passenger references.
- *Return Semantics*: Returns mutable copy via `Collectors.toList()`; original repository remains unmodifiable.
- *Null Safety*: Assumes `t.getPassenger()` never null (enforced at trip creation).

getActiveTrips (public static method)

- *Purpose*: Filters trips where `isActive()` returns true (status = `REQUESTED` or `IN_PROGRESS`).
- *Method Reference*: Leverages `Trip::isActive` predicate for stream filtering.
- *UI Integration*: Critical for real-time dashboards showing ongoing trip.

getAllTrips (public static method)

- *Purpose*: Provides defensive snapshot of entire trip repository.
- *Immutability*: Returns `new ArrayList<>(trips)` to prevent external modification.

updateTripStatus (public static method)

- *Purpose*: Executes state transitions while enforcing business rules.
- *Workflow*:
  1. Looks up trip via `findTripById(tripId)`.
  2. For valid trips, switches on `status`:
     - `IN_PROGRESS`: Invokes `trip.startTrip()` (only if current=REQUESTED).
     - `COMPLETED`: Invokes `trip.completeTrip()` (only if current=IN_PROGRESS).
     - `CANCELLED`: Invokes `trip.cancelTrip()` (if REQUESTED or IN_PROGRESS).
     - `REQUESTED`: No-op (cannot revert).
- *State Encapsulation*: Delegates transition logic to `Trip` methods, preserving invariants.

generateId (public static method)

- *Purpose*: Creates collision-resistant trip IDs using UUIDs.
- *Format*: `"TRIP-" + UUID.substring(0,8)` (e.g., `TRIP-3f7a82e1`).
- *Uniqueness Probability*: 16^8 = ~4.3 billion combinations; adequate for single-instance use.
- *Performance*: Relies on cryptographic-strength `UUID.randomUUID()`.

UserRepository (public class)

The `UserRepository` class implements a centralized in-memory data access layer for user entities (passengers and drivers) within the InsTax! system, providing query capabilities, and state management. It maintains two distinct static collections (`passengers` and `drivers`) with access patterns, enabling ID-based lookups, availability filtering, and real-time state updates. The class enforces basic validation during entity addition, generates collision-resistant IDs using UUIDs, and serves as the authoritative source for user data consumed by `UserService` and `TripService`. Critical for authentication workflows, driver assignment logic, and location tracking.

passengers (private static final List<Passenger>)

- *Data Type*: Static `ArrayList<Passenger>`
- *Purpose*: Volatile in-memory store for all passenger entities. Populated via `addPassenger()` during registration/login and accessed through query methods.
- *Lifecycle*: Survives until JVM shutdown; All data is saved onto CSV files via shutdown.

drivers (private static final List<Driver>)

- *Data Type*: Static `ArrayList<Driver>`
- *Purpose*: Centralized repository for driver entities supporting trip assignment workflows. Stores real-time availability and location states.
- *State Management*: Updated via `updateDriverAvailability()` and `updateDriverLocation()` during trip lifecycle events.

- *Integration*: `TripService` accesses this collection for nearest-driver searches using `findAvailableDrivers()`.

addPassenger (public static method)

- *Purpose*: Safely appends validated passengers to the repository.
- *Validation*:
  1. Rejects `null` passengers via `passenger != null`.
  2. Rejects passengers with null IDs via `passenger.getId() != null`.
- *Data Integrity*: Does not check for duplicate usernames; enforced externally by `UserService`.

findPassengerById (public static method)

- *Algorithm*:
  1. Streams `passengers` collection.
  2. Applies `filter(p -> p.getId().equals(id))` for exact ID match.
  3. Returns first match via `findFirst()` or `null` if absent.
- *Performance*: O(n) linear search; optimized for small datasets.

findPassengerByUsername (public static method)

- *Purpose*: Locates passengers by exact name match (case-sensitive).
- *Usage*: Critical for login workflows in `UserService.authenticate()`.

getAllPassengers (public static method)

- *Purpose*: Provides defensive snapshot of entire passenger repository.
- *Immutability*: Returns `new ArrayList<>(trips)` to prevent external modification.

addDriver (public static method)

