ChatGPT

# Photon Web Framework Architecture (FastAPI-like DX with `may` Coroutines)

## Introduction

Photon is a Rust web framework inspired by Python's FastAPI and Rust's Loco, aiming for a **fast, intuitive developer experience (DX)**. Unlike most Rust frameworks, Photon will use **stackful coroutines** via the `may` crate instead of `async/await`. This choice affects how we design routing, I/O, and concurrency. Photon's core components include **BRRTRouter** (OpenAPI-driven HTTP router), **Lifeguard** (object pool facade for SeaORM), and **MiniJinja** (Jinja2-like templating). Key goals include:

- Automatic **OpenAPI** documentation generation (with file output)
- **Project scaffolding** CLI (Rails-style code generators)
- **HTTP server** with coroutine-based routing (BRRTRouter)
- **WebSocket** support
- **Background task** execution (coroutine-based, in-process)
- Developer-friendly **macros/derives** for routes and data validation

Below, we outline a recommended architecture and tooling for each aspect, ensuring **compatibility with `may`**, good DX, and maintainability. A summary table of recommendations is provided at the end.

## Architecture Overview (Coroutines vs Async)

At a high level, Photon will run a **coroutine runtime** instead of an async executor. The `may` crate provides Go-like lightweight threads ("green threads") scheduled on OS threads [1] [2]. This means handler functions can be written as regular blocking Rust code (no `.await` needed), and `may` will schedule them efficiently across threads. Important architectural considerations:

- **Coroutine Scheduler**: Initialize the `may` runtime with an appropriate number of worker threads (configurable for multi-core support [3]). Handlers can use blocking operations (DB queries, file I/O) without stalling the entire server – `may` will time-slice and context-switch on I/O. However, CPU-bound tasks should yield or run on separate threads to avoid blocking the scheduler.
- **Integration with BRRTRouter**: BRRTRouter is explicitly designed to pair with coroutine runtimes like `may` [4] [5]. It uses an OpenAPI spec to construct a routing table (with regex path matchers) and dispatches incoming requests to handler functions in **separate coroutines** [6]. Photon's architecture can leverage this by loading an OpenAPI specification at startup (either from a file or generated in-code) and then registering handlers. Each request will be routed and handed off to a coroutine running the corresponding handler, isolating request processing. This yields FastAPI-like simplicity (each endpoint handled by a single function) with high concurrency.

**DX and Maintainability**: Hiding the coroutine complexity from end-users is key. The developer should write handlers as if they were synchronous functions (similar to FastAPI's normal `def` endpoints) but achieve concurrency under the hood. Photon can provide a clean Application struct or `photon::Server` to manage the router and runtime. Internally, this struct would: (1) initialize `may`, (2) load/construct the OpenAPI spec and router, (3) spawn the listening server coroutine. This separation keeps the **project structure** clear (e.g. `main.rs` calls into Photon's API to start the server, rather than dealing with low-level `may` details).

## Routing and OpenAPI Integration

**BRRTRouter for HTTP Routing** – Photon should use BRRTRouter as the HTTP routing core. BRRTRouter is an OpenAPI-native router: it reads an OpenAPI 3.1 spec (YAML/JSON) and builds an internal routing tree (with support for all HTTP verbs and parameterized paths) [7]. It's built atop a custom coroutine-friendly HTTP server (`may_minihttp`) to handle network I/O within `may` [8]. This yields extremely fast request dispatch (aiming for ~1M req/s on a single core) while using **OpenAPI as the source of truth** [9] [10].

**OpenAPI Generation** – To avoid manually writing the spec for each project, Photon should support **code-first OpenAPI generation**. The developer can define routes in Rust (using macros/attributes, discussed later), and Photon will compile those into an OpenAPI spec, which is then: (a) used to configure BRRTRouter, and (b) saved to a file (for documentation or client generation). Recommended approaches and libraries:

