Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА СИСТЕМЫ ОБРАБОТКИ ИНФОРМАЦИИ И УПРАВЛЕНИЯ

Отчет по домашнему заданию

Студент Вардумян Арсен Тигранович
фамилия, имя, отчество

Группа ИУ5-51Б

Студент 15.01.2021 **Вардумян А.Т.**
подпись, дата фамилия, и.о.

Преподаватель 15.01.2021 **Гапанюк Ю.Е.**
подпись, дата фамилия, и.о.

2022г.

Описание задания:

Целью домашнего задания является написание торгового робота на платформе [kraken-demo](https://futures.kraken.com/ru.html). Демонстрационная платформа не требует никаких персональных данных при регистрации и выполняет заявки в песочнице, не требуя никаких реальных средств для взаимодействия с платформой.

Критерии успешно выполненного задания:

1. Релизован механизм аутентификации робота на демо-платформе для получения информации и реализации заявок.
2. Реализованы REST-эндпоинты для работы с роботом (задание инструмента для работы, настройка параметров, запуск/стоп робота и т.д.)
3. Робот должен поддерживать http протокол для работы с демо-биржей.
4. Робот должен уметь подключаться с помощью web-socket до сервера демо-биржи для подписки на стаканы по инструментам.
5. Робот должен имплементировать один любой торговый индикатор для принятия решения по выставлению заявки.
6. Информацию о всех выставленных заявках робот должен хранить в Postgres хранилище.
7. Информацию о всех выставленных заявках робот должен отправить в Telegram-бота.
8. Критичный функционал программы должен быть покрыт unit-тестами.
9. Должна быть оформлена документация по запуску и управлению роботом (README обязательно, swagger ui необязательно)

Подробнее о каждом пункте:

1. [Регистрация/аутентификация на платформе](https://demo-futures.kraken.com/futures) - здесь можно зарегистрироваться и получить токен для работы с демо-платформой.

2. [REST, WS эндпоинты для работы с биржей](https://support.kraken.com/hc/en-us/articles/360022839491-API-URLs) -
   обратите внимание, что мы работаем с демо-биржей, у которой в url есть префикс "demo-", например demo-futures.kraken.com вместо futures.kraken.com. Документация по работе с платформой [тут](https://support.kraken.com/hc/en-us/sections/360012894412-Futures-API).

3. [Торговые индикаторы wiki](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%85%D0%BD%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B8%D0%BD%D0%B4%D0%B8%D0%BA%D0%B0%D1%82%D0%BE%D1%80) -
   если нет желания заниматься реализацией математики в коде, допустимо использовать простейший индикатор типа stop-loss/take-profit. Логика для него выглядит следующим образом: купи/продай при достижении заданного значения цены. Работа с индикатором должна быть максимально абстрагирована от остальной логики робота, для удобства проверки - мы не будем смотреть реализацию индикатора, но будем смотреть как он используется в роботе.

4. Хранение данных допустимо как с использованием ORM так и без него.

5. Создание и настройка телеграм-бота: core.telegram.org/bots

6. Чем выше покрытие юнитами - тем лучше. Но обязательным считается покрытие не менее 25% функционала.

Текст программы:

“pkg/kraken/connect.go”:

```
package kraken

import (
    "context"
    "errors"
    "fmt"
    "net/url"
    "time"

    "github.com/gorilla/websocket"
    "github.com/sirupsen/logrus"
)

var (
    ErrLostConnection     = errors.New("lost connection with Kraken WebSocket")
    ErrKrakenConnect      = errors.New("cannot connect to Kraken WebSocket")
    ErrSubscriptionFailed = errors.New("subscription failed")
)

type ResponseMessage struct {
    Event  string  `json:"event"`
    Candle *Candle `json:"candle"`
}

type Candle struct {
    Open   string  `json:"open"`
    High   string  `json:"high"`
    Low    string  `json:"low"`
    Close  string  `json:"close"`
    Time   float64 `json:"time"`
    Volume float64 `json:"volume"`
}

type SubscriptionMessage struct {
    Event      string   `json:"event"`
    Feed       string   `json:"feed"`
    ProductIDs []string `json:"product_ids"`
}

const (
    subscriptionTimeout = 5 * time.Second
    pingPeriod          = 70 * time.Second
)

func subscribeToCandle(conn *websocket.Conn, productID string, period string) error {
    subscriptionMessage := SubscriptionMessage{
```

```
        Event:      "subscribe",
        Feed:       period,
        ProductIDs: []string{productID},
    }

    err := conn.WriteJSON(subscriptionMessage)
    if err != nil {
        return err
    }

    var msg ResponseMessage
    // Read first info message
    err = conn.ReadJSON(&msg)
    if err != nil {
        return err
    }

    // Read subscription status message
    err = conn.ReadJSON(&msg)
    if err != nil {
        return err
    }

    if msg.Event == "alert" {
        return errors.New("invalid productID or period")
    }

    return nil
}

func (k *krakenService) connect(ctx context.Context, logger logrus.FieldLogger,
productID string, period string) error {
    conn, err := k.openConnection(logger)
    if err != nil {
        return fmt.Errorf("%w: %v", ErrKrakenConnect, err)
    }

    err = subscribeToCandle(conn, productID, period)
    if err != nil {
        return fmt.Errorf("%w: %v", ErrSubscriptionFailed, err)
    }

    candles = make(chan interface{})

    go func(context.Context) {
        ticker := time.NewTicker(pingPeriod)

        defer func() {
            ticker.Stop()
            conn.Close()

            if r := recover(); r != nil {
                logger.Errorf("%v: %v", ErrLostConnection, r)
```

```
                err = k.connect(ctx, logger, period, productID)
                if err != nil {
                    logger.Error(err)
                    close(candles)
                }
            } else {
                close(candles)
            }
        }()

        var recentTime float64

        for {
            select {
            case <-ctx.Done():
                return

            case <-ticker.C:
                logger.Debug("ping connect")
                err := conn.WriteMessage(websocket.PingMessage, nil)
                if err != nil {
                    logger.Error("ping message error")
                    return
                }

            default:
                var message ResponseMessage

                err = conn.ReadJSON(&message)
                if err != nil {
                    if websocket.IsCloseError(err, websocket.CloseGoingAway,
websocket.CloseAbnormalClosure) {
                        panic(err)
                    } else {
                        logger.Error(err)
                        return
                    }
                }

                if message.Candle != nil && recentTime < message.Candle.Time {
                    candles <- message.Candle
                    recentTime = message.Candle.Time
                }
            }
        }
    }(ctx)

    return nil
}

const (
    openConnectionTimeout = 10 * time.Second
```

```
    reconnectionTimeout   = 2 * time.Second
)

func (k *krakenService) openConnection(logger logrus.FieldLogger) (*websocket.Conn,
error) {
    u := url.URL{Scheme: "wss", Host: k.config.BaseURI, Path: k.config.WSEndpoint}
    logger.Infof("connecting to %s", u.String())

    ctx, ctxCancel := context.WithTimeout(context.Background(), openConnectionTimeout)
    defer ctxCancel()

    for {
        select {
        case <-ctx.Done():
            return nil, ctx.Err()
        default:
            conn, _, err := websocket.DefaultDialer.Dial(u.String(), nil)
            if err != nil {
                logger.Infof("reconnecting: %v", err)
                time.Sleep(reconnectionTimeout)
                continue
            }

            logger.Infof("connected to Kraken WebSocket")
            return conn, nil
        }
    }
}

var candles chan interface{}

func (k *krakenService) Candles(ctx context.Context, logger logrus.FieldLogger,
productID string, period string) (<-chan interface{}, error) {
    err := k.connect(ctx, logger, productID, period)
    if err != nil {
        return nil, err
    }
    return candles, nil
}
```

