Theater Management System

Prepared by:

Alexander Prospal

December 7, 2025

1. Application Background

High school theatre productions depend on a large collection of costumes, props, accessories, wigs, makeup kits, and other performance-related items. In the program for which this system was built, these items were previously tracked through scattered spreadsheets, handwritten notes, and informal communication. As the number of productions and students increased, this led to problems such as misplaced items, inconsistent naming conventions, limited visibility into which items were in use, and frequent confusion about what was available at any given time. Because multiple productions often overlap and many items are shared across rehearsals, fittings, and performances, the lack of an organized system created unnecessary delays and inefficiencies for directors, designers, wardrobe staff, and actors.

The purpose of this project is to develop a centralized, functional inventory management system that stores all information related to items, members, productions, storage locations, and checkouts in one consistent database. The application replaces the previous spreadsheet-based process with a structured PostgreSQL database and a web interface designed to be user friendly for theatre staff. The system allows users to add, edit, search, and organize items; track where each item is stored; view which items are currently checked out; and record which member has borrowed an item for a rehearsal or performance. Items can also be linked to specific productions and character assignments so that costume designers and stage managers can easily identify which garments or props belong to which show.

The database includes relations for Member, Production, Category, StorageLocation, Item, ItemProduction, and Checkout, defined with primary keys, foreign keys, and constraints to enforce consistency across all stored data. The system supports categories such as costumes, shoes, accessories, props, wigs, and makeup, as well as detailed storage units such as racks, shelves, bins, and boxes. By combining Flask for backend logic, psycopg2 for database connectivity, Jinja for dynamic page generation, and HTML, CSS, and Bootstrap for interface design, the application provides a cohesive and reliable environment for managing the theatre inventory.

This project ultimately improves accuracy, reduces item loss, increases organization, and provides a streamlined process that is more suitable for a theatre group with a growing and frequently used inventory. It ensures that staff can quickly locate items, assign them to productions, and maintain a clear record of usage throughout the rehearsal and performance process.

2. Data Description

The database for the theatre inventory system is designed to store all information needed to manage costumes, props, accessories, storage locations, productions, members, and checkout activity. The system includes several main entities along with relationship tables. Each table uses primary keys, foreign keys, and domain constraints to maintain consistency and prevent invalid data from being stored.

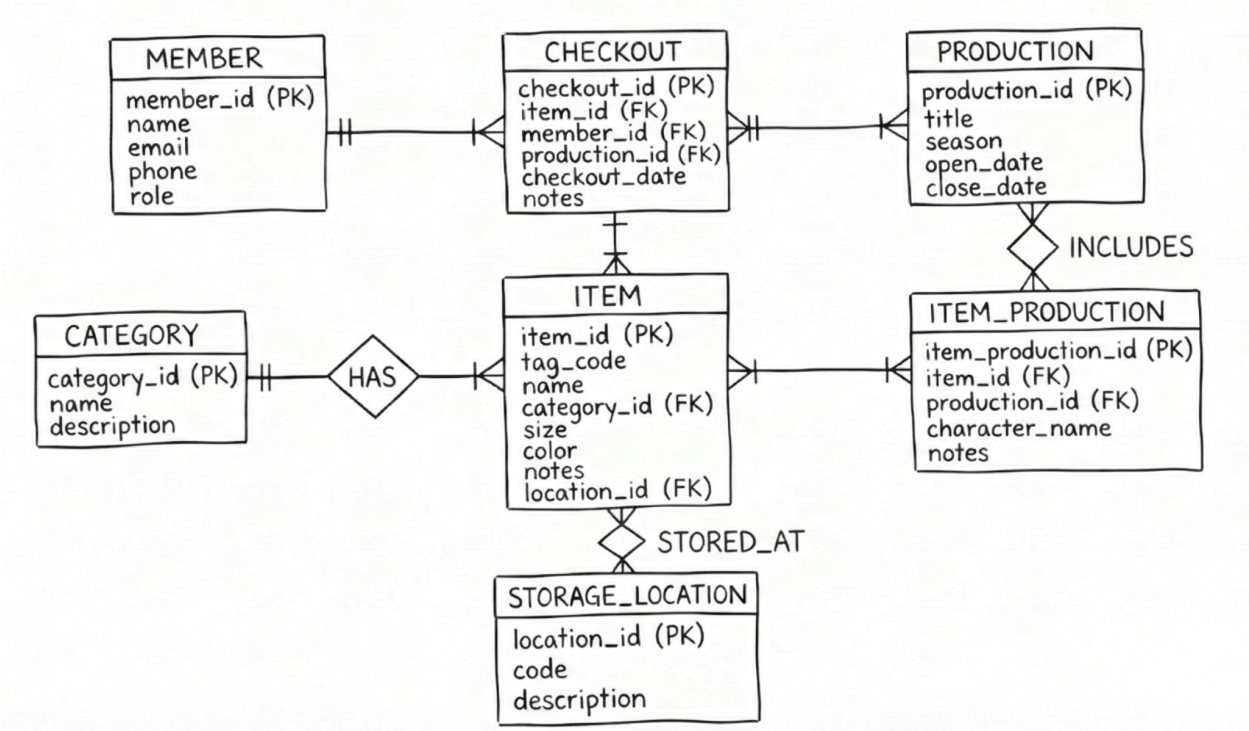
The Member relation stores information about the people who interact with inventory, including actors, directors, crew members, and designers. Members have a name, email, phone number, and role within the theatre program. Each member is uniquely identified by a member_id. The Production relation stores theatrical shows with a title, season, and optional opening and closing dates. Each production has a unique production_id. Categories classify items into broad types such as costumes, shoes, accessories, props, wigs, and makeup. Each category has a unique name and optional description.

