Выполнение практических работ и КР по предмету Разработка АС реального времени

В течении семестра вам предстоит разработать автоматизированную систему для некоторого вида производства. Рассмотрим пример. Допустим вам необходимо по заданию разработать автоматизированную систему реального времени для производства кирпичей. Что из себя будет представлять такая система, скорее всего это будет зависеть от поставленных целей. Далее приведен пример основных функций таких систем: *мониторинг производственных линий, сбор данных с датчиков IoT, управление процессами в реальном времени и предоставление интерфейсов сотрудникам и программам для анализа производительности и отслеживания работы оборудования.* Для того, чтобы ускорить процесс разработки вы можете использовать для реализации фреймворк Spring). Например, вы можете использовать следующие компоненты Spring Framework, в частности, Spring Boot, Spring WebSocket, Spring Kafka, Spring Data, Spring Security и другие модули, что позволит вам создать гибкое, масштабируемое и интерактивное решение. Рассмотрим порядок разработки АС.

Порядок разработки АС реального времени

1. **Начать необходимо с исследования самого процесса производства и его описания.**
    a. **Из каких этапов состоит процесс производства?**
    b. **Из какого оборудования (станков, печей, транспортировочных устройств и т.д.) состоит линия производства?**
    c. **Какие участки нуждаются в автоматизации?**
    d. **Какие параметры на них необходимо контролировать (температура в печи, скорость конвейера и т.д.)?**
    e. **Какими устройствами на них необходимо управлять (открывать и закрывать заслонку, изменять скорость конвейера, открывать клапан сброса давления и т.д.)?**
    f. **Какие виды сотрудников есть на предприятии?**
    g. **Какая информация нужна этим сотрудникам? В каких процессах на производстве они задействованы? Какие интерфейсы взаимодействия с системой нужны этим сотрудникам?**
    h. **Какую информацию с линии производства необходимо собирать и обрабатывать?**

    Примечание: на данном этапе всю эту информацию необходимо собрать и структурировать для понимания производственного процесса и формирования функциональных требований к системе. Пока нет необходимости собирать ее в конкретных нотациях, таких как IDEF0, DFD и т.д. Это необходимо будет сделать позже.
2. **Инициация разработки, начинается с целей (ответ на вопрос ЗАЧЕМ?)**

Для начала вам нужно определились цель вашей АС - для чего система создается и затем описать требуемый функционал системы.

Например, для АС реального времени по производству кирпичей это может быть следующий функционал:

1. Автоматизировать процесс управления оборудованием: управление потоком сырья, производить контроль температуры обжига, влажности, формовочных параметров.

2. Производить мониторинг состояния оборудования и показаний датчиков в реальном времени: сбор данных с производственных линий (температура, давление, сырье).

3. Обеспечить быстродействие системы: то есть поддерживать минимальную задержку в обмене данными для оптимальной работы всех производственных процессовпроцессов.

4. Поддерживать в системе уведомления в реальном времени и работу с аналитикой: выявлять отклонения, отправлять предупреждения, рассчитывать производственные KPI.

5. Обеспечить информационную безопасность системы: ограничить доступ к системе для несанкционированных пользователей.

На этом шаге, вам нужно на основе целей выявить требования к вашей АС, для описания функциональных требований рекомендуется использовать нотацию UML (users story, use case)

Примечание: функциональные требования к системе в данном разделе формируются на основе собранной вами информации из п.1.

3. **Проектирование системы. Необходимо смоделировать производственный процесс с учетом внедрения системы. (Разработать диаграмму IDEF0).**

4. **Разработка архитектуры системы. Необходимо продумать архитектуру системы и описать взаимодействие компонентов внутри нее. (Разработать диаграммы DFD, ERD).**

Здесь нам нужно определить основные компоненты нашей будущей системы или ее архитектуру. Система будет состоять из следующих ключевых компонентов:

1. Датчики IoT и PLC (программируемые логические контроллеры) для мониторинга производственного процесса/процессов. (Обдумать архитектуру сети датчиков и контроллеров, зачастую делают один контроллер на одну зону или устройство на линии производства)

2. Обработчик событий в реальном времени (на основе Kafka/WebSocket) для сбора и обработки данных.

3. Серверное приложение (Backend), реализованное с использованием Spring Framework для управления процессом и аналитики.

4. Панель управления (Frontend) для визуализации данных и взаимодействия с пользователем/пользователями.

5. База/базы данных(SQL или NoSQL) для хранения временных меток, информации о процессах и аналитики.

Мы перечислили основные компоненты, которые вам необходимо поэтапно реализовать в своей работе. После определения архитектуры можно перейти к разработке АС.

