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

Руководитель курса

_____/Воронец Алексей/
(подпись) (И.О. Фамилия)

«__» апреля 2024 г.

КУРСК – 2024

СОДЕРЖАНИЕ

I. 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
    a) Бизнес-модули
    b) Модули тестирования
2. Фронтенд (SPA приложение для браузера)
3. Вспомогательный код тестирования (скрипты)

Запланированные к реализации компоненты и сервисы (по блокам)

1. a)

    i. Конфигурация бэкенда в формате HOCON и сервисы (классы) чтения конфигурации.
    ii. Модуль контейнера инверсии управления и внедрения зависимостей- описывающий создание экземпляров компонентов и сервисов и биндинг экземпляров на классы интерфейсовы.
    iii. Модели текстовых протоколов в соответствии с которыми данные поступают от оборудования - паттерны регулярных выражений описывающих протокол и механизм получать эти паттерны по имени.
    iv. Экстракторы протоколов - scala-экстракторы позволяющие парсить строки протоколов и преобразовывать их в ADT - кейс классы.
    v. Web модели - включают кейс классы данных для сериализации и имплиситные Json-врайтеры - предоставляют функционал Json-сериализации в компонентах, где такая сериализация требуется.
    vi. DB модели - кейс классы на которые отображаются данные таблиц базы данных.
    vii. DB схема - классы описывающие схему базы данных.
    viii. Классы TCP серверов.
    ix. Классы WebSocket сервера.
    x. Web контроллер для работы по Http - предоставляет асинхронные http - обработчики (на основе Future) и потоковые обработчики на основе стримов.
    xi. Web контроллер для акцепта соединений по протоколу Web-socket.

xii. Актор WebSocket - сервера

xiii. HTTP роутер.

xiv. Сервис DB слоя - содержит логику запросов на DSL FRM (функционально реляционного мэппинга). Предоставляет интерфейс на основе фьючеров и стримов.

xv. Глобальные хранилища.

xvi. Сервисы бизнеслогики:

1. Диспетчеры
2. Парсеры протоколов
3. Стейт-машины
4. Менеджер диспетчеров

b)

i. Классы TCP - клиентов

ii. Менеджер TCP - клиентов

iii. Классы UDP сервера: UDP сервер, менеджер UDP сервера, NetWorker

2. SPA приложение для браузера

3. 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
    a) классические акторы - используются для реализации сетевого функционала - TCP серверы, TCP клиенты, UDP сервер, WebSocket сервер;
    b) типизированные акторы - используются для реализации сервисов бизнес-логики - диспетчеров, парсеров протоколов, стейт-машин;
6. Akka Streams - используются для следующих задач
    a) сериализация и отправка сообщений в фронтенд по протоколу WebSocket;
    b) сериализация и отправка сообщений в топики KAFKA;
    c) организации потокового взаимодействия базы данных и вэб-контроллера;
    d) 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 с 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.