Storage locations represent the physical spaces where inventory is kept. Locations include racks, bins, shelves, and boxes, and each has a unique code and a short description. Items are the central component of the database. Each item has a unique tag code, name, category, size, color, and optional notes, along with a location indicating where it is stored. Items reference both category_id and location_id as foreign keys, ensuring consistency across related relations.

The ItemProduction relation associates items with the productions in which they are used. This allows tracking of which costumes, props, or accessories were assigned to specific characters or scenes. Each association includes the item, the production, an optional character name, and notes. The combination of item, production, and character must be unique. The Checkout relation records which member currently has an item and, if applicable, which production it is being used for. A checkout record includes the checkout date, notes, and references to items, members, and productions. If a member or production is removed from the system, the foreign keys in the Checkout table are set to NULL so that historical information remains intact.

The database enforces several important constraints. Primary and foreign keys ensure that relationships are valid and that references cannot point to nonexistent data. Unique constraints require that category names, storage location codes, and item tag codes do not repeat. Some attributes, such as notes or size, may be left NULL so that the system can accommodate many different types of theatre inventory. The provided sample data includes complete sets of categories, storage locations, many items, production assignments, and checkout examples. These allow the system to demonstrate item searches, updates, and the full check-in and check-out process with a populated and realistic dataset.

3. ER Diagram & Relations



The entities in the E/R diagram map directly to relations in the database, and the relationships shown in the diagram correspond to either foreign key references or separate relationship tables. Each entity box on the diagram becomes its own relation, preserving its listed attributes and primary key. The Member, Production, Category, Storage_Location, and Item tables follow a straightforward mapping in which the primary key of each entity becomes the primary key of the corresponding relation, and all non-key attributes appear unchanged in that relation. The Item table includes two foreign keys, category_id and location_id, reflecting the many-to-one relationships shown from Item to Category and from Item to Storage_Location.

The Checkout relationship in the diagram is represented as a separate relation because it includes its own primary key, checkout_id, as well as descriptive attributes such as checkout_date and notes. In the diagram, Checkout connects Member, Item, and Production. This maps to the Checkout table containing foreign keys referencing all three of those relations. The E/R diagram shows that an item may be checked out by one member at a time and optionally associated with a production. This structure is preserved in the relational schema by making item_id not null and by allowing the member_id and production_id columns to be null when appropriate. The cardinalities drawn in the diagram indicate that multiple checkout entries can exist over time for the same item or member, meaning the Checkout relation represents a historical log rather than a current-state table.

The Item_Production relationship is also represented as its own relation in the schema. This reflects the many-to-many relationship between items and productions shown in the ER diagram, where one production may use many items and one item may be reused across multiple productions. The attributes character_name and notes are relationship-specific and therefore belong in Item_Production rather than in either the Item or Production relations. The primary key item_production_id uniquely identifies each pairing. The foreign keys item_id and production_id link the relationship back to its participating entities.

