ChatGPT

BRRTRouter Proposed Issues and Roadmap (May 2025)

Test Stability

Title: Fix Flaky Tests and Re-enable Ignored Test Cases
Labels: bug, test, good first issue
Milestone/Epic: Test Stability
Summary: Several integration tests exhibit intermittent failures or have been marked with `#[ignore]` due to unreliability. For example, tests that start an HTTP server coroutine and immediately send requests can sometimes fail or deadlock if the server isn't fully ready, especially when tests run in parallel. Some tests have been disabled (ignored) pending fixes, reducing overall coverage and confidence in the test suite. These flaky tests and ignored cases hinder CI stability and should be addressed to ensure the router behaves predictably under test conditions.
Steps to Reproduce: 1. Run the full test suite multiple times (e.g., `just test` or `cargo test -- --include-ignored`) in rapid succession.
2. Occasionally, observe failures or timeouts in certain tests (e.g., a request not receiving a response, or port binding conflicts when tests run concurrently).
3. For ignored tests, manually include them (using `cargo test -- --ignored`) and observe that they fail or hang consistently under current conditions.
Proposed Fix: Investigate each flaky test to identify the root cause (race conditions, timing issues, or shared state). Implement synchronization or configuration improvements such as waiting for the server to signal it's listening before sending requests, or using unique ports per test to avoid conflicts. For example, tests currently bind an ephemeral port and drop the listener immediately before starting the server [1] [2], which could race when multiple tests run in parallel. Introducing a small delay or handshake (e.g., the server writes to a channel or log when ready) would make startup deterministic. Likewise, replace or remove any brittle timing assumptions (like fixed sleeps). Finally, re-enable the previously ignored tests and ensure they pass reliably. Mark tests that remain flaky with detailed comments or consider isolating them in separate serial test runs if absolutely necessary.
Relevant Code/Commits: - `tests/` integration tests such as `test_headers_and_cookies` show the server being started and a request sent immediately [2]. The server is stopped via an unsafe cancel call [3], which may not guarantee a graceful shutdown, potentially causing intermittent errors.
- Commit `6f90696` ("test: add header and cookie handling") demonstrates the current test pattern for starting/stopping the server and parsing responses [4] [3].
- Look for `#[ignore]` tags in the test suite (e.g., in `tests/*.rs`) to identify which tests are disabled and why. Each ignored test should have an associated comment or commit explaining the failure (for example, a test might be ignored due to a known bug in hot-reload or OpenAPI parsing).

Title: Ensure Deterministic Server Startup/Shutdown in Tests
Labels: enhancement, test
Milestone/Epic: Test Stability
Summary: The current testing approach for starting the HTTP server is not fully deterministic, leading to

race conditions between test threads. The server is started in a background coroutine and the tests immediately issue requests, and then terminate the server by canceling its coroutine [5]. If the server hasn't finished binding or is still initializing when the request is sent, the request might fail sporadically. Similarly, the abrupt shutdown via `handle.coroutine().cancel()` can terminate the server while it's mid-processing, potentially leaving sockets open or background tasks hanging. This non-deterministic startup/shutdown sequence can cause tests to flake and might mask true issues.
Steps to Reproduce: 1. Run integration tests with `cargo test` on a machine with multiple cores (Rust will run tests in parallel by default).
2. Occasionally, tests that start the server (for example, tests for headers/cookies) may fail to get a response, or one test's server might interfere with another if they inadvertently choose the same port.
3. These issues may be hard to reproduce consistently, but adding logging around server start/shutdown can reveal timing issues (e.g., a request arriving before the server logs "listening").
Proposed Fix: Introduce a handshake mechanism to make server startup synchronous for tests. One approach is to have the `HttpServer::start` method signal when the server is actually listening on the socket (for example, by setting a synchronization primitive or writing to a channel). Tests can wait on this signal before proceeding to send requests. Additionally, consider providing a graceful shutdown mechanism: for instance, a method on the server handle to stop accepting new requests and wait for in-flight handlers to complete before returning. Replacing the raw `handle.coroutine().cancel()` (which forcefully kills the coroutine [3]) with a cooperative shutdown will ensure consistent cleanup. Implementing these changes will eliminate timing-related flakiness and make test behavior consistent run-to-run.
Relevant Code/Commits: - In `tests/test_headers_and_cookies` (and similar tests), the server is started and used immediately [2]. There is no acknowledgement that the server is ready before `send_request` is called, which can lead to a race.
- The shutdown is performed by cancelling the coroutine [5], as seen in the test commit. This is a potential source of nondeterminism.
- The `HttpServer(service).start(addr)` call (likely in the `may_minihttp` integration) might offer hooks or return a handle that we can extend to include a ready notification. We should check the `HttpServer` implementation for any existing signals or consider modifying it to support this.
- Related commit: ensure tests set stack size (#94) – while primarily about stack size, it indicates test environment tweaks were needed to get stable results, underscoring the need for robust test setup.

Improved Coroutine Safety

Title: Add Panic Recovery for Handler Coroutines
Labels: enhancement, bug
Milestone/Epic: Improved Coroutine Safety
Summary: Currently, if a handler function panics, it can unwind the coroutine and potentially bring down the entire server thread or leave it in a bad state. There is no recovery mechanism in place – a panic inside a request handler will at best close that request's connection abruptly, and at worst crash the server process. This is undesirable for a robust server: a single buggy handler or unexpected condition should not crash the whole router. Instead, the router should catch such panics and convert them into an HTTP 500 Internal Server Error response, logging the panic for debugging. This improvement will increase the fault tolerance of the server.
Steps to Reproduce: 1. Register a handler that deliberately panics (for example, a handler that calls `panic!("test panic")`).
2. Start the BRRTRouter server and call the endpoint corresponding to the panic handler.
3. Observe that the server currently likely terminates the connection (and possibly logs an abort) instead of

returning a 500 error. If running under a test, the test will fail due to the panic propagating. In some cases, the panic might even kill the runtime thread.
Proposed Fix: Wrap handler execution in a `std::panic::catch_unwind` block so that panics can be caught at the dispatcher level. When a panic is caught, log the error and stack trace, then construct a fallback `500 Internal Server Error` response to return to the client instead of crashing. Many web frameworks use this approach [6] – it ensures that non-memory-fatal panics (e.g., an index out of bounds or unwrap on `None`) only affect the single request. Implementing this may involve adjusting the dispatcher's request processing loop (where it receives a `HandlerRequest` and invokes the registered handler). If using the `may` coroutine library, ensure that catching the panic inside the coroutine is sound (the `catch_unwind` must be called inside any thread or coroutine boundary that could unwind). This change will allow the server to keep running even if one handler has a bug, improving overall resilience.
Relevant Code/Commits: - The handler invocation occurs after routing a request. For example, in the test code, the `Dispatcher` calls `header_handler(req)` directly after routing [7]. Currently, there is no panic guard around such calls. We need to modify that section (likely in `src/dispatcher.rs` or wherever `Dispatcher` sends the request to the handler) to catch unwinding panics.
- Reference: Actix-Web's issue on catching handler panics shows that without recovery, the server just drops the connection on a panic [8]. Our goal is to implement a similar "CatchPanic" middleware or wrapper as other frameworks do.
- After implementing, test by triggering a panic in a handler and confirming the response is a 500 with an appropriate error message (and the server log contains the panic details).

