Reference:

1. **Java Java Security Standard Algorithm Names**
   https://docs.oracle.com/en/java/javase/21/docs/specs/security/standard-names.html
   Notes: This serves as supplementary knowledge for enhancing the CryptoUtils implementation.
2. **Spring Cloud Gateway Server WebFlux**
   https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webflux.html
   Notes: My personal advice: Skim through the predicate/filters section to find the suitable built-in filters/ predicates for your use cases, reducing the need for writing a custom filter.
3. **Spring Security**
   https://docs.spring.io/spring-security/reference/index.html
   Notes: The documentation is extensive and covers all aspects of Spring Security. However, for better conceptual understanding, it is recommended to watch the following talks first and then use the documentation as a detailed reference.
   a. **Spring Security Architecture Principles by Daniel Garnier-Moiroux @ Spring I/O 2024**
      https://www.youtube.com/watch?v=HyoLl3VcRFY
   b. **Authorization in Spring Security: permissions, roles and beyond by Daniel Garnier-Moiroux @Spring IO**
      https://www.youtube.com/watch?v=-x8-s3QnhMQ
4. **OAuth2 Net**
   https://oauth.net/2/
   Notes: Use this as a foundational reference for understanding OAuth2 concepts, terminologies, and flow definitions. It provides the standard definitions that Spring Authorization Server and other frameworks are based on.
5. **A Short Note: Introduction to @Configuration and @Bean**
   https://sgnas.ifastfinancial.com/drive/oo/r/15PFtORR88HpADxnHTJ83rDY6T4X0e5m
   Notes: This document provides an overview of the conventions and justifications for using @Configuration and @Bean. It serves as a reference for establishing standardized coding practices that promote readability and consistency across the team. Feel free to review and update this document as needed.
6. **MB_AUTH_SD_Confidential Client Authentication**
   https://sgnas.ifastfinancial.com/drive/oo/r/15PFtP4SyWHnIkE9QSJAdTaAAlt5o9IL
   Notes: The solution design (SD) covers the security algorithm and their specifications, Software Bill of Materials (SBOM), exposed endpoints, and client registration details.
7. **A Short Note: Spring's Cloud Gateway**
   https://sgnas.ifastfinancial.com/drive/oo/r/15PFtOUYiZ10kKK1YGRZv3XWGD1kqk6f
   Note: This is a must-read document for developers working with Spring Cloud Gateway
8. **MB_AUTH_Login**
   https://sgnas.ifastfinancial.com/drive/oo/r/15PMA63tm9ZScVUF9H4gMoFnSeSwrWPw
   ***Note**: Provides a concise overview of the shared token and token exchange mechanisms designed for MY iFastPay login. It introduces how both mechanisms work*

and outlines their respective pros and cons. Both approaches are technically feasible but involve trade-offs in terms of cost and complexity.

9. **[MY iFAST PAY] Developer Starter Kit**
   https://sgnas.ifastfinancial.com/drive/oo/r/12fP5mVlnbMvvb3WLCiV65P4Dn2H06lh#slide_id=yMC7BwxAa9
   Note: *Covers most of the essential information required for development. This is a must-read document for all newcomers to the project.*
10. **[iGB] Developer Starter Kit**
    https://sgnas.ifastfinancial.com/drive/oo/r/pYf2EMhofzhd5lNJsBOloNXgCffcFhc1#slide_id=uzeEsvighO
    Note: *While the content largely overlaps with the [MY iFAST PAY] Developer Starter Kit, this document provides additional insight into specific areas such as Maker-Checker (pp. 48–58), Access Policy (pp. 59–62), and Backoffice Role-Rights (pp. 70–79). These sections are particularly useful for implementing access control and backoffice functionalities.*
11. **FSMOne x iFAST PAY integration Guidelines**
    https://sgnas.ifastfinancial.com/drive/oo/r/12qkoAPRikpsmIdhKG3TrLVHYgNbvUUn
    Note: *Becomes relevant once integration between FSMOne and iFastPay begins. It is especially valuable for understanding the Authentication and Login flows within iFastPay. Reviewing this guideline will help ensure smoother integration.*
12. **Technical Specification Documentation**
    https://sgnas.ifastfinancial.com/drive/oo/r/11LnhYSUKTDJ65Y4yb0ZASVzijq44nlV#tid=8
    Note: *Another must-read document for new developers. It should be reviewed thoroughly along with the two Developer Starter Kits mentioned above.*
13. **HTTP guides**
    https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides
    Note: *Provides comprehensive knowledge of HTTP fundamentals — including headers, authentication, caching, and Content Security Policy (CSP) directives. This is essential for understanding and improving the security layer configuration in iFastPay. The current setup is basic and lacks justification; further research and refinements are recommended.*
14. **OWASP API Security Top 10**
    https://owasp.org/API-Security/
    Note: *A valuable reference for designing secure APIs. Although not required for the initial implementation of authentication and authorization, reviewing OWASP's recommendations will contribute to a more secure and robust system. Recommended as supplementary reading if time permits.*
15. **docs-meeting-pre-27082025**
    https://sgnas.ifastfinancial.com/drive/d/f/15PNcXMzDablIDX0utoJ4zvMS46BRIcN
    Note: *Despite its name, this document provides important insights into security measures that should be implemented. It includes detailed references and enhancement suggestions. While the focus is primarily on application-level security, be mindful of its scope limitations. One enhancement worth noting is the recommendation*

to adopt RFC-compliant ProblemDetails and standardized ErrorResponse structures from Spring Boot, instead of maintaining a custom ErrorResponseModel, for improved future compatibility.