Примечание: продумав архитектуру системы по пунктам выше приступайте к разработке DFD диаграммы. С помощью нее вы сможете описать движение информационных потоков (данных) между вашими компонентами системы.

Пример: показания идут от датчика на PLC контроллер, оттуда команды могут передаваться на управляющие устройства, а также на бекенд сервер. Бекенд сервер в свою очередь уже их обрабатывает, передает на frontend, а также в БД и внешние

сервисы при необходимости. Это только один из возможных потоков данных, описанный очень кратко. Все это надо продумывать очень детально.

Также с помощью нее вы сможете более детально продумать процесс обработки данных в PLC контроллере и на бекенд сервере, продумать какие приходят в систему входные и выходные данные, а также какие методы (сервисы, репозитории, контроллеры, компораторы и т.д.) нужны для работы с ними в вашем коде, а также на какие микросервисы можно разбить ваш код.

После разработки DFD диаграммы продумайте более детально модели данных, с помощью которых вы будете хранить информацию на уровне БД, а также обрабатывать ее в коде. Для этого постройте диаграмму ERD, где продумайте все необходимые вам сущности и связи между ними.

5. Процесс разработки автоматизированной системы

Рассмотрим шаги по разработке, которые вам предстоит выполнить, чтобы реализовать требуемый функционал АС.

1) Архитектура сервера обработки данных.

Мы с вами будет использовать микросервисную архитектуру. Это означает, что наш код будет разделен на небольшие отдельные сервисы (простыми словами программы, которые одновременно работают на сервере), каждый из которых отвечает за свой функционал. Эти сервисы будут общаться между собой, а также со внешними системами

Грубо можно разделить код, отвечающий за обработку данных на сервере на ТРИ слоя.

a. Слой «контроллеров». Этот слой отвечает за Java классы, которые обрабатывают входные потоки данных из тех или иных источников. Это могут быть данные, полученные из брокера KAFKA, тогда это будет класс с методом «слушающим» брокер (этот метод может запуститься в любой момент, как только в брокере появится для него новое сообщение). Также это могут быть промышленные данные, переданные по промышленному протоколу MQTT, схема работы этого метода такая же. Также таким обработчиком может служить и REST API контроллер, который ожидает своего вызова клиентов (получает от клиента на вход HTTP запрос), затем обрабатывает данные и отдает клиенту HTTP ответ. В данном случае, в отличие от KAFKA и MQTT клиент, вызвавший этот метод обязательно получает ответ, подробнее о принципах работы с KAFKA, MQTT и HTTP вы можете прочитать самостоятельно, также ниже будут приведены соответствующие примеры кода.
b. Слой «сервисов». На данный момент мы получили входные данные извне, первоначально обработали их и извлекли из них всю нужную нам информацию. Далее эти данные передаются в классы сервисы, где описаны методы обработки этой информации. Сервисы описывают исключительно бизнес-логику приложения (алгоритмы обработки, сортировки, и т.д.), чтобы не перегружать этим более технический код. Но алгоритмы не могут работать изолированно с полученными данными. Иногда им для работы нужны какие-то уже сохраненные в системе данные, а иногда им самим нужно сохранить какие-то важные данные в

системе или передать их в другой микросервис, если там нужно выполнить другие операции на основе этой информации. Тут мы переходим к последнему уровню системы.

c. Слой «репозиториев». В данном случае мы работает с классами Java, которые отвечают за взаимодействие с БД, а также отправкой сообщений в брокер. В случае с классами БД описываются методы для получения информация (SELECT), а также для сохранения и обновления информации (INSERT, UPDATE). В случае с брокерами описываются методы, которые из данных формируют сообщения и отправляют их в очередь брокера.
d. Примечания. Для взаимодействия с внешними интерфейсами (Frontend) классически используются протоколы HTTP и WebSocket. Для взаимодействия с датчиками и ПЛК используется промышленный протокол MQTT, для взаимодействия микросервисов между собой и передачи между ними сообщений используется KAFKA. Также важно отметить, что KAFKA использует отправка сообщений без гарантии доставки. Если первый микросервис отправляет второму сообщение, то далее он не ждет никакого ответа, ему не важно получил ли второй микросервис это сообщение и удалось ли ему правильно обработать данные из него. Если в вашей связи между двумя микросервисами ВАЖНЫ ГАРАНТИЯ ДОСТАВКИ, А ТАКЖЕ РЕЗУЛТАТ ОБРАБОТКИ сообщения, то лучше использовать HTTP протокол для связи двух микросервисов, где один сервис выступает «клиентом», а второй «сервером». Однако такое соединение замедляет работу системы по сравнению с соединением через брокера, поэтому всегда необходимо обдумывать целесообразность того или иного подхода.