Title: Fix Race Conditions in Multiple Security Provider Authentication
Labels: bug, security
Milestone/Epic: Improved Coroutine Safety
Summary: When an OpenAPI operation has multiple security requirements (e.g., an API key and a JWT token both required, or multiple alternative auth schemes), the current implementation does not handle them safely. There are potential race conditions or logic flaws when validating multiple security providers. For instance, if two authentication middlewares are added (to handle separate schemes), they may both attempt to modify the request context or headers concurrently, or the router might not properly short-circuit after one provider fails. This could lead to unpredictable authentication outcomes – such as one provider allowing a request while another is denying it – resulting in inconsistent security enforcement.
Steps to Reproduce: 1. Configure an endpoint in the OpenAPI spec with multiple security schemes (e.g., both an API Key in header and an OAuth2 bearer token).
2. Implement or enable two authentication middleware components (e.g., two `AuthMiddleware` instances with different checks) and attach them to the dispatcher.
3. Send requests with only one of the two required credentials or with conflicting credentials. Currently, you may observe that the router doesn't consistently reject unauthorized requests, or one auth check's state interferes with the other (e.g., one writes a user identity that the other then overrides). In a worst-case scenario, data races could occur if shared state is not protected.
Proposed Fix: Refactor the security provider handling so that multiple auth requirements are evaluated in a controlled sequence rather than in parallel. If the intention is that **all** listed security schemes are required (logical AND), the router should check each in turn and fail fast if any fail, preventing further handler execution. If the schemes are alternatives (logical OR), the router should succeed if any one provider succeeds (and skip or short-circuit the others). Implement synchronization to avoid data races: for example, if authentication middlewares need to store an authenticated user or claims in the request context, ensure this is done with thread-safe structures (or have the checks run on the same thread/coroutine sequentially). Additionally, provide clear rules for combination: perhaps update the `AuthMiddleware` to be aware of

multiple schemes or introduce a composite security middleware that coordinates multiple checks. By doing so, we eliminate conditions where two providers race or conflict.
Relevant Code/Commits: - The middleware registration in the README shows how multiple middlewares are added to the dispatcher [9], including `AuthMiddleware` for auth. However, support for multiple auth schemes might require multiple auth middlewares or a single middleware handling multiple config entries. We need to inspect the `AuthMiddleware` implementation (likely in `src/middleware/auth.rs`) to see how it validates the `Authorization` header or others.
- If the `AuthMiddleware` currently just checks a static token or single scheme, extending it to handle multiple schemes (or designing a new `CompositeAuthMiddleware`) might be needed. Also, ensure any global state (like a user context) is passed through the request in a safe manner (perhaps via the `HandlerRequest` or a thread-local) to avoid races.
- No specific commit directly references this, as it appears to be a design limitation. However, this issue aligns with the roadmap item to improve auth (e.g., "Auth (JWT, OAuth, etc.) - routed to Sesame-IDAM or similar" in planned features). Ensuring thread-safe, deterministic evaluation of multiple auth schemes is part of that improvement.

Title: Provide Configurable Coroutine Stack Size and Debug Instrumentation
Labels: enhancement, good first issue
Milestone/Epic: Improved Coroutine Safety
Summary: BRRTRouter uses stackful coroutines (via the `may` crate) for handler execution, with a default stack size of 0x4000 bytes (16KB) for each coroutine [10]. This low default can lead to stack overflow in deeply nested operations or JSON parsing, and currently, the only way to adjust it is by setting the `BRRTR_STACK_SIZE` environment variable. Additionally, developers have limited visibility into coroutine execution (e.g., how much stack is used, how many coroutines are active, etc.). Improving the configurability and observability of coroutines will make the system more robust and easier to debug when issues arise (such as memory corruption or overflow).
Steps to Reproduce: 1. Run a scenario with deeply nested JSON or large payloads without setting `BRRTR_STACK_SIZE`. In extreme cases, a handler that recurses or allocates large structures on the stack may trigger a stack overflow (resulting in a panic or abort).
2. Observe that increasing the stack requires restarting the process with an env var set. Also, there's no built-in way to get insight into coroutine usage (for example, no easy method to dump active coroutine count or stack utilization at runtime).
Proposed Fix: Introduce a programmatic way to configure the coroutine stack size, such as a setting on `Dispatcher` or `HttpServer` initialization (in addition to the env var). This would allow tests and embedded use to specify a larger stack without relying on global environment state. Consider raising the default stack size to a safer value (e.g., 64KB or 128KB) to handle typical request sizes, or document clearly when a larger stack is needed. For debug instrumentation, leverage the capabilities of the coroutine library or add hooks: for instance, track the maximum stack depth used by each coroutine (if `may` provides APIs for this) or periodically log the number of active coroutines and their states. Another idea is to integrate with `tracing` or logging – e.g., using a debug feature flag to have each coroutine log when it starts and ends, and if possible, how many bytes of stack were consumed (though precise measurement might require custom instrumentation in the coroutine assembly code). Even a simpler approach like providing a utility to dump all active coroutine backtraces (similar to `kotlinx-coroutines` debug tools) would help developers diagnose deadlocks or stuck handlers. Overall, these changes will improve developer confidence that coroutine scheduling issues can be detected and that stack overflow conditions are less likely in production.
Relevant Code/Commits: - The environment variable handling for `BRRTR_STACK_SIZE` is documented in

the README [10]. The code reading this env is likely in the initialization of the coroutine runtime (possibly in `main.rs` or wherever `may::config()` is done). We should update that code to also read from a config struct or builder pattern.
- A recent commit addressed test reliability by ensuring the stack size was set for tests, indicating this was a real issue (see commit message "ensure tests set stack size" related to SSE tests). This suggests that without a larger stack, certain features (like SSE or heavy JSON) would fail. We can refer to that as evidence to justify increasing the default or making it adaptive.
- For instrumentation, if using `may`, check if it has debug support or consider using `tracing` spans around important coroutine events. For example, wrap each handler coroutine in a span named after the route or operation, which would at least let us see active spans via `tracing` diagnostics. The `DebugProbe` mentioned in similar contexts (for async) doesn't exist for stackful coroutines, so we might implement our own simple tracking (e.g., increment a counter in `Dispatcher` when a coroutine starts and decrement when it ends, and expose this count via an API or log it periodically).

Developer Experience

Title: Implement Hot Reload on OpenAPI Spec Changes
Labels: enhancement, developer experience
Milestone/Epic: Developer Experience
Summary: Currently, if the OpenAPI specification (which drives all routes and handlers) is updated, the server must be restarted or manually re-run the code generation to pick up the changes. This hampers development velocity because tweaking an API spec requires rebuilding the project. A "hot reload" feature would detect changes to the OpenAPI YAML/JSON file at runtime and automatically update the routing table and generated handler stubs without a full restart. This is especially useful during development, allowing developers to see changes in real-time. Implementing hot reload will significantly improve the developer experience by shortening the feedback loop when designing or modifying APIs.
Steps to Reproduce: 1. Run BRRTRouter with a given `openapi.yaml` spec file.
2. Modify a route or schema in the `openapi.yaml` while the server is running.
3. Notice that the server does not pick up this change – the new route or modified schema has no effect until the server is restarted and the code regeneration (if any) is re-run. In some cases, developers might forget to re-run the codegen, leading to discrepancies between the spec and running code.
Proposed Fix: Introduce a file watcher on the OpenAPI spec file (or the directory) that triggers a reload process. The reload can work in one of two ways: **(a) Dynamic re-registration** – if the architecture allows, parse the updated spec and update the in-memory routes (e.g., add new routes to the router, remove or modify changed routes, update the dispatcher's handler mappings). **(b) Automated codegen + reload** – invoke the code generation pipeline when a change is detected, compile the new code (possibly as a dynamic library or by restarting an internal process), and update the server. Approach (a) is more live, but requires the router to be able to alter routes at runtime; approach (b) might leverage `dynamic reload` by spawning a separate process or thread to handle compilation. Given Rust's compile-time nature, a pragmatic initial implementation might lean on dynamic dispatch: e.g., have a generic handler that calls out to an updated script or use `rlua` / `dylib` for dynamic code – but this adds complexity. A simpler route is to support hot reload in **development mode**: detect the file change, gracefully shut down the server (finish current requests), then restart it (fork a new process or re-init in the same process) with the new spec. This can be facilitated by a CLI flag (`--watch`) during development. Ensure that during reload, there's no connection leak or double-binding of ports. The goal is that within a second or two of saving the spec file, the server is serving the updated API. This was mentioned as a key planned feature by the project maintainer [11]. We will also need to handle error cases (e.g., if the spec is edited to an invalid state, we

should log the error and possibly continue running the old spec until a valid update is provided).
Relevant Code/Commits: - The OpenAPI spec parsing is handled at startup (likely in `main.rs` or a builder), using the internal parser. To implement reload, we may refactor this into a function that can be called on demand to re-init the `Router` and `Dispatcher`.
- The code generation pipeline (if using `codex` or similar) might be invoked via a `just` script currently. For hot reload, see if we can integrate that generation step into the running program (perhaps behind a feature flag).
- The forum post by the author explicitly outlines the plan for hot reload and dynamic updates [11]. This confirms that after generating typed handlers and structs, the next step is to support "code gen on any update to the OpenAPI spec" with live reload.
- No direct commits yet since this is a future feature, but architecture changes (like supporting `Dispatcher` to add/remove routes at runtime) may be required. We might consider using an underlying data structure that can be updated (the router's regex set, etc.), or plan to rebuild those on reload.

