DhiWise

Design Converter

Education

Understanding withCheckedThrowingCo A Developer's Guide

Saurabh Patel
Software Development Executive - III

Last updated on Aug 13, 2024

Swift's approach to handling asynchronous code has evolved significantly with the introduction of async and await. One of the key mechanisms for bridging between traditional callback-based APIs and Swift's modern concurrency model is withCheckedThrowingContinuation. This function is crucial for developers working with asynchronous code and aims to simplify the process of integrating callback-based functions with Swift's async/await syntax.

Overview of Continuations in Swift

To understand withCheckedThrowingContinuation, it’s essential to grasp the concept of continuations. In Swift, a continuation represents a point in the code where execution can be suspended and later resumed. Continuations are fundamental when converting completion handlers to async functions.

The withCheckedThrowingContinuation function allows you to create a checked continuation that can handle throwing errors. This is particularly useful for integrating callback-based APIs with async/await. When you use this function, you effectively wrap a callback-based API in an async function, enabling cleaner, more readable code and easier error handling.

The Basics of Continuations

A continuation is a mechanism that allows asynchronous code to pause and resume. With withCheckedThrowingContinuation, you get a continuation object that you can resume either with a value or an error. This method ensures that your asynchronous code integrates seamlessly with Swift's async/await model.

For example, consider a completion handler-based API:

```swift
func fetchData(completion: @escaping (Result<Data, Err
    // Asynchronous operation
}
```

You can use withCheckedThrowingContinuation to convert this into an async function:

```swift
func fetchData() async throws -> Data {
    return try await withCheckedThrowingContinuation {
        fetchData { result in
            switch result {
            case .success(let data):
                continuation.resume(returning: data)
            case .failure(let error):
                continuation.resume(throwing: error)
            }
        }
    }
}
```

Purpose and Use Cases

The primary purpose of withCheckedThrowingContinuation is to provide a bridge between traditional callback-based APIs and the async/await model. It's especially useful when you need to handle errors in asynchronous tasks or when you're working with APIs that use completion handlers.

By using withCheckedThrowingContinuation, you can:

• **Simplify Code:** Convert complex completion handler-based code into cleaner async functions.

• **Improve Error Handling:** Handle thrown errors more effectively within async functions.

• **Ensure Correct Execution:** The checked continuation helps in avoiding common pitfalls, such as forgetting to call resume, which could result in a task remaining suspended indefinitely.

How withCheckedThrowingContinuation Works

Understanding how withCheckedThrowingContinuation operates is crucial for leveraging its capabilities effectively in your Swift code. This function is designed to facilitate the conversion of completion handler-based APIs into async/await-based code, enhancing both readability and error management.

The Basics of withCheckedThrowingContinuation

withCheckedThrowingContinuation is a function that creates a continuation object to interact with asynchronous code. It allows you to pause the execution of an async function until a completion handler-based API provides a result.

Here’s how it generally works:

1. **Define the Async Function:** Wrap the callback-based API within withCheckedThrowingContinuation.

2. **Call the API:** Execute the API within the continuation block.

3. **Resume the Continuation:** Once the API call completes, resume the continuation either with a result or an error.

This method ensures that you handle the API's result or error in a way that's compatible with Swift’s async/await syntax, while also providing runtime checks to prevent common issues such as forgetting to call **resume**.

Key Components and Parameters

When using withCheckedThrowingContinuation, several key components and parameters are involved:

• **Continuation Object:** This is the main object you work with inside the continuation block. You use it to either resume the async function with a result or throw an error.

• **Completion Handler:** The API you’re wrapping typically uses a completion handler to return results. You need to pass this completion handler to the continuation to receive the data.

• **Resume Methods:** The **resume(returning:)** method is used to provide a result, while **resume(throwing:)** is used to pass an error. The continuation must be resumed exactly once to avoid runtime errors.

Here’s an example:

```swift
func fetchData() async throws -> Data {
    return try await withCheckedThrowingContinuation {
        fetchDataAPI { result in
            switch result {
            case .success(let data):
                continuation.resume(returning: data)
            case .failure(let error):
                continuation.resume(throwing: error)
            }
        }
    }
}
```

In this example, fetchDataAPI is the callback-based API. The continuation is resumed based on whether the API returns a success or failure.

