KEEN-RETRY

A crate to help improve the resiliency and robustness of Rust applications

Current as of **keen-retry** 0.3.0 and Rust 1.73

About the cover art and the art in crafting resilient software

The cover of this book is a vivid allegory for the resilience and robustness provided by the `keen-retry` crate within the Rust programming ecosystem. Central to the artwork is a gear imbued with the distinctive rust hue, a nod to the Rust language itself, embodying the strength and reliability of programs built with Rust. This gear is encircled by concentric shields, etched with binary patterns that evoke the digital fortification that `keen-retry` adds to Rust applications.

Below the central gear rests an anvil, a traditional symbol of craft and durability, representing the `keen-retry` crate itself, with occasional sparks that, together with the hammer (representing the programmers), emphasize the iterative process of software development – refining and reinforcing through testing, error handling and meaningful instrumentation. The gear stands unblemished amid a tumultuous sky, streaked with lightning that represents the unpredictable challenges of crafting a resilient software that adds value to users in an elegant way. Despite the tempestuous environment, the gear's integrity is uncompromised, illustrating the "Rock Solid" stability `keen-retry` aims to provide.

In the foreground, the `keen-retry` crate's logo overlays the scene, a subtle labyrinth design at the heart of the anvil. This maze signifies the intricate journey of resilient software, highlighting the inevitability of encountering errors that are not roadblocks to the operation, but rather steps towards success. The labyrinth is intentionally simple, underscoring the crate's role in simplifying the complexity of implementing resilience while maintaining thorough instrumentation.

The inclusion of the Rust logo and the stylized Rustacean – the Rust community's beloved crab mascot – at the base of the artwork signifies the fundamental role of Rust and its vibrant community in enabling the creation of resilient systems.

The artwork, in its entirety, projects a sense of strength and confidence, with a technological edge. This same sense of strenght and confidence is what `keen-retry` aims to provide: a tool for crafting steadfast Rust applications against the backdrop of digital adversity.

The prompt offered to Dall-E

Table of Contents

Introduction

The `keen-retry` crate emerged from the need to enhance the robustness of Rust applications by simplifying the complexity of retry logic. Originally inspired by the challenges of working with bounded channels where the “buffer is full” outcome is common, the crate has evolved to address a broader spectrum of error handling and recovery scenarios across Rust applications – but always with a primary focus on performance.

The aim of `keen-retry` is not only to extend the functionality of `Result` types to encourage retryability but also to provide a clear, expressive API that can be utilized by both library authors and application developers. Both the Library API and the Application API, demonstrated in the `tests/use_cases.rs` file, forms the foundation for chaining retryable operations, ensuring that error recovery logic can be composed elegantly at any level of the application stack.

The motivation behind `keen-retry` is to deliver a simple, yet powerful library that focuses on performance and code maintainability. It offers a suite of features that enable developers to build resilient and robust software with ease. It is our hope that `keen-retry` will be as beneficial to others as it has been to us.

In this book, we will explore a variety of operations typical in client/server interactions – connect, send, receive – mirroring the procedure, consumer, and producer patterns. These operations are fundamental to understanding what sets `keen-retry` apart from similar crates. By the end, we will also compare `keen-retry` with other crates to help you decide if it's the right choice for your Rust projects.

This book, in essence, is an effort into up-bringing the `keen-retry` crate to a more mature phase: here we will discuss the architecture in depth, follow through some use cases and Design Patterns – and some Anti-Patterns – as well as presenting arguments that support the crate’s decisions. In fact, while writing this book, improvements had to be done to the crate to make it more flexible and easier to describe and use, leading to version 0.3.

A Dive Into The Problem

Understanding the Challenge of Transient Errors

Transient errors are temporary issues that can disrupt the flow of a program but may not recur if the operation is retried. These can range from network timeouts to temporary resource unavailability. The transient nature of these problems means that immediate retrying could resolve the issue without further intervention. However, not all errors should be retried - some indicate more serious issues that require different handling. The challenge lies in distinguishing between these errors, handling them appropriately, and doing so with efficiency and clarity in code.

The Complications of Retry Logic

Implementing retry logic typically involves writing repetitive code, which can clutter the logic of operations and lead to maintenance challenges. Moreover, naive retry approaches can exacerbate the problem, such as by hammering a failing service with a flood of repeated requests, leading to further instability and resource exhaustion. The need for a sophisticated retry strategy that includes backoff logic and error discrimination is critical for building resilient systems – as well as instrumenting all the relevant events, so to allow the Application to continue on its iteractive evolutionary process.

Integration and Performance Concerns

Many existing solutions for handling retries come with their own set of problems: they can be cumbersome to integrate into existing codebases, they might introduce performance overheads, or they don't support the increasingly common asynchronous operations in Rust. There's a need for a solution that's easy to integrate, has no performance impact, and supports both synchronous and asynchronous paradigms.

The Role of `keen-retry`

The `keen-retry` crate addresses these issues by offering a robust framework that allows Rust applications to elegantly handle and recover from transient errors. It differentiates between retryable and fatal outcomes and integrates seamlessly with existing codebases. With zero-cost abstractions, it ensures efficiency, and with comprehensive backoff strategies, it prevents resource exhaustion. The crate's APIs, both for libraries and applications, enable clear and concise expression of retry logic, addressing the core challenges of writing resilient software.

Theory behind the Crate

The `keen-retry` crate is underpinned by the principle of resilience in software systems. In the context of distributed environments, resilience translates to a system's capacity to gracefully handle and rebound from errors, particularly transient ones such as network glitches or temporary unavailability of services. `keen-retry` offers a structured approach to tackle these issues, enabling retryable operations under defined conditions, promoting system robustness.

Understanding Retries

Retry mechanisms are pivotal in allowing applications to re-attempt operations when faced with failures, providing an opportunity for self-recovery if underlying issues are temporary. However, discerning between errors that warrant a retry from those that don't is crucial. `keen-retry` makes this distinction clear, differentiating between transient, potentially recoverable failures and permanent ones, thereby enabling developers to craft sophisticated retry policies.

Two Types of Results

Retry mechanisms are pivotal in allowing applications to re-attempt operations when faced with failures, providing an opportunity for self-recovery if underlying issues are temporary. However, discerning between errors that warrant a retry from those that don't is crucial. `keen-retry` makes this distinction clear, differentiating between transient, potentially recoverable failures and permanent ones, thereby enabling developers to craft sophisticated retry policies.

