E-Banking-Portal ReadMe

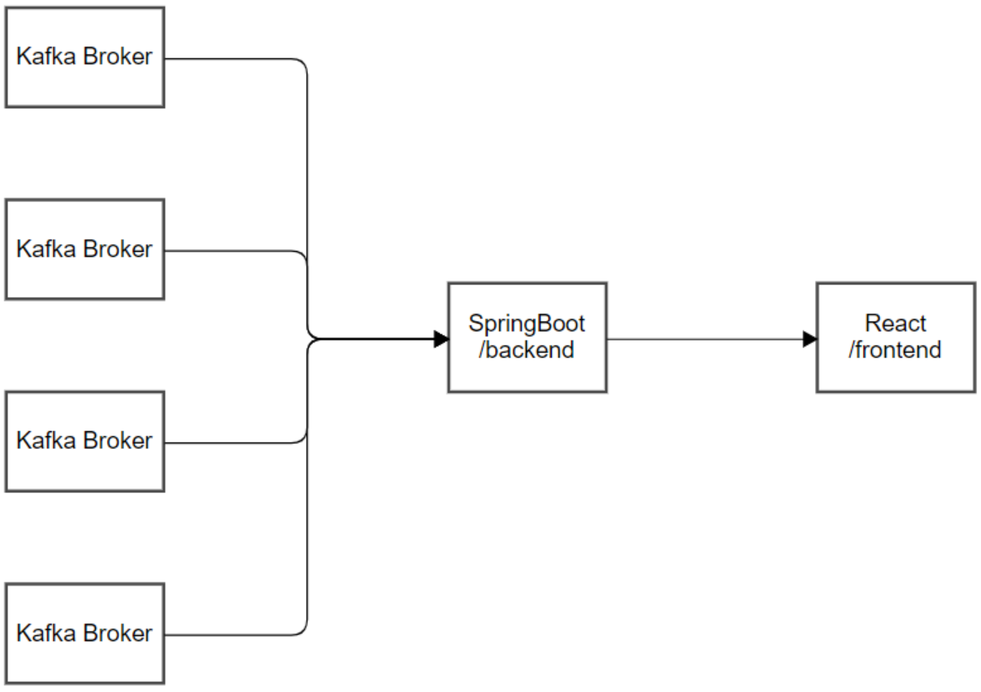
Background and Summary

For an e-Banking Portal you have been given the task to design and implement a reusable REST API for returning the paginated list of money account transactions created in an arbitrary calendar month for a given customer who is logged-on in the portal. For each transaction page returns the total credit and debit values at the current exchange rate (from the third-party provider). The list of transactions should be consumed from a Kafka topic. Build a Docker image out of the application and prepare the configuration for deploying it to Kubernetes

Main technologies

React
SpringBoot
Spring Security
Spring MVC
JUnit
Integration Test
JMeter
Kafka
Kubernetes
Docker
Jenkins CI/CD

High Level Design



User Interface

- The user interface is built using React. It consisted of two main screens: transfer money and transaction history.
- The transfer money screen allows users to enter the recipient's account details and the amount to transfer.
- The transaction history screen displays a list of all past transactions in the selected month, sorted by datetime.

Backend

- The backend is built using Spring Boot. It consists of four main components: REST APIs, authentication system, Kafka producer/consumer and exchange rate fetch.
- The REST API is composed of three main components: the controller, service, and repository.
- The REST API handles incoming requests from the front end and interacts with the Kafka producer/consumer.
- The authentication system (Spring Security) generates a token for users when login, and checks the token in each request.
- The Kafka producer stores new transactions on the server disk.
- The Kafka consumer listens for the transaction record.
- After the server starts running, a thread is launched that fetches the latest exchange rates from the OpenExchangerates API every 2 hours.

Data Storage

- Data storage is handled entirely through Kafka. All transactions are stored in Kafka, which serves as a distributed message queue.

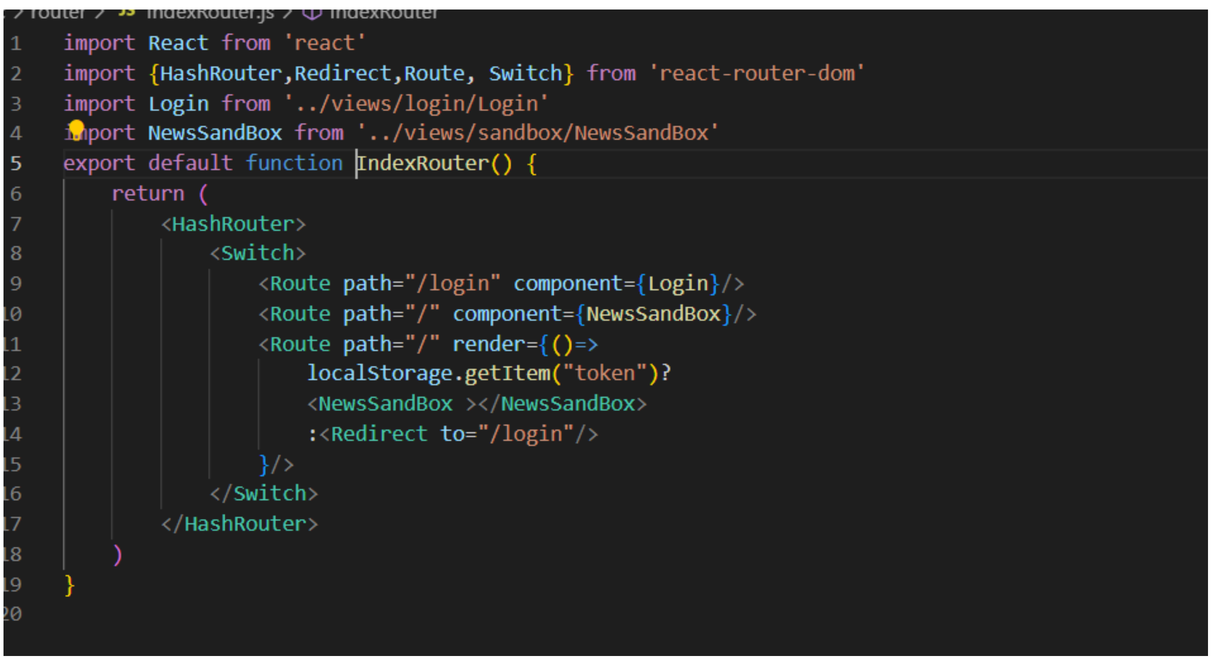
Container Management

- Container management is handled using Kubernetes. It manages the deployment and scaling of the application, as well as any necessary updates or patches.

Low Level Design

User Interface:

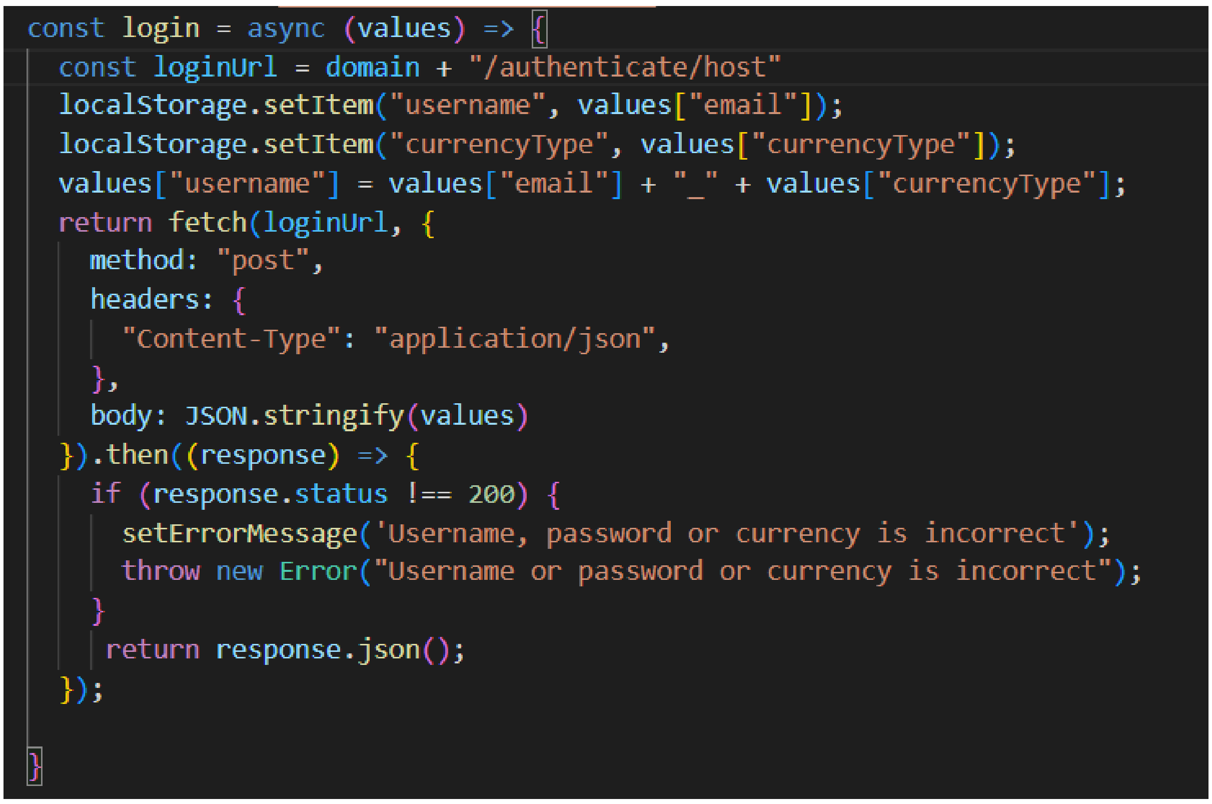
Router



```
import React from 'react'
import {HashRouter,Redirect,Route, Switch} from 'react-router-dom'
import Login from '../views/login/Login'
import NewsSandBox from '../views/sandbox/NewsSandBox'
export default function IndexRouter() {
    return (
        <HashRouter>
            <Switch>
                <Route path="/login" component={Login}/>
                <Route path="/" component={NewsSandBox}/>
                <Route path="/" render={()=>
                    localStorage.getItem("token")?
                    <NewsSandBox ></NewsSandBox>
                    :<Redirect to="/login"/>
                }/>
            </Switch>
        </HashRouter>
    )
}
```

The router checks for the presence of a token. If a token is detected, the user will be directed to the login page. If no token is found, the user will be redirected to the home page.

