ООО «Отус онлайн-образование»

Курс «Scala-разработчик»

Сафронов Валерий Евгеньевич

Сервис управления наливом и взвешиванием нептепродутов

ВЫПУСКНОЙ ДИПЛОМНЫЙ ПРОЕКТ

Выполнил: студент группы

[scala-dev-mooc-2023-09](https://github.com/ScalaOtus/scala-dev-mooc-2023-09)

|  |  |
| --- | --- |
|  | Руководитель курса   \_\_\_\_\_\_\_\_/Воронец Алексей/  (подпись)     (И.О. Фамилия) |

«\_\_» апреля 2024 г.

КУРСК – 2024

СОДЕРЖАНИЕ

ГЛАВА I. Цели проекта.

ГЛАВА II. Планируемый функционал и компоненты.

ГЛАВА III. Используемые технологии.

ГЛАВА |V. Реализованный функционал

ГЛАВА V. Архитектура системы.

ГЛАВА VI. Детали реализации.

ГЛАВА VII. Функционал тестирования. Выполнение теста.

ГЛАВА VIII. Выводы и пожелания.

1. **Цели проекта.**

Цель проекта - разработать фуллстэк приложение (бэкенд и фронтенд) реализующее часть функционала системы управления наливом и взвешиванием нефтепродуктов.

Приложение должно выполнять следующий функционал:

-принимать восходящий поток данных от оборудования автоматизированных весовых платформ (железнодорожной и автомобильных) по протоколу TCP;

- отображать состояние платформ пользователю в веб-интерфейсе браузера (взаимодействи браузера и бэкенда по протоколам Http и WebSocket);

- обрабатывать алгоритм взвешивания с применением карт доступа;

- отдавать поток событий в топики KAFKA;

- сохранять поток событий в таблицах базы данных;

- предоставлять доступ по протоколу Http к сохраненным в БД событиям для внешних сервисов;

- логировать события обработки в журнале;

- реализовать архитектуру легко масштабируемую на большее количество весовых платформж

**II. Планируемый функционал и компоненты.**

В проекте планировалось реализовать слабосвязанные компоненты разертываемые в единую работоспособную систему используя контейнер инверсии управления и внедрения зависимостей (IoK контейнер).

Весь функционал прокета разбивается на следующие крупные блоки.

1. Бэкенд
   1. Бизнес-модули
   2. Модули тестирования
2. Фронтенд (SPA приложение для браузера)
3. Вспомогательный код тестирования (скрипты)

Запланированные к реализации компоненты и сервисы (по блокам)

1. a)

* + 1. Конфигурация бэкенда в формате HOCON и сервисы (классы) чтения конфигурации.
    2. Модуль контейнера инверсии управления и внедрения зависимостей- описывающий создание экземпляров компонентов и сервисов и биндинг экземпляров на классы интерфейсовы.
    3. Модели текстовых протоколов в соответствии с которыми данные поступают от оборудования - паттерны регулярных выражений описывающих протокол и механизм получать эти паттерны по имени.
    4. Экстракторы протоколов - scala-экстракторы позволяющие парсить строки протоколов и преобразовывать их в ADT - кейс классы.
    5. Web модели - включают кейс классы данных для сериализации и имплиситные Json-врайтеры - предоставляют функционал Json-сериализации в компонентах, где такая сериализация требуется.
    6. DB модели - кейс классы на которые отображаются данные таблиц базы данных.
    7. DB схема - классы описывающие схему базы данных.
    8. Классы TCP серверов.
    9. Классы WebSocket сервера.
    10. Web контроллер для работы по Http - предоставляет асинхронные http - обработчики (на основе Future) и потоковые обработчики на основе стримов.
    11. Web контроллер для акцепта соединений по протоколу Web-socket.
    12. Актор WebSocket - сервера
    13. HTTP роутер.
    14. Сервис DB слоя - содержит логику запросов на DSL FRM (функционально реляционного мэппинга). Предоставляет интерфейс на основе фьючеров и стримов.
    15. Глобальные хранилища.
    16. Сервисы бизнеслогики:
        1. Диспетчеры
        2. Парсеры протоколов
        3. Стейт-машины
        4. Менеджер диспетчеров
    17. Классы TCP - клиентов
    18. Менеджер TCP - клиентов
    19. Классы UDP сервера: UDP сервер, менеджер UDP сервера, NetWorker

1. SPA приложение для браузера
2. Набор BASH скриптов организующих длительный тест имитирующий потоки данных от оборуддования.

