



Scalable Web Applications

Samuel Williams

samuel@ruby-lang.org www.codeotaku.com

@ioquatix



Scalable Web Applications

Samuel Williams

samuel@ruby-lang.org www.codeotaku.com

@ioquatix

What if I told you, you can take your existing web application, and make it more efficient and scalable? “It’s not possible” you’d cry! Well... you are in for a surprise. Welcome to my talk “Scalable Web Applications”. My name is Samuel Williams and I’m a Ruby core team member. Today we are going to talk about scalability, and how Ruby 3 unlocks the next evolution of concurrency, to help you build scalable web applications.

What is Scalability?

スケーラビリティとは何か？

Firstly, let's talk about scalability. From a software perspective, scalability is about how a system's performance changes as the requirements on the system grow. A simple example would be, does your system get slower as the number of users increases? There are lots of different ways to talk about scalability, but let us consider four key questions.

Can I add capacity to handle increased demand?

増加する要求に対応できる容量を追加できるのか？

Can I add capacity to handle increased demand? This is perhaps the biggest problem when building scalable web applications. While trivial applications can be easy to scale up, once you start depending on external services, such as databases, key-value stores, software-as-a-service, it can be hard to know which parts of your system are causing performance issues and how to mitigate them.

Contention

コンテンション (競合)

In many cases, limited or metered resources are something that need to be carefully monitored as they can cause contention. As you try to increase the capacity of one part of your system, you can run into limitations in another part. As an example, if you increased the number of web server instances without considering limitations in your database, you might exceed your database connection limit and this will restrict your web servers' ability to serve more users, potentially resulting in errors.

Another example of metered resources include third party software-as-a-service. These hidden scalability issues are often overlooked when developers plan their architecture, and mitigation can be especially difficult because you are depending on someone else to scale with your requirements.

Efficiency

效率

Building computationally efficient systems ensures that your systems remain stable and responsive even as the problems they are solving grow in size. Choosing the right algorithms and topologies are critical during the early stage of your web application because the wrong decision can be very costly to fix later.

As an example, ensuring your database queries are using indexes correctly can have a big impact on query performance. As your data access patterns change, your indexes may become sub-optimal and cause performance regressions.

Will my application tolerate transient and enduring faults?

このアプリケーションは一時的、永続的な欠点に対しての耐性はあるのか?

Will my application tolerate transient and enduring faults? As your system grows in use, the impact of a single fault also grows in size. While this may not be immediately obvious, it can have a significant impact on your growing business and your users.

Fault Tolerance

フォールトトレランス

Fault tolerance is the ability of a system to keep functioning even if one part fails. Ideally, systems have enough redundancy to continue working until the fault is resolved. An example of this would be a database cluster with multiple replicas. If one replica fails, the cluster continues to operate at reduced capacity until a new replica can be provisioned.

Ideally, your application code should not need to know about the fault, internally the request should be re-routed to a working replica.

Isolation

アイソレーション

By building isolated systems, faults can be contained. In web servers, this can be achieved by using a multi-process model. If one process crashes, the others keep on working. Another way to achieve this is to move logical units of work into background jobs. This works especially well for external service providers who might encounter transient failures. The background job queue acts as a load levelling mechanism.

This can also apply to security, for example having one system responsible for application logic and another isolated system responsible for sensitive information.

Do I understand the health and performance of the live system?

ライブシステムの管理と実行状況を理解できているか?

Do I understand the health and performance of the live system? Lack of visibility into an application makes it hard for developers to identify and fix problems. Capturing errors, latency and metrics from production environment is essential smooth operations, and analysis of trends can help prevent scalability issues before they occur.

Logging

ロギング

Having good tools for logging errors and latency is critical to identifying problems which need to be addressed. Capturing specific traces of an applications' execution provide valuable insights into scalability issues. As an example, database logs can help developers find and fix poorly performing queries.

Metrics

メトリック

Understanding usage patterns and tracking application metrics becomes more important as utilisation increases. Aggregations and historical data help you understand trends and changes in application performance. This data can form the basis for alerting developers when something is going wrong and to help identify different parts of the system which are impacting the scalability of an application.

Can I build the system with my time and cost budget?

時間内、予算内に作れるのか？

Can I build the system with my time and cost budget? As a business seeks to grow and acquires more customers, how their systems are designed will impact their ability to build new features and maintain existing ones. Different programming languages and frameworks have different approaches to scalability, and this can impact developers and their ability to deliver features on time and on budget.

Lighting Money on Fire

お金に火を付ける

Rewriting your web application to improve scalability can be a very difficult process. The alternative approach is known as “lighting money on fire” and is when you simply over-provision everything to the point where you achieve the service level you require. An example of this would be adding excessive amounts of hardware to the database cluster rather than optimising poorly performing queries. Unfortunately (or maybe fortunately), this approach will only get you so far.



It's also important to consider the resource consumption of any given system. Efficient, scalable systems are great for the planet. Less resources are required for the same work. The technology sector is responsible for about 2% of global carbon emissions and as a civilisation we are facing an unprecedented set of challenges relating to climate change. We can help by improving the scalability of our systems, to reduce natural resource consumption and energy utilisation. Our systems can also help to build a more efficient society. This is a good thing that we can take responsibility for.



Capacity

Isolation

Visibility

Budget

So we have talked about four areas which underpin any scalable system. The ability to expand capacity. The ability to isolate faults and problems. Visibility into your application behaviour and performance. Delivering your applications on time and within budget. But how does this apply to Ruby, and how can Ruby facilitate these goals?


How do we build scalable web applications?

スケーラブルなアプリケーションを作るにはどうすればいいか？

Ruby is a great language for building web applications. It has a certain simplicity and elegance which is hard to find in other languages. Many success stories support its continued usage... and yet... looking at the latest trends, we see alternative platforms becoming more more popular. The stability of Ruby as a platform is both its biggest strength and also a significant weakness.

Ruby 3

Ruby is ripe for innovation, and I'm proud to share with you that Ruby 3 will include a new light-weight scheduler for improved concurrency. This gives Ruby a transparent event-driven architecture. Taking advantage of this scheduler enables simpler, robust systems with improved scalability, without requiring significant changes to application code.




 Ruby »


Ruby master

Search:

OverviewActivityRoadmapIssuesNew issueWikiRepositoryMail to issueSettings

Feature #16786

 Edit  Unwatch  Copy



Light-weight scheduler for improved concurrency.
Added by [ioquatix \(Samuel Williams\)](#) about 20 hours ago. Updated about 10 hours ago.

