




APRIL 30, 2025

# TIMELINEXPRESS SYSTEM DESIGN DOCUMENT

VERSION 1.0

## ALPHA TEAM

Alexandra Lawler  
Dana Tryon  
Gitti Esmailzada  
Walter Ulicki



# TABLE OF CONTENTS

<b>Introduction .....</b>	<b>2</b>
<b>Design Level Class Diagram .....</b>	<b>3</b>
Pseudocode for Event Class .....	4
Pseudocode for TimeLine Class .....	5
Pseudocode for User Class.....	8
Pseudocode for DataBaseManager Class.....	10
Pseudocode for SearchEngine Class.....	11
Pseudocode for class Visualization: .....	12
Pseudocode for UserHandler Class .....	13
<b>Statechart Diagrams.....</b>	<b>16</b>
Timeline class .....	16
Event class .....	17
Visualization class.....	18
User class .....	18
DatabaseManager class .....	19
SearchEngine class .....	19
UserHandler Class.....	19
<b>Sequence Diagrams .....</b>	<b>20</b>
Use Case 1: Login to TimelineXpress .....	20
Use Case 2: Create Timeline .....	21
Use Case 3: Edit Timeline.....	21
Use Case 4: View ALL Saved Timelines .....	22
Use Case 5: Display a Timeline .....	23
Use Case 6: Create Event.....	24
Use Case 7: Edit Event.....	25

# INTRODUCTION

---

## Description of the Application

The Timeline app is a user-friendly, interactive application designed to help people explore historical events in an engaging and accessible way. The app lets users view, compare, and interact with historical timelines, offering a visual and customizable approach to learning about the past.

Events are stored in an integrated database, allowing users to search for key historical moments, create and manage entries, and explore timelines based on themes, periods, or event types.

## Purpose of This Design Document

The purpose of this document is to define the design details, and functionality, of the Timeline app. It serves as a shared reference point for the development team, designers, and stakeholders through the implementation phase of the project.

By outlining the app's core features, technical components, and performance expectations, this document ensures that everyone involved in the project has a mutual understanding of what the application will do, and how it will work.

## What This Document Contains

This design document includes the following sections:

- Introduction, including a high-level overview of the Timeline app.
- A design level class diagram, including pseudocode for each class.
- Statechart diagrams for each class
- Design level sequence diagrams for each use case identified in the System Requirements Document

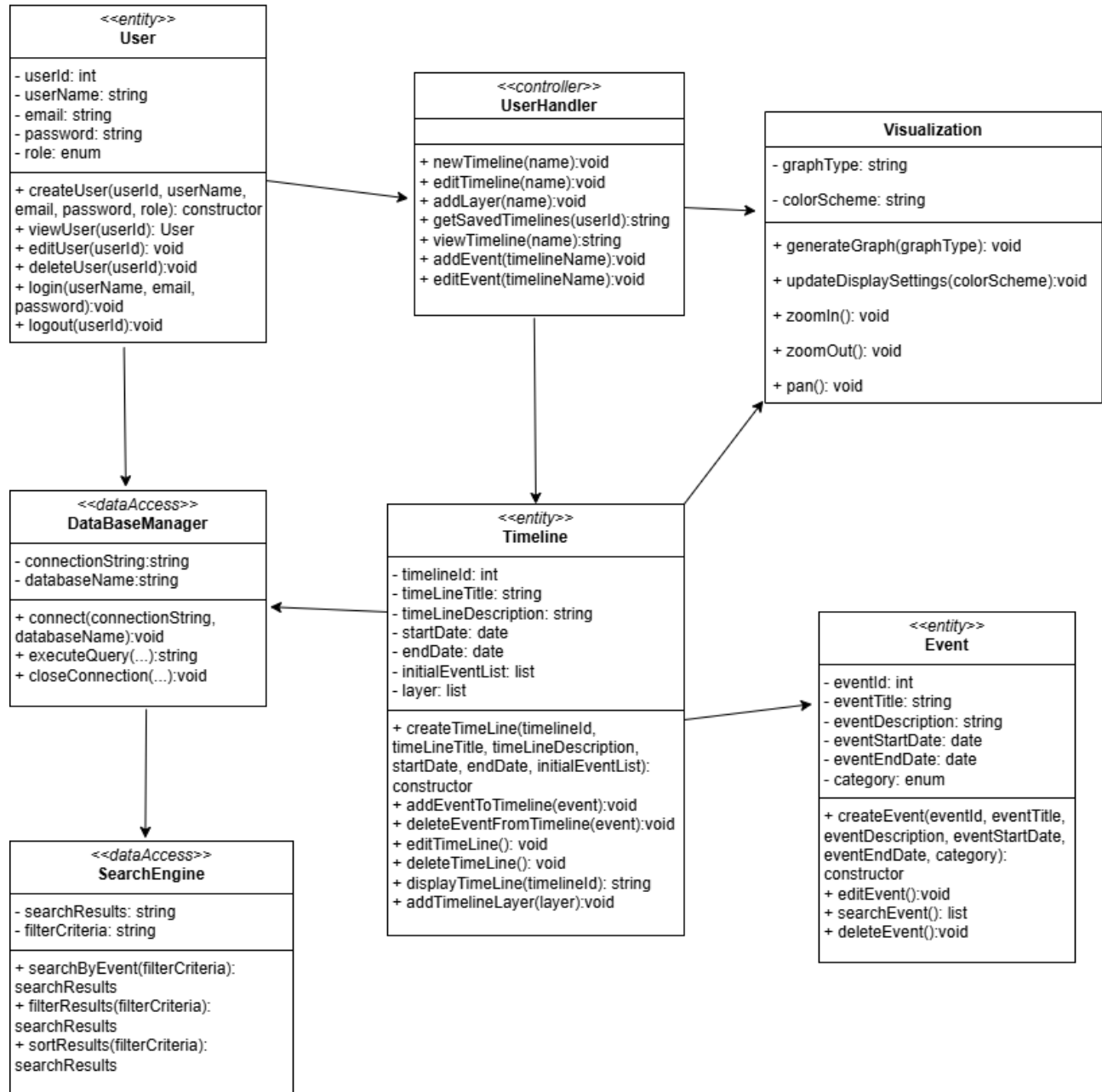
## Reference Documents

TimelineXpress Project Plan, February 17, 2025.

TimelineXpress System Requirements Document, March 26, 2025

# DESIGN LEVEL CLASS DIAGRAM

**TimeLineXpress  
Design Level  
Class Diagram**



---

## PSEUDOCODE FOR EVENT CLASS

class **Event**:

attributes:

eventID: int  
eventTitle: string  
eventDescription: string  
eventStartDate: date  
eventEndDate: date  
category: string

methods:

# Constructor to create a new event

**createEvent**(eventID: int, eventTitle: string, eventDescription: string, eventStartDate: date, eventEndDate: date, category: string):

    this.eventID = eventID  
    this.eventTitle = eventTitle  
    this.eventDescription = eventDescription  
    this.eventStartDate = eventStartDate  
    this.eventEndDate = eventEndDate  
    this.category = category  
    print("Event created successfully.")

