## Data Model

1. Medical Issue: A table to store the details of the medical issues. It can have the following fields:
   - Id: A unique identifier for the medical issue.
   - Name: Name of the medical issue
2. Care Path: A table to store the care path of a medical issues. It can the following fields:
   - Id: A unique identifier for the care path.
   - Name: The name of care path
   - Description: Description of the care path
   - issue_id: A foreign key to the medical issue table for which care path is defined.
3. Category: A table to store the different categories of a care path. It can the following fields:
   - Id: A unique identifier for the care path.
   - Name: The name of the category
   - Description: Description of the category
   - path_id: A foreign key to the care path table for which category is defined.
4. Service: A table to store the details of the services that can be assigned to the care path. It can have the following fields:
   - Id: A unique identifier for the service.
   - Name: Name of the service.
   - Criteria: A set of criteria that needs to be met for the service to be automatically approved. It can be in the form of Boolean expression or a set of rules.
   - Category_id: A foreign key to the category table for which the service is defined.
5. Request: A table to store the service request made by patients. It can have the following fields:
   - Id: A unique identifier for the request.
   - Patient_id: A foreign key to the patient table for which user is defined.
   - Issue_id: A foreign key to the medical issue table for which request is defined.
   - Category_id: A foreign key to the category table for which the request is defined.
   - Service_id: A foreign key to the service table for which the request is defined.
   - Statues: The status of the request, used to track the progress of the request.
6. User: : A table to store the service request made by patients. It can have the following fields:
   - Id: A unique identifier for the user.
   - FirstName:
   - LastName
   - Gender
   - contactNumber
   - email
   - address
   - symptom: The patient condition. Used to compare with the criteria of the service table.

7. Question: A table to store the questions. It can have the following fields:
   - Id: A unique identifier for the user.
   - Question: the text of question
   - Service_id: A foreign key to the service table for which the request is defined.

8. Answer: A table to store the service request made by patients. It can have the following fields:
   - Id: A unique identifier for the user.
   - Question: the text of question
   - Service_id: A foreign key to the service table for which the request is defined.

## API design:

I plan to use a Node.js backend with Express and a React frontend. For the database, I plan to create database model using Mongoose.

The first step is to define the API endpoints. These endpoints will correspond to the different operations I want to perform on the data, including

1. Create/Delete/Update/Get medical issue
2. Create/Delete/Update/Get care path:
3. Create/Delete/Update/Get category:
4. Create/Delete/Update/Get service:
5. Create/Delete/Update/Get request:
6. check request: check if a service can be automatically approved according to patient's symptom and service criteria and update request status.

And I'll use Express build-in routing functionality to define these endpoints.

The second step is to implement API handlers and error handling. The handler functions are responsible for processing the incoming requests, interacting with the database models, perform necessary data validation and invoking business logic required for the API operation. For the error handling, try-catch blocks can be used to handle errors that may occur during the API processing.

The third step is to enable CORS to allow communication between the Express server and React client. I plan to use the 'cors' middleware in Express to configure the necessary CORS headers and allow requests from the React client's domain.

The fourth step is to make API calls from React client and handle API responses. In the React client, 'axios' library can be used to make API calls to the server. Once the server responds to the API calls, the UI needs to be updated accordingly. Redux is efficient in managing the state of the application and updating the UI design.

The last step is to test and debug. Using tools like Postman to manually test the APIs and client-server communication works as expected.