Account App : Django Template View

`GET /home/`

View: `admin_home`
Description: Authenticates the user using `api_key` and redirects them to their appropriate dashboard (Admin, Doctor, or Patient).
Authentication: Required (`api_key` in session or query param)

`GET /admin_profile/`

View: `user_profile`
Description: Displays the profile page for the currently logged-in admin user.
Authentication: Required

`GET, POST /profile/edit/`

View: `edit_user_profile`
Description: Allows an admin to edit their own profile.
Authentication: Required

`GET, POST /create_doctor/`

View: `create_doctor`
Description: Allows the admin to create a new doctor under their clinic.
Authentication: Required

`GET /list_doctors/`

View: `list_doctors`
Description: Lists all doctors associated with the clinic.
Authentication: Required

GET, POST /edit_doctor/<int:doctor_id>/

View: `edit_doctor`
Description: Edits the information of a doctor based on their ID.
Authentication: Required
Path Parameter:

- `doctor_id`: Integer — the ID of the doctor to edit.

POST /delete_doctor/

View: `delete_doctor`
Description: Deletes a doctor from the clinic.
Authentication: Required

GET /profile/

View: `profile_view`
Description: Displays the current user's profile page.
Authentication: Required

GET /view_clinic_details/

View: `view_clinic_details`
Description: Shows clinic details accessible to the current user.
Authentication: Required

GET /admin_clinic_details/

View: `admin_clinic_details`
Description: Shows clinic details for admin users.
Authentication: Required

GET, POST /clinic/edit/

View: `edit_clinic_details`
Description: Allows editing of clinic information by an admin.
Authentication: Required

GET, POST /create_patient/

View: `create_patient`
Description: Allows an admin to create a new patient profile.
Authentication: Required

GET /list_patients/

View: `list_patients`
Description: Lists all patients registered under the clinic.
Authentication: Required

GET, POST /edit_patient/<int:patient_id>/

View: `edit_patient`
Description: Edits the details of a specific patient.
Authentication: Required
Path Parameter:

- `patient_id`: Integer — the ID of the patient to edit.

POST /delete_patient/

View: `delete_patient`
Description: Deletes a patient from the clinic.
Authentication: Required

`GET /clinic_portfolio/`

View: `clinic_portfolio`
Description: Displays the clinic's portfolio, such as services or treatments offered.
Authentication: Required

Patient Authentication Routes

`GET, POST /patient/login/`

View: `patient_login`
Description: Displays and handles the login form for patients.
Authentication: Not required

`GET, POST /patient/register/`

View: `patient_register`
Description: Allows new patients to register.
Authentication: Not required

`GET, POST /patient/forgot-password/`

View: `patient_login` (reused)
Description: Handles password recovery for patients.
Authentication: Not required

Clinic Subscription Routes

`/clinic/subscription/`

Includes: `subscription_plan.urls`
Description: All routes related to subscription plans, credit bundles, and billing for clinics.
Authentication: Depends on individual views

Appointment App : Django Template

GET /treatment-list/

View: `treatment_list`
Description: Returns a list of all available treatments.
Authentication: Required

GET /treatment_list_for_new_appointment_booking/

View: `treatment_list_for_new_appointment_booking`
Description: Provides treatment options specifically for the new appointment booking workflow.
Authentication: Required

GET /view-treatment/<int:id>/

View: `view_treatment`
Description: Displays detailed information for a specific treatment.
Path Parameter:

- `id`: Integer — ID of the treatment.
  Authentication: Required

GET /get-doctors-for-treatment/<int:treatment_id>/

View: `get_doctors_for_treatment`
Description: Returns a list of doctors associated with the specified treatment.
Path Parameter:

- `treatment_id`: Integer — ID of the treatment.
  Authentication: Required

GET /available-slots/<int:doctor_id>/<int:treatment_id>/

View: `available_slots`
Description: Returns available appointment slots for a given doctor and treatment.
Path Parameters:

- `doctor_id`: Integer — ID of the doctor

- `treatment_id`: Integer — ID of the treatment
  Authentication: Required

GET, POST /slot/<int:slot_id>/<int:treatment_id>/

View: `book_slot`
Description: Books an appointment slot for a specified treatment.
Path Parameters:

- `slot_id`: Integer — ID of the selected slot

- `treatment_id`: Integer — ID of the treatment
  Authentication: Required

POST /fetch-slots/

View: `fetch_slots`
Description: Fetches all slots based on filtering criteria (likely used for dynamic front-end slot

selection).
Authentication: Required

GET /appointment-success/

View: `appointment_success`
Description: Displays confirmation after a successful appointment booking.
Authentication: Required

GET /view_appointments/

View: `view_appointments`
Description: Lists all appointments for the logged-in user (patient).
Authentication: Required

GET /doctor_appointments/

View: `doctor_appointments`
Description: Lists all appointments for the logged-in doctor.
Authentication: Required

GET /appointment-details/<int:appointment_id>/

View: `appointment_details`
Description: Displays details of a specific appointment.
Path Parameter:

- `appointment_id`: Integer — ID of the appointment
  Authentication: Required

POST /cancel_appointment/<int:appointment_id>/

View: `cancel_appointment`
Description: Cancels a specific appointment.
Path Parameter:

- `appointment_id`: Integer — ID of the appointment
  Authentication: Required

Doctor Slot & Exclusion Date Management

`GET /slots-and-excluded-dates/`

View: `doctor_slots_and_excluded_dates_list`
Description: Lists all slots and excluded dates for the logged-in doctor.
Authentication: Required

`POST /add-slot/`

View: `add_slot`
Description: Adds a new appointment slot for a doctor.
Authentication: Required

`POST /edit-slot/`

View: `edit_slot_for_doctor`
Description: Edits an existing slot for a doctor.
Authentication: Required

`POST /delete-slot/`

View: `delete_slot_for_doctor`
Description: Deletes a slot for a doctor.
Authentication: Required

POST /add-exclusion-date/

View: `add_exclusion_date`
Description: Adds a date where the doctor is unavailable.
Authentication: Required

POST /edit-exclude-date/

View: `edit_exclude_date`
Description: Edits an existing excluded date.
Authentication: Required

POST /delete-exclude-date/

View: `delete_exclude_date`
Description: Deletes a doctor's excluded date.
Authentication: Required

Medical Record Management

GET, POST /view_appointments/add-medical-record/<int:appointment_id>/

View: `appointment_medical_record`
Description: Adds or updates medical records for a specific appointment.
Path Parameter:

- `appointment_id`: Integer — ID of the appointment
  Authentication: Required

POST /view_appointments/add-medical-record/

View: `medical_record`
Description: Handles form submission for adding a general medical record (possibly used internally via AJAX).
Authentication: Required

CoreAI :

It contains LLM Code

Subscription Plan :

It will going to contain Subscription Plan Logic

Transcription :

In Patient Dashboard and Doctor Dashboard there is chatbot to communicate with backend we are using websocket connection but before establishing socket connection an api should be called that will create user_prompt in user_specific_prompts folder with **userid_user_specific_prompts.json** after that websocket connection is established

The purpose of Websocket is real-time audio transcription via Deepgram's WebSocket API, processed by a language model (LLM) that responds to both doctors and patients. The WebSocket connection is established via Django Channels, and all user conversations are logged into MongoDB.

1. WebSocket URL Configuration

The WebSocket is defined under the `urls.py` in the Django project:

```
path('ws/transcription/', consumers.TranscriptionConsumer.as_asgi(),
name='transcription')
```

This route is used by the frontend to initiate a WebSocket connection for audio streaming.

2. WebSocket Consumer: `TranscriptionConsumer`

The main consumer for handling real-time audio data is the `TranscriptionConsumer` class. This class inherits from `AsyncWebsocketConsumer` provided by Django Channels and handles all the WebSocket events, including connection management, message processing, and transcription processing.