2) Введение в Spring

В Spring компоненты (Bean’ы) управляются IoC-контейнером, который создает, настраивает и управляет их жизненным циклом. Компоненты помечаются аннотациями:

- `@Component` – общий класс Bean’а.
- `@Service` – бизнес-логика.
- `@Repository` – работа с базой данных.

Для внедрения зависимостей используется `@Autowired`, который автоматически связывает Bean’ы внутри контейнера. Bean’ы можно конфигурировать вручную через `@Bean` в `@Configuration` классе.

Вот простой пример использования компонентов (Bean’ов) в Spring:

```
// Конфигурация приложения
@Configuration
@ComponentScan(basePackages = "com.example")
class AppConfig {}

// Репозиторий (работа с базой данных)
@Repository
class UserRepository {
```

```
    public String getUser() {
        return "John Doe";
    }
}

// Сервис (бизнес-логика)
@Service
class UserService {
    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public String fetchUser() {
        return userRepository.getUser();
    }
}

// Компонент (может быть использован в любом месте)
@Component
class UserComponent {
    private final UserService userService;

    @Autowired
    public UserComponent(UserService userService) {
        this.userService = userService;
    }

    public void printUser() {
        System.out.println(userService.fetchUser());
    }
}
```

В этом примере `@Component`, `@Service`, `@Repository` используются для управления Bean'ами, а `@Autowired` внедряет зависимости автоматически.

3) ***Шаг 1. Получение данных***

Выбор данных

Вам нужны данные, которые вы будете обрабатывать, следовательно вам нужно определить какие данные вы будете собирать.

Сбор данных с датчиков в реальном времени осуществляется с оборудования, либо вам может помочь ресурс https://www.kaggle.com/datasets, там вы можете поискать датасеты с нужными вам данными, вы также можете сэмулировать нужные данные с помощью метода Монте Карло

Для того чтобы собрать данные с реального оборудования, вам понадобятся обеспечить взаимодействие с ним.

Для взаимодействия с машинами и датчиками используются протоколы, такие как MQTT, Modbus или OPC-UA. Эти данные будут поступать на промежуточный обработчик (например, Apache Kafka), а затем обрабатываться микросервисом Spring, в зависимости от выбранной вами архитектуры.

Интеграция с датчиками через Spring

Как вам может помочь Spring? Вы можете использовать модуль Spring Integration& Используйте Spring Integration или Spring Boot для обработки входящих данных.

Например, для MQTT:

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-integration</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-mqtt</artifactId>
</dependency>
```

Настройка MQTT:

```java
@Configuration
public class MqttConfig {

  @Bean
  public MqttPahoMessageDrivenChannelAdapter mqttAdapter() {
    return new
MqttPahoMessageDrivenChannelAdapter("tcp://broker.hivemq.com:1883", "mqtt-client");
  }

  @ServiceActivator(inputChannel = "mqttInputChannel")
  public void handleMessage(String payload) {
    System.out.println("Получено сообщение от датчика: " + payload);
    // Логика обработки
  }
}
```

После того как вы определитесь с оборудованием, данными и интерфейсами для их получения, вам нужно каким-то образом их обрабатывать.

4) ***Шаг 2. Обработка данных в реальном времени***

У вас теперь есть данные, и вы знаете откуда и как их получать. Теперь их нужно обрабатывать. Как вам может помочь Spring?

Для построения системы обработки данных в реальном времени можно использовать Spring Kafka Streams.

Apache Kafka обеспечит передачу сообщений между компонентами системы без потерь и с минимальными задержками. Каждое сообщение может содержать данные о состоянии оборудования, таких как влажность, температура или статус производственной линии.

Пример. Разработка микросервиса обработки производственных данных:

- В `pom.xml` добавьте Kafka:

```xml
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

- *Определите топики Kafka:*
  - `raw-data` – для необработанных данных с датчиков.
  - `alerts` – для уведомлений о проблемах.
  - `stats` – для метрик и аналитики.

- *Пример обработки данных:*
```java
@Configuration
public class KafkaProcessor {

    @Bean
    public KStream<String, SensorData> processStream(StreamsBuilder builder) {
        KStream<String, SensorData> input = builder.stream("raw-data");

        KStream<String, SensorData> filtered = input.filter(
            (key, value) -> value.getTemperature() > 100 // Условие, например, превышение температуры
        );

        filtered.to("alerts");
        return input;
    }
}
```

5) *Шаг 3. Мониторинг состояния системы (WebSocket) в реальном времени*

Теперь у вас есть данные, вы можете их обрабатывать, но вам нужно реализовать функцию мониторинга, причем *в реальном времени*. Как вам может помочь Spring ?