Subsequently, `ResolvedResult` comes into play as the “Final” result after all retries have been exhausted, encapsulating the comprehensive outcomes of the retryable operations. Both maintain compatibility with Rust's `Result` type while offering richer state information for in-depth analysis. Refer to the library’s documentation for detailed insights.

Functional Approach to Error Handling

Adhering to Rust's functional conventions, `keen-retry` integrates with the language's `Result<>` and `Option<>` types, maintaining an idiomatic user experience. This approach encourages clear, maintainable code and sidesteps common issues like callback intricacies typically associated with imperative error handling.

High Order Functions are made available for `RetryResult` and `ResolvedResult`, incorporating detailed instrumentation at all stages of the error handling and recovery process.

Zero-Cost Abstractions

The `keen-retry` crate champions first-class integration, allowing methods to return `RetryResult` with seamless conversion to a standard `Result` at compile-time. This design eliminates the need to differentiate between original and retry-enabled operations, streamlining API design.

```
/// In the example bellow, `YourLibrary` uses the `keen-retry` API
```

```rust
/// and we may, simply, not opt-in for any retrying features
/// -- it will behave like a standard `Result<O, E>`
fn do_something() -> Result<(), StdErrorType> {
    let handle = YourLibrary::new(...);
    handle.retryable_method().into_result()?;
    Ok(())
}
```

Above, opting out of retry features simplifies the API, as you don't need to provide the "raw" methods – returning the standard `Result<>`. Please see the "Patterns" section for more details.

Retry Logic Composition at the Application and Library levels

`keen-retry` excels at composing sophisticated retry logic through chaining, a feature that is indispensable in systems reliant on multiple external services. This composability enables developers to construct clear and maintainable error recovery workflows that benefits from the separation of concerns needed by maintainable and complex software.

Bellow, you will see code that chains the retrying up to a reconnection, if sending a message fails:

```rust
self.send(payload)
    .retry_with_async(|payload| async move {
        if !self.is_connected() {
            if let Err(err) = self.connect_to_server().await {
                return RetryResult::Fatal {
                    input: payload,
                    error: TransportErrors::CannotReconnect {root_cause: err.into()}
                };
            }
        }
        self.send(payload)
    })
    .with_exponential_jitter(...)
    .await
```

The chaining highlighted above is done in the application level, where the `connect_to_server()` method also returns a `RetryResult`. Please refer to the "Patterns" section for chaining done at the library level – allowing the mentioned separation of concerns.

For guidelines on preventing excessive delays from nested retries and implementing timeouts, refer also to the "Anti-Patterns" section.

Backoff Strategies

Implementing retries without a strategy can lead to exacerbated problems, like overwhelming a struggling service with a flood of retries. `keen-retry` provides configurable backoff algorithms that can be employed to intelligently space out retry attempts, giving systems the breathing room needed to recover.

Getting Started

The crate's purpose

The `keen-retry` crate provides a robust framework for Rust applications to handle and recover from transient errors more effectively. It equips developers with the tools to transform standard operations into retryable workflows with a focus on performance and code clarity.

The crate distinguishes between two kinds of “Results”: one for outcomes that are not “final” (for which repeating the operation may fix the transient issue) and the other for final results, after passing through a possible retrying process.

The crate also offers a “Library API” and an “Application API”:

- **`Library API`**: Crafted for library developers, this API makes your functions retryable. It enables incorporating robust error handling with minimal changes, maintaining expressiveness and performance.
- **`Application API`**: Aimed at application developers, this API delivers deep control over error recovery processes, harnessing zero-copy and advanced instrumentation for creating sophisticated retryable workflows.

High-level overview of features

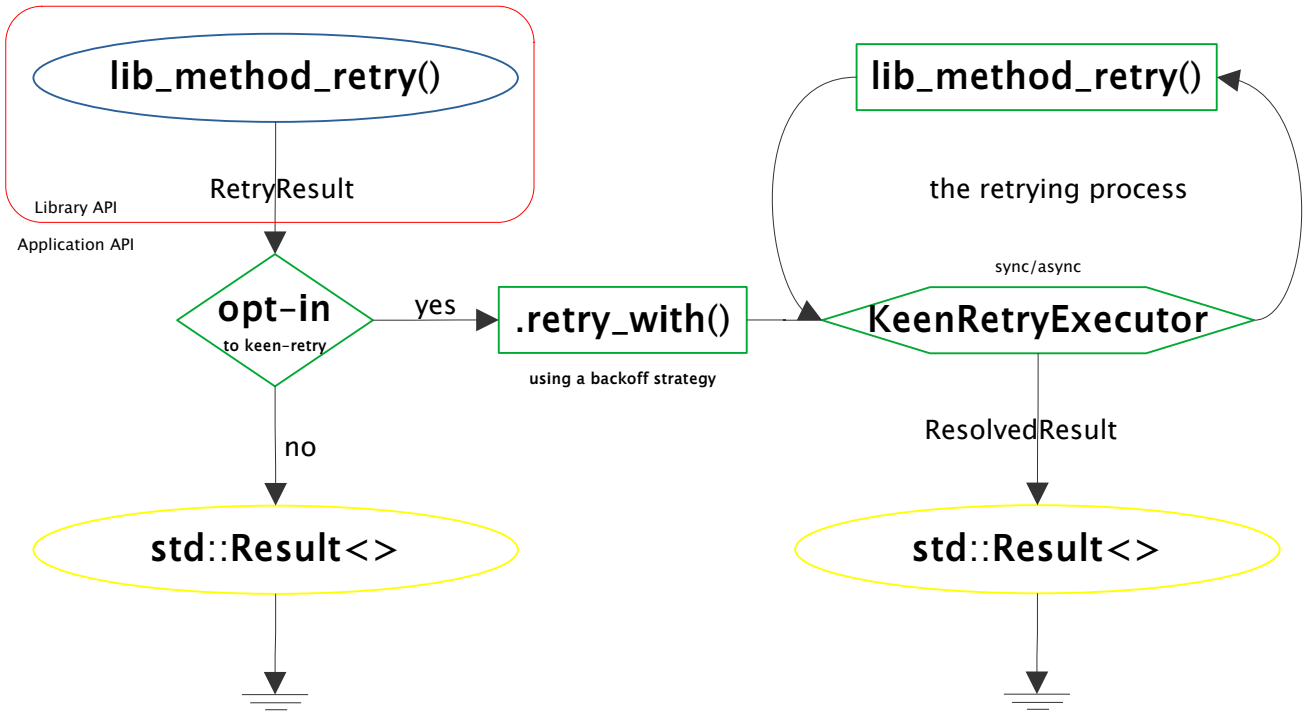
- **Retryable Result**: The “Library API” consists, primarily, of the type `RetryResult`, which enhances the standard Rust `Result` type by introducing a third variant: `Transient` (in addition to `Result::Ok` and `Result::Err`). Special care was taken, throughout the crate, to leverage zero-copy – this is particularly useful for consumers that may experience transient failures. See the “Patterns” section for more details.
- **Functional Application API**: A set of High Order Functions designed for use by the application logic, enabling detailed instrumentation and zero-copy. On the other hand, applications are free to not opt-in for retries: a `RetryResult` may be easily converted to a standard `Result` if the retrying features are not desired.
- **Zero-Cost Abstractions**: If an operation succeeds or fails fatably on the initial attempt, no extra code is executed, ensuring the usual efficiency. See the “Performance” section for more details.
- **Distinct Operations for Initial and Subsequent Attempts**: The crate allows different handling for the initial attempt and retries, providing flexibility for detailed error reporting and nested retry logic – for instance, a network `send()` might fail because the connection was dropped: the retry operation could spend some time verifying the connection and even reconnecting. See the `tests/use_cases.rs` integration test for a thorough example.
- **Sync and Async Support**: Full support for both synchronous and asynchronous operations, allowing it to fit seamlessly into various types of Rust applications. `Features` are available for turning off Tokio / async support and all related dependencies.
- **Backoff Strategies**: Configurable backoff strategies to prevent resource exhaustion during retries are available. For a thorough overview, please refer to the “Backoff” section.

