一. Kafka 的应用

1. 启动 zookeeper

   bin\windows\zookeeper-server-start.bat

   config\zookeeper.properties

2. 启动 kafka

   bin\windows\kafka-server-start.bat

   config\server.properties

3. 配置 kafka 生产者类，并通过 spring 注入

```
    Properties properties;

    public KafkaProducerDemo(Properties properties) {
        super();
        this.properties = properties;
    }

//    public KafkaProducerDemo() {
//
//    }

    public Properties getProperties() { return properties; }

    public void setProperties(Properties properties) { this.properties = properties; }

    public void sendMessage(String msg) {

        KafkaProducer<String, String> producer = new KafkaProducer<~>(properties);

        ProducerRecord<String, String> record = new ProducerRecord<~>(properties.getProperty("top
                msg);
        producer.send(record);

        producer.close();

    }
```

```
<bean id="kafkaProducerDemo" class="com.club.producer.KafkaProducerDemo">
    <constructor-arg>
        <props>
            <prop key=""></prop>
        </props>
    </constructor-arg>
    <property name="properties">
        <props>
            <prop key="topic">my-replicated-topic</prop>
            <prop key="bootstrap.servers">127.0.0.1:9092</prop>
            <prop key="acks">all</prop>
            <                                    org.apache.kafka.common.serialization.StringSerializer
            </prop>
            <prop key="value.serializer">org.apache.kafka.common.serialization.StringSerializer
            </prop>
            <prop key="buffer.memory">33554432</prop>
        </props>
    </property>
</bean>
```

Typo: In word 'acks' more... (Ctrl+F1)

4. 配置 kafka 消费者类，同样通过 spring 注入

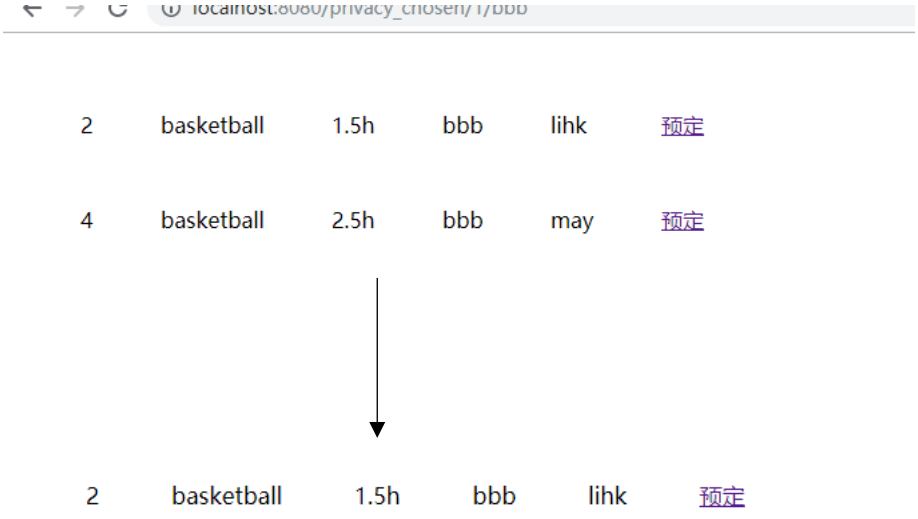
```
    private Properties props;

    public KafkaConsumerDemo(Properties props) {
        super();
        this.props = props;
    }

    public KafkaConsumerDemo() {
    }
    public Properties getProps() { return props; }

    public void setProps(Properties props) { this.props = props; }

    public ArrayList<String> receive() {

        KafkaConsumer<String, String> consumer = new KafkaConsumer<~>(props);
        consumer.subscribe(Arrays.asList(props.getProperty("topic")));

        String msg = "";
        ArrayList<String> msgs = new ArrayList<~>();
        while (true) {
            ConsumerRecords<String, String> consumerRecords = consumer.poll( timeout: 100);
            for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {
                msg += consumerRecord.value();
                msgs.add(consumerRecord.value());
            }
            consumer.close();
            return msgs;
        }
    }
```

5. 实际应用

5.1 获取所有私人课程

1	tennis	2h	bbb	lean	预定
2	basketball	1.5h	bbb	lihk	预定
4	basketball	2.5h	bbb	may	预定

5.2 每当用户选择预定一门私人课程，生产者发送消息，页面随之改变，同时利用线程进行消费，进行数据库的更新



每次消费将生产者生产的所有信息全部消费，由于操作的是同一个 topic，不仅保证了数据处理的速度（即使被堵塞也可在之后处理掉之前累积的操作），也尽可能避免了重复消费的情况，保证了操作的安全性。

二. Webflux 实现 restful 风格的 API

即相当于对之前类的重写

（1）获取全部

```
public Flux<UserEntity> getAll() {
    return Flux.fromIterable(userRepository.findAll());
}
```

（2）根据 id 进行检索

```
public Mono<UserEntity> getByid(int id) {
    return Mono.just(userRepository.findOne(id));
}
```

