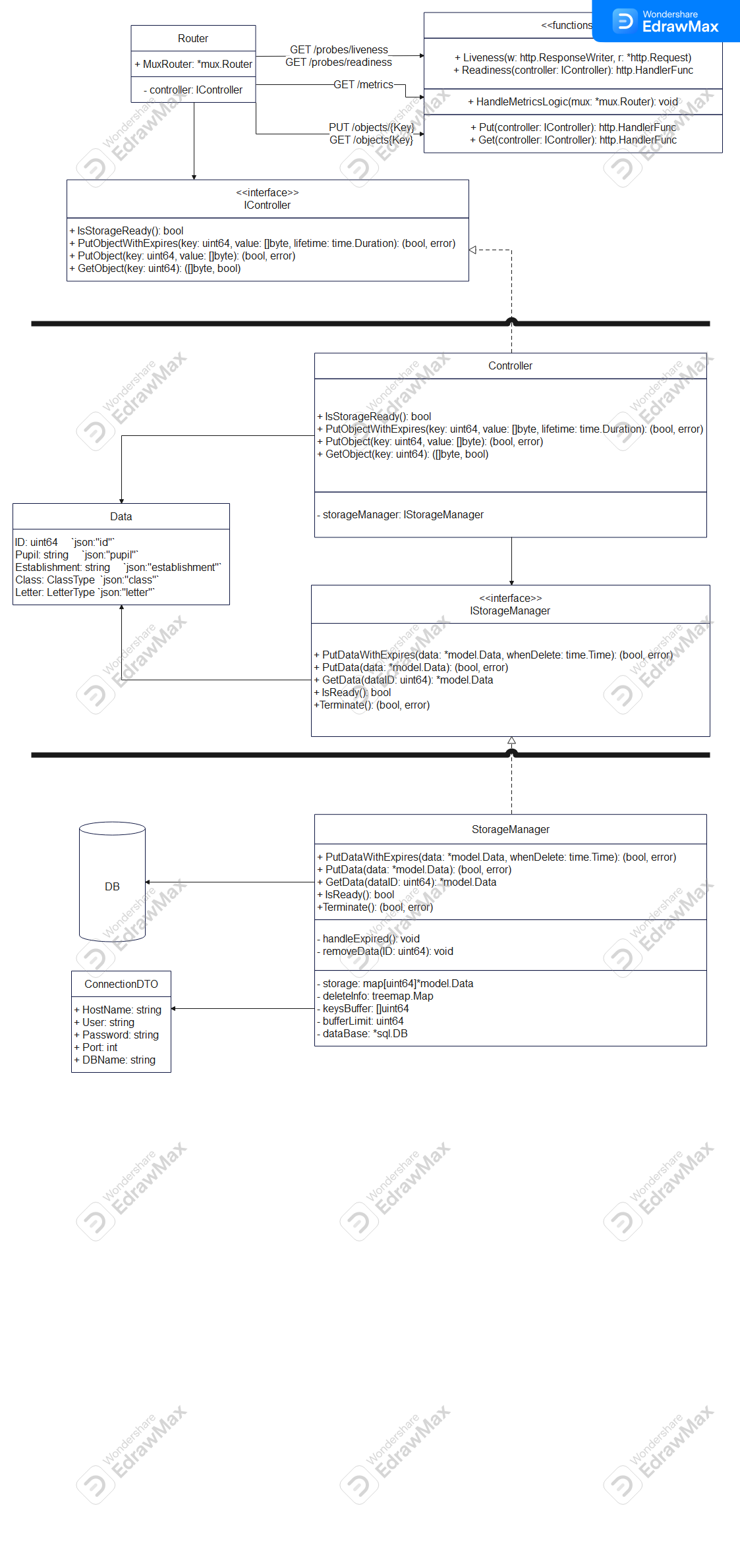
**Описание компонентов ПО**

Описываемое программное обеспечение представляет собой серверную часть клиент-серверного приложения по ведению реестра всех школьников города N на текущий год. Клиентская часть взаимодействует с серверной посредством отправки только лишь двух запросов:

* PUT /objects/{key};
* GET /objects/{key}.

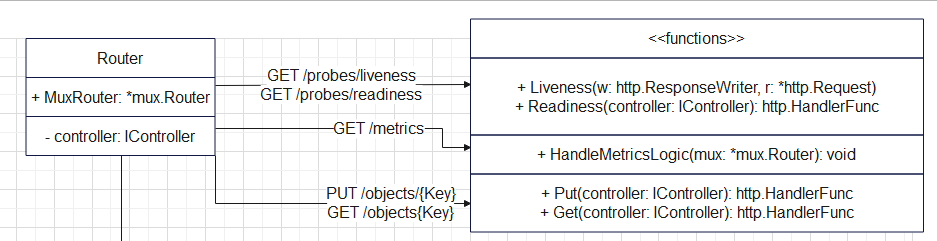
В свою очередь, в реализации серверной части выделено три слоя:

* транспортный слой;
* слой бизнес-логики;
* слой хранения.