- *Validation*: Identical to `addPassenger()` - rejects null drivers or those with null IDs.
- *Initialization*: Typically populated during startup via `FileDataHandler.loadDrivers()`.

findDriverById (public static method)

- *Mechanics*: Mirrors `findPassengerById()` but queries `drivers` collection.
- *Trip Integration*: Used during trip loading to resolve driver references from stored IDs.

findAvailableDrivers (public static method)

- *Purpose*: Filters drivers with `available=true` for trip assignment.
- *Performance*: O(n) full scan; critical path in `TripService.requestTrip()`.

getAllDrivers (public static method)

- *Purpose*: Provides defensive snapshot of entire driver repository.
- *Immutability*: Returns `new ArrayList<>(trips)` to prevent external modification.

updateDriverAvailability (public static method)

- *Workflow*:
  1. Locates driver via `findDriverById(driverId)`.
  2. If found, invokes `driver.setAvailable(available)`.

updateDriverLocation (public static method)

- *Purpose*: Updates real-time coordinates for driver movement simulation.
- *Validation*: Rejects null locations;

generatePassengerId (public static method)

- *Format*: `"PASS-" + 8-character UUID substring` (e.g., `PASS-4d3f8a2b`).
- *Uniqueness*: Relies on `UUID.randomUUID()` for cryptographic entropy.

generateDriverId (public static method)

- *Format*: `"DRIV-" + 8-character UUID substring` (e.g., `DRIV-7c5e9f01`).
- *Collision Safety*: Theoretical collision probability ≈ 1 in 4.3 billion.

service (package)

Contains business logic.

TripService (public class)

The `TripService` class orchestrates the end-to-end trip lifecycle within the InsTax! system, providing transactional workflows for trip management, driver assignment, and state transitions. It acts as the primary business logic layer between repositories (`TripRepository`, `UserRepository`), domain models (`Trip`, `Driver`), and utilities (`FareCalculator`). Key responsibilities include:

- **Trip Creation**: Generating new trips with fare calculation and nearest-driver assignment.
- **State Management**: Enforcing valid status transitions (start/end/cancel) with driver availability updates.
- **Query Handling**: Providing active trip lookups and historical trip retrieval for passengers. The class implements static methods for stateless operation, with strict validation of business rules and atomic updates across multiple repositories to maintain system consistency.

requestTrip (public static method)

- *Purpose*: Creates and assigns a new trip to the nearest available driver.
- *Workflow*:
  1. **Driver Assignment**: Invokes `findNearestAvailableDriver(start)` to locate closest available driver via linear search. Returns `null` if none found.
  2. **Fare Calculation**: Delegates to `FareCalculator.calculateFare(start, end)` for pricing.
  3. **Trip Creation**: Generates ID via `TripRepository.generateId()`, constructs `Trip` object with `REQUESTED` status.
  4. **Driver Update**: Sets driver's availability to `false` and location to trip start.
  5. **Persistence**: Saves trip via `TripRepository.addTrip(trip)`.

startTrip (public static method)

- *Purpose*: Transitions trip from `REQUESTED` to `IN_PROGRESS` state.
- *Validation*:
  1. Fetches active trip via `getActiveTrip(passenger)` (returns error if none).
  2. Verifies current status is `REQUESTED` via `activeTrip.isRequested()`.
- *Action*: Invokes `activeTrip.startTrip()` for state transition.
- *Idempotency*: Returns error message ("Trip cannot be started...") for invalid states; `null` on success.

endTrip (public static method)

- *Purpose*: Completes active trips and releases drivers.
- *Mechanics*:
  1. Validates active trip exists and is `IN_PROGRESS`.
  2. Updates driver:
     - Sets location to trip end via `driver.setCurrentLocation(activeTrip.getEnd())`.
     - Marks available via `driver.setAvailable(true)`.
  3. Transitions trip to `COMPLETED` via `activeTrip.completeTrip()`.
- *Data Flow*: Driver location updated to destination for subsequent trip assignments.

cancelTrip (public static method)

- *Purpose*: Cancels active/cancellable trips and releases drivers.
- *Rules*:
  1. Checks `activeTrip.canBeCancelled()` (true for `REQUESTED` / `IN_PROGRESS`).
  2. Marks driver available without location change.
  3. Invokes `activeTrip.cancelTrip()` for state transition.
- *UI Integration*: Returns user-friendly error messages for invalid states.

