

# Concurrencia Estructurada en Java 25

## De vuelta a la simplicidad sin sacrificar rendimiento

Andrés Alcarraz

Peru JUG - JConf Perú 2025

3 de diciembre de 2025



# ¿Sobre mí?

- Uruguay 🇺🇾
- Ingeniero Eléctrico e Ingeniero de Sistemas por la Universidad de la República del Uruguay
- Staff Software Engineer en Cabal Uruguay S.A.
- Más de 25 años desarrollando en Java
- Redes sociales:
  - @ [alcarraz@gmail.com](mailto:alcarraz@gmail.com)
  - [linkedin.com/in/andresalcarraz](https://www.linkedin.com/in/andresalcarraz)
  - [X/andresalcarraz](https://twitter.com/andresalcarraz)
  - [github.com/alcarraz](https://github.com/alcarraz)
  - [stackoverflow/3444205](https://stackoverflow.com/users/3444205)



# Agenda

## 1 Introducción

- Evolución de la Concurrencia en Java
- Programación estructurada
- Problema de ejemplo: Procesamiento de Transacción Financiera

## 2 Programación reactiva

## 3 Concurrencia Estructurada

## 4 La Funcionalidad Estrella: Fail-Fast

## 5 Scoped Values

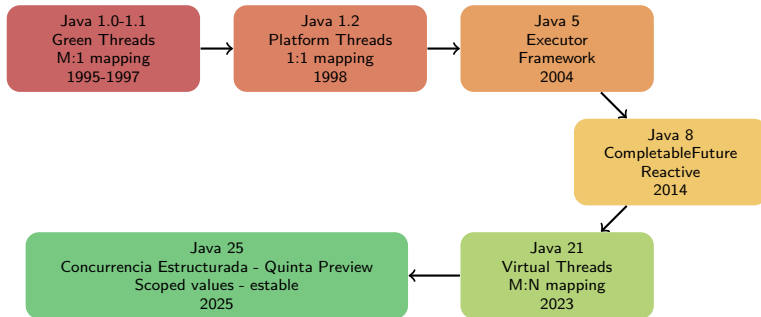
## 6 Temas avanzados

## 7 Conclusiones

## 8 Preguntas?

## 9 Referencias





## Estado en Java 25

- Virtual Threads: **Estable** (desde Java 21)
- Scoped Values: **Estable** desde Java 25 [JEP506]
- Concurrencia Estructurada: **Quinta Preview** en Java 25 [JEP505]

# ¡La promesa!

- Código que **se lee como se ejecuta**



# ¡La promesa!

- Código que **se lee como se ejecuta**
- Rendimiento equiparable a programación reactiva, con beneficios.



# ¡La promesa!

- Código que **se lee como se ejecuta**
- Rendimiento equiparable a programación reactiva, con beneficios.
- Sin callbacks.



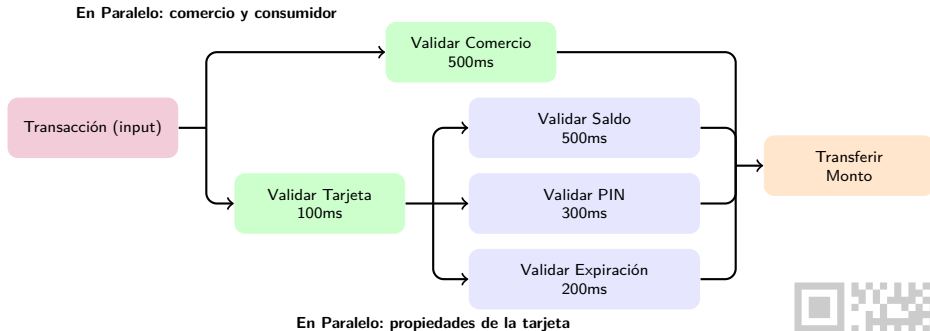
# ¡La promesa!