Title: Enhance Code Generation for Typed Handlers and Models
Labels: enhancement, codex, developer experience
Milestone/Epic: Developer Experience
Summary: The project currently generates some code from the OpenAPI spec (e.g., route matching logic and an `echo_handler` for testing), but further improvements are needed to produce fully-typed handler stubs and data models from the spec. In the ideal scenario, developers should not have to manually write boilerplate for extracting request parameters or body data – the codegen should create Rust structs for request bodies/schemas and function signatures for handlers that take these structs. For instance, if the OpenAPI spec defines a schema for `Pet` and an operation to add a pet, the codegen should create a Rust struct `Pet` and a handler function signature like `fn add_pet(pet: Pet) -> Result<Pet, Error>` (or similar), instead of the current generic `HandlerRequest`. By improving code generation in this way, we reduce manual coding and potential mistakes, and ensure that the implementation stays in sync with the spec (especially as the spec evolves).
Steps to Reproduce: 1. Define a new schema or endpoint in the OpenAPI spec (for example, a new resource with certain properties).
2. Run the current code generation (if any) or observe the current `Dispatcher` usage. Today, developers likely get a generic handler call where they must manually extract fields from `HandlerRequest` (which contains raw maps for headers, query, JSON, etc.).
3. This process is error-prone and requires writing parsing logic by hand for each new endpoint. If the spec changes (say a field is added), the developer must update the extraction code accordingly.
Proposed Fix: Expand the internal code generation tool (possibly the component labeled "codex") to output strong types and conversion implementations. Concretely, generate Rust `struct` definitions for every schema object in the OpenAPI spec (using a tool or custom codegen) and `enum` types for any OpenAPI `oneOf` / `anyOf` constructs where applicable. Then, for each operation, generate a function signature stub that takes typed parameters. For example, for an operation with an `application/json` request body of schema `Pet`, generate `fn handler_name(request: Pet) -> HandlerResponseType { ... }`. Internally, have the router automatically deserialize the incoming JSON into the `Pet` struct (using `serde` with the schema-derived struct) before calling the handler. This will likely involve implementing `From<HandlerRequest> for TypedRequest<T>` as noted in the roadmap [12] – essentially an automatic conversion from the generic request to a specific struct, which was planned (" Auto-generation of impl From<HandlerRequest> for TypedHandlerRequest<T> based on schema"). Also generate response types or utilize existing ones to serialize handler outputs. We might use a build script or `proc_macro` to achieve this generation at compile time. If using an external OpenAPI codegen library is feasible, that's an

option too (e.g., OpenAPI Generator can generate Rust types, but integrating it might be complex – a custom solution might be simpler for our use case). The deliverable is that developers can write minimal code: just the business logic in the handler, with the framework handling parsing/validation of inputs and formatting of outputs.
Relevant Code/Commits: - The README "Contributing" section lists this feature as a priority [12], confirming the need for typed handler deserialization and codegen of `From<HandlerRequest>` implementations.
- Check `src/dispatcher.rs` or codegen templates for how handlers are currently invoked. For instance, the test code registers handlers by name and uses a generic `HandlerRequest` object containing `headers`, `cookies`, etc. [13] [14]. We will extend this by generating specific request types.
- The forum mention [15] indicates that basic route and handler codegen was recently added, and the next steps include generating typed handlers and structs. We should inspect the repository's `templates/` or codegen logic (perhaps under `src/codegen` or similar) to build upon it. Possibly, the project might have a `build.rs` that reads the OpenAPI spec and outputs Rust code – if so, we'd enhance that.
- After implementation, verify by regenerating code for the Pet Store example: the handlers in the example should now accept concrete types (e.g., `NewPet` struct) rather than using the generic request object.

Title: Support Dynamic Route Registration at Runtime
Labels: enhancement
Milestone/Epic: Developer Experience
Summary: At present, routes are mainly established at server startup by parsing the OpenAPI spec and (possibly) through generated code. If the spec is extended or modified, adding a new route requires re-running codegen and restarting. Supporting dynamic route registration means the `Router`/`Dispatcher` could accept new routes while the server is running (or at least without a full recompilation). This is closely related to hot reload, but even beyond file-watching, it implies an API for programmatically adding routes. For example, an advanced use-case: a plugin system could define new endpoints and register them on the fly. Achieving dynamic route registration will increase flexibility and pave the way for more advanced runtime behaviors (like multi-tenancy or runtime feature toggles of endpoints).
Steps to Reproduce: 1. Try to add a new endpoint to the router after the server has started (currently no public API exists for this).
2. Realize that without restarting, you cannot serve a new path that wasn't in the initial OpenAPI spec.
3. This static nature means the router can't respond to configuration changes or user-defined extensions at runtime.
Proposed Fix: Refactor the router's internal structures to allow insertion and removal of routes at runtime. The `Router` likely uses a compiled regex or trie of paths – provide methods like `Router::add_route(method, path, handler_name)` that updates these structures. Similarly, extend `Dispatcher` to accept new handler function mappings via a safe API (non-unsafe, unlike the current `register_handler` which is marked `unsafe` due to static lifetime issues [16]). Manage synchronization (e.g., use a write lock or ensure single-threaded context) to avoid conflicts if a request comes in while routes are being updated. This dynamic registration ties in with hot reload: on spec file change, we could call these new APIs to update routes instead of full restart. It also means we need to handle potential conflicts (e.g., if a route being added already exists). For initial implementation, it's acceptable to require that dynamic additions happen when the server is in an idle or admin state (to simplify thread safety), but eventually it should be possible on the fly. The milestone for this is likely tied to version 1.0, ensuring that BRRTRouter isn't limited by compile-time code generation.
Relevant Code/Commits: - The roadmap explicitly lists "Dynamic dispatcher route registration" as a desired feature [17]. This indicates it's known and planned.

- Current code generation likely produces a static list of routes passed to `Router::new(vec![...])` at startup (for example, in tests they do `Router::new(vec![route])` with a hardcoded vector [18] [19]). We need to augment `Router` with methods to modify that internal vector or data structure. The internal implementation (possibly a regex set or a tree) needs to support rebuild or incremental update. We should inspect `src/router.rs` or wherever `RouteMeta` is used.
- The `Dispatcher` uses an internal mapping of handler names to function pointers (registered via `register_handler`). We might consider removing the `unsafe` requirement by managing handler lifetimes differently (perhaps storing them in a `Box` or using a `'static` requirement but ensuring the functions we register are static anyway in practice). Making `register_handler` safe and usable at runtime is part of this improvement.
- No direct commit for this yet, but our changes will likely touch core files (router, dispatcher, maybe the codegen template that currently generates static registrations).

