Alles im Fluss

Kotlin Flows

Werner Eberling

E-Mail: werner.eberling@mathema.de
Mastodon: @Wer_Eb@social.dev-wiki.de
X / Twitter: @Wer_Eb

Der Sprecher

Werner Eberling

Principal Consultant / Autor

Email: werner.eberling@mathema.de

Mastodon: @Wer_Eb@social.dev-wiki.de

X (fka. Twitter): @Wer_Eb


Koroutinen
&
Strukturierte
Nebenläufigkeit

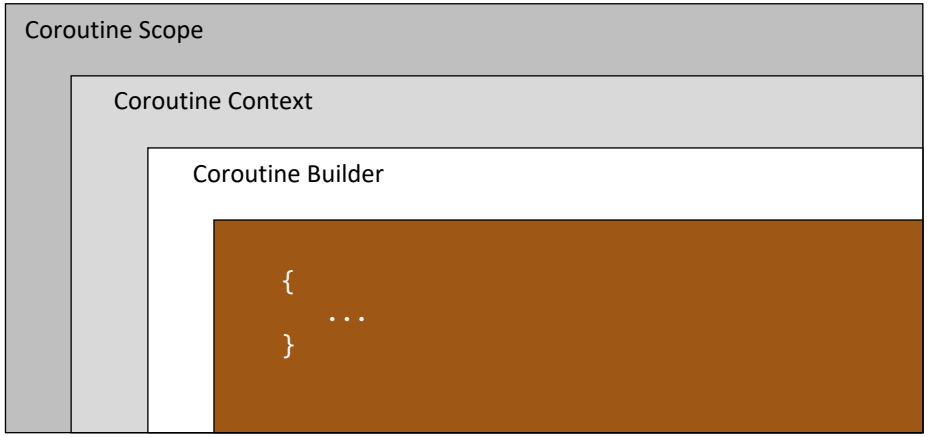
Strukturierte Nebenläufigkeit - Scopes

```
fun main() = runBlocking {
    launch {
        delay(1000L)
        println(“Campus!")
    }
    println("Willkommen zum")
}
```

- Koroutinen können unterbrochen und wieder fortgesetzt werden
- Koroutinen laufen immer innerhalb eines Scopes

  Start z.B. mit `runBlocking{…}`
- `runBlocking` als Brücke zwischen „normalem“ und „suspendable“ Code

Strukturierte Nebenläufigkeit



- Scopes

  Beschränkung der Laufzeit der Koroutinen (nur innerhalb des Scopes)

  Keine „wild laufenden“ Threads (auch nach Fehlern)

- Contexts

  Ablaufkontext der Koroutine (Welcher Thread arbeitet die Koroutine ab?)

- Builder

  Erzeugung konkreter Koroutinen

Koroutinen – Parallele Ausführung

```
fun main() = runBlocking {
    launch {
        delayedKKONWithNewJobs()
    }
    println("Willkommen zum")
}
suspend fun delayedKKONWithNewJobs() = coroutineScope {
    launch {
        delay(2000L)
        println("2024")
    }
    launch {
        delay(1000L)
        println(“Campus")
    }
    println(“MATHEMA")
}
```