Comparing with Other Continuations

Swift provides different types of continuations, such as withUnsafeThrowingContinuation and withCheckedContinuation. Understanding the differences can help you choose the right one for your needs:

• **withCheckedThrowingContinuation**: Provides runtime checks to ensure that the continuation is resumed exactly once. This helps prevent common errors like dropping the continuation or resuming it multiple times.

• **withUnsafeThrowingContinuation**: Does not include these runtime checks. It’s used when you have more control over the continuation’s lifecycle and can ensure that it is resumed correctly.

• **withCheckedContinuation**: Similar to withCheckedThrowingContinuation, but used when you don’t need to handle errors. It’s suitable for cases where only success values are returned.

Choosing the appropriate continuation depends on your specific needs and the type of API you're dealing with. For most scenarios involving error handling, withCheckedThrowingContinuation is the recommended choice due to its robust error-checking capabilities.

Practical Examples

To truly grasp the benefits of withCheckedThrowingContinuation, it's useful to see it in action through practical examples. This section will cover how to convert completion handlers, handle errors in asynchronous code, and integrate with existing APIs.

Converting Completion Handlers

Completion handlers are a common pattern in asynchronous programming, but they can be cumbersome to work with. withCheckedThrowingContinuation simplifies this by allowing you to use async/await syntax instead.

For example, consider an API that uses a completion handler to fetch user data:

```
func fetchUserData(completion: @escaping (Result<User,
    // Simulate asynchronous operation
}
```

You can convert this to use withCheckedThrowingContinuation as follows:

```
func fetchUserData() async throws -> User {
    return try await withCheckedThrowingContinuation {
        fetchUserData { result in
            switch result {
            case .success(let user):
```

```swift
                continuation.resume(returning: user)
            case .failure(let error):
                continuation.resume(throwing: error)
            }
        }
    }
}
```

In this example, the completion handler is wrapped within withCheckedThrowingContinuation. The **resume(returning:)** method resumes the task with the user data, while **resume(throwing:)** handles any errors.

Error Handling in Asynchronous Code

Handling errors in asynchronous code can be challenging, especially when working with completion handlers. withCheckedThrowingContinuation provides a straightforward way to manage errors within async functions.

Consider an API that fetches data and might throw an error:

```swift
func fetchData(completion: @escaping (Result<Data, Err
    // Simulate asynchronous operation
}
```

To handle errors, use withCheckedThrowingContinuation to transform this API into an async function that throws errors:

```swift
func fetchData() async throws -> Data {
    return try await withCheckedThrowingContinuation {
        fetchData { result in
            switch result {
            case .success(let data):
                continuation.resume(returning: data)
            case .failure(let error):
                continuation.resume(throwing: error)
            }
        }
    }
}
```

Here, any error that occurs in fetchData is propagated through the async function, making error handling more manageable and consistent with Swift's async/await model.

Integrating with Existing APIs

Integrating withCheckedThrowingContinuation with existing APIs is straightforward and enhances compatibility with modern Swift concurrency. For instance, if you have a legacy API that uses completion handlers but you want to use it within a new async function, withCheckedThrowingContinuation is the bridge you need.

Consider a legacy API for file reading:

```
func readFile(fileName: String, completion: @escaping
    // Simulate asynchronous file read
}
```

You can integrate this into an async function like this:

```
func readFile(fileName: String) async throws -> String
    return try await withCheckedThrowingContinuation {
        readFile(fileName: fileName) { result in
            switch result {
            case .success(let content):
                continuation.resume(returning: content
            case .failure(let error):
                continuation.resume(throwing: error)
            }
        }
    }
}
```

By wrapping readFile with withCheckedThrowingContinuation, you make it compatible with Swift's async/await syntax, allowing for cleaner and more readable code.

Best Practices and Considerations

When using withCheckedThrowingContinuation, adhering to best practices ensures that your asynchronous code is both robust and efficient. This section covers common pitfalls to avoid, performance implications to consider, and tips for debugging and testing.

