Software Requirements Specification

Family Service

Version 1.0.0

Copyright and License

Table of Contents

---

1. Introduction

1.1 Purpose

This Software Requirements Specification (SRS) document describes the functional and non-functional requirements for the Family Service application. It is intended to be used by the development team to implement the system according to the specified requirements.

1.2 Scope

The Family Service is a backend application that provides APIs for managing family relationships, specifically parents and their children. The system allows creating, reading, updating, and deleting parent and child records, as well as managing the relationships between them.

The system exposes a GraphQL API for clients to interact with the service. It uses a hexagonal architecture pattern to separate the core business logic from external concerns such as databases and API protocols.

1.3 Definitions, Acronyms, and Abbreviations

- **API**: Application Programming Interface
- **CRUD**: Create, Read, Update, Delete
- **DI**: Dependency Injection
- **GraphQL**: A query language for APIs and a runtime for executing those queries
- **Hexagonal Architecture**: An architectural pattern that isolates the core business logic from external concerns
- **JWT**: JSON Web Token
- **OIDC**: OpenID Connect
- **REST**: Representational State Transfer
- **UUID**: Universally Unique Identifier

1.4 References

- Hexagonal Architecture (Ports and Adapters) pattern
- GraphQL Specification
- OpenTelemetry Specification
- MongoDB Documentation
- PostgreSQL Documentation

1.5 Overview

The remainder of this document is organized as follows:

- Section 2 provides an overall description of the system, including product perspective, functions, user characteristics, constraints, and assumptions.
- Section 3 details the specific requirements, including external interface requirements, functional requirements, performance requirements, design constraints, and system attributes.
- Section 4 describes the architecture of the system, including the hexagonal architecture pattern, domain model, use cases, sequence diagrams, and state diagrams.
- Section 5 details the configuration options for the system.
- Section 6 contains appendices with additional information.

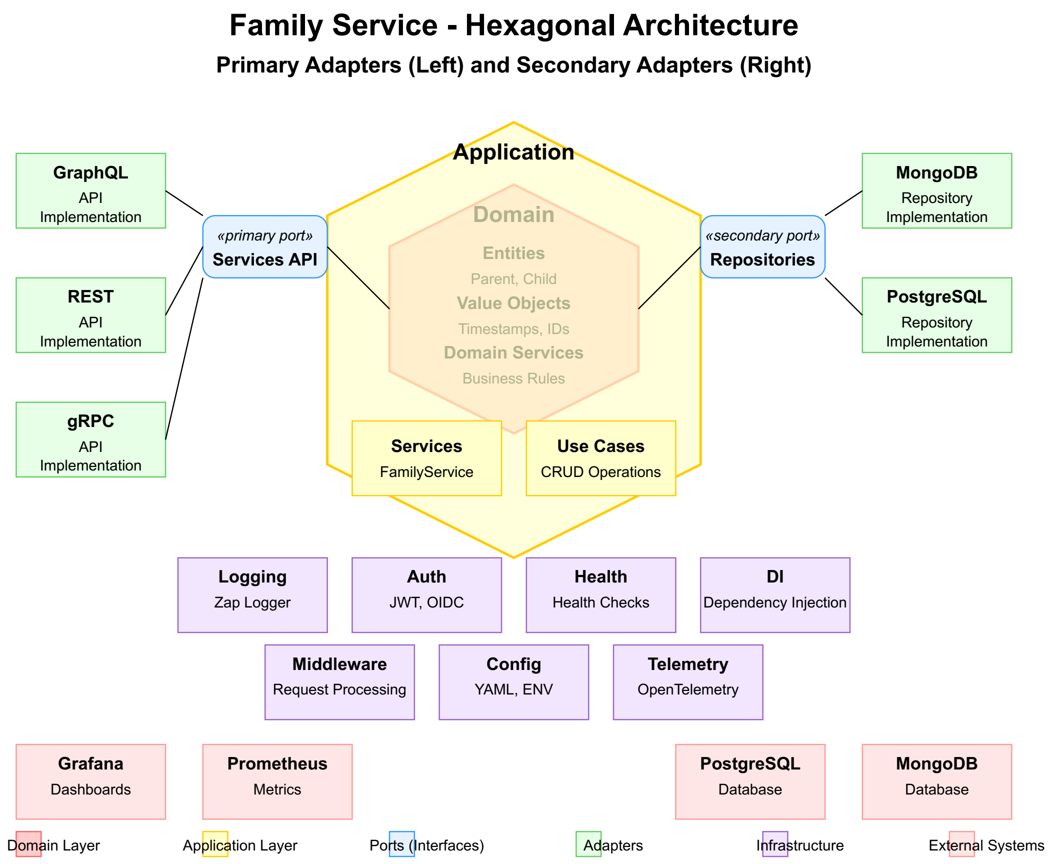
---

2. Overall Description

2.1 Product Perspective

The Family Service is a standalone backend service that provides APIs for managing family relationships. It is designed to be used by various client applications, such as web applications, mobile applications, or other services.