Title: Improve Coroutine Handler Ergonomics for Developers
Labels: enhancement, developer experience
Milestone/Epic: Developer Experience
Summary: Writing handler functions in BRRTRouter currently requires understanding of how the coroutine system passes requests and responses. For example, handlers receive a `HandlerRequest` and must populate a `HandlerResponse` and send it via a channel (as seen in the echo and header handler examples) [13] [20]. This low-level approach works, but it's cumbersome for developers used to writing a simple function that returns a response. The ergonomics can be improved by offering a more natural API to implement handlers. For instance, allowing handlers to simply return a result or use the Rust `async`/`await` syntax which then bridges into the coroutine, or at least providing macros or wrappers to hide the channel send logic. Smoother ergonomics will lower the learning curve and reduce boilerplate in user code, making BRRTRouter more appealing to contributors and users.
Steps to Reproduce: 1. Write a new handler for an endpoint using the current system. You must define a function that takes `HandlerRequest`, constructs a `HandlerResponse`, and manually sends it back on `req.reply_tx`.
2. Forgetting to send on the channel or setting the response status/body incorrectly can lead to no response or errors, and there's no compiler enforcement (easy to have a code path that never sends a reply, causing a hang).
3. Also, because handlers are not standard Rust `async fn` (they run in a separate coroutine system), using async libraries or `.await` inside them is not straightforward.
Proposed Fix: Provide a higher-level handler API. One approach is to implement a trait for handlers, e.g., `trait HandlerFunc<Req, Res> { fn call(req: Req) -> Res }`, and provide implementations for functions that match the signature. Then the router can accept closures or function pointers that simply return a `HandlerResponse` (or a type convertible to one, like the actual response body and status). Internally, the router would take that return and send it over the channel, abstracting away the `reply_tx`. Another approach: create a macro attribute (like `#[brrr_handler]`) that a developer can tag on an async function, and expand it to the necessary glue code to interface with the coroutine system. This macro could handle transforming an `async fn` (which returns a `Future`) into a form that can be executed on the coroutine runtime (perhaps by blocking on it within the coroutine, or using a small executor bridge – see "Async bridging" issue). At minimum, document and simplify the pattern of obtaining request data and sending responses. Perhaps provide utility methods on `HandlerRequest` like `req.json<T>() -> T` to get a JSON body as a type, and on `HandlerResponse` a constructor like `HandlerResponse::json(status_code, &value)` to serialize a response. These would reduce boilerplate in each handler. The end goal is that writing a handler for BRRTRouter should feel similar to

writing one in frameworks like Actix or Axum – focusing on business logic rather than manual wiring.
Relevant Code/Commits: - The example in the README's Handler Registration [21] and the test handlers [13] [22] show the current style. The handlers are free functions and must perform channel sends. We can improve this by encapsulating the channel usage inside the framework.
- No specific commit for this yet, but this is a general design enhancement. It aligns with making the router "friendly to coroutine runtimes" and ergonomic, as stated in the Vision [23].
- We may utilize Rust's standard library `Fn` traits or a custom abstraction to allow both synchronous and asynchronous code to be used as a handler. This change might involve significant refactoring of how the dispatcher calls handlers (possibly introducing trait objects for handlers).
- Testing this improvement would involve converting existing example handlers (like `echo_handler`) to the new style and ensuring all tests still pass, indicating backward compatibility or successful transition.

Title: Integrate Structured Tracing for Requests (OpenTelemetry)
Labels: enhancement, good first issue
Milestone/Epic: Developer Experience
Summary: While BRRTRouter has a basic `TracingMiddleware` that creates spans for each request [24], this can be extended to provide richer observability. Developers and operators would benefit from integration with standardized tracing systems (such as OpenTelemetry), so that request flows can be observed and correlated across services. Currently, it's unclear if the spans are logged or exported anywhere; improving this means configuring the `TracingMiddleware` to use the `tracing` crate's facilities to record info about each request (like method, path, status, duration) and optionally propagate tracing context. This will help in debugging and monitoring, especially in distributed environments.
Steps to Reproduce: 1. Enable the existing `TracingMiddleware` in the dispatcher (as shown in README).
2. Run the server and make a few requests. Currently, unless a tracing subscriber is set up by the user, these spans might not output anything visible. There's no integration with an exporter out of the box.
3. In a real-world scenario, you'd want these traces to be sent to a tracing backend or at least logged to console with request IDs. At present, one has to manually instrument the code further to achieve that.
Proposed Fix: Enhance the `TracingMiddleware` implementation to record key events and to integrate with `tracing::info_span` or `tracing::instrument`. For example, when a request starts, create a span with attributes like `method`, `path`, maybe a correlation ID if present. On completion of the request (either after handler response or on error), record the status code and duration. If OpenTelemetry is enabled (behind a feature flag), use the OpenTelemetry crate to create a span and export it (this could tie into the `tracing-opentelemetry` layer). Additionally, ensure that if an incoming request has tracing headers (e.g., `traceparent`), the middleware can extract that and continue the trace. Provide configuration to enable these features conditionally so as not to impose overhead if not needed. This might involve adding dependencies on `tracing-subscriber` or `opentelemetry` as optional features. The outcome is that with minimal setup, a developer can get structured logs or distributed trace data for each request. Documentation should also be updated to show how to initialize a tracer (for example, instruct developers to use `tracing_subscriber::fmt()` in `main` for console logs, or how to set up an OTLP exporter).
Relevant Code/Commits: - The `TracingMiddleware` is mentioned in the README with a brief description [24]. We should inspect its code (likely in `src/middleware/tracing.rs`) to see what it currently does. It probably uses the `tracing` crate's span internally. If so, hooking it up to a subscriber is the next step. If not, we will implement it accordingly.
- This task can leverage external references: e.g., Tower HTTP's `TraceLayer` or actix-web's tracing integration as inspiration. But given the context, it's likely straightforward: use `tracing::Span::current()` or create a new span per request.

- There might not be existing issues or commits specifically for this, as it's an enhancement. Testing the improvement would involve running the server with an example subscriber and verifying that spans are created and include the expected fields.

Title: Add Request Metrics Middleware (Prometheus-compatible)
Labels: enhancement, good first issue
Milestone/Epic: Developer Experience
Summary: Monitoring server performance and usage is critical. The router currently has a `MetricsMiddleware` that tracks request counts and average latency in-memory [24], but it may not expose these metrics in a standard way. Enhancing this middleware to integrate with a metrics system (like Prometheus) would allow developers to scrape metrics and get insights such as requests per second, latency distributions, and error rates. If the current implementation only keeps an average latency, it could be improved to track more granular data (e.g., histogram of latencies) and to reset statistics periodically or report them on-demand. Overall, a more comprehensive metrics integration will help in production observability.
Steps to Reproduce: 1. Enable `MetricsMiddleware` in the dispatcher.
2. Run the server and make some requests. Currently, there is no endpoint to retrieve metrics, and the data (count, avg) is likely only in memory, accessible perhaps via logging or not at all.
3. In a real deployment, you'd want to see metrics like total requests, requests broken down by path or status, etc., which isn't readily available now.
Proposed Fix: Integrate a metrics library (such as `prometheus` crate or `metrics` with exporters) into the `MetricsMiddleware`. For example, define global or lazy static counters and histograms: an HTTP request counter labeled by method and status, a histogram for request durations, etc. Each time a request completes, the middleware records these metrics. Then, provide an endpoint (e.g., `/metrics` if in standalone mode) that exposes the Prometheus metrics format so they can be scraped. If adding an endpoint within BRRTRouter is not desirable for core, at least make the metrics accessible to the host application (perhaps by returning references to the metrics objects or by using a global registry). Another approach is to integrate with the community-driven `metrics` crate, where users can install a Prometheus recorder; the middleware would simply call `metrics::increment_counter!("requests_total", ...)` with appropriate labels. Also consider tracking metrics per route (which could be high-cardinality, so maybe per operationId name) to identify hotspots. Since the current middleware mentions "average latency," we should replace that with a more robust measure (e.g., histogram or moving average, since simple average can be misleading over time). This enhancement will give users immediate insight into the performance of their API.
Relevant Code/Commits: - The README notes what `MetricsMiddleware` is supposed to do [24]. We need to check its implementation (likely in `src/middleware/metrics.rs`). It probably accumulates a count and total time, then computes average. We plan to refactor that.
- Prometheus integration might involve adding the `prometheus` crate. If we do, ensure it's behind a feature flag to keep the base lightweight. Example: `prometheus::Registry` usage to register counters/histograms for our metrics.
- There is no prior issue on this yet; it's a straightforward extension. We should test by hitting some endpoints and then querying the metrics endpoint to see if metrics reflect the calls. Possibly provide a basic `/metrics` handler as an opt-in feature (which could simply call `prometheus::gather()` and output text).
- This task is a good candidate for new contributors (labeled `good first issue`) because it's relatively isolated: it involves familiarizing with a metrics library and modifying an existing middleware.