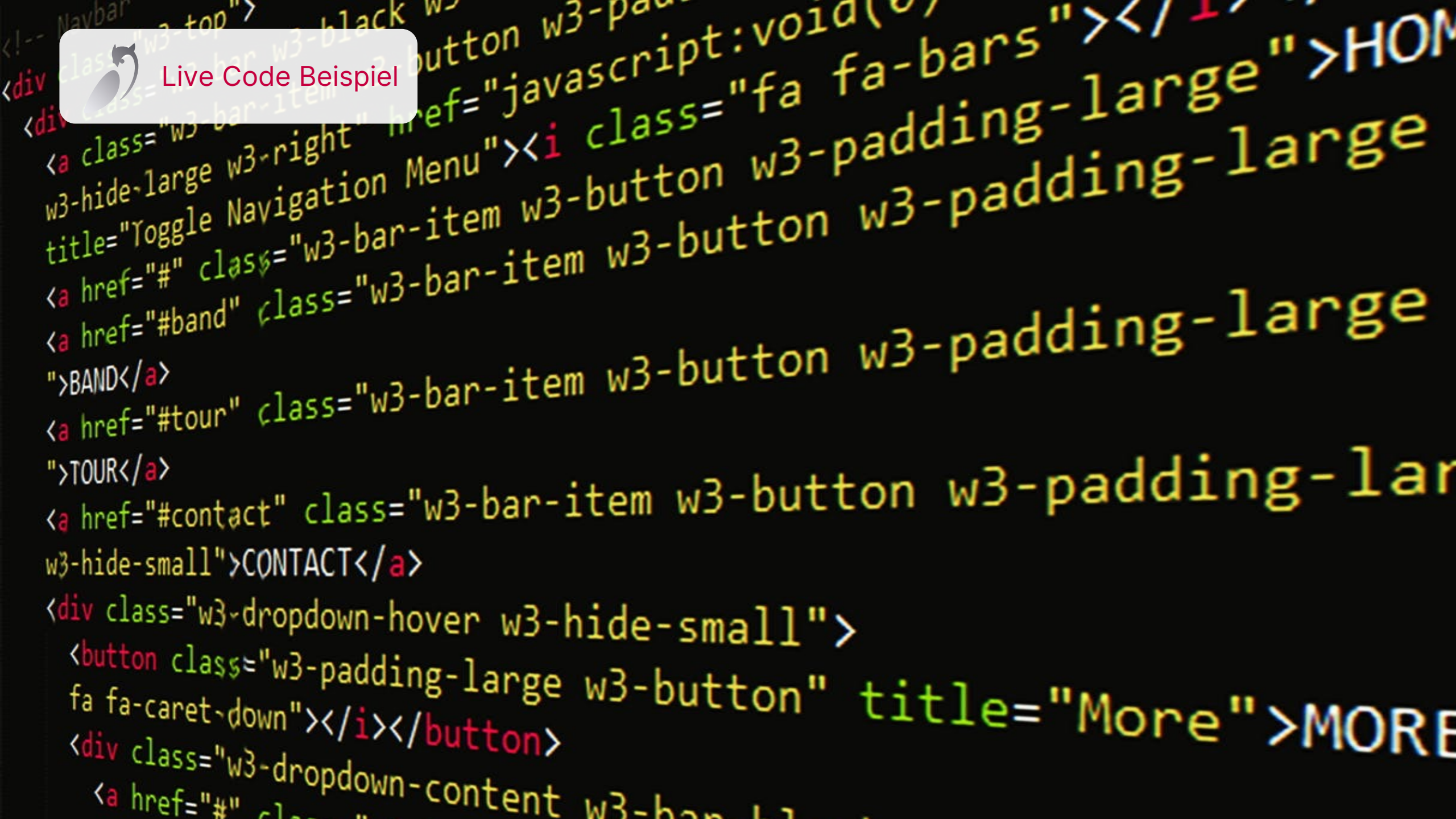
- Neuer Scope mit zwei parallelen Koroutinen

  Innerer Scope endet erst, wenn beide Koroutinen beendet wurden!

Koroutinen – Was passiert bei Fehlern?

```
fun main() = runBlocking {
    launch {
        delayedKKONWithNewJobs()
    }
    println("Willkommen zum")
}
suspend fun delayedKKONWithNewJobs() = coroutineScope {
    launch {
        delay(2000L)
        println("2024")
    }
    launch {
        delay(1000L)
        throw RuntimeException("Bang!")
    }
    println(“MATHEMA")
}
```

- Unbehandelte Fehler führen zum Abbruch des Scopes und aller enthaltenen Koroutinen (=> Strukturierte Nebenläufigkeit)


Live Code Beispiel
<!-- Navbar
<div
<a class="
w3-hide-large w3-right
title="Toggle Navigation Menu"><i class="fa fa-bars
<a href="#" class="w3-bar-item w3-button w3-padding-large
<a href="#band" class="w3-bar-item w3-button w3-padding-large
">BAND
<a href="#tour" class="w3-bar-item w3-button w3-padding-large
">TOUR
<a href="#contact" class="w3-bar-item w3-button w3-padding-lar
w3-hide-small">CONTACT