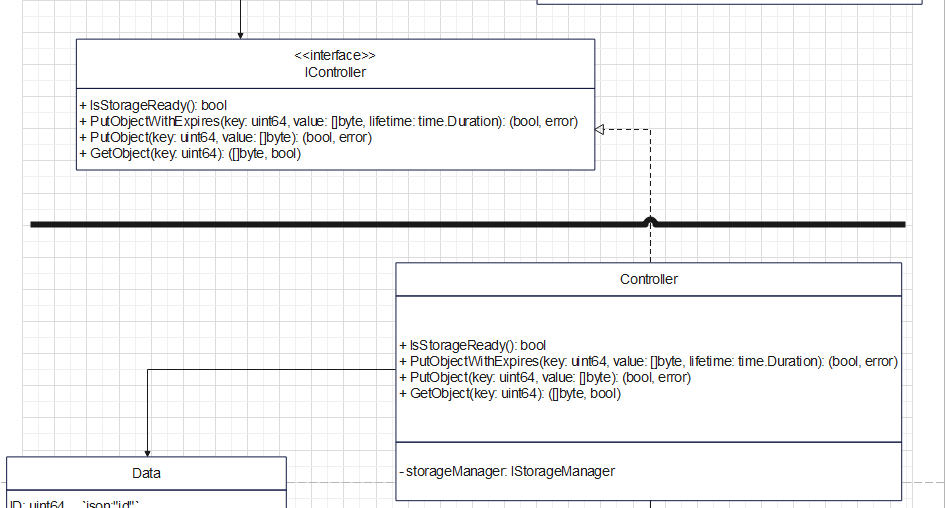
  
**Рисунок 1 – UML-диаграмма серверной части приложения**

**Транспортный слой**

Пакет, инкапсулирующий логику транспортного слоя, - *rest* (расположение: «*/internal/transport/rest*») Ключевым компонентом, отражающим суть транспортного слоя, является класс *Router*. Помимо обработки входящих запросов от клиентской стороны, данная сущность отвечает за свойства «liveness» и «readiness» для возможности развертывания сервиса в *Kubernetes*, а также предоставляет функционал «экспортера» для анализа метрик через *Prometheus*.

  
**Рисунок 2 – UML-описание класса Router**

*Router* – входная точка для клиентской стороны, перенаправляющая каждый запрос в ту часть слоя бизнес-логики, где запрос должен быть обработан. Чтобы создать объект класса Router, необходим инициализированный объект интерфейса *IController* (в качестве параметра функции *CreateRouter*, создающей и инициализирующей объект класса *Router*). Интерфейс *IConroller* описан в пакете *transport*. Так, граница взаимодействия между пакетами *rest* и *service* (между транспортным и бизнес слоями соответственно) проходит через интерфейс.

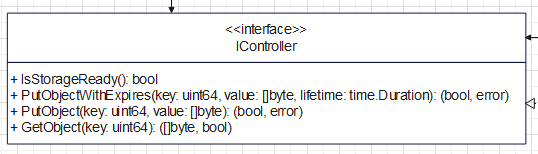
  
**Рисунок 3 – Иллюстрация Interface Segregation принципа между транспортным слоем и слоем бизнес-логики**

**Чтение объектов из хранилища.**

Запросы от клиентской стороны, относящиеся к чтению объекта из хранилища или к обновлению хранилища, проводятся *Router*-ом к посредством функций *Put* и *Get*, работающих с интерфейсом *IController* и с клиентскими запросами.

**Проверка работоспособности ПО.**

Запросы, связанные с проверкой работоспособности, обрабатываются посредством функций *Liveness* и *Readiness* пакета *rest*. Так же, как и функции *Put*/*Get*, суть их реализации напрямую увязана с интерфейсом *IController*.