- **Comprehensive "Final" Result Types**: `ResolvedResult` encapsulate all possible outcomes of retryable operations, through which advanced instrumentation can be added.
- **Ease of Integration**: The library is designed to integrate smoothly with existing Rust codebases, requiring minimal changes to adopt retry capabilities.

This comprehensive suite of features is crafted to provide a scalable and expressive error handling and recovery solution that fits naturally within the Rust ecosystem.

The keen-retry Diagram

This is the birds-eye-view of how to work with the `keen-retry` crate:



Please notice the following from the above diagram:

- `lib_method_retry()` is a method inside your library, enabling the `keen-retry` capabilities to your users by returning a `RetryResult`.
- `RetryResult` enhances the kinds of `Result<>` your library reports: they may be `Ok`, `Transient` and `Fatal` – enabling the distinction of the two nature of errors.
- On the application, developers have the choice to opt-in to the `keen-retry` features. Opting-out is as easy as calling `.into_result()`. Otherwise, call `.retry_with()` and one of the backoff strategies to get a `ResolvedResult`, which, then, may be converted back to a standard `Result<>`.

This birds-eye-view is enough to fully expose the "Retry Features" of the library. Looking in more details reveal another aspect: instrumentation.

- The `RetryResult` type may be used to log successes on the first attempt, fatal failures on the first attempt and transient failures that will be subjected to retrying. Apart from the

inspections, High Order Functions are available to enable, for instance, measuring the time spent in retrying – which is done by enriching the “input” with time measurements data. Please refer to the “Patterns” section for more details.

- The closure ingested by `.retry_with()` typically recalls the operation. This is the point where nested retryable operations may be invoked to make the transient failure go away. Additional instrumentation may be added here as well.
- Once the retrying process is over, a `ResolvedResult` is generated. With this object, the application may log additional information as the number of retries performed, the errors encountered, and, if you have enriched the input with time measurements, you may also log the time lost in the retrying process.

Understanding the Library API

The Library API centers around the `RetryResult` type, which your library functions will return to signal retryable operations. It's easy to integrate:

```
fn your_lib_method() -> RetryResult<YourSuccessType, YourErrorType> {
    // Your method's logic that may return Ok, Transient, or Fatal
}
```

Developers can then use your method with `keen-retry` like so:

```
let result = your_lib_method().retry_with(backoff_strategy);
```

This API also offers High Order Functions for adding time measurements and other metrics, enriching the data available for logging and performance monitoring.

Understanding the Application API

At the application level, `keen-retry` provides a functional API that allows for intricate control over retry logic. Developers can opt-in for retries, specify backoff strategies, and convert `RetryResult` to `ResolvedResult` for final outcomes:

```
let resolved = library_function()
    .retry_with(|input| handle_transient(input), backoff_strategy)
    .<one-of-the-backoff-strategies>(...);
```

Application developers can enrich inputs with additional data, like timestamps, to track the duration of retries, and the `ResolvedResult` type facilitates detailed logging of the retry process, including the number of attempts and the nature of encountered errors. Please refer to the “Patterns” section for instrumentation examples.

Use Cases

The `keen-retry` crate is a versatile tool designed to enhance reliability in scenarios prone to transient errors. Below are key use cases where `keen-retry` can significantly improve the resilience of your application:

Event-Driven Programming

Whether it's local event handling with channels or remote events in a microservices architecture, `keen-retry` ensures that messages are processed reliably, even in the presence of network hiccups or service interruptions.

Network Operations and Inter-Service Communication

For network requests, database transactions, or remote services interactions, transient failures are common. `keen-retry` wraps these operations in a retryable layer, applying strategies like immediate retries, exponential backoff, or custom logic tailored to the specific use case.

Resource Access

Accessing files or other system resources can fail temporarily due to temporary spikes in usage. `keen-retry` allows these operations to be retried smoothly, enhancing the stability and user experience of the system.

User-Level Tasks

When user-initiated actions – like data submissions – fail, `keen-retry` steps in. It automatically retries these tasks, reducing the frustration and manual retry attempts from the user end.

Application Start-Up

`keen-retry` is particularly useful during an application's start-up routine, ensuring that connections to external services or databases are established without hiccup, despite any transient issues.

Background Jobs and Workers

For background tasks that fail due to temporary issues, `keen-retry` can automatically re-attempt processing, minimizing manual oversight and intervention.

Custom Application API Implementations

The Application API of `keen-retry` allows developers to implement custom retry logic that conforms to the specific demands of their applications, offering granular control over how retries are managed.

Patterns

In the first part of the book, we delved into the foundational concepts and the `keen-retry` crate's basic usage. Now we proceed to explore advanced patterns and best practices for integrating `keen-retry` into your systems, ensuring that they are not only resilient but also maintainable and efficient.