Status:

Open

Priority:

Normal

Assignee:

-

Target version:

-

[\[ruby-core:97878\]](#)

Description

 Quote

Abstract

We propose to introduce a light weight fiber scheduler, to improve the concurrency of Ruby code with minimal changes.

Background

The initial design was conceived of in 2017, and finally this year, 2020, we made a formal proposal for Ruby 3. This proposal specifies the scheduler interface which can work across different interpreters, including CRuby, JRuby and TruffleRuby. The actual implementation of the scheduler is provided by a separate Ruby gem.

Callback-based Concurrency

コールバック式並列処理

The most popular event-driven platform would probably be Node.js. Initially, they used callbacks exclusively to implement continuation-passing-style concurrency. Callbacks can be hard to reason about because they invert the flow-control of your code - instead of sequential execution, you end up with your program split up into separate callbacks, and the order of operations is often ambiguous.

Promise-based Concurrency

プロミス式並列処理

To simplify chaining callbacks together, promises were introduced. A promise can be resolved and this triggers the attached callbacks. They are semantically identical to callbacks and present all the same problems, although they are slightly easier to use in practice.

Async/Await-based Concurrency

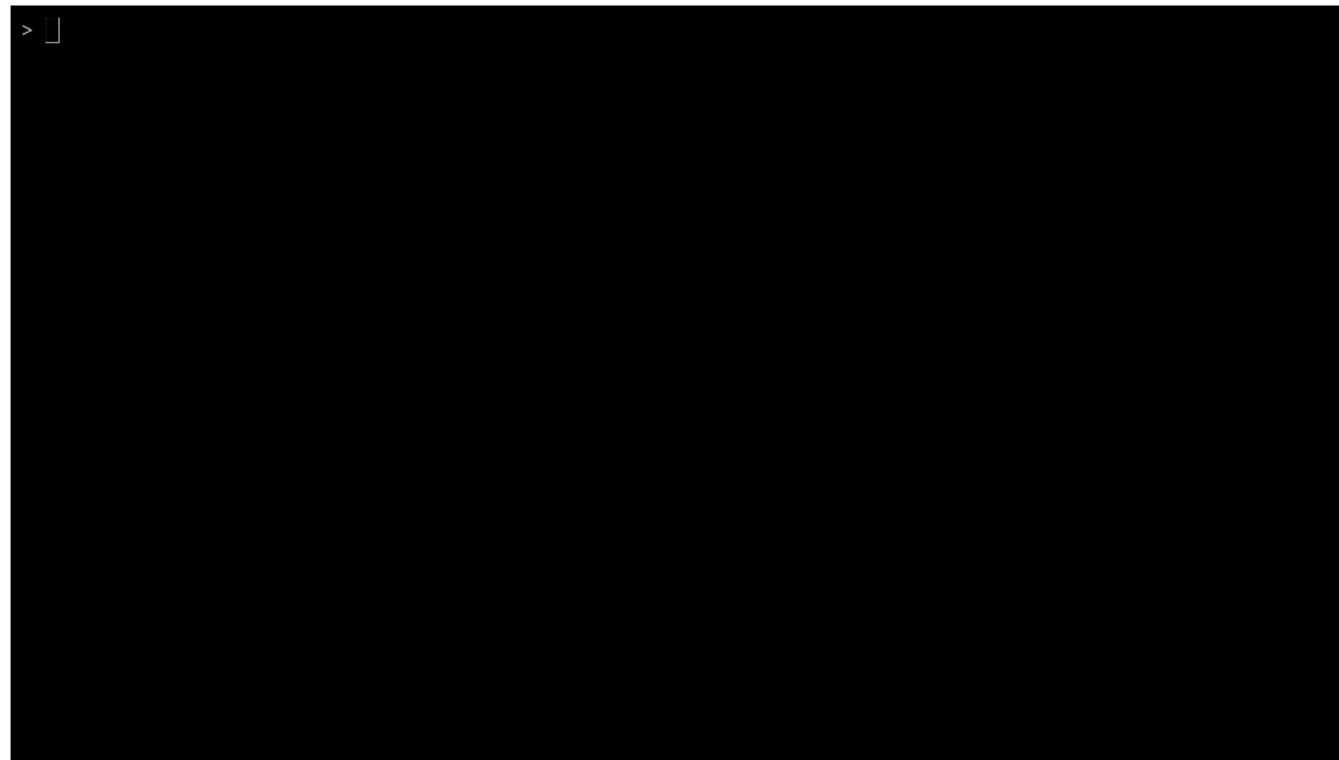
Async/Await式並列処理

In order to fix the inversion of flow control, new keywords were introduced to hide the use of promises and callbacks. Many languages have adopted this approach but the biggest problem is that it requires existing code to be rewritten. This ultimately results in functions which are incompatible with each other - normal functions and asynchronous functions.