The service follows a hexagonal architecture pattern, which separates the core business logic from external concerns such as databases and API protocols. This allows the service to be easily adapted to different technologies and environments.



Hexagonal Architecture

2.2 Product Functions

The Family Service provides the following main functions:

1. **Parent Management**: Create, read, update, and delete parent records.
2. **Child Management**: Create, read, update, and delete child records.
3. **Relationship Management**: Associate children with parents and manage these relationships.

4. **Health Monitoring**: Provide health check endpoints for monitoring the service's health.

2.3 User Characteristics

The primary users of the Family Service are:

1. **Client Applications**: Web applications, mobile applications, or other services that need to manage family relationships.
2. **System Administrators**: Responsible for deploying, configuring, and monitoring the service.

2.4 Constraints

The Family Service has the following constraints:

1. **Technology Stack**: The service is implemented in Go and uses GraphQL for API communication.
2. **Database Support**: The service supports MongoDB and PostgreSQL as database backends.
3. **Authentication**: The service requires OIDC authentication for API access.
4. **Deployment**: The service is designed to be deployed in a Docker container.

2.5 Assumptions and Dependencies

The Family Service assumes the following:

1. **Database Availability**: MongoDB or PostgreSQL is available and accessible.
2. **Authentication Provider**: An OIDC provider is available for authentication.
3. **Monitoring Infrastructure**: Prometheus is available for metrics collection.

---

3. Specific Requirements

3.1 External Interface Requirements

3.1.1 User Interfaces

The Family Service does not provide a direct user interface. It exposes a GraphQL API that client applications can use to interact with the service.

3.1.2 Hardware Interfaces

The Family Service does not have any specific hardware interface requirements.

3.1.3 Software Interfaces

The Family Service interfaces with the following software systems:

1. **MongoDB**: For document-based storage of family data.
2. **PostgreSQL**: For relational storage of family data.
3. **Prometheus**: For metrics collection and monitoring.
4. **OIDC Provider**: For authentication and authorization.

3.1.4 Communication Interfaces

The Family Service communicates with clients and other systems using the following protocols:

1. **HTTP/HTTPS**: For GraphQL API communication.
2. **TCP/IP**: For database communication.

3.2 Functional Requirements

3.2.1 Parent Management

1. **Create Parent**
   - The system shall allow creating a new parent with first name, last name, email, and birth date.
   - The system shall validate the input data before creating a parent.
   - The system shall generate a unique identifier (UUID) for each parent.
   - The system shall record creation and update timestamps for each parent.
2. **Get Parent by ID**
   - The system shall allow retrieving a parent by their unique identifier.
   - The system shall return an error if the parent is not found.
3. **Update Parent**
   - The system shall allow updating a parent's first name, last name, email, and birth date.
   - The system shall validate the input data before updating a parent.
   - The system shall update the update timestamp when a parent is updated.
4. **Delete Parent**
   - The system shall allow marking a parent as deleted (soft delete).
   - The system shall update the update timestamp and set the delete timestamp when a parent is deleted.
   - The system shall not allow deleting a parent with associated children.
5. **List Parents**
   - The system shall allow listing all parents with pagination.
   - The system shall allow filtering parents by various criteria.
   - The system shall not include deleted parents in the list by default.
6. **Count Parents**
   - The system shall allow counting the number of parents based on filter criteria.

3.2.2 Child Management

1. **Create Child**
    - The system shall allow creating a new child with first name, last name, birth date, and parent ID.
    - The system shall validate the input data before creating a child.
    - The system shall generate a unique identifier (UUID) for each child.
    - The system shall record creation and update timestamps for each child.
    - The system shall verify that the specified parent exists.
2. **Get Child by ID**
    - The system shall allow retrieving a child by their unique identifier.
    - The system shall return an error if the child is not found.
3. **Update Child**
    - The system shall allow updating a child’s first name, last name, and birth date.
    - The system shall validate the input data before updating a child.
    - The system shall update the update timestamp when a child is updated.
4. **Delete Child**
    - The system shall allow marking a child as deleted (soft delete).
    - The system shall update the update timestamp and set the delete timestamp when a child is deleted.
    - The system shall remove the child from the parent’s list of children.
5. **List Children by Parent ID**
    - The system shall allow listing all children for a specific parent with pagination.
    - The system shall not include deleted children in the list by default.
6. **List Children**
    - The system shall allow listing all children with pagination.
    - The system shall allow filtering children by various criteria.
    - The system shall not include deleted children in the list by default.
7. **Count Children**
    - The system shall allow counting the number of children based on filter criteria.

3.2.3 Relationship Management

1. **Add Child to Parent**
    - The system shall allow adding a child to a parent’s list of children.
    - The system shall verify that both the parent and child exist.
    - The system shall update the parent’s update timestamp.
2. **Remove Child from Parent**
    - The system shall allow removing a child from a parent’s list of children.