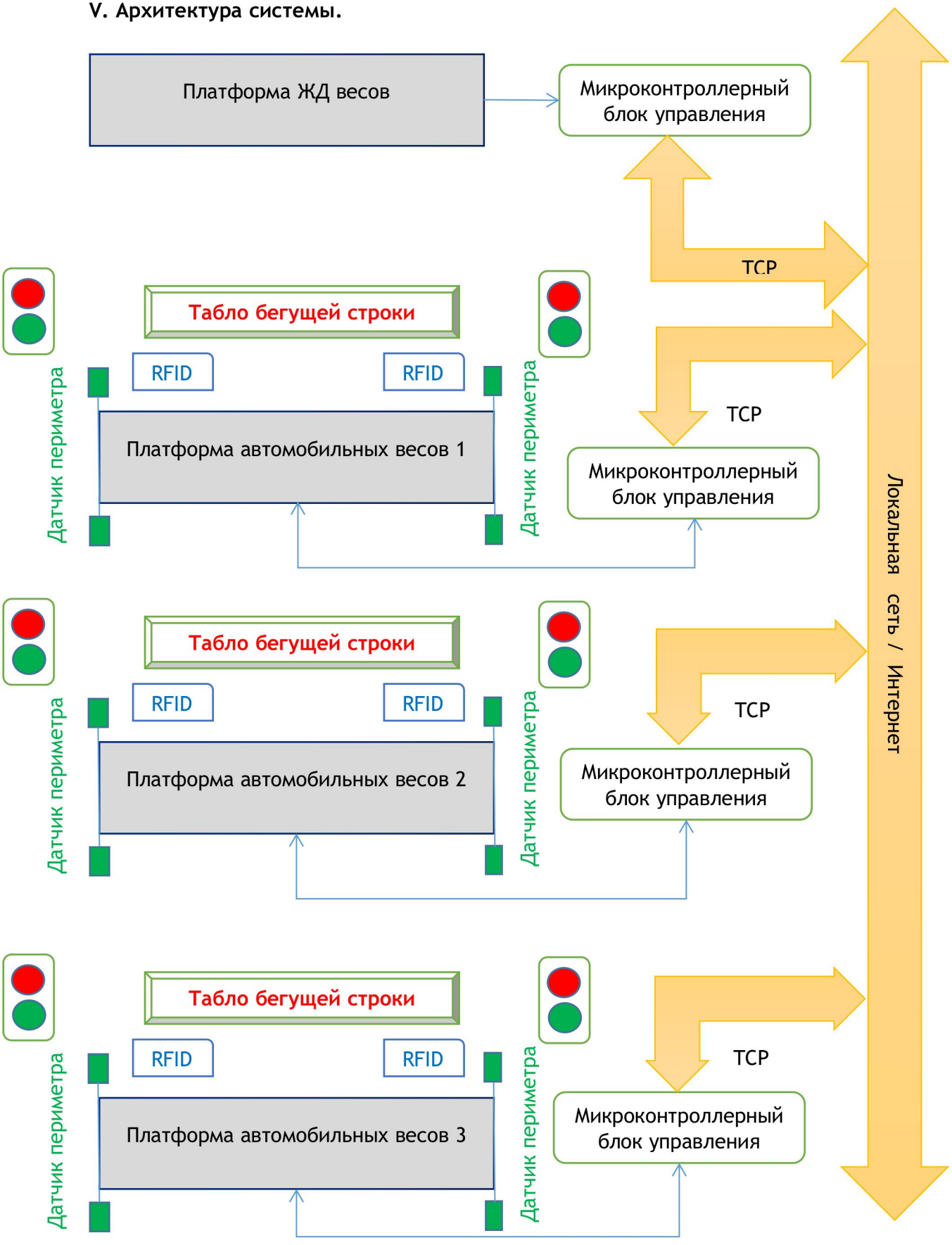

Рисунок 1 - физические объекты системы

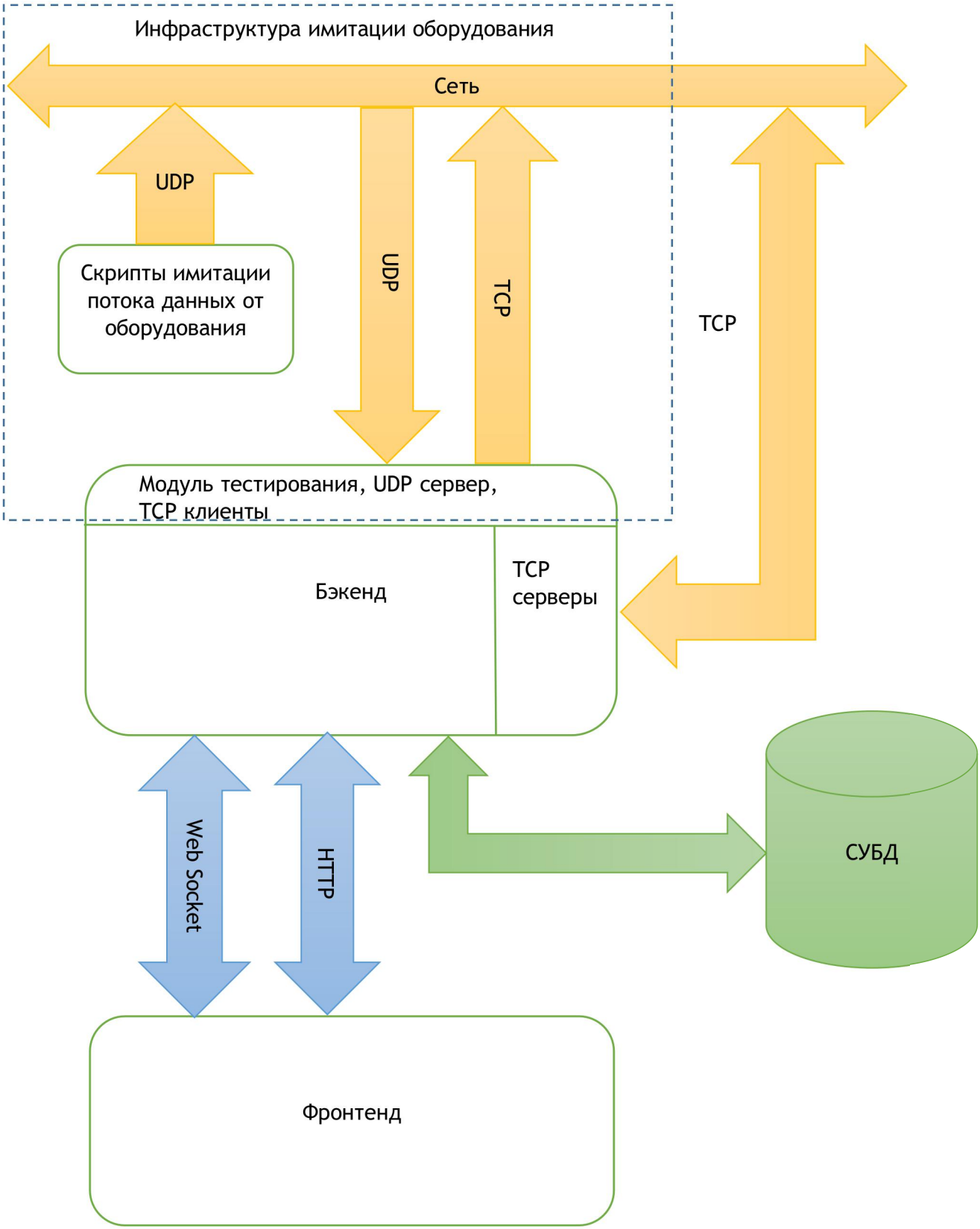