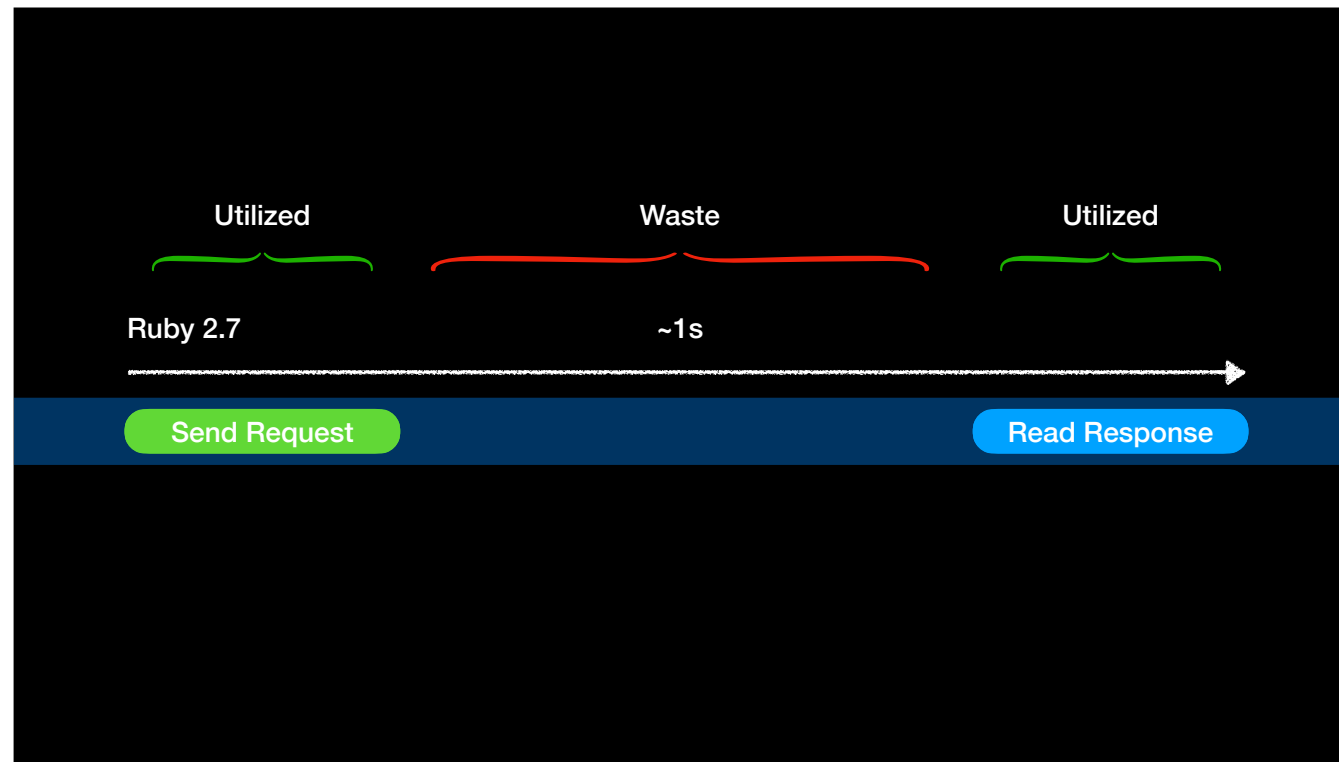
Fiber-based Concurrency

ファイバー式並列処理

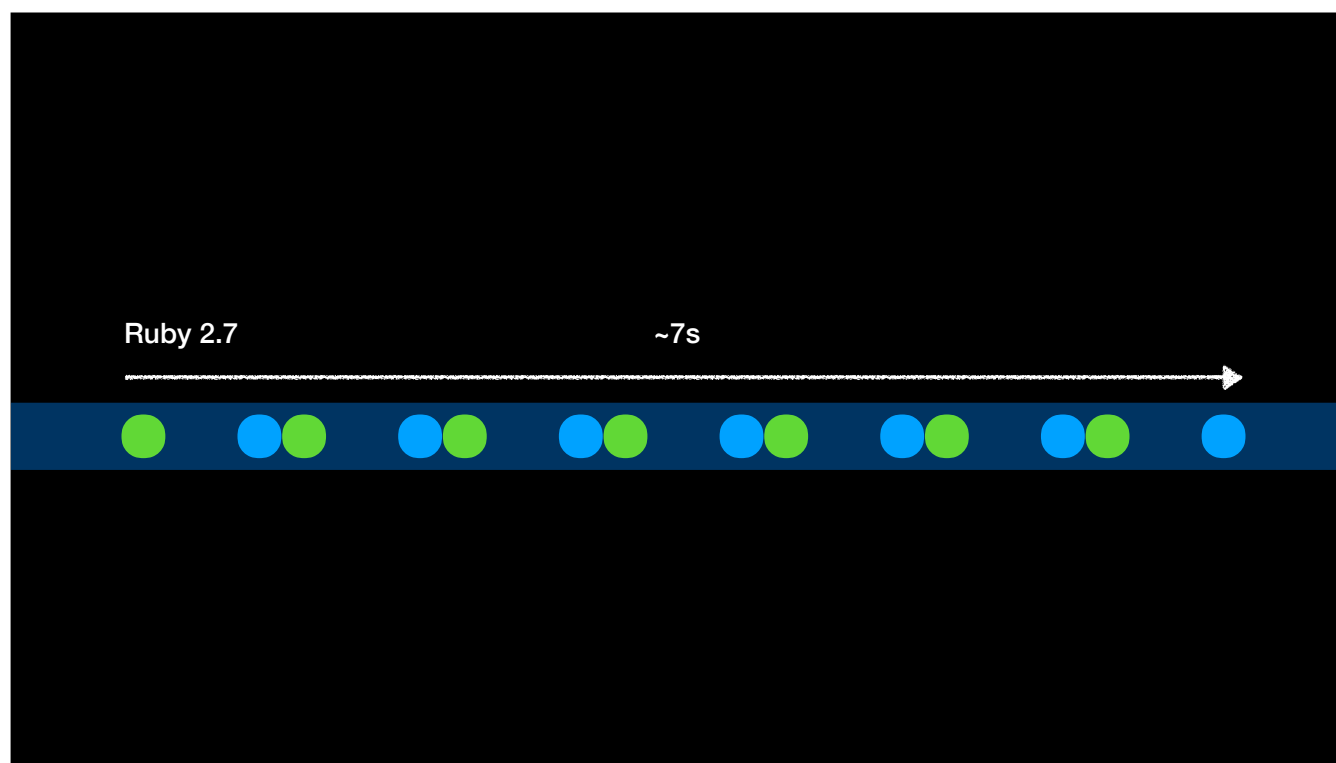
I wanted to avoid this in Ruby, so about 4 years ago I started experimenting with fiber-based concurrency. Fibers allow you to write asynchronous code without the inversion of flow control that occurs with callbacks. In effect, they present the same sequential execution as `async/await` but without the extra keywords. Every function is optionally asynchronous, and the fiber itself suspends its execution until the result is ready.