**III. Используемые технологии.**

Технологии бэкенда:

1. Play Framework - базовый связующий фреймворк
2. Конфигурация в формате HOCON. Configuration API.
3. Google Guice - контейнер инверсии управления и внедрения зависимостей.
4. Асинхронный HTTP, потоковый HTTP.
5. Akka
   1. классические акторы - используются для реализации сетевого функционала - TCP серверы, TCP клиенты, UDP сервер, WebSocket сервер;
   2. типизированные акторы - используются для реализации сервисов бизнес-логики - диспетчеров, парсеров протоколов, стейт-машин;
6. Akka Streams - используются для следующих задач
   1. сериализация и отправка сообщений в фронтенд по протоколу WebSocket;
   2. сериализация и отправка сообщений в топики KAFKA;
   3. организации потокового взаимодействия базы данных и вэб-контроллера;
   4. оптимизация батч режима вставки данных в таблицы БД - разбиение на чанки и параллелизм;
7. Play Json - применяются различные способы сериализации ADT в т.ч. имплиситные Writes.
8. PostgreeSQL.
9. Scala Slick - библиотека функционально-реляционного мэппинга. API DB-слоя используют как потоки так и фьючеры.
10. Протокол WebSocket - может использоваться в качестве альтернативы HTTP для взаимодействия бэкенда и фронтенда.
11. Использование scala-экстракторов для парсинга протоколов.

Технологии фронтенда:

1. Язык разработки - TypeScript
2. Фреймворк мультиплатформенной и кроссплатформенной сборки c библиотекой компонентов - Quasar Framework.
3. Внутренний реактивный web фрэймворк - Vue JS 3.
4. Библиотека управления состоянием - Pinia
5. Сеть: AJAX библиотека Axios, WebSocket библиотека reconnecting-websocket.
6. Сборка и траспилирование: webpack, babel.

**IV. Реализованный функционал.**

Весь функционал запланированый в пунктах I,II реализован.

**V. Архитектура системы.**

Платформа ЖД весов

Микроконтроллерный блок управления

Микроконтроллерный блок управления

Платформа автомобильных весов 2

Датчик периметра

Датчик периметра

**Табло бегущей строки**

RFID

RFID

Платформа автомобильных весов 1

Датчик периметра

Датчик периметра

**Табло бегущей строки**

RFID

RFID

Платформа автомобильных весов 3

Датчик периметра

Датчик периметра

**Табло бегущей строки**

RFID

RFID

Микроконтроллерный блок управления

Микроконтроллерный блок управления

Локальная сеть / Интернет

TCP

TCP

TCP

Рисунок 1 - физические объекты системы

TCP

Скрипты имитации потока данных от оборудования

Сеть

UDP

UDP

TCP

Бэкенд

Модуль тестирования, UDP сервер, TCP клиенты

TCP серверы

TCP

Инфраструктура имитации оборудования

Фронтенд

HTTP

Web Socket

Рисунок 2 - общая схема компонентов программной системы и коммуникаций

СУБД

Сеть

Конфигурация

Модуль Google Guice

TCPStorage Singleton

параметры конфигурации

Конффигурация TCP серверов

TCPServerBuilder Singleton

ActorRef созданного TCP сервера помещается в хранилище

TCP Server 1

TCP Server 3

TCP Server 4

TCP Server 2

Handler

Handler

Handler

Handler

ActorRef

ActorRef

ActorRef

ActorRef

TCP-серверы и их хэндлеры - классические акторы

Менеджер диспетчеров Singleton

создание акторов диспетчеров и внедрение ссылок

Диспетчер RailWeighbridgeер

Диспетчер TruckScale[1]

Диспетчер TruckScale[2]

Диспетчер TruckScale[3]