Title: Add Built-in CORS Support (Cross-Origin Resource Sharing)
Labels: enhancement
Milestone/Epic: Developer Experience
Summary: Many web APIs need to support CORS so that browsers can call them from front-end applications. While a `CorsMiddleware` is mentioned in the project [25], it likely adds some default `Access-Control-Allow-*` headers. However, it may not be fully configurable (e.g., to specify allowed origins, headers, or credentials). A robust CORS implementation should allow the user to configure the policy according to their needs and properly handle preflight (`OPTIONS`) requests. By improving the CORS support, we make BRRTRouter ready for use in real web scenarios where front-end clients are served from different origins.
Steps to Reproduce: 1. Enable `CorsMiddleware` in the dispatcher. (The README example suggests it simply adds CORS headers unconditionally.)
2. Run the server and make a cross-origin request from a browser or use `curl -H "Origin: http://example.com"` to see what CORS headers are returned.
3. If the current implementation is basic, it might allow all origins or have some fixed behavior, which might not suit all use cases (e.g., not allowing credentials or custom headers properly).
Proposed Fix: Expand the `CorsMiddleware` to be configurable. It could accept a configuration struct or builder that allows setting: allowed origins (either `*` or a list of domains), allowed methods, allowed headers, whether to allow credentials, and maximum age for preflight caching. Implement logic so that for an incoming request, if it's a preflight (OPTIONS request with `Origin` and `Access-Control-Request-Method` headers), the middleware short-circuits and responds with the appropriate CORS headers (and no further handler processing). For normal requests, the middleware should add the `Access-Control-Allow-Origin` (echoing the request's `Origin` if allowed or `*` if open) and related headers (`Access-Control-Allow-Credentials` if enabled, `Access-Control-Expose-Headers` if any). Perhaps use an existing crate like `cors` as inspiration or just follow the MDN guidelines. Ensure that if multiple middleware are stacked, the CORS one runs at the appropriate phase (likely as a "before" and maybe "after" wrapper around handlers). After implementing, document how to configure CORS in BRRTRouter (maybe via the `AuthMiddleware::new()` pattern or a config file). This will save users the trouble of writing their own CORS logic and avoid common pitfalls.
Relevant Code/Commits: - The `CorsMiddleware` is demonstrated in use [25], but we need to see its actual implementation (likely in `src/middleware/cors.rs`). It might currently be a placeholder that adds `Access-Control-Allow-Origin: *` always. We will extend this.
- No specific commit references this, but adding comprehensive CORS was likely expected as part of making the router production-ready. - Testing will involve sending various CORS scenarios: simple GET with Origin, preflight OPTIONS, requests with custom headers, etc., and verifying the responses contain correct CORS headers as configured. We should also ensure that the middleware does not interfere with non-CORS requests.

Release 1.0 Readiness

Title: Implement OpenAPI Schema Validation for Requests and Responses
Labels: enhancement, OpenAPI, validation
Milestone/Epic: Release 1.0
Summary: One of the key advantages of driving the router by an OpenAPI spec is the ability to automatically validate incoming requests (and even outgoing responses) against the schema. Currently, the router parses and deserializes request bodies and parameters, but it doesn't enforce all schema constraints defined in the OpenAPI (like string length limits, numeric ranges, required properties, formats, etc.).

Implementing validation means that if a request JSON body or parameters do not conform to the OpenAPI schema (for example, a required field is missing, or a string is too long), the router should reject the request with a 400 Bad Request and a message about the violation. Similarly, it can validate responses produced by handlers to ensure the server isn't returning something that contradicts the spec (this could be a development-time check or optional feature in production). Adding this feature will greatly improve reliability and help catch errors early, ensuring spec compliance.
Steps to Reproduce: 1. Send a request that violates the OpenAPI spec to an endpoint (e.g., omit a required field in the JSON body, or use an invalid value for an enum parameter).
2. Currently, the handler will receive the deserialized data regardless (missing required fields might be `null` or defaulted), and likely proceed without noticing the issue, potentially causing an error later or storing bad data. The router does not currently generate a validation error on its own.
3. The expected behavior after this improvement is that the request would be rejected immediately with an error response explaining the validation failure.
Proposed Fix: Utilize a JSON Schema validator or custom validation logic tied to the OpenAPI spec. OpenAPI 3.1 schemas are JSON Schema compliant, so we could integrate a crate like `jsonschema` or `schemars` to validate incoming JSON against the schema definitions. Concretely, when an operation has a requestBody with a schema, after deserializing it into a `serde_json::Value` (or into a typed struct), run it through a validator with the OpenAPI schema (which we have parsed). If validation fails, return an HTTP 400 with details. Similarly, for query parameters and headers, enforce types and formats (OpenAPI parameters have `schema` or `content` definitions we can validate). For response validation, we can optionally (perhaps in debug mode or test mode) validate the `HandlerResponse` body against the response schema defined for that endpoint, logging or asserting mismatches. This ensures our server truly adheres to the contract. To implement, leverage the existing parsed spec structures: the project likely has a representation of components schemas and a function to resolve `$ref` (as seen in tests for `resolve_schema_ref`) [26]. Use those to feed into a validator. If using `jsonschema` crate, we can convert OpenAPI Schema object to a generic JSON Schema `Value` and validate. Performance-wise, this adds overhead to each request, so consider making it togglable or optimize by compiling schemas once at startup. In sum, this feature will make BRRTRouter act as a gatekeeper that ensures clients and server adhere to the API contract strictly.
Relevant Code/Commits: - The test commit for spec helper functions shows we can resolve schema references [26]. Building on that, we have the pieces to retrieve the correct schema for a given request.
- The README "Contributing" section hints at "spec validation" as a goal (under test coverage and spec validation [27]). This aligns with implementing runtime validation.
- We should examine how the OpenAPI spec is represented in code (likely a struct `OpenApiV3Spec` and model structs for Schema, etc.). The `docs/ROADMAP.md` or existing parser logic might already have partial support for things like format checks. It will be useful to reuse any codegen or parsing work already done for validation (perhaps the types generated for codegen can have validation annotations or functions).
- After implementing request validation, test with various invalid inputs to ensure the router responds with appropriate errors and that valid requests still pass through unchanged.

Title: Add WebSocket Support per OpenAPI Spec Extension
Labels: enhancement, protocol
Milestone/Epic: Release 1.0
Summary: Beyond standard HTTP request/response, some real-time APIs require WebSocket support. The OpenAPI specification doesn't natively describe websockets, but BRRTRouter could handle them via an extension or a separate code path. In fact, Server-Sent Events (SSE) are already supported via an `x-sse: true` extension on GET endpoints [28]. Similarly, we want to support WebSockets, perhaps through an `x-ws: true` extension or by recognizing when the Upgrade header is present. WebSocket support means