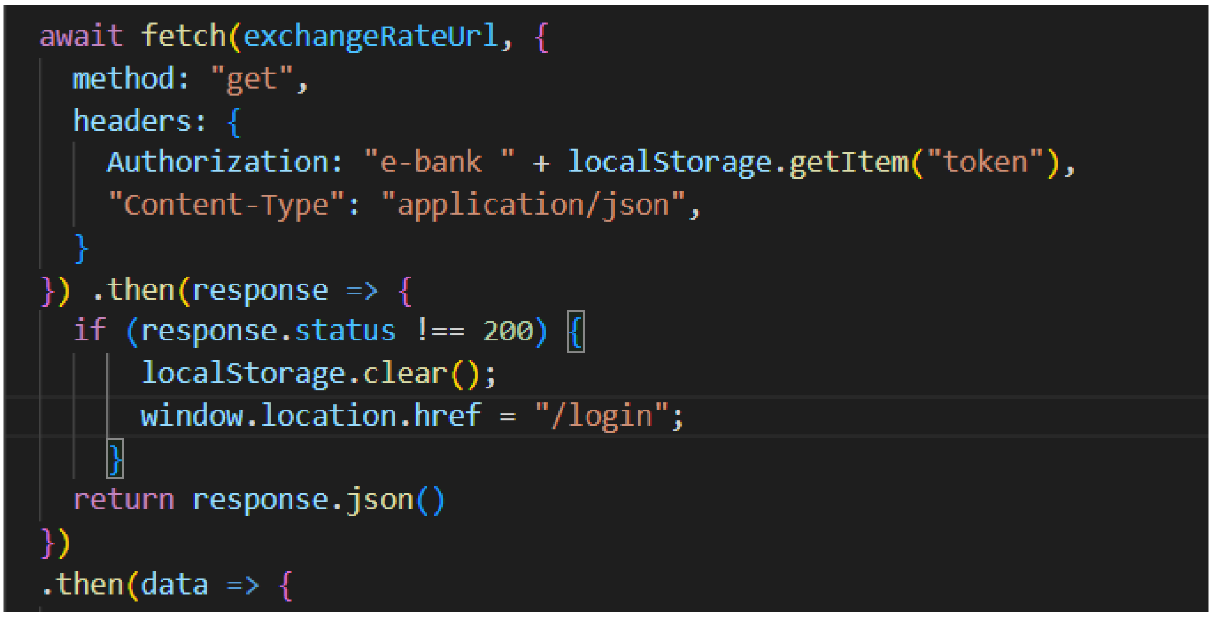
AJAX/Fetch



```
const login = async (values) => {
  const loginUrl = domain + "/authenticate/host"
  localStorage.setItem("username", values["email"]);
  localStorage.setItem("currencyType", values["currencyType"]);
  values["username"] = values["email"] + "_" + values["currencyType"];
  return fetch(loginUrl, {
    method: "post",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(values)
  }).then((response) => {
    if (response.status !== 200) {
      setErrorMessage('Username, password or currency is incorrect');
      throw new Error("Username or password or currency is incorrect");
    }
     return response.json();
  });

}
```

The frontend uses Fetch to manage the communication between the front-end and backend. For example, on the login page, after the user submits the login form, the system will fetch data from the backend to verify the correctness of the username, password, and currency type entered.

Token Check

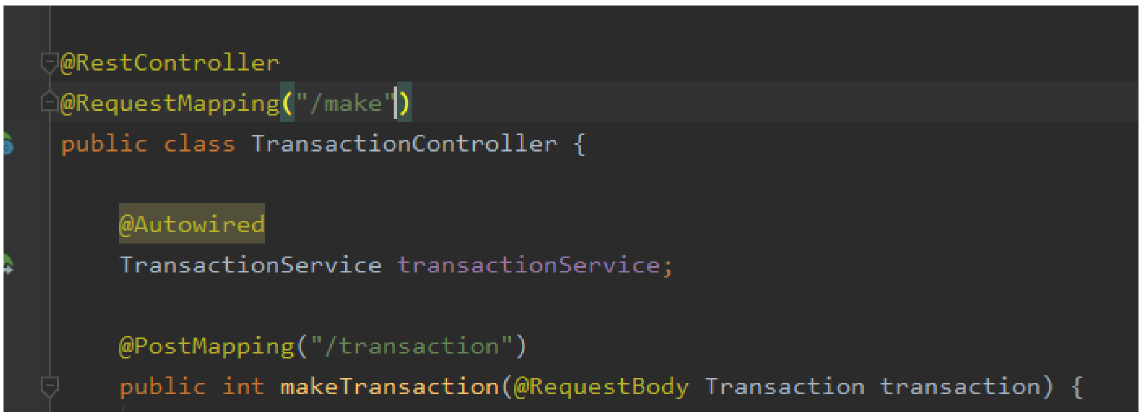


```
await fetch(exchangeRateUrl, {
  method: "get",
  headers: {
    Authorization: "e-bank " + localStorage.getItem("token"),
    "Content-Type": "application/json",
  }
}) .then(response => {
  if (response.status !== 200) {
     localStorage.clear();
     window.location.href = "/login";
    }
  return response.json()
})
.then(data => {
```

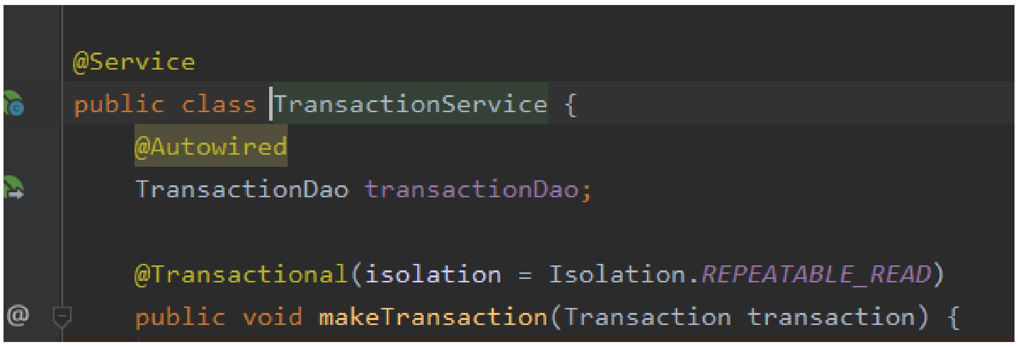
Once login, the token will be included in the header of each subsequent request. If the backend returns a status code of 400, the frontend will clear the cache and redirect the user to the login page.

Backend

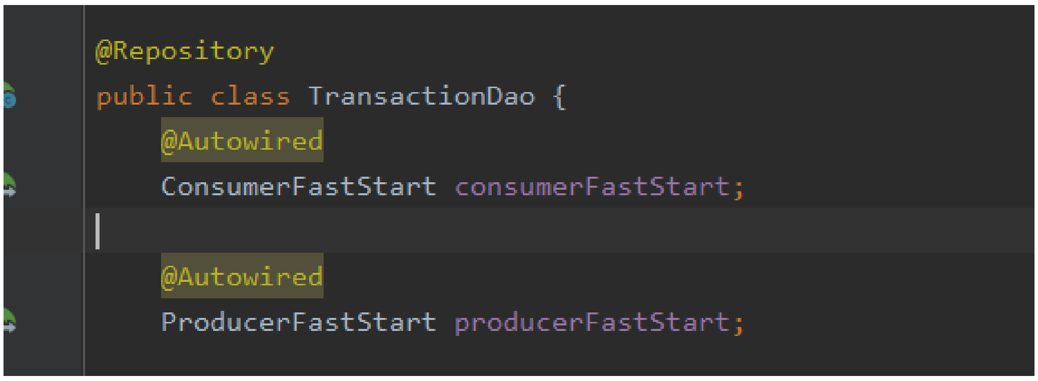
RestAPI Structure



```java
@RestController
@RequestMapping("/make")
public class TransactionController {

    @Autowired
    TransactionService transactionService;

    @PostMapping("/transaction")
    public int makeTransaction(@RequestBody Transaction transaction) {
```



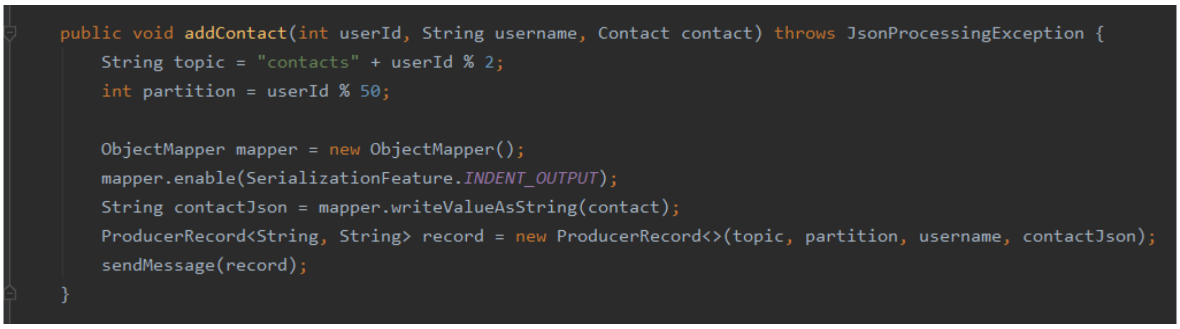
```java
@Service
public class TransactionService {
    @Autowired
    TransactionDao transactionDao;

    @Transactional(isolation = Isolation.REPEATABLE_READ)
    public void makeTransaction(Transaction transaction) {
```



