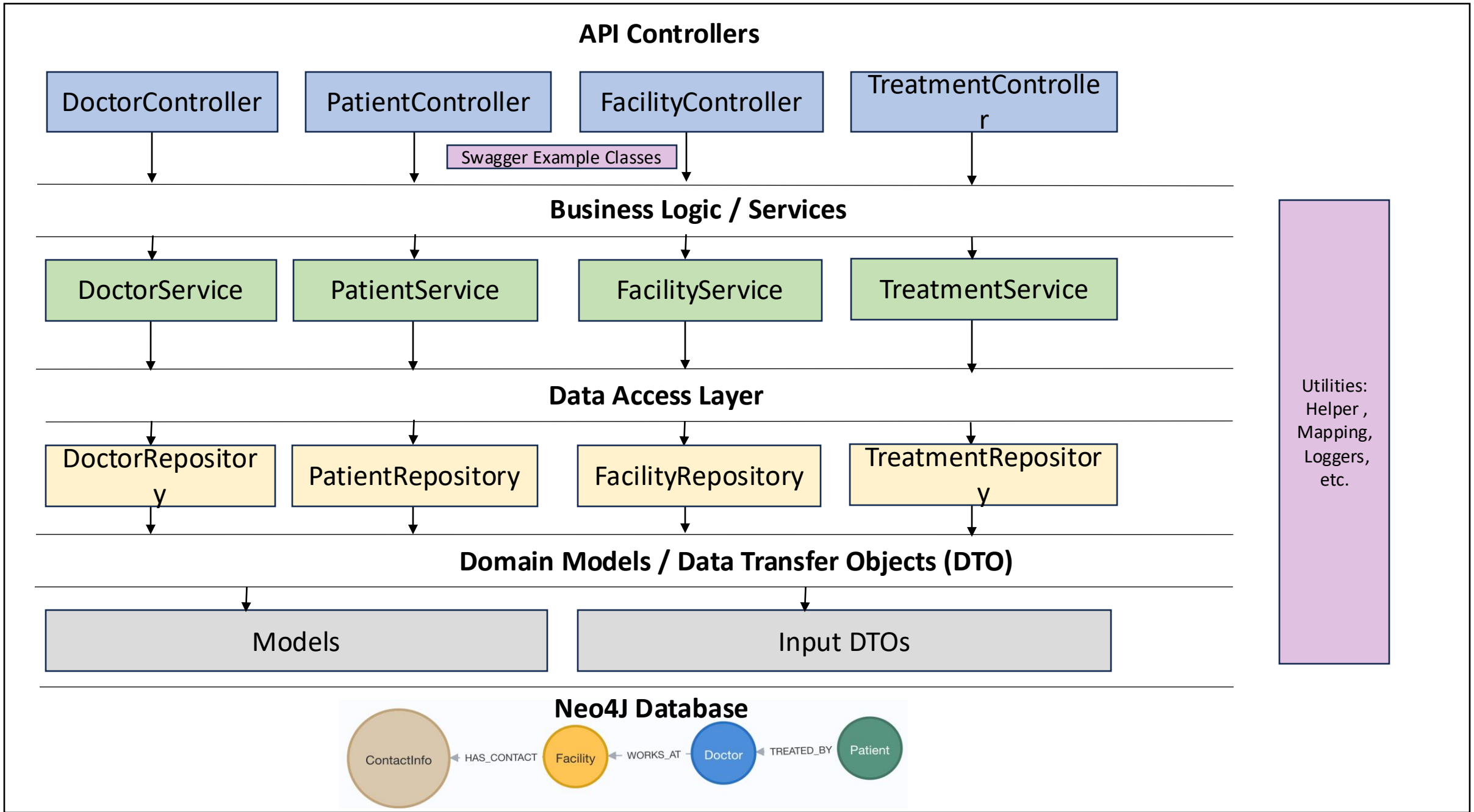
Patient Health Graph Project

Design choice & Approach

Project Overview

- **Project Purpose & Scope**
  - Develop a Healthcare Graph API that leverages graph database technology to model and manage complex relationships between key healthcare entities (patients, doctors, treatments, and facilities). The project aims to simplify the visualization and querying of interconnected healthcare data in a scalable and efficient manner.
- **Key Problem Addressed**
  - Traditional relational databases can struggle with handling intricate, many-to-many relationships common in healthcare systems. This project solves that problem by using Neo4j to efficiently store and query these relationships, enabling enhanced insights and improved decision-making for healthcare management.
- **Technologies Used**
  - **.NET:** To build the robust backend and API services.
  - **Neo4j:** As the graph database designed for efficiently handling interconnected data.
  - **Swagger:** Integrated to generate comprehensive, self-descriptive API documentation. This offers clear endpoint definitions, code examples, and simplifies the testing and integration process.
  - **LLM-based Retrieval-Augmented Generation (RAG):** Utilized to automatically generate a detailed, context-aware HealthContextSummary field. This method first retrieves relevant patient data and then uses a large language model to produce a concise summary, offering faster insights.

