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 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, embodying the essence of `keen-retry` as a tool for crafting steadfast Rust applications against the backdrop of digital adversity.

Table of Contents

Introduction....4
A Dive Into The Problem....5
  Explanation of the crate's purpose....5
  High-level overview of features....5
Theory behind the API....7
  Understanding Retries....7
  Two Types of Results....7
  Functional Approach to Error Handling....7
  Zero-Cost Abstractions....8
  Retry Logic Composition....8
  Backoff Strategies....9
  Error Discrimination....9
  Application API and Retry Chains....9
Getting Started....10
  Understanding the Library API....11
  Understanding the Application API....11
Core Concepts....12
Use Cases....13
  Network Operations....13
  Resource Access....13
  User-Level Tasks....13
  Application Start-Up....13
  Inter-Service Communication....13
  Background Jobs and Workers....13
  Custom Application API Implementations....14
  Examples from `tests/use_cases.rs`....14
Patterns....15
  Pattern: Adding a keen-retry wrapper to your library....15
   Pattern: Backoff Strategy Configuration....16
  --- How to write a `keen-retry` wrapper for your library....16
Anti-Patterns....17
--- Some anti-patterns:....17
Advanced Features....18
Instrumentation and Logging....19
Performance Considerations....20
Backoff....21
  Retrying Strategies without Backoff....21
  Retrying Strategies with Backoff....22
  Common Backoff Algorithms and Analysis....22
    Constant....22
    Arithmetic Progression....23
    Geometric Progression....23
    The *crèam de la crèam*: Exponential Backoff with Random Jitter....24
Important Staging Rust Features....25
  Async fn in Traits....25
  The Try trait....25
Other Crates Similar to keen-retry....26
Index....27

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” scenario 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

At the heart of the `keen-retry` crate is the concept of resilience in software systems. Resilience, in this context, refers to the system's ability to handle and recover from failures gracefully. This is particularly important in distributed systems where network hiccups, transient service unavailability, or temporary resource constraints are common. The `keen-retry` library provides a structured way to approach these challenges by enabling operations to be retried under predefined conditions.

Understanding Retries

Retry patterns allow applications to automatically repeat an operation in the face of failure, giving the system a chance to recover, if the temporary underlying issue is resolved, before a fatal error is finally computed. However, not all failures should trigger a retry. The `keen-retry` crate distinguishes between transient failures that are potentially recoverable and permanent failures that are not, allowing developers to define nuanced retry policies.

Two Types of Results

Operations are expected to succeed on the first attempt, having their outputs extracted and their inputs consumed. When they don't, a first "Result" is obtained – we call it `RetryResult`: this type distinguishes between three states: `Ok`, `Transient` and `Fatal`.

`Ok` and `Fatal` are terminal states – they are ready to be converted back to a standard Rust `Result` type. The `Transient` state allows triggering a set of additional steps to attempt recovery.

When the retrying process takes place, the second "Result" – `ResolvedResult`, plays a role: it is considered a "Final" Result, as no more retries can be performed. This type can also be converted to a standard `Result` type, although it contains more nuanced states in case the application wants to inspect it: `Ok`, `Fatal`, `Recovered`, `GivenUp` and `Unrecoverable`. For more more details, please consult the library's documentation.

Functional Approach to Error Handling

`keen-retry` adopts a functional paradigm akin to Rust's own `Result<>` and `Option<>` types, ensuring a familiar and idiomatic experience for Rust developers. This design choice facilitates clean and maintainable code, avoiding common pitfalls such as "callback hell" that can arise with more imperative error-handling strategies.

A set of High Order Functions are provided to manipulate both the `RetryResult` and the `ResolvedResult` types, allowing detailed instrumentation to be incorporated in all phases of the error handling and recovery process.

Zero-Cost Abstractions

`keen-retry`-enabled methods, that you expose in your own API or library, are first class citizens: the `keen-retry` "final" result type `ResolvedResult` may be easily converted to an ordinary `Result`, so you don't need to distinguish your "original operations" from the "`keen-retry`-enabled ones" – and the conversion is done at compile time:

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

In the example above, not using the `keen-retry` result features works as treating both permanent and transient errors as hard failures – no retrying measures will be taken and `Err` will be propagated.