Для отображения текущего состояния производственной линии и уведомлений в интерфейсе панели управления вы можете использовать Spring WebSocket. Это позволит передавать обновления в режиме реального времени на фронтенд. Эта технология поваляет установить двухстороннюю связь клиента с сервером и передавать данные в дуплексном режиме и представляет собой протокол взаимодействия транспортного уровня (TCP). Она нам как раз инужна чтобы передавать в режиме реального времени, с помощью htpp вы этого не получите.

Пример Spring WebSocket

- *Настроить WebSocket:*
```java
@Configuration
@EnableWebSocket
```

```
public class WebSocketConfig implements WebSocketConfigurer {
    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(new MonitoringWebSocketHandler(), "/monitoring")
                .setAllowedOrigins("*");
    }
}
```

- *Обработчик WebSocket:*

```java
public class MonitoringWebSocketHandler extends TextWebSocketHandler {
    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message)
throws Exception {
        String payload = message.getPayload();
        // Логика для отправки обновлений с датчиков
        session.sendMessage(new TextMessage("Current Status: " + payload));
    }
}
```

6) Шаг 4. Хранение данных

Производственные системы часто работают с большими объемами данных. Вы можете использовать *Time-Series базы данных* (например, InfluxDB) для временных рядов данных или *PostgreSQL/MySQL* для обработки метрик

Примеры обработки метрик:

- Для временных данных: `sensor_data` (температура, влажность, давление и т.д.).
- Для критических ошибок: `alerts`.

Как вам может помочь в этом Spring? Вы можете использовать модуль Spring Data JPA для работы с БД. (если не используете, но используете Java, то это интерфейсы JDBC см лекции и методичку по практическим)

Пример сущности с использованием Spring Data JPA:

```java
package com.example.gofarbot.models;


import com.fasterxml.jackson.annotation.JsonInclude;
import jakarta.persistence.*;
import jakarta.validation.constraints.NotNull;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
```

```java
@Data
@NoArgsConstructor
@Builder
@AllArgsConstructor
@Entity
@Table(name = "conferences")
public class Conference {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    private String link;

    @NotNull
    private String name;

    @NotNull
    private LocalDateTime timeOfConference;

    @OneToMany(fetch = FetchType.EAGER)
    @JoinColumn(name = "conference")
    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private List<UserRegistration> registrations;

    @ManyToMany(fetch = FetchType.EAGER, targetEntity = Notification.class)
    @JoinTable(
            name = "conferences_notifications_rel",
            joinColumns = {@JoinColumn(name = "conference")},
            inverseJoinColumns = {@JoinColumn(name = "notification")}
    )
    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private List<Notification> notifications;

    public List<User> getUsers() {
        List<UserRegistration> registrations = this.registrations;
        List<User> users = new ArrayList<>();
        for(UserRegistration registration: registrations) {
            users.add(registration.getUser());
        }
        return users;
    }
}
```

```

Пример репозитория с использованием Spring Data JPA

Примечание: обратите внимание на три возможности реализации метода запроса к БД:

А) Реализация ТОЛЬКО через «название». Для стандартных запросов типа поиска по значению параметра можно лишь правильно указать сигнатуру метода. Далее Spring сам поймет какой запрос вам нужен.

В) Реализация через JPQL. В данном случае можно писать SQL код в аннотации @Query на специальном языке JPQL, который работает сразу с сущностями, написанными на классах Java. Условный гибрид из SQL и Java.
```

C) Реализация «чистым» SQL запросом. Также пишется через аннотацию @Query с флагом nativeQuery=true. Позволяет выполнить чистый SQL запрос, но все равно автоматически проверяет на наличие инъекций и осуществляет преобразование данных в объекты Java.

```
package com.example.gofarbot.data;


import com.example.gofarbot.models.Message;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;

@Repository
public interface MessageRepository extends CrudRepository<Message, Long> {

    Optional<Message> findByCode(String code);

    @Query("""
        SELECT m
        FROM Message m
        JOIN m.buttons b
        WHERE b.code = :actualMessageCode
    """)
    List<Message> findBackMessages(@Param("actualMessageCode") String
actualMessageCode);

    @Query(value = """
        SELECT *
        FROM messages m
        WHERE m.next_message = :next_message_id
        LIMIT 1
    """, nativeQuery = true)
    Optional<Message> findPreviousMessageInChain(@Param("next_message_id")
long messageId);
}
```

7) *Шаг 5. Безопасность и управление доступом*

Для обеспечения требований по безопасности, что очень важно для производственной системы, вы можете использовать модуль Spring Security для авторизации и аутентификации пользователей системы.