Architectural Overview



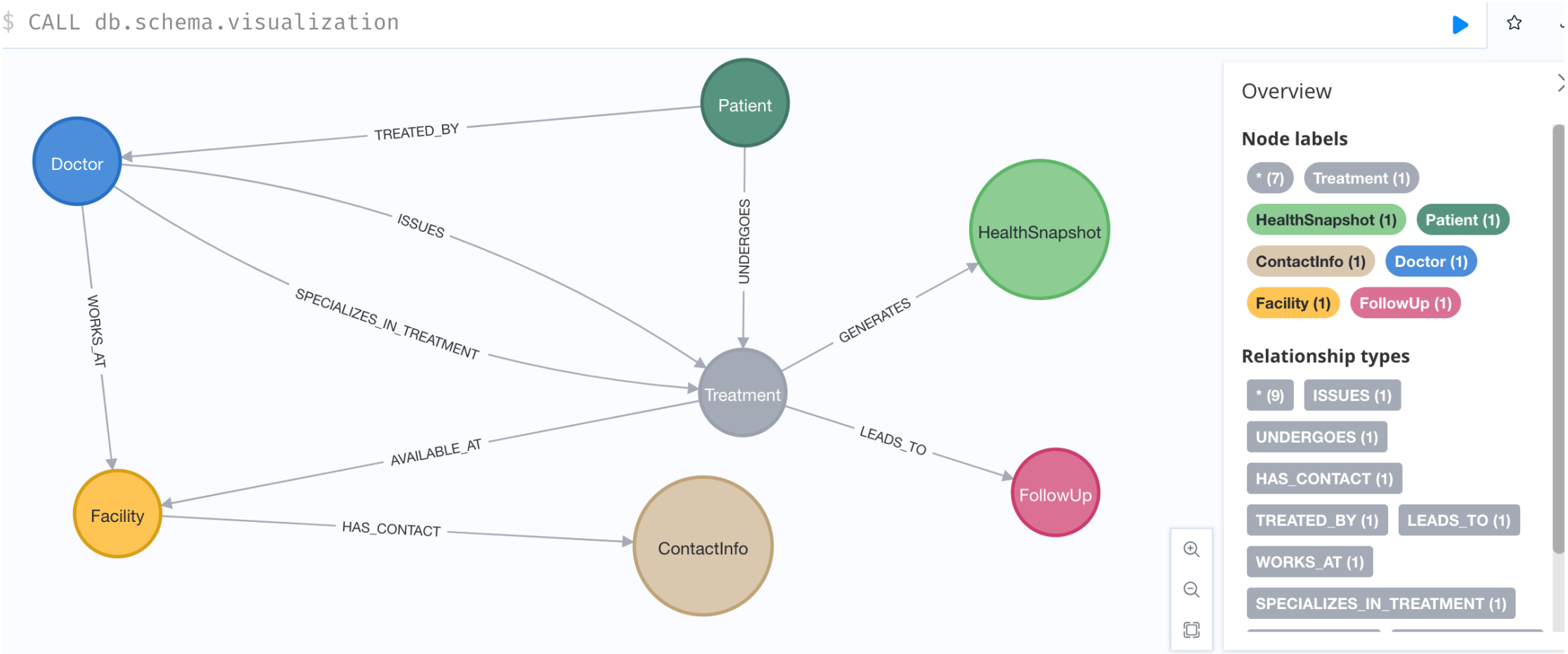
Details of the Architecture (1/2)

- **Presentation Layer (API Controllers)**
  - **API Controllers:**
    - **TreatmentController:** Handles endpoints related to treatments (e.g., creation, retrieval).
    - **DoctorController:** Manages doctor-related endpoints.
    - **PatientController:** Manages patient-related endpoints.
    - **FacilityController:** Manages facility-related endpoints.
  - **Swagger Example Classes:**
    - Located in a dedicated folder (/Examples) within the API project.
    - Provide sample payloads and documentation examples for each domain to help consumers fill the needed data (e.g., PatientExample).
- **Business Logic Layer (Services)**
  - Implements the core business rules and orchestrates calls between the presentation and data access layers.
  - Validate business-specific rules (e.g., ensuring a treatment is only created if a patient and doctor exist). – Coordinate operations among different entities (e.g., associating a treatment with the correct doctor, patient, and facility). – Simplify error handling and logging across operations.
- **Data Access Layer (Repositories)**
  - Directly interface with the Neo4j graph database.
  - Manually map raw Neo4j nodes/records into domain models and DTOs.
  - Handle nullable fields and optional data via defensive programming practices.