As general guidelines, when using `keen-retry`, it is recommended:

For Libraries:

- Define clear conditions for when an operation should be retried and when it should fail immediately. Usually this is done by having custom error types and placing this knowledge along with them, in a `.is_fatal()` method.
- Make your methods return `RetryResult` instead of the standard `Result<>`: if you don't own the library, create an adapter type to put the new methods there.

Having the `RetryResult` methods available inside the libraries are advantageous, as other libraries may use these methods for an easy determination of the transient failures.

For Applications:

- Define the retry algorithm – possibly, by just calling the same operation again or by chaining retryable operations for complex workflows with enhanced resiliency.
- Consider one of the backoff strategies to avoid overwhelming resources.
- Use the High Order Functions to create rich, informative and meaningful instrumentation.

Bellow follow the details of the established patterns that promote resiliency and clarity.

Adding a `keen-retry` wrapper to your library

For libraries and for the operations you want to add retrying features to, wrap them with a method that converts the standard Rust's error handling to ours:

```
/// Assume `raw_broadcast()` is a function that returns a `Result`.
fn broadcast(message, targets) -> RetryConsumerResult<InputType, ErrorType> {
  match raw_broadcast(message, targets) {
    Ok(output) => RetryResult::Ok(output),
    Err(e) => match e {
      ErrorType::FailedNodes(failed_nodes) => RetryResult::Transient(failed_nodes),
      _ => RetryResult::Fatal(e),
    },
  }
}
```

This pattern shows how to convert a standard `Result` into one of the `RetryResult` types, differentiating between transient and fatal errors.

As seen, zero-cost abstractions guarantee we can use `broadcast()` as if it returned the standard `Result` type when the caller doesn't opt-in for retries: simply call `.into_result()` and the occurrence of a single "failed node" will be translated to `Result::Err`.

Backoff Strategy Configuration

After writing a `keen-retry` wrapper for your API, you may configure a simple retry & backoff strategy that defines the delay between retries and the maximum number of re-attempts:

```
// Use a simplistic linear backoff strategy for retries
let result = broadcast(message, targets)
  .retry_with(|remaining_targets| broadcast(message, remaining_argets))
  .with_delays((100...=1500).step_by(100).map(|s| Duration::from_millis(s)))
  .into_result();
```

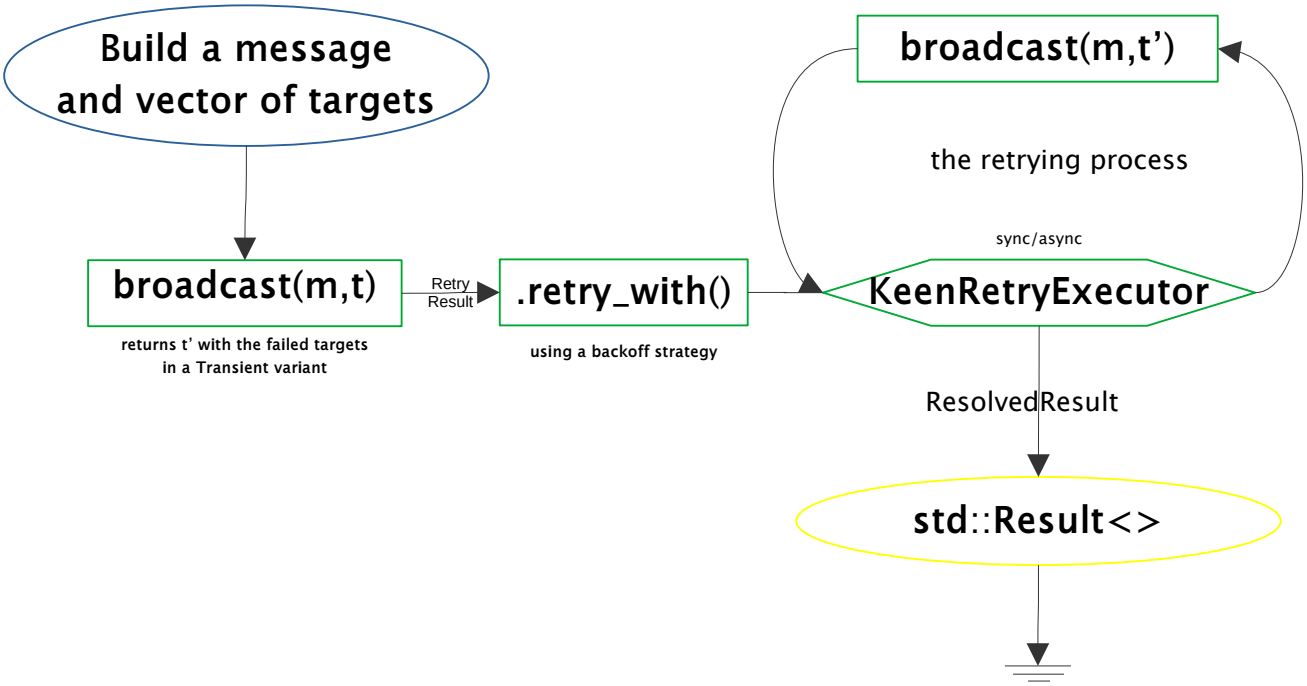
Please refer to the section “Common Backoff Algorithms and Analysis” for additional information on common strategies and other usage scenarios.

Partial Completion with Continuation Data

The “Partial Completion with Continuation Data” pattern is intended to handle operations that can be completed in stages. It's especially useful when an operation targets multiple recipients or systems and each may independently succeed or fail. The pattern ensures that partial success is captured and the remaining operation can be attempted again.

Participants and Interactions

- **Initiator**: The input or output of partially completable operations are, typically, a list of elements.
- **Recipients/Providers**: For partially completable operations that produces an output, the elements usually come from methods produce them – for instance, grouping together information from distinct sources. On the other hand, for partially completable operations that consumes a set of inputs, the caller may, for example, put a set of “targets” in a vector that a message needs to be dispatched to – Transient failures returns back the vector with the remaining elements in it, for a new attempt.



Partial Completion with Continuation Closure

This pattern is similar to the previous one – "Partial Completion with Continuation Data" – but covers the case where side effects are the expected outcome, rather than consuming caller inputs or producing outputs back to the caller. For this reason, rather than returning data on the `Transient` variant, it returns a closure.

An example would be a variation of the example above, where the equivalent for the vector of targets is not visible to the caller: instead of returning the internal library types directly, a Boxed closure is returned containing the call equivalent to `broadcast(m, v')`.

Using this pattern also enables libraries to implement their retry logic locally, preserving the "separation of concerns" by allowing the applications to skip confusing details.