ActorRef

Диспетчеры - типизированные акторы

Корневой актор

Поток событий - чанки (фрагменты данных) протоколов

Парсер протокола RailProtokol

Парсер протокола AutoProtokol

Парсер протокола AutoProtokol

Парсер

протокола AutoProtokol

Потоки событий

1. чанки от диспетчеров к парсерам
2. Сбытия состояний и карт к диспетчерам

Поток событий

1. события состояния платформы
2. события карт

Рисунок 3 - детальная схема компонентов и сервисов и коммуникаций между ними (начало)

Парсеры протоколов -типизированные акторы

Стейт машина 2

Auto

Стейт машина 1 Rail

Стейт машина 2

Auto

Стейт машина 3

Auto

Стейт машина 4 Auto

Слой работы с БД

Slick

Schema, FRM

PostgreSQL

Stream API

Future API

Поток событий

1. события состояния платформы
2. события карт

Типизированныце акторы стейт-машин

Actor ( получен через ActorSource.actorRef )

И

Интерфейс с внешними сервисами

Actor

И

Интерфейс с внешними сервисами

Actor

Интерфейс с внешними сервисами

Source

FLow

И

Интерфейс с внешними сервисами

Actor

FLow

FLow

Sink для KAFKA State

Sink для KAFKA Card

Sink для WebSocket

KAFKA

(имитируется)

Топик State

Топик Card

Поток данных в WebSocket Server

Рисунок 4 - детальная схема компонентов и сервисов (продолжение -стейт машины)

Потоко безопасная карта текущего состояния платформ

Web Socket сервер (классический актор)

Роуты

Web Controller Web Socket (основан на ActorFlow)

AcceptWebSocket connection

Карта WebSocket соединений

Поток данных из Sink стримов

WebSocket

Web Controller Main (http)

Асинхронный HTTP

Потоковый HTTP

Браузер

HTTP

HTTP

Потоко безопасная карта текущего состояния платформ

Внешние сервисы

Слой работы с БД

Slick

PostgreSQL

Stream API

Source

Flow

Chunked

DatabasePublisher

Рисунок 5 - детальная схема компонентов и сервисов (продолжение) - WEB

**VI. Детали реализации.**

1. Конфигурация.
   1. Конфигурация ExecutionContext для блокирующих и длительных операций.

## фиксированный пул для блокирующих и длительных олераций  
fixedBlockPool = **30**blocking-io-dispatcher {  
 type = Dispatcher  
 executor = "thread-pool-executor"  
 thread-pool-executor {  
 fixed-pool-size = **$**{fixedBlockPool}  
 }  
 throughput = **1**}

* 1. Конфигурация TCP Серверов.

# конфиурация TCP серверов  
tcp-servers {  
 host-ip: "127.0.0.1"  
 servers: [  
 {  
 id: "Первый"  
 port: **8876** phisicalObject: "RailWeighbridge"  
 channelName: RailsMain  
  
 }**,** {  
 id: "Второй"  
 port: **8877** phisicalObject: "TruckScale[1]"  
 channelName: AutoMain  
 }**,** {  
 id: "Третий"  
 port: **8878** phisicalObject: "TruckScale[2]"  
 channelName: AutoMain  
 }**,** {  
 id: "Четвертый"  
 port: **8879** phisicalObject: "TruckScale[3]"  
 channelName: AutoMain  
 }  
  
 ]  
}

* 1. Конфигурация протоколов.

## конфигурация протоколов  
protocols: {  
 AutoMain: SCALE\_DATA\_PATTERN\_PROTOCOL2  
 RailsMain: SCALE\_DATA\_PATTERN\_RAIL\_PROTOCOL  
}  
  
card: SCALE\_DATA\_PATTERN\_PROTOCOL2\_EMMARIN  
  
useCRC: false  
  
convert\_HexEmMarine\_to\_TextEmMarine: true

* 1. Конфигурация таймаута обработки карт.

# таймаут на который блокируется обработчика карты  
timeoutCardResponce: **3**

* 1. Конфигурация веб-протокола - между фронтендом и бэкендом.

# конфигурация веб-протокола  
webProtocols: {  
 Http {  
 name: "Http"**,** endPoint: ""  
 }  
 WebSocket {  
 name: "WebSocket"  
 endPoint: "ws://192.168.0.252:9000/websocket"  
 }  
 Any {  
 name: "Any"  
 endPoint: "ws://192.168.0.252:9000/websocket"  
 }  
  
 use: **$**{webProtocols.WebSocket}  
}

* 1. Конфигурация БД и Slick.