# Edit an event based on eventID

**editEvent**(eventID: int):

    if this.eventID == eventID:  
        input("Enter new eventTitle: ") -> this.eventTitle  
        input("Enter new eventDescription: ") -> this.eventDescription  
        input("Enter new eventStartDate: ") -> this.eventStartDate  
        input("Enter new eventEndDate: ") -> this.eventEndDate  
        input("Enter new category: ") -> this.category  
        print("Event updated successfully.")

    else:

        print("Event ID not found.")

```

# Search for an event based on eventID
searchEvent(eventID: int) -> list:
    if this.eventID == eventID:
        return [this.eventID, this.eventTitle, this.eventDescription, this.eventStartDate,
this.eventEndDate, this.category]
    else:
        print("Event ID not found.")
        return []

```

```

# Delete an event based on eventID
deleteEvent(eventID: int):
    if this.eventID == eventID:
        this.eventID = null
        this.eventTitle = null
        this.eventDescription = null
        this.eventStartDate = null
        this.eventEndDate = null
        this.category = null
        print("Event deleted successfully.")
    else:
        print("Event ID not found.")

```

#### Explanation of the Methods:

- **createEvent**: Initializes a new event with all the attributes provided.
- **editEvent**: Allows modification of an event's details if the eventID matches.
- **searchEvent**: Searches for an event using eventID and returns the details in a list format.
- **deleteEvent**: Clears all the attributes of the event if the eventID matches.

#### PSEUDOCODE FOR TIMELINE CLASS

class **TimeLine**:

```

attributes:
    timelineId: int
    timeLineTitle: string
    timeLineDescription: string
    startDate: date

```

```
endDate: date
initialEventList: list # List of events in some type of data structure
layer: list # Additional timeline layers in some type of data structure
```

---

methods:

# Constructor to create a new timeline

```
createTimeLine(timelinelid: int, timeLineTitle: string, timeLineDescription: string, startDate:
date, endDate: date, initialEventList: list):
    this.timelinelid = timelinelid
    this.timeLineTitle = timeLineTitle
    this.timeLineDescription = timeLineDescription
    this.startDate = startDate
    this.endDate = endDate
    this.initialEventList = initialEventList
    this.layer = []
    print("Timeline created successfully.")
```

# Add an event to the timeline

```
addEventToTimeline(event: Event):
    this.initialEventList.append(event)
    print("Event added to timeline.")
```

# Delete an event from the timeline

```
deleteEventFromTimeline(event: Event):
    if event in this.initialEventList:
        this.initialEventList.remove(event)
        print("Event deleted from timeline.")
    else:
        print("Event not found in timeline.")
```

# Edit a timeline's details

```
editTimeLine(timelinelid: int):
    if this.timelinelid == timelinelid:
        input("Enter new timeLineTitle: ") -> this.timeLineTitle
        input("Enter new timeLineDescription: ") -> this.timeLineDescription
        input("Enter new startDate: ") -> this.startDate
```

```

        input("Enter new endDate: ") -> this.endDate
        print("Timeline updated successfully.")
    else:
        print("Timeline ID not found.")

```

# Delete a timeline

```

deleteTimeLine(timelineld: int):
    if this.timelineld == timelineld:
        this.timelineld = null
        this.timeLineTitle = null
        this.timeLineDescription = null
        this.startDate = null
        this.endDate = null
        this.initialEventList = null
        this.layer = null
        print("Timeline deleted successfully.")
    else:
        print("Timeline ID not found.")

```

# Display timeline details

```

displayTimeLine(timelineld: int) -> string:
    if this.timelineld == timelineld:
        return f"Timeline: {this.timeLineTitle}\nDescription: {this.timeLineDescription}\nStart Date:
{this.startDate}\nEnd Date: {this.endDate}\nEvents: {len(this.initialEventList)}\nLayers:
{len(this.layer)}"
    else:
        print("Timeline ID not found.")
        return ""

```

# Add a layer to the timeline

```

addTimelinelayer(layer: string):
    this.layer.append(layer)
    print("Layer added to timeline.")

```

Explanation of the Methods:

- **createTimeLine**: Initializes a timeline with attributes like ID, title, description, and event list.
- **addEventToTimeline**: Appends a new event to the initialEventList.
- **deleteEventFromTimeline**: Removes an event from the list if it exists.



- editTimeLine: Updates timeline details such as title, description, and dates.
- deleteTimeLine: Deletes a timeline by clearing all its attributes.
- displayTimeLine: Returns a string with timeline details, including the number of events and layers.
- addTimelinelayer: Appends a new layer to the timeline's layer list.

## PSEUDOCODE FOR USER CLASS

class **User**:

attributes:

    userId: int  
    userName: string  
    email: string  
    password: string  
    role: enum # Example values: Admin, Regular, Guest

methods:

# Create a new user

**createUser**(userId: int, userName: string, email: string, password: string, role: enum):  
    this.userId = userId  
    this.userName = userName  
    this.email = email  
    this.password = password  
    this.role = role  
    print("User created successfully.")

# View user details

**viewUser**(userId: int) -> string:  
    if this.userId == userId:  
        return f"User ID: {this.userId}\nName: {this.userName}\nEmail: {this.email}\nRole:  
{this.role}"  
    else:  
        print("User ID not found.")  
        return ""

# Edit user details

```

editUser(userId: int):
    if this.userId == userId:
        input("Enter new userName: ") -> this.userName
        input("Enter new email: ") -> this.email
        input("Enter new password: ") -> this.password
        input("Enter new role: ") -> this.role
        print("User details updated successfully.")
    else:
        print("User ID not found.")

```

# Delete a user

```

deleteUser(userId: int):
    if this.userId == userId:
        this.userId = null
        this.userName = null
        this.email = null
        this.password = null
        this.role = null
        print("User deleted successfully.")
    else:
        print("User ID not found.")

```

# Log in a user

```

login(userName: string, email: string, password: string):
    if this.userName == userName and this.email == email and this.password == password:
        print(f"Login successful for user: {this.userName}")
    else:
        print("Invalid credentials.")

```

# Log out a user

```

logout(userId: int):
    if this.userId == userId:
        print(f"User {this.userName} logged out successfully.")
    else:
        print("User ID not found.")

```

Explanation of the Methods:

- **createUser**: Initializes a new user by setting all the attributes.
- **viewUser**: Returns a formatted string with user details if the userId matches.
- **editUser**: Updates user information such as name, email, password, and role when the userId matches.
- **deleteUser**: Clears the attributes of a user if the userId matches.
- **login**: Authenticates the user based on their username, email, and password.
- **logout**: Logs the user out by validating the userId.

