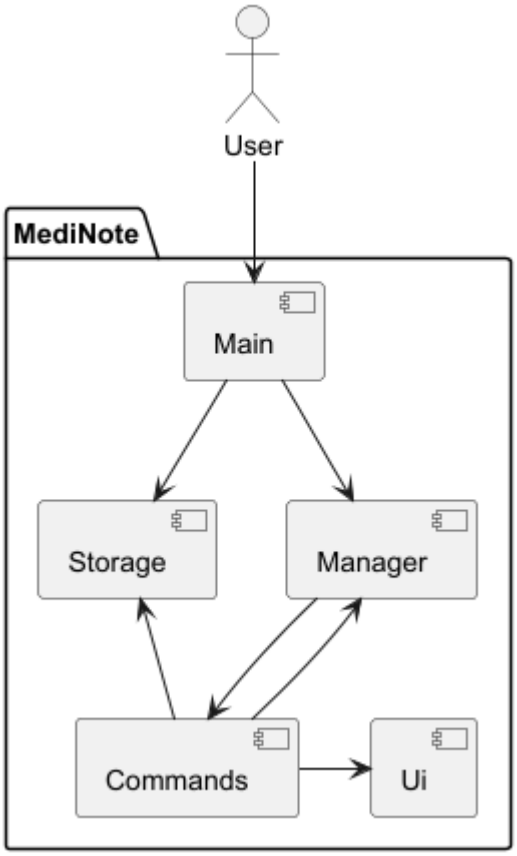
Developer Guide

Acknowledgements

Design & implementation

Overall Architecture



This Architecture Diagram represents the high-level design of MediNote.

Main Components of Architecture

Main consists of the *MediNote* class which is in charge of startup and shutting down.

- At launch, if a save file exists, it will load all patient and doctor information into MediNote.

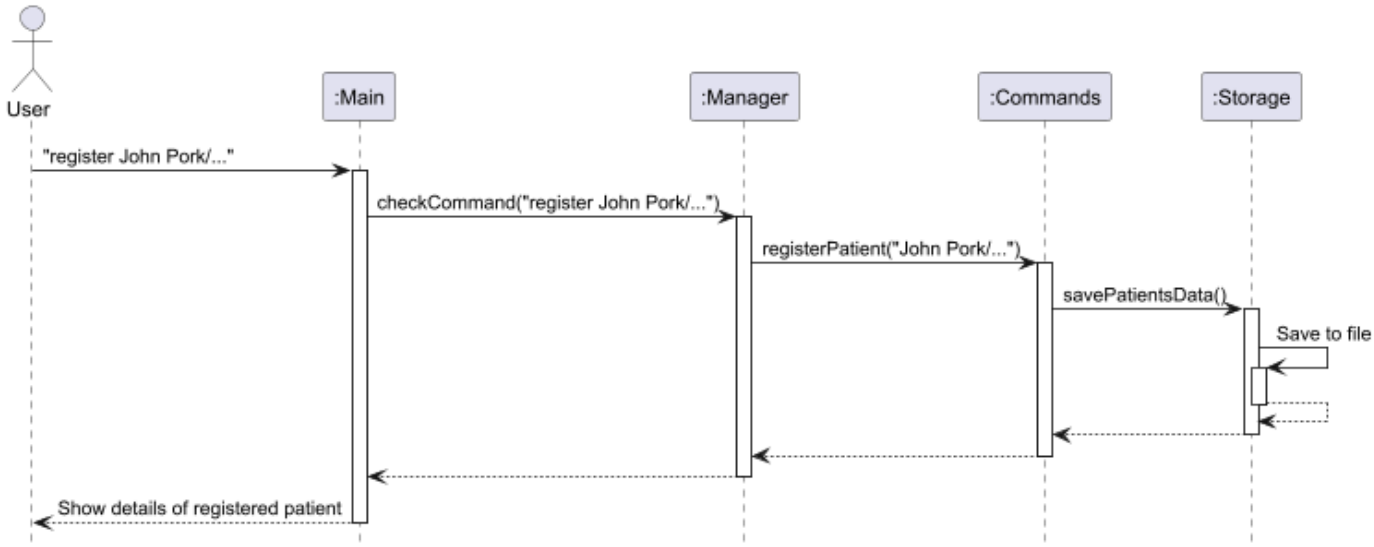
In an overview, most work is done by these components:

- **Main**: Reads user input.
- **Storage**: Loads and writes information as MediNote is running.
- **Manager**: Handles overall patient, doctor information and command calls.

- **Commands**: Executes commands.
- **Ui**: Prints to user *(Currently only help command)*.

How the architecture components interact with each other

The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `register John Pork/High Fever/5 Jan 2025 1730/Cheese allergy`



Each of the main components are separated into functional packages with concrete classes that handle specific responsibilities.