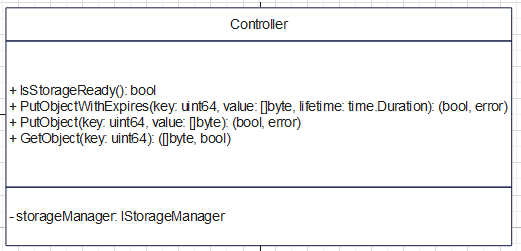
  
**Рисунок 4– UML-описание интерфейса IController**

**Сбор метрик.**

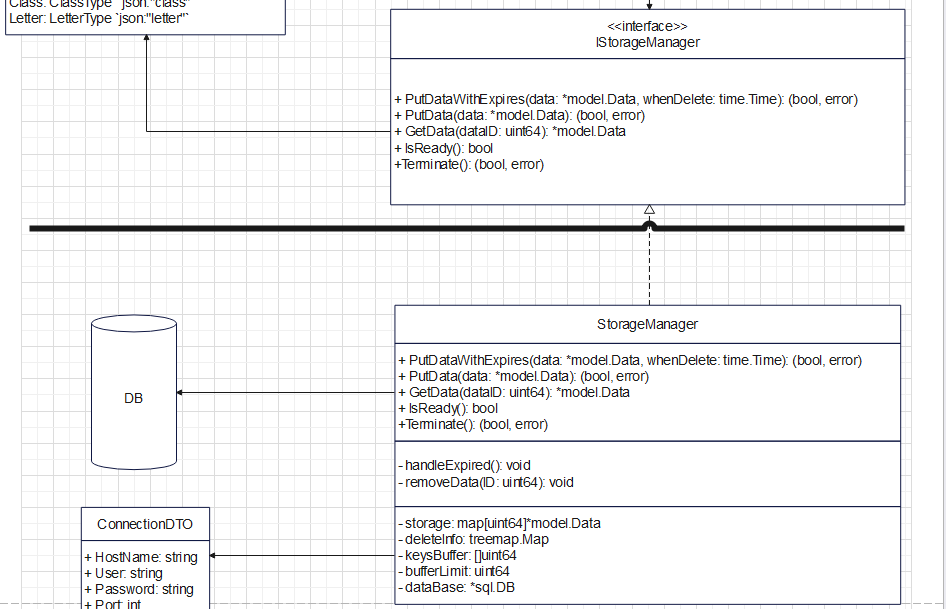
Сбор метрик реализован при помощи open-source решения *prometheus/client\_golang*. Сам сервис является экспортером.

**Слой бизнес-логики**

С точки зрения программной реализации данный слой представлен пакетом *service* (расположение: «*/internal/services*»). На данном уровне присутствует всего один класс – *Controller*, имплементация интерфейса *IController* пакета *transport*.

  
**Рисунок 5 – UML-описание класса Controller**

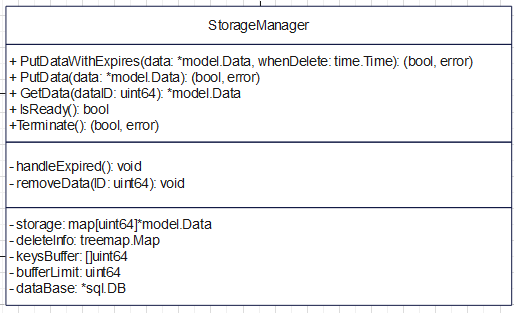
Чтобы правильно создать экземпляр класса, необходимо воспользоваться функцией *InitController*, где в качестве одиночного параметра выступает интерфейс *IStorageManager*, объявленный в том же пакете *service*. Граница взаимодействия слоя бизнес-логики и слоя хранения проходит через данный интерфейс:

  
**Рисунок 6 - Иллюстрация Interface Segregation принципа между слоем бизнес-логики и слоем хранения**

**Слой хранения**

Пакет, содержащий в себе программную реализацию хранилища, - *postgresql* (расположение – «*/internal/database/postgresql*»). Так, в качестве дискового хранилища выбрана *postgresql*, в качестве хранилища, находящегося в оперативной памяти, - *map*, где ключом является *id* объекта, а значением – экземпляр касса *Data* (пакет *model*).

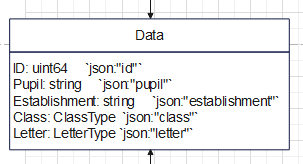
Ключевой класс на данном уровне – *StorageManager*, являющийся имплементацией интерфейса *IStorageManager* пакета *services*.

  
**Рисунок 7 – UML-описание класса *StorageManager***

Так как взаимодействие с БД – это «дорогая операция», при каждом исполнении *PutData* не происходит запись в БД. Обновление таблицы *Pupils* происходит только тогда, когда размер внутреннего буфера *keysBuffer* достигает значения *bufferLimit* (по умолчанию выставлено значение 30), а также в момент вызова *Terminate* (для будущей поддержки *graceful* *shutdown*). Дополнительно хранится внутренняя переменная *deleteInfo*, где указаны времена, когда необходимо удалить объект из хранилища. Представляет собой красно-черно дерево, так как нужно иметь отсортированный порядок ключей для реализации функции *lowerBound* с целью непосредственного определения объектов, которые нужно удалить.

**Модель данных**

И слой бизнес логики, и слой хранилища взаимодействуют с объектами типа *Data*. Данный тип – структура пакета *model* (расположение – «*/internal/data-model*»), соответствующая телу клиентского запроса и атрибутам таблицы *Pupils* в БД

  
**Рисунок 8 – UML-описание класса Data**

**Интеграция с Kubernetes**

Чтобы приложение могло работать в среде *Kubernetes*, в папке *build* проекта располагаются Docker-файлы, по которым головной *docker-compose.yaml* создает образ на каждый компонент.

В данном случае достаточно иметь Docker-файл для самого сервиса, то есть той части, что написана на GO. Контейнеры для БД и для Prometheus присутствуют для того, чтобы проверить функциональность приложения. По хорошему счету БД не следует хранить и разворачивать в docker-контейнере, в тестовом же случае это делается с учетом инициализации БД какими-то значениями (посредством *database\_creation.sql*) в целях отладки.

Последовательность развертывания с учетом отладки:

1. cmd && cd <путь к головной папке с *docker-compose.yaml* >
2. docker-compose build
3. docker-compose up
4. docker exec -it <ID контейнера части, что написана на GO> bash
5. cd /Application &&
6. ./main