    - The system shall verify that both the parent and child exist.
    - The system shall update the parent's update timestamp.

3.2.4 Health Monitoring

1. **Health Check**
    - The system shall provide a health check endpoint that returns the health status of the service.
    - The health check shall verify the database connection.

3.3 Performance Requirements

1. **Response Time**
    - The system shall respond to API requests within 10 seconds.
    - The system shall respond to health check requests within 5 seconds.
2. **Throughput**
    - The system shall support at least 100 concurrent API requests.
3. **Resource Utilization**
    - The system shall operate within the following resource limits:
        - CPU: 0.5 CPUs
        - Memory: 512 MB

3.4 Design Constraints

1. **Architectural Pattern**
    - The system shall follow the hexagonal architecture pattern.
    - The system shall separate the core business logic from external concerns.
2. **Programming Language**
    - The system shall be implemented in Go.
3. **API Protocol**
    - The system shall use GraphQL for API communication.
4. **Database**
    - The system shall support both MongoDB and PostgreSQL as database backends.
    - The system shall use a repository pattern to abstract database access.
5. **Authentication**
    - The system shall use OIDC for authentication.
6. **Containerization**
    - The system shall be deployable as a Docker container.

3.5 System Attributes

3.5.1 Reliability

1. The system shall handle errors gracefully and provide meaningful error messages.
2. The system shall implement retry mechanisms for database operations.

3. The system shall implement graceful shutdown to prevent data loss.

3.5.2 Availability

1. The system shall be designed for high availability.
2. The system shall provide health check endpoints for monitoring.

3.5.3 Security

1. The system shall require authentication for API access.
2. The system shall validate and sanitize all input data.
3. The system shall implement proper error handling to prevent information leakage.

3.5.4 Maintainability

1. The system shall follow clean code principles.
2. The system shall have comprehensive test coverage.
3. The system shall use dependency injection for better testability.

3.5.5 Portability

1. The system shall be deployable in different environments (development, testing, production).
2. The system shall use configuration files and environment variables for environment-specific settings.

3.6 Other Requirements

3.6.1 Logging

1. The system shall log all API requests and responses.
2. The system shall log all errors with appropriate context.
3. The system shall support different log levels (debug, info, warn, error).