For example, the Manager component contains a TaskManager.java class that parses the input and delegates execution to the respective functions.

In the context of this example:

Package	Key Classes	Responsibilities
main	MediNote	Receives raw user input and initialises the command flow
manager	TaskManager	Parses inputs and delegates execution to the respective command class
commands	RegisterPatient	Contains bulk of code logic
storage	SaveData	Persists data to text files

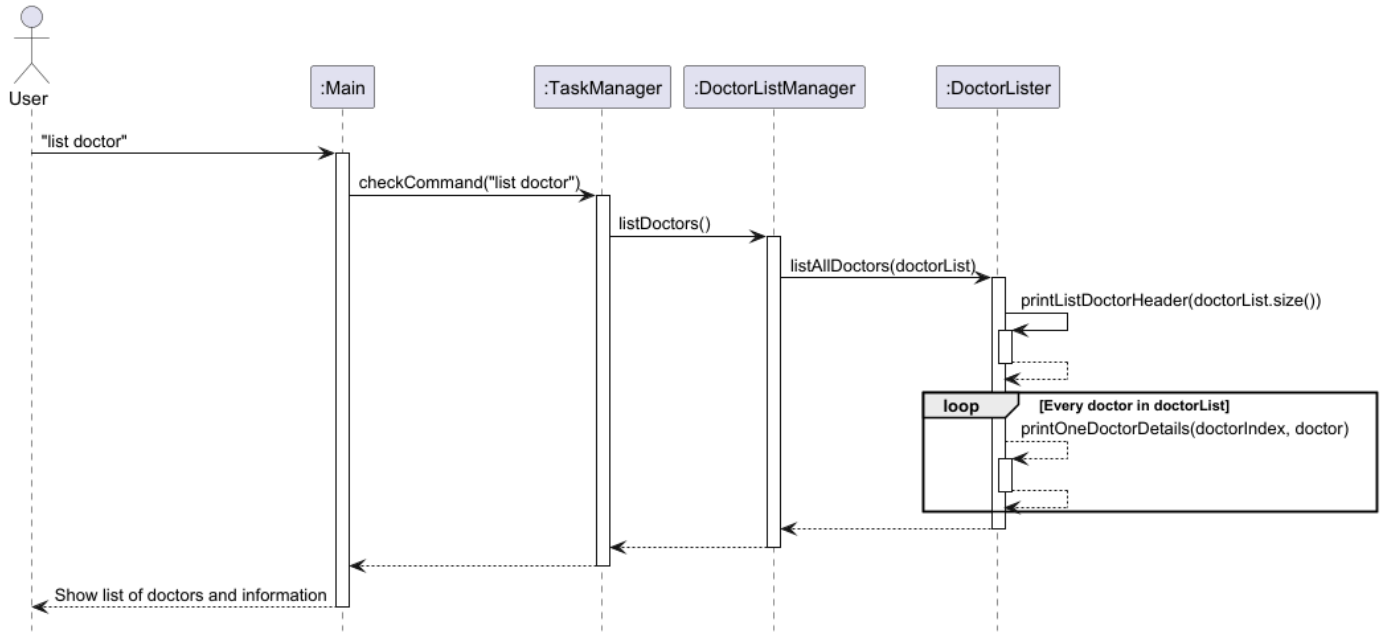
Management of Tracked Doctors

The `DoctorListManager` class main purpose is to maintain `ArrayList<Doctor> doctorList` , which keeps track of the doctors currently working in the hospital.
This class also contains methods that directly modifies the state of `doctorList` .

1. **Adding New Doctors:**
    - `DoctorListManager` contains `addDoctor()` which is called by the `RegisterDoctor` class.
    - `addDoctor()` takes in one `Doctor` type and adds it to `doctorList` .

2. **Removing Existing Doctors:**
    - `DoctorListManager` contains `removeDoctor()` which is called by `DeleteDoctor` class.
    - `removeDoctor()` takes in one `Doctor` type removes it from `doctorList` .
    - It then searches `patientList` and removes the doctor from all patients' `doctorAssigned` attribute.
3. **Listing Existing Doctors:**
    - `DoctorListManager` contains `listDoctors()` which is called by `TaskManager` class.
    - It then calls the `DoctorLister` class which contains the printing logic.