- Código que **se lee como se ejecuta**
- Rendimiento equiparable a programación reactiva, con beneficios.
- Sin callbacks.
- **De vuelta a la simplicidad**





# Flujo ficticio



## Flujo Optimizado

1. En paralelo: validar comercio y consumidor (tarjeta)
2. Validaciones paralelas de tarjeta (saldo, PIN, expiración)
3. Transferir sólo si todas las validaciones pasan

# Agenda

- 1 Introducción
- 2 Programación reactiva**
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Scoped Values
- 6 Temas avanzados
- 7 Conclusiones
- 8 Preguntas?
- 9 Referencias



# Problemas del Enfoque Reactivo

- **Legibilidad:** Callback Hell → Lógica anidada difícil de seguir



# Problemas del Enfoque Reactivo

- **Legibilidad:** Callback Hell → Lógica anidada difícil de seguir
- **Manejo de recursos:** No es trivial garantizar su liberación.



# Problemas del Enfoque Reactivo

- **Legibilidad:** Callback Hell → Lógica anidada difícil de seguir
- **Manejo de recursos:** No es trivial garantizar su liberación.
- **Excepciones:** Trazas del stack confusas, pueden saltar en el framework en lugar del código que ls genera



# Problemas del Enfoque Reactivo

- **Legibilidad:** Callback Hell → Lógica anidada difícil de seguir
- **Manejo de recursos:** No es trivial garantizar su liberación.
- **Excepciones:** Trazas del stack confusas, pueden saltar en el framework en lugar del código que ls genera
- **Carga cognitiva:** Alto costo mental para entender el código



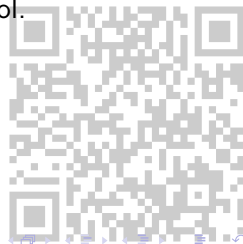
# Problemas del Enfoque Reactivo

- **Legibilidad:** Callback Hell → Lógica anidada difícil de seguir
- **Manejo de recursos:** No es trivial garantizar su liberación.
- **Excepciones:** Trazas del stack confusas, pueden saltar en el framework en lugar del código que ls genera
- **Carga cognitiva:** Alto costo mental para entender el código
- **Mantenibilidad:** Lógica de negocio mezclada con la lógica de control.



# Problemas del Enfoque Reactivo

- **Legibilidad:** Callback Hell → Lógica anidada difícil de seguir
- **Manejo de recursos:** No es trivial garantizar su liberación.
- **Excepciones:** Trazas del stack confusas, pueden saltar en el framework en lugar del código que ls genera
- **Carga cognitiva:** Alto costo mental para entender el código
- **Mantenibilidad:** Lógica de negocio mezclada con la lógica de control.
- **Depuración:** Difícil de seguir la secuencia de instrucciones.

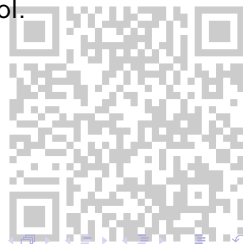




# Problemas del Enfoque Reactivo

- **Legibilidad:** Callback Hell → Lógica anidada difícil de seguir
- **Manejo de recursos:** No es trivial garantizar su liberación.
- **Excepciones:** Trazas del stack confusas, pueden saltar en el framework en lugar del código que ls genera
- **Carga cognitiva:** Alto costo mental para entender el código
- **Mantenibilidad:** Lógica de negocio mezclada con la lógica de control.
- **Depuración:** Difícil de seguir la secuencia de instrucciones.

► Veamos un código de ejemplo



# Agenda

- 1 Introducción
- 2 Programación reactiva
- 3 Concurrencia Estructurada**
  - Joiners: Estrategias de Coordinación
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Scoped Values
- 6 Temas avanzados
- 7 Conclusiones
- 8 Preguntas?
- 9 Referencias



- **Código Legible:** Se lee como se ejecuta



- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático



- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático
- **Depuración:** Las trazas del stack muestran dónde se produce el error.



- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático
- **Depuración:** Las trazas del stack muestran dónde se produce el error.
- **Rendimiento:** Ejecución paralela similar, no se pierde.



- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático
- **Depuración:** Las trazas del stack muestran dónde se produce el error.
- **Rendimiento:** Ejecución paralela similar, no se pierde.
- **Ciclo de vida estructurado:** Tareas anidadas mueren con el alcance



- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático
- **Depuración:** Las trazas del stack muestran dónde se produce el error.
- **Rendimiento:** Ejecución paralela similar, no se pierde.
- **Ciclo de vida estructurado:** Tareas anidadas mueren con el alcance
- **Fallar tempranamente por defecto:** `open()` cancela automáticamente si una tarea tira una excepción.





- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático
- **Depuración:** Las trazas del stack muestran dónde se produce el error.
- **Rendimiento:** Ejecución paralela similar, no se pierde.
- **Ciclo de vida estructurado:** Tareas anidadas mueren con el alcance
- **Fallar tempranamente por defecto:** `open()` cancela automáticamente si una tarea tira una excepción.
- **Observabilidad:** Mejor monitoreo y depuración, volcado estructurado de hilos

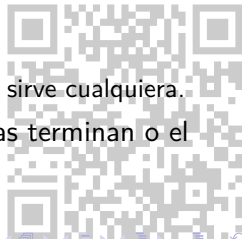


- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático
- **Depuración:** Las trazas del stack muestran dónde se produce el error.
- **Rendimiento:** Ejecución paralela similar, no se pierde.
- **Ciclo de vida estructurado:** Tareas anidadas mueren con el alcance
- **Fallar tempranamente por defecto:** `open()` cancela automáticamente si una tarea tira una excepción.
- **Observabilidad:** Mejor monitoreo y depuración, volcado estructurado de hilos

► Show me the code!!



- `awaitAllSuccessfulOrThrow()`
  - Fail-fast
  - Por defecto para `open()` sin argumentos
- `allSuccessfulOrThrow`
  - Igual que el anterior, para cuando todas las tareas deben de retornarretornan el mismo tipo.
  - `'join()'` devuelve un stream de resultados
- `awaitAll()`
  - Espera por todas las tareas, incluso si una tira excepción.
  - Útil para cuando no debemos cancelar tareas en proceso.
- `anySuccessfulResultOrThrow()`
  - Termina con la primer tarea sin error.
  - Útil para cuando queremos obtener un recurso de varios lugares y nos sirve cualquiera.
- `allUntil(Predicate<Subtask>)` - Termina cuando todas las tareas terminan o el predicado retorna true.



# Agenda

- 1 Introducción
- 2 Programación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast**
  - Cancelando tempranamente
  - Por Qué Esto Importa
  - Ejemplo de mejora
- 5 Scoped Values
- 6 Temas avanzados
- 7 Conclusiones
- 8 Preguntas?
- 9 Referencias



# ¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

## Reactive Basic: Sin fail-fast

- `allOf()` espera que TODAS las tareas terminen



# ¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

## Reactive Basic: Sin fail-fast

- `allOf()` espera que TODAS las tareas terminen
- Sin cancelación → desperdicio de recursos



# ¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

## Reactive Basic: Sin fail-fast

- `allOf()` espera que TODAS las tareas terminen
- Sin cancelación → desperdicio de recursos
- 570ms incluso si falla en 200ms



# ¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

## Reactive Basic: Sin fail-fast

- `allOf()` espera que TODAS las tareas terminen
- Sin cancelación → desperdicio de recursos
- 570ms incluso si falla en 200ms

## Reactive "Fixed": Fail-fast manual





# ¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

## Reactive Basic: Sin fail-fast

- `allOf()` espera que TODAS las tareas terminen
- Sin cancelación → desperdicio de recursos
- 570ms incluso si falla en 200ms

## Reactive "Fixed": Fail-fast manual

- Se puede lograr ... pero requiere lógica compleja



# ¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

## Reactive Basic: Sin fail-fast

- `allOf()` espera que TODAS las tareas terminen
- Sin cancelación → desperdicio de recursos
- 570ms incluso si falla en 200ms