3.6.2 Metrics

1. The system shall expose metrics for monitoring.
2. The system shall use Prometheus for metrics collection.

3.6.3 Tracing

1. The system shall support distributed tracing.
2. The system shall use OpenTelemetry for tracing.

---

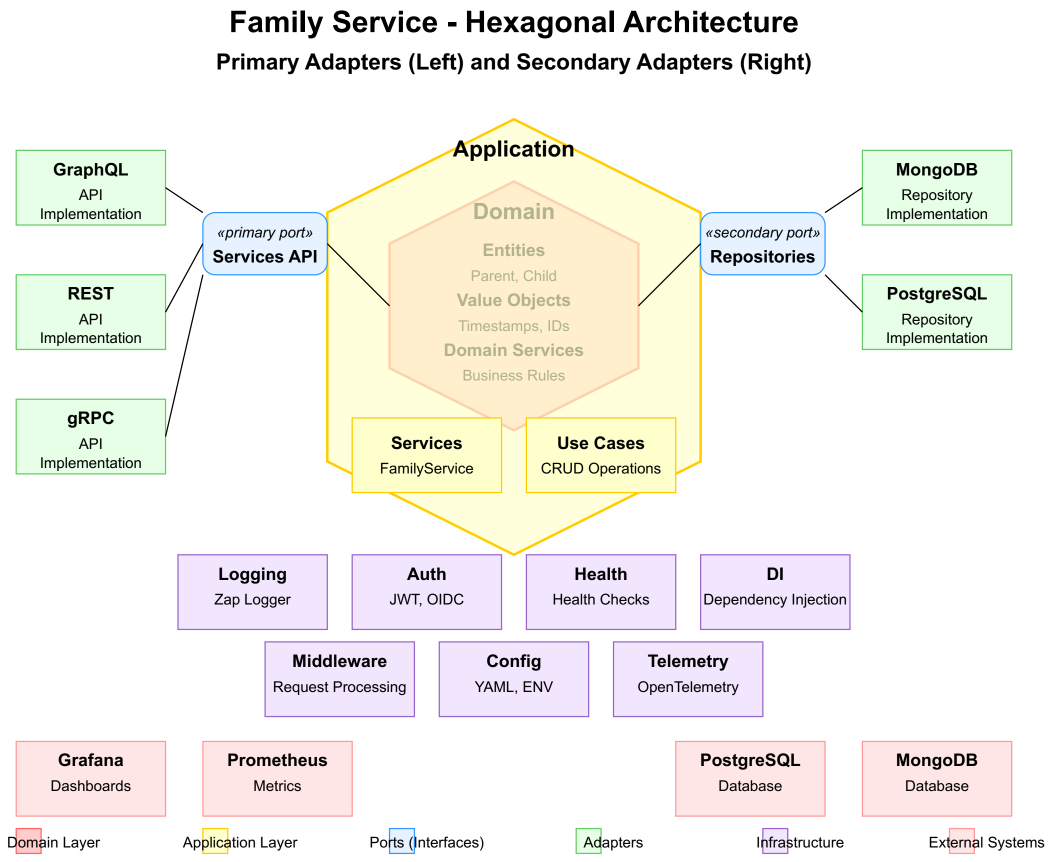
4. Architecture

4.1 Hexagonal Architecture

The Family Service follows the hexagonal architecture pattern, also known as the ports and adapters pattern. This architectural style isolates the core business logic from external concerns such as databases and API protocols.

The architecture consists of the following layers:

1. **Domain Layer**: Contains the core business entities and business rules.
2. **Application Layer**: Contains the use cases and application services.
3. **Ports Layer**: Defines the interfaces for interacting with the application.
4. **Adapters Layer**: Implements the interfaces defined in the ports layer.
5. **Infrastructure Layer**: Provides technical capabilities to the system.



Hexagonal Architecture

4.1.1 Domain Layer

The domain layer contains the core business entities and business rules. It is independent of any external concerns and defines the domain model for the system.

4.1.2 Application Layer

The application layer contains the use cases and application services. It orchestrates the flow of data to and from the domain entities and applies domain rules to achieve the goals of a use case.

4.1.3 Ports Layer

The ports layer defines the interfaces for interacting with the application. It includes:

- **Primary Ports**: Interfaces that allow the application to be driven by external actors.

- **Secondary Ports**: Interfaces that allow the application to drive external systems.

4.1.4 Adapters Layer

The adapters layer implements the interfaces defined in the ports layer. It includes:

- **Primary Adapters**: Implement the primary ports and are driven by external actors (e.g., GraphQL, REST).
- **Secondary Adapters**: Implement the secondary ports and are driven by the application (e.g., repositories).

4.1.5 Infrastructure Layer

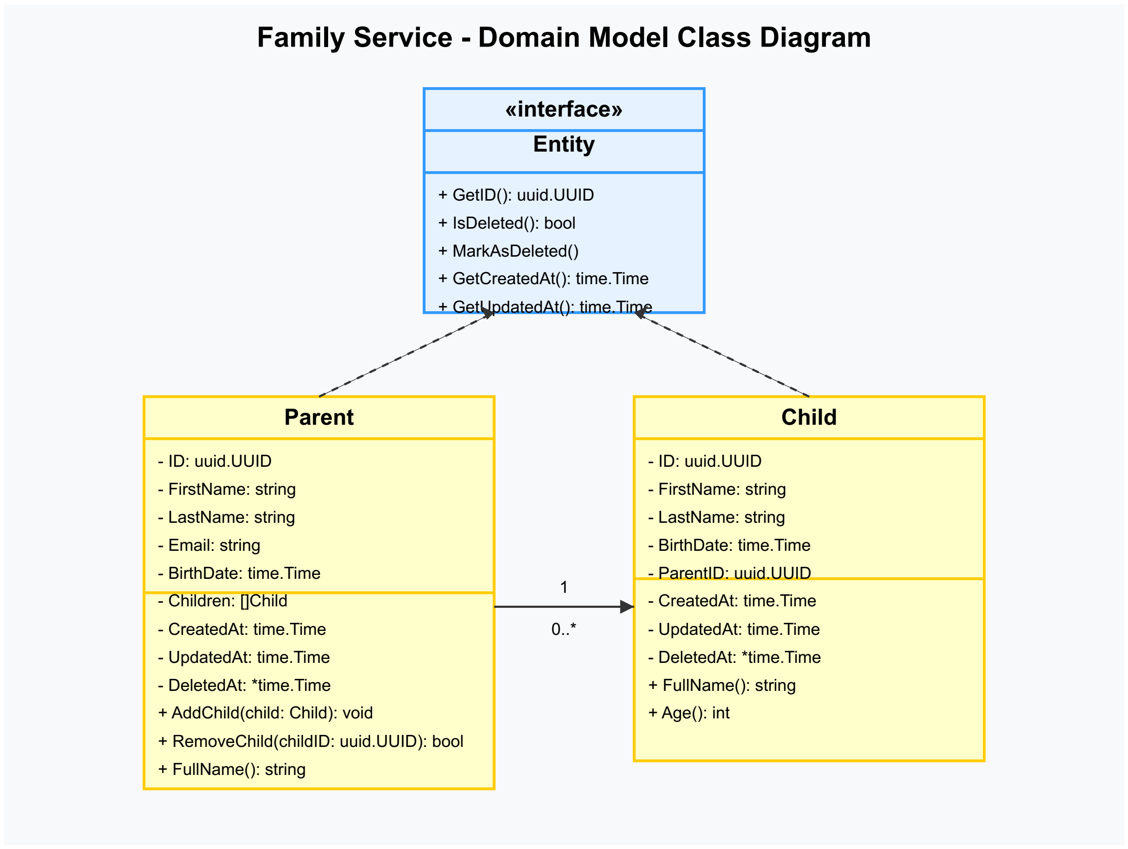
The infrastructure layer provides technical capabilities to the system, such as:

- **Logging**: For recording events and errors.
- **Authentication**: For verifying user identity.
- **Configuration**: For managing application settings.
- **Dependency Injection**: For managing object creation and dependencies.
- **Health Monitoring**: For checking the health of the system.
- **Telemetry**: For collecting metrics and traces.