Key Methods:

a. `connect(self)`

The `connect` method is triggered when a client connects via WebSocket. It is responsible for:

- Initializing internal flags and variables.
- Accepting the WebSocket connection.
- Storing user metadata, such as user ID, email, and role (doctor or patient).
- Preparing to start the connection with Deepgram's transcription service.

```
async def connect(self):
    self.llm = LanguageModelProcessor()  # Initializes the LLM
processor
    self.is_llm_process_enabled = False  # Flag to prevent concurrent
LLM processing
    self.audio_capture_paused = False  # Flag for pausing audio
processing
    self.no_audio_counter = 0  # Counter for audio absence tracking
    await self.accept()  # Accept the WebSocket connection
```

b. `start_deepgram_connection(self)`

This method connects to Deepgram's WebSocket API to receive live audio transcriptions. It uses `aiohttp` to establish a WebSocket session and attaches specific query parameters like smart formatting, punctuation, and interim results.

```
async def start_deepgram_connection(self):
    self.session = aiohttp.ClientSession()
    self.ws = await self.session.ws_connect(

f"{DEEPGRAM_WS_URL}?model=nova-2&smart_format=true&filler_words=true&p
unctuate=true",
        headers={"Authorization": f"Token {DEEPGRAM_API_KEY}"}
    )
```

c. `receive(self, text_data)`

This method processes incoming messages from the frontend. It handles the following actions:

- **"start"**: Initializes user metadata (ID, email, role) and starts the transcription process by calling `start_deepgram_connection`.
- **"pause_backend" / "resume_backend"**: Pauses or resumes the backend audio processing.
- **Audio Data**: Decodes base64-encoded audio data and forwards it to Deepgram for transcription.

```
async def receive(self, text_data):
    text_data_json = json.loads(text_data)
    action = text_data_json.get("action")

    if action == "start":
        self.user_id = text_data_json.get("user_id")
        self.user_email = text_data_json.get("user_email")
        await self.start_deepgram_connection()
    elif action == "pause_backend":
        self.audio_capture_paused = True
    elif action == "resume_backend":
        self.audio_capture_paused = False
```

d. `receive_transcriptions(self)`

This method listens for transcription results from Deepgram and processes them. When a final transcription result is received, it is saved to MongoDB and forwarded to the frontend. The transcription is further processed by the LLM, based on the user role (doctor or patient).

```
async def receive_transcriptions(self):
    async for message in self.ws:
        if message.type == aiohttp.WSMsgType.TEXT:
            response = json.loads(message.data)
            transcript =
response["channel"]["alternatives"][0].get("transcript", "")
            if response["is_final"]:
                # Save and send the final transcription
                save_message(self.user_id, self.user_email,
transcript, self.tenant_id)
```

Data Flow Overview

1. **Frontend to Backend Communication**:
    - The frontend sends audio data to the backend via WebSocket.
    - The backend decodes the audio data and sends it to Deepgram's WebSocket for transcription.
2. **Deepgram Transcription**:
    - Deepgram returns transcribed text in real time.
    - Once the transcription is finalized, it is sent to the LLM processor for further processing (if the user is a doctor or patient).
3. **LLM Response**:
    - The LLM processes the transcribed text and generates a response.
    - This response is sent back to the frontend for display.

4. **MongoDB Logging**:
    - Both the user messages and LLM responses are saved to MongoDB, ensuring full conversation history is logged.

User Role Handling

- **Doctors**: Transcriptions from doctors are processed with an LLM. The system checks if the doctor has remaining subscription credits, ensuring only a limited number of responses are generated based on the subscription.
- **Patients**: Patients receive LLM responses without any subscription limit.

MongoDB Integration

The method `save_message()` is responsible for saving both user messages and LLM responses to MongoDB:

```
save_message(self.user_id, self.user_email, data, self.tenant_id)
```

The logged data includes:

- **Message Type**: Whether the message is from the user or the LLM.
- **Timestamp**: The time at which the message was sent.
- **Message Content**: The actual text of the transcription or LLM response.

Error Handling and Logging

All key operations, such as WebSocket connection management, audio data processing, and Deepgram interaction, are wrapped in try-except blocks to ensure errors are properly caught and logged.