- **Utoipa**: A code-first OpenAPI library that uses procedural macros to generate the OpenAPI schema at compile time [11]. Developers can annotate handler functions with `#[utoipa::path(...)]` and data models with `#[derive(ToSchema)]`, then derive a master `OpenApi` document. This produces a static OpenAPI JSON that can be written to disk or served via an endpoint [12] [13]. Utoipa integrates with popular frameworks, but Photon can use it standalone (or adapt its ideas) since our routing is custom.
- **Okapi/Paperclip**: Alternatively, the Okapi crate (with Rocket) or Paperclip (for Actix) offer OpenAPI generation. However, they are tied to specific frameworks and older OpenAPI versions [14]. Utoipa is more flexible and up-to-date.
- **Custom Macro Collection**: Photon could implement its own attribute macros to gather route info. For example, a macro could parse attributes like `#[route(method = "GET", path = "/users/{id}", responses(...))]` on handler functions and accumulate these into a global OpenAPI builder. This might use an *inventory* pattern or a compile-time generated registry. Given the complexity, using a well-tested library (like Utoipa) for the spec structure is beneficial for maintainability.

**Integration with BRRTRouter** – Once the OpenAPI spec is generated (either at build-time or startup), Photon will load it into BRRTRouter. BRRTRouter parses paths, methods, parameters, and even custom extensions like `x-handler-*` for operation-to-handler mapping [7]. A good pattern is to use the OpenAPI **operationId** field to link to handler functions. For instance, the code macro could assign each route a unique operationId (perhaps derived from the function name or an attribute). Photon at startup can call `Dispatcher::register_handler(operation_id, handler_fn)` for each defined route, as shown in BRRTRouter's example [15]. The router then dispatches incoming requests by matching the URL to an operationId and invoking the associated handler coroutine [16] [6]. This separation means the OpenAPI spec drives the routing (ensuring docs and behavior match exactly), but developers primarily write Rust code – the spec can be generated behind the scenes for convenience.

**DX Considerations**: The goal is to let developers define routes **declaratively and type-safely**, similar to FastAPI's decorator syntax. Photon's routing API should minimize boilerplate – e.g. using a macro rather than requiring the user to manually register each route string. By leveraging OpenAPI, we also automatically get **request validation** at the HTTP layer (unsupported methods or paths yield a 404/405, parameter types can be validated if specified in the spec). The **OpenAPI file output** provides immediate feedback to the developer and can be used to generate client code or documentation. For maintainability, ensure that the spec generation code stays in sync with any new features (auth, middleware, etc.), or allow extension via OpenAPI extensions and metadata.

## Project Scaffolding via CLI

A hallmark of Rails (and Loco) is powerful CLI generators for project setup and code scaffolding. Photon should provide a CLI (possibly as a `cargo-photon` subcommand or standalone `photon` binary) to bootstrap projects and add new components. Key features and recommended implementation:

- **Initial Project Generator**: Similar to `loco new`, provide `photon new <project_name>` to create a new project with a standardized layout (src folders, Cargo.toml, example routes, config files, etc.). This could present interactive prompts (for DB choice, template engine, etc.) or use flags for a non-interactive creation. Internally, you can use template files stored in the Photon CLI binary (for example, embedded using `include_str!` or packaged with `photon`) and simply write them out. Tools like **Baker** demonstrate using MiniJinja templates for scaffolding projects [17] [18], which Photon could adopt. Using **MiniJinja** for code templates is fitting since it's already a dependency for HTML templating – you can reuse it to fill placeholders in template files (e.g. project name, struct names).

- **Generators (Scaffolds)**: Provide commands to generate common components: e.g. `photon generate scaffold <name> [fields...]` similar to Rails/Loco. For example, `photon generate scaffold post title:string content:text` would create a `Post` model, a controller with CRUD routes, and maybe a template or API responses. Loco's CLI illustrates this DX: running `cargo loco generate scaffold post title:string content:text --api` creates a controller file and updates mod registries automatically [19]. Photon can mirror this: after generation, print out files created and any modifications (like inserting route registration in a central file, if needed).

- **Implementation**: Build the CLI using a crate like **Clap** (or its derive macro) for parsing subcommands and flags. Clap makes it easy to define nested commands (e.g. `photon new`, `photon generate model`, `photon generate controller`, etc.). Use descriptive help messages to guide users. Under the hood, each generator subcommand would invoke code that creates files or modifies existing ones. For file creation, maintain template files within the CLI. For modifying existing files (like adding a new route to an `app.rs`), consider keeping identifiable markers in those files (comments like `// <photon-append routes>`), so the generator can insert code at the right spot. This automated injection is demonstrated by Loco (it injects new lines into `src/controllers/mod.rs` and `src/app.rs` on scaffold [19]), improving DX by reducing manual editing.