Например ваша система может реализовывать следующие функции:

- Операторы могут управлять оборудованием.
- Технические специалисты получают доступ к данным мониторинга.
- Руководство может видеть статистику и выводы.

Пример настройки Spring Security:

```java
```

1) Реализуем сущность «Пользователя» с защищенным доступом. Обратите внимание на наследование от стандартного класса UserDetails и переписанные методы этого класса.

```
2) package com.example.gofarbot.models;

   import jakarta.persistence.*;
   import lombok.AllArgsConstructor;
   import lombok.Data;
   import lombok.NoArgsConstructor;
   import org.springframework.security.core.GrantedAuthority;
   import
   org.springframework.security.core.authority.SimpleGrantedAuthority;
   import org.springframework.security.core.userdetails.UserDetails;

   import java.util.Collection;
   import java.util.List;

   @Data
   @NoArgsConstructor
   @AllArgsConstructor
   @Entity
   @Table(name = "admins")
   public class Admin implements UserDetails {
       @Id
       @GeneratedValue(strategy = GenerationType.IDENTITY)
       private Long id;

       private String username;

       private String password;

       @Override
       public Collection<? extends GrantedAuthority> getAuthorities() {
           return List.of(new SimpleGrantedAuthority("ADMIN"));
       }

       @Override
       public String getUsername() {
           return username;
       }

       @Override
       public boolean isAccountNonExpired() {
           return true;
       }

       @Override
       public boolean isAccountNonLocked() {
           return true;
       }

       @Override
       public boolean isCredentialsNonExpired() {
           return true;
       }

       @Override
       public boolean isEnabled() {
           return true;
```

```
        }
    }
```

2) Реализуем репозиторий для получения пользователей из БД

```
package com.example.gofarbot.data;

import com.example.gofarbot.models.Admin;
import org.springframework.data.repository.CrudRepository;

import java.util.Optional;

public interface AdminRepository extends CrudRepository<Admin, Long> {
    Optional<Admin> findAdminByUsername(String username);
}
```

3) Реализуем стандартный сервис для загрузки пользователя по логину из БД

```
package com.example.gofarbot.security.services;


import com.example.gofarbot.data.AdminRepository;
import lombok.AllArgsConstructor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
@AllArgsConstructor
public class AdminDetailsService implements UserDetailsService {
    private AdminRepository ourUserRepo;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return ourUserRepo.findAdminByUsername(username).orElseThrow();
    }
}
```

4) Будем осуществлять аутентификацию по JWT токену.

Подробности о нем можно прочитать в интернете. Вкратце, пользователь авторизуется через логин и пароль, после этого ему выдается зашифрованный токен, который он отправляет в каждом следующем запросе, а сервер проверяет его правильность. Если пользователь отправил неправильный токен, то запрос не выполняется.

В данном сервисе реализованы методы для создания токена, а также его проверки на подлинность.

```
package com.example.gofarbot.security.services;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;

import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
```

```
import java.nio.charset.StandardCharsets;
import java.util.Base64;
import java.util.Date;
import java.util.HashMap;
import java.util.function.Function;

@Component
public class JWTUtils {

    private SecretKey Key;
    private  static  final long EXPIRATION_TIME = 86400000; //24hours or
86400000 milisecs

    private static final long EXPIRATION_TIME_REFRESH = 172800000;
    public JWTUtils(){
        String secreteString =
"843567R634976R7453874567386578367854873568 7R3";
        byte[] keyBytes =
Base64.getDecoder().decode(secreteString.getBytes(StandardCharsets.UTF_8
));
        this.Key = new SecretKeySpec(keyBytes, "HmacSHA256");
    }

    public String generateToken(UserDetails userDetails){
        return Jwts.builder()
                .subject(userDetails.getUsername())
                .issuedAt(new Date(System.currentTimeMillis()))
                .expiration(new Date(System.currentTimeMillis() +
EXPIRATION_TIME))
                .signWith(Key)
                .compact();
    }
    public String generateRefreshToken(HashMap<String, Object> claims,
UserDetails userDetails){
        return Jwts.builder()
                .claims(claims)
                .subject(userDetails.getUsername())
                .issuedAt(new Date(System.currentTimeMillis()))
                .expiration(new Date(System.currentTimeMillis() +
EXPIRATION_TIME_REFRESH))
                .signWith(Key)
                .compact();
    }

    public String extractUsername(String token){
        return extractClaims(token, Claims::getSubject);
    }
    private <T> T extractClaims(String token, Function<Claims, T>
claimsTFunction){
        return
claimsTFunction.apply(Jwts.parser().verifyWith(Key).build().parseSignedC
laims(token).getPayload());
    }

    public boolean isTokenValid(String token, UserDetails userDetails){
        final String username = extractUsername(token);
        return (username.equals(userDetails.getUsername()) &&
!isTokenExpired(token));
    }
    public boolean isTokenExpired(String token){
        return extractClaims(token, Claims::getExpiration).before(new
```

```
Date());
    }

}
```