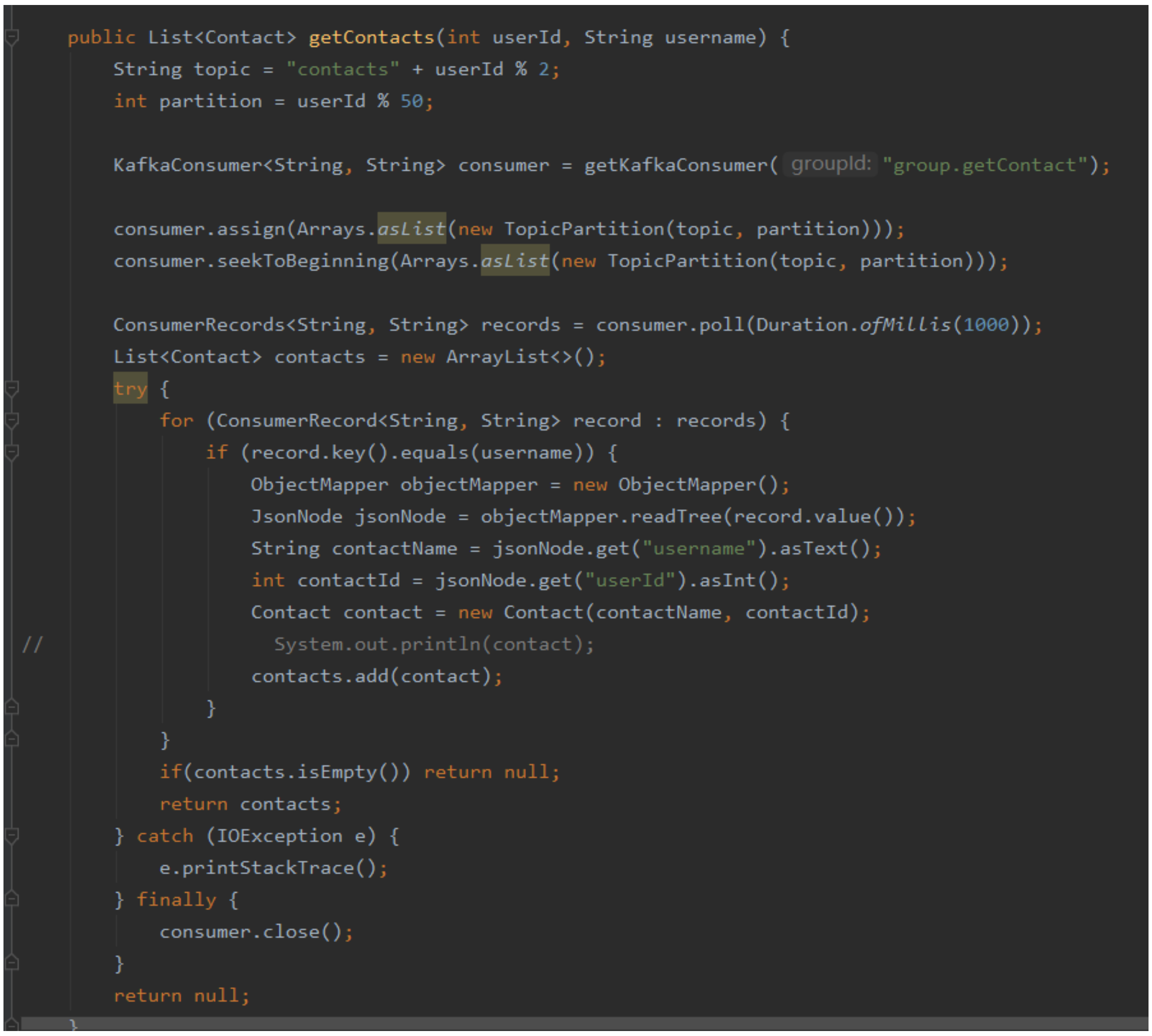
```java
@Repository
public class TransactionDao {
    @Autowired
    ConsumerFastStart consumerFastStart;

    @Autowired
    ProducerFastStart producerFastStart;
```

The Rest API architecture comprises several layers: starting with the controller, the request is passed on to the service, and then to the data access object (DAO). Finally, the request is processed by the Kafka producer or consumer. For instance, when a transaction request is received, the TransactionController handles it, which then passes the request to TransactionService, followed by TransactionDao, before finally dealing with the transaction through the Kafka producer.

Kafka Producer/Consumer



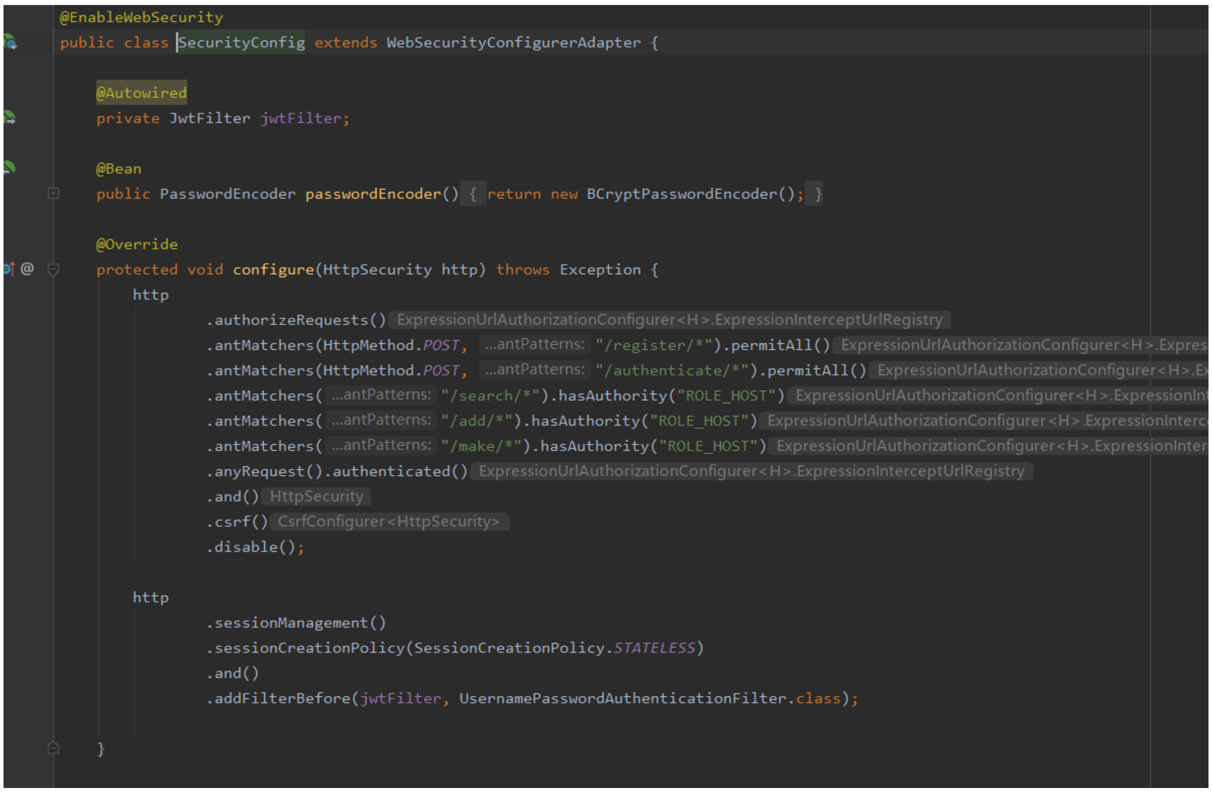
```java
public void addContact(int userId, String username, Contact contact) throws JsonProcessingException {
    String topic = "contacts" + userId % 2;
    int partition = userId % 50;

    ObjectMapper mapper = new ObjectMapper();
    mapper.enable(SerializationFeature.INDENT_OUTPUT);
    String contactJson = mapper.writeValueAsString(contact);
    ProducerRecord<String, String> record = new ProducerRecord<>(topic, partition, username, contactJson);
    sendMessage(record);
}
```



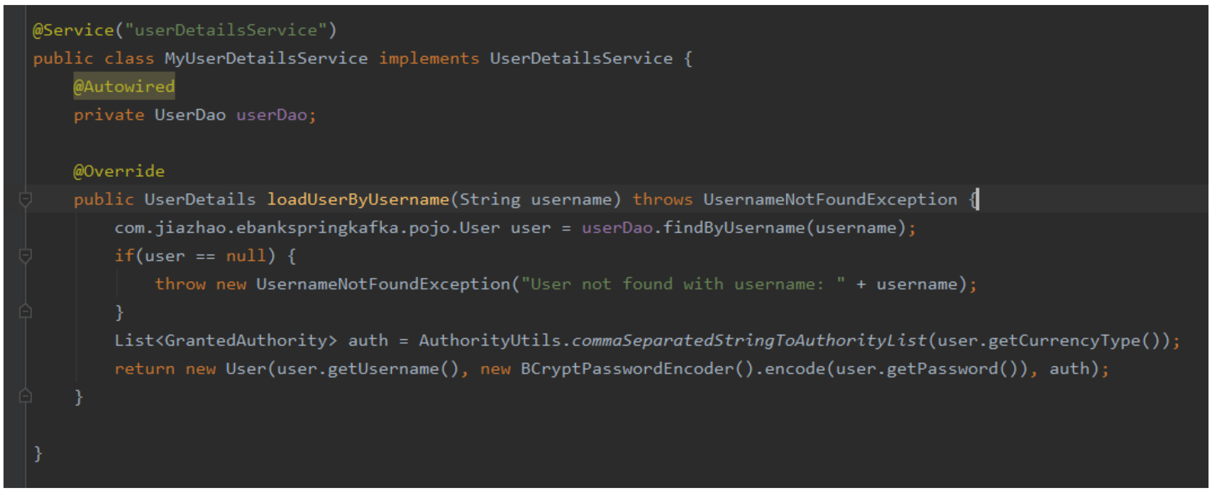
```java
public List<Contact> getContacts(int userId, String username) {
    String topic = "contacts" + userId % 2;
    int partition = userId % 50;

    KafkaConsumer<String, String> consumer = getKafkaConsumer( groupId: "group.getContact");

    consumer.assign(Arrays.asList(new TopicPartition(topic, partition)));
    consumer.seekToBeginning(Arrays.asList(new TopicPartition(topic, partition)));

    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
    List<Contact> contacts = new ArrayList<>();
    try {
        for (ConsumerRecord<String, String> record : records) {
            if (record.key().equals(username)) {
                ObjectMapper objectMapper = new ObjectMapper();
                JsonNode jsonNode = objectMapper.readTree(record.value());
                String contactName = jsonNode.get("username").asText();
                int contactId = jsonNode.get("userId").asInt();
                Contact contact = new Contact(contactName, contactId);
//                System.out.println(contact);
                contacts.add(contact);
            }
        }
        if(contacts.isEmpty()) return null;
        return contacts;
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        consumer.close();
    }
    return null;
```

When a user registers with the system, a random ID is generated. To determine the topic and partition in which the user's data is stored, the user ID is hashed. For instance, there are two topics for storing contact data, each with 50 partitions, so the topic and partition to which the user's contact data should belong can be determined by calculating userID % 2 and userID % 50, respectively.