Рисунок 2 - общая схема компонентов программной системы и коммуникаций

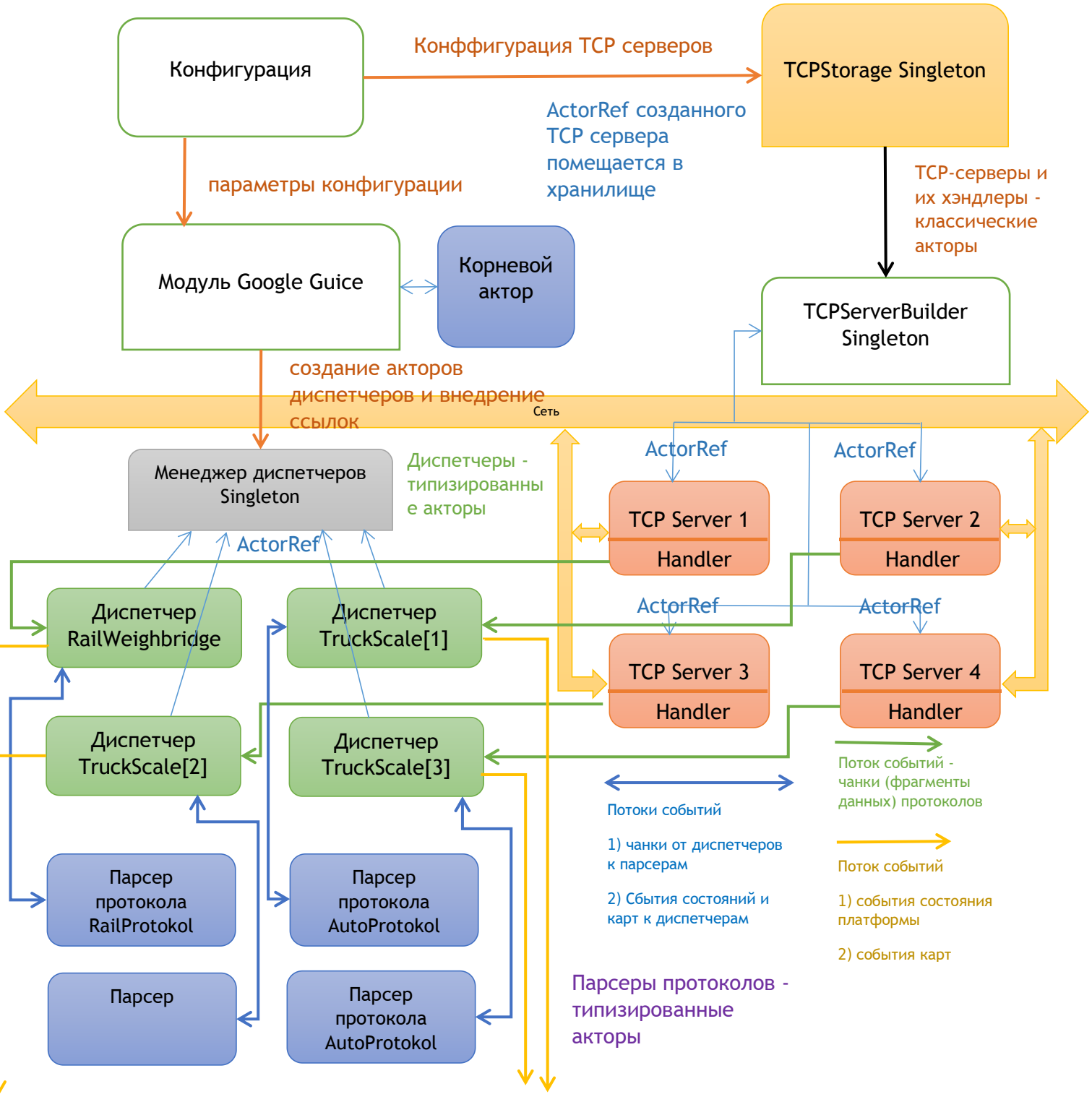

Рисунок 3 - детальная схема компонентов и сервисов и коммуникаций между ними (начало)