5) Далее пишем «Фильтр». Класс с методом, который будет обрабатывать проверку токена на подлинность при каждом запросе

```
package com.example.gofarbot.security.config;


import com.example.gofarbot.security.services.AdminDetailsService;
import com.example.gofarbot.security.services.JWTUtils;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.AllArgsConstructor;
import
org.springframework.security.authentication.UsernamePasswordAuthenticati
onToken;
import org.springframework.security.core.context.SecurityContext;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.web.authentication.WebAuthenticationDetails
Source;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

@Component
@AllArgsConstructor
public class JWTAuthFilter extends OncePerRequestFilter {
    private final JWTUtils jwtUtils;
    private final AdminDetailsService ourUserDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain) throws
ServletException, IOException {
        final String authHeader = request.getHeader("Authorization");
        final  String jwtToken;
        final String username;
        if (authHeader == null || authHeader.isBlank()) {
            filterChain.doFilter(request, response);
            return;
        }
        jwtToken = authHeader.substring(7);
        username = jwtUtils.extractUsername(jwtToken);
        if (username != null &&
SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails =
ourUserDetailsService.loadUserByUsername(username);

            if (jwtUtils.isTokenValid(jwtToken, userDetails)) {
                SecurityContext securityContext =
SecurityContextHolder.createEmptyContext();
                UsernamePasswordAuthenticationToken token = new
UsernamePasswordAuthenticationToken(
```

```
                        userDetails, null, userDetails.getAuthorities()
                );
                token.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
                securityContext.setAuthentication(token);
                SecurityContextHolder.setContext(securityContext);
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

6) Далее наконец пишем конфигурацию безопасности. В ней мы: подключаем шифрование, дополнительно автоматический метод аутентификации по логину и паролю для проверки первого входа пользователя, который будет расписан чуть ниже. Также настраиваем защищенные и не защищенные маршруты и подключаем наш фильтр.

```
package com.example.gofarbot.security.config;

import com.example.gofarbot.security.services.AdminDetailsService;
import lombok.AllArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.AuthenticationProvider;
import
org.springframework.security.authentication.dao.DaoAuthenticationProvide
r;
import org.springframework.security.config.Customizer;
import
org.springframework.security.config.annotation.authentication.configurat
ion.AuthenticationConfiguration;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity
;
import
org.springframework.security.config.annotation.web.configuration.EnableW
ebSecurity;
import
org.springframework.security.config.annotation.web.configurers.AbstractH
ttpConfigurer;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenti
cationFilter;

@Configuration
@EnableWebSecurity
@AllArgsConstructor
public class SecurityConfig {

    private final AdminDetailsService ourUserDetailsService;
    private final JWTAuthFilter jwtAuthFIlter;
```

```
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity
httpSecurity) throws Exception {
        httpSecurity.csrf(AbstractHttpConfigurer::disable)
                .cors(Customizer.withDefaults())
                .authorizeHttpRequests(request -> request
                        .requestMatchers("/auth/**").permitAll()

.requestMatchers("/api/**").hasAnyAuthority("ADMIN")
                        .anyRequest().authenticated())
                .sessionManagement(manager ->
manager.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

.authenticationProvider(authenticationProvider()).addFilterBefore(
                        jwtAuthFIlter,
UsernamePasswordAuthenticationFilter.class
                );
        return httpSecurity.build();
    }

    @Bean
    public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider daoAuthenticationProvider = new
DaoAuthenticationProvider();

daoAuthenticationProvider.setUserDetailsService(ourUserDetailsService);
        daoAuthenticationProvider.setPasswordEncoder(passwordEncoder());
        return daoAuthenticationProvider;
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public AuthenticationManager
authenticationManager(AuthenticationConfiguration
authenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }
}
```

7) Далее осталось написать метод первичного входа пользователя по логину и паролю. Для этого используем REST API контроллер.