Avoiding Common Pitfalls

Using withCheckedThrowingContinuation can be straightforward, but several common pitfalls might arise. Here's how to avoid them:

1. **Ensure Proper Resumption:** Always resume the continuation exactly once. Failing to call **resume(returning:)** or **resume(throwing:)** can cause the task to remain suspended indefinitely, leading to unexpected behavior.

```
func fetchData() async throws -> Data {
    return try await withCheckedThrowingContinuation {
        fetchDataAPI { result in
            switch result {
            case .success(let data):
                continuation.resume(returning: data)
            case .failure(let error):
                continuation.resume(throwing: error)
            }
        }
    }
}
```

In the example above, ensure that **continuation.resume** is called exactly once.

2. **Handle Errors Appropriately:** If the underlying API might throw errors, ensure you handle them correctly by resuming the continuation with an error. This prevents unhandled exceptions and ensures the async function behaves as expected.

3. **Avoid Multiple Resumptions:** Attempting to resume a continuation more than once can result in runtime errors. Ensure your logic only resumes the continuation a single time.

4. **Check for Swift Task Continuation Misuse:** Be cautious of misuse patterns, such as resuming the continuation in multiple places or failing to resume it altogether. These issues can lead to difficult-to-debug problems.

Performance Implications

While withCheckedThrowingContinuation offers numerous benefits, it's important to consider its performance implications:

1. **Overhead of Continuations:** Each call to withCheckedThrowingContinuation introduces a small amount of overhead due to the need for runtime checks and managing the continuation state. In most cases, this overhead is minimal, but it's worth being aware of in performance-critical applications.

2. **Avoiding Unnecessary Conversions:** Don't use withCheckedThrowingContinuation unnecessarily. Only convert completion handler-based APIs that benefit from async/await conversion. Overusing it in performance-critical sections of code can impact overall efficiency.

3. **Efficiency in Asynchronous Tasks:** When working with high-throughput asynchronous tasks, ensure that the use of continuations doesn't introduce significant latency or bottlenecks. Optimize where possible to balance readability and performance.

Debugging and Testing Tips

Effective debugging and testing are crucial for ensuring that your use of withCheckedThrowingContinuation is correct and efficient:

1. **Use Xcode's Debugging Tools:** Leverage Xcode's debugging tools to step through your async code and verify that continuations are resumed properly. Pay attention to potential issues like unhandled exceptions or incorrect continuation states.

2. **Write Unit Tests:** Create unit tests to verify that your async functions handle both success and failure cases correctly. Use XCTest to assert that the continuation is resumed with the expected values or errors.

```
func testFetchDataSuccess() async throws {
    let data = try await fetchData()
    XCTAssertNotNil(data)
}

func testFetchDataFailure() async throws {
    do {
        _ = try await fetchData()
        XCTFail("Expected error, but succeeded.")
    } catch {
        XCTAssertEqual(error as? MyError, MyError.expe
    }
}
```

3. **Check for Common Issues:** Be on the lookout for issues such as forgetting to call resume, resuming the continuation multiple times, or errors not being thrown correctly. These issues can lead to subtle bugs that are difficult to diagnose.

4. **Review Runtime Warnings:** Pay attention to any runtime warnings or errors related to continuations. Swift's runtime checks are designed to catch common issues, and addressing

these warnings can prevent runtime crashes and ensure correct behavior.

Conclusion

withCheckedThrowingContinuation simplifies the integration of callback-based APIs with Swift's async/await model. By wrapping completion handler-based functions, you achieve cleaner, more readable asynchronous code and robust error handling.

Key Points:

• **Enhanced Readability:** Converts complex completion handler-based APIs into straightforward async functions.

• **Effective Error Handling:** Ensures proper management of errors in asynchronous code.

• **Best Practices:** Always resume the continuation exactly once, handle errors appropriately, and be mindful of performance impacts.

• **Debugging and Testing:** Utilize Xcode's tools and unit tests to ensure correct behavior and catch potential issues.