getActiveTrip (public static method)

- *Purpose*: Retrieves a passenger's single active trip (`REQUESTED` or `IN_PROGRESS`).
- *Algorithm*:
  1. Fetches all passenger trips via `TripRepository.getTripsByPassengerId()` .
  2. Linear search for first trip where `trip.isActive() == true` .
- *Assumption*: Passengers limited to one active trip; returns first match ignoring multiples.

getTripHistory (public static method)

- *Delegation*: Directly returns `TripRepository.getTripsByPassengerId(passenger.getId())` .
- *Semantics*: Includes all historical trips regardless of status.

findNearestAvailableDriver (private static method)

- *Purpose*: Locates closest available driver to a given location.
- *Optimization*:
  1. Gets `availableDrivers` list via `UserRepository.findAvailableDrivers()` .
  2. Iterates with distance tracking:

```
double minDistance = Double.MAX_VALUE;
for (Driver driver : availableDrivers) {
    double distance = driver.calculateDistanceTo(location);
    if (distance < minDistance) {
        minDistance = distance;
        nearestDriver = driver;
    }
}
```

  - *Performance*: O(n) where n = available drivers; efficient for small fleets.

getAllTrips (public static method)

Returns `List<Trip>` using `TripRepository.getAllTrips()` .

addTrip (public static method)

Adds the trip using `TripRepository.addTrip(trip)` if `trip != null` .

UserService (public class)

The `UserService` class implements the core authentication and user management logic for the InsTax! system, serving as the intermediary between repository operations (`UserRepository`) and domain models (`Passenger` , `Driver`). It provides transactional workflows for passenger registration, credential validation, and entity persistence while enforcing business rules such as username uniqueness and input validation. The class exposes static methods for stateless operation, acting as the primary entry point for GUI controllers (`loginScene.fxml`, `registerScene.fxml`) to perform user-related operations. Critical integrations include `FileDataHandler` for data persistence and `Session.java` for maintaining active user state across JavaFX scenes.

registerPassenger (public static method)

- *Purpose*: Creates new passenger accounts with comprehensive validation.
- *Validation Workflow*:
  1. `username == null || username.isEmpty()` → Returns "Username cannot be empty".
  2. `password == null || password.isEmpty()` → Returns "Password cannot be empty".
  3. `UserRepository.findPassengerByUsername(username) != null` → Returns "Username already exists".
- *Account Creation*:
  1. Generates ID via `UserRepository.generatePassengerId()`.
  2. Instantiates `Passenger` object with password.
  3. Persists via `UserRepository.addPassenger()`.
- *Return Semantics*: Returns `null` on success; error message string on failure.

login (public static method)

- *Purpose*: Authenticates passengers using credentials.
- *Authentication Flow*:
  1. Fetches passenger by username via `UserRepository.findPassengerByUsername()`.
  2. Returns `null` if username not found.
  3. Verifies password match via `passenger.authenticate(password)`.
  4. Returns `Passenger` object on success; `null` on credential mismatch.

addPassenger (public static method)

- *Purpose*: Persists pre-validated passenger objects.
- *Validation*: Checks `passenger != null` and `passenger.isValid()` (enforces non-empty ID/name/password).
- *Usage*: Primarily invoked by `FileDataHandler` during system startup to load persisted passengers.

getAllPassengers (public static method)

- *Delegation*: Directly returns `UserRepository.getAllPassengers()`.
- *Behavior*: Provides defensive copy of all passengers; O(n) space complexity.

- *Persistence Integration*: Invoked by `FileDataHandler.saveUserData()` for CSV serialization.

getAllDrivers (public static method)

Similar to `getAllPasssenerѕ()`.

addDriver (public static method)

- *Purpose*: Persists driver entities.
- *Integration*: Used by `FileDataHandler.loadDrivers()` during initialization.

findPassengerById (public static method)

- *Purpose*: Locates passengers by primary key.
- *Mechanics*: Delegates to `UserRepository.findPassengerById(id)`.
- *Error Handling*: Returns `null` for missing IDs.

findDriverById (public static method)

- *Purpose*: Resolves driver references by ID.
- *Usage*: Critical for `TripService` when reconstructing trip objects during deserialization.
- *Edge Case*: Returns `null` for invalid IDs.

util (package)

Utility classes.

