Doctor Appointment Application – Jira Backlog & Design

---

1. Jira Product Backlog

This section details the features required for the Doctor Appointment Application, organized into Epics and user stories, including preliminary estimation via Story Points.

EPIC 1: Doctor Management

This epic focuses on the foundational data required for the system: storing and defining the core properties of medical practitioners.

User Story 1: Create Doctor List As a system, I want to store doctors with their specialization and working hours, So that users can view available doctors.

Attribute	Value
Description Detail	The system must establish the core repository for all practitioners. This includes mandatory fields required for future availability checks and scheduling.
Specialization Examples	Cardiology, Dermatology, General Practice, Pediatrics.
Working Hours Logic	Fixed shift definition. If an appointment slot is 30 minutes, the maximum number of slots is calculated based on the 8-hour window.

Acceptance Criteria (ACs): 1. Each doctor record must uniquely identify via an `ID`. 2. Fields required: Doctor `name` and primary `specialization`. 3. Working hours must be strictly defined as `08:00` (start) to `16:00` (end). 4. The system must enforce a Daily Capacity limit of 20 slots per doctor per day, which needs to be reflected in the initial setup (e.g., via the `Doctor_Daily_Capacity` table initialization).

Story Points: 3

EPIC 2: Doctor Filtering & Viewing

This epic covers the user interface components necessary for patients to find a suitable doctor based on time and medical needs.

User Story 2: Filter Doctors by Date As a user, I want to select a date, So that I can see doctors available on that day.

Attribute	Value
Constraint	The system must check the `Doctor_Daily_Capacity` table for the selected date against the required minimum capacity threshold (e.g., > 0).
Implementation Detail	The UI component must prevent selection of dates in the past.

Acceptance Criteria (ACs): 1. A functional, intuitive date picker component must be implemented on the search interface. 2. The resulting doctor list must dynamically update to show only doctors who have at least one remaining appointment slot available on the selected date.

Story Points: 3

User Story 3: Filter Doctors by Specialization As a user, I want to select a specialization, So that only relevant doctors are displayed.

Attribute	Value
Data Source	The list of specializations populated in the filter dropdown must be derived dynamically from the unique values present in the `Doctors` table.

Acceptance Criteria (ACs): 1. A standard HTML `<select>` or equivalent dropdown menu listing all available specializations must be present. 2. Selection in this filter must narrow the displayed results. 3. Combination Logic: This filter must successfully interact with the Date Filter (User Story 2) to provide an intersection of available doctors (i.e., Doctor X must be a Cardiologist AND have slots available on Date Y).

Story Points: 2

User Story 4: View Doctor List As a user, I want to see a list of doctors, So that I can choose one to book.

Attribute	Value
Data Aggregation	This view requires joining data from `Doctors` and `Doctor_Daily_Capacity` based on the selected filters (Date).

Acceptance Criteria (ACs): 1. The list must clearly display the Doctor's `name` and their associated `specialization`. 2. For the selected date, the current `remaining_capacity` must be prominently displayed (e.g., "Slots Left: 5"). 3. The "Book Appointment" button associated with a doctor must be visually disabled (greyed out) if their `remaining_capacity` for that day is zero (0).

Story Points: 3

EPIC 3: Appointment Booking

This epic covers the critical user action: securing a time slot.

User Story 5: Book Appointment As a user, I want to book an appointment with a doctor, So that I can secure my visit.

Attribute	Value
Criticality	High. This operation is transactional and must enforce data integrity against concurrent requests.

Acceptance Criteria (ACs): 1. The booking process is initiated only after a doctor, date, and time slot (implicit via capacity check) have been successfully chosen. 2. Upon successful submission, a new, unique record representing the appointment must be persisted in the `Appointments` table. 3. The system must strictly prevent any attempt to book if the doctor's capacity for that specific date is already zero at the moment of commit. (This is the critical overlap with Capacity Management).

Story Points: 5

User Story 6: Booking Confirmation As a user, I want to see a confirmation message, So that I know my booking was successful.

Attribute	Value
User Experience	Clear, immediate feedback is essential after a successful transactional operation.

Acceptance Criteria (ACs): 1. Upon successful completion of US 5, the user is immediately redirected to a confirmation screen or modal. 2. The confirmation display must explicitly state the name of the Doctor booked and the Date of the appointment. 3. A unique Appointment ID generated during the save must be displayed for reference.