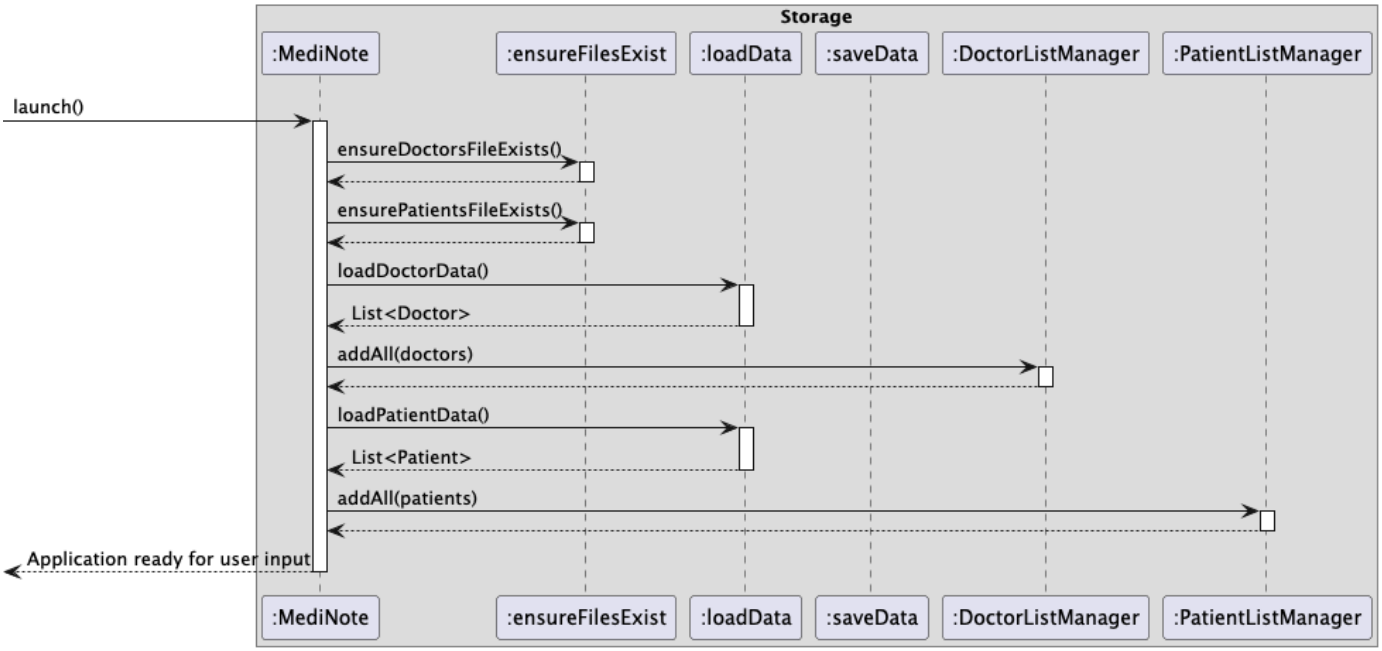
The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `list doctor`



Application Startup Process (Loading Data)

This sequence diagram illustrates the steps executed when the application is launched. The **MediNote** application ensures the necessary data files exist, loads doctor and patient data, and prepares the application for user input.