DistanceCalculator (public class)

The `DistanceCalculator` class implements geographical distance computations for the InsTax! system, providing static utility methods to calculate Cartesian distances between `Location` objects and validate distance results. It serves as a foundational component for driver-passenger proximity calculations, and fare estimations. The class assumes a Euclidean coordinate system (x/y as planar coordinates rather than lat/long), making it suitable for simulated environments. All methods are stateless and thread-safe, with deterministic outputs for given inputs.

calculateDistance (public static method)

- *Purpose*: Computes straight-line Euclidean distance between two points in 2D space.
- *Algorithm*:
  1. Null Check: Returns `-1` immediately if either location is null (error sentinel value).
  2. Delta Calculation:
     - `deltaX = loc2.getX() - loc1.getX()`

        - `deltaY = loc2.getY() - loc1.getY()`
    3. Euclidean Formula: `Math.sqrt(deltaX * deltaX + deltaY * deltaY)`
- *Mathematical Foundation*: Implements Pythagorean theorem for Cartesian coordinates:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- *Error Handling*: Negative return value (`-1`) signals invalid inputs; callers must validate via `isValidDistance()` .
- *Coordinate System*: Units are arbitrary (meters/km based on `Location` scale); consistent with `FareCalculator` 's expectations.

isValidDistance (public static method)

- *Purpose*: Verifies if a distance value is physically plausible.
- *Validation Logic*: Returns `true` if `distance >= 0` , rejecting negative values and error codes (`-1`).

FareCalculator (public class)

The `FareCalculator` class implements the pricing logic for InsTax! trips, providing configurable fare computations based on distance traveled. It combines a fixed base fare with a variable distance-based rate, using Euclidean distance calculations from `DistanceCalculator` as input. The class features static configuration parameters (`baseFare` , `perUnitRate`) with setters, and enforces monetary rounding to two decimal places for financial accuracy. Critical for trip creation workflows in `TripService.requestTrip()` , where it determines trip costs before persistence. The implementation assumes a planar coordinate system where distance units directly correspond to fare calculations.

baseFare (private static double)

- *Data Type*: Static 64-bit floating point (`double`)
- *Purpose*: Fixed initial cost applied to all trips regardless of distance. Represents minimum fare for very short journeys.
- *Mutability*: Modified via `setBaseFare()` with validation (`newBaseFare > 0`). Default value: $5.00.

perUnitRate (private static double)

- *Data Type*: Static 64-bit floating point (`double`)
- *Purpose*: Cost multiplier applied per distance unit (e.g., $0.50perunit).Combinedwith\backslash baseFare\backslash fortotal$: $ \text{Fare} = \text{baseFare} + (\text{distance} \times \text{perUnitRate}) $$
- *Unit Assumption*: Distance units must be consistent with `DistanceCalculator` outputs (e.g., km or simulated units).

- *Validation*: `setPerUnitRate()` rejects values ≤0.

calculateFare - Location input (public static method)

- *Purpose*: Computes trip fare from geographical coordinates.
- *Workflow*:
  1. Null Check: Returns `baseFare` if either location is null (safe fallback).
  2. Distance Calculation: Delegates to `DistanceCalculator.calculateDistance()`.
  3. Fare Computation: Passes distance to overloaded `calculateFare(distance)`.

calculateFare - double input (public static method)

- *Purpose*: Core fare computation from pre-calculated distance.
- *Algorithm*:
  1. Sanitization: Clamps negative distances to 0 (`if (distance < 0) distance = 0`).
  2. Linear Pricing: Computes `baseFare + (distance * perUnitRate)`.
  3. Rounding: Passes result to `roundToTwoDecimals()` for financial precision.
- *Example*: For distance=10, base=5, rate=0.5 → 5 + (10×0.5) = 10 → $10.00

roundToTwoDecimals (private static method)

- *Purpose*: Rounds monetary values to nearest cent.
- *Mechanics*:
  1. Scales: `value * 100.0` (e.g., 10.567 → 1056.7)
  2. Rounds: `Math.round()` → 1057
  3. Descales: `/ 100.0` → 10.57
- *Precision Notes*:
  - Mitigates floating-point errors in financial sums.
  - Handles HALF_UP rounding via `Math.round()`.

setter methods

- *Validation*: Rejects values ≤0 (no-op for invalid inputs).
- *Usage*: Typically invoked from admin configuration panels.

getter methods

- *Purpose*: Provide read access for UI display and fare validation.
- *Return Semantics*: Returns current configuration values without rounding.

gui (package)

Contains the JavaFX controllers, FXML files, and the main application class.