“pkg/kraken/order.go”:

```
package kraken

import (
    "crypto/hmac"
    "crypto/sha256"
    "crypto/sha512"
    "encoding/base64"
    "encoding/json"
    "errors"
    "fmt"
```

```
    "io"
    "net/http"
    "net/url"
    "strconv"
    "strings"
    "time"
)

const clientRequestTimeout = 10 * time.Second

var (
    ErrSignRequest      = errors.New("sign request error")
    ErrCreateNewRequest = errors.New("cannot create new request")
    ErrSendRequest      = errors.New("send request error")
    ErrBadStatusCode    = errors.New("bad status code error")
    ErrResponseBodyRead = errors.New("response body read error")
)

type OrderRequest struct {
    OrderType  string // ioc
    Symbol     string // ticker's symbol
    Side       string // buy or sell
    Size       float64
    LimitPrice float64
    // Not necessary for ioc order
    ReducedOnly   bool    `json:"reduceOnly"` // default false
    StopPrice     float64 `json:"stopPrice"`
    TriggerSignal string  `json:"triggerSignal"`
    ClientID      string  `json:"cliOrdID"`
}

// type OrderResponse struct {
//  Result       string       `json:"result"`
//  UID          string       `json:"uid"`
//  Status       string       `json:"status"`
//  ServerTime   time.Time    `json:"serverTime"`
//  RecievedTime time.Time    `json:"receivedTime"`
//  Errors       []Error      `json:"errors"`
//  OrderEvents  []OrderEvent `json:"orderEvents"`
// }

// type Error struct {
//  Code    float64 `json:"code"`
//  Message string  `json:"message"`
// }

type OrderResponse struct {
    Result     string     `json:"result"`
    SendStatus SendStatus `json:"sendStatus"`
    ServerTime time.Time  `json:"serverTime"`
    Error      string     `json:"error"`
}
```

```go
type SendStatus struct {
    OrderID      string       `json:"order_id"`
    Status       string       `json:"status"`
    RecievedTime time.Time    `json:"recievedTime"`
    OrderEvents  []OrderEvent `json:"orderEvents"`
}

type OrderEvent struct {
    Type            string  `json:"type"`
    ReducedQuantity float64 `json:"reducedQuantity"` // for place and edit orders
    Order           Order   `json:"order"`           // except edit order
    UID             string  `json:"uid"`             // for reject and cancel orders
    // For reject order
    Reason string `json:"reason"`
    // For execution order
    Amount               float64 `json:"amount"`
    Price                float64 `json:"price"`
    ExecutionID          string  `json:"executionId"`
    TakerReducedQuantity float64 `json:"takerReducedQuantity"`
    OrderPriorEdit       Order   `json:"orderPriorEdit"`
    OrderPriorExecution  Order   `json:"orderPriorExecution"`
    // // For edit order
    // Old domain.Order `json:"old"`
    // New domain.Order `json:"new"`
}

type Order struct {
    OrderID             string    `json:"orderId"`
    ClientID            string    `json:"cliOrdID"`
    ReducedOnly         bool      `json:"reduceOnly"`
    Symbol              string    `json:"symbol"`
    Quantity            float64   `json:"quantity"`
    Side                string    `json:"side"`
    LimitPrice          float64   `json:"limitPrice"`
    StopPrice           float64   `json:"stopPrice"`
    Filled              float64   `json:"filled"`
    Type                string    `json:"type"`
    Timestamp           time.Time `json:"timestamp"`
    LastUpdateTimestamp time.Time `json:"lastUpdateTimestamp"`
}

func (k *krakenService) MakeSendBuyIOCOrderRequest(symbol string, size float64,
limitPrice float64) (*OrderResponse, error) {
    return k.MakeSendOrderRequest(OrderRequest{
        OrderType:  "ioc",
        Symbol:     symbol,
        Side:       "buy",
        Size:       size,
        LimitPrice: limitPrice,
    })
}
```

```go
func (k *krakenService) MakeSendSellIOCOrderRequest(symbol string, size float64,
limitPrice float64) (*OrderResponse, error) {
    return k.MakeSendOrderRequest(OrderRequest{
        OrderType:  "ioc",
        Symbol:     symbol,
        Side:       "sell",
        Size:       size,
        LimitPrice: limitPrice,
    })
}

func (k *krakenService) MakeSendOrderRequest(order OrderRequest) (*OrderResponse,
error) {
    req, err := k.createSendOrderRequest(order)
    if err != nil {
        return nil, err
    }

    client := http.Client{
        Timeout: clientRequestTimeout,
    }

    res, err := client.Do(req)
    if err != nil {
        return nil, fmt.Errorf("%w: %v", ErrSendRequest, err)
    }
    defer res.Body.Close()

    if !(res.StatusCode >= 200 && res.StatusCode < 300) {
        return nil, ErrBadStatusCode
    }

    b, err := io.ReadAll(res.Body)
    if err != nil {
        return nil, fmt.Errorf("%w: %v", ErrResponseBodyRead, err)
    }

    var response OrderResponse
    err = json.Unmarshal(b, &response)
    if err != nil {
        return nil, fmt.Errorf("%w: %v", ErrResponseBodyRead, err)
    }

    return &response, nil
}

func (k *krakenService) createSendOrderRequest(order OrderRequest) (*http.Request,
error) {
    endpointPath := k.config.RestEndpoint + "/sendorder"
    u := url.URL{Scheme: "https", Host: k.config.BaseDemoURI, Path: endpointPath}

    // Query parameters
    v := url.Values{}
```

```
    v.Add("orderType", order.OrderType)
    v.Add("symbol", order.Symbol)
    v.Add("side", order.Side)
    v.Add("size", strconv.FormatFloat(order.Size, 'f', -1, 64))
    v.Add("limitPrice", strconv.FormatFloat(order.LimitPrice, 'f', -1, 64))
    queryString := v.Encode()

    // Creating request
    req, err := http.NewRequest(http.MethodPost, u.String()+"?"+queryString, nil)
    if err != nil {
        return nil, ErrCreateNewRequest
    }

    // Nonce
    nonce := strconv.FormatInt(time.Now().UnixMilli(), 10)

    // Authentication
    authent, err := k.signRequest(endpointPath, nonce, queryString)
    if err != nil {
        return nil, ErrSignRequest
    }

    // Headers
    req.Header.Add("APIKey", k.config.APIKey)
    req.Header.Add("Authent", authent)
    req.Header.Add("Nonce", nonce)

    return req, nil
}

func (k *krakenService) signRequest(endpoint string, nonce string, postData string)
(string, error) {
    endpoint = strings.TrimPrefix(endpoint, "/derivatives")
    message := []byte(postData + nonce + endpoint)

    sha := sha256.New()
    sha.Write(message)

    secretDecoded, err := base64.StdEncoding.DecodeString(k.config.APISecret)
    if err != nil {
        return "", err
    }

    h := hmac.New(sha512.New, secretDecoded)
    h.Write(sha.Sum(nil))

    return base64.StdEncoding.EncodeToString(h.Sum(nil)), nil
}
```

“pkg/kraken/kraken.go”:

```
package kraken

import (
    "context"
    "net/http"

    "github.com/sirupsen/logrus"
)

//go:generate mockgen -source=kraken.go -destination=mocks/kraken.go

type KrakenService interface {
    Candles(ctx context.Context, logger logrus.FieldLogger, productID string, period
string) (<-chan interface{}, error)
    MakeSendOrderRequest(order OrderRequest) (*OrderResponse, error)
    createSendOrderRequest(order OrderRequest) (*http.Request, error)
    MakeSendBuyIOCOrderRequest(symbol string, size float64, limitPrice float64)
(*OrderResponse, error)
    MakeSendSellIOCOrderRequest(symbol string, size float64, limitPrice float64)
(*OrderResponse, error)
}

type krakenService struct {
    config *KrakenConfig
}

type KrakenConfig struct {
    BaseURI      string
    BaseDemoURI  string
    WSEndpoint   string
    RestEndpoint string
    APIKey       string
    APISecret    string
}

func NewKrakenService(config KrakenConfig) KrakenService {
    return &krakenService{
        config: &config,
    }
}
```