---

## PSEUDOCODE FOR DATABASEMANAGER CLASS

class **DataBaseManager**:

attributes:

connectionString: string  
databaseName: string  
isConnected: boolean # To track the connection status (optional)

methods:

# Connect to the database using connectionString and databaseName

**connect**(connectionString: string, databaseName: string):

this.connectionString = connectionString  
this.databaseName = databaseName  
print(f"Connecting to database: {this.databaseName} using {this.connectionString}...")  
# Simulate successful connection  
this.isConnected = true  
print("Connection successful.")

# Execute a query on the connected database

**executeQuery**() -> string:

if this.isConnected:  
    print("Executing query on the database...")  
    # Simulate query execution  
    return "Query executed successfully."  
else:  
    print("No active database connection.")  
    return "Query execution failed."

# Close the database connection

**closeConnection**():

if this.isConnected:  
    print(f"Closing connection to database: {this.databaseName}...")  
    this.isConnected = false  
    print("Connection closed.")  
else:  
    print("No active connection to close.")

Explanation of the Methods:

- **connect:** Initializes the database connection using `connectionString` and `databaseName` and updates the connection status to `isConnected` to be true.
- **executeQuery:** Executes a database query if the connection is active (`isConnected` is true). Returns a success message if the query is executed or a failure message if the connection is not active.
- **closeConnection:** Closes the database connection by setting `isConnected` to false and ensures resources are released.

## PSEUDOCODE FOR SEARCHENGINE CLASS

class **SearchEngine**:

attributes:

`searchResults: string` # Stores the results of the search as a string (can be modified for a list in real implementation)

`filterCriteria: string` # Stores the criteria to filter or sort results

methods:

    # Search for events based on filter criteria

**searchByEvent**(`filterCriteria: string`) -> string:

`this.filterCriteria = filterCriteria`

`print(f"Searching for events using criteria: {this.filterCriteria}...")`

        # Simulate search logic

`this.searchResults = f"Results found for criteria: {this.filterCriteria}"`

`return this.searchResults`

    # Filter the search results based on additional criteria

**filterResults**(`filterCriteria: string`) -> string:

`this.filterCriteria = filterCriteria`

`print(f"Filtering results using criteria: {this.filterCriteria}...")`

        # Simulate filtering logic

`this.searchResults = f"Filtered results for criteria: {this.filterCriteria}"`

`return this.searchResults`

    # Sort the search results based on specified criteria

**sortResults**(`filterCriteria: string`) -> string:

`this.filterCriteria = filterCriteria`

`print(f"Sorting results using criteria: {this.filterCriteria}...")`

        # Simulate sorting logic

`this.searchResults = f"Sorted results by criteria: {this.filterCriteria}"`

`return this.searchResults`

---

### Explanation of the Methods:

- **searchByEvent:** Accepts a `filterCriteria` as input. Simulates a database search using the provided criteria and updates `searchResults` with matching entries.
- **filterResults:** Applies additional filtering on the current `searchResults` based on new `filterCriteria`. Updates `searchResults` to reflect the filtered data.
- **sortResults:** Sorts the current `searchResults` based on the specified `filterCriteria`. Updates `searchResults` to reflect the sorted data.

Usage Flow: The user would typically invoke `searchByEvent` to get a list of results based on initial criteria. The results can then be further refined using `filterResults`. Finally, the results can be sorted as needed using `sortResults`.

### PSEUDOCODE FOR CLASS VISUALIZATION:

#### Class **Visualization**:

##### attributes:

`graphType: string` # Type of graph to represent the timeline (e.g., Bar, Gantt, Line)  
`colorScheme: string` # Color scheme for the display (e.g., Light, Dark, Custom)

##### methods:

# Set the type of graph for visualization

**generateGraphType**(`graphType: string`):

`this.graphType = graphType`

`print(f"Graph type set to: {this.graphType}")`

# Simulate generating the graph

`print(f"Generating a {this.graphType} graph...")`

# Update the display settings with a specific color scheme

**updateDisplaySettings**(`colorScheme: string`):

`this.colorScheme = colorScheme`

`print(f"Color scheme updated to: {this.colorScheme}")`

# Simulate applying the color scheme

`print(f"Applying {this.colorScheme} color scheme to the visualization.")`

# Zoom into the timeline view

**zoomIn**():

```

print("Zooming in on the timeline...")
# Simulate the zoom-in effect
print("Timeline zoomed in.")

# Zoom out of the timeline view
zoomOut():
    print("Zooming out from the timeline...")
    # Simulate the zoom-out effect
    print("Timeline zoomed out.")

```

```

# Pan across the timeline
pan():
    print("Panning across the timeline...")
    # Simulate the panning effect
    print("Timeline panned.")

```

#### Explanation of the Methods:

- **generateGraphType:** Sets the type of graph for timeline visualization and simulates the process of generating it. The graph types could include bar charts, Gantt charts, line graphs, etc.
- **updateDisplaySettings:** Updates the color scheme used for the timeline display (e.g., light mode, dark mode) and applies the new settings.
- **zoomIn:** Simulates zooming into the timeline to provide a closer view of events or layers.
- **zoomOut:** Simulates zooming out to show a broader view of the timeline and its events.
- **pan:** Allows the user to move across the timeline horizontally, providing navigation functionality.

#### PSEUDOCODE FOR USERHANDLER CLASS

class **UserHandler:**

    methods:

```

# Create a new timeline with a given name
newTimeline(name: string):
    print(f"Creating a new timeline: {name}")
    # creating a new timeline
    timeline = TimeLine(name)
    print(f"Timeline '{name}' created successfully.")

```

---

# Edit an existing timeline by name

```
editTimeline(name: string):  
    print(f"Editing timeline: {name}")  
    # Simulate editing timeline logic  
    input("Enter new title for timeline: ") -> newTitle  
    timeline = getTimelineByName(name) # function to retrieve the timeline  
    if timeline:  
        timeline.timeLineTitle = newTitle  
        print(f"Timeline '{name}' updated to '{newTitle}'.")  
    else:  
        print("Timeline not found.")
```

# Add a layer to a timeline by name

```
addLayer(name: string):  
    print(f"Adding a layer to timeline: {name}")  
    timeline = getTimelineByName(name) # function to retrieve the timeline  
    if timeline:  
        input("Enter layer name: ") -> layerName  
        timeline.addTimelinelayer(layerName)  
        print(f"Layer '{layerName}' added to timeline '{name}'.")  
    else:  
        print("Timeline not found.")
```

# Retrieve saved timelines for a specific user

```
getSavedTimelines(userId: int) -> string:  
    print(f"Retrieving saved timelines for user ID: {userId}")  
    # retrieval of timelines  
    timelines = getTimelinesByUserId(userId)  
    if timelines:  
        return f"Saved Timelines for User {userId}: {timelines}"  
    else:  
        return "No saved timelines found."
```