<button class="w3-padding-large w3-button" title="More">MORE
fa fa-caret-down"></i></button>
<div class="w3-dropdown-content
<a href="#"

Alles im Fluss

Ergebnisse nebenläufig Erzeugen mit asynchronen Flows

```
fun main() = runBlocking {
    println("Teilnehmer:")
    launch {
       // Start paralleler Verarbeitungen
    }
    participantFlow().collect { println(it) }
}

fun participantFlow() = flow {
    participantDatabase.forEach {
        delay(1000L) // aufwendige Datenverarbeitung
        emit(it) // Ausgabe des naechsten Wertes
    }
}
```

- `flow` erzeugt eine asynchronen Flow

  Der *Flow Builder* ist **keine** *Suspending Function*

  Ein *Flow* kann *Suspending Functions* aufrufen

  Werte werden **einzeln** durch `emit()` zurückgegeben und mit `collect()` abgerufen

Flows sind „cold“

```
fun main() = runBlocking {
    println("Teilnehmer:")
    val partFlow = participantFlow()

    println("Flow erzeugt...")
    delay(500)
    println("...nichts passiert!")

    // Erst collect() startet den Flow!
    partFlow.collect { println(it) }
    println(“Alle aufgelistet.")
}
```

```
fun participantFlow() = flow {
    participantDatabase.forEach {
        delay(1000L)
        println(“Liefere $it")
        emit(it)
    }
}
```

- Flows sind grundsätzlich „*cold*“

  Werden nicht selbst aktiv, d.h. die Erzeugung startet den Flow noch nicht

  Erst bei Verbrauch (z.B. mit `collect()`) wird der Flow abgearbeitet

- Ausnahme: SharedFlows

Einfluss nehmen

```
fun main() = runBlocking {
    println("Teilnehmer:")
    participantFlow()
      .onStart { println("Flow startet...") }
      .map { p -> p.length }
      .take(2)
      .collect { println(it) }
}
```

```
fun participantFlow() = flow {
    participantDatabase.forEach {
        delay(1000L)
        println(“Liefere $it")
        emit(it)
    }
}
```

- Operatoren können auf gelieferte Werte oder das Verhalten des Flows Einfluss nehmen