This simplifies your API design by not requiring the distinction of original / `keen-retry`-enabled methods: simply enable your methods to return a `RetryResult` and user code will require only a minor change. Nonetheless, this is not the recommended pattern: it is recommended that Libraries and APIs do create additional methods with the suffix "_retry()": those are the ones that should return a `RetryResult`. Please consult the "Patterns" section for more details.

Retry Logic Composition

A key feature of the `keen-retry` crate is its support for composing complex retry logic through chaining. This composability is crucial for building robust systems where operations depend on multiple, potentially flaky, services or resources. By allowing operations to be chained, `keen-retry` makes it possible to articulate clear and maintainable error recovery flows.

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

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

Notice that `connect_to_server()` may perform its own retrying process – which may lead to highly delayed retrying procedures if nesting is not done properly. Please refer to the "Anti-Patterns" section for more details regarding this phenomenon and how to enforce time-outs.

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.

Error Discrimination

The `keen-retry` API enables operations to signal whether a failure is transient or fatal. This distinction is crucial for avoiding unnecessary retries in situations where they would be futile, such as with authentication failures or other permanent issues.

This requires that Libraries are able to distinguish between these classes of errors. So, before the integration effort, maybe refactorings have to be made.

Application API and Retry Chains

The key feature of `keen-retry` is its Application API, that empowers developers to implement retry logic at the application level. This extends the resilience capabilities all the way from low-level libraries to the final user-facing applications, creating a cohesive retry mechanism throughout the stack.

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`**: designed to be used by libraries that want to incorporate error handling & recovery (or “retrying”, for short) into their features. By wrapping your library or API with `keen-retry`, operations can be made retryable with minimal overhead and maximum expressiveness.
- **`Application API`**: designed to be exposed down to the application level, providing a functional API to allow deep control of the handling and recovery of errors, leveraging zero-copy and advanced instrumentation. This is foundational to the chaining of retryable operations, allowing developers to create complex, resilient workflows that gracefully handle transient failures in simple or complex scenarios.

High-level overview of features

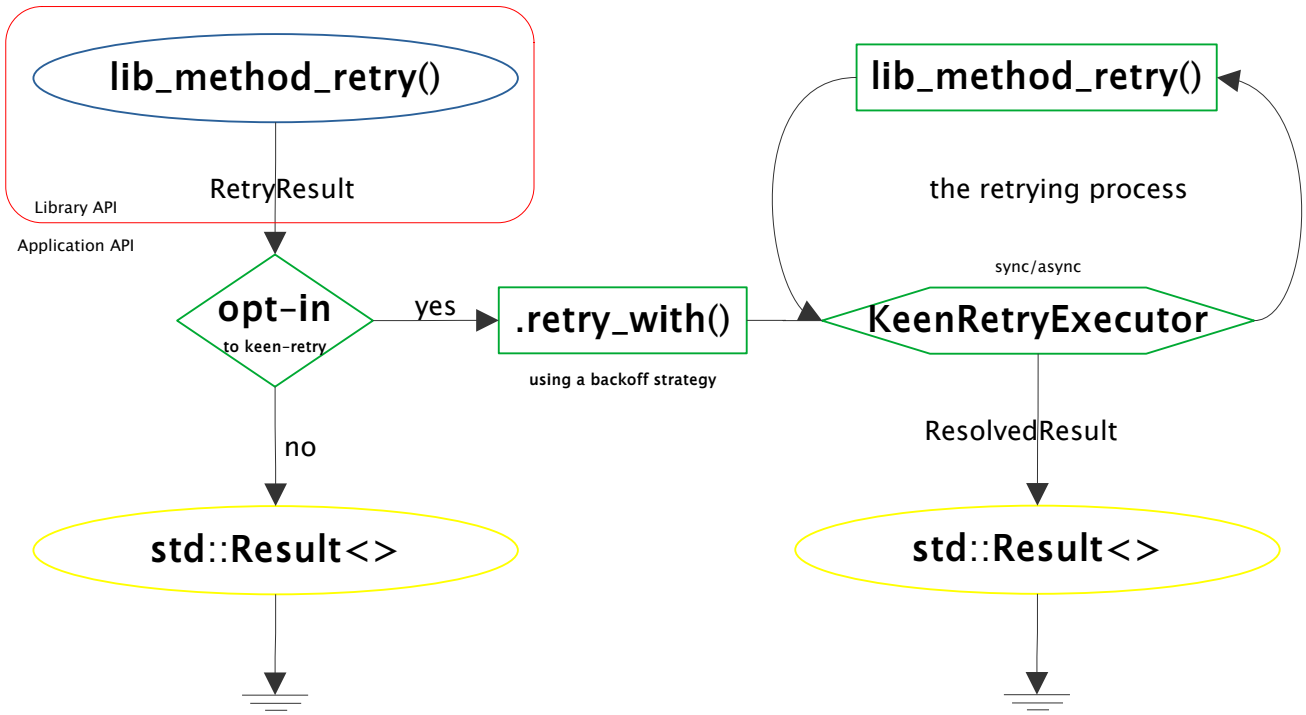
- **Retryable Result**: The “Library API” consists, primarily, of the type `RetryResult`, which enhances the standard Rust `Result` type by introducing a third variant: `Transient` (in addition to `Result::Ok` and `Result::Err`). Special care was taken, throughout the crate, to leverage zero-copy – this 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 fatally 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 library 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