“pkg/telegram/telegram.go”:

```
package telegram

import (
    "errors"
    "fmt"
    "net/http"
    "net/url"
```

```
    "time"
)
//go:generate mockgen -source=telegram.go -destination=mocks/telegram.go

type TelegramService interface {
    Send(text string) error
}

type TelegramConfig struct {
    Token  string
    ChatID string
}

type telegramService struct {
    config TelegramConfig
}

func NewTelegramService(config TelegramConfig) TelegramService {
    return &telegramService{
        config: config,
    }
}

var ErrTelegramSendMessage = errors.New("telegram send message error")

const clientRequestTimeout = 10 * time.Second

func (t *telegramService) Send(text string) error {
    endpoint := "https://api.telegram.org/bot" + t.config.Token + "/sendMessage"

    v := url.Values{}
    v.Add("chat_id", t.config.ChatID)
    v.Add("text", text)
    queryString := v.Encode()

    req, err := http.NewRequest(http.MethodGet, endpoint+"?"+queryString, nil)
    if err != nil {
        return fmt.Errorf("%v: %w", ErrTelegramSendMessage, err)
    }

    client := http.Client{
        Timeout: clientRequestTimeout,
    }

    res, err := client.Do(req)
    if err != nil {
        return fmt.Errorf("%v: %w", ErrTelegramSendMessage, err)
    }

    if !(res.StatusCode >= 200 && res.StatusCode < 300) {
        return fmt.Errorf("%v: %w", ErrTelegramSendMessage, errors.New(res.Status))
    }
```

```
    return nil
}
```

“pkg/postgres/postgres.go”:

```
package postgres

import (
    "context"

    "github.com/jackc/pgx/v4"
    "github.com/jackc/pgx/v4/pgxpool"
    "github.com/sirupsen/logrus"
)

func NewPool(dsn string, logger logrus.FieldLogger) (*pgxpool.Pool, error) {
    poolConfig, err := pgxpool.ParseConfig(dsn)
    if err != nil {
        return nil, err
    }

    poolConfig.ConnConfig.Logger = &loggerAdapter{logger: logger}

    pool, err := pgxpool.ConnectConfig(context.Background(), poolConfig)
    if err != nil {
        return nil, err
    }

    return pool, nil
}

type loggerAdapter struct {
    logger logrus.FieldLogger
}

func (l *loggerAdapter) Log(_ context.Context, level pgx.LogLevel, msg string, data
map[string]interface{}) {
    switch level {
    case pgx.LogLevelTrace:
        l.logger.WithFields(data).Debugf(msg)
    case pgx.LogLevelDebug:
        l.logger.WithFields(data).Debugf(msg)
    case pgx.LogLevelInfo:
        l.logger.WithFields(data).Infof(msg)
    case pgx.LogLevelWarn:
        l.logger.WithFields(data).Warnf(msg)
    case pgx.LogLevelError:
        l.logger.WithFields(data).Errorf(msg)
    case pgx.LogLevelNone:
        l.logger.WithFields(data).Errorf(msg)
    }
```