4.2 Domain Model

The domain model for the Family Service consists of the following entities:

1. **Entity**: The base interface for all domain entities.
2. **Parent**: Represents a parent entity.
3. **Child**: Represents a child entity.



Class Diagram

4.2.1 Entity Interface

The Entity interface defines the common behavior for all domain entities:

- **GetID()**: Returns the entity's unique identifier.
- **IsDeleted()**: Checks if the entity is marked as deleted.
- **MarkAsDeleted()**: Marks the entity as deleted.
- **GetCreatedAt()**: Returns the entity's creation timestamp.
- **GetUpdatedAt()**: Returns the entity's last update timestamp.
- **GetDeletedAt()**: Returns the entity's deletion timestamp, if any.

4.2.2 Parent Entity

The Parent entity represents a parent in the system:

- **ID**: The unique identifier for the parent.
- **FirstName**: The parent's first name.
- **LastName**: The parent's last name.
- **Email**: The parent's email address.
- **BirthDate**: The parent's date of birth.
- **Children**: A list of children associated with the parent.

- **CreatedAt**: The timestamp when the parent was created.
- **UpdatedAt**: The timestamp when the parent was last updated.
- **DeletedAt**: The timestamp when the parent was marked as deleted, if any.

Methods: - **AddChild()**: Adds a child to the parent's list of children. - **RemoveChild()**: Removes a child from the parent's list of children. - **FullName()**: Returns the parent's full name.

4.2.3 Child Entity

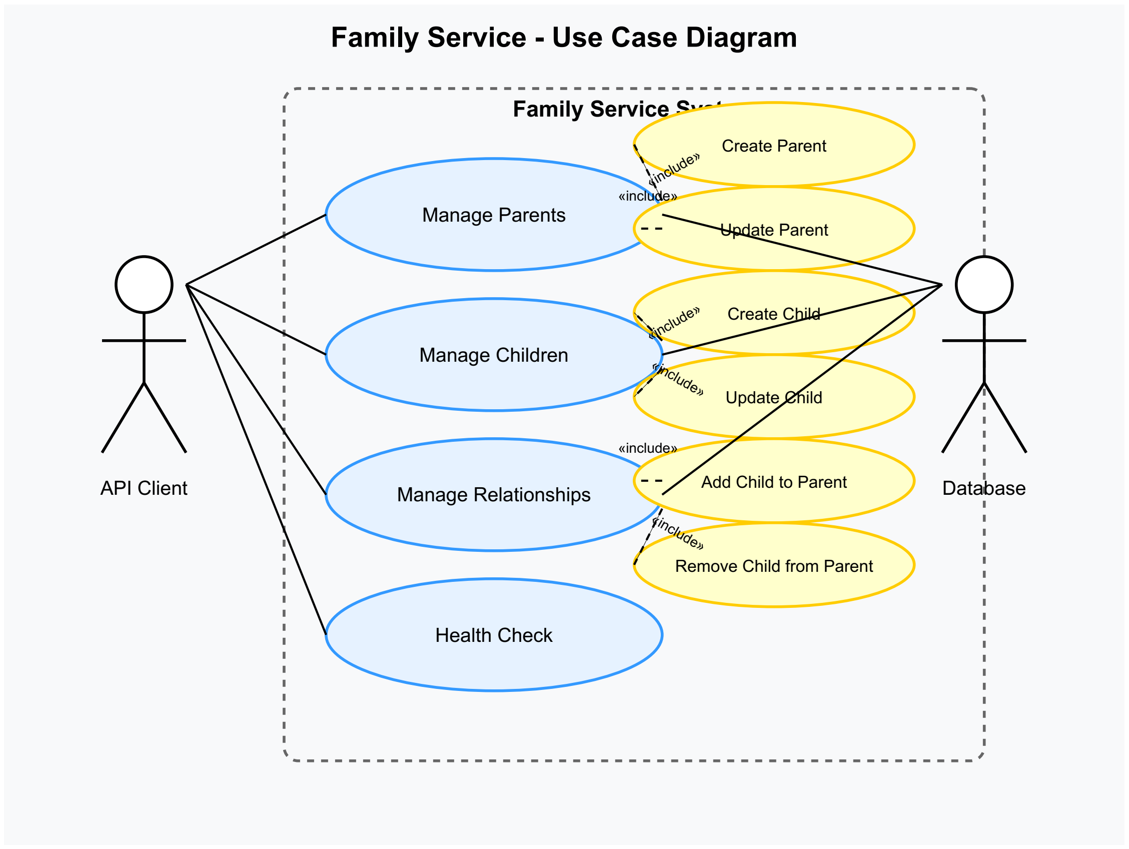
The Child entity represents a child in the system:

- **ID**: The unique identifier for the child.
- **FirstName**: The child's first name.
- **LastName**: The child's last name.
- **BirthDate**: The child's date of birth.
- **ParentID**: The unique identifier of the parent this child belongs to.
- **CreatedAt**: The timestamp when the child was created.
- **UpdatedAt**: The timestamp when the child was last updated.
- **DeletedAt**: The timestamp when the child was marked as deleted, if any.

Methods: - **FullName()**: Returns the child's full name. - **Age()**: Calculates the child's current age based on their birth date.

4.3 Use Cases

The Family Service supports the following use cases:



Use Case Diagram

4.3.1 Parent Management

1. **Create Parent**: Create a new parent with personal information.
2. **Get Parent by ID**: Retrieve a parent by their unique identifier.
3. **Update Parent**: Update a parent's personal information.
4. **Delete Parent**: Mark a parent as deleted.
5. **List Parents**: List all parents with pagination and filtering.
6. **Count Parents**: Count the number of parents based on filter criteria.

4.3.2 Child Management

1. **Create Child**: Create a new child with personal information and associate with a parent.
2. **Get Child by ID**: Retrieve a child by their unique identifier.
3. **Update Child**: Update a child's personal information.
4. **Delete Child**: Mark a child as deleted.
5. **List Children by Parent ID**: List all children for a specific parent.
6. **List Children**: List all children with pagination and filtering.
7. **Count Children**: Count the number of children based on filter criteria.

4.3.3 Relationship Management

1. **Add Child to Parent**: Associate a child with a parent.

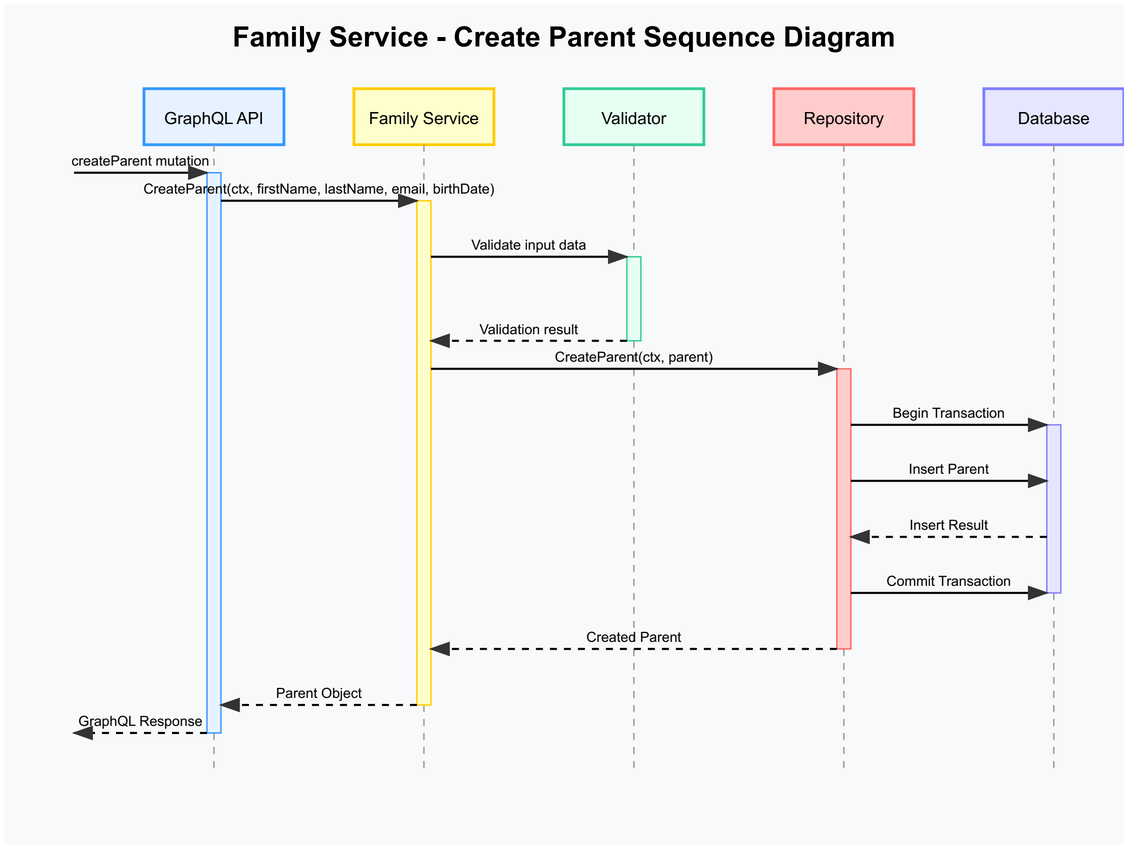
2. **Remove Child from Parent**: Remove a child from a parent.