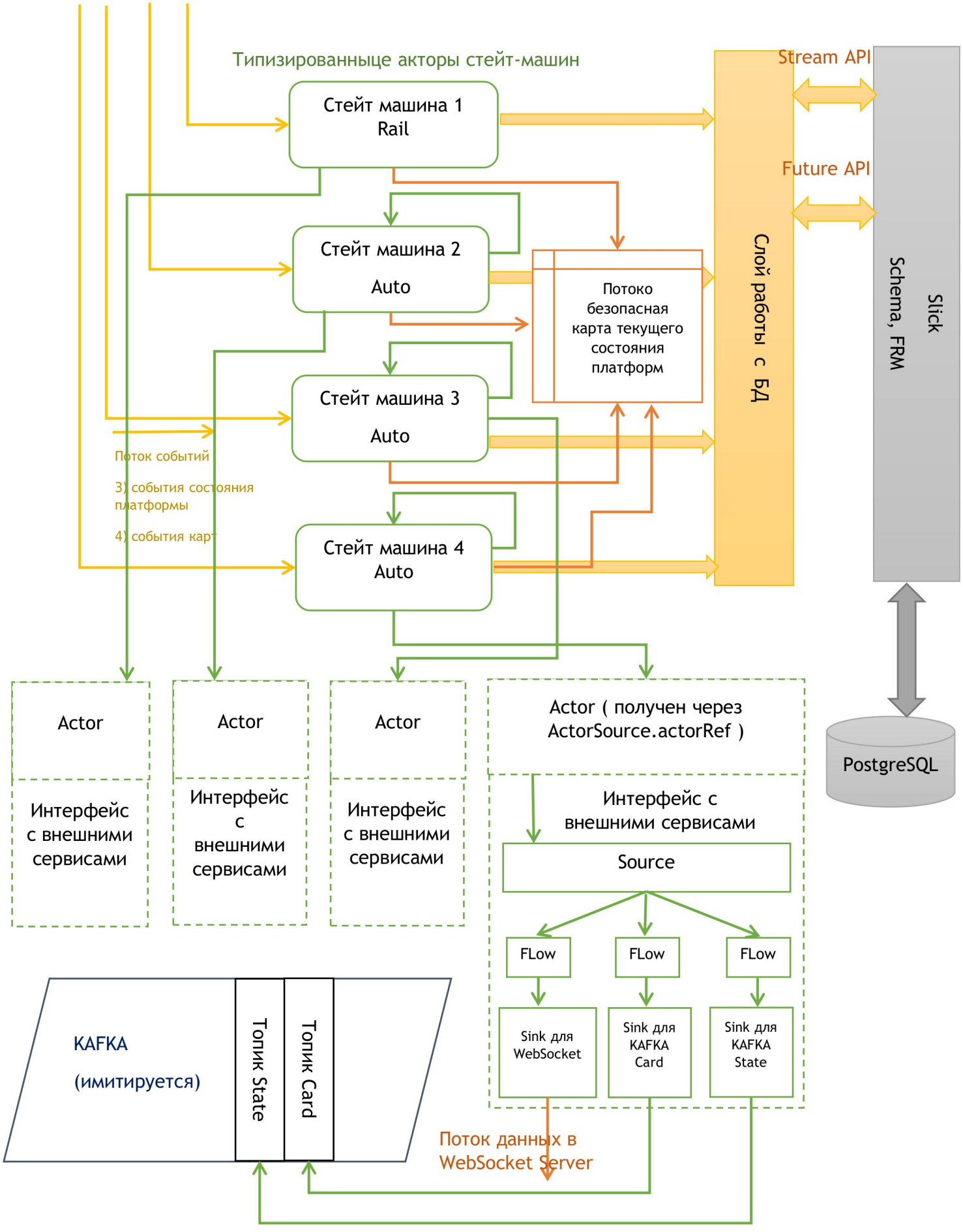

Рисунок 4 - детальная схема компонентов и сервисов (продолжение -стейт машины)