Story Points: 2

EPIC 4: Capacity Management

This epic handles the necessary backend logic to maintain the integrity of the daily appointment limits.

User Story 7: Reduce Capacity After Booking As a system, I want to reduce doctor capacity after each booking, So that the daily limit of 20 is respected.

Attribute	Value
Concurrency Control	This logic must be protected, typically using database transactions and locking mechanisms (e.g., `SELECT FOR UPDATE` in SQL) to avoid race conditions where two users book the last slot simultaneously.

Acceptance Criteria (ACs): 1. The `remaining_capacity` field in `Doctor_Daily_Capacity` must decrease by exactly 1 for the corresponding doctor and date. 2. The system logic must ensure that the `remaining_capacity` field never decrements below zero (0). If an attempted transaction would cause capacity to drop below zero, the entire transaction (booking + capacity update) must fail. 3. The reduction operation must be atomic; both the appointment creation and the capacity update must succeed or fail together (AC for transactional integrity).

Story Points: 5

User Story 8: Handle Full Capacity As a system, I want to block booking when capacity is full, So that no extra appointments are created.

Attribute	Value
Mechanism	This is the enforcement layer triggered by the UI (US 4) and the backend validation (US 7).

Acceptance Criteria (ACs): 1. UI Level: If capacity is 0, the "Book Appointment" button must be visually disabled (as per US 4). 2. API Level: If a user attempts to submit a booking request when capacity is 0 (e.g., due to a client-side validation bypass), the backend API must return an appropriate error code (e.g., 409 Conflict) and refuse to proceed with creating the appointment record. 3. A user-friendly message indicating "Capacity Full" must be returned or displayed upon failed booking attempts due to exhaustion of slots.

Story Points: 2

---

2. Agile Sprint Planning

This section outlines the proposed sequencing of work based on dependencies. Core setup and basic search/booking must precede final refinement and robust error handling.

Sprint 1 (MVP Focus: Core Functionality)

The goal of Sprint 1 is to deliver a functioning, though not fully resilient, booking pipeline.

User Story	Story Points	Rationale
User Story 1: Create Doctor List	3	Foundation setup. Must exist before any filtering or booking.
User Story 2: Filter Doctors by Date	3	Essential for narrowing the search scope.
User Story 3: Filter Doctors by Specialization	2	Secondary filtering requirement.
User Story 4: View Doctor List	3	Presentation layer for filtered results.
User Story 5: Book Appointment (Initial Save)	5	The core transactional goal. Assumes basic capacity check is in place.
Total Sprint Points	16	

Sprint 2 (Capacity & Refinement)

Sprint 2 focuses on enforcing data integrity, locking down capacity, and handling confirmations.

User Story	Story Points	Rationale
User Story 6: Booking Confirmation	2	Post-booking user feedback. Dependent on US 5 success.
User Story 7: Reduce Capacity After Booking	5	Critical backend integrity work, including transactional safety.
User Story 8: Handle Full Capacity	2	Implementation of the frontend and backend locks based on capacity state.
Total Sprint Points	9	

---

3. Wireframe Mockup Description (Filter + Doctor List)

This describes the layout and components of the primary search screen.

A. Filter Section (Top Bar/Sidebar)

This section must be persistent and allow for dynamic searching.

1. Date Picker Component:
    - Label: "Appointment Date"
    - Input Type: Calendar widget (allowing navigation through months, restricting selection to current/future dates).
2. Specialization Dropdown:
    - Label: "Select Specialization"
    - Options: Dynamically populated from the `Doctors` table (e.g., "All," "Cardiology," "Dermatology").
3. Apply Filters Button:
    - Label: "Search" or "Apply Filters"
    - Action: Triggers the backend query combining the selected date and specialization, refreshing the Doctor List Section below.

B. Doctor List Section (Main Content Area)

This area displays the results card-by-card, based on the active filters.

For Each Doctor Card/Row:

Element	Data Source / State	Display Condition
Doctor Name	`Doctors.name`	Always visible.
Specialization	`Doctors.specialization`	Always visible.
Remaining Slots	`Doctor_Daily_Capacity.remaining_capacity`	Required: Must show the current available count for the selected date.
Book Appointment Button	Action Trigger	State: Enabled if `remaining_capacit` `> 0`. State: Disabled/ Greyed out if `remaining_capacit` `== 0`.