```
 package com.example.gofarbot.security;

import com.example.gofarbot.security.models.RefreshTokenRequest;
import com.example.gofarbot.security.models.SignInRequest;
import com.example.gofarbot.security.services.AuthService;
import lombok.AllArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
import java.util.Map;

@RestController
@RequestMapping("/auth")
@AllArgsConstructor
@Slf4j
public class AuthController {

    private AuthService authService;

    @PostMapping("/signin")
    public ResponseEntity<Map<String, Object>> signIn(@RequestBody
SignInRequest signInRequest){
        Map<String, Object> response = authService.signIn(signInRequest);
        HttpStatus httpStatus;
        if((int) response.get("status") == 200) {
            httpStatus = HttpStatus.OK;
        } else {
            httpStatus = HttpStatus.BAD_REQUEST;
        }
        return new ResponseEntity<>(response, httpStatus);
    }
    @PostMapping("/refresh")
    public ResponseEntity<Map<String, Object>> refreshToken(@RequestBody
RefreshTokenRequest refreshTokenRequest){
        Map<String, Object> response =
authService.refreshToken(refreshTokenRequest);
        HttpStatus httpStatus;
        if((int) response.get("status") == 200) {
            httpStatus = HttpStatus.OK;
        } else {
            httpStatus = HttpStatus.BAD_REQUEST;
        }
        return new ResponseEntity<>(response, httpStatus);
    }
}
```

8) К нему подключаем сервис с бизнес-логикой.

```
package com.example.gofarbot.security.services;


import com.example.gofarbot.data.AdminRepository;
import com.example.gofarbot.models.Admin;
import com.example.gofarbot.security.models.RefreshTokenRequest;
import com.example.gofarbot.security.models.SignInRequest;
import lombok.AllArgsConstructor;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationT
oken;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

import java.util.*;

@Service
@AllArgsConstructor
public class AuthService {
```

```
    private final AdminRepository ourUserRepo;
    private final JWTUtils jwtUtils;
    private final PasswordEncoder passwordEncoder;
    private final AuthenticationManager authenticationManager;

    public Map<String, Object> signIn(SignInRequest request){
        try {
            authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(request.getUsername(),
request.getPassword()));
            var user =
ourUserRepo.findAdminByUsername(request.getUsername()).orElseThrow();

            var jwt = jwtUtils.generateToken(user);
            var refreshToken = jwtUtils.generateRefreshToken(new
HashMap<>(), user);
            return Map.ofEntries(
                Map.entry("status", 200),
                Map.entry("token", jwt),
                Map.entry("refreshToken", refreshToken),
                Map.entry("expirationTime", "24 Hours"),
                Map.entry("message", "Successfully Signed In")
            );
        }catch (Exception e){
            return Map.ofEntries(
                    Map.entry("status", 400),
                    Map.entry("message", e.getMessage())
            );
        }
    }

    public Map<String, Object> refreshToken(RefreshTokenRequest
refreshTokenRequest) {
        try {
            String ourUsername =
jwtUtils.extractUsername(refreshTokenRequest.getRefreshToken());
            if(Objects.equals(ourUsername,
refreshTokenRequest.getUsername())) {
                authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(ourUsername,
refreshTokenRequest.getPassword()));
                Admin user =
ourUserRepo.findAdminByUsername(ourUsername).orElseThrow();
                if
(jwtUtils.isTokenValid(refreshTokenRequest.getRefreshToken(), user)) {
                    var jwt = jwtUtils.generateToken(user);
                    return Map.ofEntries(
                            Map.entry("status", 200),
                            Map.entry("token", jwt),
                            Map.entry("refreshToken",
refreshTokenRequest.getRefreshToken()),
                            Map.entry("expirationTime", "24 Hours"),
                            Map.entry("message", "Successfully Refreshed
Token")
                    );
                }
                else {
                    throw new Exception("Invalid token");
                }
            }
            else {
```

```
                throw new Exception("Invalid username");
            }
        }
        catch(Exception e) {
            return Map.ofEntries(
                    Map.entry("status", 400),
                    Map.entry("message", e.getMessage())
            );
        }
    }
}
```

В данном коде предполагается, что пользователь уже существует в БД, обратите внимание с шифрованным паролем, если вы хотите писать логику регистрации пользователя и его создания в БД, то немного подумайте, как дополнить этот код :)
```

---

*Пример использования панелью управления*

Данные, отправляемые через WebSocket/Kafka, могут быть визуализированы через реализацию графского интерфейса пользователя, это могут быть панели управления, дэшборды, например для отображения графиков данных из базы, уведомлений и обезличенной аналитики (напримерт для АС для производства кирпичей это может быть среднее время обжига кирпича, и так далее). Для реализации интерфейса используйте React.js или Angular.

*Пример интеграции через API (REST) для исторических данных и WebSocket для реального времени:*
```java

Представлен пример контроллера с необходимыми аннотациями для удобной работы, а также с примерами классов DTO (Data transfer objects). Про их назначение прочитайте в интернете.

