📗 FlowScript – Especificación Formal y Guía Maestra

Esta guía es la referencia **oficial, formal y definitiva** de FlowScript. Mi objetivo es ofrecerte una comprensión exhaustiva de la sintaxis, la semántica y las capacidades del lenguaje. Al terminar su estudio podrás modelar procesos, escribir lógica compleja e implementar flujos de trabajo robustos con confianza: dominarás FlowScript.

Este lenguaje está inspirado en BPMN:

https://www.youtube.com/watch?v=E4yHqTh7NMA

https://www.youtube.com/watch?v=BbT0IN3y2V4

https://www.youtube.com/watch?v=RtxViAI1VPE&t=30s

---

1. Filosofía y principios de diseño

FlowScript nace para unificar **el qué** con **el cómo** en el desarrollo de software orientado a procesos:

Enfoque tradicional	Brecha	FlowScript
Diagramas BPMN describen el flujo («qué») de forma visual y agnóstica.	El modelo y la ejecución viven separados.	**Texto isomórfico a BPMN**: cada elemento gráfico tiene una construcción sintáctica equivalente.
Código (Java, Python, C#) implementa la lógica («cómo»).		

Principios clave

Principio	Qué implica
Legibilidad unificada	Un único archivo `.flow` guarda estructura y lógica; es *la* fuente de verdad.
Separación de incumbencias	La orquestación del flujo reside en procesos; la lógica de negocio, en funciones.
Seguridad y previsibilidad	No hay `goto` arbitrarios; `ir_a` solo enlaza nodos definidos, garantizando rutas claras y fáciles de depurar.

---

2. Estructura léxica y sintaxis formal

2.1. Comentarios

Forma	Sintaxis	Ejemplo
Una línea	`# …`	`flowscript # Comentario`
Multilínea	`/* … */`	`flowscript /* Comentario extenso */`

2.2. Identificadores

- **Regla**: `[a-zA-Z_][a-zA-Z0-9_]*`
- **Sensibles** a mayúsculas y minúsculas.
- **Válidos**: `cliente_id`, `_temp`, `Paso_3_Validacion`.

- **Inválidos**: `3pasos`, `total-ventas`, `mi variable`.

2.3. Palabras reservadas

Categoría	Palabras
Estructura	`proceso`, `funcion`, `importar`, `importar_jar`, `como`, `retornar`
Flujo	`inicio`, `fin`, `tarea`, `gateway`, `ir_a`, `cuando`, `rama`, `unir`, `sino`
Control	`si`, `sino_si`, `intentar`, `capturar`, `lanzar`
Tipos / Valores	`entero`, `decimal`, `booleano`, `texto`, `lista`, `objeto`, `nulo`, `verdadero`, `falso`
Operadores	`y`, `o`, `no`
Futuras	`asinc`, `esperar`, `evento`, `clase`

2.4. Tipos de datos fundamentales

Tipo	Descripción	Ejemplo
entero	64 bits sin fracción	`42`, `1_000_000`
decimal	Doble precisión	`3.14`, `1.23e-5`

booleano	`verdadero` / `falso`	—
texto	Cadena Unicode `"..."`	`"Hola\n"`
lista	Colección ordenada `[…]`	`[1, "manzana"]`
objeto	Pares clave-valor `{…}`	`{ nombre: "Ana" }`
nulo	Ausencia intencional	`nulo`

2.5. Operadores (por precedencia)

Precedencia	Operador	Significado	Asociatividad
15	`. []`	Acceso / indexación	Izq.
14	`()`	Llamada a función	Izq.
13	`no`	Negación lógica	Der.
12	`* / %`	Aritmética	Izq.
11	`+ -`	Aritmética	Izq.

10	`< > <= >=`	Comparación	Izq.
9	`== !=`	Igualdad	Izq.
8	`y`	AND lógico	Izq.
7	`o`	OR lógico	Izq.
1	`=`	Asignación	Der.

---

3. Funciones: lógica reutilizable

3.1. Definición y llamada

```
None
funcion calcular_impuesto(monto: decimal, tasa: decimal) -> decimal {

    si monto <= 0 { retornar 0.0 }

    retornar monto * tasa

}


total = calcular_impuesto(500.0, 0.19)
```

- `funcion` define, `retornar` devuelve.
- `-> tipo` se omite o usa `-> vacio` si no retorna.

3.3. Recursión

```
None

funcion factorial(n: entero) -> entero {

    si n <= 1 { retornar 1 }

    retornar n * factorial(n - 1)

}
```

---

4. Procesos: orquestación del flujo

Un `proceso` es el corazón de FlowScript. Define la secuencia de pasos, las decisiones y el paralelismo de un flujo de trabajo.