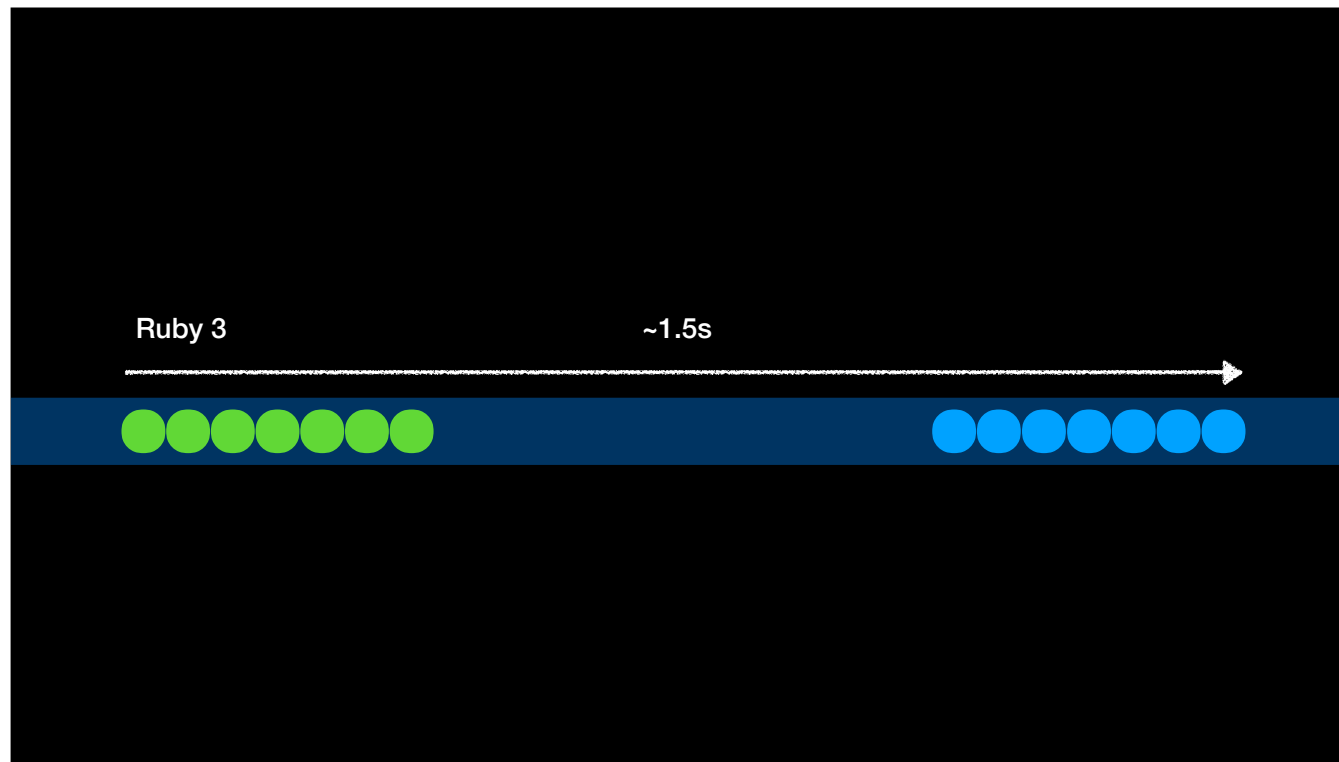
Let's compare the performance of Ruby 2.7 and Ruby 3 with the same Net::HTTP code.



In Ruby 2.7, each request consists of two parts: sending the request to the remote server and reading the response when it comes back. In between those two operations, we need to wait for the network.



When we look at 7 sequential requests, each request happens after the previous one, because `Net::HTTP.get` is a blocking operation, and it has a very high latency going from my laptop to google and back again. We must wait for a long time for all the operations to complete.

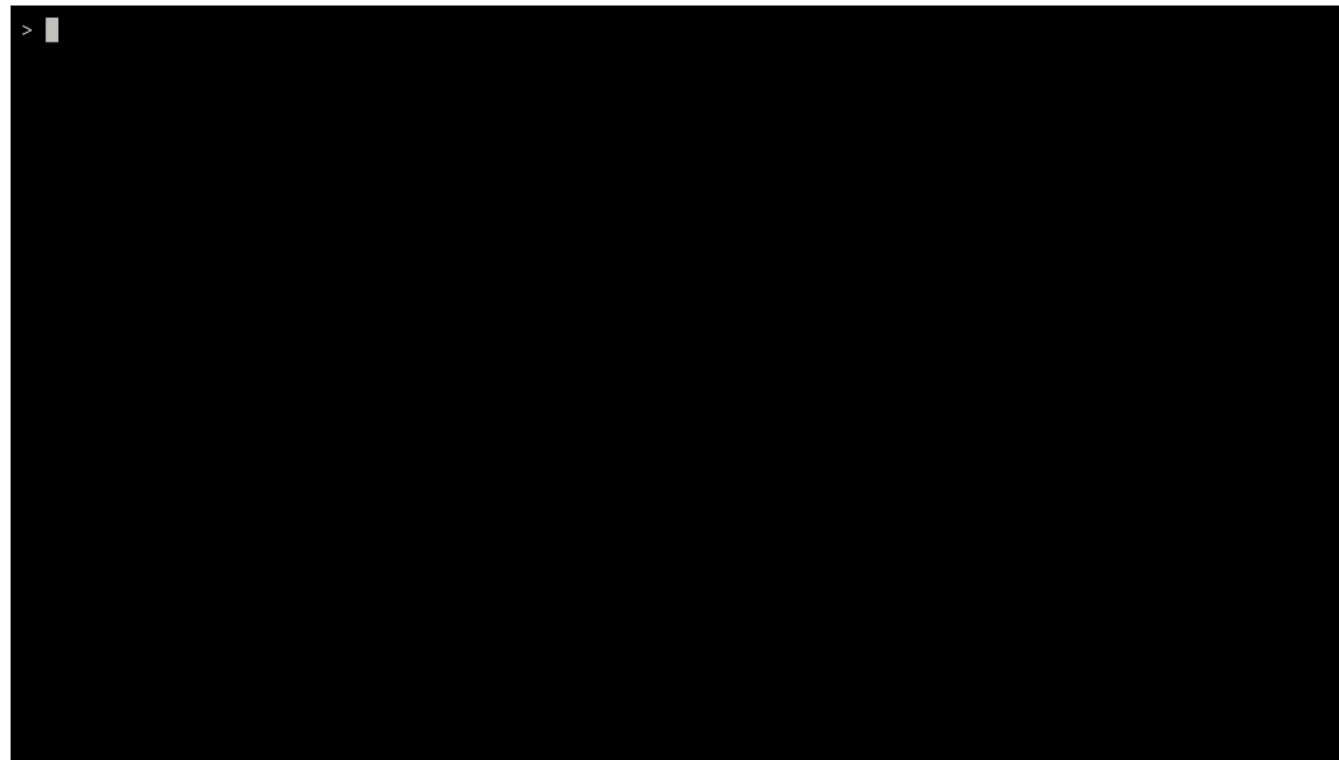


But there is no reason for us to wait like this. In Ruby 3, this operation becomes non-blocking, and we can interleave all 7 requests which improves total latency significantly. This is something you can easily take advantage of in your web application, if you are, for example, accessing external services.