Writing a `keen-retry` adapter for a third-party library

Ideally, the retryable methods would be incorporated directly into the library. So, when making a third-party library retryable, in `keen-retry` terms, it is advisable to create a new type with all the library methods you use there.

If you have a big system and the library is widely used, the best approach would be to have the adapters in their own crates. To easily distinguish those crates, lets call them "retryable-library".

When diving into the details, it is important to inspect methods that consume their inputs. These kind of methods offer a particular challenge to retrying: if they do not give the input back, on transient failures, we would either have to copy its contents before the first attempt or regenerate the input if retrying is needed, both with distinct performance impacts.

However, there isn't a specific, widely-recognized term for this programming semantic in the Rust community, even when it's a pattern that is occasionally seen in Rust APIs, such as `Crossbeam::Channels`. The idea is to allow for recovery or reuse of the inputs in the face of an error, which can be particularly important in Rust due to its ownership system.

Since this pattern ensures that resources are not lost when an error occurs in “consumer methods” and allows the caller to attempt recovery or retry, we could coin a term that reflects this functionality. Lets call it “**Recoverable Consumption**”, as it suggests that the consumption of the input is recoverable, as the input should be returned on error.

So, for creating optimal consumption operations that are retryable, keep in mind the original semantics must adhere to the “Recoverable Consumption” already, or you must force it to adhere to it (most probably by copying the input) before the first attempt. Don’t forget to hint the original library’s author about this proposed semantics, asking them to adhere to it on future versions.

With that particularity addressed, please proceed just as specified in the pattern “Adding a `keen-retry` wrapper to your library”. Notice that no special semantics are needed for methods that either only produce an output or only takes inputs by reference – they will be already performant when put in a `keen-retry` adapter.

Writing Meaningful Instrumentation for Retry Operations

The `keen-retry` crate allows you not only to inspect each state in the retry process in order to increment counters or log messages, but also offers a rich set of High Order Functions – akin to the ones in the standard `Result<>` and `Option<>` – designed specifically for manipulating the retry data, in the retry pipeline, to offer richer instrumentation.

For instance, a consumer method may have its input returned back to the retry process, but the High Order Functions offered by `RetryResult` can upgrade that type to a tuple that starts a stop-watch, if you would like to measure how much time was spent in retrying.

For an in-depth overview of the associated features, please consult the dedicated section entitled “Instrumentation and Logging”.

-- Using the powerful “composable retry policies” in the application level

-- Using the powerful “composable retry policies” in the library level

Anti-Patterns in Resilient Software Design

Introduction

In the realm of software resilience, particularly when using `keen-retry`, it's crucial to recognize and avoid certain anti-patterns that can lead to inefficiency, resource drain, or outright failures. This section aims to elucidate some of these common missteps, why they should be avoided, and how to elegantly sidestep them.

Avoid Non-Transient Retries

Rationale: Retrying operations that fail due to non-transient errors (such as logic bugs or permanent network configuration issues) is futile and can waste resources.

Elegant Solution: Implement logic to detect and classify errors accurately, distinguishing between transient and non-transient. Utilize `keen-retry`'s ability to differentiate error types and tailor retry policies accordingly.

Infinite Loops and Lack of Backoff

Rationale: Infinite retry loops, or even just those without appropriate backoff strategies, can quickly escalate into performance bottlenecks or denial-of-service incidents.

Elegant Solution: Use `keen-retry`'s built-in backoff strategies to intelligently space out retries. Establish clear conditions for ceasing retries, such as a maximum number of attempts or a cumulative timeout.

Overcomplexity in Retry Logic

Rationale: Complex retry policies can obfuscate the underlying logic, making the system harder to debug, test, and maintain.

Elegant Solution: Leverage `keen-retry`'s composability features to break down complex retry logic into simpler, reusable components. This enhances clarity and maintainability.

Library Integration Without Clear Error Handling

Rationale: Failing to delineate transient and permanent errors within a library, or creating complex retry wrappers without proper testing, can lead to unexpected behavior and difficult-to-trace bugs.

Elegant Solution: Clearly define error states and create robust integration tests, using the available counters, to ensure the library's retry mechanisms perform as expected. Document the behavior to aid consumers in understanding the intended operation.

Unit Testing when using Wrapped Libraries

Rationale: Utilizing libraries with `keen-retry` wrappers without verifying your application-specific retry and instrumentation logic can leave blind spots in error handling.

Elegant Solution: Write comprehensive unit tests that specifically test the retry capabilities and instrumentation provided by the library, but customized by the application. Ensure these tests cover various failure scenarios to validate the resilience of the retry logic.

Appropriate Backoff Strategy

Rationale: Inadequate backoff strategies, especially in distributed systems, can lead to ‘retry storms’ that exacerbate system outages rather than ameliorate them.

Elegant Solution: Employ `keen-retry`'s advanced backoff strategies that include randomization and jitter to distribute the retry load. This can prevent synchronized retries that can overwhelm the system. Consult the “Backoff” section for deeper insights.

Instrumentation of Retry Processes

Rationale: Retries without proper instrumentation can silently perform redundant work or mask underlying issues, especially when bugs are present.

Elegant Solution: Instrument retry operations to provide visibility into their behavior, using `keen-retry`'s logging and metrics facilities. This allows for proactive monitoring and debugging, particularly useful in pre-production environments. Consult the “Instrumentation and Logging” section for insights.

Nesting Retries Without Timeouts

Rationale: Nested retry operations without timeouts can lead to an explosion in wait times, as backoff intervals compound across layers.

Elegant Solution: Always use timeouts when implementing nested retries. Design these timeouts to be adaptive, ensuring that the cumulative delay is reasonable and that the system remains responsive.

The “Backoff” contain some built-in timeout facilities – specially useful when using Tokio’s Timeout is not a possibility.

By understanding and avoiding these anti-patterns, developers can harness the full potential of `keen-retry` to create resilient systems. The key lies in thoughtful implementation, strategic testing, and a deep understanding of the library's capabilities and best practices.

In addition to the `keen-retry`-specific anti-patterns already outlined, consider the following additional ones that can commonly occur when implementing retry logic in software systems in general:

Lack of Idempotency

Rationale: Retrying operations that are not idempotent can cause duplicate processing and data inconsistency, especially in distributed systems.

Elegant Solution: Ensure that operations are idempotent or that the system can handle duplicate requests gracefully. Implement mechanisms like unique transaction IDs or check tokens to prevent side effects on retries.