4.1. Estructura de un Proceso

- **`proceso <NombreProceso>`**: Define el contenedor principal.
- **`entrada`**: Una palabra clave especial que representa un objeto con los datos iniciales con los que se ejecuta el proceso.
- **`inicio`**: El punto de entrada único del proceso. Debe apuntar a un primer nodo.
- **`tarea <NombreTarea>`**: Una unidad de trabajo atómica. Contiene `accion`es a realizar.
- **`fin <NombreFin>`**: Un punto de terminación del proceso. Puede haber múltiples fines para representar diferentes resultados (éxito, error, etc.).
- **`ir_a <NodoDestino>`**: Instrucción exclusiva de las tareas para transferir el control a otro nodo (tarea, gateway o fin).

4.1. Estructura básica

```
None

proceso GestionCliente {

    inicio -> CargarCliente
```

```
    tarea CargarCliente {
        accion: cliente = db_get("clientes", entrada.id)
        ir_a ValidarActivo
    }

    tarea ValidarActivo {
        accion:
            si cliente.activo {
                ir_a EnviarCorreo
            } sino {
                ir_a FinInactivo
            }
    }

    tarea EnviarCorreo { /* … */ }

    fin FinInactivo
}
```

4.2. Objeto `contexto`

Cada instancia de un proceso en ejecución tiene un `contexto` implícito. Este es un objeto interno que almacena el estado del proceso. Todas las variables declaradas en el bloque `accion` de una `tarea` se guardan en este `contexto` y están disponibles para las tareas y gateways posteriores.

4.3. Gateways: Nodos de Decisión y Paralelismo

Los gateways dirigen el flujo por diferentes caminos. Se definen dentro de una `tarea` o como nodos independientes.

4.3.1. Gateway Exclusivo (XOR)

Solo un camino de salida es elegido. Las condiciones se evalúan en orden y se toma el primer camino cuya condición sea

None

```
proceso AprobacionFactura {

    inicio -> ClasificarMonto


    tarea ClasificarMonto {

        accion: gateway DecisionMonto {

            # Se evalúa cada 'cuando' en secuencia

            cuando entrada.monto > 10000 ->
RequiereAprobacionGerente

            cuando entrada.monto > 1000  ->
RequiereAprobacionSupervisor

            sino -> AprobacionAutomatica # 'sino' es el camino
por defecto si ninguna condición se cumple

        }

    }


    tarea AprobacionAutomatica { accion: imprimir("Aprobada
automáticamente"); ir_a FinOK }

    tarea RequiereAprobacionSupervisor { /* ... */ }
```

```
    tarea RequiereAprobacionGerente { /* ... */ }


    fin FinOK

}
```

4.3.2. Paralelo (AND)

Divide el flujo en múltiples ramas que se ejecutan **concurrentemente**. El flujo no continúa después del gateway hasta que **todas** las ramas paralelas hayan finalizado en un nodo de unión (`unir`).

```
None

proceso VerificacionAntecedentes {

    inicio -> Iniciar


    gateway Iniciar paralelo {

        rama -> VerificarCredito

        rama -> VerificarPenal

        unir -> Consolidar

    }


    tarea VerificarCredito {

        accion: reporte_credito = http.get("api/credito/" +
entrada.cedula)

        ir_a FinCredito

    }
```

```
    fin FinCredito


    tarea VerificarPenal {
        accion: reporte_penal = http.get("api/penal/" +
entrada.cedula)
        ir_a FinPenal
    }
    fin FinPenal


    tarea Consolidar {
        accion:
            si reporte_credito.aprobado y reporte_penal.limpio
{
                imprimir("Todo en orden"); ir_a Exito
            } sino { ir_a Falla }
    }


    fin Exito
    fin Falla
}
```

---

5. Manejo de Errores Robusto

Un lenguaje profesional necesita un mecanismo para manejar fallos inesperados. FlowScript introduce el bloque `intentar...capturar`.

None

```
funcion dividir_seguro(a, b) -> decimal {

    intentar {

        si b == 0 {

            lanzar { tipo: "ErrorMatematico", mensaje:
"División por cero" }

        }

        retornar a / b

    }

    capturar (e) {

        imprimir("Error: " + e.mensaje)

        retornar 0.0

    }

}


tarea ProcesarPago {

    accion: intento = http.post("api/pagos", entrada.pago)

    intentar {

        confirmacion = http.post("api/confirmar", { id:
intento.id })

        ir_a FinOK

    }

    capturar (ex) {
```

```
        imprimir("Falló confirmación: " + ex.mensaje)
        ir_a ReversarPago
    }
}
```

---

6. Modularidad e interoperabilidad

6.1. Módulos `.flow`

`utilidades.flow`

```
None
funcion es_par(n: entero) -> booleano { retornar n % 2 == 0 }

PI = 3.14159
```

`proceso_principal.flow`

```
None
importar "utilidades"

importar "std/json" como Json


proceso Principal {

    inicio -> Tarea


    tarea Tarea {

        accion:
```

```
            numero = 6
            si utilidades.es_par(numero) { imprimir("Es par")
}
            texto = Json.stringify({ valor: utilidades.PI })
            imprimir(texto)
        ir_a Fin
    }

    fin Fin
}
```

6.2. Integración con JAR (Java)

7. Biblioteca Estándar (`std`) Ampliada

FlowScript incluye una potente biblioteca estándar para tareas comunes.