“internal/service/algorithm.go”:

```
package service

import (
    "course-project/internal/domain"
    "course-project/internal/repository"
    "course-project/pkg/telegram"

    "context"
    "errors"
    "fmt"

    "github.com/sirupsen/logrus"
)

//go:generate mockgen -source=algorithm.go -destination=mocks/algorithm.go

type AlgorithmService interface {
    RunStochasticCross(ctx context.Context, period domain.CandlePeriod, symbol string,
size float64, limitCoef float64)
}

type algorithmService struct {
    logger    logrus.FieldLogger
    repo      repository.Repository
    exchange  ExchangeService
    indicator IndicatorService
    sender    telegram.TelegramService
}

func NewAlgorithmService(logger logrus.FieldLogger, repo repository.Repository,
exchange ExchangeService, indicator IndicatorService, sender telegram.TelegramService)
AlgorithmService {
    return &algorithmService{
        logger:    logger,
        repo:      repo,
        exchange:  exchange,
        indicator: indicator,
        sender:    sender,
    }
}

var (
    ErrOrderDBSave     = errors.New("order db save error")
    ErrCreateTemplate  = errors.New("create template error")
    ErrSendTextMessage = errors.New("send text message error")
)

func (a *algorithmService) GetPrices(ctx context.Context, productID string, period
domain.CandlePeriod) (<-chan float64, error) {
    return a.exchange.GetPrices(ctx, a.logger, productID, period)
}
```

```
func (a *algorithmService) BuyIOCOrder(symbol string, size float64, limitPrice
float64) error {
    orderResponse, err := a.exchange.MakeSendBuyIOCOrderRequest(symbol, size,
limitPrice)
    if err != nil {
        return err
    }

    return a.SendOrder(*orderResponse)
}

func (a *algorithmService) SellIOCOrder(symbol string, size float64, limitPrice
float64) error {
    orderResponse, err := a.exchange.MakeSendSellIOCOrderRequest(symbol, size,
limitPrice)
    if err != nil {
        return err
    }
    return a.SendOrder(*orderResponse)
}

func (a *algorithmService) SendOrder(orderResponse domain.OrderResponse) error {
    a.logger.Debugln("Writing to DB: ", orderResponse)
    if orderResponse.SendStatus.OrderEvents[0].Type != "REJECT" && orderResponse.Error
== "" {
        err := a.repo.CreateOrder(context.Background(),
orderResponse.SendStatus.OrderEvents[0].OrderPriorExecution)
        if err != nil {
            return fmt.Errorf("%w: %v", ErrOrderDBSave, err)
        }
    }

    text, err := orderResponse.ToText()
    if err != nil {
        return fmt.Errorf("%w: %v", ErrCreateTemplate, err)
    }

    a.logger.Debug("Sending to telegram bot")
    err = a.sender.Send(text)
    if err != nil {
        return fmt.Errorf("%w: %v", ErrSendTextMessage, err)
    }

    return nil
}

func (a *algorithmService) RunStochasticCross(ctx context.Context, period
domain.CandlePeriod, symbol string, size float64, limitCoef float64) {
    a.logger.Debug("RunStochasticCross")
    prices, err := a.GetPrices(ctx, symbol, period)
    if err != nil {
        a.logger.Error(err)
```

```
        return
    }
    <-prices

    currentPrice := <-prices
    highLimit := float64(int(currentPrice * (1 + limitCoef)))
    lowLimit := float64(int(currentPrice * (1 - limitCoef)))

    data := a.indicator.Stochastic(prices, 2)
    s := <-data

    var havePosition bool
    KIsHigherD := s.K > s.D

    for stochastic := range data {
        a.logger.Debug(stochastic)
        if stochastic.K > stochastic.D && !KIsHigherD {
            KIsHigherD = true
            if !havePosition {
                err := a.BuyIOCOrder(symbol, size, highLimit)
                if err != nil {
                    a.logger.Error(err)
                    return
                }
                havePosition = true
            }
        } else if stochastic.K < stochastic.D && KIsHigherD {
            KIsHigherD = false
            if havePosition {
                err := a.SellIOCOrder(symbol, size, lowLimit)
                if err != nil {
                    a.logger.Error(err)
                    return
                }
                havePosition = false
            }
        }
    }
}
```

“internal/service/exchange.go”:

```
package service

import (
    "course-project/internal/domain"
    "course-project/pkg/kraken"
    "strconv"

    "context"

    "github.com/sirupsen/logrus"
```

```
)

//go:generate mockgen -source=exchange.go -destination=mocks/exchange.go

type ExchangeService interface {
    GetPrices(ctx context.Context, logger logrus.FieldLogger, productID string, period
domain.CandlePeriod) (<-chan float64, error)
    MakeSendOrderRequest(order domain.OrderRequest) (*domain.OrderResponse, error)
    MakeSendBuyIOCOrderRequest(symbol string, size float64, limitPrice float64)
(*domain.OrderResponse, error)
    MakeSendSellIOCOrderRequest(symbol string, size float64, limitPrice float64)
(*domain.OrderResponse, error)
}

type exchangeService struct {
    kraken kraken.KrakenService
}

func NewExchangeService(config kraken.KrakenConfig) ExchangeService {
    return &exchangeService{
        kraken: kraken.NewKrakenService(config),
    }
}

func (w *exchangeService) GetPrices(ctx context.Context, logger logrus.FieldLogger,
productID string, period domain.CandlePeriod) (<-chan float64, error) {
    candles, err := w.kraken.Candles(ctx, logger, productID, string(period))
    if err != nil {
        return nil, err
    }

    prices := make(chan float64)
    go func() {
        var closePrice float64
        for candle := range candles {
            closePrice, err = strconv.ParseFloat(candle.(*kraken.Candle).Close, 64)
            if err == nil {
                prices <- closePrice
            }
        }
        close(prices)
    }()

    return prices, nil
}

func (w *exchangeService) MakeSendOrderRequest(order domain.OrderRequest)
(*domain.OrderResponse, error) {
    krakenOrder := kraken.OrderRequest{
        OrderType:  order.OrderType,
        Symbol:     order.Symbol,
        Side:       order.Side,
        Size:       order.Size,
```

```
        LimitPrice: order.LimitPrice,
    }

    krakenOrderResponse, err := w.kraken.MakeSendOrderRequest(krakenOrder)

    orderResponse := domain.OrderResponse{
        Result:     krakenOrderResponse.Result,
        ServerTime: krakenOrderResponse.ServerTime,
        Error:      krakenOrderResponse.Error,
        SendStatus: domain.SendStatus{
            OrderID:      krakenOrderResponse.SendStatus.OrderID,
            Status:       krakenOrderResponse.SendStatus.Status,
            RecievedTime: krakenOrderResponse.SendStatus.RecievedTime,
        },
    }

    for _, event := range krakenOrderResponse.SendStatus.OrderEvents {
        orderResponse.SendStatus.OrderEvents =
append(orderResponse.SendStatus.OrderEvents, domain.OrderEvent{
            Type:                event.Type,
            UID:                 event.UID,
            Reason:              event.Reason,
            ExecutionID:         event.ExecutionID,
            Price:               event.Price,
            Amount:              event.Amount,
            TakerReducedQuantity: event.TakerReducedQuantity,
            OrderPriorExecution: domain.Order{
                OrderID:     event.Order.OrderID,
                ReducedOnly: event.Order.ReducedOnly,
                Symbol:      event.Order.Symbol,
                Quantity:    event.Order.Quantity,
                Side:        event.Order.Side,
                Filled:      event.Order.Filled,
                Type:        event.Order.Type,
                Timestamp:   event.Order.Timestamp,
            },
        })
    }

    // for _, err := range krakenOrderResponse.Errors {
    //  orderResponse.Errors = append(orderResponse.Errors, domain.Error{
    //      Code:    err.Code,
    //      Message: err.Message,
    //  })
    // }

    // for _, event := range krakenOrderResponse.OrderEvents {
    //  orderResponse.OrderEvents = append(orderResponse.OrderEvents,
domain.OrderEvent{
    //      Type:                event.Type,
    //      UID:                 event.UID,
    //      Reason:              event.Reason,
    //      ExecutionID:         event.ExecutionID,
```

```
    //      Price:               event.Price,
    //      Amount:              event.Amount,
    //      TakerReducedQuantity: event.TakerReducedQuantity,
    //      OrderPriorExecution: &domain.Order{
    //          OrderID:     event.Order.OrderID,
    //          ReducedOnly: event.Order.ReducedOnly,
    //          Symbol:      event.Order.Symbol,
    //          Quantity:    event.Order.Quantity,
    //          Side:        event.Order.Side,
    //          Filled:      event.Order.Filled,
    //          Type:        event.Order.Type,
    //          Timestamp:   event.Order.Timestamp,
    //      },
    //  })
    // }

    return &orderResponse, err
}

func (w *exchangeService) MakeSendBuyIOCOrderRequest(symbol string, size float64,
limitPrice float64) (*domain.OrderResponse, error) {
    return w.MakeSendOrderRequest(domain.OrderRequest{
        OrderType:  "ioc",
        Symbol:     symbol,
        Side:       "buy",
        Size:       size,
        LimitPrice: limitPrice,
    })
}

func (w *exchangeService) MakeSendSellIOCOrderRequest(symbol string, size float64,
limitPrice float64) (*domain.OrderResponse, error) {
    return w.MakeSendOrderRequest(domain.OrderRequest{
        OrderType:  "ioc",
        Symbol:     symbol,
        Side:       "sell",
        Size:       size,
        LimitPrice: limitPrice,
    })
}
```