**DX and Maintainability**: A well-designed scaffolding CLI **accelerates development** (especially for CRUD apps) and encourages convention. It should generate idiomatic Rust code so that new users can learn from the scaffold. Maintainability of the CLI templates is important – keep them minimal and clearly commented. Also, document how to perform common tasks both with and without the CLI (some users prefer manual coding). Photon's CLI can significantly close the gap with FastAPI's ease of starting a project by providing one-command setup and generators for resources.

## Templating with MiniJinja (Static Templating)

For server-side HTML rendering or email templates, Photon plans to use **MiniJinja**, a lightweight Jinja2-like engine. MiniJinja is well-suited here because it has **minimal dependencies and a simple, synchronous API** [20]. Key points for integration and compatibility with `may`:

- **Synchronous Rendering**: MiniJinja does not require async/await; template rendering is a blocking operation that takes a context (any `serde`-serializable value) and produces a String or writes to a stream. In a coroutine runtime, this is acceptable because each request handler runs in its own coroutine – rendering a template will block that coroutine only. The `may` scheduler can continue running other coroutines on the same OS thread if the template I/O (e.g. reading template from disk) yields. In practice, **template rendering is CPU-bound**; if templates are complex, it could occupy the OS thread. Mitigation strategies: (a) **Compile templates ahead** of time (load them at startup into the `Environment`) so that file I/O is not done per request; (b) If rendering becomes a bottleneck, consider spawning a separate thread (or coroutine on a different thread) for particularly heavy render tasks, to keep the main thread free – but this is usually not needed unless templates are huge.

- **Integration**: Photon can provide a global or per-request `Environment` (MiniJinja's container for templates). A good practice is to initialize the `Environment` at application startup, load all template files from a `templates/` directory, and store the `Environment` in a global (protected by a mutex) or in a coroutine-local variable if `may` supports coroutine-local storage (similar to thread-local). Given MiniJinja supports adding templates programmatically and then rendering by name [21] [22], Photon can offer a helper like `photon::templates::render("template_name", &context)` that uses the pre-loaded environment. This avoids reading template files for each request and aligns with how frameworks like Rails or Django load templates.

- **Static vs Dynamic**: By "static templating," likely Photon will treat templates as static files (not modify them at runtime, as opposed to something like live reload). This is straightforward: load templates at startup and reload them only if the application is restarted. If hot-reloading templates is desired for development DX, Photon could watch the template directory for changes and reload the `Environment` (this is a nice-to-have, not essential). MiniJinja's API should allow replacing templates even at runtime if needed.

**Compatibility with `may`**: There are no special incompatibilities – MiniJinja is pure Rust and uses CPU and memory. It does use `serde` for injecting context data [23], but that's orthogonal to async. One thing to note: if Photon's handlers are all running on a small set of threads (say 4 threads for 4 CPU cores), a template render will use one core fully during that operation. In practice, this is fine and similar to running template rendering in synchronous frameworks. Because `may` coroutines are cooperatively scheduled,

**long-running computations** (like a very large template loop) won't yield unless performing I/O. If this becomes an issue, the handler could explicitly call `coroutine::yield_now()` (if available) to allow others to run. However, typical template rendering should be fast enough not to require this.

**DX considerations**: Using MiniJinja means Photon developers can leverage familiar Jinja2 syntax. This is good for **productivity** (many Python/JavaScript developers know it). Ensure to document the subset of Jinja2 that MiniJinja supports and provide examples for using it (loops, conditionals, includes, filters, etc.). If Photon wants to offer an alternative (like **Askama** for compile-time templates), it could – Askama provides compile-time checked templates (which improve performance and catch errors early) but it is less dynamic (requires recompilation to update templates). As a default, MiniJinja is a solid choice for flexibility. It keeps the learning curve shallow and the development process closer to FastAPI (where Jinja2 templates are commonly used via Starlette's Jinja integration).