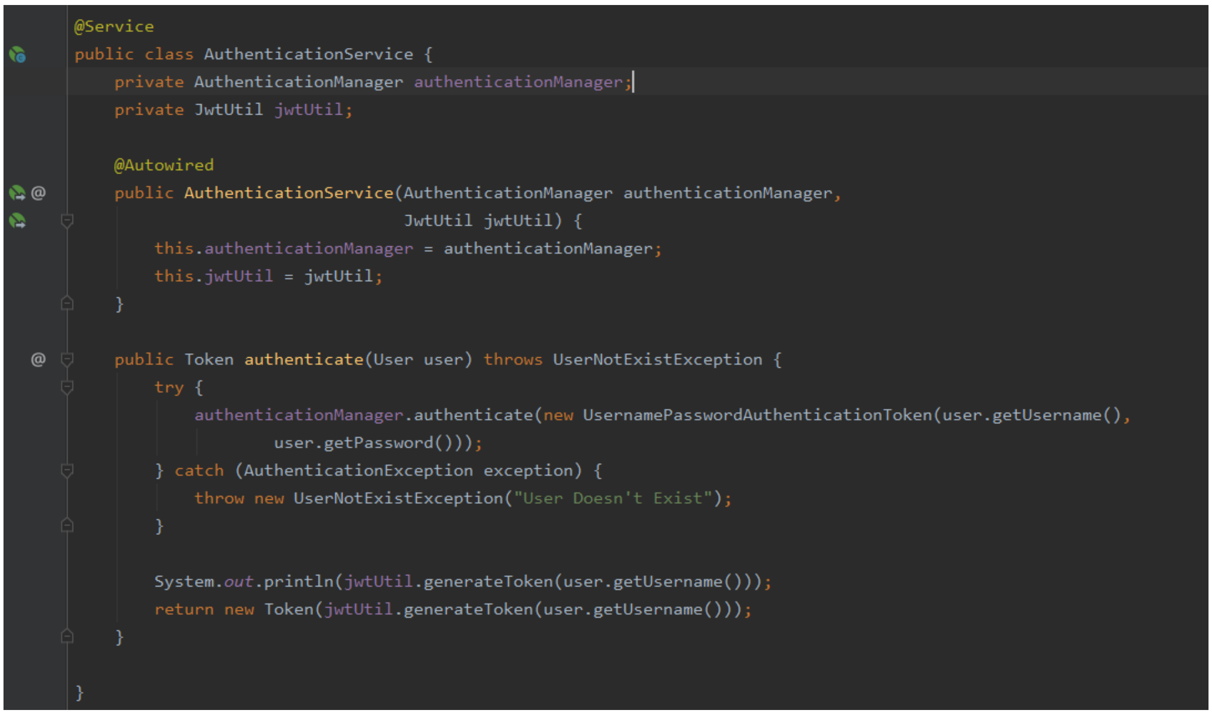
Authentication System



```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private JwtFilter jwtFilter;

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

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
                .antMatchers(HttpMethod.POST,  ...antPatterns: "/register/*").permitAll() ExpressionUrlAuthorizationConfigurer<H>.Expres
                .antMatchers(HttpMethod.POST,  ...antPatterns: "/authenticate/*").permitAll() ExpressionUrlAuthorizationConfigurer<H>.Ex
                .antMatchers( ...antPatterns: "/search/*").hasAuthority("ROLE_HOST") ExpressionUrlAuthorizationConfigurer<H>.ExpressionInt
                .antMatchers( ...antPatterns: "/add/*").hasAuthority("ROLE_HOST") ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterc
                .antMatchers( ...antPatterns: "/make/*").hasAuthority("ROLE_HOST") ExpressionUrlAuthorizationConfigurer<H>.ExpressionInter
                .anyRequest().authenticated() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
                .and() HttpSecurity
                .csrf() CsrfConfigurer<HttpSecurity>
                .disable();

        http
                .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and()
                .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);

    }
```

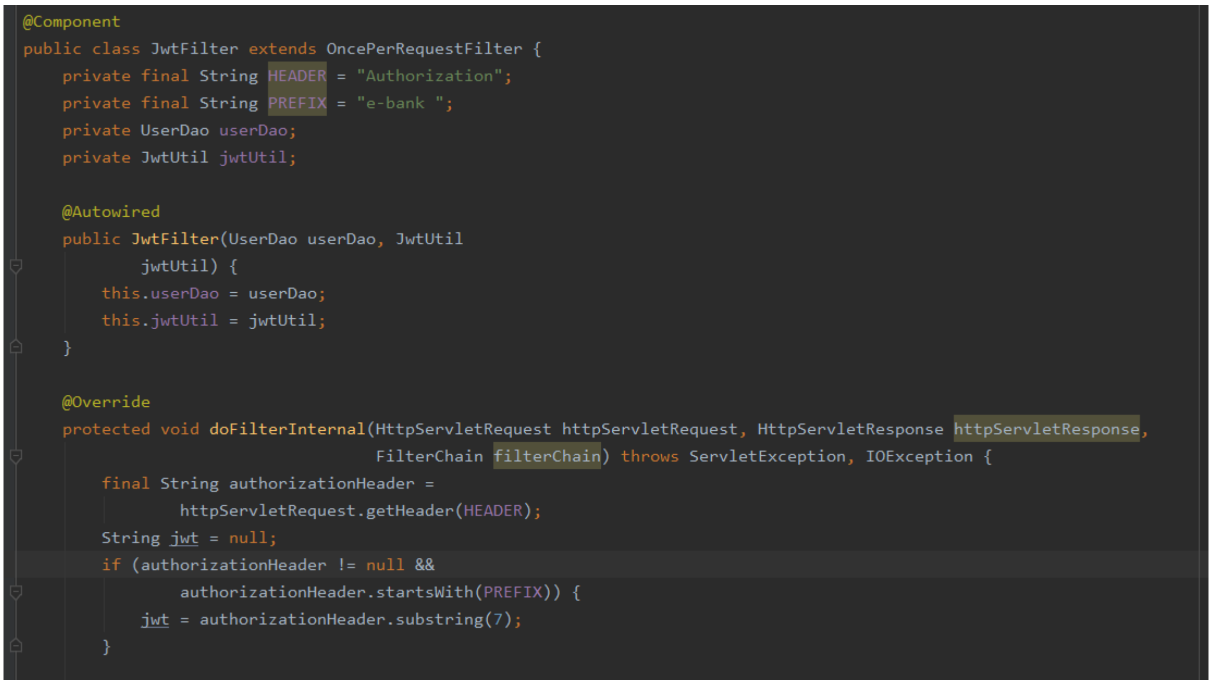


```
@Service("userDetailsService")
public class MyUserDetailsService implements UserDetailsService {
    @Autowired
    private UserDao userDao;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        com.jiazhao.ebankspringkafka.pojo.User user = userDao.findByUsername(username);
        if(user == null) {
            throw new UsernameNotFoundException("User not found with username: " + username);
        }
        List<GrantedAuthority> auth = AuthorityUtils.commaSeparatedStringToAuthorityList(user.getCurrencyType());
        return new User(user.getUsername(), new BCryptPasswordEncoder().encode(user.getPassword()), auth);
    }

}
```

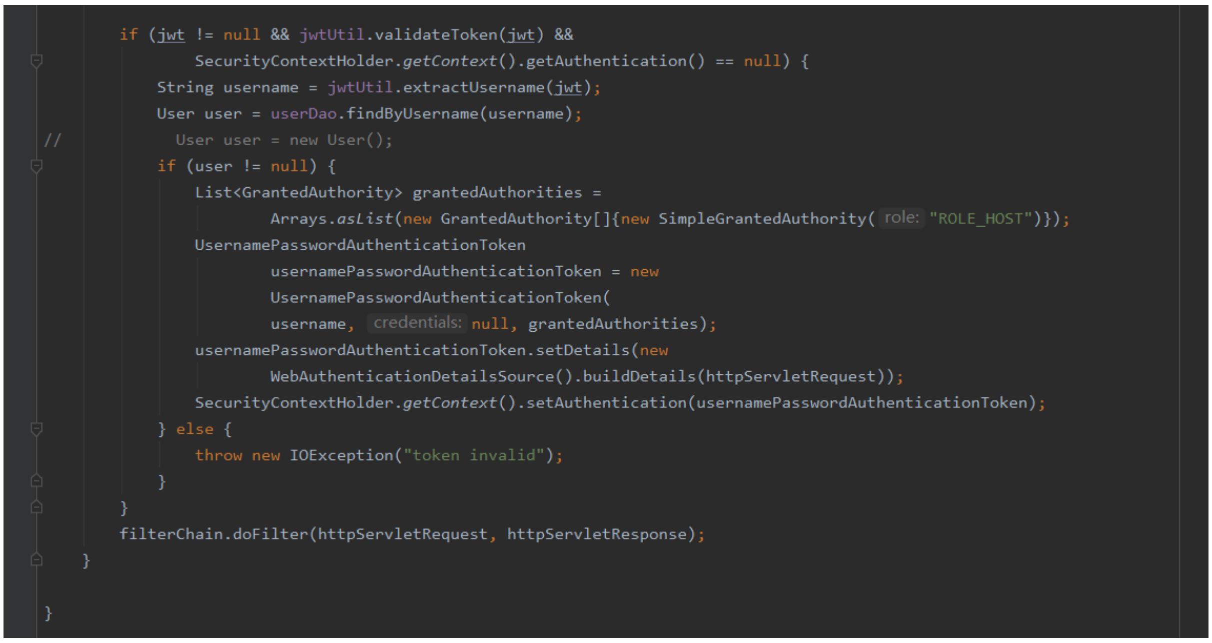


```java
@Service
public class AuthenticationService {
    private AuthenticationManager authenticationManager;
    private JwtUtil jwtUtil;

    @Autowired
    public AuthenticationService(AuthenticationManager authenticationManager,
                                 JwtUtil jwtUtil) {
        this.authenticationManager = authenticationManager;
        this.jwtUtil = jwtUtil;
    }

    public Token authenticate(User user) throws UserNotExistException {
        try {
            authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(user.getUsername(),
                    user.getPassword()));
        } catch (AuthenticationException exception) {
            throw new UserNotExistException("User Doesn't Exist");
        }

        System.out.println(jwtUtil.generateToken(user.getUsername()));
        return new Token(jwtUtil.generateToken(user.getUsername()));
    }

}
```

The project uses Spring Security for authentication. To this end, the project extends WebSecurityConfigurerAdapter, UserDetailServices, and creates an AuthenticationService to verify the user's email, currency type, and password when they first log in. If the authentication is successful, the backend generates an MD5 token and sends it to the client side.