“internal/service/indicator.go”:

```
package service

import (
    "container/ring"
)

//go:generate mockgen -source=indicator.go -destination=mocks/indicator.go

type StochasticStruct struct {
```

```
    K float64
    D float64
}

type IndicatorService interface {
    Stochastic(prices <-chan float64, n int) <-chan *StochasticStruct
    EMA(prices <-chan float64, n int) <-chan float64
}

type indicatorService struct{}

func NewIndicatorService() IndicatorService {
    return &indicatorService{}
}

func (i *indicatorService) Stochastic(prices <-chan float64, n int) <-chan
*StochasticStruct {
    output := make(chan *StochasticStruct)

    go func() {
        var min, max float64
        r := ring.New(n)

        sendEMA := make(chan float64)
        getEMA := i.EMA(sendEMA, 3)

        for k := 0; k < n; k++ {
            r.Value = <-prices
            r = r.Move(1)
        }

        sendEMA <- r.Prev().Value.(float64)

        for price := range prices {
            r.Value = price
            max = price
            min = max

            for k := 0; k < n; k++ {
                elem := r.Value.(float64)
                if elem > max {
                    max = elem
                }
                if elem < min {
                    min = elem
                }
            }

            K := (price - min) / (max - min) * 100
            sendEMA <- K
            D := <-getEMA

            output <- &StochasticStruct{
```

```
                K: K,
                D: D,
            }
            r.Move(1)
        }
        close(sendEMA)
        close(output)
    }()

    return output
}

func (i *indicatorService) EMA(prices <-chan float64, n int) <-chan float64 {
    output := make(chan float64)
    alpha := 2 / (float64(n) + 1)

    go func() {
        x1, x2 := 0.0, <-prices
        for price := range prices {
            x1 = x2
            x2 = alpha*price + (1-alpha)*x1
            output <- x2
        }
        close(output)
    }()

    return output
}
```

“internal/service/service.go”:

```
package service

type Service struct {
    Algorithm AlgorithmService
}

func NewService(algorithmService AlgorithmService) *Service {
    return &Service{
        Algorithm: algorithmService,
    }
}
```

“internal/repository/ repository.go”:

```
package repository

import (
    "course-project/internal/domain"
```

```
    "context"

    "github.com/jackc/pgx/v4/pgxpool"
)

type Repository interface {
    CreateOrder(ctx context.Context, order domain.Order) error
}

type repo struct {
    pool *pgxpool.Pool
}

func NewRepository(pool *pgxpool.Pool) Repository {
    return &repo{
        pool: pool,
    }
}

const createOrderQuery = `INSERT INTO orders(order_id, order_type, symbol, side,
quantity, filled, order_time, reduced_only)
VALUES ($1, $2, $3, $4, $5, $6, $7, $8);`

func (r *repo) CreateOrder(ctx context.Context, order domain.Order) error {
    _, err := r.pool.Exec(ctx, createOrderQuery,
        order.OrderID,
        order.Type,
        order.Symbol,
        order.Side,
        order.Quantity,
        order.Filled,
        order.Timestamp,
        order.ReducedOnly,
    )
    if err != nil {
        return err
    }

    return nil
}
```