# View the details of a timeline by name

```
viewTimeline(name: string) -> string:  
    print(f"Viewing timeline: {name}")  
    timeline = getTimelineByName(name) # function to retrieve the timeline  
    if timeline:  
        return f"Timeline Details: {timeline.displayTimeLine(timeline.timelineId)}"  
    else:  
        return "Timeline not found."
```

```

# Add an event to a specific timeline by name
addEvent(timelineName: string):
    print(f"Adding an event to timeline: {timelineName}")
    timeline = getTimelineByName(timelineName) # function to retrieve the timeline
    if timeline:
        input("Enter event details (ID, title, description, startDate, endDate, category): ") ->
eventDetails
        newEvent = Event(eventDetails) # event creation
        timeline.addEventToTimeline(newEvent)
        print(f"Event added to timeline '{timelineName}'.")
    else:
        print("Timeline not found.")

# Edit an event within a specific timeline by name
editEvent(timelineName: string):
    print(f"Editing an event in timeline: {timelineName}")
    timeline = getTimelineByName(timelineName) # function to retrieve the timeline
    if timeline:
        input("Enter event ID to edit: ") -> eventId
        event = getEventById(timeline, eventId) # function to retrieve the event
        if event:
            event.editEvent(event.eventID)
            print(f"Event with ID {eventId} in timeline '{timelineName}' updated.")
        else:
            print("Event not found in the timeline.")
    else:
        print("Timeline not found.")

```

#### Explanation of the Methods:

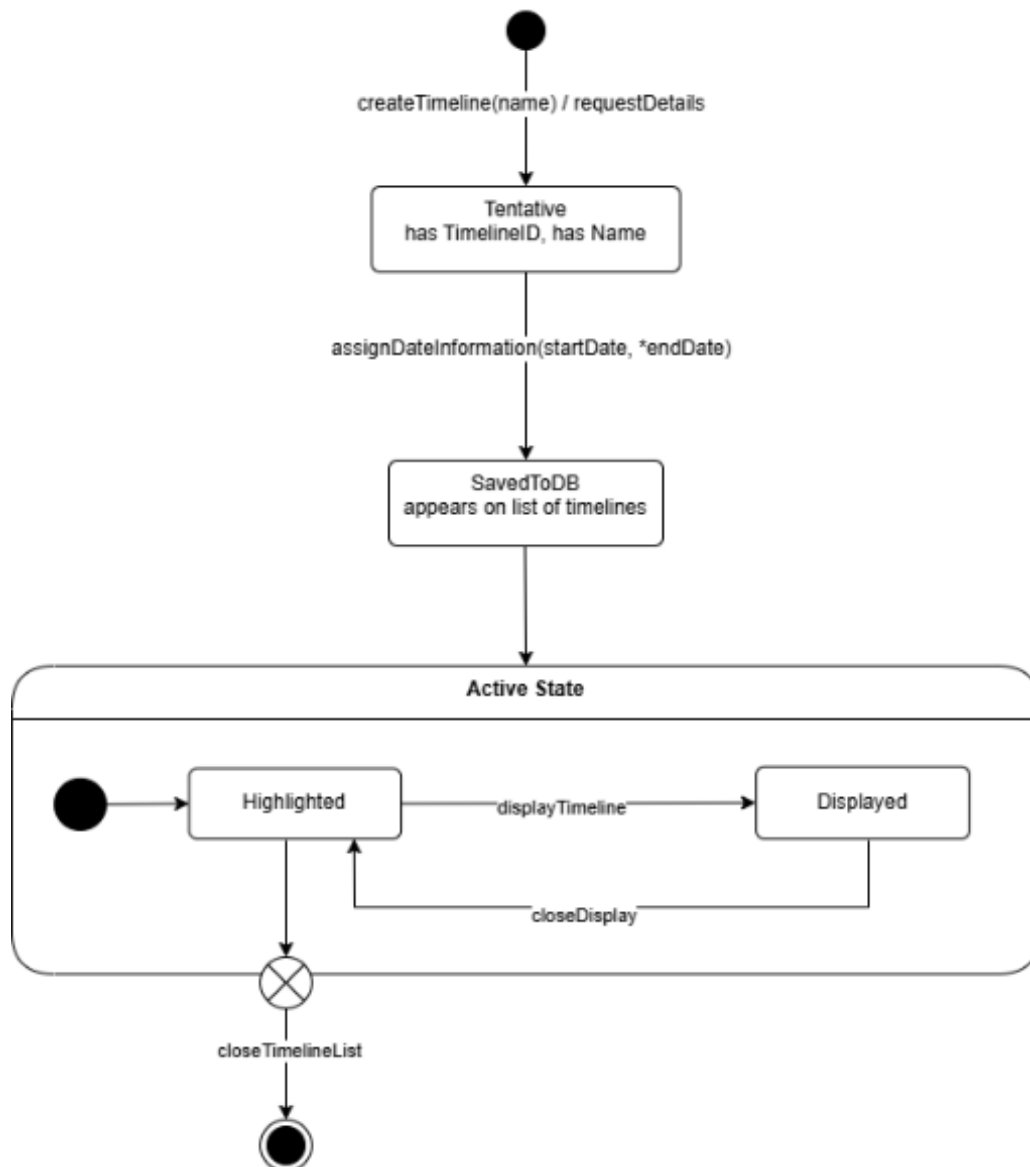
- **newTimeline:** Creates a new TimeLine instance with the provided name.
- **editTimeline:** Updates the details of an existing timeline.
- **addLayer:** Adds a new layer to an existing timeline.
- **getSavedTimelines:** Fetches all saved timelines for a specific user based on their `userId`.
- **viewTimeline:** Displays the details of a specific timeline.
- **addEvent:** Creates and adds a new event to a timeline.
- **editEvent:** Edits an event within a specified timeline by event ID.