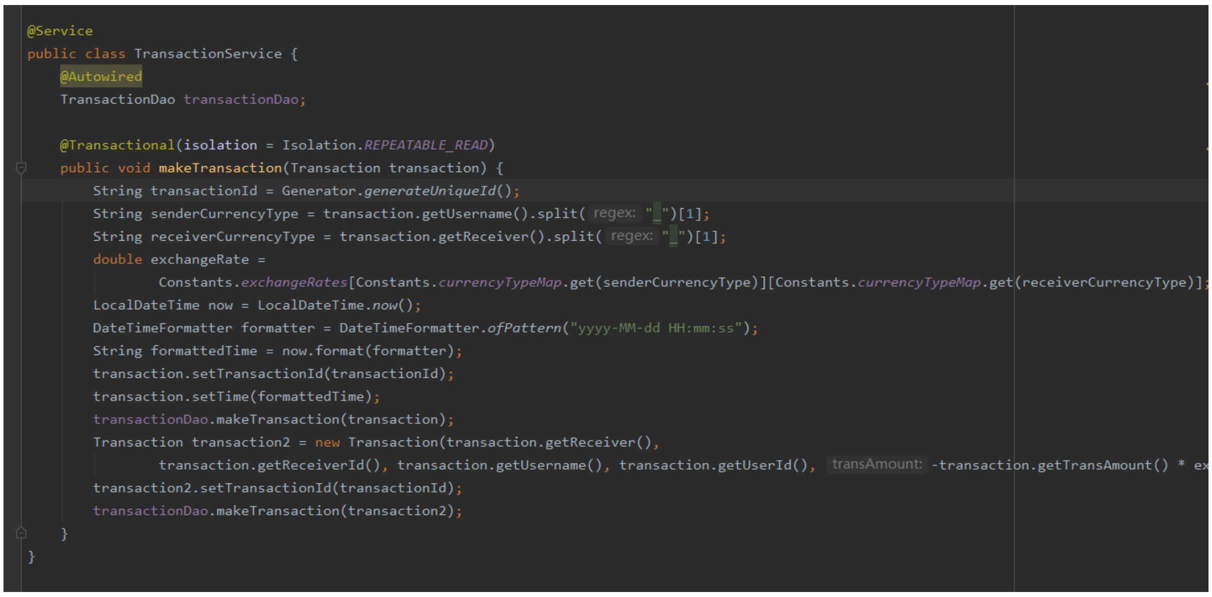
```java
@Component
public class JwtFilter extends OncePerRequestFilter {
    private final String HEADER = "Authorization";
    private final String PREFIX = "e-bank ";
    private UserDao userDao;
    private JwtUtil jwtUtil;

    @Autowired
    public JwtFilter(UserDao userDao, JwtUtil
            jwtUtil) {
        this.userDao = userDao;
        this.jwtUtil = jwtUtil;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse,
                                    FilterChain filterChain) throws ServletException, IOException {
        final String authorizationHeader =
                httpServletRequest.getHeader(HEADER);
        String jwt = null;
        if (authorizationHeader != null &&
                authorizationHeader.startsWith(PREFIX)) {
            jwt = authorizationHeader.substring(7);
        }
```



```
        if (jwt != null && jwtUtil.validateToken(jwt) &&
                SecurityContextHolder.getContext().getAuthentication() == null) {
            String username = jwtUtil.extractUsername(jwt);
            User user = userDao.findByUsername(username);
//            User user = new User();
            if (user != null) {
                List<GrantedAuthority> grantedAuthorities =
                        Arrays.asList(new GrantedAuthority[]{new SimpleGrantedAuthority( role: "ROLE_HOST")});
                UsernamePasswordAuthenticationToken
                        usernamePasswordAuthenticationToken = new
                        UsernamePasswordAuthenticationToken(
                        username,  credentials: null, grantedAuthorities);
                usernamePasswordAuthenticationToken.setDetails(new
                        WebAuthenticationDetailsSource().buildDetails(httpServletRequest));
                SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
            } else {
                throw new IOException("token invalid");
            }
        }
        filterChain.doFilter(httpServletRequest, httpServletResponse);
    }

}
```

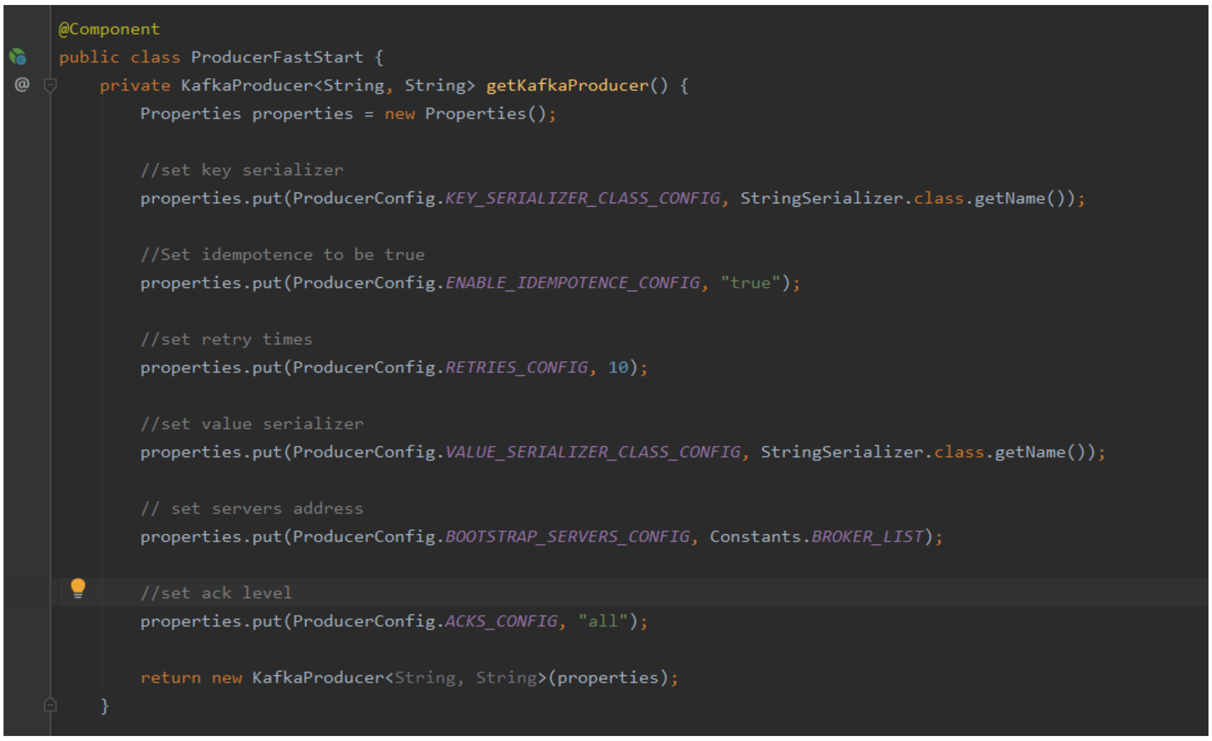
The project also extends OncePerRequestFilter. If there is a Token in the Authorization header of the user request, the JWTFilter will verify the token and check if the username and currency type is correct. If everything seems good, the request is allowed to be made.

Transaction security insurance



```
@Service
public class TransactionService {
    @Autowired
    TransactionDao transactionDao;

    @Transactional(isolation = Isolation.REPEATABLE_READ)
    public void makeTransaction(Transaction transaction) {
        String transactionId = Generator.generateUniqueId();
        String senderCurrencyType = transaction.getUsername().split( regex: "_")[1];
        String receiverCurrencyType = transaction.getReceiver().split( regex: "_")[1];
        double exchangeRate =
                Constants.exchangeRates[Constants.currencyTypeMap.get(senderCurrencyType)][Constants.currencyTypeMap.get(receiverCurrencyType)];
        LocalDateTime now = LocalDateTime.now();
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        String formattedTime = now.format(formatter);
        transaction.setTransactionId(transactionId);
        transaction.setTime(formattedTime);
        transactionDao.makeTransaction(transaction);
        Transaction transaction2 = new Transaction(transaction.getReceiver(),
                transaction.getReceiverId(), transaction.getUsername(), transaction.getUserId(),  transAmount: -transaction.getTransAmount() * ex
        transaction2.setTransactionId(transactionId);
        transactionDao.makeTransaction(transaction2);
    }
}
```

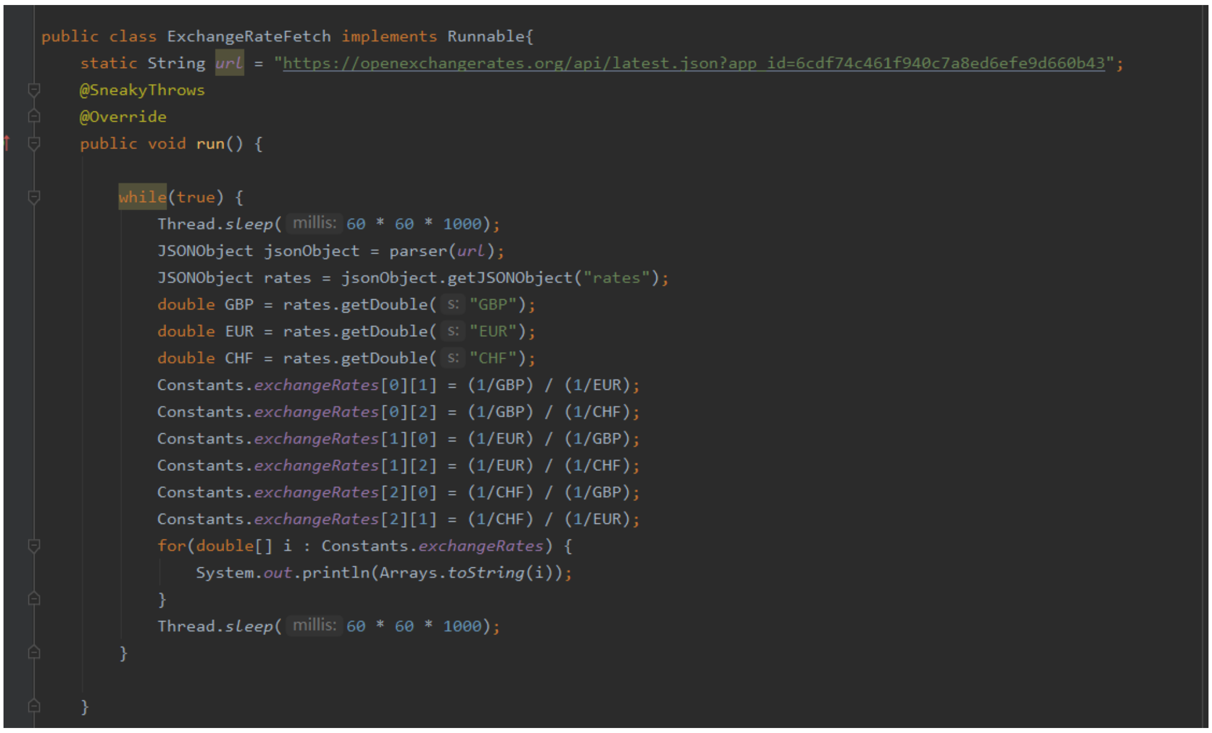
@transaction
To ensure that transactions are executed correctly, the project uses the @Transactional annotation with the isolation level set to Repeatable_read.