the router can upgrade an HTTP connection to a long-lived WebSocket connection and then route messages (frames) to a handler. Implementing this will broaden the use cases for BRRTRouter (e.g., real-time notifications, chat services, etc.) and make it a more complete alternative to frameworks like FastAPI (which also supports WebSockets).
Steps to Reproduce: 1. Currently, try to define a WebSocket endpoint. There's no official way in OpenAPI to do so, but one might attempt to use `x-ws: true` similar to SSE.
2. Run the server and attempt a WebSocket handshake (an HTTP GET with `Upgrade: websocket` header) to the defined route.
3. At present, BRRTRouter likely does not distinguish this from a normal GET, so it would either treat it as a normal request (and not upgrade) or just not support it at all.
Proposed Fix: Introduce a convention to declare WebSocket endpoints in the spec (for example, an `x-websocket: true` extension on a GET path, or perhaps using a special scheme like `ws://` in the servers URL). When the router encounters a request with `Upgrade: websocket`, it should perform the WebSocket handshake (respond with 101 Switching Protocols) and then hand off control to a WebSocket handler loop. We'd need to integrate a WebSocket library (e.g., tungstenite for the WebSocket protocol framing, or use `may_minihttp` if it has WebSocket support). The handler for a websocket could be a different type of handler – likely an async loop that receives messages and can send messages back. Perhaps define a trait like `WebSocketHandler` with callbacks for on_message, on_open, on_close. For now, even a basic echo or broadcast capability would be a start. Manage these connections within the router so that they don't block the main request loop (each WebSocket can run in its own coroutine). This feature will require careful resource management: tracking open sockets, closing them on server shutdown, etc. For OpenAPI integration, document how to specify a WS endpoint (even though it's outside official OpenAPI spec, many projects use extensions for this). Ensure that normal HTTP and SSE paths continue to work as before.
Relevant Code/Commits: - SSE support is implemented by marking routes with `x-sse: true` and using `brrrouter::sse::channel()` in handlers [28]. We can parallel this approach for websockets. Check if there's any placeholder or mention of WebSocket in the code or roadmap. The README's list of contributions explicitly lists "WebSocket support" as a desired feature [29] – meaning it's acknowledged but not done.
- We will likely need to use a crate like `tungstenite` or `tokio-tungstenite` (if we bridge to async) or perhaps `may` has something for websockets. Research needed to pick an implementation that fits our coroutine model (maybe using standard library's low-level socket after upgrading).
- After implementing, test by writing a simple WebSocket client (or use `wscat`) to connect to a BRRTRouter endpoint and exchange messages. We should be able to handle at least text frames (and possibly binary) and ensure no memory leaks when clients disconnect.

Title: Integrate JWT/OAuth2 Authentication Support
Labels: enhancement, security
Milestone/Epic: Release 1.0
Summary: Secure APIs often require JWT (JSON Web Tokens) or OAuth2 access tokens. While BRRTRouter's `AuthMiddleware` currently performs a "simple header token check" [30] (likely checking for a static token or basic auth), it doesn't have full JWT validation or OAuth2 integration. Adding first-class support for JWT/OAuth2 means the router can parse and validate a JWT from the `Authorization: Bearer <token>` header: verifying its signature, checking expiration (`exp` claim), and possibly scopes/claims for authorization. This would allow BRRTRouter to serve as an authentication gateway out-of-the-box for protected endpoints. For OAuth2, integration might involve introspecting tokens or integrating with an external IDP (Identity Provider) or using JWKS (JSON Web Key Sets) for signature verification. This is a

substantial feature but crucial for real-world API use where security is needed.
Steps to Reproduce: 1. Configure `AuthMiddleware::new("Bearer secret")` as shown in README. This likely only checks if the `Authorization` header exactly matches "Bearer secret" and allows or denies accordingly.
2. Try using a real JWT instead – the current middleware will not decode or validate the JWT's signature, so it either rejects it (if expecting the literal "secret") or allows any token if not properly implemented. In either case, fine-grained auth (like checking token expiry or claims) is not happening.
3. This is insufficient for most use cases where tokens are dynamic and signed (e.g., by an OAuth2 server).
Proposed Fix: Upgrade the `AuthMiddleware` to handle JWT verification. This could be done by allowing the middleware to be configured with a JWKS endpoint or a public key (for HS256 shared secrets or RS256 public keys, etc.). The middleware would, for each request, parse the `Authorization` Bearer token, decode the JWT (using a crate like `jsonwebtoken` or `jsonwebtoken-lite`), and validate its signature and claims. Provide configuration options for expected issuer, audience, allowed algorithms, etc. If the token is invalid or missing, the middleware should short-circuit and return 401 Unauthorized. We can also map certain claims to a user identity in the request context (for use by handlers). For OAuth2 integration beyond JWT, perhaps provide hooks to call an introspection endpoint, but that might be out of scope for core (JWTs cover common cases of stateless auth). Additionally, support multiple auth strategies via the security definitions (which ties back to the multi-provider race condition issue). Possibly introduce an `OAuthMiddleware` variant for opaque tokens that calls an introspection URL. Aim to comply with common patterns so that integrating with providers like Auth0, Okta, or custom IAM is straightforward. Include thorough documentation on how to enable JWT auth (e.g., providing the secret or JWKS URL via env or config). This will elevate BRRTRouter to production-grade in terms of security.
Relevant Code/Commits: - The current `AuthMiddleware` implementation (likely in `src/middleware/auth.rs`) needs to be examined. It probably just checks a header contains a token matching a stored string. We will replace or extend that logic.
- We may introduce new dependencies: e.g., `jsonwebtoken` crate for HS/RS256 verification. Ensure to handle errors from token parsing (expired, invalid signature, etc.). Possibly provide caching for JWKS keys to avoid fetching them too often (if using RS256 and a JWKS URL).
- No existing commit specifically addresses JWT, but the roadmap's mention of "Auth (JWT, OAuth, etc.) – routed to Sesame-IDAM or similar" [31] indicates the vision to integrate with an identity management system. Our implementation can be a step towards that by making the middleware extensible (e.g., allow plugging custom validators, so someone could route to an external IDAM as needed).
- After implementing, test with real JWTs: generate a JWT with a known secret or RSA key, send requests with and without valid tokens, and ensure the middleware permits or blocks appropriately. Also test edge cases like expired token.

Title: Optimize Routing Performance to Achieve 1M req/sec Goal
Labels: enhancement, performance
Milestone/Epic: Release 1.0
Summary: A core promise of BRRTRouter is high performance routing – the vision statement targets "1 million route matches per second on a single-core Raspberry Pi 5" [32]. To reach this goal, we should identify and optimize any bottlenecks in the request routing path. This includes the path matching algorithm, parameter extraction, and dispatch mechanism. As development has added features (middleware, validation, etc.), it's important to ensure the critical path is as lightweight as possible for production builds. This issue is about creating benchmarks, measuring current throughput, and making targeted optimizations (or adjustments like feature gating debug code) to meet performance targets before the 1.0 release.

Steps to Reproduce: 1. Use the provided benchmarking tools (`just bench` which runs Criterion benchmarks [33]) on a system similar to the target (if available) or any modern machine.
2. Observe the current requests per second the router can handle for basic routes (likely there is a benchmark for matching). Let's assume currently it's below the 1M req/sec mark (especially if debug assertions or unoptimized regex are in use).
3. Identify components where latency exists – e.g., regex matching overhead, channel communication cost for each request, copying of data, etc.
Proposed Fix: Undertake a profiling and optimization cycle. Potential areas to look at:
- **Routing Algorithm**: If using regexes for path matching (as indicated by `Compiles OpenAPI paths into regex matchers` [34]), ensure they are pre-compiled and reused. Regex can be fast, but consider using a faster path matcher (like a trie or segment-wise match) if regex is a bottleneck. Alternatively, use Rust's fancy regex set optimizations if not already.
- **Parameter Extraction**: Make sure we're not allocating excessively when extracting path or query parameters. Perhaps reuse buffers or pre-allocate the structures.
- **Coroutine Yield/Resume Overhead**: The use of `may` coroutines should be quite efficient, but profile if switching contexts frequently is costly. Possibly tune the coroutine stack size (smaller stack is faster to switch) – though we balanced this with safety in another issue.
- **Channel Communication**: The dispatcher uses channels to send the response back from handler [35]. Check if this is a lock-free channel and if it's adding latency. If so, explore alternatives (maybe direct function call for simple handlers in the same thread, or use a thread-local channel).
- **Feature Flags**: Ensure debug features (like validation, heavy logging) can be disabled in production builds to not impact throughput. We might make runtime validation optional or very efficient.
- **Benchmark Harness**: Expand the Criterion benchmarks to cover more scenarios (deeply nested routes, varied parameter counts) to detect any slowness in those cases. Use flamegraph or similar tools to pinpoint hotspots.
After optimizations, update the documentation or release notes with achieved performance numbers. Hitting the 1M req/sec on Pi 5 (or extrapolating from a PC to that performance) would be a huge selling point for the 1.0 release, so we should iterate until we're close.
Relevant Code/Commits: - The `benches/` directory likely contains the benchmarking code. We should look at `benches/router_bench.rs` (for example) to see current performance. Perhaps commit history has notes on performance tweaks (like vectorization or avoiding certain std ops).
- The README's performance goal [32] sets a clear target. Also, it indicates excluding handler execution cost, so our focus is purely on routing overhead.
- If a particular regex crate or method is used, consider alternatives: e.g., if using `regex` crate, ensure to use `RegexSet` or `Aho-Corasick` for multiple patterns. If using custom matching, ensure it's optimized in Rust (maybe replace some parts with direct byte matching for static segments).
- No one commit to cite; this is an ongoing effort. We should document any changes (like "Optimized route matching by doing X, yielding Y% throughput improvement in benchmarks"). Possibly open separate sub-issues or PRs for each micro-optimization.