16. **bo-rbac-solution-planning**
    https://sgnas.ifastfinancial.com/drive/d/f/15PNcXwKHHjwz2jZbh6qwwHiYfQfkhUx
    Note*: This document proposes several approaches for implementing Backoffice Role-Based Access Control (RBAC) using Java 21 and Spring Boot 3.x, based on the measures used in iGB. For conceptual background on role-rights implementation, refer to the [iGB] Developer Starter Kit.*
17. **IT Main Sheet**
    https://sgnas.ifastfinancial.com/drive/oo/r/12QGjVNi5Ec4ufsvhjmYFsepYq8rp5yN

Trace Id
Status: *Work in Progress*

The setup for Trace ID propagation is mostly complete, but a few aspects still require validation and documentation.

Next Steps / Checklist:

1. Trigger an action in each service.
2. Verify that the Trace ID appears in both:
   a. Application logs, and
   b. The X-Trace-Id HTTP header across all services.

Confirm consistent propagation through asynchronous or threaded operations (e.g., batch jobs, thread pools). A detailed testing procedure will be documented once validation is complete.

Aspect	**Status**	**Changes needed**
Normal Synchronous Request	**Pending to check** Previously works well, but still need to double check before merge to develop/ branch.	
Feign Client	**Pending to check** Previously works well, but still need to double check before merge to develop/ branch.	
Cron Job	**Planned.** Previously not covered. Might need to cover it.	
Multithreading & Setup	**Pending to check** Previously works well, but still need to double check before merge to develop/ branch.	
Request Header (X-Trace-Id)	**Pending to check** Previously works well, but still need to double check before merge to develop/ branch.	
WebClient	**Planned.** Previously not covered. Might need to cover it.	
Message Queue	Not covered	

A Short note on CryptoUtils

Usages and Default algorithms

Refer to utils/CryptoUtils and the corresponding Generator/Runner under the my-oauth2-ws module for implementation references.

- **A brief introduction (Can Skip)**
    - **Key Generator**: Generates secret (symmetric) keys used for encryption and decryption algorithms such as AES.
    - **Key Pair Generator**: Generates a public/private key pair used in asymmetric encryption algorithms such as RSA or EC (Elliptic Curve).
    - **Cipher**: Performs encryption and decryption operations using symmetric or asymmetric algorithms. It transforms plain text into cipher text (encryption) and vice versa (decryption).
    - **Message Digest**: Generates a fixed-size hash from a variable-length message, typically used for integrity verification. Algorithms include SHA-256, SHA-512, etc. In legacy systems, message digests were sometimes misused for key generation, which is insecure due to the lack of randomness and susceptibility to collision attacks. Modern implementations instead use PKCS#8-formatted keys with stronger algorithms for enhanced security.
    - **MAC (Message Authentication Code)**: Similar to a message digest but includes a secret key during hashing to provide both integrity and authentication. Common algorithms include HMAC-SHA256 and HMAC-SHA512.
    - **Key Derivation Function (KDF)**: Transforms a user-provided secret (like a password) into a cryptographically strong key by applying multiple rounds of hashing and salting. This prevents brute-force attacks. Examples include PBKDF2, bcrypt, and Argon2.
    - **Cryptographically Secure Pseudo-Random Number Generator (CSPRNG)**: Generates high-entropy random values that are unpredictable and suitable for cryptographic use cases, such as key generation and salt creation. Examples include SecureRandom in Java and /dev/urandom in Linux.
- **Reference**:
    - Refer to this docs for detailed specification (algorithm/param used) https://sgnas.ifastfinancial.com/drive/oo/r/15PFtP4SyWHnIkE9QSJAdTaAAlt5o9IL

How It Is designed

- **ShadedCryptoUtils**:
    - ShadedCryptoUtils encapsulates low-level cryptographic operations, making them safer and less error-prone to use.
    - It is named “shaded” because it is intentionally hidden from direct access to prevent misuse by developers unfamiliar with secure cryptography practices. The intent is to minimize the risk of weak or insecure algorithm usage by providing a controlled set of trusted cryptographic primitives.
    - Currently, it lacks defensive checks for invalid inputs, though its defects are

minimal. For future enhancements, additional input validation and defensive programming measures should be implemented.