Throughout

Understanding the Application API

…

Core Concepts

`<<here I should introduce how the library works>>`
`keen-retry` introduces several key abstractions:

- `RetryConsumerResult`, `ProducerRetryResult`, `RetryResult` serve as wrappers around retryable operations, similar to `Result<>`.
- `KeenRetry[Async]Executor` is responsible for executing the retry logic according to the selected backoff algorithm and retry limits.
- `ResolvedResult` provides a comprehensive result type encompassing all possible outcomes of a retry operation.

Use Cases

The `keen-retry` crate is versatile, finding its place in a wide array of scenarios where operations may fail due to transient issues. Here are several use cases that exemplify how the library can be employed:

Network Operations

In the realm of network requests – such as API calls, database transactions, or remote service interactions – failures can occur due to temporary network instability or service unavailability. `keen-retry` can wrap these operations, allowing them to be retried with a strategy appropriate for the context, whether it be immediate, delayed, exponential backoff, or a custom approach.

Resource Access

Operations involving file systems or other resources can intermittently fail due to locks, contention, or resource limitations. Employing `keen-retry` enables developers to smoothly handle these scenarios, improving user experience and system reliability.

User-Level Tasks

Tasks initiated by end-users that are prone to failure, such as uploading files to a server or posting data to a web service, can benefit from `keen-retry`. By integrating retry logic, applications can reduce the need for manual retry attempts by users, thereby enhancing the overall usability.

Application Start-Up

During the start-up phase of applications, especially those that rely on external services or databases, `keen-retry` can ensure that initial connections are established reliably, even in the face of transient service delays or unavailability.

Inter-Service Communication

In microservices architectures, services often need to communicate with each other. Given the distributed nature of such systems, `keen-retry` is invaluable for managing the inherent instability and ensuring that communication between services is resilient.

Background Jobs and Workers

For background processing systems, workers may encounter temporary issues while handling jobs. `keen-retry` can be used to automatically retry failed jobs according to specified policies, thereby reducing the need for manual intervention and monitoring.

Custom Application API Implementations

`keen-retry`'s Application API allows library consumers to define their own retry logic, tailored to the specific requirements of their application. This ensures that the retry mechanisms are as fine-grained and context-specific as necessary.

Examples from `tests/use_cases.rs`

The `tests/use_cases.rs` file provides concrete examples of these scenarios. It showcases the `keen-retry` crate in action, demonstrating how to handle both synchronous and asynchronous operations, incorporate detailed instrumentation, and define complex retry policies. These examples are instrumental in illustrating the practical application of the library's features.

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.
- If you own the library, additional methods should be added to the API: `*_retry()`. These methods should map all the original methods suitable for retrying and return a `RetryResult`.
- If you don't own the library, these same methods may be added on an adapter type.

Having the *_retry() methods available in 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.