  Intermediate Operatoren verändern das Verhalten und liefern einen neuen Flow

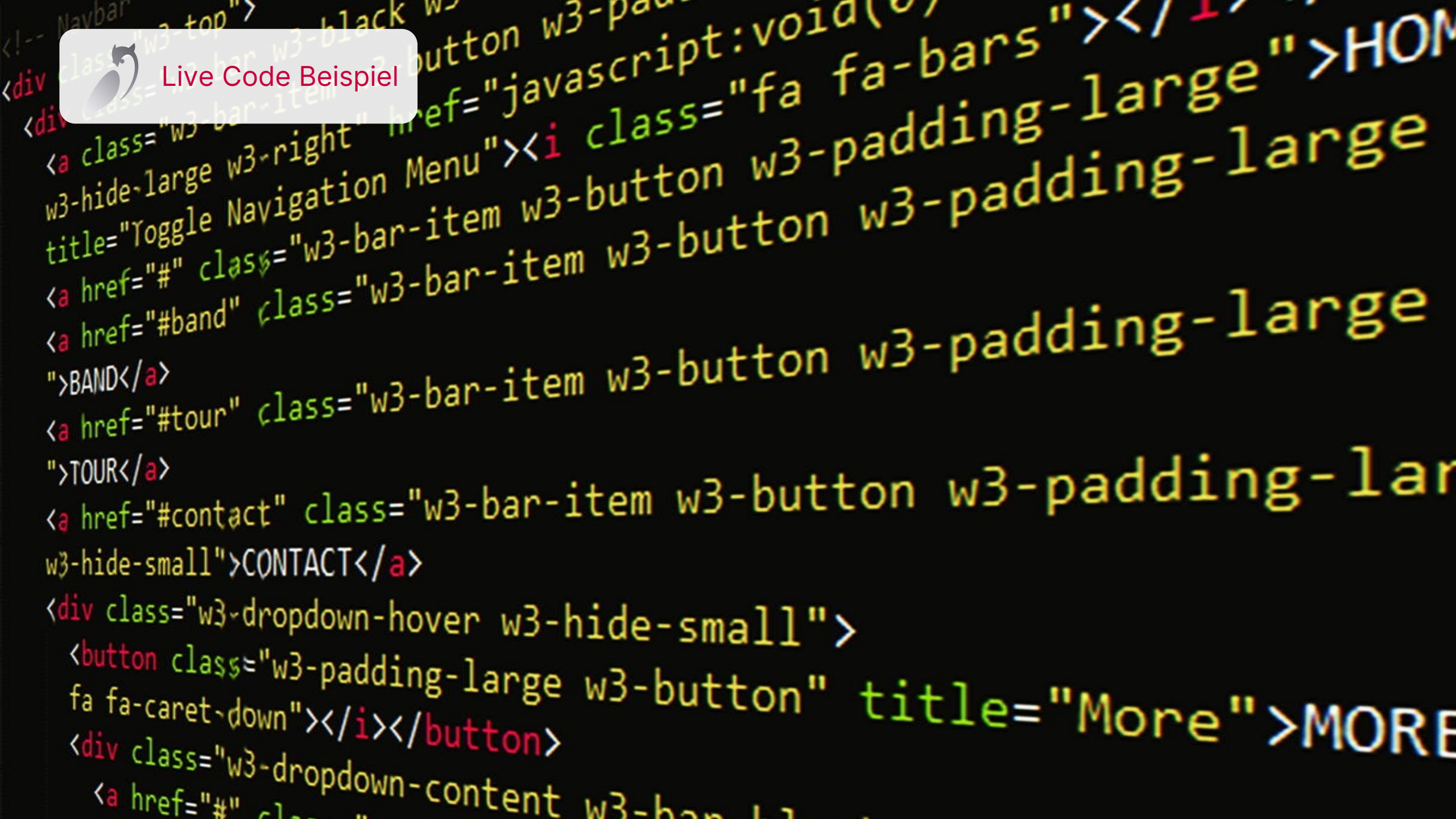
  Terminale Operatoren starten die Abarbeitung und sammeln die Ergebnisse ein

Flows nebenläufig ausführen

```
fun main() = runBlocking {
    println("Teilnehmer:")
    participantFlow()
        .onEach { println(it) }
        .launchIn(this)
    println("Weiter geht‘s...")
}
```

```
fun participantFlow() = flow {
    participantDatabase.forEach {
        delay(1000L)
        emit(it)
    }
}
```

- Die Abarbeitung von Flows mit `collect()` blockiert den weiteren Ablauf des Aufrufers
  - Auslagerung in eigene Koroutine ermöglicht eine nebenläufige Abarbeitung
  - Verarbeitung muss in `onEach()` ausgelagert werden


Live Code Beispiel
<!-- Navbar
<div
<a class="
w3-hide-large w3-right
title="Toggle Navigation Menu"><i class="fa fa-bars
<a href="#" class="w3-bar-item w3-button w3-padding-large
<a href="#band" class="w3-bar-item w3-button w3-padding-large
">BAND
<a href="#tour" class="w3-bar-item w3-button w3-padding-large
">TOUR
<a href="#contact" class="w3-bar-item w3-button w3-padding-lar
w3-hide-small">CONTACT

<button class="w3-padding-large w3-button" title="More">MORE
fa fa-caret-down"></i></button>
<div class="w3-dropdown-content
<a href="#"

Flows kombinieren

```
fun main() = runBlocking {
    println("Teilnehmer:")
    positionFlow()
        .zip(participantFlow())
            { pos, part -> "$pos. $part“ }
    .collect { println(it) }
}
```

```
fun participantFlow() = flow {
    participantDatabase.forEach {
        delay(1000L)
        emit(it)
    }
}

fun positionFlow() = (1..4).asFlow()
```

- Flows können zu einem neuen, kombinierten Flow verbunden werden

  Regel für die Bildung der Werte des neuen Flows muss dabei mit angegeben werden

Exceptions

```
fun main() = runBlocking {
    println("Teilnehmer:")
    launch {
       // Start paralleler Verarbeitungen
    }
    participantFlow().collect { println(it) }
}
```

```
fun participantFlow() = flow {
    participantDatabase.forEach {
        delay(1000L)
        emit(it)
        check(it.length < 5)
            { "BANG!" }
    }
}
```

- Exceptions während der Flow Abarbeitung
  - Brechen die Abarbeitung ab
  - Werden entlang des Flows propagiert
  - Beenden ohne Behandlung den umgebenden Coroutine Scope!