- **CryptoUtils**:
    - CryptoUtils provides a higher-level, opinionated API built on top of ShadedCryptoUtils.
    - It does not aim for full flexibility but instead offers a secure and consistent set of defaults that balance security and performance. for developers to use without needing to know the security internals. All parameters, including salt, initialization vector (IV), and algorithm choices, follow industry best practices and secure defaults.
    - The goal is to allow developers to perform cryptographic operations safely without needing deep knowledge of security internals.
- **DefaultConstants**:
    - **DefaultConstants** defines a centralized set of cryptographic constants to improve maintainability and readability.
    - Although placing them directly inside CryptoUtils would be acceptable, keeping them in a dedicated constants class achieves better separation of concerns and makes configurations easier to manage or externalize.
- **RsaKeyUtils**:
    - RsaKeyUtils is responsible for managing RSA key pairs and related operations.
    - It could theoretically be integrated into CryptoUtils for centralized management, but it remains separate to isolate its unique logic, such as key generation, JWK handling, and private utility methods. This separation improves readability and simplifies referencing.
    - The justification for this split is modest and may be revisited in future refactoring efforts if consolidation becomes more practical.

Summary:
The current design distinguishes between:

- Encapsulation of low-level cryptographic details (ShadedCryptoUtils)
- High-level, opinionated utilities following best practices (CryptoUtils)

This separation ensures simplicity, prevents misuse of insecure algorithms, and reduces potential security vulnerabilities.

Maintaining this clear boundary between low-level implementations and developer-facing APIs is essential for both safety and ease of use in future development.

Authentication Overview

Reference:
https://sgnas.ifastfinancial.com/drive/oo/r/15PFtP4SyWHnIkE9QSJAdTaAAlt5o9IL

Notes:
Basically, we strengthen authentication by enforcing all internal repositories, such as my-web-ifast-pay and my-web-backoffice, to act as OAuth2 clients, which was not explicitly mentioned in the original solution design.

In summary: All repositories act as clients, except for the following:

- my-modular → purely a Resource Server
- api-gateway → acts as both Resource Server and Client
- my-oauth2-ws → Authorization Server

For my-ifast-pay db: Refer to application.yml (Copy the database url to IntelliJ Ultimate since database connection plugin is only available for IntelliJ Ultimate).

Naming Convention
Follow the naming conventions below for consistent client identification:

Context	**Naming Convention**	**Example**
Within my-ifast-pay	x-<repo-name>	x-my-web-ifast-pay
Internal clients	internal-<client-name>	internal-igb
External clients	public-<client-name>	public-visa

Note: This serves only as a reference. Update it/ modify it as needed.

my-batch-ws (client)
client id/ name: x-my-batch-ws
Secret: XMrZWdl9bYWjVPiLD7YDgNFmD/VO+DJ3J/svRKVxnrc=
Encrypted: $2a$10$dA1gPJA/s2o6kU9f0edbqeN/OYRf6YFA/AVj73GFhUKVq.cKFQmdK

my-modular-ws (resource)
client id/ name: x-my-modular-ws
Secret: lq29LsHjAijGzqZGYZ5XFmRtagRTXGyV86EHeArQZm4=
Encrypted: $2a$10$wtWI0SdvXM5ZNhsbco.wee4uyEe.m0VoO1sO40DgPKJe5hJkvMgiy

my-internal-api (resource, client)
client id/ name: x-my-internal-api
Secret: hq2JXeT8RgxjBub/nfwMMh2sjZ1m88nUjQbJB6wboI0=
Encrypted: $2a$10$st70jptP2Li9.oYA3R8wP.TBcxUsT1VzM578u5IdNk9.6Ba.L980G

my-public-api (resource, client)
client id/ name: x-my-public-api
Secret: wm4s3E1Sm8BG3Kbj5poww/g3SuuhAk2prADjNva2N3E=
Encrypted: $2a$10$q.iTlbVRSZp0E5LYlX0IguhgGkqe1G06iOw8oEGG9IuBBnfAZ6tua

my-web-backoffice (client)
client id/ name: x-my-web-backoffice
Secret: NHiVeGI/g8m1q2ohZ/6CXf8VhO55y6kowQl6QvFF2T8=
Encrypted:
$2a$10$EhFd7m8YSpgdDFeUgxp7pOiehWRwN3/0nKebX3JqOKF5UNJSH3g6m

my-web-ifast-pay (client)
client id/ name: x-my-web-ifast-pay
Secret: zac3cGg1ty6dIw0ESGyPjvqnv5BVsHsHyB2IfVE7a8I=
Encrypted: $2a$10$6ur0yAkdZeVFkfpGO7RKDuRwdvkdN2qFIWuazIiWqYFS9hVgluOW2

Postman/ Other testing mediums
client id/ name: x-test-postman
Secret: qjO1uEU2J62kw/4H62T/o3AZ4+55veD9pJ4hO0TnaMA=
Encrypted: $2a$10$AM93/tXtZVEqZrNLvw57/uPLS0103i8.RA.p1vE6G3vOOWcjpGt9K

All secrets/ encrypted secrets are automatically generated through Java code using the generators and runners described below.