Pattern: 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 `produce_operation` is a user-defined function that returns a `Result`.
fn produce_operation_retryable() -> RetryProducerResult<OutputType, ErrorType> {
    match produce_operation() {
        Ok(output) => RetryResult::Ok(output),
        Err(e) => match e {
            ErrorType::Transient(_) => RetryResult::Retry(e),
            _ => 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 guarantees we can use `produce_operation_retryable()` as if it returned the standard `Result` type: this gives us the possibility to simply name this method `produce_operation()`.

Notice the following `keen-retry` error types exist to leverage zero-copy:

- `RetryConsumerResult`: use this for operations that require expensive input data – for instance, adding a message to a queue collecting the "processing results": in this case, retrying (in case adding to the queue fails) would be done by reusing the input;
- `RetryProducerResult`: use this for operations use resources to produce an output;
- `RetryResult`: this is the general type, encapsulating both input, output and the error type

Pattern: 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 an exponential backoff strategy for retries.
let result = produce_operation_retryable()
    .retry_with(|_| produce_operation_retryable())
    // backoff exponentially, from 100ms to 15 seconds in 20 re-attempts
    // with +/- 10% random variance — ideal for retrying network requests
    .with_exponential_jitter(100..=15000, 20, 0.1)
    .into();
```

Here, an exponential backoff strategy is applied in 20 steps, where each sleeping duration is subjected to a random variation of 10%.

Please refer to the appendix "Common Backoff Algorithms and Analysis" for additional information on common strategies and other usage scenarios.

--- How to write a `keen-retry` wrapper for your library

--- How to add a `keen-retry` layer for a third-party library

--- How to write meaningful instrumentation for retry operations

-- Write tests that asserts on the number of retries actually done. Preferably, don't put counters only in tests, but in the application, where applicable, so these statistics are more visible.

-- Use the powerful "composable retry policies"

-- Choose carefully the appropriate backoff strategy

`keen-retry` supports custom backoff strategies, which may be easily expressed through iterators – be it with fixed or randon delays, or even with arithmetic or geometric progressions. Choosing the right strategy helps in avoiding resource contention and giving dependent services time to recover.

Anti-Patterns

The general recommendations are to always avoid:

- Retrying operations that fail due to non-transient errors.
- Infinite retry loops without backoff strategies, which can lead to resource exhaustion.
- Overly complex retry policies that make the code difficult to understand and maintain.

--- Some anti-patterns:

-- When enabling `keen-retry` in your library:

1) Implement a `keen-retry` wrapper for your library's API but not writing integration tests to assure they work as intended.

2) Not distinguishing among transient and permanent errors

-- When using a library that has a `keen-retry` wrapper

3) Using a library that has a `keen-retry` wrapper around retryable operations but not writing unit tests to prove they perform the retries & instrumentation as intended.

-- General

4) Avoiding Retry Storms

In distributed systems, synchronizing retries can lead to 'retry storms' where many clients retry simultaneously, overwhelming the system. Use `keen-retry`'s features to randomize retry intervals and spread out the load.

5) Not adding instrumentation to retries

Retrying may have the downside, when bugs happen, of causing too much extra / unnecessary work to be going on silently. Instrumenting the retries is key to detect any logic misbehaviors as soon as possible – preferably, before reaching production. When writing tests, use counters on the number of retries and assert on them.

6) Nesting without Timeouts

Nested retry operations may scale exponentially, regarding the time they take to exhaust all nested backoffed retries. For this reason, it is always advisable to use "Timeouts" when nested retries are in place.

Advanced Features

Discuss the nuances of using `keen-retry` with both synchronous and asynchronous operations, and how to handle nested retries for operations that depend on other retryable operations.

Instrumentation and Logging

`keen-retry` offers features for detailed instrumentation, allowing developers to log retry attempts, errors, and successes in a structured manner. The `ResolvedResult` type provides facilities for building succinct reports of retry attempts.

Performance Considerations

Discuss the zero-cost abstractions provided by `keen-retry`, which ensure that if an operation succeeds or fails fatally on the first attempt, no additional overhead is incurred.

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 on a local queue (with multiple consumers). 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, 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 workloads.

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 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:

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

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, increasing exponentially the backoff time with a given ratio:

```
.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 are 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.