## Reactive "Fixed": Fail-fast manual

- Se puede lograr ... pero requiere lógica compleja
- `failFast.completeExceptionally()` + coordinación manual



# ¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

## Reactive Basic: Sin fail-fast

- `allOf()` espera que TODAS las tareas terminen
- Sin cancelación → desperdicio de recursos
- 570ms incluso si falla en 200ms

## Reactive "Fixed": Fail-fast manual

- Se puede lograr ... pero requiere lógica compleja
- `failFast.completeExceptionally()` + coordinación manual

## Structured: Fail-fast automático

# ¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

## Reactive Basic: Sin fail-fast

- `allOf()` espera que TODAS las tareas terminen
- Sin cancelación → desperdicio de recursos
- 570ms incluso si falla en 200ms

## Reactive "Fixed": Fail-fast manual

- Se puede lograr ... pero requiere lógica compleja
- `failFast.completeExceptionally()` + coordinación manual

## Structured: Fail-fast automático

- `StructuredTaskScope.open()` = fail-fast por defecto

# ¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

## Reactive Basic: Sin fail-fast

- `allOf()` espera que TODAS las tareas terminen
- Sin cancelación → desperdicio de recursos
- 570ms incluso si falla en 200ms

## Reactive "Fixed": Fail-fast manual

- Se puede lograr ... pero requiere lógica compleja
- `failFast.completeExceptionally()` + coordinación manual

## Structured: Fail-fast automático

- `StructuredTaskScope.open()` = fail-fast por defecto
- Cancelación automática en cascada

## Impacto en la Experiencia de Usuario

- **Falla rápido:** Reporta errores inmediatamente, mejor UX



## Impacto en la Experiencia de Usuario

- **Falla rápido:** Reporta errores inmediatamente, mejor UX
- **Menor latencia** en casos de error



## Impacto en la Experiencia de Usuario

- **Falla rápido:** Reporta errores inmediatamente, mejor UX
- **Menor latencia** en casos de error
- **Feedback claro:** Usuario no espera innecesariamente





## Impacto en la Experiencia de Usuario

- **Falla rápido:** Reporta errores inmediatamente, mejor UX
- **Menor latencia** en casos de error
- **Feedback claro:** Usuario no espera innecesariamente

## Impacto en Recursos y Costos



## Impacto en la Experiencia de Usuario

- **Falla rápido:** Reporta errores inmediatamente, mejor UX
- **Menor latencia** en casos de error
- **Feedback claro:** Usuario no espera innecesariamente

## Impacto en Recursos y Costos

- **Eficiencia:** Cancela trabajo innecesario automáticamente



## Impacto en la Experiencia de Usuario

- **Falla rápido:** Reporta errores inmediatamente, mejor UX
- **Menor latencia** en casos de error
- **Feedback claro:** Usuario no espera innecesariamente

## Impacto en Recursos y Costos

- **Eficiencia:** Cancela trabajo innecesario automáticamente
- **Ahorro de CPU:** No procesa validaciones que no se usarán.



## Impacto en la Experiencia de Usuario

- **Falla rápido:** Reporta errores inmediatamente, mejor UX
- **Menor latencia** en casos de error
- **Feedback claro:** Usuario no espera innecesariamente

## Impacto en Recursos y Costos

- **Eficiencia:** Cancela trabajo innecesario automáticamente
- **Ahorro de CPU:** No procesa validaciones que no se usarán.
- **Menor carga:** Sistema responde más rápido en picos de errores



# Resumen de rendimiento: Éxito vs Fallo

## Caso Exitoso: Paridad

- Sin fallo temprano: 750ms
- Con fallo temprano: 750ms
- **Resultado:** Mismo rendimiento



# Resumen de rendimiento: Éxito vs Fallo

## Caso Exitoso: Paridad

- Sin fallo temprano: 750ms
- Con fallo temprano: 750ms
- **Resultado:** Mismo rendimiento

## ¿Por qué iguales?

Ambos deben esperar que todas las validaciones terminen.