The Category and Storage_Location relationships to Item, labeled HAS and STORED_AT on the diagram, are implemented by placing the corresponding foreign keys inside Item. This removes the need for separate relationship tables because each of these is a many-to-one relationship. Each item belongs to exactly one category and one storage location, matching the clean foreign key structure present in the mapped schema.

4. Functional Dependencies

The functional dependencies listed below are derived from the semantics of the attributes in each relation and from the structure of the E/R diagram. In each case, the primary key determines all other attributes in the relation. No additional nontrivial dependencies exist among non-key attributes, and therefore all relations satisfy BCNF.

Member
The Member relation stores identifying and contact information for individuals involved in productions. Each member has a unique identifier. Since none of the descriptive attributes determine any others, the only functional dependency is

- `member_id → {member_id, name, email, phone, role}`

Because member_id is a superkey, the relation is in BCNF.

Production
Each production is identified by a unique production_id, and its descriptive attributes such as title, season, open_date, and close_date do not determine one another. The functional dependency is

- `production_id → {production_id, title, season, open_date, close_date}`

Since production_id is a superkey, the relation is in BCNF.

Category
The Category relation includes a unique identifier category_id and a unique category name. Both category_id and name can uniquely determine the row. The dependencies are

- `category_id → {category_id, name, description}`
- `name → {category_id, name, description}`

Because both determining attributes are superkeys, the relation is in BCNF.

Storage_Location
This relation contains a unique location_id along with a storage code and description. The dependencies follow directly from the keys:

- `location_id → {location_id, code, description}`

Because code is also unique, it determines the same attributes, although location_id functions as the primary superkey in the schema. The relation is in BCNF.

Item
Each item has a unique item_id as well as a unique tag_code. All descriptive fields, including category_id and location_id, depend on the item identifier. The dependencies are

- `item_id → {item_id, tag_code, name, category_id, size, color, notes, location_id}`
- `tag_code → {item_id, tag_code, name, category_id, size, color, notes, location_id}`

Both item_id and tag_code act as superkeys, so the relation is in BCNF.

Checkout
The Checkout relation records individual checkout events. Each checkout_id uniquely identifies a record. Since multiple checkouts may involve the same item, member, or production, none of those foreign keys determine the others. The dependency is

- `checkout_id → {checkout_id, item_id, member_id, production_id, checkout_date, notes}`

With checkout_id as the superkey, the relation is in BCNF.

Item_Production
This relation represents use of an item in a production. Each record is uniquely identified by item_production_id. Because items may appear in multiple productions, and productions may reuse items, neither item_id nor production_id alone determines the remaining attributes. The only nontrivial dependency is

- `item_production_id → {item_production_id, item_id, production_id, character_name, notes}`

Since item_production_id is a superkey, the relation is in BCNF.

Overall, every relation in the schema uses keys that fully determine all attributes, and no relation contains partial or transitive dependencies. As a result, the full database design satisfies BCNF and avoids redundancy while preserving the intended semantics of the theatre inventory system.

5. Relational Schemas

Member

```
CREATE TABLE Member (
    member_id   SERIAL PRIMARY KEY,
    name        TEXT NOT NULL,
    email       TEXT UNIQUE,
    phone       TEXT,
    role        TEXT
);
```

Production

```
CREATE TABLE Production (
    production_id SERIAL PRIMARY KEY,
    title         TEXT NOT NULL,
    season        TEXT,
    open_date     DATE,
    close_date    DATE
);
```

Category

```
CREATE TABLE Item (
    item_id      SERIAL PRIMARY KEY,
    tag_code     TEXT NOT NULL UNIQUE,
    name         TEXT NOT NULL,
    category_id  INT REFERENCES Category(category_id),
    size         TEXT,
    color        TEXT,
    notes        TEXT,
    location_id  INT REFERENCES StorageLocation(location_id)
);
```

Storage_Location

```
CREATE TABLE Item (
    item_id      SERIAL PRIMARY KEY,
    tag_code     TEXT NOT NULL UNIQUE,
    name         TEXT NOT NULL,
    category_id  INT REFERENCES Category(category_id),
    size         TEXT,
    color        TEXT,
    notes        TEXT,
    location_id  INT REFERENCES StorageLocation(location_id)
);
```