4.3.4 Health Monitoring

1. **Health Check**: Check the health of the system.

4.4 Sequence Diagrams

The following sequence diagram illustrates the flow of the “Create Parent” use case:



Sequence Diagram

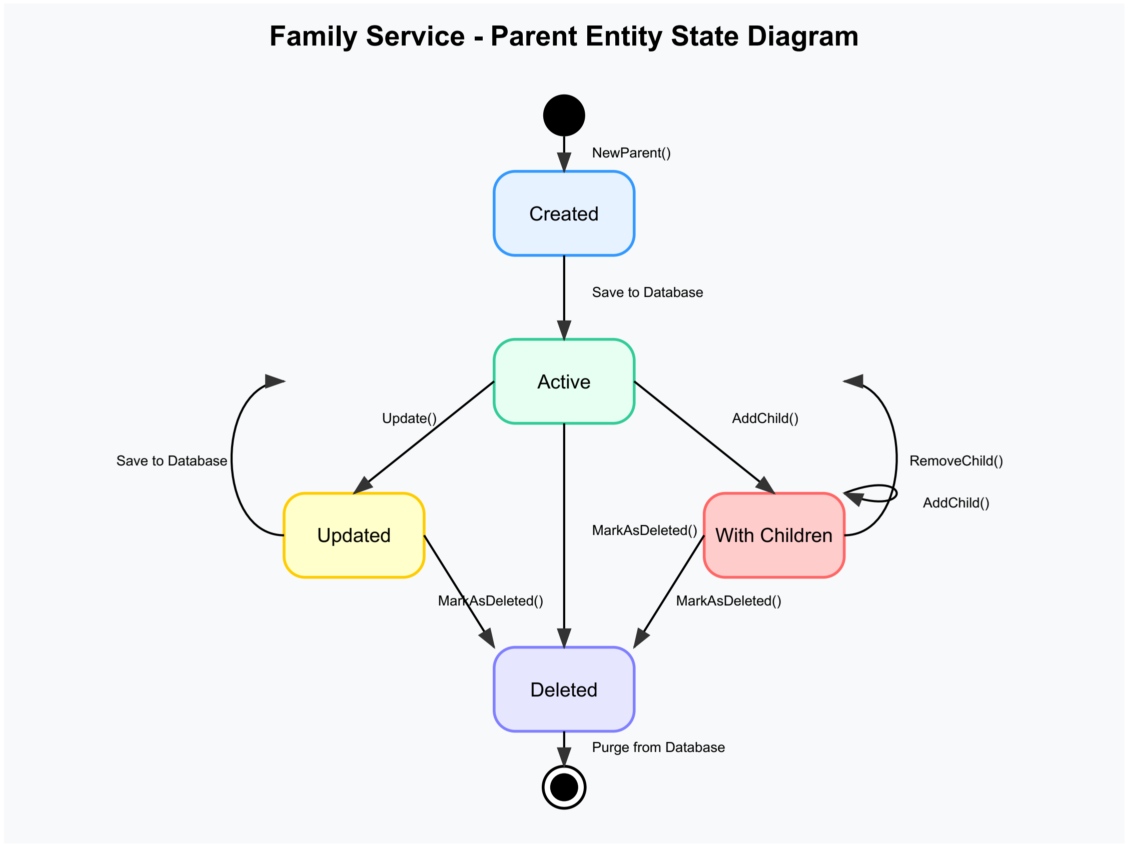
The sequence of events is as follows:

1. The client sends a createParent mutation to the GraphQL API.
2. The GraphQL API calls the Family Service’s CreateParent method.
3. The Family Service validates the input data using the Validator.
4. The Family Service creates a new Parent entity.
5. The Family Service calls the Repository to save the parent.
6. The Repository begins a transaction with the Database.
7. The Repository inserts the parent into the Database.
8. The Database returns the insert result to the Repository.
9. The Repository commits the transaction.
10. The Repository returns the created parent to the Family Service.

11. The Family Service returns the parent to the GraphQL API.
12. The GraphQL API returns the response to the client.

4.5 State Diagrams

The following state diagram illustrates the lifecycle of a Parent entity:



State Diagram

The Parent entity can be in the following states:

1. **Created**: The parent has been created in memory but not yet saved to the database.
2. **Active**: The parent has been saved to the database and is active.
3. **Updated**: The parent's information has been updated.
4. **With Children**: The parent has one or more children associated with them.
5. **Deleted**: The parent has been marked as deleted (soft delete).