- **`std/io`**:
  - `imprimir(...)`: Imprime en la consola.
  - `leer_linea(prompt: texto)`: Lee una línea de la entrada estándar.
- **`std/http`**:
  - `get(url, headers: objeto)`
  - `post(url, body: objeto, headers: objeto)`
  - `put(...)`, `delete(...)`
- **`std/json`**:
  - `parse(json_texto: texto) -> objeto | lista`: Convierte un texto JSON a un objeto/lista de FlowScript.
  - `stringify(valor: objeto | lista, indentacion: entero) -> texto`: Convierte un objeto/lista a su representación en texto JSON.
- **`std/db`**: (Simulación para un motor genérico SQL)
  - `query(sql: texto, params: lista) -> lista_de_objetos`
  - `execute(sql: texto, params: lista) -> entero` (retorna filas afectadas)
  - `get(tabla: texto, id: entero | texto) -> objeto`

    - `insert(tabla: texto, registro: objeto) -> id`
- **`std/fechas`**:
    - `ahora() -> entero`: Retorna la fecha/hora actual como timestamp Unix.
    - `formatear(timestamp: entero, formato: texto) -> texto`

None

```
importar_jar "libs/bouncycastle.jar" as crypto

importar_jar "libs/com.google.guava.jar" as guava


funcion generar_hash(data: texto) -> texto {

    hasher = crypto.MessageDigest.getInstance("SHA-256")

    bytes  = hasher.digest(data.getBytes("UTF-8"))

    retornar guava.io.BaseEncoding.base64().encode(bytes)

}
```

Módulo	Funciones principales
std/io	`imprimir(...)`, `leer_linea(prompt)`
std/http	`get(url, headers)`, `post(url, body, headers)`, `put`, `delete`
std/json	`parse(texto) -> objeto

std/db	`query(sql, params)`, `execute(sql, params)`, `get(tabla, id)`, `insert(tabla, registro)`
std/fechas	`ahora() -> timestamp`, `formatear(timestamp, formato)`

---

8. Caso práctico avanzado: proceso de *e-commerce*

None

```
# --- Importaciones ---

importar "std/http" as http

importar "std/db"  as db

importar "std/json" as json

importar_jar "libs/notificaciones.jar" as email


# --- Funciones de negocio ---

funcion validar_stock(items: lista) -> booleano {

    para cada item en items {

        prod = db.get("inventario", item.id_producto)

        si prod.stock < item.cantidad {

            lanzar { tipo: "ErrorStock", mensaje: "Sin stock
de " + prod.nombre }

        }

    }

    retornar verdadero
```

```
}


# --- Proceso principal ---
proceso ProcesarOrden {
    inicio -> Validar

    tarea Validar {
        accion:
            intentar {
                validar_stock(entrada.items)
                ir_a ProcesarPago
            }
            capturar (err) {
                imprimir("Stock insuficiente: " + err.mensaje)
                ir_a FinRechazado
            }
    }

    tarea ProcesarPago {
        accion:
            resp =
http.post("https://api.stripe.com/v1/charges", {
                monto: entrada.total,
```

```
            fuente: entrada.token_pago
        })
        si resp.status == "succeeded" {
            id_pago_confirmado = resp.id
            ir_a PrepararEnvioYNotificar
        } sino {
            ir_a FinPagoFallido
        }
}

# Gateway paralelo
gateway PrepararEnvioYNotificar paralelo {
    rama -> ActualizarInventarioYLogistica
    rama -> NotificarCliente
    unir -> CompletarOrden
}

tarea ActualizarInventarioYLogistica {
    accion:
        db.execute("START TRANSACTION")
        para cada item en entrada.items {
            db.execute(
```

```
                    "UPDATE inventario SET stock = stock - ?
WHERE id = ?",
                    [item.cantidad, item.id_producto]
                )
            }
            http.post("https://api.logistica.com/envios", {
orden_id: entrada.id })
            db.execute("COMMIT")
        ir_a FinLogistica
    }
    fin FinLogistica


    tarea NotificarCliente {
        accion:
            email.ClienteEmail.enviar(
                entrada.cliente.email,
                "Confirmación de tu orden #" + entrada.id,
                "Tu pago fue exitoso. Estamos preparando tu
envío."
            )
        ir_a FinNotificacion
    }
    fin FinNotificacion
```

```
    tarea CompletarOrden {
        accion:
            db.execute(
                "UPDATE ordenes SET estado = 'COMPLETADO'
WHERE id = ?",
                [entrada.id]
            )
            imprimir("Orden " + entrada.id + " completada.")
        ir_a FinExitoso
    }

    fin FinExitoso
    fin FinRechazado
    fin FinPagoFallido
}
```

Conclusión

Con esta guía completa, he recorrido desde los fundamentos léxicos hasta los patrones de diseño más avanzados de FlowScript. Ahora cuento con los conocimientos necesarios para modelar cualquier proceso de negocio con claridad y potencia sin precedentes. **Bienvenido a la maestría en FlowScript.**