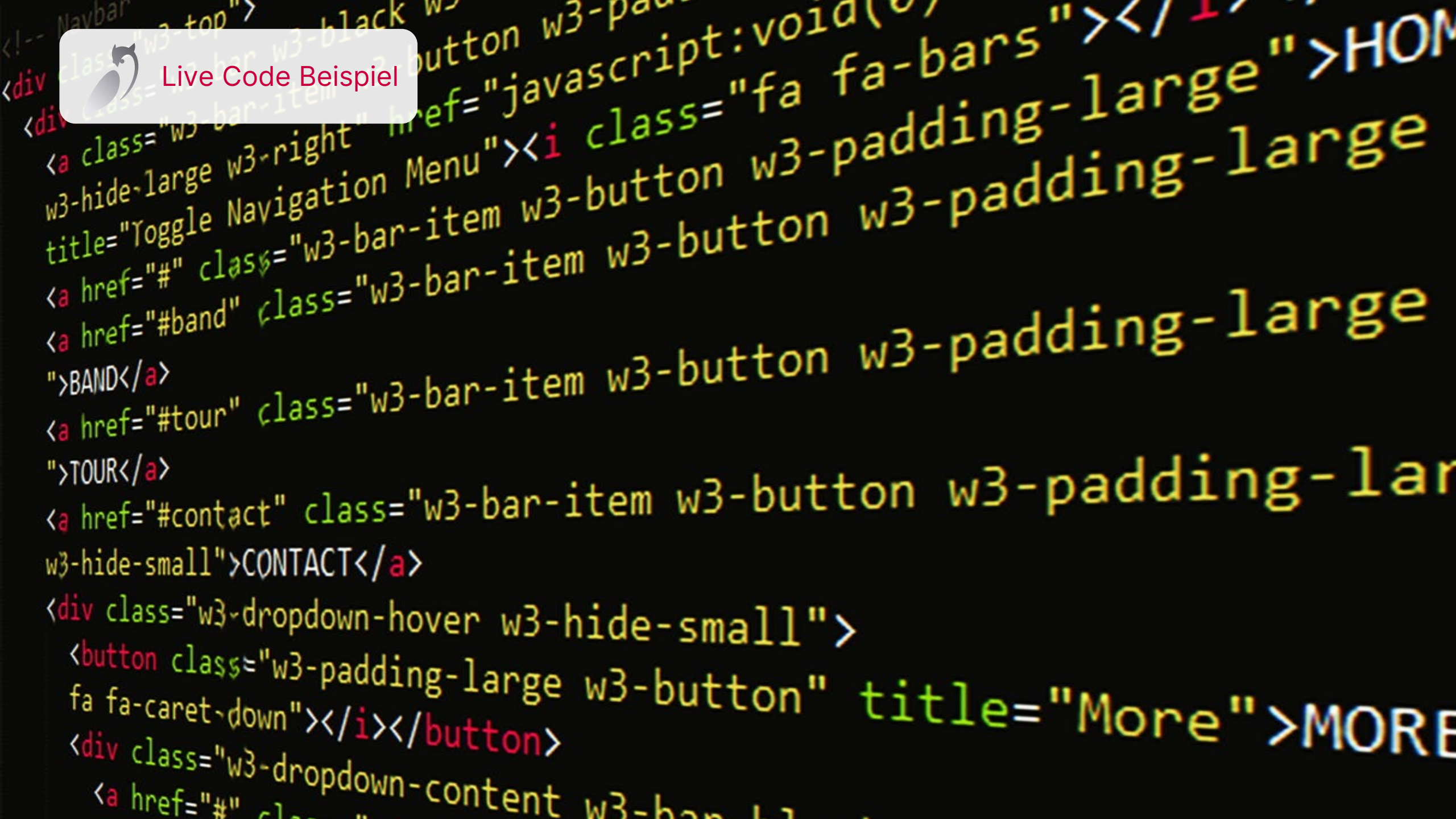
Auf Fehler reagieren

```
fun main() = runBlocking {
    println("Teilnehmer:")
    launch {
       // Start paralleler Verarbeitungen
    }

    try {
        participantFlow()
            .collect { println(it) }
    } catch (e : Exception) {
        println("Ups... '$e'")
    }
}
```

```
fun main() = runBlocking {
    println("Teilnehmer:")
    launch {
       // Start paralleler Verarbeitungen
    }

    participantFlow()
        .retry (2)
           { e -> (e is IllegalStateException)
             .also { println("Retrying...") } }
        .catch { e -> println("Ups... '$e'")}
        .collect { println(it) }
}
```