```
package com.example.gofarbot.controllers.web_controllers.dto.conferences;

import com.example.gofarbot.models.Notification;
import jakarta.validation.constraints.NotNull;
import lombok.Builder;
import lombok.Getter;

import java.time.LocalDateTime;
import java.util.List;

@Getter
@Builder
public class CreateConferenceRequest {
    @NotNull
    private final String name;
    @NotNull
    private final String link;
    @NotNull
```

```
    private final LocalDateTime time;
}
```

```
package com.example.gofarbot.controllers.web_controllers.dto.conferences;

import com.example.gofarbot.controllers.web_controllers.dto.BaseResponse;
import com.example.gofarbot.models.Conference;
import lombok.Getter;
import lombok.experimental.SuperBuilder;

@Getter
@SuperBuilder
public class CreateConferenceResponse extends BaseResponse {
    private final Conference conference;
}
```

```
package com.example.gofarbot.controllers.web_controllers.dto;


import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.experimental.SuperBuilder;

@AllArgsConstructor
@Getter
@SuperBuilder
public class BaseResponse {
    private final short code;
    private final String message;
}
```

```
package com.example.gofarbot.controllers.web_controllers;


import com.example.gofarbot.controllers.web_controllers.dto.conferences.*;
import com.example.gofarbot.services.web_services.ConferenceService;
import jakarta.validation.Valid;
import lombok.AllArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.HttpStatusCode;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.Objects;

@RestController
@RequestMapping(path = "/api/conferences", produces = "application/json")
@AllArgsConstructor
@Slf4j
public class ConferencesController {
    private final ConferenceService conferenceService;

    @GetMapping
    public ResponseEntity<GetConferencesResponse> getConferences() {
        GetConferencesResponse response =
conferenceService.getConferences();
        HttpStatus httpStatus;
        if(Objects.equals(response.getCode(), (short) 200)) {
```

```
            httpStatus = HttpStatus.OK;
        } else {
            httpStatus = HttpStatus.BAD_REQUEST;
        }
        return new ResponseEntity<>(response, httpStatus);
    }

    @GetMapping("/{id}")
    public ResponseEntity<GetConferenceResponse>
getConference(@PathVariable("id") long conferenceId) {
        GetConferenceResponse response =
conferenceService.getConference(conferenceId);
        HttpStatus httpStatus;
        if(Objects.equals(response.getCode(), (short) 200)) {
            httpStatus = HttpStatus.OK;
        } else {
            httpStatus = HttpStatus.BAD_REQUEST;
        }
        return new ResponseEntity<>(response, httpStatus);
    }

    @PostMapping
    public ResponseEntity<CreateConferenceResponse> createConference(
            @RequestBody @Valid CreateConferenceRequest request
    ) {
        CreateConferenceResponse response =
conferenceService.createConference(request);
        HttpStatus httpStatus;
        if(Objects.equals(response.getCode(), (short) 200)) {
            httpStatus = HttpStatus.OK;
        } else {
            httpStatus = HttpStatus.BAD_REQUEST;
        }
        return new ResponseEntity<>(response, httpStatus);
    }

    @PutMapping("/{id}")
    public ResponseEntity<UpdateConferenceResponse> updateConference(
            @PathVariable("id") long conferenceId,
            @RequestBody UpdateConferenceRequest request
    ) {
        UpdateConferenceResponse response =
conferenceService.updateConference(request, conferenceId);
        return new ResponseEntity<>(response,
HttpStatusCode.valueOf((response.getCode())));
    }
}
```

Заключение.

Итак, что же у нас получилось? Опишем итоговую функциональность вашей системы

- Организация сбора данных с датчиков с помощью MQTT/Kafka в реальном времени.
- Обработка данных в реальном времени (например, проверка максимальных значений, расчет среднего времени, уведомления).

- визуализация данных, полученных через WebSocket на панели пользовательского интерфейса.
- Анализ производительности для оптимизации выполняемых процессов.
- Автоматизированные действия системы на основе событий (например, сигнал об остановке оборудования при сбое).

Разработка такой системы требует тесного сотрудничества с инженерными специалистами, работающими на производственной линии.
Но это еще не все нам нужно провести тестирование.

6. Процесс тестирования

После тога как мы разработали АС , нам необходимо провести тестирование нашей системы.

Тестирование автоматизированной системы реального времени— это важный этап разработки, так как любая ошибка в системе может привести к значительным последствиям, особенно в производственных процессах (например производство тех же кирпичей). Система в реальном времени обычно состоит из согласованного взаимодействия множества компонентов, поэтому подход к тестированию должен быть системным и многоэтапным.

Опишем основные шаги, подходы и инструменты, которые можно использовать для тестирования разработанной системы (см часть 2).