## WebSocket Support in a Coroutine Runtime

WebSockets are crucial for real-time features. Implementing WebSockets with `may` (without async/await) is feasible using existing synchronous libraries. Photon can integrate **Tungstenite**, a lightweight WebSocket protocol library, to handle upgrades and frame management. Here's how to approach it:

- **HTTP Upgrade Handling**: BRRTRouter (via `may_minihttp`) currently focuses on HTTP request/response routing. To accept a WebSocket, an HTTP request with `Connection: Upgrade` and `Upgrade: websocket` headers is sent by the client. Photon needs to detect these and perform the WebSocket handshake. This likely requires a small extension in the HTTP server layer: when a route is designated as a WebSocket endpoint, the handler should **not** send a normal HTTP response. Instead, it should take control of the underlying TCP stream after the handshake. If BRRTRouter doesn't directly support websocket upgrades in the spec (OpenAPI doesn't natively cover WS), Photon can handle this by providing a separate registration for WS routes (e.g., `app.websocket("/chat", handler)` outside of the OpenAPI spec). The server would still listen on the same port, but if a request path matches a WS route, Photon would bypass normal routing and initiate the upgrade.

- **Tungstenite Integration**: Tungstenite's blocking API can work with any stream implementing `std::io::Read + Write`. The `may_minihttp` server likely uses `may::net::TcpStream` internally for connections. We can obtain the `TcpStream` (or a wrapper) from the request context if exposed, or modify the server to expose a way to get it for upgrades. Tungstenite's `accept()` function will perform the WebSocket handshake on a given stream [24]. It supports standard and TLS streams. Once accepted, it returns a `WebSocket<Stream>` object which the handler can use to send/receive messages. Because `may::net::TcpStream` is coroutine-friendly, reading/writing from it will not block the OS thread – it will yield if no data is available, allowing other coroutines to run. This means we can treat Tungstenite's blocking calls as cooperative within `may`. Each WebSocket connection can be handled by a **dedicated coroutine** (just like HTTP handlers), which loops on `ws.read_message()` and processes or broadcasts messages.

- **Designing the API**: Provide an ergonomic way to define WS handlers. For example:

```
photon::route::websocket("/chat", on_chat_connected);
fn on_chat_connected(ws: PhotonWebSocket) {
    // e.g., spawn a coroutine to listen for messages
    while let Ok(msg) = ws.read_text() {
        println!("Received: {}", msg);
        ws.write_text("echo: ".to_owned() + &msg).unwrap();
    }
}
```

Internally, `PhotonWebSocket` could be a wrapper around Tungstenite's `WebSocket<may::net::TcpStream>` to provide convenient methods. Photon should handle the low-level errors (when a client disconnects, etc.) by ending the coroutine gracefully. For multiple clients or chat rooms, you can use `may::sync::mpsc` channels or broadcast mechanisms to distribute messages between coroutines.

- **Concurrency**: Each WebSocket connection lives in its own coroutine (and possibly on a separate OS thread, since `may` can schedule coroutines across threads). This model is similar to how Go or synchronous Python frameworks handle websockets – straightforward and **easy to reason about**. The main difference from an async model is that here you don't need to `.await` each send/receive; the code looks sequential. This should give a FastAPI-like feel (FastAPI itself uses `await` for WS, but the logic is similar).

**Maintainability**: Using Tungstenite (which is well-maintained) avoids reinventing the WebSocket protocol. The integration requires careful handling of the handshake. If `may_minihttp` does not natively allow taking over the socket, we might consider using a separate listener for WS (e.g., using a plain `may::net::TcpListener` on another port). However, it's better for DX to serve WS on the same port/URL. If needed, contribute a patch or extension to BRRTRouter's server to support an upgrade hook. Documentation should clearly mark which routes are HTTP vs WebSocket, since the OpenAPI spec file will not list the WS routes (we can't easily describe them in OpenAPI 3.1; at best, we could document them informally).

**Alternatives**: There are other synchronous WS libraries (e.g., the older `ws` crate), but Tungstenite is simple and supports the latest WebSocket standards (and can interoperate with async if needed later). Photon could also expose lower-level hooks allowing advanced users to use any I/O type within `may`. But for high-level DX, wrapping Tungstenite behind a Photon API is recommended.