- Exceptionhandling kann imperativ oder deklarativ erfolgen
- Deklarativ auch automatisches Retry ist möglich

  Kann abhängig vom jeweiligen Fehler erfolgen


Live Code Beispiel
<!-- Navbar
<div
<a class="
w3-hide-large w3-right
title="Toggle Navigation Menu"><i class="fa fa-bars
<a href="#" class="w3-bar-item w3-button w3-padding-large
<a href="#band" class="w3-bar-item w3-button w3-padding-large
">BAND
<a href="#tour" class="w3-bar-item w3-button w3-padding-large
">TOUR
<a href="#contact" class="w3-bar-item w3-button w3-padding-lar
w3-hide-small">CONTACT

<button class="w3-padding-large w3-button" title="More">MORE
fa fa-caret-down"></i></button>
<div class="w3-dropdown-content
<a href="#"

Verarbeitung abbrechen – Suspending Flows

```
fun main() = runBlocking {
    println("Teilnehmer:")
    withTimeout(350) {
        participantFlow()
          .collect { println(it) }
    }
}
```

```
fun participantFlow() = flow {
    participantDatabase.forEach {
        delay(1000L)
        emit(it)
    }
}
```

- Abarbeitung des Flows kann mit Timeout versehen werden
  - Abbruch nach Ablauf
  - Benötigt Unterstützung durch die verwendeten Suspend Functions
- Auch explizites `cancel` möglich
  - Siehe nächstes Beispiel

Verarbeitung abbrechen – Busy Flows

```
fun main() = runBlocking {
    println("Teilnehmer:")

    participantFlow()
      .collect {
        if( it.length > 4 ) cancel()
        println(it)
      }

    positionFlow()
      .cancellable()
      .collect {
        if ( it > 2) cancel()
        println(it)
    }
}
```

```
fun participantFlow() = flow {
    participantDatabase.forEach {
        emit(it)
    }
}

fun positionFlow() = (1..4).asFlow()
```

- Abbruch bei Busy Flows nicht garantiert!