Misplaced Retry Logic

Rationale: Implementing retry logic too close to the core logic can lead to a mix of concerns, making the code harder to reason about and maintain.

Elegant Solution: Abstract the retry logic into a dedicated layer or component, keeping business logic separate from resilience logic. This separation of concerns facilitates cleaner architecture and easier testing.

Over-reliance on Retries

Rationale: Overusing retries as a band-aid for underlying stability issues can mask the root causes and lead to a fragile system and overloaded system.

Elegant Solution: Use retries judiciously, not abusing on the time allowed for recovery nor on the number of reattempts performed, and investigate the root causes of transient errors. Address these issues at their source to reduce the need for retries.

Ignoring Contextual Information

Rationale: Not taking into account the broader context of the operation being retried (e.g., user experience, state of the system) can lead to poor decisions about when and how to retry.

Elegant Solution: Design retry logic that is aware of context and can adjust its behavior based on the current state of the system, user needs, and other relevant factors.

Retry Logic with Static Parameters

Rationale: Using static parameters for retry logic, such as fixed delays or a set number of retries, doesn't adapt to varying system loads or error conditions.

Elegant Solution: Implement adaptive retry logic that can change parameters based on current conditions, using metrics like error rates, system load, and performance thresholds.

The “Exponential Backoff with Random Jitter” strategy takes in a closure (instead of a constant or variable) to remind of / emphasize the dynamic aspect of a good retrying strategy.

Instrumentation and Logging

Effective instrumentation and logging are crucial for the observability and reliability of applications that implement retry logic. The `keen-retry` crate offers a range of functionalities to enable detailed monitoring of retry operations. Below we explore how to instrument and log different aspects of retry processes.

The code exerpts presented here comes from `tests/use_cases.rs` – specifically the functions `keen_connect_to_server()`, `keen_send()` and `keen_receive()`. Please, refer to them for further insights offered by their comments.

Monitoring Retry Outcomes

To inform when an operation succeeds after retries or fails after giving up, use the `.inspect_recovered()` and `.inspect_given_up()` methods respectively. These methods are part of the `ResolvedResult` type and allow you to log the number of retry attempts, the duration of the retry process (if you opt-in for this), and the errors encountered.

For example, in the `keen_connect_to_server()` function, the `.inspect_recovered()` method logs a success message, including the count of retries and a list of errors that occurred before a successful connection:

```
/// Shows off a simple retry logic with a simple instrumentation, ensuring any retry
/// attempts wouldn't go on silently.\
/// This is the minimum recommended instrumentation, which would be lost after
/// downgrading the [keen_retry::ResolvedResult] to a standard `Result<>`.
pub async fn keen_connect_to_server(socket) -> Result<(), ConnectionErrors> {
    socket.connect_to_server().await
        .retry_with_async(|_| socket.connect_to_server())
        .with_exponential_jitter(keen_retry::ExponentialJitter::FromBackoffRange {
backoff_range_millis: 10..=130, re_attempts: 10, jitter_ratio: 0.1 })
        .await
        .inspect_recovered(|node, _, retry_errors_list|
            warn!("connected to {} after retrying {} times (failed attempts: [{}])",
                  node,
                  retry_errors_list.len(),
                  keen_retry::loggable_retry_errors(retry_errors_list)))
        .into_result()
}
```

Similarly, in `keen_send()`, the `.inspect_given_up()` method is used to log when the operation has failed after all retry attempts have been exhausted:

```
.inspect_given_up(
  |(_loggable_payload, payload, retry_start), retry_errors_list, fatal_error|
    error!("`keen_send({payload:?})` FAILED after exhausting all {} retrying attempts
            in {:?} with error {fatal_error:?}. Previous transient failures: [{}]",
            retry_errors_list.len(), retry_start.elapsed().unwrap_or_default(),
            keen_retry::loggable_retry_errors(retry_errors_list)))
```

The time-related `retry_start` parameter above is the result of employing the High Order mapping Functions to opt-in for time measurements, as we will see ahead.

Logging First Attempt Outcomes

The `.inspect_fatal()` method can be used to log outcomes of the first attempt. If the first attempt results in a fatal error, meaning no retries will be performed, this method provides a mechanism to log that immediate failure.

In both `keen_send()` and `keen_receive()` functions, the `.inspect_fatal()` method is employed to log fatal errors:

```
.inspect_fatal(|payload, fatal_err|
    error!("`keen_send({payload:?})`: fatal error (won't retry): {fatal_err:?}"))
```

This method is available at either `RetryResult` and `ResolvedResult` types, allowing the pipeline to be composed with this functionality either before or after the retrying process.

For simmetry, the `.inspect_ok()` methods are also available, allowing to log succesful outcomes of the first attempt – although one would refrain from using it for log levels other than `trace` or `debug`, so not to polute the logs with meaningless information.

Enriching Retry Data

To add a stopwatch for measuring the time spent in the retrying process, use the `.map_input()` method to transform the input into a tuple that includes the starting time, which can then be used to calculate the elapsed time for retries.

This pattern is demonstrated in the `keen_send()` function:

```
.map_input(|payload| ( payload, SystemTime::now() ) )
```

And then the elapsed time is calculated and logged:

```
.inspect_recovered(|(loggable_payload, duration), _output, retry_errors_list|
    warn!("`keen_send({loggable_payload})`: succeeded after retrying {} time(s) in
           {:?}. Transient failures were: [{}]",
           retry_errors_list.len(),
           duration,
           keen_retry::loggable_retry_errors(retry_errors_list)))
```

There is a small portion of the code, not shown here, to convert the `SystemTime` into a `Duration`. For details, please head to the source of the exerpts used throughout this section.

Additional Features

The `keen-retry` crate also allows for logging and handling unrecoverable errors with `.inspect_unrecoverable()`, mapping different types of input data with `.map_unrecoverable_input()`, and chaining operations with `.map_reported_input_and_output()` for more detailed control over what gets logged and returned from the retry operations.

For instance, in `keen_receive()`, unrecoverable errors after retries are logged, providing insight into the number of attempts made and the total duration before the failure:

```
.inspect_unrecoverable(|retry_start, retry_errors_list, fatal_error| {
    warn!("`keen_receive()`: fatal error after trying {} time(s) in {:?}:
           {fatal_error:?} -- prior to that fatal failure, these retry attempts
           also failed: [{}]",
          retry_errors_list.len()+1,
          retry_start.elapsed().unwrap_or_default(),
          keen_retry::loggable_retry_errors(retry_errors_list));
})
```

Performance Analysis

Introduction

In this section, we will dive into the performance characteristics of the `keen-retry` crate, demonstrating its efficiency and the zero-cost abstraction it offers. We'll examine how `keen-retry` integrates with Rust applications and compare its runtime overhead against standard `Result<>` types under various conditions.