App (public class)

The `App` class serves as the primary entry point for the InsTax! JavaFX application, orchestrating GUI initialization, scene configuration, and bootstrap workflows. It extends `javafx.application.Application`, leveraging JavaFX's lifecycle management to handle application startup via the `start()` method. Key responsibilities include:

- **Scene Construction**: Loading FXML-defined UI layouts (`openScene.fxml`) and applying CSS styling (`style.css`).
- **Window Configuration**: Setting fixed window dimensions (500x700px), background color (RGB 95,201,255), and non-resizable behavior.
- **Resource Management**: Loading application icon (`appIcon.png`) and initializing controller logic via `initializeSystem()` .
  The class bridges the JVM launch process with JavaFX runtime, ensuring proper resource cleanup and event handling while enforcing a consistent visual identity across all scenes.

start (public method)

- *Purpose*: JavaFX lifecycle method for primary stage initialization. Called after `Application.launch()` .
- *Workflow*:
  1. **FXML Loading**:

     ```
     FXMLLoader loader = new
     FXMLLoader(getClass().getResource("openScene.fxml"));
     Parent root = loader.load();
     ```

     - Loads `openScene.fxml` as root node using `FXMLLoader` .
     - Implicitly instantiates associated `Controller` class.
  2. **Scene Configuration**:

     ```
     Scene scene = new Scene(root, 500, 700, Color.rgb(95, 201, 255));
     scene.getStylesheets().add(getClass().getResource("style.css").toExternalF
     orm());
     ```

     - Sets fixed dimensions (500x700px) with light blue background.
     - Attaches `style.css` for centralized UI styling.
  3. **Stage Setup**:

     ```
     stage.setResizable(false);
     stage.setTitle("InsTax! _ Your Instant Taxi App");
     stage.getIcons().add(new
     ```

```
Image(getClass().getResourceAsStream("/gui/img/appIcon.png")));
stage.setScene(scene);
```

- Enforces non-resizable window for consistent layout.
- Sets title and taskbar icon (`appIcon.png`).

4. **Controller Initialization**:

```
Controller controller = loader.getController();
controller.initializeSystem();
```

- Retrieves FXML-injected controller instance.
- Invokes `initializeSystem()` for data loading/initialization.

5. **Display**: `stage.show()` renders the configured UI.

main (public static method)

- *Purpose*: JVM entry point that delegates to JavaFX application framework.
- *Mechanics*:
  1. Invokes `Application.launch()` to bootstrap JavaFX runtime.
  2. Triggers JavaFX lifecycle: `init()` → `start(Stage)` → idle → `stop()` .
- *Threading*: Executes on JavaFX Application Thread for thread-safe UI operations.

Controller (public class)

The `Controller` class serves as the central command hub for the InsTax! JavaFX application, orchestrating all user interactions, business logic execution, and UI state management across multiple FXML scenes. It implements the following core responsibilities:

- **Scene Management**: Handles transitions between 7+ FXML scenes with dynamic data binding
- **User Workflows**: Manages authentication (login/registration), trip lifecycle (request/start/cancel/end), and historical data display
- **Data Synchronization**: Bridges UI components with backend services (`TripService` , `UserService`) and repositories
- **Session Handling**: Maintains user context via the `Session` singleton
- **Persistence Integration**: Coordinates with `FileDataHandler` for atomic data saves during state changes
- **Error Handling**: Validates inputs and provides user feedback through dedicated UI elements

The class leverages JavaFX's dependency injection via `@FXML` to bind UI components, implements complex multi-scene state synchronization, and employs background threading for data initialization. With 50+ methods handling diverse application workflows, it forms the nervous system connecting the JavaFX frontend to the domain model.