# Resumen de rendimiento: Éxito vs Fallo

## Caso Exitoso: Paridad

- Sin fallo temprano: 750ms
- Con fallo temprano: 750ms
- Resultado: Mismo rendimiento

## ¿Por qué iguales?

Ambos deben esperar que todas las validaciones terminen.

## Caso Fallido: Más Rápido

- Reactive: 750ms
- Structured: 330ms
- Mejora: **56 %**



# Resumen de rendimiento: Éxito vs Fallo

## Caso Exitoso: Paridad

- Sin fallo temprano: 750ms
- Con fallo temprano: 750ms
- Resultado: Mismo rendimiento

## ¿Por qué iguales?

Ambos deben esperar que todas las validaciones terminen.

## Caso Fallido: Más Rápido

- Reactive: 750ms
- Structured: 330ms
- Mejora: **56 %**

## ¿Por qué?

- Sin fail fast: Esperar a que todas las validaciones terminen.
- Con fail fast: Cancela automáticamente cuando la primera falla.







## Conclusión

Mejor experiencia del usuario y menor uso de recursos en casos de error.  
Misma eficiencia en casos de éxito.

# Agenda

- 1 Introducción
- 2 Programación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Scoped Values**
- 6 Temas avanzados
- 7 Conclusiones
- 8 Preguntas?
- 9 Referencias



# Scoped Values = Contexto Sin Parameter Drilling

## El Problema

¿Cómo pasar contexto (audit info, request ID, user context) a través de validaciones paralelas?

- **Parameter drilling:** Pasar en cada firma de método (verbose, invasivo)
- **ThreadLocal:** Problemático con hilos virtuales masivos

## La Solución: Scoped Values

- **Propagación automática:** `ScopedValue.where(KEY, value).run(...)`
- **Acceso directo:** `KEY.get()` desde cualquier método
- **Thread-safe:** Funciona perfectamente con structured concurrency
- **Casos de uso:** Audit trails, request IDs, user context, tracing



## ► ¡ESTABLE en Java 25!

**JEP 506:** Scoped Values están ESTABLES.

¡Ya podemos usarlo en producción! 🎉

► Veamos la propagación de contexto en vivo...



# Agenda

- 1 Introducción
- 2 Programación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Scoped Values
- 6 Temas avanzados**
  - Best Practices: Patrones Recomendados
  - Anti-Patterns: Lo Que NO Debes Hacer
  - Custom Joiners: Estrategias de Coordinación
  - Scopes anidados: Composición Jerárquica
  - Timeout Patterns: Deadlines con Concurrencia Estructurada
- 7 Conclusiones
- 8 Preguntas?



## ✓ 1. Try-With-Resources Siempre

- Usa `try (var scope = ...)` para garantizar cleanup
- El scope cancela tasks pendientes automáticamente
- Evita leaks y garantiza resource management correcto

## ✓ 2. Manejo de Excepciones en Tareas

- Captura y maneja excepciones dentro de cada tarea forked
- Usa joiner apropiado: `fail-fast` para críticos, `awaitAll` para recolección
- Propaga contexto de error de forma clara

## ✓ 3. Scoped Values para Contexto

- Usa `ScopedValue` en lugar de `ThreadLocal`
- Inmutable, thread-safe, heredado por tareas forked
- Más eficiente con millones de virtual threads

## × Leaking Tasks

- **NO hagas:** Fork tasks sin asegurarte de esperar `join()`
- **Problema:** Tasks huérfanas, resource leaks
- **Solución:** Siempre usa try-with-resources

## × Ignorar InterruptedException

- **NO hagas:** Catch vacío o sin lógica de cancelación
- **Problema:** Los hilos dependen de interrupciones para cancelación
- **Solución:** Si no quieres relanzar la excepción, asegúrate de retornar lo más rápido posible y establecer la marca de interrupción

```
try {  
    doSomething();  
} catch (InterruptedException e) {  
    Thread.currentThread().interrupt();  
}
```