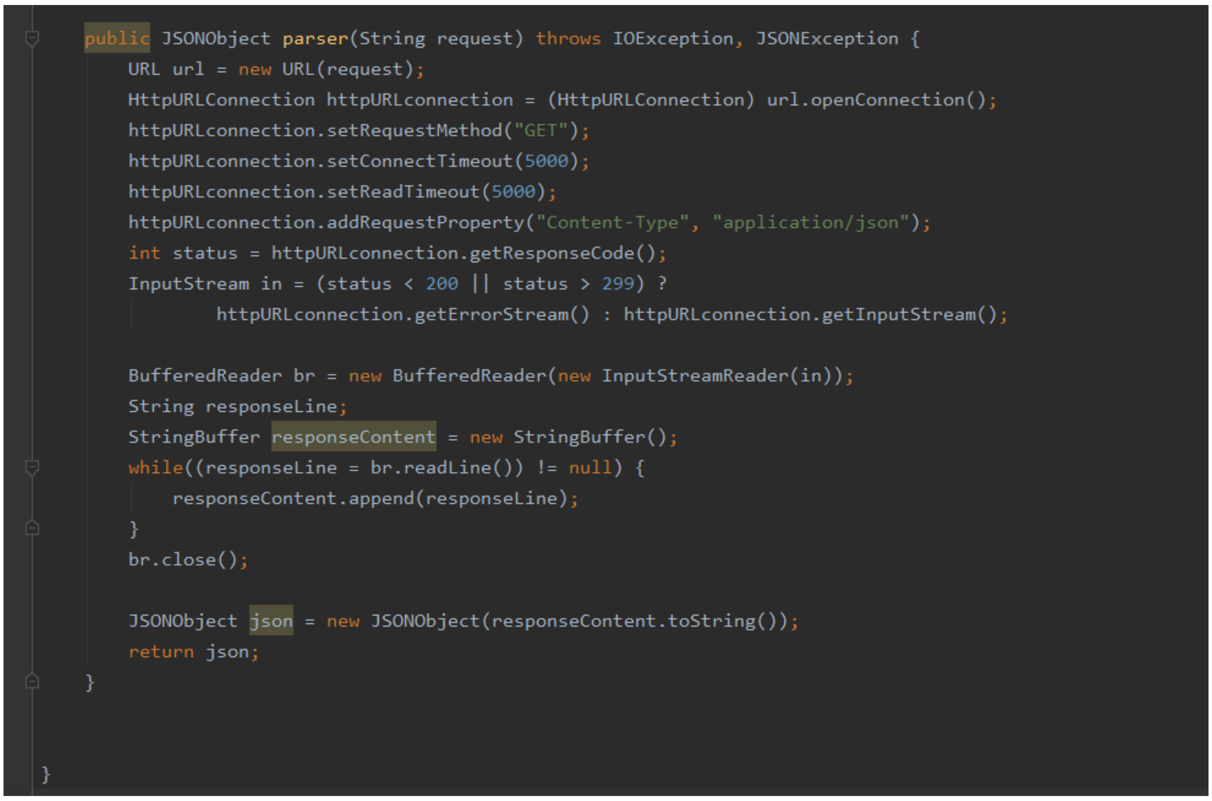
```java
@Component
public class ProducerFastStart {
    private KafkaProducer<String, String> getKafkaProducer() {
        Properties properties = new Properties();

        //set key serializer
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        //Set idempotence to be true
        properties.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");

        //set retry times
        properties.put(ProducerConfig.RETRIES_CONFIG, 10);

        //set value serializer
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        // set servers address
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, Constants.BROKER_LIST);

        //set ack level
        properties.put(ProducerConfig.ACKS_CONFIG, "all");

        return new KafkaProducer<String, String>(properties);
    }
```

Also, for the best security, the project sets the ACKS_CONFIG level to 'all', which ensures that the producer waits for all in-sync replicas to acknowledge the message before considering it being sent. This option provides the highest level of reliability but can impact performance since it requires more time to wait for acknowledgments from all the replicas.

Exchange Rate Fetch



```java
public class ExchangeRateFetch implements Runnable{
    static String url = "https://openexchangerates.org/api/latest.json?app_id=6cdf74c461f940c7a8ed6efe9d660b43";
    @SneakyThrows
    @Override
    public void run() {

        while(true) {
            Thread.sleep( millis: 60 * 60 * 1000);
            JSONObject jsonObject = parser(url);
            JSONObject rates = jsonObject.getJSONObject("rates");
            double GBP = rates.getDouble( s: "GBP");
            double EUR = rates.getDouble( s: "EUR");
            double CHF = rates.getDouble( s: "CHF");
            Constants.exchangeRates[0][1] = (1/GBP) / (1/EUR);
            Constants.exchangeRates[0][2] = (1/GBP) / (1/CHF);
            Constants.exchangeRates[1][0] = (1/EUR) / (1/GBP);
            Constants.exchangeRates[1][2] = (1/EUR) / (1/CHF);
            Constants.exchangeRates[2][0] = (1/CHF) / (1/GBP);
            Constants.exchangeRates[2][1] = (1/CHF) / (1/EUR);
            for(double[] i : Constants.exchangeRates) {
                System.out.println(Arrays.toString(i));
            }
            Thread.sleep( millis: 60 * 60 * 1000);
        }

    }
```



```java
    public JSONObject parser(String request) throws IOException, JSONException {
        URL url = new URL(request);
        HttpURLConnection httpURLconnection = (HttpURLConnection) url.openConnection();
        httpURLconnection.setRequestMethod("GET");
        httpURLconnection.setConnectTimeout(5000);
        httpURLconnection.setReadTimeout(5000);
        httpURLconnection.addRequestProperty("Content-Type", "application/json");
        int status = httpURLconnection.getResponseCode();
        InputStream in = (status < 200 || status > 299) ?
                httpURLconnection.getErrorStream() : httpURLconnection.getInputStream();

        BufferedReader br = new BufferedReader(new InputStreamReader(in));
        String responseLine;
        StringBuffer responseContent = new StringBuffer();
        while((responseLine = br.readLine()) != null) {
            responseContent.append(responseLine);
        }
        br.close();

        JSONObject json = new JSONObject(responseContent.toString());
        return json;
    }


}
```

After the server starts running, a thread is launched that fetches the latest exchange rates from the OpenExchangerates API every 2 hours.

Data Storage/Kafka

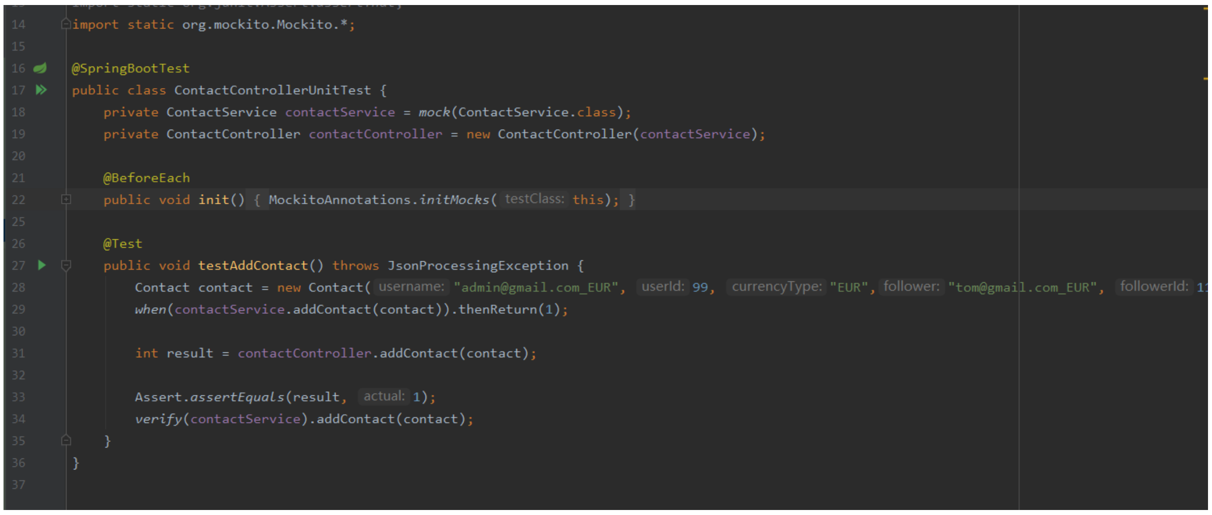
The project deploys 4 Zookeepers and 4 Kafka brokers that form a cluster to manage a total of 11 topics. These topics include a Test topic for storing test data, a User topic to store user data, 7 Transaction topics to store transactions, and 2 Contact topics to store contacts, with each topic having 50 partitions.

Container Management/Kubernetes

The project leverages a 6-node Tencent Cloud Kubernetes cluster to streamline the automated deployment, scaling, and monitoring of the application. This cluster includes 4 servers designated as Kafka brokers, 1 server as the backend, and 1 server as the frontend.