1. Generator
    a. TokenSettingsJsonGenerator
    b. ClientSettingsJsonGenerator
    c. RegisteredClientSqlGenerator
2. Runner
    a. ClientSettingsJsonGeneratorRunner
    b. TokenSettingsJsonGeneratorRunner
    c. RegisteredClientSqlGeneratorRunner
    d. EncryptedSecretKeyGeneratorRunner
    e. JwkGeneratorRunner

Notes:

- **Conceptual Differences**
    - A *Generator* defines the logic for data or configuration generation.
    - A *Runner* provides an interface or entry point to execute the generator.
    - This separation follows the Separation of Concerns (SoC) principle, improving code readability and maintainability.
- **Usage Guidance**
    - Use RegisteredClientSqlGeneratorRunner to generate sql for inserting new clients.
    - Use EncryptedSecretKeyGeneratorRunner to generate secure client secrets. This runner internally uses CryptoUtils, which encapsulates cryptographically secure

utilities for secret generation. The output includes both the plain secret key and the encrypted version. The plain secret key should be used by the client itself while the encrypted version should be stored to database.
- Use JwkGeneratorRunner only when necessary. The JWK set should remain static unless a JWK rotation mechanism is implemented. Uncontrolled JWK regeneration can cause severe authentication issues.

Steps to Register a New Repository
Since new repositories are often introduced to meet evolving requirements, follow the steps below to register them properly within the OAuth2 ecosystem:

1. **Determine the Server Type**
    a. Identify whether the repository acts as a Client Server (initiates requests to others) or a Resource Server (receives and responds to requests). Some repositories can be both. This classification determines how the YAML configuration and security setup should be defined.
2. **Generate a Secret Key**
    a. Run the EncryptedSecretKeyGeneratorRunner. It uses CryptoUtils to generate a cryptographically strong secret key. The output includes both the plain and encrypted secret keys.
3. **Register the Client**
    a. Use the encrypted secret key in RegisteredClientSqlGeneratorRunner.
    b. Some examples are provided. Comment out or replace any existing client records as needed.
    c. Keeping old records may help with traceability, but remove them if security isolation is required.
    d. Always use the encrypted key as the BCryptPasswordEncoder will not decrypt plaintext values.
4. **Execute the SQL Script**
    a. Run the generated SQL to insert the new client record. Verify the insertion using SELECT * to confirm that data was persisted successfully.
5. **Configure the YAML File**: Adjust the configuration based on the repository type. Examples are provided.
6. Define the SecurityFilterChain
    a. Configure the security chain according to the server role:
        i. **Client Server (e.g., my-batch-ws)**
            1. Deny all inbound requests by default.
            2. Disable unused features unless explicitly required for outbound API calls.
        ii. **Resource Server**
            1. Authenticate all incoming requests.
            2. Use fine-grained endpoint rules for public access.

Client
```
cloud:
    // Setup this for Resilience, a good to have feature
    openfeign:
      circuitbreaker:
        enabled: true
    // Consider to use this for intercept the OAuth2 access token.
      oauth2:
        clientRegistrationId: api-client
        enabled: true
      client:
        config:
          default:
            connectTimeout: 3000
            readTimeout: 3000

    security:
    oauth2:
      client:
        provider:
          oauth2-auth-server:
            token-uri: ${oauth2-auth-server.url}/oauth2/token
            jwk-set-uri: ${oauth2-auth-server.url}/oauth2/jwks
        registration:
          <your-client-id>:
            provider: oauth2-auth-server
            client-id: <your-client-id>
            client-secret: <your-client-secret>
            authorization-grant-type: client_credentials
            client-authentication-method: client_secret_basic
```

Notes: Only change <your-client-id> and <your-client-secret> according to your use Also, make sure to define the oauth2-auth-server.url property. Other configurations c safely copied and reused without modification.