# конфигурация пула для HicariCP  
fixedConnectionPool = **24**# параметры подключения к БД  
jdbcUrl = "jdbc:postgresql://localhost:2345/oildata"  
db\_username = postgres  
db\_password = \*\*\*\*\*\*\*\*\*  
  
# конфигурация Slick  
slick.dbs {  
 default.profile = "slick.jdbc.PostgresProfile$"  
 default.db.driver = "org.postgresql.Driver"  
 default.db.numThreads=**$**{fixedConnectionPool}  
 default.db.maxConnections=**$**{fixedConnectionPool}  
 default.db.url = **$**{jdbcUrl}  
 default.db.user = **$**{db\_username}  
 default.db.password = **$**{db\_password}  
}

* 1. Конфигурация группировки и параллелизма для batch-insert данных.

# конфигурации группировки и параллелизма для вставки данных  
insertConf {  
 test {  
 listMaxSize: **50** groupSize: **10** parallelism: **5** }  
  
 state {  
 listMaxSize: **10** groupSize: **5** parallelism: **2** }  
  
 card {  
 listMaxSize: **2** groupSize: **1** parallelism: **2** }  
}

1. Структура проекта.

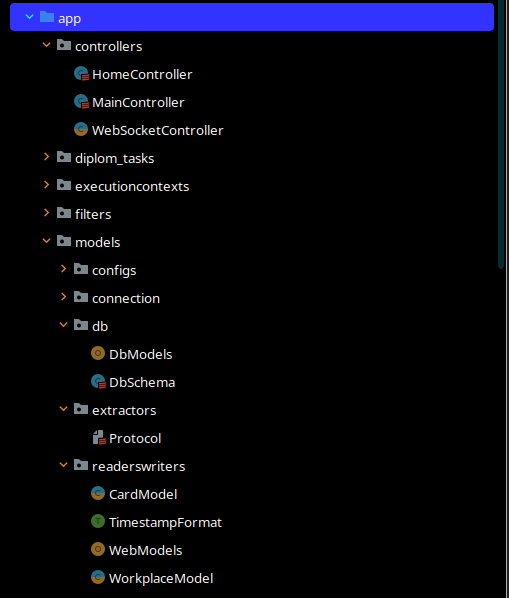


Рисунок 6 - структура проекта (начало)

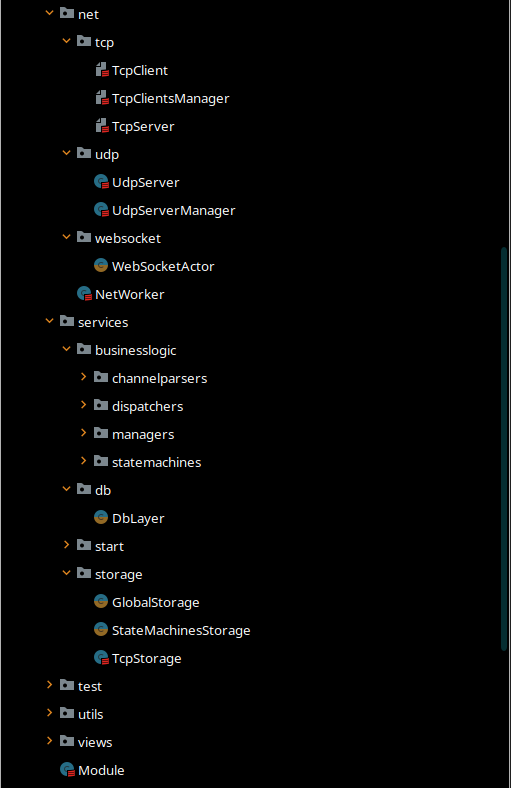


Рисунок 7 - структура проекта (продолжение)

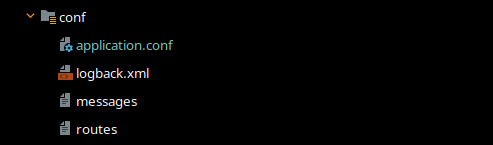


Рисунок 8 - структура проекта (продолжение)

1. Описание способа создания и внедрения типизированных акторов (диспетчеры, парсеры стейт машины)

Сообщение Create......