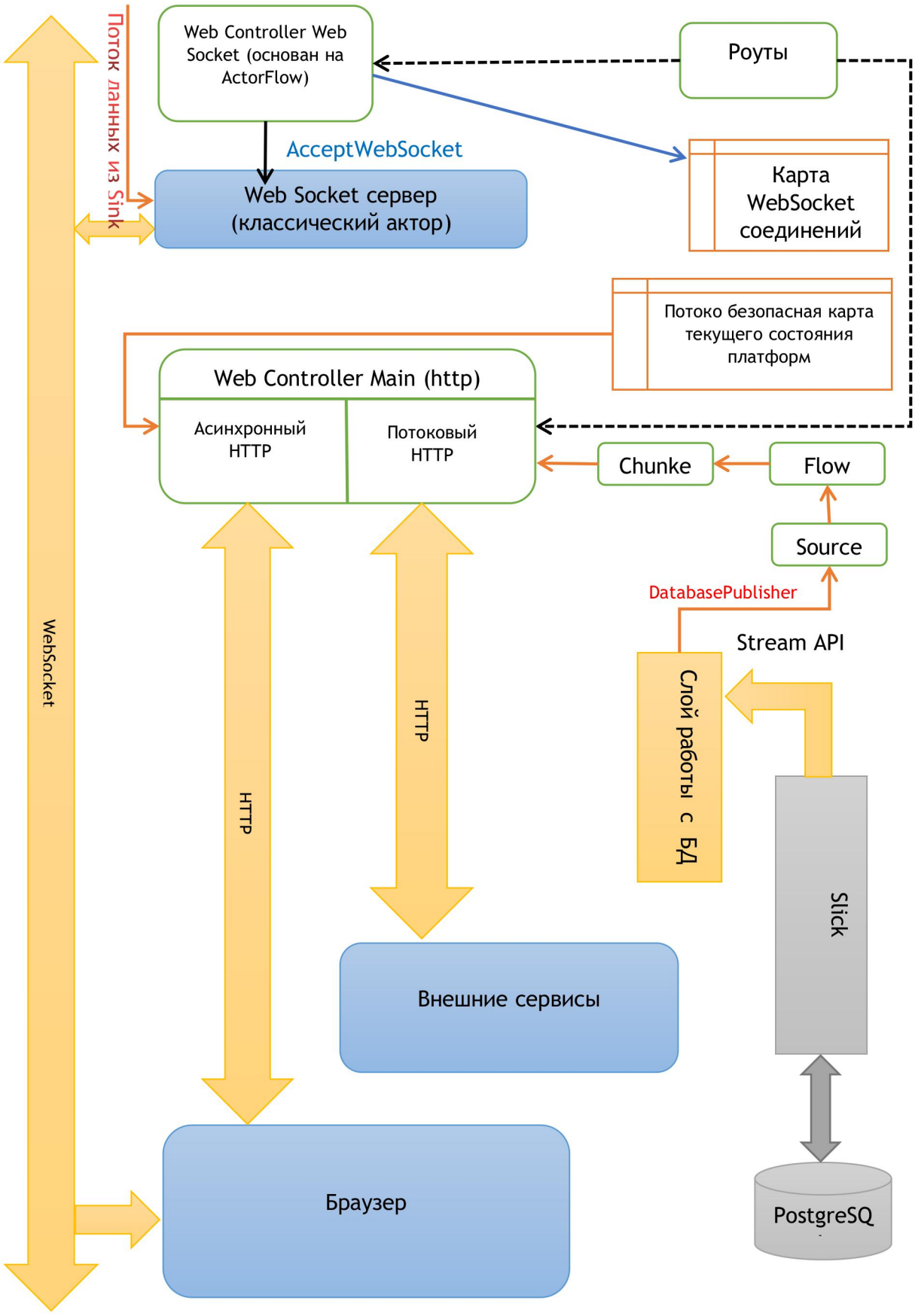

Рисунок 5 - детальная схема компонентов и сервисов (продолжение) - WEB

VI. Детали реализации.

1. Конфигурация.

   a) 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
}
```

   b) 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
  }

 ]
}
```

c) 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
```

d) Конфигурация таймаута обработки карт.

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

e) 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}
}
```

f) 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}
}
```

g) 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
}
}
```

2. Структура проекта.

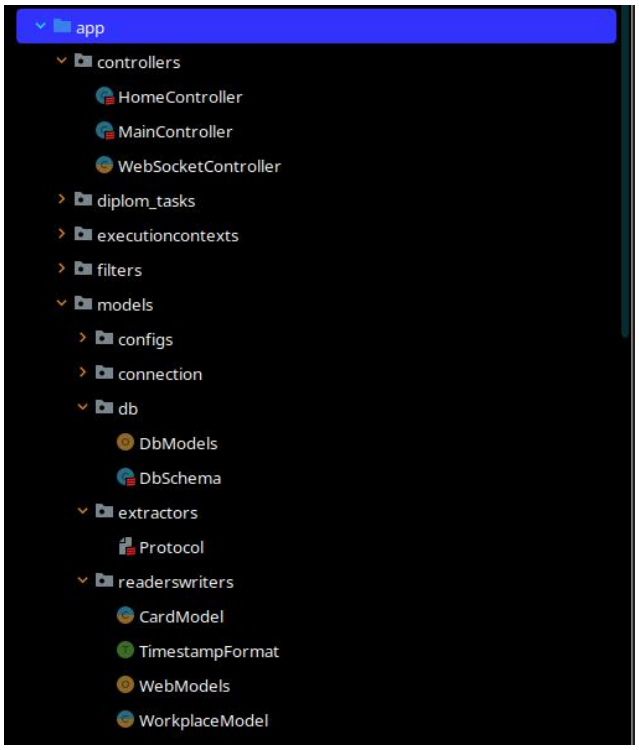

Рисунок 6 - структура проекта (начало)