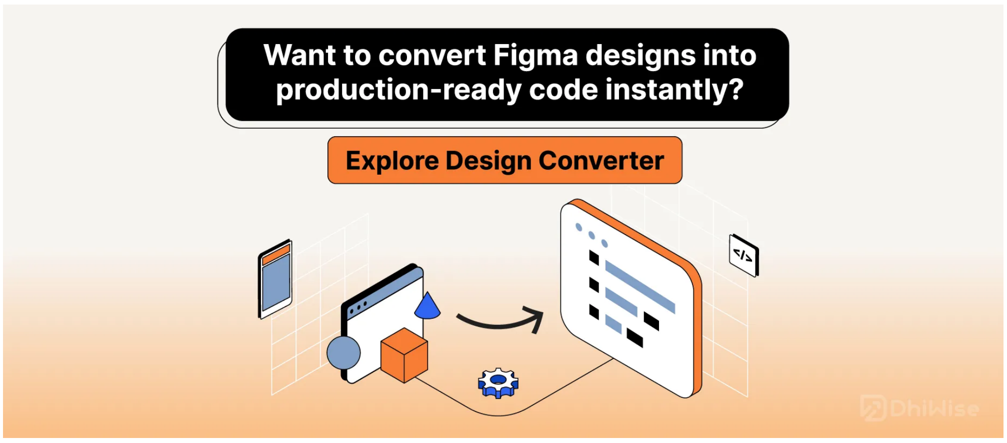
Using withCheckedThrowingContinuation helps streamline your asynchronous code, bridging traditional and modern programming practices efficiently.

Short on time? Speed things up with DhiWise!

Tired of manually designing screens, coding on weekends, and technical debt? Let DhiWise handle it for you!

You can build an e-commerce store, healthcare app, portfolio, blogging website, social media or admin panel right away. Use our library of 40+

pre-built free templates to create your first application using DhiWise.


Want to convert Figma designs into production-ready code instantly?
Explore Design Converter
DhiWise

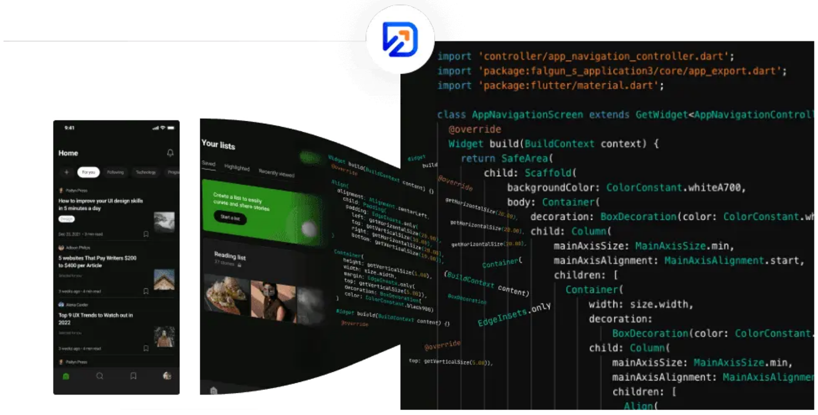
Frequently asked questions

What is withCheckedThrowingContinuation?

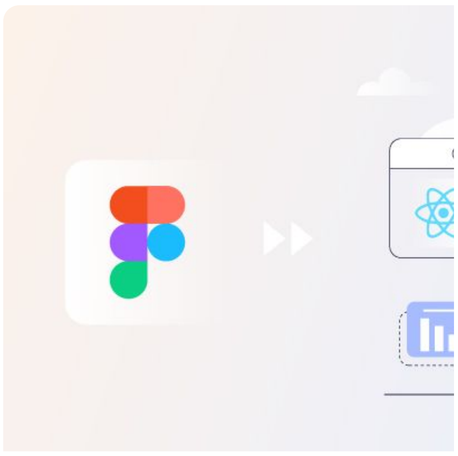
What is a checked continuation?

How to handle concurrency in Swift?

Read More



How to Convert Your Design in Figma to Flutter Code



Convert Figma to React C

Products

- Requirement Builder
- Project Planner
- Design Converter
- Coding Assistant

Company

- About Us
- Contact Us
- Career

Legal

- Terms & Conditions
- Privacy Policy

Help

Blog

Discord