The transitions between states are triggered by the following events:

1. **NewParent()**: Creates a new parent entity.
2. **Save to Database**: Saves the parent to the database.
3. **Update()**: Updates the parent's information.
4. **AddChild()**: Adds a child to the parent.

5. **RemoveChild()**: Removes a child from the parent.
6. **MarkAsDeleted()**: Marks the parent as deleted.
7. **Purge from Database**: Permanently removes the parent from the database.

---

5. Configuration

The Family Service is configured using a YAML configuration file and environment variables. The configuration is loaded at startup and used to configure the various components of the system.

5.1 Application Configuration

The application configuration includes general settings for the application:

```yaml
app:
  version: 1.0.0
```

- **version**: The version of the application.

5.2 Database Configuration

The database configuration includes settings for connecting to the database:

```yaml
database:
  mongodb:
    connection_timeout: 10s
    disconnect_timeout: 50s
    index_timeout: 10s
    migration_timeout: 30s
    ping_timeout: 30s
    uri: 
mongodb://${MONGODB_ROOT_USERNAME}:${MONGODB_ROOT_PASSWORD}@mongodb:27017/fam
ily_service?authSource=admin&tlsMode=disable
  postgres:
    dsn: 
postgres://${POSTGRESQL_USERNAME}:${POSTGRESQL_PASSWORD}@postgresql:5432/fami
ly_service?sslmode=disable
    migration_timeout: 30s
  type: mongodb
```

- **mongodb.connection_timeout**: The timeout for connecting to MongoDB (10 seconds).
- **mongodb.disconnect_timeout**: The timeout for disconnecting from MongoDB (50 seconds).
- **mongodb.index_timeout**: The timeout for creating indexes in MongoDB (10 seconds).

- **mongodb.migration_timeout**: The timeout for running migrations in MongoDB (30 seconds).
- **mongodb.ping_timeout**: The timeout for pinging MongoDB (30 seconds).
- **mongodb.uri**: The URI for connecting to MongoDB.
- **postgres.dsn**: The data source name for connecting to PostgreSQL.
- **postgres.migration_timeout**: The timeout for running migrations in PostgreSQL (30 seconds).
- **type**: The type of database to use (mongodb or postgres).

5.3 Server Configuration

The server configuration includes settings for the HTTP server:

```
server:
  health_endpoint: /health
  idle_timeout: 12s
  port: '8080'
  read_timeout: 10s
  shutdown_timeout: 10s
  write_timeout: 10s
```

- **health_endpoint**: The endpoint for health checks (/health).
- **idle_timeout**: The maximum amount of time to wait for the next request when keep-alives are enabled (12 seconds).
- **port**: The port to listen on (8080).
- **read_timeout**: The maximum duration for reading the entire request, including the body (10 seconds).
- **shutdown_timeout**: The maximum duration to wait for the server to shut down gracefully (10 seconds).
- **write_timeout**: The maximum duration before timing out writes of the response (10 seconds).

5.4 Telemetry Configuration

The telemetry configuration includes settings for metrics and tracing:

```
telemetry:
  shutdown_timeout: 5s
  exporters:
    metrics:
      prometheus:
        enabled: true
        listen: 0.0.0.0:8080
        path: /metrics
```

- **shutdown_timeout**: The maximum duration to wait for telemetry to shut down gracefully (5 seconds).
- **exporters.metrics.prometheus.enabled**: Whether to enable Prometheus metrics (true).
- **exporters.metrics.prometheus.listen**: The address to listen on for Prometheus metrics (0.0.0.0:8080).
- **exporters.metrics.prometheus.path**: The path for Prometheus metrics (/metrics).

5.5 Authentication Configuration

The authentication configuration includes settings for OIDC authentication:

```
auth:
  oidc_timeout: 30s
```

- **oidc_timeout**: The timeout for OIDC operations (30 seconds).

5.6 Logging Configuration

The logging configuration includes settings for logging:

```
log:
  development: true
  level: debug
```

- **development**: Whether to use development mode for logging (true).
- **level**: The log level (debug).

5.7 Feature Flags

The feature flags configuration includes settings for enabling or disabling features:

```
features:
  use_generics: true
```

- **use_generics**: Whether to use generics in the code (true).

---

6. Appendices

Appendix A: Glossary

- **Entity**: A domain object with a unique identity.
- **Repository**: A component that provides data access and persistence for entities.
- **Use Case**: A specific business goal or task that the system must accomplish.
- **Port**: An interface that defines how the application interacts with external systems.
- **Adapter**: An implementation of a port that connects the application to external systems.

- **Soft Delete**: A technique where records are marked as deleted rather than being physically removed from the database.

Appendix B: Analysis Models

The analysis models for the Family Service include:

1. **Domain Model**: Defines the core business entities and their relationships.
2. **Use Case Model**: Defines the use cases supported by the system.
3. **Sequence Diagrams**: Illustrate the flow of interactions between components.
4. **State Diagrams**: Illustrate the lifecycle of entities.

Appendix C: Issues List

No known issues at this time.