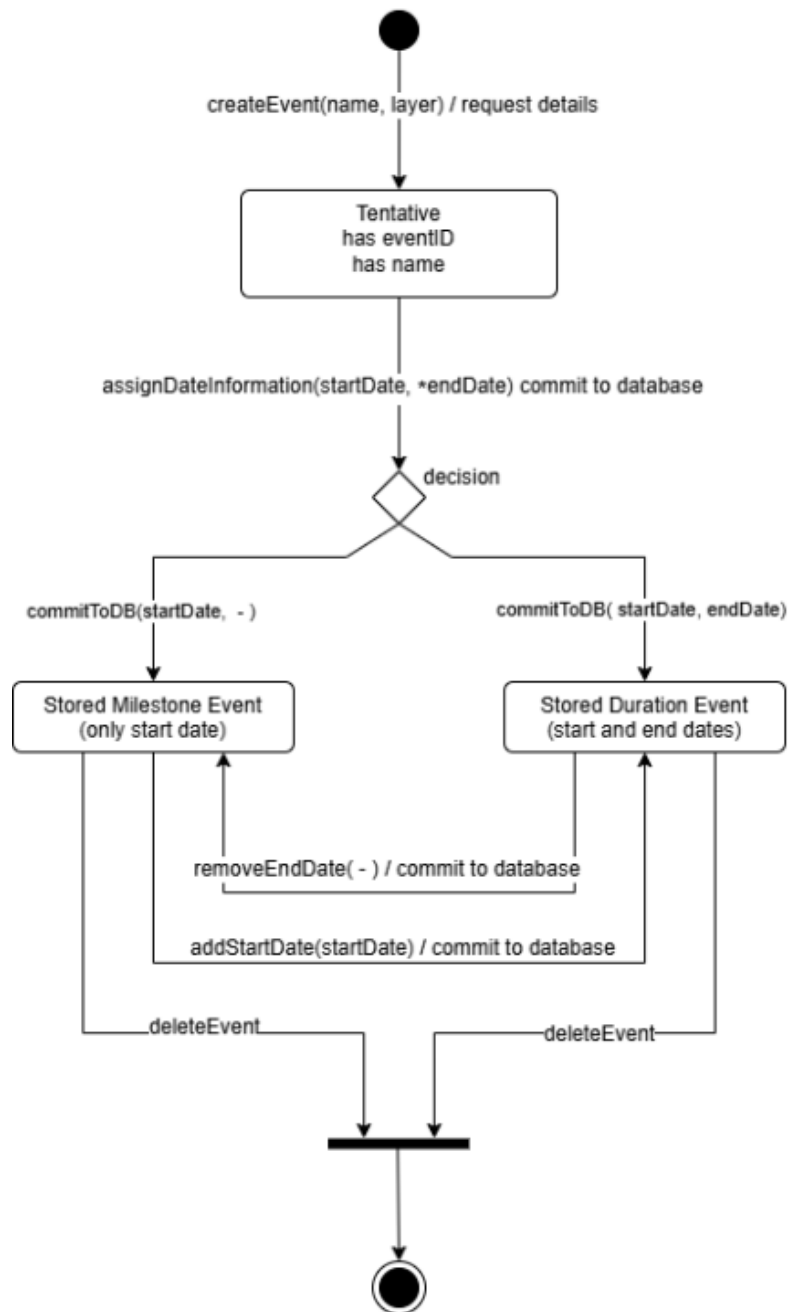
# STATECHART DIAGRAMS

---

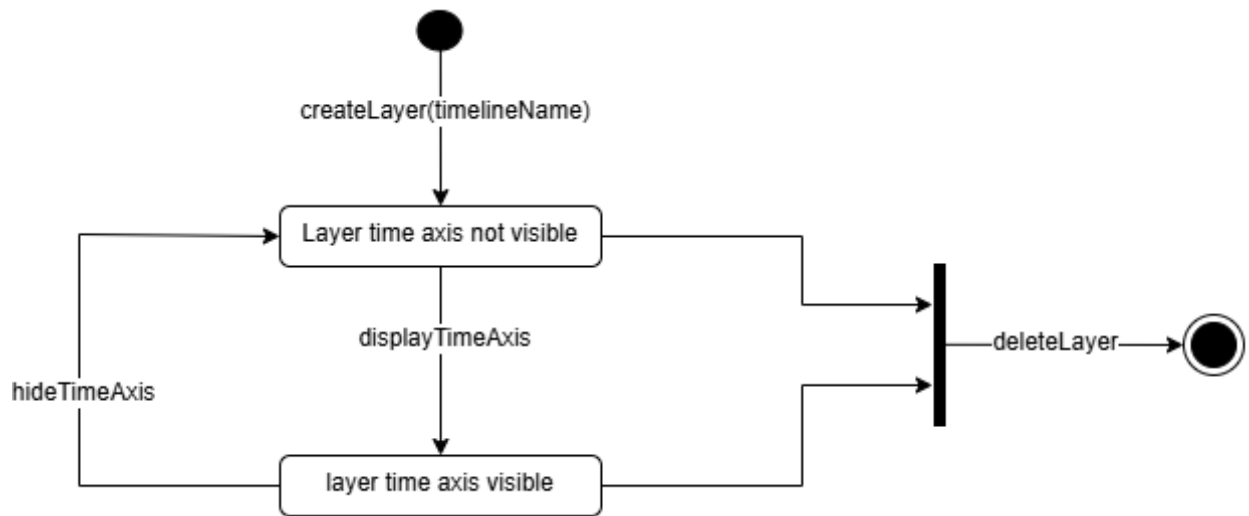
## TIMELINE CLASS



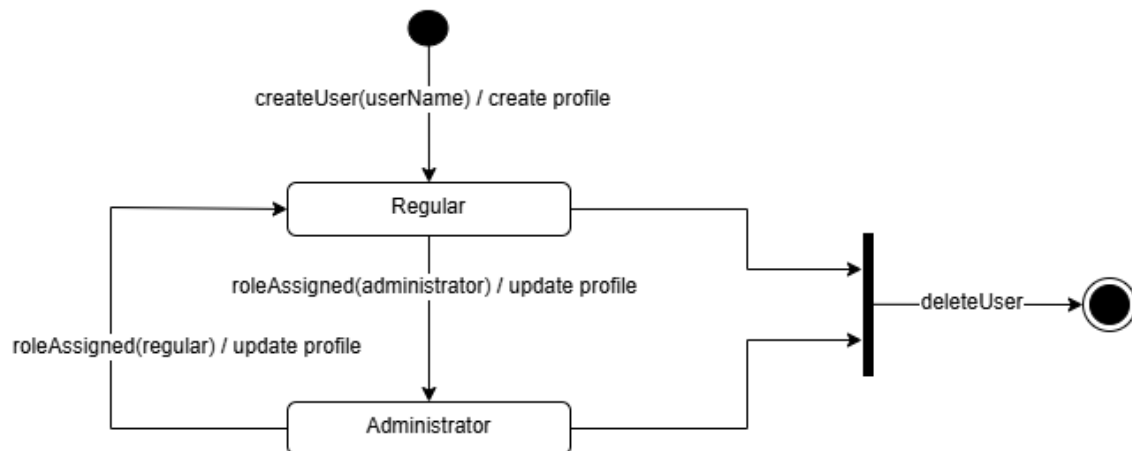
## EVENT CLASS



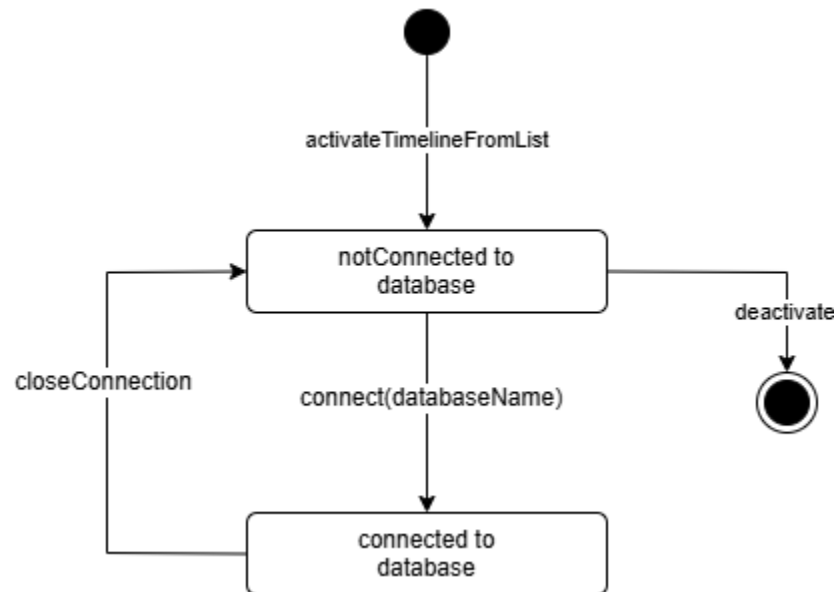
## VISUALIZATION CLASS



## USER CLASS



## DATABASEMANAGER CLASS