Programmatic way for Feign Interceptors

```
package com.ifast.ipaymy.batch.ws.config.feign;

import feign.RequestInterceptor;
import feign.RequestTemplate;
import org.apache.http.HttpHeaders;
import org.springframework.beans.factory.annotation.Value;
import
org.springframework.boot.autoconfigure.security.oauth2.client.OAuth2Clien
tProperties;
```

```
import org.springframework.security.oauth2.client.OAuth2AuthorizeRequest;
import org.springframework.security.oauth2.client.OAuth2AuthorizedClient;
import
org.springframework.security.oauth2.client.OAuth2AuthorizedClientManager;
import org.springframework.security.oauth2.core.AbstractOAuth2Token;
import org.springframework.stereotype.Component;

import java.util.Optional;

@Component
public class OAuth2ClientCredentialsInterceptor implements
RequestInterceptor {

    private final String registrationId;
    private final OAuth2ClientProperties oauth2ClientProperties;
    private final OAuth2AuthorizedClientManager
oAuth2AuthorizedClientManager;

    public OAuth2ClientCredentialsInterceptor(OAuth2ClientProperties
oauth2ClientProperties,

OAuth2AuthorizedClientManager oAuth2AuthorizedClientManager,
                                              @Value(
                                                  "${oauth2-client.my-
ifast-pay.registration-id}")
                                              String registrationId) {
        this.oauth2ClientProperties = oauth2ClientProperties;
        this.oAuth2AuthorizedClientManager =
oAuth2AuthorizedClientManager;
        this.registrationId = registrationId;
    }

    @Override
    public void apply(RequestTemplate template) {
        OAuth2ClientProperties.Registration registration =
oauth2ClientProperties.getRegistration()
            .get(registrationId);

        String principalName =
Optional.ofNullable(registration.getClientName())
            .orElse("UNKNOWN-SYSTEM");

        OAuth2AuthorizeRequest oauth2AuthorizeRequest =
OAuth2AuthorizeRequest
```

```
            .withClientRegistrationId(registrationId)
            .principal(principalName)
            .build();

        // 2. Get the access token from Authorization Server
        OAuth2AuthorizedClient oAuth2AuthorizedClient =
oAuth2AuthorizedClientManager.authorize(
            oauth2AuthorizeRequest);

        String token = Optional.ofNullable(oAuth2AuthorizedClient)
            .map(OAuth2AuthorizedClient::getAccessToken)
            .map(AbstractOAuth2Token::getTokenValue)
       .orElseThrow(() -> new IllegalStateException("Failed to obtain
access token"));

        // 3. Inject the token into the header
        template.header(HttpHeaders.AUTHORIZATION, "Bearer " + token);
  }
}
```

Notes: When using a programmatic configuration, you must explicitly define the oaut client.my-ifast-pay.registration-id property.

Resource

```
security:
    oauth2:
      resourceserver:
        jwt:
          jwk-set-uri: ${oauth2-auth-server.url}/oauth2/jwks
      client:
        provider:
          oauth2-auth-server:
            token-uri: ${oauth2-auth-server.url}/oauth2/token
            jwk-set-uri: ${oauth2-auth-server.url}/oauth2/jwks
        registration:
          <your-client-id>:
            provider: oauth2-auth-server
            client-id:  <your-client-id>
            client-secret: <your-client-id>
            authorization-grant-type: client_credentials
            client-authentication-method: client_secret_basic
```

Notes: Only change <your-client-id> and <your-client-secret> according to your use

Also, make sure to define the oauth2-auth-server.url property. Other configurations c safely copied and reused without modification.

SecurityFilterChain

```
@Configuration
public class OAuth2SecurityConfig {

    @Bean
    public SecurityFilterChain oauth2SecurityFilterChain(HttpSecurity
http)
        throws Exception {
        // @formatter:off
        return http
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().denyAll()
            )
            .cors(AbstractHttpConfigurer::disable)
            .httpBasic(AbstractHttpConfigurer::disable)
            .csrf(AbstractHttpConfigurer::disable)
            .formLogin(AbstractHttpConfigurer::disable)
            .build();
        // @formatter:on
    }
}
```

```
@EnableMethodSecurity
public class WebSecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
        throws Exception {
        return http
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .cors(AbstractHttpConfigurer::disable)
            .httpBasic(AbstractHttpConfigurer::disable)
            .csrf(AbstractHttpConfigurer::disable)
            // Disable Login first, adapted to changes
            .formLogin(AbstractHttpConfigurer::disable)
            .build();
    }

    @Bean
    @Profile(SharedConstant.LOCAL)
    public UserDetailsService userDetailsService() {
        UserDetails user = User.withUsername("user")
```

```
            .password("{noop}secret123")
            .build();

        return new InMemoryUserDetailsManager(user);
    }
}
```

Notes:

- For client-facing interfaces, it is generally recommended to enable: CORS, CSR and formLogin. However, this depends on business requirements, so treat this s as a reference only.
- For local testing, it is advisable to configure an in-memory user service to ensur every request can be authenticated easily. This allows developers to test endpoi without being restricted by production-level security rules.
- Annotate your configuration class with @EnableWebSecurity and @EnableMethodSecurity to activate Spring Security support.
- Typically:
  - Use .authenticated() for all requests.
  - Permit only authentication-related endpoints, such as /login and /token.
- For the API Gateway, use @EnableWebFluxSecurity and configure a SecurityWebFilterChain when working with reactive (WebFlux) applications. Fail do so will result in runtime errors due to incompatible security configurations.