Title: Complete Documentation and Prepare Crate for Publication
Labels: documentation, enhancement
Milestone/Epic: Release 1.0
Summary: Before releasing version 1.0, we need to ensure that documentation is comprehensive and that the crate meets publishing criteria. This includes a complete user guide (how to use BRRTRouter in a project, how to configure it, examples of advanced features), generated API docs (with `cargo doc` coverage for all public items), and possibly a tutorial or doc site. Additionally, the repository should be tidied

up: all examples working, CI passing, and maybe choose a license confirmation and code of conduct (already present). Publishing to crates.io requires a unique package name (likely `brrrouter`) and removing any path dependencies or unpublished crates. The goal is that a new user can add `brrrouter = "1.0"` to their `Cargo.toml` and follow docs to get started easily.
Steps to Reproduce: 1. Generate documentation using `cargo doc`. Check for any missing docs or items with `#[doc(hidden)]` that should be public.
2. Review the existing README and docs for completeness. Are all major features described? Are there usage examples for common tasks (routing, middleware, error handling, etc.)?
3. Attempt to publish (dry-run with `cargo publish --dry-run`). This can catch issues like missing files in package, or references to paths that won't exist.
Proposed Fix: Break this into a few tasks:
- **User Guide/Doc Book**: Perhaps under `docs/` or as a GitHub Pages site, create a multi-section guide. The maintainer mentioned "doc book" in the forum [15]. Finish writing that: sections might include Introduction, Quickstart, Defining Routes via OpenAPI, Writing Handlers, Using Middleware (with subtopics for Auth, CORS, etc.), Testing and Debugging, Performance Tuning, and Advanced Topics (like adding protocols or integrating with Photon).
- **API Reference**: Ensure every public struct, trait, and function has a clear rustdoc comment. Include examples in doc comments for important APIs (for example, show how to register a handler, how to broadcast an SSE, etc., right in the rustdoc).
- **Examples**: Verify that the provided example (Pet Store example) works out-of-the-box. Possibly add more examples (like a minimal Hello World, an example with JWT auth, etc.). These can live in the `examples/` directory and be referenced in documentation.
- **Crate Metadata**: Update `Cargo.toml` with proper package metadata: description, homepage or repository link, categories (e.g., web-programming, async), keywords (router, OpenAPI, etc.), authors. Ensure license is specified (Apache-2.0 as per LICENSE file).
- **Publishing Checks**: Remove or adjust any path dependencies (maybe the project has none, but if Photon is referenced, ensure it's optional or published). Bump version to 1.0.0.
- **Changelog**: Consider adding a changelog for 1.0 summarizing major changes and improvements from pre-1.0.
Completing these steps will polish the project for public release, making it attractive and easy to adopt.
Relevant Code/Commits: - Check `docs/ROADMAP.md` (if any) to see what documentation was planned or is incomplete. The README already provides a lot of info (vision, usage, etc.), which can be expanded into the user guide.
- The forum discussion [15] indicates docs would be completed around the time of crate publishing. We should follow through on that plan.
- Ensure that issues like this one (all the improvements) are closed or referenced in the changelog so that users know what's been fixed by 1.0.
- After changes, do a final `cargo publish --dry-run` and possibly actually publish a 1.0.0-rc or beta to crates.io for testing integration in a fresh project.

Future Roadmap

Title: Plan Photon Framework Integration with BRRTRouter
Labels: epic, enhancement, ecosystem
Milestone/Epic: Future Ideas
Summary: Photon is a higher-level web framework intended to be built on top of BRRTRouter [36]. The idea is to provide a convenient developer experience (perhaps similar to FastAPI or Flask) while leveraging

BRRTRouter's coroutine-powered core. Although Photon is currently just a placeholder repository, we should outline how integration will work so that both projects can evolve in tandem. Key considerations include: how Photon will invoke BRRTRouter (likely as a library or via an embedded spec), what additional functionality Photon will add (templating, database integration, etc.), and any extension points needed in BRRTRouter to support Photon (for example, hooks for template rendering or easier route definition without writing OpenAPI by hand). This issue is to track design discussions and ensure BRRTRouter's development doesn't conflict with or limit Photon's goals.

Steps to Reproduce: (Design discussion, not a bug)

1. Review the Photon repository and any design notes (Photon has no code yet but has a README with overall objectives).
2. Identify what Photon will need from BRRTRouter. For instance, Photon might want to allow defining routes via code annotations or functions, which under the hood must generate an OpenAPI spec or directly register routes in BRRTRouter.
3. Ensure any changes we make in BRRTRouter (like dynamic route registration, better codegen) align with facilitating Photon's features.

Proposed Plan: Coordinate with Photon's design by possibly adding a mode to BRRTRouter that allows programmatic route definitions (since Photon's users may prefer code-first route definitions which then produce an OpenAPI spec, opposite of BRRTRouter's current spec-first approach). One approach is "design-first" (BRRTRouter native) vs "code-first" (Photon) – ensure BRRTRouter can accommodate both. Concretely:

- Provide an API in BRRTRouter to ingest an OpenAPI spec at runtime (already does) **and** an API to build that spec via code. For Photon, we might expose internal structures like `RouteMeta` or a builder pattern to add paths, parameters, etc., then generate the OpenAPI doc from it.
- Photon Integration could also involve sharing middlewares: e.g., Photon might have its own middlewares but under the hood use BRRTRouter's, so design middleware traits in BRRTRouter to be extensible.
- Identify any missing pieces Photon would need: template engine support (maybe Photon handles that at the handler level), session management (Photon could add, but may need hooks in router for state).
- Possibly implement a prototype within BRRTRouter examples to simulate Photon behavior (for instance, a small code-first DSL that populates the router) to validate feasibility.

The outcome of this issue would be a document or set of tasks ensuring BRRTRouter is Photon-ready. This might not result in immediate code in BRRTRouter, but it influences architecture (for example, ensuring that dynamic registration is efficient enough, or that all features in OpenAPI spec have equivalents in code-first definitions).

Relevant Code/Commits: - The Rust forum post by the author mentions Photon and its purpose [36], giving context that Photon is meant to wrap BRRTRouter. We should use that as guiding info.

- Photon's repo (microscaler/Photon) may have a README or issues; even if empty, its existence suggests planned features (web framework amenities). We might create a parallel issue in Photon's repo once concrete tasks are clear.
- Keep this issue open as an Epic with sub-tasks once we identify specifics (like "BRRTRouter: expose route builder API for Photon"). It's more of a roadmap item until Photon development starts.