  Flow Builder `flow{}` unterstützt Cancelation unabhängig von Suspensions

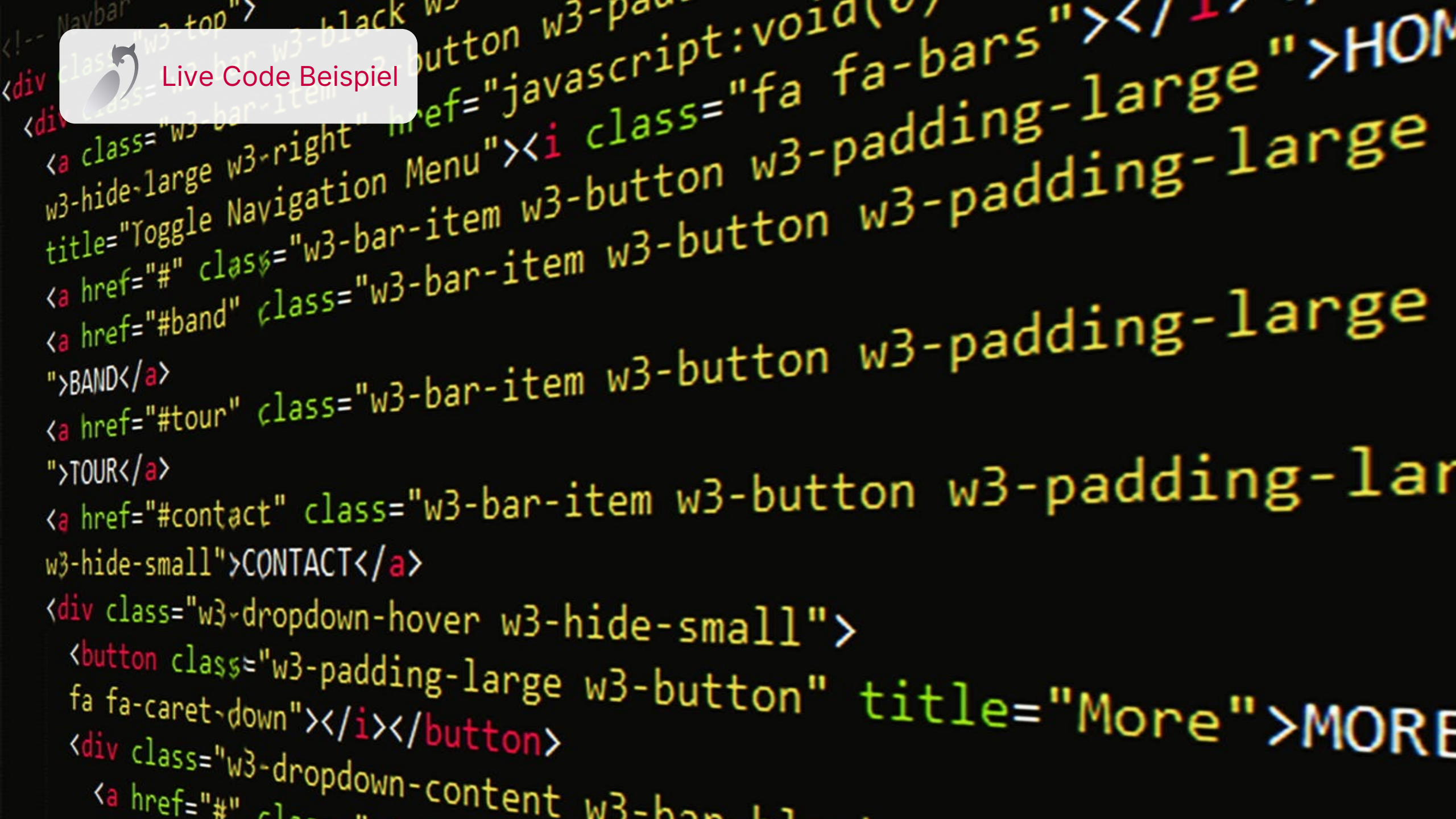
  Bei anderen Flow Buildern explizite Deklaration via `cancellable()` nötig

Am Ende des Flusses

```
fun main() = runBlocking {
    println("Teilnehmer:")

    try {
        participantFlow()
            .collect {
               println(it)
             }
    } finally {
        println(“Fertig.")
    }
}
```