stage (private Stage)

- *Type*: `javafx.stage.Stage`
- *Purpose*: Primary application window reference updated during scene transitions
- *Lifecycle*: Injected via FXML; reassigned in every `switchTo*()` method using `Button.getSource()` event tracing

scene (private Scene)

- *Type*: `javafx.scene.Scene`
- *Purpose*: Container for scene graphs during transitions; reconfigured for each FXML load

root (private Parent)

- *Type*: `javafx.scene.Parent`
- *Purpose*: Root node of the current scene graph; reloaded for every scene switch

nameLable (private Text)

Displays "Hi" + username in main menu scenes

tripStatusAndPrice (private Text)

Shows status + fare ("Trip Status: Requested ($12.50)")

tripCords (private Text)

Displays formatted locations ("Origin: (x,y), Destination: (x,y)")

driverName/driverVehicle (private Text)

Driver details from assigned trip

tripHistoryList (private TextArea)

The textArea showing user's trip history in appropriate scene

loginUsername (private TextField)/loginPassword (private PasswordField)

Credential inputs

loginError (private Text)

Validation feedback for login

registerUsername (private TextField)/registerPassword (private PasswordField)

Registration inputs

registerError (private Text)

Registration validation feedback

startX/startY/endX/endY (private TextField)

Coordinate inputs for trip creation

requestError (private Text)

Trip validation feedback

setter methods

Standard setter methods for variables that need loading upon scene switches

initializeSystem (public method)

- *Purpose*: Bootstraps application data and configures shutdown handler
- *Threading*:

```
Thread initThread = new Thread(() -> {
    // Load persisted data
    FileDataHandler.loadInitialData();
    // Initialize drivers if empty
    if (UserRepository.getAllDrivers().isEmpty()) {
        DriverRepository.initializeDrivers();
        for (Driver driver : DriverRepository.getInitialDrivers()) {
            UserRepository.addDriver(driver);
        }
        FileDataHandler.saveDriverData();
    }
});
initThread.start();
```

- *Shutdown Hook*:

```
Platform.runLater(() -> {
    Stage stage = (Stage) rootNode.getScene().getWindow();
    stage.setOnCloseRequest(event -> FileDataHandler.saveAllData());
});
```

- *Concurrency*: Data loading in background thread to prevent UI freeze

Scene Switching methods

- *Generic Scene Load*:

```
root = FXMLLoader.load(getClass().getResource("scene.fxml"));
stage = (Stage)((Node)eventSource).getScene().getWindow();
scene = new Scene(root);
stage.setScene(scene);
stage.show();
```

- *Stateful Scene Load (Needed for data loading in switches)*:

```
FXMLLoader loader = new
FXMLLoader(getClass().getResource("sceneAndContent.fxml"));
root = loader.load();
Controller controller = loader.getController();
// Inject session data
controller.setNameLabel("Hello " + Session.getCurrentUser().getName());
controller.setTripStatusAndPrice(...);
// Configure scene
stage.setScene(new Scene(root));
```

  - *Dynamic Content Loading (Trip History)*:

```
for (Trip trip : TripService.getTripHistory(Session.getCurrentUser())) {
    controller.appendTripHistoryListText(
        "["+trip.getId()+"] " +
        trip.getStart() + " -> " + trip.getEnd() +
        " ("+trip.getFare()+"$) - " + trip.getStatusString() + "\n"
    );
}
```

login (public method)

- *Workflow*:

```
1. Authenticate via `UserService.login()`
2. On success:
    - Set user in `Session` singleton
    - Determine active trip state using `TripService.getActiveTrip()`
    - Navigate to appropriate main menu scene based on trip status
3. On failure: Update `loginError` text
```

register (public method)

- *Validation*: Delegates to `UserService.registerPassenger()`
- *Error Handling*: Directly binds service errors to `registerError` text

requestTrip (public method)

- *Coordinate Validation*:

  ```
  startX.getText().matches("-?\\d+(\\.\\d+)?") // Regex for integers/decimals
  //same for the rest of coordinates
  ```