“internal/handler/handler.go”:

```
package handler

import (
    "course-project/internal/domain"
    "course-project/internal/service"

    "context"
    "net/http"
    "strconv"
```

```
    "github.com/go-chi/chi"
    "github.com/go-chi/chi/middleware"
    "github.com/sirupsen/logrus"
)

type Handler struct {
    serverCtx     context.Context
    serverStopCtx context.CancelFunc
    logger        logrus.FieldLogger
    services      *service.Service
}

func NewHandler(serverCtx context.Context, serverStopCtx context.CancelFunc, logger
logrus.FieldLogger, services *service.Service) *Handler {
    return &Handler{
        serverCtx:     serverCtx,
        serverStopCtx: serverStopCtx,
        logger:        logger,
        services:      services,
    }
}

func (h *Handler) InitRoutes() *chi.Mux {
    r := chi.NewRouter()
    r.Use(middleware.Logger)

    r.Get("/run", h.Run)
    r.Get("/stop", h.Stop)

    return r
}

func (h *Handler) Run(w http.ResponseWriter, r *http.Request) {
    productID := r.URL.Query().Get("productID")
    if productID == "" {
        w.WriteHeader(http.StatusBadRequest)
    }

    period := r.URL.Query().Get("period")
    if period == "" {
        w.WriteHeader(http.StatusBadRequest)
    }

    size := r.URL.Query().Get("size")
    sz, err := strconv.ParseFloat(size, 64)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
    }

    limitCoef := r.URL.Query().Get("limitCoef")
    lc, err := strconv.ParseFloat(limitCoef, 64)
    if err != nil {
```

```
        w.WriteHeader(http.StatusBadRequest)
    }

    go h.services.Algorithm.RunStochasticCross(h.serverCtx,
domain.CandlePeriod(period), productID, sz, lc)
    w.WriteHeader(http.StatusOK)
}

func (h *Handler) Stop(w http.ResponseWriter, r *http.Request) {
    h.serverStopCtx()
    w.WriteHeader(http.StatusOK)
}
```

“configs/congifSetup.go”:

```
package configs

import (
    "course-project/pkg/kraken"
    "course-project/pkg/telegram"

    "errors"
    "fmt"
    "os"

    "github.com/joho/godotenv"
    "github.com/spf13/viper"
)

var (
    ErrInitConfig     = errors.New("init config error")
    ErrKrakenConfig   = errors.New("kraken config error")
    ErrTelegramConfig = errors.New("telegram config error")
    ErrDBConfig       = errors.New("db config error")
)

func init() {
    viper.AddConfigPath("./configs")
    viper.SetConfigName("config")
}

type AppConfig struct {
    Host     string
    Port     string
    Kraken   kraken.KrakenConfig
    Telegram telegram.TelegramConfig
    DB       DBConfig
}

type DBConfig struct {
    Server   string
    User     string
```

```
    Password string
    Host     string
    Port     string
    DBName   string
    SSLMode  string
}

func (d *DBConfig) DNS() string {
    return fmt.Sprintf("%v://%v:%v@%v:%v/%v?sslmode=%v",
        d.Server,
        d.User,
        d.Password,
        d.Host,
        d.Port,
        d.DBName,
        d.SSLMode,
    )
}

func NewAppConfig() (*AppConfig, error) {
    // Init
    err := viper.ReadInConfig()
    if err != nil {
        return nil, ErrInitConfig
    }

    err = godotenv.Load()
    if err != nil {
        return nil, ErrInitConfig
    }

    // Kraken config
    APIKey := os.Getenv("API_KEY")
    if APIKey == "" {
        return nil, ErrKrakenConfig
    }

    APISecret := os.Getenv("API_SECRET")
    if APISecret == "" {
        return nil, ErrKrakenConfig
    }

    krakenConfig := kraken.KrakenConfig{
        BaseURI:      viper.GetString("kraken.base_uri"),
        BaseDemoURI:  viper.GetString("kraken.base_demo_uri"),
        WSEndpoint:   viper.GetString("kraken.ws_endpoint"),
        RestEndpoint: viper.GetString("kraken.rest_endpoint"),
        APIKey:       APIKey,
        APISecret:    APISecret,
    }

    // Telegram config
    token := os.Getenv("BOT_TOKEN")
```

```
    if token == "" {
        return nil, ErrTelegramConfig
    }

    chatID := os.Getenv("CHAT_ID")
    if chatID == "" {
        return nil, ErrTelegramConfig
    }

    telegramConfig := telegram.TelegramConfig{
        Token:  token,
        ChatID: chatID,
    }

    // DB config
    password := os.Getenv("DB_PASSWORD")
    if password == "" {
        return nil, ErrDBConfig
    }

    dbConfig := DBConfig{
        Server:   viper.GetString("db.server"),
        User:     viper.GetString("db.user"),
        Password: password,
        Host:     viper.GetString("db.host"),
        Port:     viper.GetString("db.port"),
        DBName:   viper.GetString("db.dbname"),
        SSLMode:  viper.GetString("db.sslmode"),
    }

    // App config
    appConfig := AppConfig{
        Host:     viper.GetString("host"),
        Port:     viper.GetString("port"),
        Kraken:   krakenConfig,
        Telegram: telegramConfig,
        DB:       dbConfig,
    }

    return &appConfig, nil
}
```