Типизированный актор

Wrapper class.

Внедрение зависимостей.

Фабрика Create

Корневой актор

Создание актора

Синглтон

Хранилище ссылок на акторы

ActorRef помещается в хранилище

Модуль Google Guice

Биндинг интерфейса Wrapper на реализацию

Провайдер предоставляющий ActorRef из хранлища

ActorRef внедряется в конструктор сервиса

Рисунок 9 - механизм создания типизированных акторов и внедрения ссылок в конструкторы сервисов.

bind(*classOf*[ParserWraper]).annotatedWith(Names.*named*("AutoParserW")).to(*classOf*[ParserAutoProtocolWraper])  
bind(*classOf*[ParserWraper]).annotatedWith(Names.*named*("RailParserW")).to(*classOf*[ParserRailProtokolWraper])  
bindActorFactory[WebSocketActor**,** WebSocketActor.Factory]

@tailrec  
def getRefParser(id: String): ActorRef[ParserCommand] = {  
 val optref = GlobalStorage.*getRefParser*(id)  
 optref match {  
 case Some(ref) => ref  
 case None => *getRefParser*(id)  
 }  
}

//провайдеры акторов парсеров  
@Provides  
@Named("AutoParserA")  
def getAutoProtocolParserActor(@Named("AutoParserW") wraper: ParserWraper): ActorRef[ParserCommand] = {  
 val id = wraper.create()  
 GetRefWhenExist.*getRefParser*(id)  
}  
  
@Provides  
@Named("RailParserA")  
def getRailProtocolParserActor(@Named("RailParserW") wraper: ParserWraper): ActorRef[ParserCommand] = {  
 val id = wraper.create()  
 GetRefWhenExist.*getRefParser*(id)  
}

Рисунок 10 - пример биндинга Wrappers и провайдеров ActorRef обеспечивающих инжекцию в конструкторы сервисов.

1. Пример конфига протоколов

private val *SCALE\_DATA\_PATTERN\_PROTOCOL2\_QR*: String = "(v|V)" +  
 "(\\+|\\-|\\?){4}(\\?{6}|\\s{6}|\\s{5}[0-9]{1}|\\s{4}-[0-9]{1}|\\s{4}[0-9]{2}|" +  
 "\\s{3}-[0-9]{2}|\\s{3}[0-9]{3}|\\s{2}-[0-9]{3}|\\s{2}[0-9]{4}|\\s{1}-[0-9]{4}|" +  
 "\\s{1}[0-9]{5}|\\-[0-9]{5}|[0-9]{6})" +  
 "((Q[0-9a-fA-F\\-]{36})(R|G)?%[0-9a-fA-F]{4}.)"  
  
  
private val *SCALE\_DATA\_PATTERN\_PROTOCOL2\_EMMARIN*: String = "(v|V)" +  
 "(\\+|\\-|\\?){4}(\\?{6}|\\s{6}|\\s{5}[0-9]{1}|\\s{4}-[0-9]{1}|\\s{4}[0-9]{2}|" +  
 "\\s{3}-[0-9]{2}|\\s{3}[0-9]{3}|\\s{2}-[0-9]{3}|\\s{2}[0-9]{4}|\\s{1}-[0-9]{4}|" +  
 "\\s{1}[0-9]{5}|\\-[0-9]{5}|[0-9]{6})" +  
 "((M[0-9a-fA-F]{8})(R|G)?%[0-9a-fA-F]{4}.)"  
  
  
private val *SCALE\_DATA\_PATTERN\_RAIL\_PROTOCOL*: String = "=([0-9]{6}|-[0-9]{5}|[0-9]{7}|-[0-9]{6})."  
  