Tests

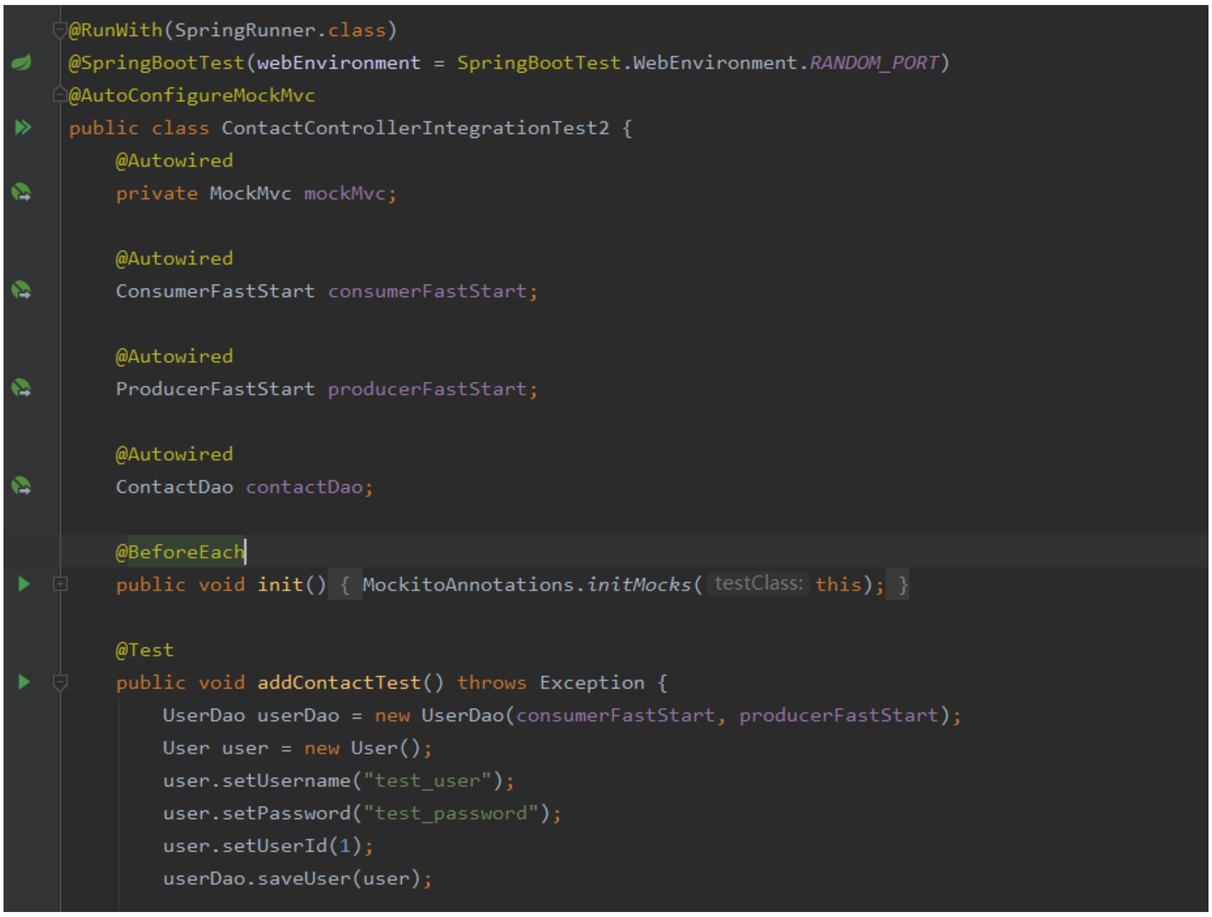
JUnit Test



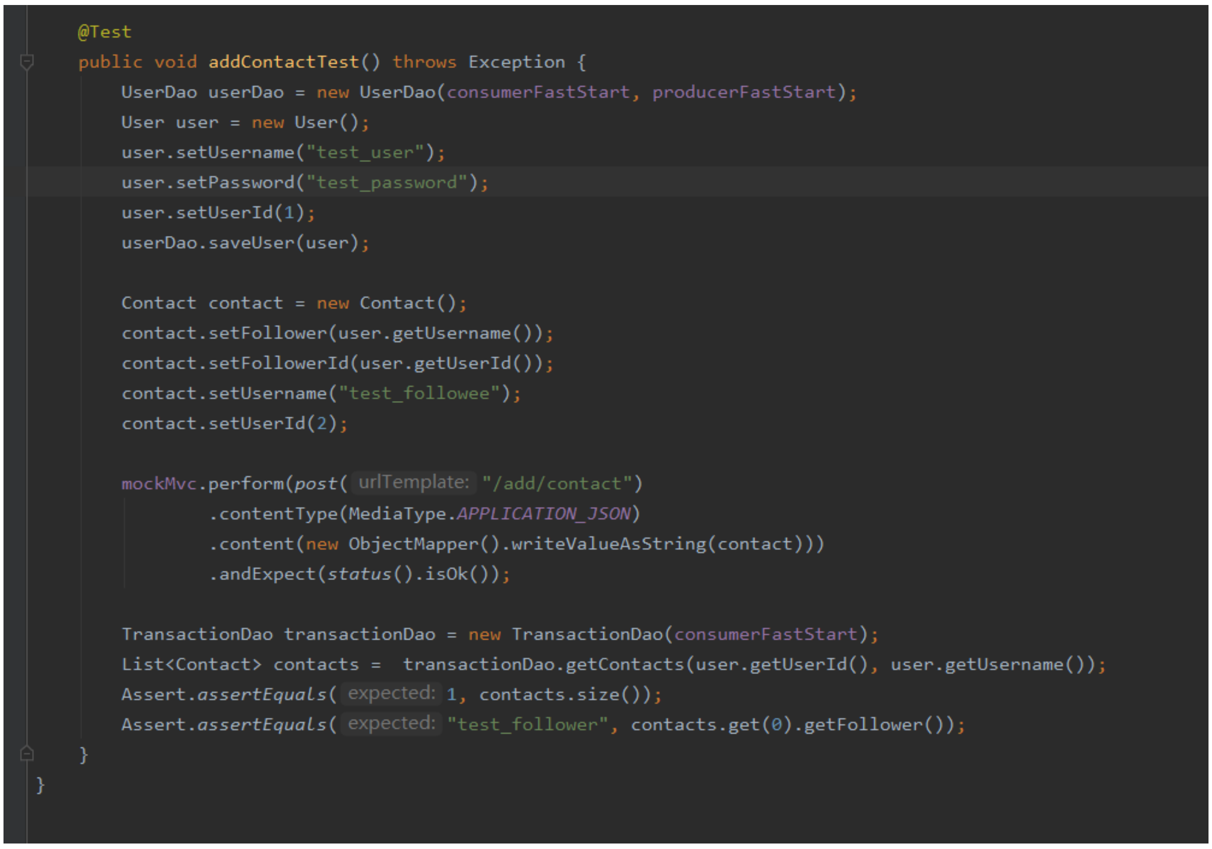
```java
import static org.mockito.Mockito.*;

@SpringBootTest
public class ContactControllerUnitTest {
    private ContactService contactService = mock(ContactService.class);
    private ContactController contactController = new ContactController(contactService);

    @BeforeEach
    public void init() { MockitoAnnotations.initMocks( testClass: this); }

    @Test
    public void testAddContact() throws JsonProcessingException {
        Contact contact = new Contact( username: "admin@gmail.com_EUR", userId: 99, currencyType: "EUR", follower: "tom@gmail.com_EUR", followerId: 1
        when(contactService.addContact(contact)).thenReturn(1);

        int result = contactController.addContact(contact);

        Assert.assertEquals(result, actual: 1);
        verify(contactService).addContact(contact);
    }
}
```

The project includes unit tests for each important small component. For example, in the above test, we used the mock() method to create a mock object of ContactService, and then passed it to the constructor of ContactController. We tested the correctness of the addContact() method, set the behavior of the mock object using the when() method, and verified whether the mock object was called using the verify() method.

Integration Test



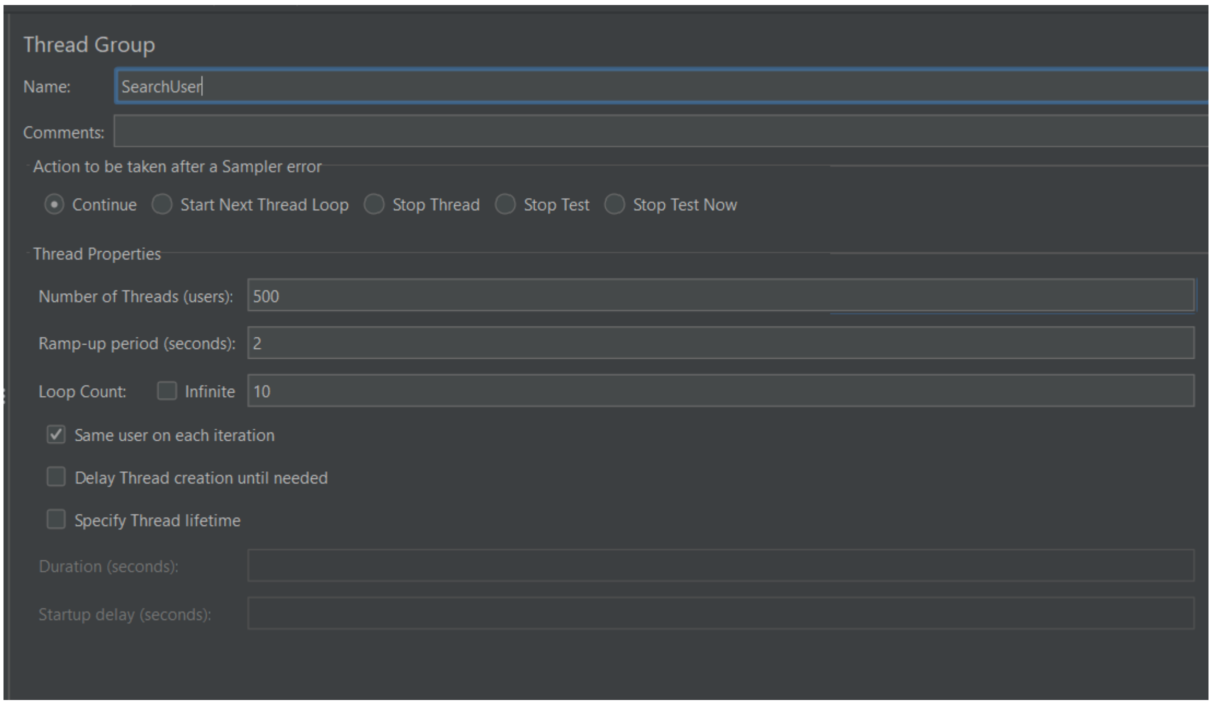
```java
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureMockMvc
public class ContactControllerIntegrationTest2 {
    @Autowired
    private MockMvc mockMvc;

    @Autowired
    ConsumerFastStart consumerFastStart;

    @Autowired
    ProducerFastStart producerFastStart;

    @Autowired
    ContactDao contactDao;

    @BeforeEach
    public void init() { MockitoAnnotations.initMocks( testClass: this); }

    @Test
    public void addContactTest() throws Exception {
        UserDao userDao = new UserDao(consumerFastStart, producerFastStart);
        User user = new User();
        user.setUsername("test_user");
        user.setPassword("test_password");
        user.setUserId(1);
        userDao.saveUser(user);
```



```java
    @Test
    public void addContactTest() throws Exception {
        UserDao userDao = new UserDao(consumerFastStart, producerFastStart);
        User user = new User();
        user.setUsername("test_user");
        user.setPassword("test_password");
        user.setUserId(1);
        userDao.saveUser(user);

        Contact contact = new Contact();
        contact.setFollower(user.getUsername());
        contact.setFollowerId(user.getUserId());
        contact.setUsername("test_followee");
        contact.setUserId(2);

        mockMvc.perform(post( urlTemplate: "/add/contact")
                .contentType(MediaType.APPLICATION_JSON)
                .content(new ObjectMapper().writeValueAsString(contact)))
                .andExpect(status().isOk());

        TransactionDao transactionDao = new TransactionDao(consumerFastStart);
        List<Contact> contacts =  transactionDao.getContacts(user.getUserId(), user.getUsername());
        Assert.assertEquals( expected: 1, contacts.size());
        Assert.assertEquals( expected: "test_follower", contacts.get(0).getFollower());
    }
}
```

The project has integration tests for each controller. For example, in above test, we use MockMvc to simulate HTTP requests, and set webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT in the SpringBootTest annotation to test with an actual HTTP port. We then call the mockMvc.perform() method to send a POST request and check if the returned status code is 200. After that, we verify in Kafka that the Contact object has been correctly added. Note that in the test, we use the actual Kafka connection and configuration. For testing purposes, there are over 10,000 users, each with approximately 1,000 transactions stored in Kafka.