Details of the Architecture (2/2)

- **Domain Models and Data Transfer Objects (DTOs)**
  - **Domain Models Concepts:** Represent the core business entities and shape the data that flows between layers
  - **Domain Specific Enumerations :** encapsulate and standardize the set of allowable values for key domain concepts, such as facility types, service types, or genders. This helps ensure consistency throughout the codebase since everyone uses the same definitions instead of ad-hoc string values.
  - **Input DTOs (e.g., CreateTreatmentDTO):** Contain data validation attributes ([Required]) and guide API consumers.
- **Utilities / Common Components**
  - In a separate folder (named Utilities) that can be referenced by all layers.
- **Neo4j Graph Database:**
  - Contains the nodes and relationships representing your business entities:
    - **Nodes:** Treatment, FollowUp, HealthSnapshot, Doctor, Patient, Facility.
    - **Relationships:** E.g., a doctor issues a treatment, a treatment leads to follow-up actions, a facility is associated with a doctor, etc.

Neo4J Schema visualisation



Key Design Choices

Separation of Concerns: The project is organized into distinct layers—such as Controllers (API layer), Services (Business Logic), and Repositories (Data Access)—to ensure that each part of the system focuses on a single responsibility. This separation facilitates easier testing, maintenance, and collaboration.

SOLID Principles:

- **Single Responsibility:** Each class or module is designed to handle one specific aspect of the system, minimizing overlapping responsibilities.
- **Open/Closed:** Classes are designed to be extendable without modifying existing code, promoting stability as new features are added.
- **Liskov Substitution, Interface Segregation, Dependency Inversion:** These principles guide our design to keep dependencies abstracted and our components loosely coupled, thereby enabling flexibility and ease of integration.
- **Dependency Injection:** We use DI to manage the creation and lifetime of services and repositories, ensuring that components depend on abstractions (interfaces) rather than concrete implementations.

Generic Data Access Methods & Parameterized Operations

- **Reusable Query Methods:**
  - We've implemented generic methods (e.g., `GetAllTargetsByCriteriaAsync` and `GetAllSourcesByCriteriaAsync`) across our classes.
  - These methods accept dynamic filter criteria as parameters, enabling them to handle various query scenarios without rewriting similar logic.
- **Enhanced Maintainability & Consistency:**
  - By defining these generic methods, we reduce code duplication and increase consistency across data retrieval operations.
  - Changes or optimizations to the query logic only need to be made in one place, making the system easier to maintain and evolve.
- **Improved Performance and Security:**
  - The parameters ensure that each query is both parameterized—improving performance via query plan caching—and secure against injection attacks.

Key Design Choices

Separation of Concerns: The project is organized into distinct layers—such as Controllers (API layer), Services (Business Logic), and Repositories (Data Access)—to ensure that each part of the system focuses on a single responsibility. This separation facilitates easier testing, maintenance, and collaboration.

SOLID Principles:

- **Single Responsibility:** Each class or module is designed to handle one specific aspect of the system, minimizing overlapping responsibilities.
- **Open/Closed:** Classes are designed to be extendable without modifying existing code, promoting stability as new features are added.
- **Liskov Substitution, Interface Segregation, Dependency Inversion:** These principles guide our design to keep dependencies abstracted and our components loosely coupled, thereby enabling flexibility and ease of integration.
- **Dependency Injection:** We use DI to manage the creation and lifetime of services and repositories, ensuring that components depend on abstractions (interfaces) rather than concrete implementations.

Generic Data Access Methods & Parameterized Operations

- **Reusable Query Methods:**
  - We've implemented generic methods (e.g., `GetAllTargetsByCriteriaAsync` and `GetAllSourcesByCriteriaAsync`) across our classes.
  - These methods accept dynamic filter criteria as parameters, enabling them to handle various query scenarios without rewriting similar logic.
- **Enhanced Maintainability & Consistency:**
  - By defining these generic methods, we reduce code duplication and increase consistency across data retrieval operations.
  - Changes or optimizations to the query logic only need to be made in one place, making the system easier to maintain and evolve.
- **Improved Performance and Security:**
  - The parameters ensure that each query is both parameterized, improving performance via query plan caching and secure against injection attacks.

Utilizing Constraints and Indexes in Neo4J database: ensures data integrity and accelerates query performance.

Key Design Perspectives

Enhance Security

- Use fully parameterized queries to prevent injection attacks and optimize query caching.
- Implement robust authentication (e.g., JWT, RBAC) to control access.
- Manage sensitive settings via secure configuration and secrets managers.
- Enable data encryption in transit and at rest.
- Enforce strict input validation and output encoding.

Improve Exception and Error Handling

- Centralize error handling with global middleware returning standardized error formats.
- Integrate structured logging and proactive monitoring for anomalies.
- Leverage custom exceptions and resilience patterns (retry, circuit breaker) for robust operations.

Refine Services and Business Logic

- Encapsulate business logic and business semantics in dedicated service layers.
- Develop generic, reusable data access methods.

Optimize the Neo4j Data Model

- Enforce constraints and use indexes to ensure data integrity and speed query performance.
- Adjust relationship granularity based on query patterns.