Item

```
CREATE TABLE Item (
    item_id      SERIAL PRIMARY KEY,
    tag_code     TEXT NOT NULL UNIQUE,
    name         TEXT NOT NULL,
    category_id  INT REFERENCES Category(category_id),
    size         TEXT,
    color        TEXT,
    notes        TEXT,
    location_id  INT REFERENCES StorageLocation(location_id)
);
```

Item_Production

```
CREATE TABLE Checkout (
    checkout_id   SERIAL PRIMARY KEY,
    item_id       INT NOT NULL REFERENCES Item(item_id) ON DELETE CASCADE,
    member_id     INT REFERENCES Member(member_id) ON DELETE SET NULL,
    production_id INT REFERENCES Production(production_id) ON DELETE SET NULL,
    checkout_date DATE NOT NULL DEFAULT CURRENT_DATE,
    notes         TEXT
);
```

Checkout

```
CREATE TABLE Checkout (
    checkout_id   SERIAL PRIMARY KEY,
    item_id       INT NOT NULL REFERENCES Item(item_id) ON DELETE CASCADE,
    member_id     INT REFERENCES Member(member_id) ON DELETE SET NULL,
    production_id INT REFERENCES Production(production_id) ON DELETE SET NULL,
    checkout_date DATE NOT NULL DEFAULT CURRENT_DATE,
    notes         TEXT
);
```

6. Example Queries

6.1 Items Currently in Stock

This query finds all items that are not currently checked out to anyone. It returns the item id, name, and tag code for every item whose id does not appear in the Checkout relation. In other words, it lists everything that should physically be on the shelves or racks.

Query in SQL:

```
SELECT
    item_id,
    name,
    tag_code
FROM Item
WHERE item_id NOT IN (
    SELECT item_id
    FROM Checkout
);
```

Query in relational algebra:

$$\text{InStockItems} = \pi_{\text{item\_id, name, tag\_code}}(\text{Item}) \; - \; \pi_{\text{Item.item\_id, Item.name, Item.tag\_code}}(\text{Item} \bowtie \text{Checkout})$$

Partial query result:

	item_id [PK] integer	name text	tag_code text
1	2	White Dress Shirt	C002
2	4	Blue Chorus Dress	C004
3	6	Green Peasant Skirt	C006
4	7	Gold Vest	C007
5	8	Grey Suit Jacket	C008
6	9	Black Tuxedo Pants	C009
7	11	Leather Jacket	C011
8	12	Navy Blazer	C012
9	13	Plaid School Skirt	C013
10	16	Jazz Shoes	S002
11	17	Tap Shoes	S003
12	19	Ballet Flats	S005

6.2 Count Items by Category

This query counts how many items fall into each category, such as Costume, Shoes, Accessory, and so on. It uses a left join so that categories with zero items still appear in the result with a count of zero. The output lists each category name and the number of items assigned to that category, ordered from most to fewest items.

Query in SQL:

```
SELECT
    c.name AS category,
    COUNT(i.item_id) AS item_count
FROM Category c
LEFT JOIN Item i ON i.category_id = c.category_id
GROUP BY c.name
ORDER BY item_count DESC;
```

Query in relational algebra:

$$\text{CatCounts} = \gamma_{\text{name, count(item\_id)} \to \text{item\_count}} \left(\text{Category} \bowtie_{\text{Category.category\_id} = \text{Item.category\_id}} \text{Item} \right)$$

Query result:

	category text	item_count bigint
1	Costume	14
2	Accessory	8
3	Shoes	8
4	Prop	6
5	Makeup	2
6	Wig	2

6.3 Example Query 3: Checked-Out Items with Member and Production

This query lists all items that are currently checked out, along with the member who has each item and the production it is for when applicable. It joins Checkout with Item, Member, and Production so that the result shows human-readable names rather than just IDs. This is useful for wardrobe staff or stage managers who need a quick overview of what is out, who has it, and which show it belongs to.