For example, when establishing the Deepgram WebSocket connection:

```
except Exception as e:
    logger.error(f"Error connecting to Deepgram: {e}")
    logger.error(traceback.format_exc())
```

Logs are written to a logger set up with `setup_logging()`, which ensures that the application logs critical events and errors for monitoring and debugging.

Session Management

- **Audio Capture Pause**: The `audio_capture_paused` flag controls whether audio data is being processed. This allows pausing and resuming transcription dynamically.

- **LLM Process Enable/Disable**: The `is_llm_process_enabled` flag prevents multiple LLM processes from running concurrently.

Treatment Application :

---

1. `add_treatment/`

- **URL Path**: `add_treatment/`

- **View Function**: `views.add_treatment`

- **Purpose**: This route allows users (typically admin or authorized personnel) to create a new treatment. When a user navigates to this route, they will likely be presented with a form to input details about the treatment, which will then be saved to the database.

- **HTTP Method**: Typically `POST` for creating new records, with a form submission.

2. `list_treatments/`

- **URL Path**: `list_treatments/`
- **View Function**: `views.list_treatments`
- **Purpose**: This route is used to list all available treatments in the system. The view will likely retrieve all treatment records from the database and present them in a list format to the user.
- **HTTP Method**: Typically `GET` for displaying the list of treatments.

3. `edit_treatment/<int:treatment_id>/`

- **URL Path**: `edit_treatment/<int:treatment_id>/`
- **View Function**: `views.edit_treatment`
- **Purpose**: This route allows the user to edit an existing treatment. The `<int:treatment_id>` part of the URL is a dynamic parameter that identifies which treatment is being edited. When the user navigates to this route, the system will likely fetch the treatment record based on the ID and provide a form to update its details.
- **HTTP Method**: Typically `GET` to pre-populate the form with current treatment details and `POST` to submit the updated information.

4. `delete_treatment/<int:treatment_id>/`

- **URL Path**: `delete_treatment/<int:treatment_id>/`
- **View Function**: `views.delete_treatment`
- **Purpose**: This route allows users to delete a specific treatment based on its ID. When the user navigates to this route, the system will typically ask for confirmation before deleting the treatment from the database.
- **HTTP Method**: Typically `POST` for deleting the treatment, after confirmation.

Hygen Chatbot Interaction : API

Clinic App has all the hygen logic

Backend Flow

Endpoint : Create Chatbot token

`POST /generate-chatbot-token/<clinic_id>/`

Description

Generates a chatbot token for a specific clinic by its `clinic_id`.

Request Type

`POST`

Path Parameters

Parameter	Type	Description
`clinic_id`	int	ID of the clinic to generate the token for

Request Body

No body is required for this request.

Response

- **200 OK**

json

```
{
  "data_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```

Error Responses

- **404 Not Found**

json

```
{
  "detail": "Clinic matching query does not exist."
}
```

Example cURL

```
curl -X POST http://app.medcor.ai/generate-chatbot-token/5/
```

Endpoint : Chatbot Verify Token

```
GET /verify-token/<clinic_id>/<token>/
```

Description

Verifies a chatbot token for a given clinic. Returns whether the token is valid.

Request Type

```
GET
```

Path Parameters

Parameter	Type	Description
`clinic_id`	int	ID of the clinic
`token`	string	The chatbot token to be verified

Request Body

No body is required for this request.

Response

- **200 OK**

json

```
{
  "success": true
}
```

- or if invalid:

json

```
{
  "success": false
}
```

Error Responses

- **404 Not Found**

json

```
{
  "detail": "Clinic matching query does not exist."
}
```

Example cURL

```
curl -X GET
http://app.medcor.ai/verify-token/5/eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVC
J9..
```

Endpoint : Chatbot User Login

`POST /chatbot/login/<clinic_id>/<token>/`

Description

Authenticates a user (specifically a patient) via chatbot using their email and password, scoped to a clinic and a chatbot token. Returns a JWT `api_key`, user profile info, recent appointment details (if any), and sets a refresh token in HTTP-only cookies.

Request Type

`POST`

Path Parameters

Parameter	Type	Description
`clinic_id`	int	ID of the clinic (tenant)
`token`	string	Encrypted chatbot token to verify

Request Body (JSON)

json