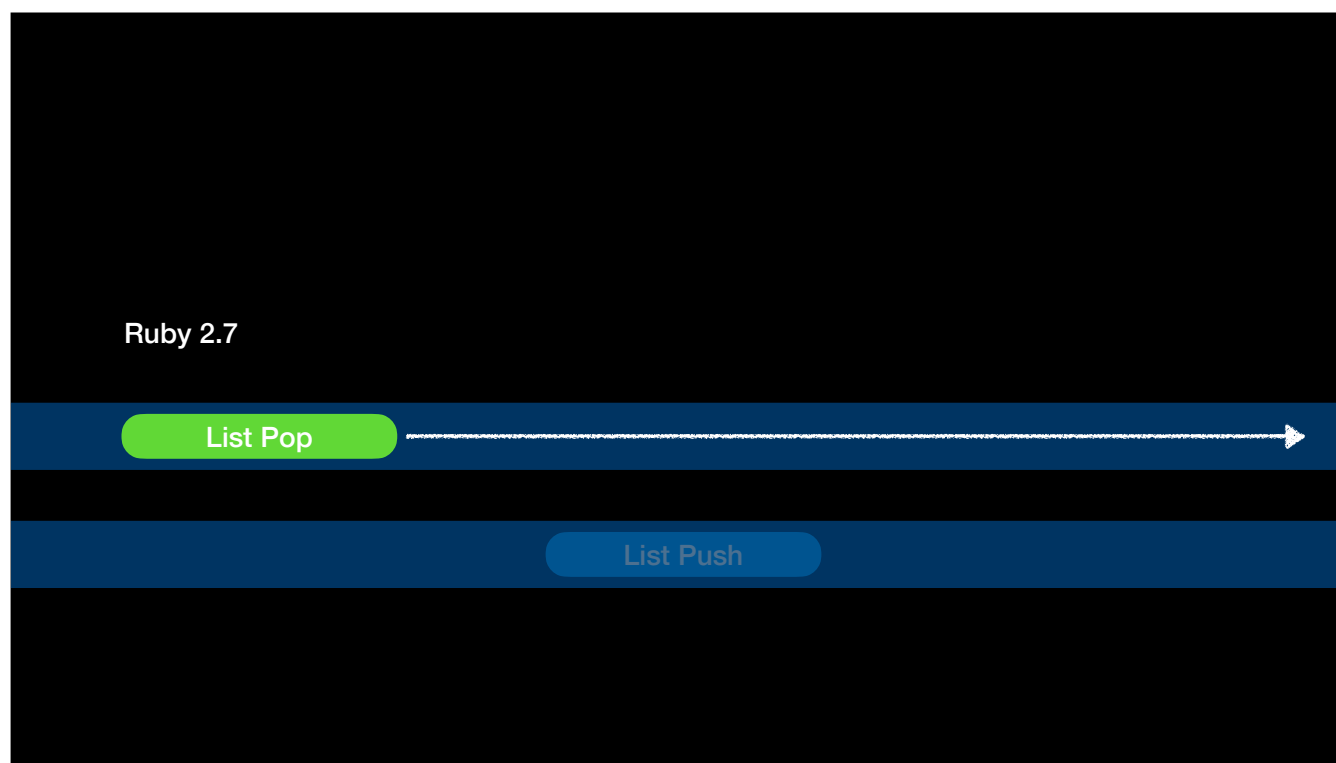
Compatibility

コンパチビリティ（互換性）

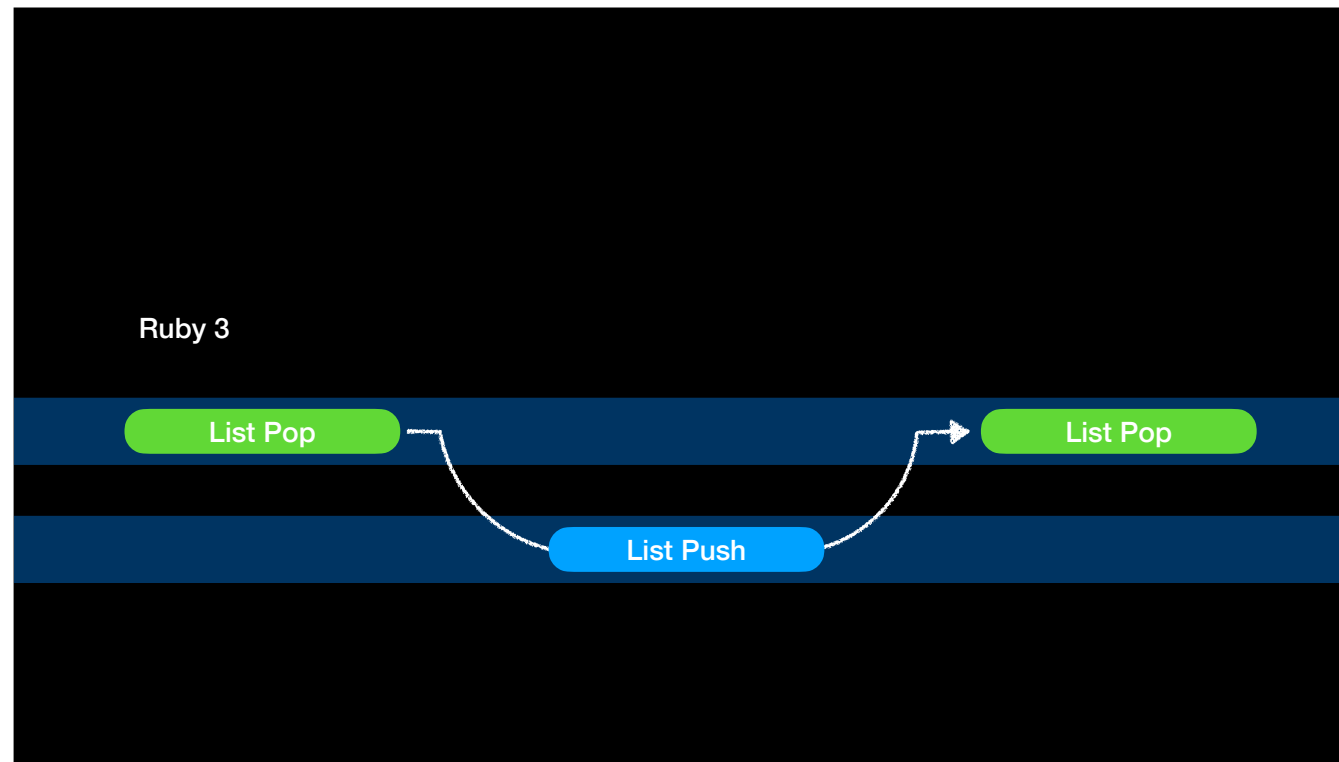
We have worked hard to make this compatible with existing code where possible. The only requirement is for you to break your code up into logical blocks that can be run concurrently. However, in many cases, this can be done automatically. Individual web requests are already able to be executed concurrently. The same applies to batch processing systems. By taking advantage of these existing models, we can improve the scalability of Ruby web applications with minimal changes.