This analysis was done based on the findings of the benchmark at `benches/zero_cost_abstractions.rs`.

Benchmarking Methodology

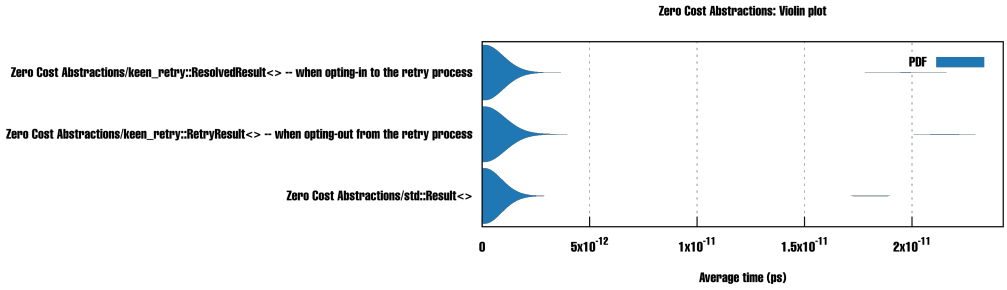
The performance benchmarks were conducted using the `criterion` crate, a powerful Rust library for setting up and running benchmarks. The focus was on comparing the execution time of operations that either succeed or fail fatally at the first attempt, as these scenarios are directly comparable to the standard Rust `Result<>`. Operations resulting in a `Transient` state were not included, as they involve additional retry logic.

The benchmarks were set up to measure the following:

1. **Raw Operation**: A simple function returning a `Result<>` type, alternating between `Ok` and `Err` states.
2. **Keen-Retry Operation**: An equivalent operation wrapped in `keen-retry`'s `RetryResult<>`, which – as mentioned – ommits the third possible outcome, `Transient`.
3. **Application-Level Operation**: A function simulating application-level retry logic, which involves instrumenting and potentially retrying the operation using `keen-retry`'s Application API.

To ensure fairness and accuracy, each benchmark was run multiple times, with the system state reset between runs to minimize the impact of caching and other external factors.

Results



The benchmarks revealed that `keen-retry` operations exhibit no overhead when compared with the standard `Result<>` in cases where operations do not involve retries. Specifically:

- **Raw Operation**: Served as the baseline for performance comparison.

- **Keen-Retry Operation**: Demonstrated no overhead when opting out of the retry feature by converting `RetryResult<>` directly to `Result<>`, which is done at compile-time.
- **Application-Level Operation**: Showed that even when opting into the retry logic, but without actual retries occurring, the overhead was zero due to compile-time optimizations.

For achieving such astonishing optimizations, annotating with `#[inline(always)]` the methods that utilize the “Library API” and “Application API” was needed.

Implications

The zero-cost abstraction principle is vital in systems programming, where every cycle counts. The `keen-retry` crate adheres to this principle by providing powerful error handling and retry capabilities without sacrificing performance. This ensures that developers do not have to compromise on efficiency when building resilient applications in Rust.

For library authors, the `keen-retry` crate offers a seamless way to enhance the robustness of their code with no performance impact. Application developers benefit from the flexibility to implement detailed retry logic tailored to their specific needs, without worrying about runtime penalties.

Conclusion

The performance analysis confirms that `keen-retry` stands up to the demands of high-performance Rust applications. By adhering to zero-cost abstraction principles, `keen-retry` ensures that developers can build upon a foundation of resilience without impacting the speed and efficiency that Rust is known for.

Backoff

Backoff refers to the practice of deliberately delaying retries after encountering failures. This delay serves several important purposes:

1. **Prevents Overloading Systems:** By introducing a delay between retries, programs avoid overwhelming overloaded systems with repeated requests. This allows the system to recover and handle the requests more effectively.
2. **Reduces Contention:** Back off helps reduce contention for resources, especially when multiple programs or services are attempting to access the same resource simultaneously. This can prevent congestion and improve overall performance.
3. **Prevents Thrashing:** Thrashing occurs when a system spends more time retrying failed operations than performing useful work. Back off helps prevent thrashing by slowing down the retry process and giving the system time to recover.
4. **Reduces Error Rates:** By delaying retries, programs can avoid repeating the same mistakes that caused the original failures. This can lead to lower error rates, fewer log warnings and more reliable operation.

For these reasons, retrying with a backoff is an essential component of resilient programming, enabling programs to handle transient failures gracefully and maintain overall system stability and performance.

Retrying Strategies without Backoff

Nonetheless, there are a few cases where not using a backoff is advantageous:

- Hard real-time systems, with local communications and low error rates. An example of this may be a single producer trying to put elements in a local queue. If the queue gets full, it may be acceptable for the producer to spin during retries, instead of sleeping.
- Services that may recover from transient errors faster than the time it takes to prepare a new retry attempt. The `keen-retry` library allows custom code to run on each retry attempt – which may, for instance, log the occurrence, compute a metric or do any other long operation, such as storing something in a database. Should these pre-retry operations take too long, backing off might just not be needed at all.

To account for these scenarios, the following methods are provided:

- `spinning_forever()`: keeps the thread busy retrying, without context-switching, but putting the CPU in the “relaxed” state, suitable for spin loops that reacts the a very low latency. Use with caution, as this method may dead-lock the thread, at 100% CPU usage, as there is no limit for the number of retries:

```
let result = produce_operation_retryable()
    .retry_with(|_| produce_operation_retryable())
    // spin-loops until success
    .spinning_forever()
    .into();
```

- `spinning_until_timeout(duration, timeout_error)`: also keeps the thread busy retrying, possibly context-switching to consult the system time – but limits the locking to the specified timeout `duration`, which, if elapsed, causes the operation to fail with `timeout_error`:

```
let result = produce_operation_retryable()
    .retry_with(|_| produce_operation_retryable())
    // spin-loops for up to the given timeout
    .spinning_until_timeout(1000ms, MyError::TimeoutError)
    .into();
```

For `async` programming, there are additional methods available:

- `yielding_forever()`: similar to `spinning_forever()` above, but let Tokio execute other tasks instead of simply looping:

```
let result = produce_operation_retryable()
    .retry_with(|_| produce_operation_retryable())
    // let Tokio run other tasks until success
    .yielding_forever()
    .into();
```

- `yielding_until_timeout(duration, timeout_error)`: similar to `spinning_until_timeout()` above, but let Tokio execute other tasks instead of simply looping:

```
let result = produce_operation_retryable()
    .retry_with(|_| produce_operation_retryable())
    // let Tokio run other tasks for up to the given timeout
    .yielding_until_timeout(1000ms, TimeoutError)
    .into();
```

Retrying Strategies with Backoff

For all other usage scenarios, a backoff strategy is recommended.