```
{
  "email": "user@example.com",
  "password": "securepassword"
}
```

Successful Response (200 OK)

json

```
{
  "api_key": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "recent_appointment": {
    "id": 123,
    "treatment__name": "Dental Cleaning",
```

```
    "doctor__users__first_name": "John",
    "doctor__users__last_name": "Doe",
    "appointment_slot_date": "2025-04-12",
    "appointment_slot_start_time": "10:00:00",
    "appointment_slot_end_time": "10:30:00"
  },
  "appointement_prompt": "Hi John Doe, you have an appointment for
Dental Cleaning with Dr. John Doe. Your appointment is scheduled from
10:00:00 to 10:30:00 on 2025-04-12.",
  "user_data": {
    "user_id": 42,
    "email": "user@example.com",
    "name": "John Doe",
    "profile_pic": "https://app.medcor.ai/media/profile_pic.jpg",
    "schema_name": "clinic_42"
  }
}
```

> ℹ️ A `refresh_token` will also be set in an **HTTP-only cookie**.

Error Responses

- **400 Bad Request** (Missing fields or invalid credentials):

json

```
{
  "error": "Email, password, and tenant ID are required."
}
```

json

```
{
  "error": "Invalid credentials or please verify your email."
}
```

- **404 Not Found** (Invalid clinic ID):

json

```
{
  "error": "Invalid tenant ID."
}
```

- **500 Internal Server Error** (Unexpected error):

json

```
{
  "error": "Something went wrong. Please try again later.",
  "error": "Full traceback error string..."
}
```

Example cURL

```
curl -X POST https://app.medcor.ai/chatbot/login/5/your_token_here/ \
-H "Content-Type: application/json" \
-d '{
  "email": "user@example.com",
  "password": "securepassword"
}'
```

Endpoint : User Refresh Token

```
POST /chatbot/token/login/<clinc_token>/
```

Description

Refreshes the user's access token using the `refresh_token` stored in the HTTP-only cookie. Validates the token against the provided clinic token and returns a new access `api_key` and user profile data.

Request Type

`POST`

Path Parameters

Parameter	Type	Description
`clinc_token`	string	The clinic's chatbot token for verification

Cookies Required

Cookie Name	Type	Description
`refresh_token`	string	Refresh token used for re-authentication

Request Body

No body is required for this request.

Successful Response (200 OK)

json

```
{
  "api_key": "new_access_token_here",
  "user_data": {
    "user_id": 42,
    "email": "user@example.com",
    "name": "John Doe",
```

```
    "profile_pic": "https://app.medcor.ai/media/profile_pic.jpg",
    "schema_name": "clinic_42"
  }
}
```

Error Responses

- **401 Unauthorized** (Mismatched clinic token):

json

```
{
  "error": "Authentication Failed."
}
```

- **404 Not Found** (Refresh token missing or invalid):

json

```
{
  "message": "login required"
}
```

- **500 Internal Server Error** (Unexpected error):

json

```
{
  "error": "Something went wrong. Please try again later.",
  "message": "Detailed error message here"
}
```

Example cURL

```
curl -X POST
https://app.medcor.ai/chatbot/token/login/clinic_token_here/ \
--cookie "refresh_token=your_refresh_token_here"
```

Endpoint : User Logout

`POST /chatbot/token/logout/<clinc_token>/`

Description

Logs the user out by blacklisting their access token and removing the `refresh_token` from browser cookies. Also expires the refresh token immediately.

Request Type

`POST`

Path Parameters

Parameter	Type	Description
`clinc_token`	string	The clinic's chatbot token to verify

Cookies Required

Cookie Name	Type	Description
`refresh_token`	string	Refresh token used for re-authentication

Request Body

No request body is required.

Successful Response (200 OK)

json

```
{
  "message": "logout successfully"
}
```

> 🔐 The `refresh_token` cookie will be **deleted** and **expired** immediately.

Error Responses

- **401 Unauthorized** (Invalid clinic token):

json

```
{
  "error": "Authentication Failed."
}
```

- **404 Not Found** (Missing or invalid refresh token):

json