Query in SQL:

```
SELECT
    i.name        AS item_name,
    i.tag_code    AS tag_code,
    m.name        AS member_name,
    p.title       AS production_title,
    c.checkout_date
FROM Checkout c
JOIN Item i
    ON i.item_id = c.item_id
LEFT JOIN Member m
    ON m.member_id = c.member_id
LEFT JOIN Production p
    ON p.production_id = c.production_id
ORDER BY c.checkout_date DESC, item_name;
```

Query in relational algebra:

$$\text{CheckedOutItems} = \pi_{\text{Item.name, Item.tag\_code, Member.name, Production.title, Checkout.checkout\_date}}\Big((\text{Checkout} \bowtie_{\text{Checkout.item\_id}=\text{Item.item\_id}} \text{Item}) \bowtie_{\text{Checkout.member\_id}=\text{Member.member\_id}} \text{Member} \Big) \bowtie_{\text{Checkout.production\_id}=\text{Production.production\_id}} \text{Production}$$

Partial query result:

	item_name text	tag_code text	member_name text	production_title text	checkout_date date
1	Opera Gloves	A007	Riley Chen	[null]	2025-12-06
2	Stage Foundation Pale…	M001	Parker Smith	[null]	2025-12-06
3	White Lab Coat	C014	Alex Rivera	Hamlet	2025-12-06
4	Brown Trench Coat	C005	Sam Owens	Chicago	2025-12-05
5	Knee-High Boots	S007	Jamie Torres	Chicago	2025-12-05
6	Lantern	P004	Drew Martinez	Into the Woods	2025-12-04
7	Red Ball Gown	C003	Taylor Kim	Hamlet	2025-12-04
8	Character Heels	S001	Jordan Lee	Into the Woods	2025-12-03
9	Long Curly Wig	W001	Jamie Torres	Chicago	2025-12-03
10	Black Tailcoat	C001	Jordan Lee	Hamlet	2025-12-02
11	Work Boots	S004	Drew Martinez	[null]	2025-12-01
12	Floral Summer Dress	C010	Morgan Patel	Into the Woods	2025-11-30

7. Implementation: Technologies Used

The theatre inventory system was implemented as a web application backed by a relational database. PostgreSQL served as the database management system, providing reliable storage for all entities and relationships in the schema. The database structure was created using SQL, and tables were populated with initial sample data to support development and testing of the interface and system functionality.

The application backend was written using the Flask web framework. Flask handled routing, form submission, and the execution of SQL queries. When a user added, updated, or deleted an item, category, storage location, production, or checkout entry, Flask processed the corresponding request, constructed the required SQL command, and executed it through the psycopg2 database adapter. psycopg2 enabled direct communication between the Python application and the PostgreSQL server, ensuring that all changes made in the interface were immediately reflected in the underlying data.

Dynamic page rendering was accomplished using the Jinja templating engine. Jinja allowed item lists, category tables, storage location inventories, and production usage pages to update automatically based on the current contents of the database. HTML and CSS were used to structure and style the interface, while Bootstrap provided layout tools, responsive design options, and consistent formatting for tables, forms, and navigation elements. These technologies together produced a clean, functional interface that was accessible to users without technical backgrounds.

Throughout the development process, the focus was on building a system that handled the full workflow of a theatre inventory environment. This included adding new items, editing existing ones, checking items in and out, tracking which items belonged to which productions, and searching the inventory through filters and keyword queries. The design emphasized clarity, ease of use, and accuracy, ensuring that the system could replace the previous spreadsheet-based workflow with a more dependable and scalable solution.

8. What I Learned

Through building this project, I learned how to connect a Flask application to a PostgreSQL database and execute queries securely and efficiently. I gained experience designing a clean, normalized database schema that reduces redundancy and supports the operations of a real inventory system. I learned how to construct many-to-one and many-to-many relationships, enforce constraints, and maintain data consistency across interconnected tables.

I also developed a deeper understanding of how dynamic web pages are generated using Jinja templates. By integrating backend logic with template rendering, I learned how user actions translate into database queries and how query results are transformed into HTML content. This process helped me understand the complete path from data storage to user interface.

Another important aspect of the project was implementing checkout logic. I learned how to represent real-world workflows, such as checking items in and out, linking them to members and productions, and determining whether an item is available or already in use. Building this logic strengthened my understanding of how application rules can be enforced at both the database level and the application level.

Debugging the system taught me how to diagnose issues arising from SQL statements, foreign key constraints, rendering errors, and template logic. Each iteration helped refine the user interface and improve the reliability of the application. Finally, integrating all components into a functional web app gave me a complete view of how backend systems, databases, and front-end templates combine to form a working software solution.