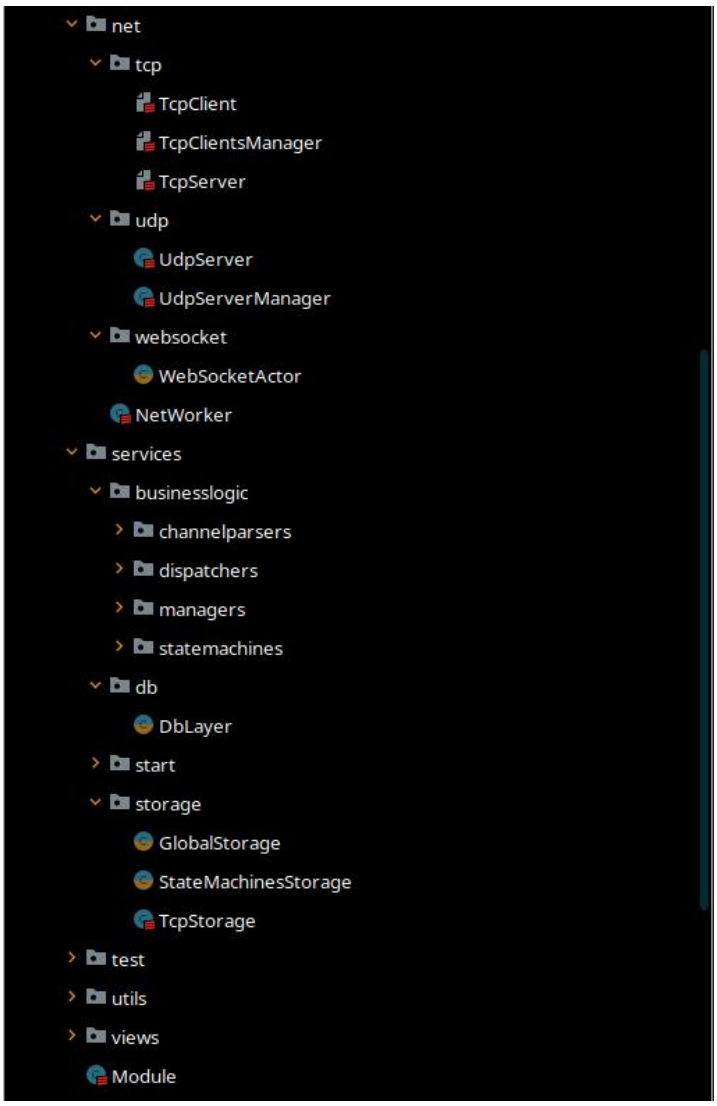

Рисунок 7 - структура проекта (продолжение)

Рисунок 8 - структура проекта (продолжение)