Let us look at how Ruby 3 enables us to use the existing Redis gem with minimal changes.



In Ruby 2.7, because the Redis “blpop” is a blocking operation, the parent task never gets a chance to add the item to the list, so the program hangs.



However, taking advantage of the scheduler in Ruby 3, this operation becomes non-blocking, so while we are waiting to pop an item off the list from one task, we can use another task to push the item. This kind of concurrency enables new ways of building scalable systems.




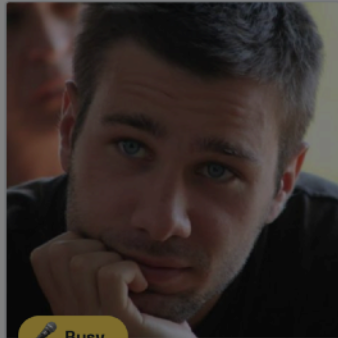
So we have used a gem called “Async” several times now. But what does it do and why is it important? Async provides the scaffolding for building reliable, scalable applications. It implements the event-loop which is used for scheduling the non-blocking operations, and it can transparently take advantage of the new Ruby 3 scheduler interface.

Sequential tasks.

シーケンシャルタスク

As we have seen, Async provides a structure for executing your programs concurrently. Within those tasks, your code executes sequentially; no special keywords or changes to program logic is required.

[Why GitHub?](#) [Enterprise](#) [Explore](#) [Marketplace](#) [Pricing](#) [Sign in](#)



Busy

Preparing for wroc_love.rb

Janko Marohnić

janko

Block or report user

“Lately I've been looking into async,
as one of my projects would really
benefit from non-blocking I/O.

It's really beautifully designed.”

Janko Marohnić, on the general design of Async.

Stream...

Ruby 495 17

mini replacement for RMagick

Ruby 2.4k 305

Web Server

Webサーバ

The next critical component for building scalable web applications is the web server.



Falcon is an Async-based web server which helps you build highly scalable web applications. It executes each request in an asynchronous task and transparently handles non-blocking operations.

Pure Ruby web application server.

Pure Ruby ウェブアプリケーションサーバ

Falcon is implemented in pure Ruby, yet in performance, it rivals other servers which have native components. Because of this, it is very easy to use, deploy and customise.

**Supports HTTP/1, HTTP/2 with
zero configuration.**

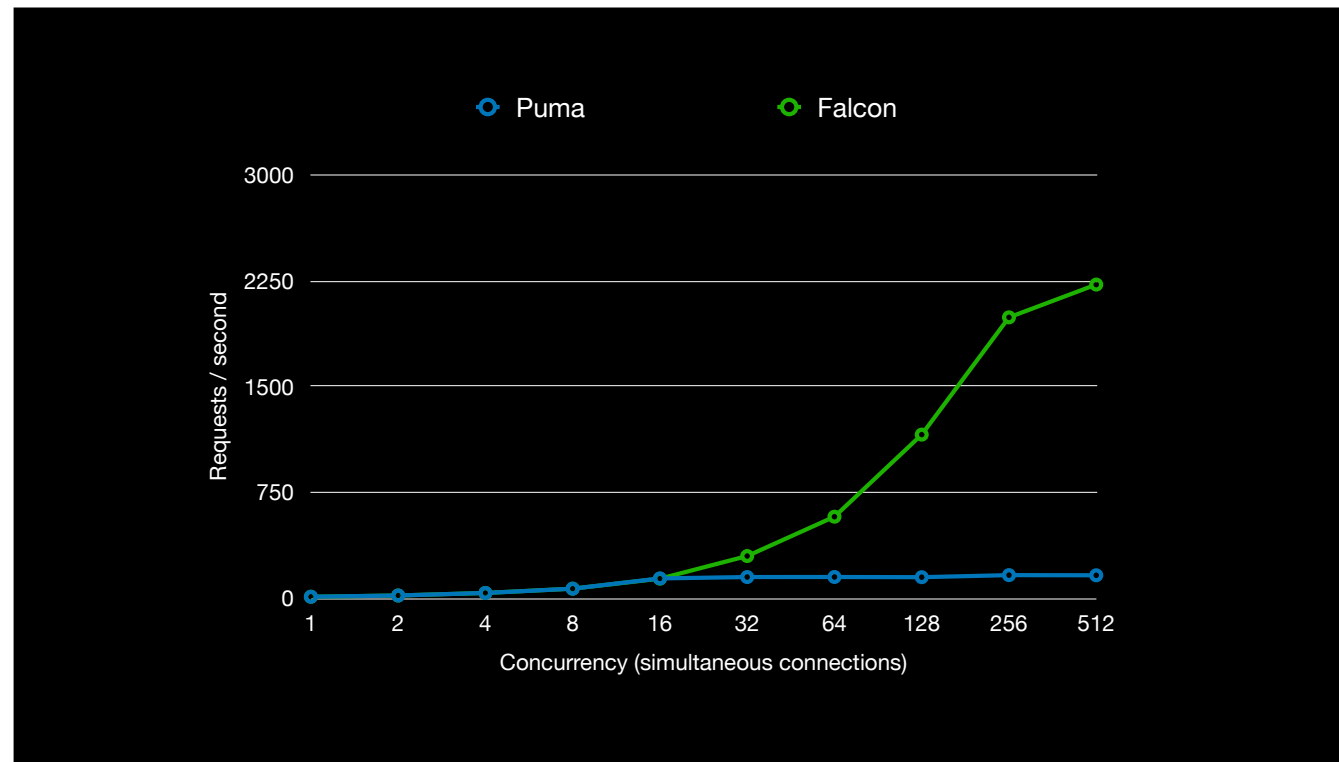
ゼロコンフィギュレーションでHTTP/1,
HTTP/2のサポート

Out of the box, it supports HTTP/1 & HTTP/2, with zero configuration. This provides a modern development & production platform for Ruby web applications.


Is it scalable?


スケーラブルなのか？

Is it scalable? Well, like anything, it depends on how you use it...



But various micro-benchmarks show that it can perform very well, especially when there are operations which are non-blocking. This graph shows Puma vs Falcon. Puma plateaus at 16 workers, while Falcon is able to scale up until all CPU cores are pegged. We also recently demonstrated one million WebSocket connections to a single Falcon instance.

 Why GitHub? ▾ Enterprise


:P

Bryan Powell
bryanp
[Block or report user](#)

“Async is the right model because web apps are almost always I/O bound. The Ruby web ecosystem is really lacking in scalability (e.g. WebSockets on Puma). Async unlocks the next tier of scalability in the most Ruby-like way possible.”