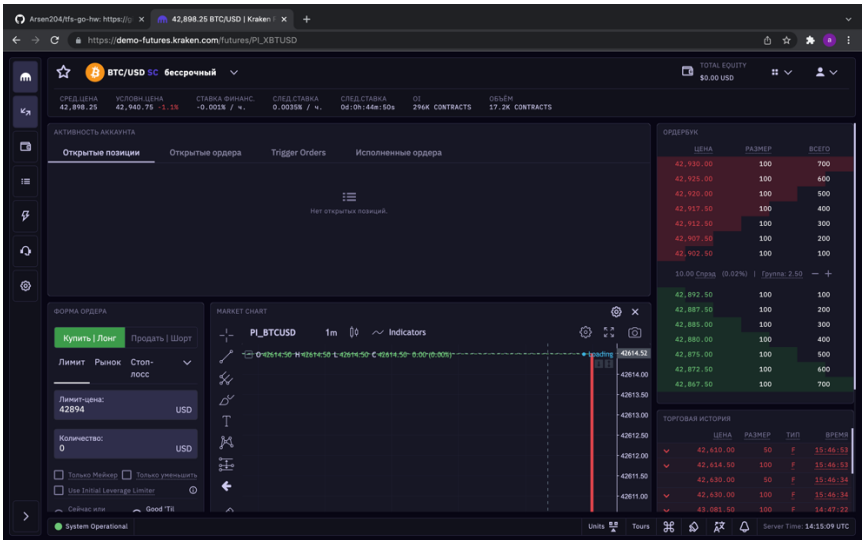
“cmd/main.go”:

```
package main

import (
    "course-project/configs"
    "course-project/internal/domain"
    "course-project/internal/handler"
    "course-project/internal/repository"
    "course-project/internal/server"
```

```
    "course-project/internal/service"
    pkgpostgres "course-project/pkg/postgres"
    "course-project/pkg/telegram"

    "context"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "time"

    "github.com/sirupsen/logrus"
)

const shutdownTimeout = 30 * time.Second

func main() {
    // Logger init
    logger := logrus.New()
    logger.SetLevel(logrus.DebugLevel)
    // logger.Formatter = &logrus.JSONFormatter{}

    // Templates init
    err := domain.InitTemplate()
    if err != nil {
        logger.Error(err)
    }

    // Configs init
    config, err := configs.NewAppConfig()
    if err != nil {
        logger.Fatal(err)
    }

    // Repository
    pool, err := pkgpostgres.NewPool(config.DB.DNS(), logger)
    if err != nil {
        logger.Fatalf("failed to initialize db: %s", err.Error())
    }
    defer pool.Close()

    repo := repository.NewRepository(pool)

    // Server
    srv := new(server.Server)
    serverCtx, serverStopCtx := context.WithCancel(context.Background())

    // Dependencies
    telegramService := telegram.NewTelegramService(config.Telegram)

    exchangeService := service.NewExchangeService(config.Kraken)
    indicatorService := service.NewIndicatorService()
```

```
    algorithmService := service.NewAlgorithmService(logger, repo, exchangeService,
indicatorService, telegramService)

    services := service.NewService(algorithmService)
    handlers := handler.NewHandler(serverCtx, serverStopCtx, logger, services)

    // Running App
    logger.Print("app running")

    go func() {
        err := srv.Run(config.Port, handlers.InitRoutes())
        if err != http.ErrServerClosed {
            logger.Fatalf("error occurred while running http server: %s", err.Error())
        }
    }()

    quit := make(chan os.Signal, 1)
    signal.Notify(quit, syscall.SIGTERM, syscall.SIGINT)
    go func() {
        <-quit
        serverStopCtx()
    }()

    <-serverCtx.Done()

    logger.Print("app shutting down")

    shutdownCtx, shutdownStopCtx := context.WithTimeout(context.Background(),
shutdownTimeout)
    defer shutdownStopCtx()

    if err := srv.Shutdown(shutdownCtx); err != nil {
        logger.Errorf("error occurred on server shutting down: %s", err.Error())
    }
}
```

Экранные формы с примерами выполнения программы:



Result: success

OrderID: 9e3c78ca-126b-4a6b-affc-99b56d8161f5
Status: placed
RecievedTime: 0001-01-01 00:00:00 +0000 UTC

Type: EXECUTION

Amount: 1
Price: 58578.5

ServerTime: 2021-12-01 15:30:00.161 +0000 UTC

18:28

Result: success

OrderID: a801ffe8-691f-40ce-ae0d-6c34be44982f
Status: placed
RecievedTime: 0001-01-01 00:00:00 +0000 UTC

Type: EXECUTION

Amount: 1
Price: 58657

ServerTime: 2021-12-01 15:31:00.642 +0000 UTC

18:29

Result: success

OrderID: 00bd2e51-5385-4d94-b4dd-e78120436098
Status: placed
RecievedTime: 0001-01-01 00:00:00 +0000 UTC

Type: EXECUTION

Amount: 1
Price: 58761

ServerTime: 2021-12-01 15:34:00.741 +0000 UTC

18:32