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class OAuth2SecurityConfig {

    @Bean
    public SecurityFilterChain oauth2SecurityFilterChain(HttpSecurity
http,

OAuth2AccessDeniedHandler oAuth2AccessDeniedHandler,

OAuth2AuthenticationEntryPoint oAuth2AuthenticationEntryPoint)
        throws Exception {
        // @formatter:off
        return http
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .headers(headers -> headers
                .contentTypeOptions(contentTypeOptions -> {})
                .contentSecurityPolicy(csp -> csp
                    .policyDirectives(
                        "default-src 'none'; frame-ancestors 'none';
object-src 'none'; base-uri 'none'; form-action 'none';"
                    )
                )

.frameOptions(HeadersConfigurer.FrameOptionsConfig::sameOrigin)
            )
            .cors(AbstractHttpConfigurer::disable)
            .httpBasic(AbstractHttpConfigurer::disable)
            .csrf(AbstractHttpConfigurer::disable)
            .formLogin(AbstractHttpConfigurer::disable)
            .oauth2ResourceServer(jwt -> jwt
                .jwt(Customizer.withDefaults())
            )
            .exceptionHandling(exception -> exception
                .accessDeniedHandler(oAuth2AccessDeniedHandler)
                .authenticationEntryPoint(oAuth2AuthenticationEntryPoint)
            )
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )
            .build();
        // @formatter:on
    }
}
```

Notes: Resource server

Test Plan
Present as Postman Collection and will not explained here. Use for reference/overview only.

- my-ifast-pay-backend
    - my-batch-ws: Can call my-modular-ws
- my-ifast-pay-frontend
    - api-gateway
        - **/.well-known/oauth-authorization-server**
            - **Need further refinement**
        - /oauth2/jwks
        - /oauth2/token
            - main (status, X-Trace-Id, response body)
                - Post Request Script
                    - Status = 200
                    - X-Trace-Id= exists

            - Response body= contains token
        - Valid Basic Auth (Fullset of Scope)
        - Valid Basic Auth (Subset of scope)
    - exception
        - X-No Basic Auth
        - X-Unregistered Client
        - X-Invalid Basic Auth (clientId)
        - X-Invalid Basic Auth (clientName)
        - X-Invalid Basic Auth (No scope)
        - X-Invalid Basic Auth (More scope)
        - scope-handling
            - X-No Scope Provided
            - X-More Scope
        - grant-handling
            - X-No Grant Type Provided
            - X-Unknown Grant Type
            - X-Disallowed Grant Type
        - X-With Bearer Token
        - X-Incorrect Form Encoding (Request Body)
        -
  - Making request to my-modular-ws
    - main
        - Valid Bearer Token
    - exception
        - X-Invalid Bearer Token (Expired)
        - X-Invalid Bearer Token (Corrupted)
        - X-Invalid Bearer Token (Invalid iss)
        - X-With Basic Auth
- my-web-*
  - Making request to my-modular-ws

Security Configuration
Some simple setup has been configured for the sake of security. Some are left out due to the pending, confirmed requirements.

Security
- my-ifast-pay-backend
    - my-batch-ws
        - **Options**
            - Disabled: cors, httpBasic, csrf, formLogin
        - **Requests**
            - Deny All: All requests
    - my-modular-ws
        - **Options**:
            - **Disabled**: cors, csrf, httpBasic, formLogin
        - **Resource Server**:
            - JWT+ JWKs from oauth2 auth server; return unauthorized when not authenticated.
        - **Session Management:** Stateless
        - **Requests**:
            - Authenticated: All requests
        - **Options**:
            - Content Type Options: enabled
            - CSP: default-src 'none'; frame-ancestors 'none'; object-src 'none'; base-uri 'none'; form-action 'none';
            - frame options: same origin
    - my-oauth2-ws
        - Options:
            - Disabled: http basic, csrf, form login
        - Requests
            - Enabled: /.well-known/oauth-authorization-server, /oauth2/token, /oauth2/introspect, /oauth2/jwks
            - Others: Disabled by marking them as __disabled and deny all __disabled requests.
        - Exception Handling
            - Authentication: 403 Forbidden
            - Access Denied: 403 Forbidden
- my-ifast-pay-frontend
    - api-gateway
        - **Options**:
            - Disabled: csrf, httpBasic, formLogin
        - **Requests**:
            - Permit All: /oauth2/token, /.well-known/**, /actuator/**
            - Authenticated: Any requests
        - **Resource Server**: JWT

    - **Headers**:
        - CSP: default-src 'none'; frame-ancestors 'none'; object-src 'none'; base-uri 'none'; form-action 'none'
        - Referrer Policy: No referrer
- my-web-ifast-pay
    - **Requests**:
        - Authenticated: Any requests
    - **Options**:
        - Disabled: cors, httpBasic, csrf, formLogin
    - **In Memory User**
        - username: user
        - password: secret123
- my-web-backoffice
    - **Requests**:
        - Authenticated: Any requests
    - **Options**:
        - Disabled: cors, httpBasic, csrf, formLogin
    - **In Memory User**
        - username: user
        - password: secret123

About Spring Cloud Gateway

This section discuss the convention when dealing with api-gateway. This section aims to promote reusability and maintain clear documentation. Below are *personally preferred* conventions to be considered. Amend these conventions to fit the actual usage.

1. **Reuse existing filters first**.