## SEARCHENGINE CLASS

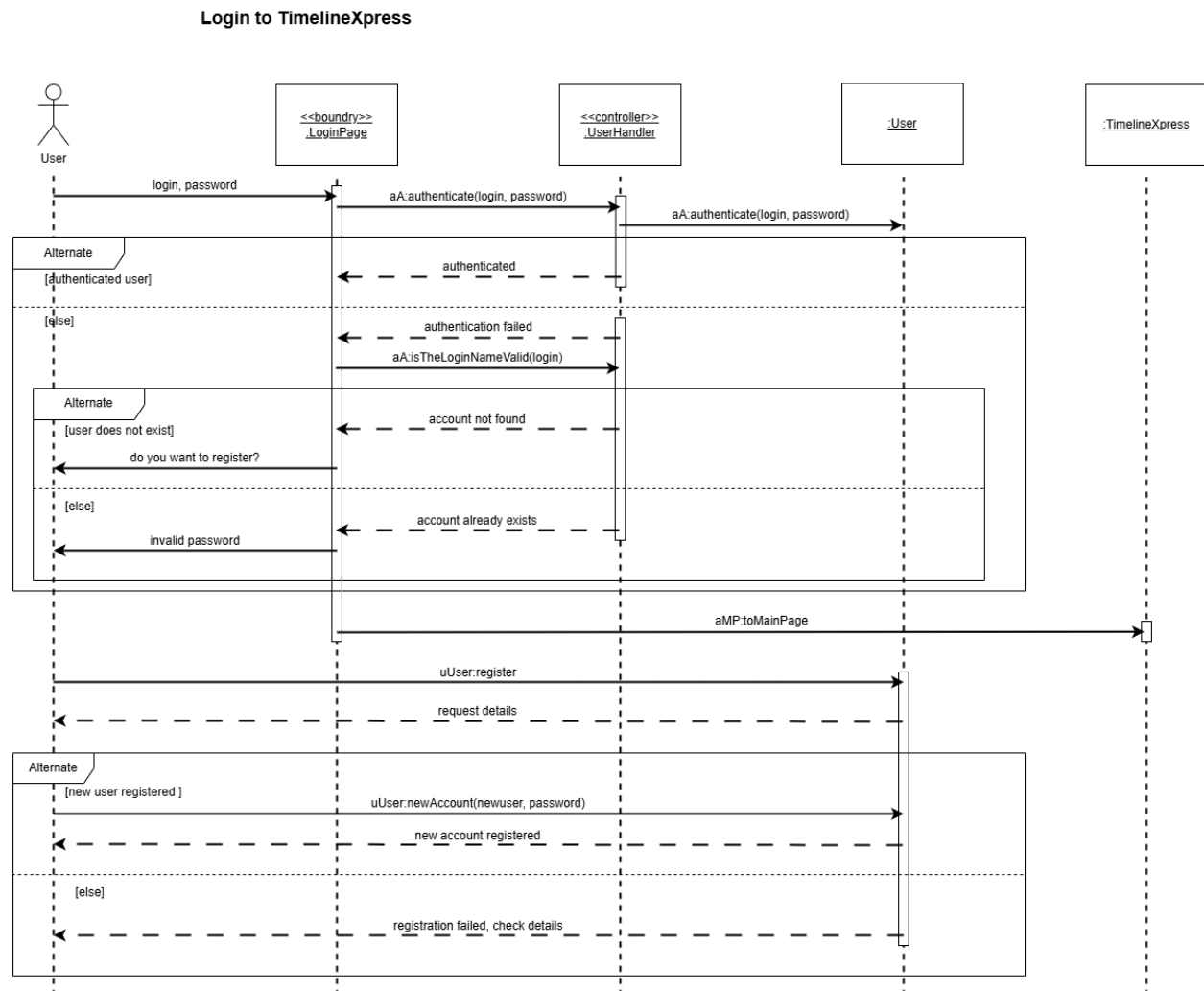
The SearchEngine class provides only services and thus does not have a state machine diagram.

## USERHANDLER CLASS

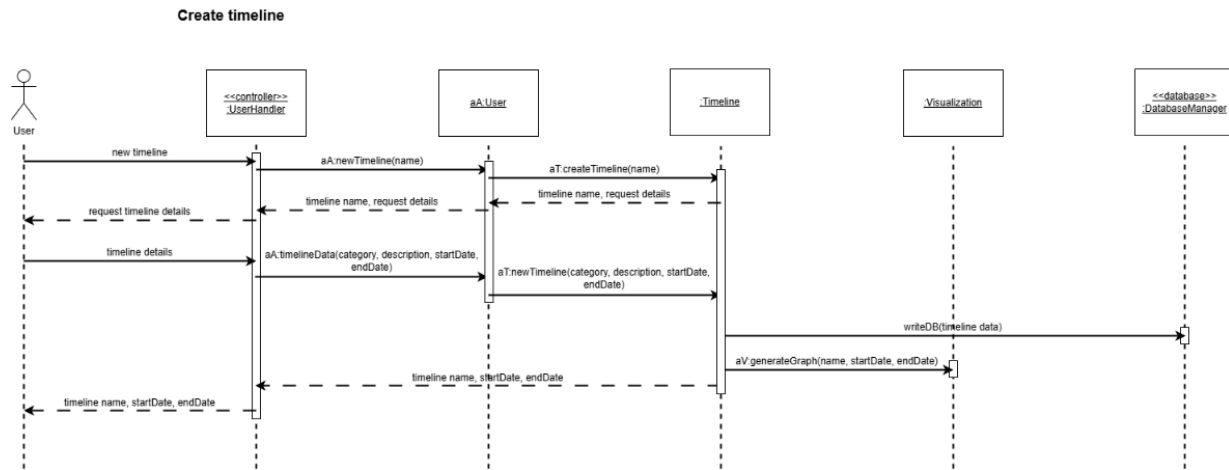
The UserHandler class is a controller class that does not have a state machine diagram.