Title: Async/Await Bridge for External Async Libraries

Labels: enhancement, async, research

Milestone/Epic: Future Ideas

Summary: BRRTRouter's custom coroutine runtime is independent of Rust's standard `async/await` (Futures) ecosystem. Many ecosystem libraries (database ORMs, HTTP clients, etc.) use `async/await` and require a Tokio or async-std runtime. To allow using those within BRRTRouter handlers (which run on `may` coroutines), we need a strategy to bridge between our stackful coroutines and the async futures. Without

this, users might be unable to easily call, say, a SQLx query or reqwest client from inside a BRRTRouter handler, limiting practicality. The goal is to enable awaiting futures inside our handlers, either by running a small async runtime on top of the coroutine or by offloading to a threadpool. Achieving this will vastly broaden compatibility with existing Rust ecosystem code.
Steps to Reproduce: 1. In a handler function, attempt to call an async function from an external crate (e.g., `my_db.fetch_one().await`).
2. This cannot be directly awaited because our handler is not an async function and not running on a futures executor. If you block on it (e.g., using `.wait()` or `block_on` hacks), it may block the entire coroutine thread, defeating concurrency.
3. Currently, there's likely no provided solution, so users might resort to running a separate Tokio runtime and doing sync calls via channels – cumbersome and inefficient.
Proposed Fix: Investigate implementing a compatibility layer. One possibility is to use `tokio::task::block_in_place` or spawn a separate OS thread for the future – but that's not ideal for performance. Instead, consider running a minimal single-threaded executor for futures on each coroutine thread. The `may` library might allow integration with `futures` via something like driving a future to completion by polling it within the coroutine. For example, we could implement a function `block_on_future(f: impl Future)` that uses `futures::pin_mut!` and then repeatedly calls `poll` on it, while yielding the coroutine when the future is not ready. This effectively schedules the future on our coroutine scheduler. This is complex but feasible: since `may` is M:N, we can perhaps incorporate a waker that wakes the coroutine when the future is ready to make progress. Another approach: provide an API for handlers to easily spawn a Tokio runtime in a separate thread and await using that – heavier but easier, perhaps as an opt-in (the user supplies a runtime handle to the middleware). Ideally, for 1.0 we might not implement fully, but this remains on the roadmap. This issue would involve prototyping and possibly coordination with the Rust async community to see if any prior art exists for integrating custom schedulers with futures (maybe using `futures::executor::LocalPool` inside our coroutine). The deliverable would be either a working `block_on_future` utility or guidelines for users to call async code (like an official pattern using threadpool).
Relevant Code/Commits: - No direct code yet in BRRTRouter for this. We might consider writing a small example to test bridging: e.g., use `futures_timer::Delay` future inside a coroutine and see if we can poll it to completion.
- The mention in the forum [37] about Rust possibly adding a native coroutine (`gen` keyword) in the future implies that eventually standard async and our approach might converge. Until then, we have to do it manually.
- This is a research-heavy task. It could also be broken out into a standalone crate (like an adapter that others could reuse). Monitor community projects or discussions for "sync-await bridging" or usage of `async_task` crate or such in non-async contexts.
- Given its complexity, mark this issue as future/experimental and maybe target a 1.x release once the core is stable.

Title: Explore Expanded Protocol Support (GraphQL, gRPC, etc.)
Labels: research, enhancement
Milestone/Epic: Future Ideas
Summary: After achieving a solid REST framework, it's worth exploring whether BRRTRouter's core can be extended to support other API styles/protocols. Two commonly requested ones are GraphQL and gRPC. While these are quite different from OpenAPI-driven REST, the underlying high-performance router and coroutine execution could be leveraged to implement them. For instance, GraphQL could be integrated by adding a GraphQL query parser/executor that runs within a coroutine, possibly using an existing library but

offloading execution to our runtime for parallel field resolution. gRPC support would involve speaking HTTP/2 and protocol buffers – which is a larger undertaking and might not align directly with OpenAPI (though OpenAPI 3.1 can describe some gRPC routes in theory, gRPC usually uses Protocol Buffers definitions). The aim of this issue is to research feasibility: can BRRTRouter unify these under one framework, or provide bridging so that, say, a GraphQL query on a certain endpoint is handled? Even if not immediate, having a plan or plugin interface for these will guide long-term roadmap.
Steps to Consider: 1. **GraphQL**: Users might want to define a GraphQL schema alongside OpenAPI. Possibly allow mounting a `/graphql` endpoint that handles POST GraphQL queries. We could integrate with a library like `juniper` or `async-graphql`. The main integration challenge is that those libraries are async (tying into the above async bridging issue) and have their own execution logic. But perhaps our coroutine model could actually speed up GraphQL resolvers by parallelizing them (GraphQL allows parallel resolution of independent fields; our coroutine pool could be great for that).
2. **gRPC**: This would require HTTP/2 support and reading `.proto` files. Perhaps out of scope for now, but we could consider converting a gRPC service definition to an OpenAPI spec as an intermediary (tools exist) and then serving it. However, performance and type fidelity might suffer. Alternatively, treat it as a separate mode where BRRTRouter can host a tonic (Tokio-based gRPC) service if we manage to run a Tokio runtime (again ties to async bridging).
Proposed Plan: For GraphQL, an easier start: create an example integration (maybe not in core, but in an example or separate module) where a GraphQL schema is served. E.g., use `async-graphql` crate in a BRRTRouter handler by using the async bridging or spawning a thread. Measure performance and see how well it plays with our coroutine scheduler. If promising, design a more first-class integration (maybe `GraphQLMiddleware` that intercepts a certain path and handles the request via a GraphQL executor). Document how to use it. For gRPC, likely leave it as a longer-term project; but keep the door open by ensuring our architecture (especially around HTTP/2 support if we ever add it) could accommodate it. Possibly collaborate with the Photon layer, as Photon might aim to support GraphQL or gRPC on top of BRRTRouter.
Relevant Code/Commits: - None in the core right now. This is largely exploratory. - GraphQL note: Since GraphQL is typically served on a single endpoint (`/graphql`), BRRTRouter's high-performance routing is less of an advantage (because it's not routing many endpoints, just one). However, our coroutine model could help in executing resolvers concurrently. - gRPC note: If we consider HTTP/2, check if `may_minihttp` or `may` supports HTTP/2 or if we'd need a different engine (perhaps integrate with hyper for HTTP/2 but then we lose coroutine benefit unless hyper can be driven by threads). Possibly this is beyond the scope unless Rust adds cooperative threading to async. - This issue is mainly to track interest and any external contributions. Mark it as discussion/research-oriented. Community feedback could be solicited here for use cases requiring these protocols.

These future issues are not blockers for 1.0 but represent directions the project could grow. Each would likely spawn its own design discussions and require significant work, so having them outlined helps in attracting contributors interested in those areas and prevents duplicating efforts if someone starts on it.

---

1 2 3 4 5 7 13 14 18 19 20 22 35 test: add header and cookie handling (#76) · microscaler/BRRTRouter@6f90696 · GitHub
https://github.com/microscaler/BRRTRouter/commit/6f9069689abddcbcd453cfa9b88a56843c84b26d

6 Catching panics in rust web service? Is this a sound error handling strategy? - The Rust Programming Language Forum
https://users.rust-lang.org/t/catching-panics-in-rust-web-service-is-this-a-sound-error-handling-strategy/41515

8 Panic within a handler should return an HTTP 500 · Issue #1501 · actix/actix-web · GitHub
https://github.com/actix/actix-web/issues/1501

9 10 12 16 17 21 23 24 25 27 28 29 30 31 32 33 34 GitHub - microscaler/BRRTRouter: BRRTRouter is a high-performance, coroutine-powered request router for Rust, driven entirely by an OpenAPI 3.1.0 Specification
https://github.com/microscaler/BRRTRouter

11 15 36 37 Rust Alternative to FastAPI - The Rust Programming Language Forum
https://users.rust-lang.org/t/rust-alternative-to-fastapi/129619

26 Add spec helper tests and expose functions (#77) · microscaler/BRRTRouter@0bca246 · GitHub
https://github.com/microscaler/BRRTRouter/commit/0bca246c38ba8a5a587c3c3d910f902151ff0032