    a. Always reuse global/ gateway filters provided by Spring Cloud Gateway whenever possible. This promotes reusability and ensures stability since these filters are well-tested and well-documented.
    b. Only write custom filters when existing ones do not meet your needs, which is rare in most cases. Ref:
        i. https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webflux.html
        ii. https://sgnas.ifastfinancial.com/drive/oo/r/15PFtOUYiZ10kKK1YGRZv3XWGD1kqk6f
2. **Integrate with the Spring Ecosystem first**.
    a. Whenever possible, integrate with existing Spring components instead of reinventing the wheel.
    b. For example, use Spring Security's Resource Server to validate incoming requests instead of manually calling the OAuth2 server using a WebClient. While the manual approach works, using the Resource Server is cleaner, has first-class support, and is easier to maintain.
    c. Only implement custom logic (referring to existing codebases like iGB or iGV) when standard solutions cannot fulfill your needs or require further customization.
3. **Organizing Global Filters**.
    a. If a global filter is reusable (and it usually should be), place it under the util/ package in the api-gateway module. Use @Configuration and @Bean annotations to enable it across different gateways. This promotes reusability and centralized configuration.
4. **Organizing Gateway Filters**.
    a. When writing Gateway Filters:
        i. **Business-Agnostic Filters**: If the filter is business-agnostic (not tied to a specific business logic), make it reusable. Implement it as a GatewayFilterFactory and annotate it with @Component. Reusable filters should be placed under the util/ package.
        ii. Business-Specific Filters: If the filter contains business-specific logic, first reconsider whether existing filters or reusable components can be used instead. Only create a custom filter when existing solutions do not meet your requirements. Business-specific filters should reside in their dedicated gateway modules and directories.
5. **Keeping GatewayFilterFactory in One Class**
    a. In most cases, a single GatewayFilterFactory class is sufficient. Avoid separating logic into multiple classes such as GatewayFilterFactory, GatewayFilter, and

RewriteFunction, since these components typically serve the same purpose (modifying requests or responses). Splitting them unnecessarily increases complexity without improving clarity. Only separate them if the logic is genuinely complex.

   b. If you need to set execution order, use new OrderedGatewayFilter(...) or, preferably, define filter order in YAML. And usually, you should rely on YAML-based ordering as it is explicit and easier to understand. Avoid programmatic ordering unless necessary, as it can lead to unpredictable and developer confusion.
   c. When dealing with complex operations, pause and evaluate:
      i. Can the logic be simplified?
      ii. Can it be broken down into smaller, reusable filters instead?

6. **Naming Convention**
   a. Use Verb + Subject or Verb-as-Noun naming patterns for filters. This improves clarity and consistency.
      i. Verb + Subject: AddRequestHeader, RemoveResponseHeader
      ii. Verb as Noun: TokenRelay, ResponseLogging
   b. Global Filters should be named as XGlobalFilter, while a Gateway Filter Factory should be named as XGatewayFilterFactory.

Discussion on Minor Changes

This section summarizes the minor changes made to the codebase, along with comments explaining their use cases, the problems they aim to solve, and the reasoning behind each design decision for future enhancements or refactoring.

These changes currently lack unit tests, so they may be somewhat error-prone. However, since the logic is relatively straightforward, potential defects should be minimal.

- **utils/cryptographic/**
    - **CryptoUtils**: Mainly used for cipher operations (encryption/decryption). Suitable use case: value masking. For further details, please refer to ***A Short Note on CryptoUtils***.
    - **RsaKeyUtils**: Handles public/private key pairs generated using the RSA algorithm, mainly for use with JWKs. It is less commonly used unless involved in JWK rotation.
- **utils/codec/Base64Utils**: Encapsulates common operations for Base64 encoding and decoding. It can serve as an adapter layer between Apache Commons Base64 and Java's built-in Base64 implementations. Java's built-in Base64 is preferred now.
- **utils/json/JsonUtils**: Encapsulates common ObjectMapper operations, such as serializing an object to a JSON string and deserializing JSON back to an object. Provides an unchecked-exception version of these operations to make the calling code cleaner and easier to read.
- **utils/querydsl/QueryDslUtil**: Intended to encapsulate common operations for building complex sorting maps. Although it is not fully generic, it significantly simplifies verbose and error-prone sorting statements.
- **utils/observability/**
    - This package provides the necessary components for database logging and context propagation for trace IDs.
    - **Database Logging**:
        - Currently, two mechanisms are available: OpenTelemetry (OTel) and p6spy.
            - p6spy allows inspection of the SQL values being executed but consumes more resources and is no longer actively maintained.
            - OpenTelemetry, on the other hand, is actively maintained and offers more features out of the box.
        - Spring Boot 4 is gradually adopting OpenTelemetry for observability, replacing Spring Sleuth. Therefore, OTel should be used by default, while p6spy may be retained for local debugging and SQL value printing.
        - DataSourceConfig, FormatterConfig, and PrettySqlFormat are related to p6spy configuration.