```
{
  "message": "login required"
}
```

- **500 Internal Server Error** (Unexpected error):

json

```
{
```

```
  "error": "Something went wrong. Please try again later.",
  "message": "Detailed error message"
}
```

Example cURL

```
curl -X POST
https://app.medcor.ai/chatbot/token/logout/clinic_token_here/ \
--cookie "refresh_token=your_refresh_token_here"
```

Endpoint : List of Treatments

```
GET /<clinic_id>/treatments/
```

Description

Returns a **paginated list of treatments** for a specific clinic. Requires a valid chatbot token passed in the request headers for authentication.

Request Type

```
GET
```

Path Parameters

Parameter	Type	Description
`clinic_id`	int	ID of the clinic to query treatments from

Headers Required

Header Name	Type	Description
`Authorization`	string	Bearer token (chatbot token) used for validation

Example:

makefile

```
Authorization: Bearer your_data_token_here
```

Successful Response (200 OK)

json

```
{
  "count": 12,
  "next": "https://app.medcor.ai/2/treatments/?page=2",
  "previous": null,
  "results": [
    {
      "id": 1,
      "name": "Teeth Whitening",
      "description": "A procedure to brighten your smile.",
      "price": "100.00"
    },
    ...
  ]
}
```

The structure of each treatment object will match your `TreatmentSerializer`.

Error Responses

- **404 Not Found** (Clinic not found):

json

```
{
  "error": "Clinic not found"
}
```

- **401 Unauthorized** (Missing or invalid token):

json

```
{
  "detail": "Invalid or missing chatbot token."
}
```

Example cURL

```
curl -X GET https://app.medcor.ai/42/treatments/ \
-H "Authorization: Bearer your_data_token_here"
```

Endpoint : List of Doctors based on Treatment

```
GET /<clinic_id>/treatments/<treatment_id>/get-doctors/
```

Description

Fetches a list of **doctors associated with a specific treatment** for a given clinic. Requires a valid chatbot token for access.

Request Type

```
GET
```

Path Parameters

Parameter	Type	Description
`clinic_id`	int	ID of the clinic
`treatment_id`	int	ID of the treatment

Headers Required

Header Name	Type	Description
`Authorization`	string	Bearer token (chatbot token for validation)

Example:

makefile

```
Authorization: Bearer your_data_token_here
```

Successful Response (200 OK)

json


```
{
  "treatment": {
    "id": 3,
    "name": "Root Canal"
  },
  "doctors": [
    {
      "id": 12,
      "name": "Dr. John Doe",
      "email": "john.doe@example.com",
      "is_active": true,
      "profile_picture":
"https://app.medcor.ai/media/profile_pics/doctor1.jpg",
      "total_year_of_experience": 8,
```

```
      "specializations": ["Root Canal", "General Dentistry"],
      "contact_number": "+1234567890",
      "gender": "Male"
    },
    ...
  ],
  "clinic": "Bright Smiles Dental"
}
```

Error Responses

- **404 Not Found** (No doctors for treatment):

json

```
{
  "error": "No doctors available for this treatment."
}
```

- **401 Unauthorized** (Missing or invalid token):

json

```
{
  "detail": "Invalid or missing chatbot token."
}
```

- **500 Internal Server Error**:

json

```
{
  "error": "An error occurred while fetching doctors."
}
```

Example cURL

```
curl -X GET https://app.medcor.ai/42/treatments/3/get-doctors/ \
-H "Authorization: Bearer your_data_token_here"
```

Endpoint : Available Doctor Slots

`POST /<clinic_id>/treatments/<treatment_id>/get-doctors/slots/`

Description

This API returns either:

- The **available days** (if `date` is not provided).
- The **booked and available slots** for a specific doctor on a **given date**.

Request Type

`POST`

Path Parameters

Parameter	Type	Description
`clinic_id`	int	ID of the clinic
`treatment_id`	int	ID of the treatment

Headers Required

Header Name	Type	Description
`Authorization`	string	Bearer token (chatbot token for validation)

Request Body

Field	Type	Required	Description
`doctor_id`	int	Yes	ID of the doctor
`date`	string	No	Date in format `YYYY-MM-DD` (optional)

Responses

✅ When only `doctor_id` is sent (no `date`) – 200 OK

json