Bellow we will see some common strategies, going from the simplest to the more sophisticated ones.

Common Backoff Algorithms and Analysis

As seen, the `keen-retry` crate offers flexible backoff configurations through iterators, which specify both the amount of sleeping time between attempts as well as the number of attempts. Although the users may build their own strategies, here goes a summary & analysis of the most common ones:

Constant

This is the simplest backoff possible: simply sleep for s on each of the n attempts:

```
.with_delays((1..=n).map(|_| Duration::from_millis(s))
```

The total time spent sleeping between re-attempts is a simple multiplication:

$$\sum = n . s$$

Although simple, you must be careful when choosing this strategy, as the number of simultaneous attempts quicky scale up with the number of instances running, which may be of concern if the operation make requests to external services: overloading the resource becomes a possibility. For this reason, for large n, this strategy may only be acceptable if, at most, a few instances are running simultaneously – lets say, on a scheduled job.
On the "pros" side, this strategy offers a very predicable backoff sleeping time, which may be desired for some time-constrained scenarios.

Arithmetic Progression

This is also a simple backoff strategy, but goes an extra mile in avoiding the overload of external services by varying the sleeping time between re-attempts progressively:

```
.with_delays((100..=1000).step_by(100).map(|millis| Duration::from_millis(millis)))
```

In the above expression, the number of re-attempts, n, is not directly expressed – but it may be determined through the initial element a_1, the last element a_n, and the step s between elements with:

$$n = 1 + \frac{a_n - a_1}{s}$$

With that, the maximum time spent sleeping between re-attempts is given by sum of the arithmetic progression, through the formula:

$$\sum = \frac{n}{2}(a_1 + a_n)$$

Where:

- n is the number of terms in the progression,
- a_1 is the first term,
- a_n is the last term.

Geometric Progression

This strategy is a step towards preserving resources, as it increases the backoff time exponentially (from a growth ratio) to reduce the chances of congestion:

```
.with_delays((1..=15).map(|i| Duration::from_millis(1.5849f64.powi(i))))
```

In this example, up to 15 retries are performed. Considering the retry attempts fail immediately, the total delay may be calculated with the formula for the sum of a geometric progression:

$$\sum = \frac{a_1(1 - r^n)}{1 - r}$$

which, on our case, may be simplified to:

$$\sum = \frac{1 - r^{(n+1)}}{1 - r}$$

Where:

- **a1** is the first term of the progression.
- **r** is the common ratio between terms.
- **n** is the number of terms.

For the geometric progression in the given example, the whole retry operation may backoff for up to 2.7 seconds.

The *crèam de la crèam*: Exponential Backoff with Random Jitter

If you are using a shared resource, such as a network service, the problem known as “thundering herd problem” may arise. Imagine a scenario where a network problem cause connections to hang and multiple nodes are waiting. Suddenly, the problem is solved – meaning all hanged connections are dropped at the same time, while new ones may be accepted. If all the nodes are running the same code, whatever the backoff strategy used (from the ones we’ve seen so far) is likely to cause all the retry attempts to be done simultaneously, as all instances would be backing off for the exact same amount. This sudden surge in requests can overwhelm the resource, causing it to become unavailable or significantly slow down.

The term “thundering herd” is an analogy to a herd of animals rushing towards a water source. Just as a large herd of animals can congest the access to the water source, a large number of concurrent requests can overwhelm a resource and cause it to malfunction.

To mitigate this, we must go beyond predictable behaviors – and this is best done by progressing the backoff time exponentially with an added random jitter.

To use the jittered, exponential backoff:

```
/// backoff exponentially, from 100ms to 15 seconds in 10 re-attempts
/// with +/- 20% random variance — ideal for retrying network requests
const EXPONENTIAL_JITTER_CONFIG: keen_retry::ExponentialJitter =
    keen_retry::ExponentialJitter::FromBackoffRange {
        backoff_range_millis: 100..=15000,
        re_attempts: 10,
        jitter_ratio: 0.2,
    };
let result = produce_operation_retry()
    .retry_with(|_| produce_operation_retry())
    .with_exponential_jitter(EXPONENTIAL_JITTER_CONFIG)
    .into();
```

High available systems with low error rates may benefit from using an initial backoff of 0: if one of the nodes on a highly distributed system fail, retrying by waiting zero time may be the best approach (as failures are rare) – in this scenario, the request might be promptly picked by another node. Subsequent retries may wait progressively more, if this doesn’t hold true.

Important Staging Rust Features

In the future, the `keen-retry` crate may benefit from the following features that are awaiting their turn to make it in stable Rust:

Async `fn` in Traits

Despite we can use `Box<>` around a `Future` to simulate an "async" trait, as the `sync-trait` crate does, this rules out many code optimizations that could be done by the compiler – apart from requiring a `malloc()` on every method call. This is, currently, an unacceptable performance hit and the `keen_retry_executors` module does the next best thing: to repeat the method signatures, in order for both the sync and async executors to have the same API.

This is, obviously, not a great solution, as it impacts code maintainability, but, at least, it has no impacts in performance. Once the "Async fn in Traits" make it to stable Rust, that portion of the code can be improved. For more info, see Stabilizing async fn in traits in 2023.

The `Try` trait

Refactoring code that uses `keen-retry` enabled libraries is easy. As mentioned, our `ResolvedResult` may be very easily converted into a standard `Result` by calling `.into()` or `.into_result()`.

When stable Rust allows user types to implement the `Try` trait, that extra call won't be needed in most cases. The `Try` trait, if implemented, allows the compiler to work with the ? operator directly, so code like the following, in the application logic, could be rewritten from this:

```
let output = handle.retryable_method().into()?;
```

into this slightly simplified form:

```
let output = handle.retryable_method()?;
```

which is exactly the same as if `retryable_method()` returned a standard `Result` type.

Notice that, by now, the recommended Pattern for wrapping calls to retryable methods locally overcomes most of this trouble.

Other Crates Similar to keen-retry

...

Index

An index to quickly locate information within the booklet.