4. Database Design Schema

The system requires three primary tables to manage practitioners, track daily availability limits, and record confirmed appointments.

Table 1: Doctors

Stores static information about the practitioners.

Column Name	Data Type	Constraints / Notes	Description
id	INT (or UUID)	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for the doctor.
name	VARCHAR(255)	NOT NULL	Full name of the doctor.
specialization	VARCHAR(100)	NOT NULL	Medical field (e.g., Cardiology).
work_start_time	TIME	NOT NULL, Fixed to `08:00:00`	Defines the start of the working day.
work_end_time	TIME	NOT NULL, Fixed to `16:00:00`	Defines the end of the working day.

Table 2: Doctor_Daily_Capacity

Tracks the dynamic availability for each doctor on a specific calendar day. This is the key table for filtering and capacity enforcement.

Column Name	Data Type	Constraints / Notes	Description
id	INT (or UUID)	PRIMARY KEY, AUTO_INCREMENT	Unique capacity entry ID.
doctor_id	INT (or UUID)	FOREIGN KEY references Doctors(id), NOT NULL	Links capacity to a specific doctor.
date	DATE	NOT NULL	The specific calendar day this capacity applies to.

Column Name	Data Type	Constraints / Notes	Description
total_capacity	INT	NOT NULL, DEFAULT 20	The maximum slots allowed for this doctor on this date (based on US 1).
remaining_capacity	INT	NOT NULL	The current available slots. Initialized to `total_capacity`.
Unique Index	(doctor_id, date)	Ensures only one entry exists per doctor per day.	

Table 3: Appointments

Records every successfully booked consultation slot.

Column Name	Data Type	Constraints / Notes	Description
id	INT (or UUID)	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for the appointment.
doctor_id	INT (or UUID)	FOREIGN KEY references Doctors(id), NOT NULL	The doctor the appointment is with.
date	DATE	NOT NULL	The date of the scheduled appointment.
appointment_time	TIME	NULLABLE (If time slots are implemented later)	If not used initially, this can be derived or set to a placeholder.
created_at	TIMESTAMP	NOT NULL, DEFAULT CURRENT_TIMESTAMP	Time the booking request was processed.

---

5. Detailed Booking Transaction Flow

The booking process (US 5, 6, 7, 8 combined) must be executed within a single, robust database transaction to maintain data consistency.

Scenario: User A attempts to book Doctor D on Date X. At the start, Capacity is 1.

Step 1: Start Transaction

Begin the database transaction (`BEGIN TRANSACTION`).

Step 2: Pre-check Capacity

Execute a read query that locks the relevant capacity row to prevent concurrent modification:

```
 SELECT remaining_capacity
FROM Doctor_Daily_Capacity
WHERE doctor_id = [D\_ID] AND date = [X]
FOR UPDATE; -- Locks the row until COMMIT or ROLLBACK
```

Capacity Check (US 8 Validation): If the returned `remaining_capacity` is less than or equal to 0: * Execute `ROLLBACK` . * Return Error: "Capacity Full." (Terminates flow).

If `remaining_capacity` > 0: Proceed.

Step 3: Create Appointment Record (US 5)

Insert the new appointment record into the `Appointments` table.

```
 INSERT INTO Appointments (doctor_id, date, created_at)
VALUES ([D\_ID], [X], NOW());
```

Step 4: Update Capacity (US 7)

Decrement the capacity count in the locked row. Since the row is locked, this update is safe from race conditions.

$$\text{New Capacity} = \text{Old Capacity} - 1$$

```
 UPDATE Doctor_Daily_Capacity
SET remaining_capacity = remaining_capacity - 1
WHERE doctor_id = [D\_ID] AND date = [X];
```

(Note: Since Step 2 validated capacity > 0, the result of the decrement will always be ≥ 0.)

Step 5: Commit and Confirm (US 6)

If both Step 3 (Insert) and Step 4 (Update) complete successfully: * Execute `COMMIT` . * Display confirmation message to the user, referencing the newly created Appointment ID.

If any step fails (e.g., network interruption during the update): * Execute `ROLLBACK` . * Return Error: "Booking Failed due to system error. Please try again."

This document is now complete and contains extensive detail spanning backlog definition, agile planning, UI representation, detailed database schemas, and transactional integrity logic, as requested.