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

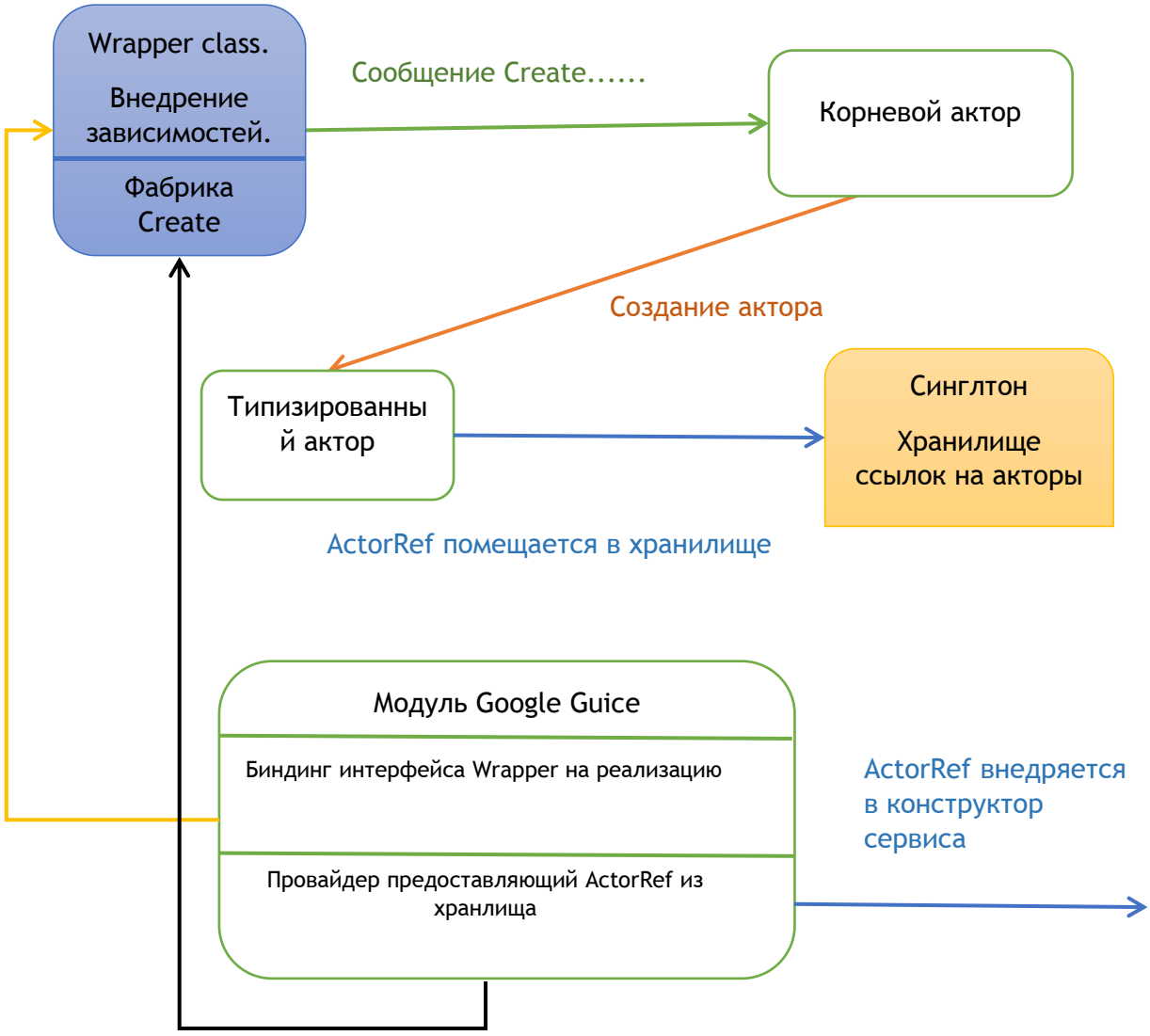

Рисунок 9 - механизм создания типизированных акторов и внедрения ссылок в конструкторы сервисов.

```
bind(classOf[ParserWraper]).annotatedWith(Names.named('AutoParserW')).to(classOf[Parser
AutoProtocolWraper])
bind(classOf[ParserWraper]).annotatedWith(Names.named('RailParserW')).to(classOf[Parser
RailProtokolWraper])
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.

4. 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 _ =| ‘‘
}
```

5. 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
 }
}
```

6. 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))
```

7. 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


}
```

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

(см. файл AutoStateMachineTyped)

9. Пример реализации интерфейса с внешними сервисами с помощью Akka Streams (см. файл StateMachineTyped)

10. 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)(_+_))
```

11. 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
 }
```

12. 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 соединенение 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

   a) добавить инжекцию диспетчеров в файл PhisicalObjectsManager и внести в него минимальные очевидные добавления.

   b) 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:

   a) 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.

   b) 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