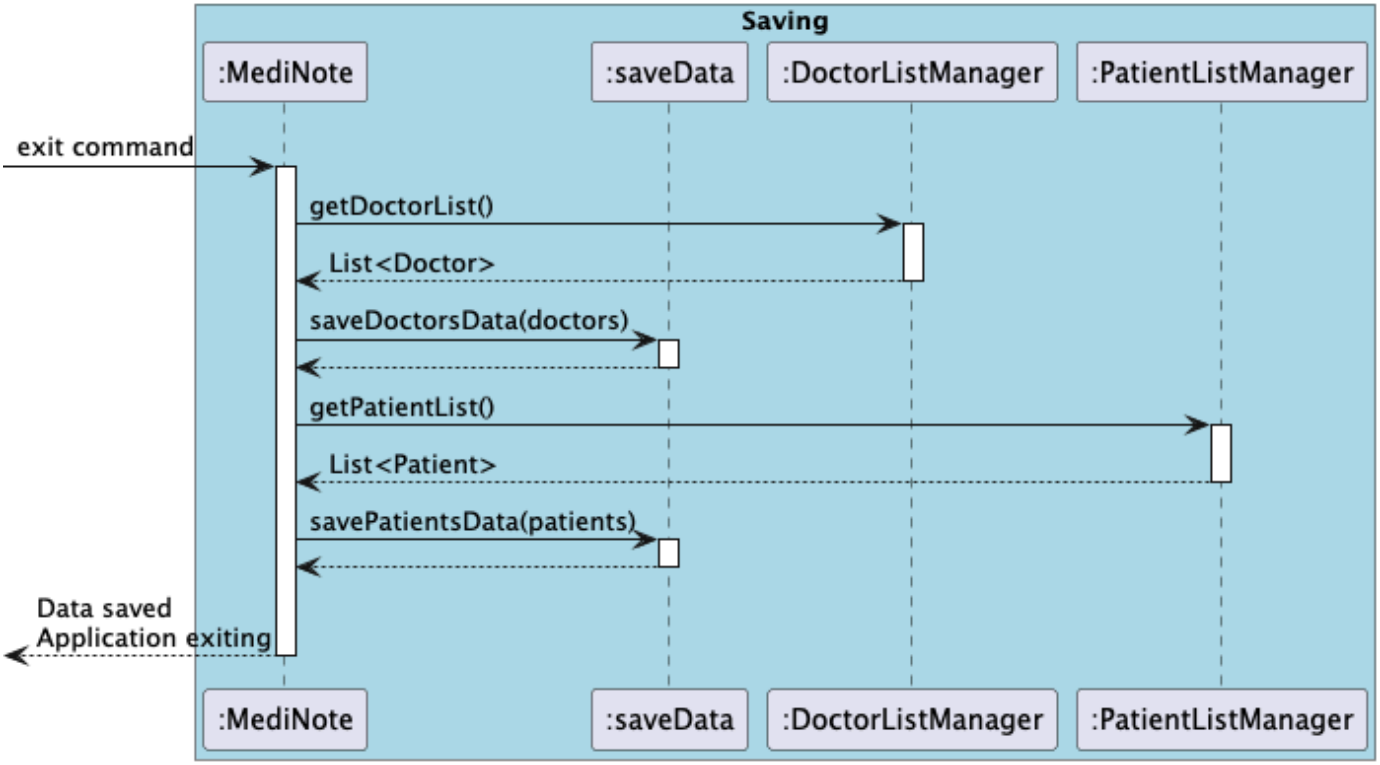
1. **File Existence Check:**
    - ▶ `MediNote` calls `ensureDoctorsFileExists()` and `ensurePatientsFileExists()` to confirm the presence of required storage files.
2. **Doctor Data Loading:**
    - The `loadDoctorData()` method is called from `loadData` , returning a `List<Doctor>` .
    - The retrieved doctor list is then added to `DoctorListManager` using `addAll(doctors)` .
3. **Patient Data Loading:**
    - The `loadPatientData()` method is called from `loadData` , returning a `List<Patient>` .
    - The retrieved patient list is then added to `PatientListManager` using `addAll(patients)` .
4. **Application Readiness:**
    - Once all necessary data is loaded, the application signals readiness for user input.



Application Shutdown Process (Saving Data)

This sequence diagram describes the data-saving process when the application exits. Upon receiving an exit command, the system saves the doctor and patient data before shutting down.

1. **Doctor Data Retrieval & Saving:**
   - ▶ `MediNote` calls `getDoctorList()` on `DoctorListManager` to obtain the list of doctors.
   - The retrieved doctor list is passed to `saveDoctorsData(doctors)`, ensuring it is stored persistently.
2. **Patient Data Retrieval & Saving:**
   - ▶ `MediNote` calls `getPatientList()` on `PatientListManager` to obtain the list of patients.
   - The retrieved patient list is passed to `savePatientsData(patients)`, ensuring it is stored persistently.
3. **Application Shutdown:**
   - Once all data is saved, the application exits gracefully.



Product scope

Target user profile

The target users are hospital management staff. MediNote provides a way to compile the list of patients and which patients the doctors are assigned to, and has features to help edit and keep track of changes in the hospital.

Value proposition

MediNote provides a way to easily track and edit patient and doctor assignments in the hospital. MediNote aims to improve the management capacity and efficiency of hospitals.

User Stories

Version	As a ...	I want to ...	So that I can ...
v1.0	Hospital receptionist	View medical history of patients	Inform the doctor about their past conditions
v1.0	New hospital receptionist	View the list of commands available	Easily navigate data
v1.0	Hospital receptionist	Be able to put in patient and doctor information	Start tracking new patient progress

Version	As a ...	I want to ...	So that I can ...
v1.0	Hospital receptionist	Update patient and doctor information	Fix any mistakes and update records
v1.0	Hospital receptionist	Delete patient or doctor records	Maintain accuracy and cleanliness of data
v2.0	Doctor	View patient's information	So that I know how to treat them
v2.0	Doctor	Update doctor availability	Inform the next patient for treatment
v2.0	Doctor	See patient symptoms	Provide good medication quickly
v2.0	Hospital receptionist	View the status of patients	Check whether they have been discharged
v2.0	Hospital management	View the doctors that were visited the most	Reward them with a break or a pay raise
v2.0	Hospital management	View the type of most frequently visited doctors	Hire more doctors of that specialisation for increased efficiency

Non-Functional Requirements

1. Should work on any *mainstream* OS as long as it has Java `17` or above installed.
2. Should be able to hold up to 1000 persons without a noticeable sluggishness in performance for typical usage.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to key in most of the records faster using commands than using the mouse.

Glossary

- **Mainstream OS**: Windows, Linux, Unix, macOS

Instructions for manual testing