- *Trip Creation*:

  ```
  Location start = new Location(Double.parseDouble(startX.getText()), ...);
  //same for Location end...
  Trip trip = TripService.requestTrip(Session.getCurrentUser(), start, end);
  ```

- *Persistence & Error Handling*:
  - On success: Save trip/driver data → Switch to requested scene
  - On failure: Show "No available drivers" in `requestError`

startTrip/cancelTrip/endTrip (public method)

- *Pattern*:
  1. Execute state change via `TripService`
  2. Save trip + driver data atomically
  3. Transition to appropriate scene
- *Driver State Management*:

  ```
  Driver driver = Session.getActiveTrip().getDriver();
  // State-specific logic
  FileDataHandler.saveDriver(driver); // Persist driver availability change
  ```

backFromTripHistory (public method)

- *Intelligent Routing*: Checks `Session.getActiveTrip()` status to determine:
  - Return to no-trip menu (null trip)

    - Return to requested-trip menu (`isRequested`)
    - Return to in-progress menu (`isInProgress`)

logout (public method)

Clears session → Returns to open scene

exit (public method)

Executes graceful shutdown sequence:

```
stage.close();     // Close JavaFX window
Platform.exit();   // Terminate JavaFX thread
System.exit(0);    // Kill JVM
//automatically saves data duo to shutdown hook in initializeSystem()
```

Session (public class)

The `Session` class implements a singleton pattern to manage global application state across JavaFX scenes, serving as the central data repository for the authenticated user and their active trip context. It provides thread-safe access to the current passenger's identity and trip state through static access points, enabling consistent state propagation between disconnected controllers and scenes. The class eliminates the need for parameter passing between controllers, reduces redundant service calls, and maintains strict encapsulation of user session data throughout the application lifecycle.

session (private static Session)

- *Data Type*: Static self-referential instance
- *Purpose*: Holds the singleton session instance.
- *Initialization*: Created on first call to `getSession()` ; remains until JVM shutdown.
- *Thread Safety*: Relies on JavaFX single-threaded GUI model (no synchronization needed).

currentUser (private Passenger)

- *Data Type*: `model.Passenger` reference
- *Purpose*: Stores the authenticated passenger entity for the current session.
- *State Semantics*:
    - `null` → No user logged in
    - `Passenger` instance → Active authenticated user
- *Lifecycle*: Set during login; cleared during logout.

getSession (public static method)

- *Purpose*: Singleton accessor.
- *Algorithm*:

  ```
  if (session == null) session = new Session();
  return session;
  ```

- *Thread Guarantee*: JavaFX Application Thread exclusivity prevents race conditions.
- *Invocation Pattern*: Used in all controllers via `Session.getSession().getCurrentUser()`.

getCurrentUser (public method)

- *Purpose*: Provides read access to the authenticated passenger.
- *Return Semantics*:
  - Returns `Passenger` object when user logged in
  - Returns `null` when session inactive
- *Usage*: Primary access point for user-dependent operations (e.g., trip requests).

getActiveTrip (public method)

- *Purpose*: Retrieves the current user's active trip through abstraction.
- *Delegation*:

  ```
  TripService.getActiveTrip(currentUser);
  ```

- *Design Rationale*:
  - Avoids storing redundant trip state
  - Ensures real-time trip status from authoritative source
- *Return Semantics*:
  - `Trip` object if active trip exists
    - `null` if no active trip

setCurrentUser (public method)

- *Purpose*: Mutator for session authentication state.
- *Lifecycle Transitions*:
  - Login: `setCurrentUser(authenticatedPassenger)`
  - Logout: `setCurrentUser(null)`
- *Implicit State Clearance*: Setting `currentUser=null` invalidates active trip context.

Main (public class)

The `Main` class implements a comprehensive terminal-based testing interface for the InsTax! system, providing an alternative entry point that bypasses the JavaFX GUI to validate core business logic, repository operations, and service workflows. It features a console menu system with 10+ interactive commands covering the full taxi service lifecycle, including user registration, trip management, and real-time status queries. The class simulates application startup/shutdown sequences with proper persistence hooks, implements robust input validation, and maintains session state identical to the GUI version. Designed exclusively for backend validation, it enables rapid testing of system functionality without GUI dependencies.