# SEQUENCE DIAGRAMS

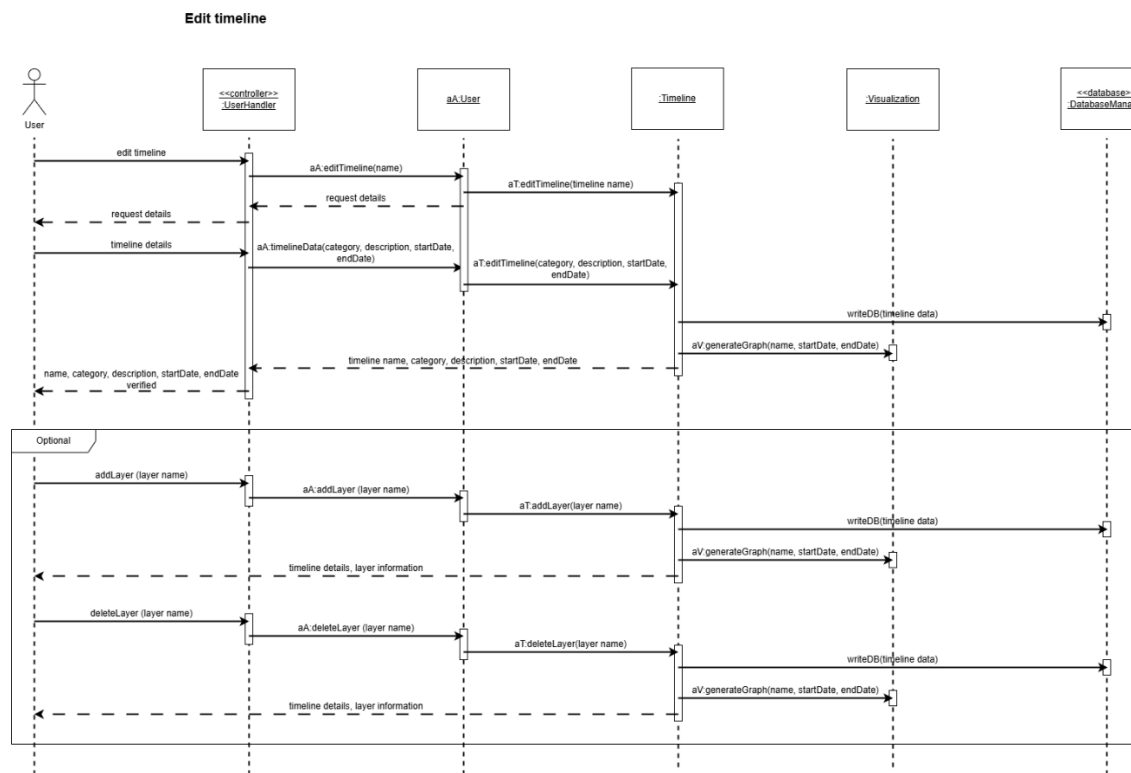
## USE CASE 1: LOGIN TO TIMELINEXPRESS



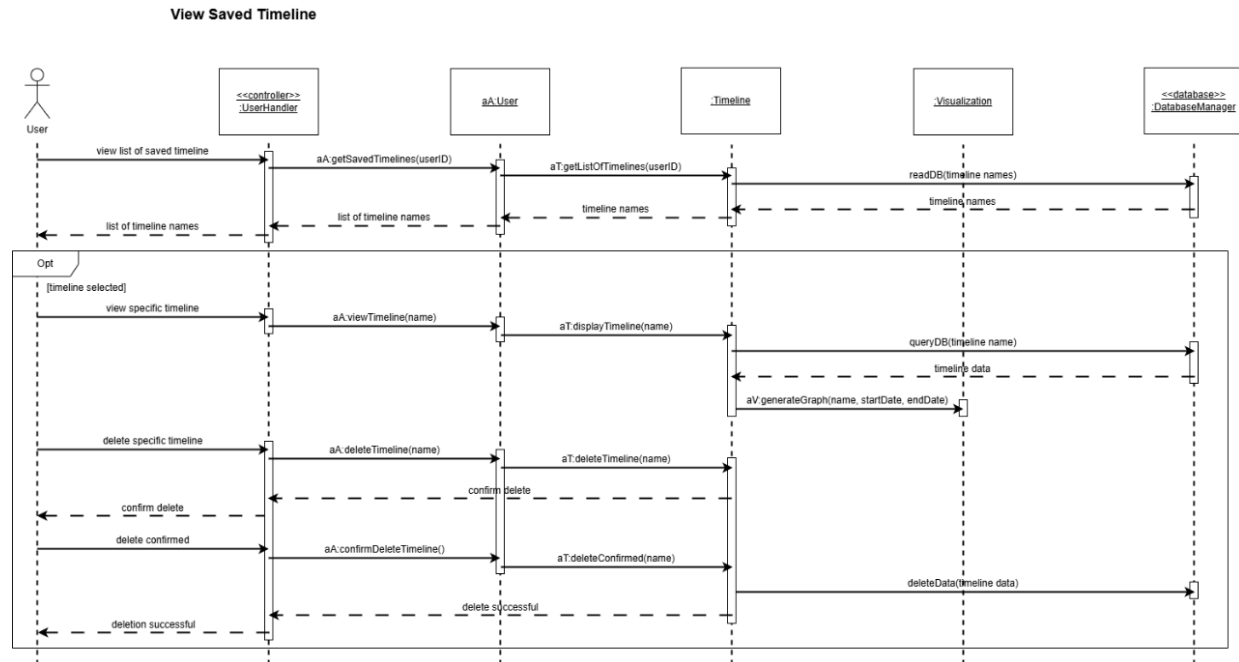
## USE CASE 2: CREATE TIMELINE



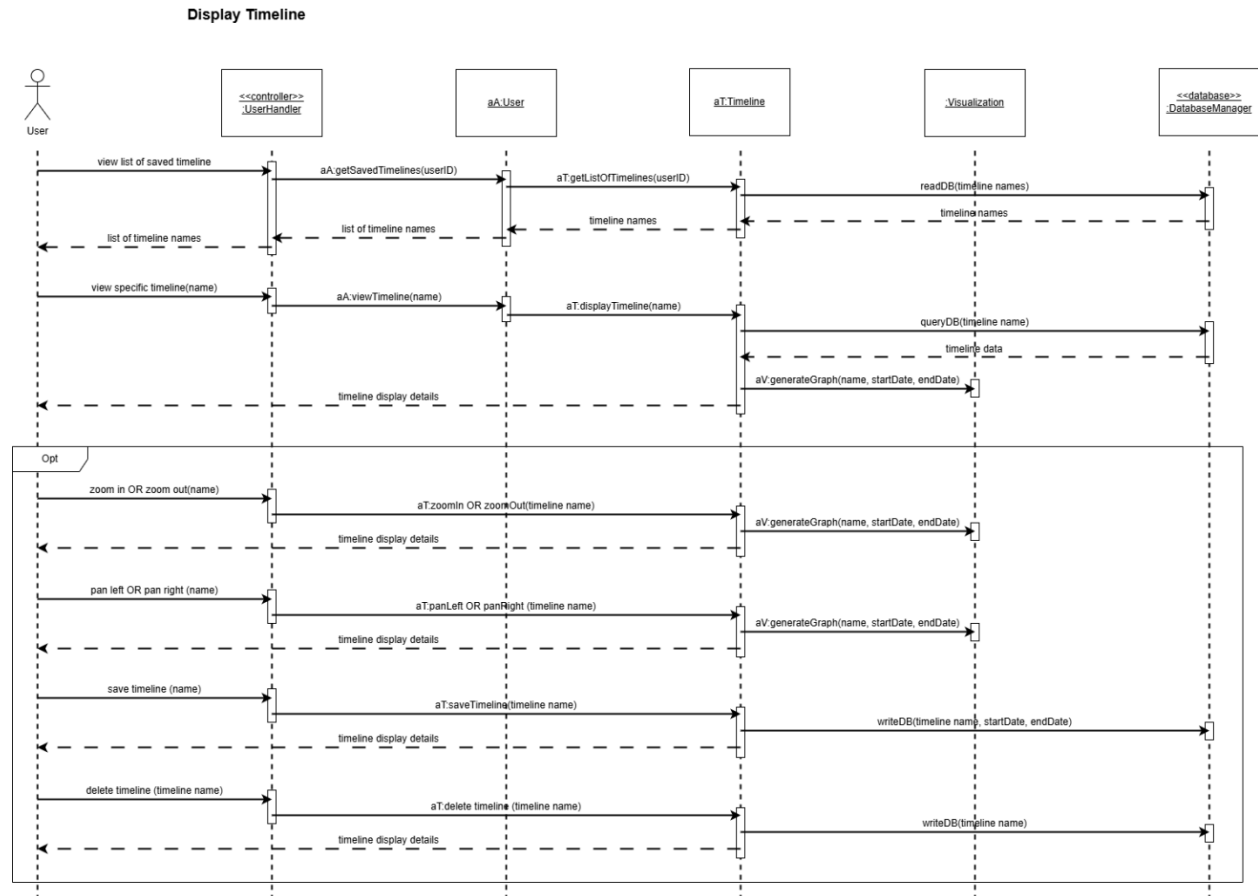