## Background Tasks and Scheduling (Coroutine Style)

FastAPI allows launching background tasks (after returning a response) and also has integration with task queues (like Celery) for out-of-process jobs. Photon should enable **background task execution** in a similar spirit. With `may`, we can spawn new coroutines to handle background work. Here are patterns and recommendations:

- **In-Process Async Tasks**: The simplest scenario is running background tasks in the same process (just not blocking the request response). For example, a request might need to send an email after responding; the handler can return the HTTP response, then queue a task to send the email in the background. In Photon, this can be as easy as: `may::coroutine::spawn(move || { ... })` to run a closure in a new coroutine. Since `may` coroutines are scheduled on the thread pool, this background task will run concurrently. This is analogous to how Loco's default "async" workers mode uses Tokio to run jobs in-process [25] [26]. We achieve the same with `may` (no Tokio needed for this, just `may`'s scheduler). Photon can provide a helper, e.g. `photon::spawn_background(task_fn)` to abstract the raw `may` API. Under the hood it might simply call `may::go!` (the macro provided by `may` for spawning) or a similar function.

- **Task Management**: For DX, it's useful to have a **BackgroundTasks** mechanism (FastAPI has a `BackgroundTasks` dependency injection). Photon can offer a parameter or context where handlers can register tasks to run after the response. For example, a handler could push a closure to a list of background jobs, and the framework executes them after sending the HTTP response. Implementation might involve the request dispatcher capturing any tasks the handler registers (perhaps by returning a special response type that carries background jobs). Simpler yet, since our handlers can spawn tasks directly, we may just document that "if you need to run work after responding, call `photon::spawn_background` at the end of your handler". The coroutine will continue running after the main function returns a response (especially if the handler's response is sent through a channel or a future). Care must be taken that the HTTP response is not held up by these tasks – likely the handler needs to signal response completion first. In practice, one can spawn the background task **before** returning the result to avoid any delay.

- **Scheduling and Timers**: The `may` crate supports efficient timers [2] [27], which means Photon can provide functions like `photon::sleep(duration)` to pause a coroutine, or schedule tasks to run after a delay. For periodic tasks (like cron jobs or scheduled cleanups), Photon could include a small scheduler that uses `may` timers or simply spawns a coroutine that loops forever with a sleep. This would be analogous to Celery periodic tasks or cron jobs, but in-process. Ensure that such tasks are started on server startup (perhaps via configuration or annotations).

- **External Task Queues**: While initially Photon can focus on in-process tasks (for simplicity), it's wise to design an abstraction that could later integrate an external queue. Loco, for example, can switch between Tokio in-process tasks and external Sidekiq/Redis queues with minimal code changes [28] [29]. Photon could define a trait or simple interface for "task executors" (one implementation spawns local coroutines, another could enqueue to Redis). In the short term, providing just the local executor is fine, but keeping this extensible improves maintainability and scaling options.

**DX Considerations**: The ability to fire-and-forget a task is extremely useful – ensure it's as easy as calling a function. Document clearly that background tasks run in the same binary concurrently, and thus share memory (so users can update database, etc., but also need to handle errors/logging properly since there's no external monitor). Provide hooks for logging failures in background tasks (maybe any panic inside a spawned coroutine is caught by `may` which supports panic isolation [30] – we should confirm that). Indeed, `may` supports *graceful panic handling that will not affect other coroutines* [30], which is great for isolating a failing background job. Still, encourage users to use `Result` returns and handle errors in tasks.

**Maintainability**: Using `may` directly for tasks ties Photon to that runtime. If one day a switch to another coroutine library or even tokio is considered, abstracting the spawn mechanism through our own API will help. Also, ensure that any global state (DB connections, etc.) is accessible from within these tasks (maybe via injecting an app context). Avoid global mutable state, but a context object (with DB pool handle, config, etc.) can be passed to tasks or made accessible via coroutine-local storage (since `may` has coroutine-local similar to thread-local). This is akin to FastAPI's dependency injection – not as automated in Rust, but can be managed via context passing.