def getProtocolByName(name: String): String = name match {  
 case "SCALE\_DATA\_PATTERN\_PROTOCOL1" => *SCALE\_DATA\_PATTERN\_PROTOCOL1* case "SCALE\_DATA\_PATTERN\_PROTOCOL1\_EMMARIN" => *SCALE\_DATA\_PATTERN\_PROTOCOL1\_EMMARIN* case "SCALE\_DATA\_PATTERN\_PROTOCOL2" => *SCALE\_DATA\_PATTERN\_PROTOCOL2* case "SCALE\_DATA\_PATTERN\_PROTOCOL2\_MIFARE" => *SCALE\_DATA\_PATTERN\_PROTOCOL2\_MIFARE* case "SCALE\_DATA\_PATTERN\_PROTOCOL2\_QR" => *SCALE\_DATA\_PATTERN\_PROTOCOL2\_QR* case "SCALE\_DATA\_PATTERN\_PROTOCOL2\_EMMARIN" => *SCALE\_DATA\_PATTERN\_PROTOCOL2\_EMMARIN* case "SCALE\_DATA\_PATTERN\_RAIL\_PROTOCOL" => *SCALE\_DATA\_PATTERN\_RAIL\_PROTOCOL* case \_ => ""  
}

1. Пример экстрактора протокола.

object ProtocolRail extends Protocol {  
 case class RailWeight(prefix:String**,** weight:String) extends NoCardOrWithCard with PhisicalObjectEvent  
  
 def apply(prefix:String**,** weight:String):String = {  
 *Try* {  
  
 if (!*patternRailPrefix*.matches(prefix)) throw new ProtocolCreateException(s"Не корректный префикс потокола: **$**prefix")  
 if (!*patternRailWeight*.matches(weight)) throw new ProtocolCreateException(s"Не корректный вес потокола: **$**weight")  
  
 } match {  
 case Failure(exception) =>  
 *logger*.error(exception.getMessage)  
 ""  
 case Success(value) => prefix + weight + "."  
 }  
  
 }  
  
 def apply(obj: RailWeight):String = *apply*(obj.prefix**,** obj.weight)  
  
 def unapply(str: String):Option[RailWeight] = {  
 val isProtocol =*protokolRail*.r.matches(str)  
 if (isProtocol) {  
 val prefix = str.substring(**0, 1**)  
 val weight = str.substring(**1,** str.indexOf("."))  
 Some(RailWeight(prefix**,** weight))  
 }  
 else None  
 }  
}

1. Пример JsonWriter.

implicit val *WebCardWrites*: Writes[WebCard] = (  
 (JsPath \ "id").write[String] and  
 (JsPath \"name").write[String] and  
 (JsPath \ "execute").write[**Boolean**] and  
 (JsPath \ "resp").write[**Boolean**] and  
 (JsPath \ "timeout").write[**Boolean**] and  
 (JsPath \ "card" ).write[String] and  
 (JsPath \ "param" ).write[String] and  
 (JsPath \ "modified" ).write[String]  
 ) (*unlift* (WebCard.unapply))

1. Менеджер диспетчеров.

@Singleton  
class PhisicalObjectsManager @Inject()(@Named("RailWeighbridge") rail: ActorRef[PhisicalObjectEvent]**,** @Named("TruckScale") truck1: ActorRef[PhisicalObjectEvent]**,** @Named("TruckScale") truck2: ActorRef[PhisicalObjectEvent]**,** @Named("TruckScale") truck3: ActorRef[PhisicalObjectEvent]) {  
  
 private val *logger*: Logger = *Logger*(this.getClass)  
 *logger*.info("Загружен PhisicalObjectsManager")  
  
 *logger*.info(s"rail **$**rail")  
 *logger*.info(s"truck1 **$**truck1")  
 *logger*.info(s"truck2 **$**truck2")  
 *logger*.info(s"truck3 **$**truck3")  
  
  
 rail ! NameEvent("RailWeighbridge")  
  
 truck1 ! NameEvent("TruckScale[1]")  
 truck2 ! NameEvent("TruckScale[2]")  
 truck3 ! NameEvent("TruckScale[3]")  
  
   
 def getPhisicalObjectByNameT(name: String): Option[ActorRef[PhisicalObjectEvent]] = {  
 name match {  
 case "RailWeighbridge" => Some(rail)  
 case "TruckScale[1]" => Some(truck1)  
 case "TruckScale[2]" => Some(truck2)  
 case "TruckScale[3]" => Some(truck3)  
 case \_ => None  
 }  
 }  
  
 def getValidNames: List[String] = GlobalStorage.*getValidNames*}

1. Пример актора стейт-машины с переключением поведения и обработкой таймаута.