Bryan Powell, on migrating from Puma to Falcon.

[Fork](#)
● Ruby 2 Updated on 8 Feb

Does it work with Rails?

Railsに使用できるのか？

Does it work with rails?

**Yes.
It's Rack compatible.**

はい、Rackに対応しています。

Yes, it's rack compatible, and with the additions of the Ruby 3 scheduler, it can transparently improve the scalability of your existing applications.

Database Access

データベースアクセス

Accessing data is a critical part of most web applications.

ActiveRecord

ActiveRecord is a commonly used gem for accessing databases. It currently uses one connection per thread. Because of that, multiple independent fibers on the same thread may be held up waiting for the connection. This is a design choice that limits concurrency. Fortunately, there is discussion happening right now about how we can improve this situation.

Make pg more Ruby-3.0 fiber scheduler friendly #342



larskanis opened this issue on 7 Jul · 1 comment



larskanis commented on 7 Jul

Collaborator

This was raised by @ioquatix in [ffi/ffi#797](#) (comment) :

In order to support the upcoming Ruby 3.0 fiber scheduler, you need to do only one thing:

When you would call a blocking function that does I/O, instead call `IO#wait_readable` or `IO#wait_writable` respectively.

If you want to support Ruby 2.x, then allow the user to inject a custom IO class, which responds to `.new` and implements `#wait_readable` and `#wait_writable`.

You can check how to do it here:

<https://github.com/socketry/db-postgres/blob/6fbdd03dd6b04c57c5b7995b60e3f626c674e609/lib/db/postgres/native/connection.rb#L106-L139>

We will be augmenting the C interface with some improvements for invoking `IO#wait_readable/wait_writable` but it hasn't landed yet.

Feature description is here: <https://bugs.ruby-lang.org/issues/16786>




3



3

Not all database drivers use non-blocking I/O. However these issues can be addressed. The maintainer of the Postgres gem is interested to add support for non-blocking I/O. I believe we will see expanded support from other maintainers once the Ruby 3 is released.

 socketry / db

Sponsor

Unwatch

<> Code

Issues 1

Pull requests

Actions

Projects

Wiki

Security

Insights

master


2 branches








5 tags

Go to file

Add file

Code

 ioquatix Fix documentation typo. ✖ d8c1dff on 8 Jul 🕒 35 commits

 .github/workflows	Fix test database user creation.	last month
 docs	Fix documentation typo.	last month
 guides	Update documentation.	last month
 lib	Minor version bump.	last month
 spec	Improved query builder and consistency of interface.	last month
 .editorconfig	Modernize gem.	last month
 .gitignore	Modernize gem.	last month

As an alternative, we recently started implementing a low level database gem, called `db`. It currently provides non-blocking Postgres and MariaDB/MySQL access. You can use this today if you want low-latency streaming database queries.

Redis Access

Redisアクセス

Redis is a commonly used system for key-value storage and in-memory data structures.

Redis Gem

As we have demonstrated, the Redis gem is compatible with the Ruby 3 scheduler with no changes to application code required.

socketry / **async-redis**

Sponsor

Unwatch

<> Code

Issues 2

Pull requests

Actions

Projects

Wiki

Security

Insights

master


6 branches

12 tags

Go to file

Add file

Code



davidor and ioquatix sentinels: split #select_slaves

df588b4 on 10 Apr 107 commits

.github/workflows	Add redis service.	6 months ago
lib/async	sentinels: split #select_slaves	4 months ago
spec	Update specs for changed interface.	6 months ago
.editorconfig	Standardise .editorconfig.	2 years ago
.gitignore	Ignore .coverage.db.	14 months ago
.rspec	Initial implementation - basic client and protocol.	2 years ago
Gemfile	Prefer frozen string literals.	6 months ago

However, we also have our own implementation called `async-redis`, which was built from the ground up for concurrency and has more convenient connection pooling built in.

HTTP Access

HTTPアクセス

HTTP is a key protocol of the modern web. There are many clients, and they should all be compatible with the Ruby 3 scheduler.

socketry / **async-http**

Sponsor

Unwatch

<> Code

Issues 13

Pull requests 2

Actions

Projects 1

Wiki

Security

Insights

master

7 branches

153 tags

Go to file

Add file

Code

ioquatix Patch version bump.

0cdf8c2 4 days ago

691 commits

github/workflows	Modernize gem.	2 months ago
bake/async	Initial (working?) support for request/response trailers.	8 months ago
examples	examples/fetch: update rack version	4 days ago
lib/async	Patch version bump.	4 days ago
spec	Whitespace.	4 days ago
.editorconfig	Modernize gem.	2 months ago
.gitignore	Modernize gem.	2 months ago
.rspec	RSpec does color by default now.	4 years ago

In order to support all the features we wanted for Falcon, we built our own `async-http`. It provides client and server support for HTTP/1, HTTP/2, and hopefully soon HTTP/3.