（3）更新

```
public Mono<UserEntity> putUser(Mono<UserEntity> userEntityMono) {
    Mono<UserEntity> responseMono = userEntityMono.doOnNext(user -> {
        userRepository.saveAndFlush(user);
    });
    return responseMono;
}
```

（4）添加

```
public Mono<Void> saveUser(Mono<UserEntity> userEntityMono) {
    Mono<UserEntity> reponseMono = userEntityMono.doOnNext(user -> {
        userRepository.save(user);
    });
    return reponseMono.then();
}
```

（5）删除

```
public Mono<String> deleteUser(int id) {
    userRepository.delete(id);
    return Mono.just("Deleted");
}
```

以上是对用户的操作，对于教练和课程的操作与此类似

三. 函数式编程

使用{HandlerFunctions,RouteFunctions}进行开发，同样展示用户的相关配置，教练与课程与此类似

（1）Handelr

```
private UserRepositoryImpl userRepository;

public UserHandler(UserRepositoryImpl userRepository) { this.userRepository=userRepository; }

public Mono<ServerResponse> getAll(ServerRequest request) {
    Flux<UserEntity> userEntityFlux=userRepository.getAll();
    return ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(userEntityFlux,UserEntity.class);
}

public Mono<ServerResponse> getByid(ServerRequest request) {
    int userid=Integer.valueOf(request.pathVariable( name: "id"));
    Mono<ServerResponse> notFound = ServerResponse.notFound().build();
    Mono<UserEntity> userEntityMono = userRepository.getByid(userid);
    return userEntityMono.flatMap(CourseEntity->ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(userEntityMono,UserEntity.class))
}

public Mono<ServerResponse> postUser(ServerRequest request) {
    Mono<UserEntity> userEntityMono = request.bodyToMono(UserEntity.class);
    return ServerResponse.ok().build(userRepository.saveUser(userEntityMono));
}

public Mono<ServerResponse> putUser(ServerRequest request) {
    int userId = Integer.valueOf(request.pathVariable( name: "id"));
    Mono<UserEntity> userEntityMono = request.bodyToMono(UserEntity.class);
    Mono<UserEntity> responseMono =  userRepository.putUser(userEntityMono);
    return responseMono
            .flatMap(user -> ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(fromObject(user)));
}

public Mono<ServerResponse> deleteUser(ServerRequest request) {
    int userId = Integer.valueOf(request.pathVariable( name: "id"));
    Mono<String> responseMono = userRepository.deleteUser(userId);
    return responseMono
            .flatMap(strMono -> ServerResponse.ok().contentType(MediaType.TEXT_PLAIN).body(fromObject(strMono)));
}
}
```

（2）RouteFunction

```
public RouterFunction<ServerResponse> monoRouterFunctionUser(UserHandler userHandler) {
    return route(GET( pattern: "/api/customer").and(accept(MediaType.APPLICATION_JSON)), userHandler::getAll)
            .andRoute(GET( pattern: "/api/customer/{id}").and(accept(MediaType.APPLICATION_JSON)), userHandler::getByid)
            .andRoute(POST( pattern: "/api/customer/post").and(accept(MediaType.APPLICATION_JSON)), userHandler::postUser)
            .andRoute(PUT( pattern: "/api/customer/put/{id}").and(accept(MediaType.APPLICATION_JSON)), userHandler::putUser)
            .andRoute(DELETE( pattern: "/api/customer/delete/{id}").and(accept(MediaType.APPLICATION_JSON)), userHandler::deleteUser);
}
```

四. Webflux 与 springsecurity 的结合

在 springsecurity 的基础上进行修改

（1）配置信息

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig  {


    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http
                .authorizeExchange() ServerHttpSecurity.AuthorizeExchangeSpec
                .pathMatchers( ...antPatterns: "/","/home","logout","index").permitAll()  //无需进行权限过滤的请求路径
                .anyExchange().authenticated() ServerHttpSecurity.AuthorizeExchangeSpec
                .and() ServerHttpSecurity
                .httpBasic().and() ServerHttpSecurity
                .formLogin() ServerHttpSecurity.FormLoginSpec
                .loginPage("/indexs") ServerHttpSecurity.FormLoginSpec
                ;
        return http.build();
    }

}
```

（2）配置 FilterSecurityInterceptor

```
public void init(FilterConfig filterConfig) throws ServletException {

}

@Override
public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletExc
    FilterInvocation fi = new FilterInvocation(servletRequest, servletResponse, filterChain);
    invoke(fi);
}

public void invoke(FilterInvocation fi) throws IOException, ServletException {
    InterceptorStatusToken token = super.beforeInvocation(fi);
    try {
        fi.getChain().doFilter(fi.getRequest(), fi.getResponse());
    } finally {
        super.afterInvocation(token, returnedObject: null);
    }
}

@Override
public void destroy() {

}

@Override
public Class<?> getSecureObjectClass() {
    return FilterInvocation.class;
}

@Override
public SecurityMetadataSource obtainSecurityMetadataSource() {
    return this.securityMetadataSource;
}
}
```

（3）根据 restful api 和函数式方程从数据库读取权限进行

加载和判断