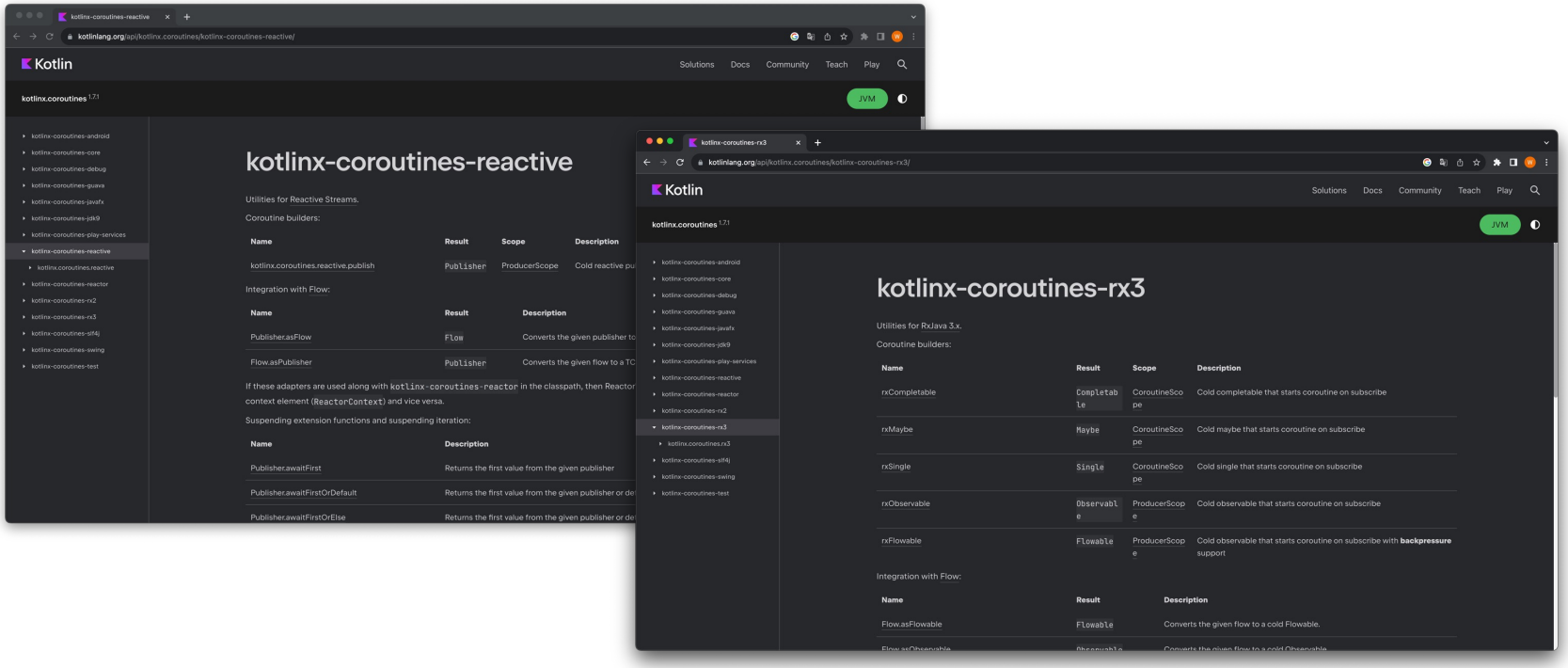
```
fun main() = runBlocking {
    println("Teilnehmer:")

    participantFlow()
        .onCompletion {
            println("Flow ${ if(it == null)
                                "erfolgreich."
                            else
                                "fehlerhaft!"}")
         }
        .catch { e -> println("Ups... '$e'")}
        .collect { println(it) }
}
```

- Auf das Ende der Abarbeitung kann ebenfalls imperativ oder deklarativ reagiert werden
  - Deklarative Lösung ermöglicht implizite Prüfung auf Fehler


Live Code Beispiel
<!-- Navbar
<div
<a class="
w3-hide-large w3-right
title="Toggle Navigation Menu"><i class="fa fa-bars
<a href="#" class="w3-bar-item w3-button w3-padding-large
<a href="#band" class="w3-bar-item w3-button w3-padding-large
">BAND
<a href="#tour" class="w3-bar-item w3-button w3-padding-large
">TOUR
<a href="#contact" class="w3-bar-item w3-button w3-padding-lar
w3-hide-small">CONTACT

<button class="w3-padding-large w3-button" title="More">MORE
fa fa-caret-down"></i></button>
<div class="w3-dropdown-content
<a href="#"

Flows und reaktive Programmierung



- Flows sind grundsätzlich reaktive Streams
  - Umwandlung in bzw. aus Typen bekannter reaktiver Frameworks möglich

Alles fließt?

- Einfaches, eingängiges Programmiermodell
  - Koroutinen „unter der Haube“
- Gut zur Auslagerung „teurer“ Operationen
  - Nicht nur zur Datenlieferung
- Strukturierte Nebenläufigkeit durch Aufsatz auf Koroutinen
  - Nutzung der Vorteile bei reduzierter Komplexität
- Integration in bestehende Frameworks möglich

Vielen Dank! Fragen?

Werner Eberling

E-Mail: werner.eberling@mathema.de
Mastodon: @Wer_Eb@social.dev-wiki.de
X / Twitter: @Wer_Eb

Beispiele: https://github.com/wern/kotlin-flow-samples

www.mathema.de

Beispiele?
Scan me ;)