sa-AssertionUtil

```
package com.example.demo.assertions;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Component;

@Component
@RequiredArgsConstructor
public class WritingStyle {

    private final DomainAssert<AccountErrorCodeEnum, AccountException>
accountDomainAssert;
    private final ErrorCodeDomainAssert<AccountErrorCodeEnum>
accountAssertions;

    public void doSomething() {
        // AssertionsWithContext
        AssertionsWithContext
            .validateOn(
                ValidatorsUtil.isNotBlank(""),
                () -> new
AccountException(AccountErrorCodeEnum.INVALID_ACCOUNT_ID)
            )
            .withContext(
                ErrorContext
                    .with("id", 12345L)
                    .thenWith("name", "fyiernzy")
            )
            .withLogMessage(() -> String.format("There’s an error, id=%s",
"12345"))
            .makeAssertion();

        // V1 - ErrorCode-based solution
        // Use an IfastPayException (a generic exception) to hold the enum.
        // Pros: Simple, straightforward, and less verbose.
        // Cons: Error-prone (no type checking for the error code); uses a
generic
        //       IfastPayException, which might not preserve domain-specific
details.
        AssertionUtil.notNull("", AccountErrorCodeEnum.INVALID_ACCOUNT_ID);

        // Variant of V1 that uses reflection to preserve both the exception
type and the error code.
        // Cons: Reflection adds overhead (memory + CPU). Also requires
registering a map, which increases manual effort and complexity.
        AssertionUtil.notNullReflected("",
AccountErrorCodeEnum.INVALID_ACCOUNT_ID);

        // V2 - Supplier-based solution
        // Developers provide an exception supplier.
        // Pros: Good balance between simplicity and preserving both the
exception type and error code type.
        // Cons: Slightly more verbose due to Supplier boilerplate.
        AssertionUtil.notNull(
            "",
            () -> new
AccountException(AccountErrorCodeEnum.INVALID_ACCOUNT_ID)
        );

        // Centralized static class that returns Suppliers for specific
exceptions.
        // Pros: Less verbose than writing Suppliers inline; preserves both
exception and error code information.
        // Cons: A central registry can undermine decoupling and become a
“god” class.
        AssertionUtil.notNull(
            "",
            Exceptions.supply(AccountErrorCodeEnum.INVALID_ACCOUNT_ID)
        );

        // Each specific exception provides a static supply() that returns a
Supplier.
        // Pros: Keeps things decoupled; preserves both exception and error
code.
        // Cons: Adds repetitive static methods to each exception; hard to
enforce consistent usage; can confuse semantics.
        AssertionUtil.notNull(
            "",
         AccountException.supply(AccountErrorCodeEnum.INVALID_ACCOUNT_ID)
        );

        // V3 - Bean-based solution
        // Define a DomainAssert<ErrorCode, Exception> bean and autowire it.
        // Pros: Retains V1’s simplicity while addressing its weaknesses;
enables centralized policies (logging, i18n, metrics) via DI.
        // Cons: Using a utility-style class as a Spring component can be
confusing. Requires declaring DomainAssert fields in classes, which might be
overlooked.
        accountDomainAssert.notBlank("hello",
AccountErrorCodeEnum.INVALID_ACCOUNT_ID);

        // Variant: specify only the ErrorCode type and use a generic
IfastPayException.
        // Pros: Simpler DI and wiring.
        // Cons: Falls back to a generic exception; weaker type-specific
```

```
handling.
        accountAssertions.notBlank("",
AccountErrorCodeEnum.INVALID_ACCOUNT_ID);

        // V4 - Fluent DSL
        // Use a fluent, chained assertion style (similar to AssertJ).
        // Pros: Preserves both exception and error code information; fluent
design is easy to extend and maintain; high cohesion and extensibility.
        // Cons: More verbose; a different style from typical guard checks;
each withException/assertThat chain may allocate more objects.
        FluentAssert
            .withException(() -> new
AccountException(AccountErrorCodeEnum.INVALID_ACCOUNT_ID))
            .assertThat("hello")
            .isNotBlank();

        FluentAssert
            .withException(() -> new
AccountException(AccountErrorCodeEnum.INVALID_ACCOUNT_ID))
            .assertThat(35)
            .isSmallerThan(64);

        FluentAssert
            .withErrorCode(AccountErrorCodeEnum.INVALID_ACCOUNT_ID)
            .assertThat("hello")
            .isNotBlank();

        FluentAssert
            .withErrorCodeAndMessage(
                AccountErrorCodeEnum.INVALID_ACCOUNT_ID,
                "This is an error message that will be logged using log.error"
            )
            .assertThat(25L)
            .isSmallerThan(78L);

        // Fluent, minimal “simple” variant that accepts Suppliers directly.
        FluentAssert.simple()
            .notBlank("", () -> new
AccountException(AccountErrorCodeEnum.INVALID_ACCOUNT_ID))
            .notNull("", () -> new
AccountException(AccountErrorCodeEnum.INVALID_ACCOUNT_ID));

        // Fluent variant with a default fallback error code; can override the
exception later.
        FluentAssert
            .withDefaultFallback(AccountErrorCodeEnum.INVALID_ACCOUNT_ID)
            .notBlank("")
            .notNull("")
            .withException(() -> new
AccountException(AccountErrorCodeEnum.INVALID_ACCOUNT_ID))
                .notBlank("");
    }
}
```