```
{
  "available_days": ["Monday", "Wednesday", "Friday"]
}
```

✅ When both `doctor_id` and `date` are sent – 200 OK

json

```
{
  "available_slots": [
    {
      "id": 12,
      "doctor": "John",
      "start_time": "10:00",
      "end_time": "10:30",
      "booked": false
    },
    ...
```

```
  ],
  "booked_slots": [
    {
      "id": 14,
      "doctor": "John",
      "start_time": "11:00",
      "end_time": "11:30",
      "booked": true
    },
    ...
  ]
}
```

❌ Error Responses

- **400 Bad Request** – Missing or invalid inputs:

json

```
{
  "error": "Missing required parameter: doctor_id."
}
```

json

```
{
  "error": "Invalid date format. Use YYYY-MM-DD."
}
```

json

```
{
  "message": "Past dates are not allowed."
}
```

- **404 Not Found** – Doctor not found:

json

```
{
  "error": "Doctor not found."
}
```

- **200 OK** – No slots available for selected date:

json

```
{
  "message": "No slots available for the selected date."
}
```

- **500 Internal Server Error** – Unexpected error:

json

```
{
  "error": "Detailed error message."
}
```

Example cURL

1. Fetch available days:

```
curl -X POST https://app.medcor.ai/42/treatments/5/get-doctors/slots/
\
-H "Authorization: Bearer your_chatbot_token" \
-H "Content-Type: application/json" \
-d '{"doctor_id": 7}'
```

2. Fetch booked and available slots for a date:

```
curl -X POST https://app.medcor.ai/42/treatments/5/get-doctors/slots/
\
-H "Authorization: Bearer your_chatbot_token" \
-H "Content-Type: application/json" \
-d '{"doctor_id": 7, "date": "2025-04-12"}'
```

Endpoint : Book Appointment

`POST /<clinic_id>/treatments/<treatment_id>/get-doctors/slots/<slot_id>/book/`

Description

Books an available slot for a **specific doctor** and **treatment** on a given date for a **registered patient**.

Request Type

`POST`

Path Parameters

Parameter	Type	Description
`clinic_id`	int	ID of the clinic
`treatment_id`	int	ID of the treatment
`slot_id`	int	ID of the slot to book

Headers Required

Header Name	Type	Description
`Authorization`	string	Bearer token (chatbot token for validation)

Request Body

Field	Type	Required	Description
`patient_email`	string	Yes	Email of the patient
`doctor_id`	int	Yes	ID of the doctor
`appointment_date`	string	Yes	Date in format `YYYY-MM-DD`

Successful Response – 201 Created

json

```
{
  "message": "Appointment booked successfully.",
  "appointment_id": 87,
  "appointment_status": "Pending"
}
```

❌ Error Responses

- **400 Bad Request** – Missing or invalid inputs:

json

```
{
  "error": "Missing required parameters."
}
```

json

```
{
  "error": "Invalid date format. Use YYYY-MM-DD."
}
```

json

```
{
  "error": "Cannot book past dates."
}
```

json

```
{
  "error": "Slot is unavailable on this date."
}
```

json

```
{
  "error": "Slot is already booked."
}
```

- **404 Not Found** – Entities not found:

json

```
{
  "error": "Patient not found."
}
```

json

```
{
  "error": "Doctor not found."
}
```

json

```
{
  "error": "Treatment not found."
```

```
}
```

json

```
{
  "error": "Slot not found."
}
```

- **500 Internal Server Error** – Unexpected failure:

json

```
{
  "error": "An unexpected error occurred. Please try again later."
}
```

Example cURL

```
curl -X POST
https://app.medcor.ai/42/treatments/7/get-doctors/slots/21/book/ \
-H "Authorization: Bearer your_chatbot_token" \
-H "Content-Type: application/json" \
-d '{
  "patient_email": "john@example.com",
  "doctor_id": 13,
  "appointment_date": "2025-04-12"
}'
```