(см. файл AutoStateMachineTyped)

1. Пример реализации интерфейса с внешними сервисами с помощью Akka Streams (см. файл StateMachineTyped)
2. Пример batch-insert в BD с разбиением на чанки и параллелизмом.

private def insertProtokols(seq: Seq[DbProtokol]): Future[**Int**] = db.run(*protokol* ++= seq).map(\_.getOrElse(**0**))  
def insertProtokolsFuture(listProtokol: List[DbProtokol]): Future[**Int**] =  
 Source.*fromIterator*(() => listProtokol.iterator)  
 .via(*Flow*[DbProtokol].grouped(insertConf.state.groupSize))  
 .mapAsync(insertConf.state.parallelism)((ps: Seq[DbProtokol]) => insertProtokols(ps))  
 .runWith(Sink.*fold*(**0**)(\_+\_))

1. Пример получения данных в Slick используя Stream API и Future API

private def getByIdProtokolsWithPerimetersQuery(idd:String) = for {  
 (prot**,** per) <- *protokol*.filter(\_.id === *UidREF*(idd)) joinLeft *perimeters* on (\_.id === \_.id)  
} yield (prot.id**,** prot.weight**,** prot.crc**,** prot.prefix**,** prot.name**,** prot.humanName**,** prot.svetofor**,** prot.modified**,** per.map(\_.value))  
  
def getByIdProtokolsWithPerimetersF(idd: String): Future[Option[ProtokolWithCards]] =  
 db.run(getByIdProtokolsWithPerimetersQuery(idd).result.headOption)  
  
def getByIdProtokolsWithPerimetersS(idd: String): DatabasePublisher[ProtokolWithCards] =  
 db.stream{  
 val result = getByIdProtokolsWithPerimetersQuery(idd).result  
 result.withStatementParameters(  
 rsType = ResultSetType.ForwardOnly**,** rsConcurrency = ResultSetConcurrency.ReadOnly**,** fetchSize = **10000** ).transactionally  
 }

1. Пример потокового Web-контроллера.

def getAllProtokols: Action[AnyContent] = Action { implicit request =>  
 val publisher = dbLayer.getAllProtokolsWithPerimetersS  
 val protokolSource = Source.*fromPublisher*(publisher)  
 .via(*Flow*[ProtokolWithCards].map(x => x.toString() + "\n"))  
 *Ok*.chunked(protokolSource)  
}

VII. Функционал тестирования. Выполнение теста.

Инфраструктура тестирования показана на Рисунке 2.

Для тестирования необходимо

1. запустить бэкенд в Intellig Idea
2. Выполнить в проекте фронтенда команду quasar dev - автоматически откроется браузер и установит соедиенение с бэкендом по настроеному Web-протоколу.
3. Из папки test/diplom проекта запустить скрипт demo.sh
4. Наблюдать работу теста в браузере.
5. Тестирование потоковых веб-контроллеров выполняется выполнением запросов в программе Postman.

VIII. Выводы и пожелания.

1. Весь функционал предложенный к реализации в дипломном проекте реализован и функционирует.
2. Проект легко МАСШТАБИРУЕТСЯ на большее количество весовых платформ- достаточно
   1. добавить инжекцию диспетчеров в файл PhisicalObjectsManager и внести в него минимальные очевидные добавления.
   2. добавить данные в конфигурацию TCP серверов в файле application.conf.
3. Корректность реализации подтверждена длительным тестом.
4. Пожелание - этот проект может быть развит в направлениях:
   1. реализации ПРОТИВОДАВЛЕНИЯ - но это должно охватывать все компоненты обрабатывающие потоки данных от TCP сервера до реализации интерфейса с внешними сервисами в Akka Streams и включать тщательное тестирование.
   2. настройка СУПЕРВАЙЗИНГА и восстановления Акторов после сбоев. Обязательно тщательное тестирование.

Ссылки на документацию.

<https://www.playframework.com/>

<https://akka.io/>

<https://doc.akka.io/docs/akka/current/typed/index.html>

<https://doc.akka.io/docs/akka/current/index-classic.html>

<https://doc.akka.io/docs/akka/current/stream/index.html>

<https://github.com/google/guice>

<https://netvl.github.io/guice/users-guide.html>