socketry / **async-websocket**

Sponsor

Unwatch

<> Code

Issues 1

Pull requests

Actions

Projects

Wiki

Security

Insights

Se

master

4 branches

23 tags

Go to file

Add file

Code

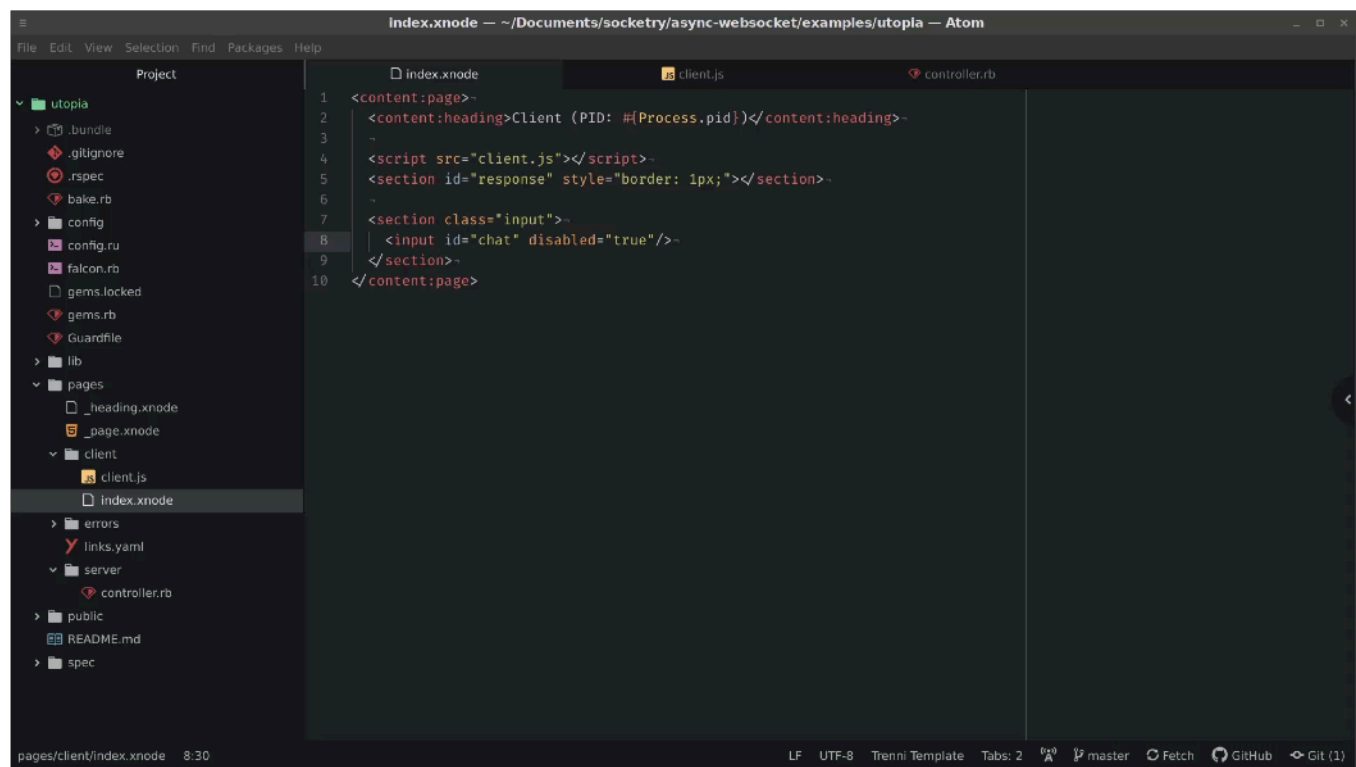
ioquatix Use normal GC.

f34b4db 4 days ago

161 commits

github/workflows	Modernize gem.	2 months ago
docs	Add static documentation.	6 months ago
examples	Use normal GC.	4 days ago
guides	Modernize gem and improve documentation.	6 months ago
lib/async	Bump dependencies.	4 days ago
spec	Update specs to handle authority argument.	2 months ago
.editorconfig	Modernize gem.	2 months ago
.gitignore	Modernize gem and improve documentation.	6 months ago

Along with this, we also have `async-websocket` which supports WebSockets over HTTP/1 and HTTP/2. This makes it trivial to build scalable web applications with shared client and server state.



127.0.0.1:6379>
127.0.0.1:6379> SUBSCRIBE chat:general
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "chat:general"
3) (integer) 1
}

bandit@bandit:~/Documents/asynctrippy/asynctrippyexamples/utopia\$
bandit exec falcon serve
0.0s info: Falcon::Command::Serve [pid=0x11bc] [pid=13654] [2020-11-20 02:46:55 +1300]
| Falcon v0.36.7 taking flight! Using Async::Container::forked (count=8).
| - Binding to: #Falcon::Endpoint https://localhost:9292/ ()>
| - To terminate: Ctrl-C or kill 13654
| - To reload configuration: kill -HUP 13654
0.02s info: Falcon::Controller::Serve [pid=0xc60c] [pid=13654] [2020-11-20 02:46:55 +1300]
| Starting Falcon Server on https://localhost:9292/
0.04s info: Async::Container::Process::Instance [pid=0xc6d4] [pid=13656] [2020-11-20 02:46:55 +1300]
| - Per-process status: kill -USR1 13656
0.04s info: Async::Container::Process::Instance [pid=0xc6fc] [pid=13657] [2020-11-20 02:46:55 +1300]
| - Per-process status: kill -USR1 13657
0.05s info: Async::Container::Process::Instance [pid=0xc724] [pid=13658] [2020-11-20 02:46:55 +1300]
| - Per-process status: kill -USR1 13658
0.05s info: Async::Container::Process::Instance [pid=0xc74c] [pid=13659] [2020-11-20 02:46:55 +1300]
| - Per-process status: kill -USR1 13659
0.05s info: Async::Container::Process::Instance [pid=0xc6ac] [pid=13655] [2020-11-20 02:46:55 +1300]
| - Per-process status: kill -USR1 13655
0.05s info: Async::Container::Process::Instance [pid=0xc79c] [pid=13661] [2020-11-20 02:46:55 +1300]
| - Per-process status: kill -USR1 13661
0.06s info: Async::Container::Process::Instance [pid=0xc774] [pid=13660] [2020-11-20 02:46:55 +1300]
| - Per-process status: kill -USR1 13660
0.06s info: Async::Container::Process::Instance [pid=0xc7c4] [pid=13662] [2020-11-20 02:46:55 +1300]
| - Per-process status: kill -USR1 13662
0.06s error: Async::Task [pid=0xc7c] [pid=13655] [2020-11-20 02:47:10 +1300]
| openssl: error: SSL_accept returned=1 errno=0 state=error: sslv3 alert certifi-
cate unknown
| - /home/bandit/.gem/ruby/2.7.0/gems/openssl-3.0.1-1/lib/openssl/bio.rb:275 in 'accept-
with_lock'

Client (PID: 13660) - Utopia - Chromium

UTOPIA

Client (PID: 13660)

Client (PID: 13662) - Utopia - Chromium

UTOPIA

Client (PID: 13662)

Performance

パフォーマンス

How does it perform?

**~5,000 active WebSockets per
process.**

We are comfortable recommending around 5,000 active WebSockets per process. More than this, and the Ruby garbage collector overhead becomes an issue. As we push the limits of Ruby, we find more areas that require attention, and we look forward to tackling this problem.

**Similar or better performance
than other Ruby web servers.**

<https://github.com/socketry/falcon-benchmark>

We also have a benchmarks which compares Falcon with Puma, Unicorn, Passenger and Nginx in different situations. Falcon is consistently on top, especially when it comes to non-blocking work-loads.

What are the next steps?

次のステップは何か？

What are the next steps? Well...

It depends on you.

それはあなた次第です。

It depends on you...

The community.

コミュニティー
実際に使用して、意見を下さい。

The community. Please try out these new features in Ruby 3 and give me feedback.

The businesses.

ビジネス
このプロジェクトへの支援をお願いします。

And the businesses. I hope these new ideas can provide opportunities for innovation. Please support this important work.

Thank you.

ありがとうございました。



Scalable Web Applications

Samuel Williams

samuel@ruby-lang.org www.codeotaku.com

@ioquatix