# Podemos implementar nuestros propios Joiners

```
// Custom Joiner: Quorum de validación anti-fraude
try (var scope = StructuredTaskScope.open(
    Joiner.quorum(2, 3))) { // 2 de 3 deben aprobar

    scope.fork(() -> fraudDetector1.check(transaction));
    scope.fork(() -> fraudDetector2.check(transaction));
    scope.fork(() -> fraudDetector3.check(transaction));

    scope.join();
    boolean approved = scope.result(); // Decisión por mayoría
}
```



## ¿Por qué Custom Joiner?

- `allUntil(Predicate)` siempre retorna `Stream<Subtask<T>>`
- Custom Joiner permite retornar tipos personalizados (Boolean, agregaciones, etc.)
- Implementa lógica de coordinación sofisticada: voting, quorum, best-result



```
try (var outerScope = StructuredTaskScope.open()) {  
    var userData = outerScope.fork(() -> {  
        // Inner scope para validaciones paralelas del usuario  
        try (var userScope = StructuredTaskScope.open()) {  
            var profile = userScope.fork(() -> fetchProfile(userId));  
            var preferences = userScope.fork(() -> fetchPreferences(userId));  
  
            userScope.join();  
            return new UserData(profile.get(), preferences.get());  
        }  
    });  
    var orderData = outerScope.fork(() -> fetchOrders(userId));  
    outerScope.join();  
    return combine(userData.get(), orderData.get());  
}
```



## Ventaja

Cancela automáticamente TODOS los scopes anidados si falla el padre

```
// Timeout con Configuration API (Java 25)
try (var scope = StructuredTaskScope.open(
    Joiner.allSuccessfulOrThrow(),
    config -> config.withTimeout(Duration.ofSeconds(5)))) {

    var result1 = scope.fork(() -> slowOperation1());
    var result2 = scope.fork(() -> slowOperation2());
    var result3 = scope.fork(() -> slowOperation3());

    scope.join(); // Cancela todos si se excede el timeout

    return combineResults(
        result1.get(), result2.get(), result3.get());

} catch (TimeoutException e) {
    return TransactionResult.failure(
        "Timeout: operations took longer than 5 seconds");
}
```



# Agenda

- 1 Introducción
- 2 Programación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Scoped Values
- 6 Temas avanzados
- 7 Conclusiones**
  - ¿Por qué adoptar concurrencia estructurada?
  - Mensajes Clave
- 8 Preguntas?
- 9 Referencias



## Beneficios Inmediatos

- **Legibilidad:** Código que lee como se ejecuta
- **Mantenibilidad:** Más fácil de modificar y extender
- **Depuración:** Stack traces claros y útiles
- **Rendimiento:** Misma velocidad, mejor código
- **Manejo de recursos:** Garantías automáticas de cleanup

## ★ Roadmap de Adopción

- **HOY (Producción):** Scoped Values (**ESTABLES** en Java 25)
- **AHORA (Experimentar):** Concurrencia Estructurada (quinta preview)
- **Java 26:** Aún en preview (sexta, [JEP 525](#)).
- **Java 29 (Siguierte LTS):** Concurrencia Estructurada estable 🍌



- 1 **Scoped Values ya son estables:** Empieza a adoptarlos
- 2 **De vuelta a la simplicidad sin sacrificar rendimiento**
- 3 **Concurrencia Estructurada aún está en preview**
- 4 **Código que lee como se ejecuta**
- 5 **Mejor experiencia del desarrollador sin trade-offs**

**GitHub:** <https://github.com/alcarraz/structured-concurrency>



# Agenda

- 1 Introducción
- 2 Programación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Scoped Values
- 6 Temas avanzados
- 7 Conclusiones
- 8 Preguntas?**
- 9 Referencias



# ¿Preguntas?

¡Gracias!

**Andrés Alcarraz**  
@ [alcarraz@gmail.com](mailto:alcarraz@gmail.com)



# Agenda

- 1 Introducción
- 2 Programación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Scoped Values
- 6 Temas avanzados
- 7 Conclusiones
- 8 Preguntas?
- 9 Referencias**