## USE CASE 3: EDIT TIMELINE



## USE CASE 4: VIEW ALL SAVED TIMELINES



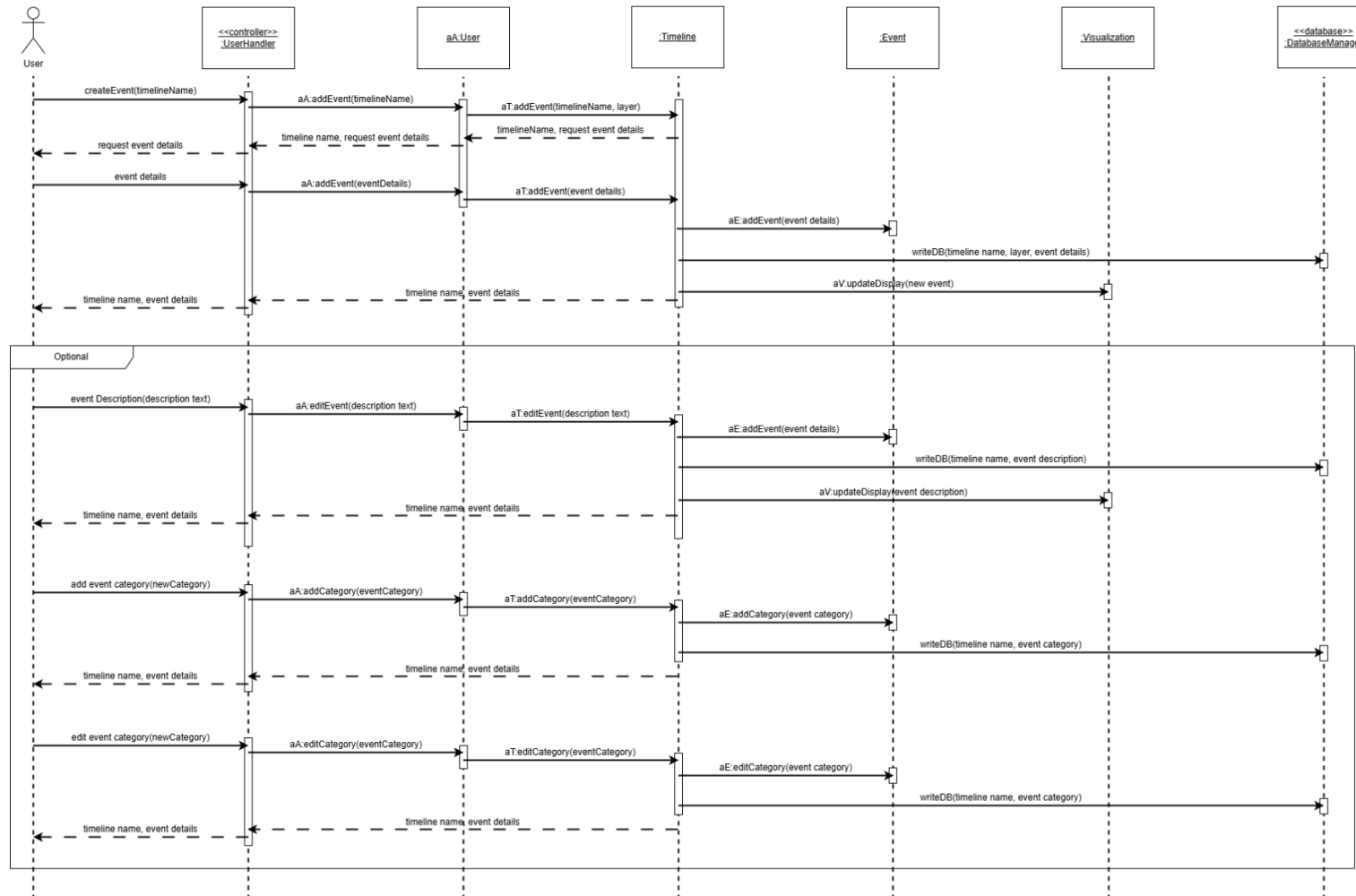
## USE CASE 5: DISPLAY A TIMELINE





## USE CASE 6: CREATE EVENT

### Create Event



## USE CASE 7: EDIT EVENT

### Edit Event