JMeter API tests



Thread Group

Name: SearchUser

Comments:

Action to be taken after a Sampler error

[● Continue] ○ Start Next Thread Loop ○ Stop Thread ○ Stop Test ○ Stop Test Now

Thread Properties

Number of Threads (users): 500

Ramp-up period (seconds): 2

Loop Count: ☐ Infinite 10

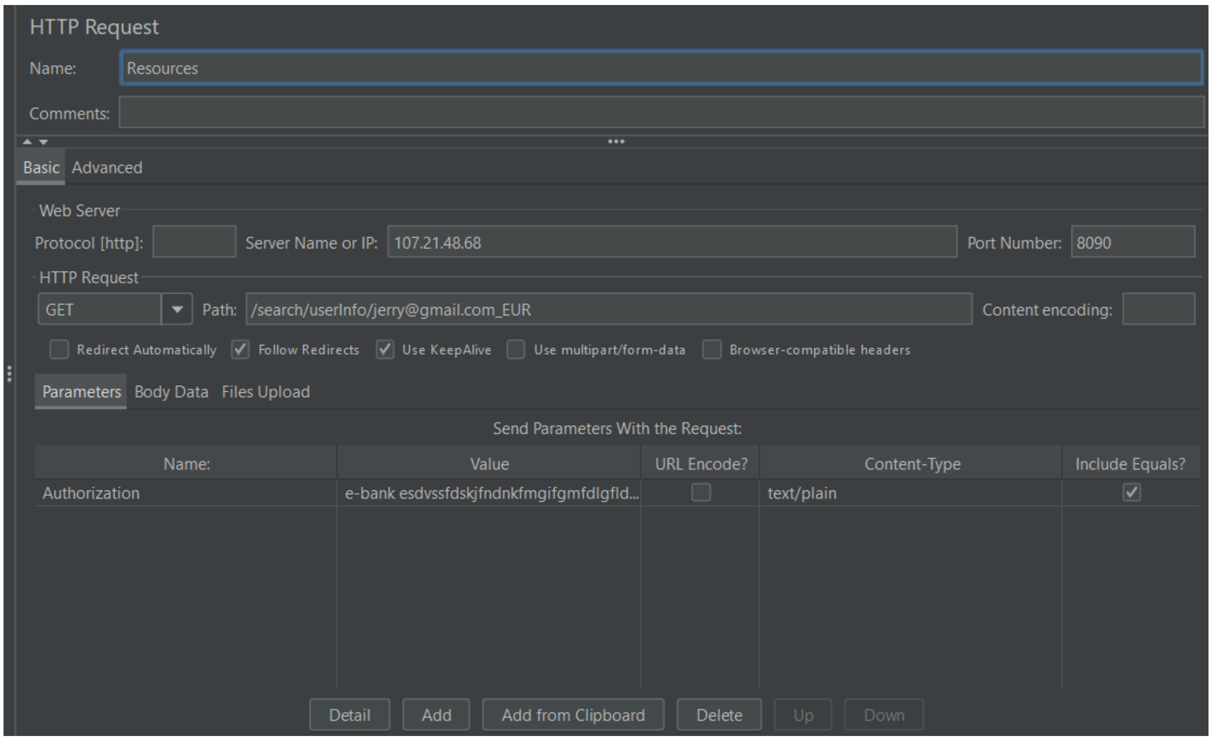
☑ Same user on each iteration

☐ Delay Thread creation until needed

☐ Specify Thread lifetime

Duration (seconds):

Startup delay (seconds):



HTTP Request

Name: Resources

Comments:

Basic Advanced

Web Server

Protocol [http]: Server Name or IP: 107.21.48.68 Port Number: 8090

HTTP Request

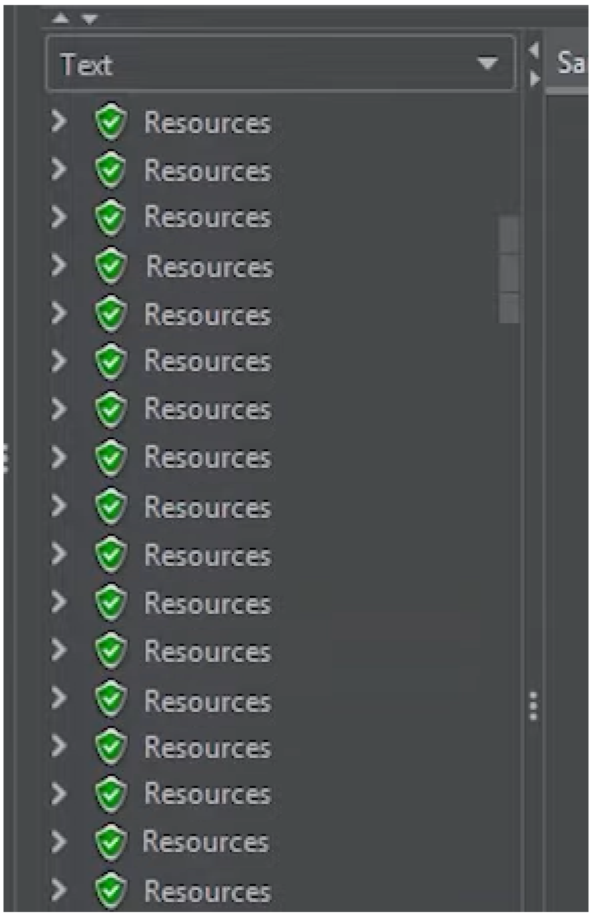
GET Path: /search/userInfo/jerry@gmail.com_EUR Content encoding:

☐ Redirect Automatically ✓ Follow Redirects ✓ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compatible headers

Parameters Body Data Files Upload

Send Parameters With the Request:

Name:	Value	URL Encode?	Content-Type	Include Equals?
Authorization	e-bank esdvssfdskjfndnkfmgifgmfdlgfld...	☐	text/plain	✓

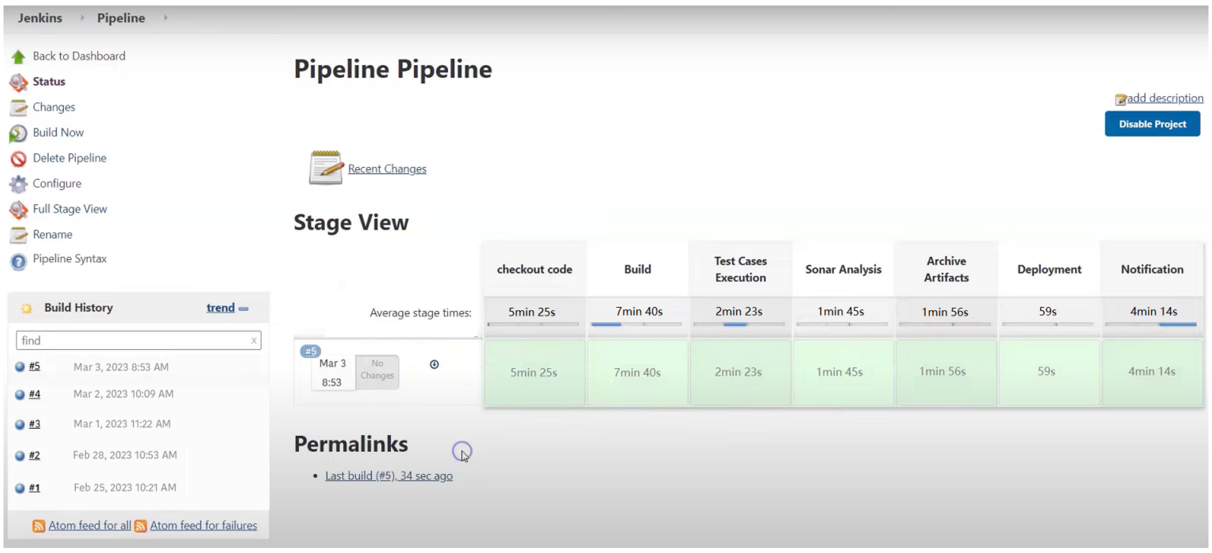


Text

Resources
Resources
Resources
Resources
Resources
Resources
Resources
Resources
Resources
Resources
Resources
Resources
Resources
Resources
Resources
Resources
Resources

Every API in the project has been tested using JMeter, and has been verified to handle up to 300 queries per second.

Jenkins CI/CD



Jenkins › Pipeline ›

- Back to Dashboard
- Status
- Changes
- Build Now
- Delete Pipeline
- Configure
- Full Stage View
- Rename
- Pipeline Syntax

Build History trend

find

#5	Mar 3, 2023 8:53 AM
#4	Mar 2, 2023 10:09 AM
#3	Mar 1, 2023 11:22 AM
#2	Feb 28, 2023 10:53 AM
#1	Feb 25, 2023 10:21 AM

Atom feed for all Atom feed for failures

Pipeline Pipeline

add description

Disable Project

Recent Changes

Stage View

	checkout code	Build	Test Cases Execution	Sonar Analysis	Archive Artifacts	Deployment	Notification
Average stage times:	5min 25s	7min 40s	2min 23s	1min 45s	1min 56s	59s	4min 14s
#5 Mar 3 8:53 No Changes	5min 25s	7min 40s	2min 23s	1min 45s	1min 56s	59s	4min 14s

Permalinks

- Last build (#5), 34 sec ago

The project is deployed through the use of Jenkins CI/CD pipeline.