        - The formatter/ package handles SQL logging format. Modify it if the current layout is difficult to read or consumes excessive resources (no benchmarking has been done yet as other priorities took precedence).
        - FormatterConfig and OpenTelemetrySqlExporter are used for OpenTelemetry-

        based SQL logging.
    - **Context Propagation**
        - Trace-related information such as traceId and spanId is collectively referred to as context, which is stored using ThreadLocal.
        - Because ThreadLocal data is tied to the current thread, context information (like traceId) is lost when switching to another thread unless explicitly propagated. This explains why trace information may be missing in background or batch threads. Hence there's a need for context propagation.
        - The threadpool/ package enables context propagation within thread pools. It uses the decorator pattern (wrapping ThreadPoolTaskExecutor) to preserve the context by wrapping all thread pools via TracingThreadPoolBeanPostProcessor. This approach is harmless and should have minimal runtime overhead.
- **utils/wrapper/RequestWrapper**:
    - Encapsulates common operations for reading values from ServerHttpRequest or ServerWebExchange.
    - Use RequestWrapper.of(serverWebExchange) in custom gateway filters for convenience and consistency.
- **utils/exception/writer**:
    - **ErrorResponseWriter**:
        - Designed to simplify error response writing, though it has limited usage.
        - It defines a simple interface that allows developers to specify the HttpStatus and ErrorCode to produce standardized error responses (based on ErrorResponseModel) without needing to manage the underlying syntax differences between requests and responses.
        - Implementations should include a static factory method that accepts both the request and response. See HttpServletErrorResponseWriter for reference.
    - **ErrorCode**:
        - Acts as both an extension point and a contract for use with ErrorResponseWriter.
        - While a simpler implementation could allow direct passing of a String errorCode and String errorMessage, doing so would create an abstraction leak—developers might bypass enum-based codes and use custom messages directly, causing issues with i18n.
        - Enforcing the use of predefined error codes makes the design more robust and scalable.
    - **ErrorResponseWriters**:
        - A convenience/ companion class similar in role to Collections for the Collection framework. It uses method overloading to create the appropriate ErrorResponseWriter instance based on the provided parameters.
        - Developers can simply call ErrorResponseWriters.create(request, response).write(); without worrying about the specific implementation class. This approach makes error writing cleaner and more intuitive.
- **utils/idempotent, utils/audittrail, utils/apilog**

    - Added @ComponentScan, @EntityScan, and @EnableJpaRepositories annotations and removed corresponding annotations from my-modular-ws and my-batch-ws.
    - **Justification**:
        - These modules always require entities, repositories, and related components to function. By including these annotations within their respective packages, developers no longer need to configure them manually.
        - This adheres to the **Principle of Least Astonishment:** When developers use an annotation such as @EnableXSupport, they expect it to automatically include all necessary dependencies. If it doesn't, it causes confusion and wasted debugging effort.
        - This design also simplifies module imports in the future and improves refactor safety, since hard-coded strings are more likely to break during refactoring.
- **logback-spring.xml**: Updated to improve highlighting and simplify each column, making logs more readable and easier to debug.
- **spy.log**: A spy.log file is always generated after running the services. The root cause has not been identified yet, but it appears to have no functional impact. It is mostly an annoyance rather than a critical issue.
- **my-batch-ws/service/feign/fx/FxtsFeignService**:
    - I have moved the Config class inside FxtsFeignService. The rationale is to separate business-specific request interceptors from global interceptors.
    - The Config class can be moved out later if needed, but its location should be carefully reconsidered.
    - A dedicated folder for each Feign service is recommended for better organization.
    - Note: This change has not yet been tested, so it may introduce unforeseen issues.
    - However, separating global and business-specific interceptors significantly improves readability and maintainability.
- **utils/validators/AssertUtils, ValidatorUtils**
    - Enhanced ValidatorUtils to better handle double and float values.
    - Introduced AssertUtils to simplify exception throwing. Previously, code patterns looked like: if (ValidatorUtils.XXX()) { throw new …}. Now, this can be replaced with a cleaner syntax: AssertUtils.isNotEmpty(obj, ()-> new DomainException())
- **utils/security/AuthenticationUtils**
    - A convenient utility for retrieving the current authentication, authorities, and user details. Provide a clean facade for standardized, convenient. handling
    - Alternative approaches include using SecurityContextHolder.getContext().getAuthentication(), injecting a Principal object, or annotating a controller parameter with @AuthenticationPrincipal. @CurrentSecurityContext and SecurityContextHolderAwareRequestWrapper can be used as well.
    - Refer to the official documentation for more information.