## Route Declaration, Validation, and Macros (FastAPI-like)

FastAPI excels by using decorators to declare routes and Pydantic models for data validation. We can emulate these patterns in Rust for Photon:

**Attribute Macros for Routes** – Define procedural macros to decorate handler functions, reducing boilerplate in route definitions. For example:

```
#[photon_route(method = "GET", path = "/items/{id}", query_params(Q), json_body
= ItemIn)]
fn get_item(id: u32, Q: ItemQuery) -> Result<ItemOut, HttpError> { ... }
```

Here, `#[photon_route]` could be a macro that does multiple things: register the route (possibly by appending to a static route list or generating a call to the dispatcher registration), and also ensure that the function signature is respected (e.g. types implement required traits). The macro can auto-generate an `operationId` (maybe `"get_item"` from the function name) and even produce OpenAPI annotations for this route. If using Utoipa internally, the macro might expand to `#[utoipa::path(...)]` with appropriate arguments, plus code to register the handler. This way, one annotation serves both documentation and runtime registration. Using macros in this fashion can provide a **clean DX**: developers just annotate normal functions, similar to FastAPI's `@app.get("/items/{id}")`. The compile-time magic sets up the routing and documentation.

**Typed Request Parameters and Validation** – We want Photon to parse path, query, and body parameters into Rust types automatically, akin to FastAPI using Pydantic models. BRRTRouter already extracts path params, query strings, and JSON body for each request and makes them available to the handler [31]. Currently, it passes them likely as untyped (e.g. a map or `serde_json::Value`). Photon should improve this by mapping them to handler arguments: if a handler function expects `id: u32`, `may_minihttp`/ BRRR will have captured `"id": "123"` as a string – Photon should convert that to `u32` (with error handling if parse fails, returning 400). We can implement a `FromRequest` **trait** for different types: e.g., any type that implements `std::str::FromStr` can be auto-parsed from a path or query param. For JSON body, any type that implements `serde::Deserialize` can be parsed from the request body JSON. The route macro can generate code to do these conversions before calling the handler. For example, expanding the macro might yield something like:

```
// Inside the macro expansion for get_item
fn get_item_wrapper(request: PhotonRequest) -> PhotonResponse {
    // extract and parse "id" from path
    let id: u32 = request.param("id").parse().map_err(|e| bad_request(e))?;
    // parse query string to ItemQuery struct
    let Q: ItemQuery =
serde_qs::from_str(request.query())?; // or similar crate for query to struct
    // parse JSON body to ItemIn struct if present
    let body: ItemIn = serde_json::from_slice(request.body())?;
    let result = get_item(id, Q); // call the user-defined handler
    // convert Result<ItemOut, HttpError> to HTTP response
    PhotonResponse::from(result)
}
```

Then `get_item_wrapper` is what actually gets registered as the coroutine handler with BRRTRouter. The user's `get_item` remains clean and focused on business logic, not parsing.

To achieve this cleanly, Photon can introduce simple extractor types: similar to Actix-web's `Path<T>` or `Json<T>` wrappers. For instance, if a user wants the raw body, they could use a parameter type `Body: photon::RequestBody` and we implement `FromRequest for RequestBody` to provide it. But often, directly using concrete types (with traits) is simpler.

**Validation** – Rust doesn't have something exactly like Pydantic's field validation out-of-the-box, but we have libraries like **validator** (which provides a `Validate` derive trait). A pattern is: define your request models with `serde` for deserialization and `validator` for extra checks, then call `model.validate()` after deserializing. Photon's framework could do this automatically if the type implements `validator::Validate`. For example, if `ItemIn` derives `Validate`, the macro expansion can insert `body.validate().map_err(|e| bad_request(e))?;`. This gives a pydantic-like guarantee that all invariants (min lengths, ranges, regex patterns, etc.) are met or a clear error is returned.

Another aspect is **type-driven OpenAPI schema**: using `schemars` or Utoipa's `ToSchema`, we can automatically document the request/response models. With Utoipa, for instance, deriving `ToSchema` on `ItemIn` and `ItemOut` and referencing them in the path macro will include their definitions in the OpenAPI output [13]. This is greatly beneficial for DX, as the API documentation stays in sync with the code without extra effort.

**Macros and Derives for Models** – Encourage developers to use `serde::Deserialize` and `serde::Serialize` on their data models (as needed for JSON in/out). Additionally, if they use `#[derive(Validate)]` from the validator crate or custom validation logic in constructors, it improves safety. Photon could re-export common crates for convenience (like re-export `validator::Validate` trait and macros, or provide its own attribute e.g. `#[photon_model]` to wrap serde + validate derives).

**Example**: A developer could write:

```
#[derive(Deserialize, Validate, ToSchema)]
struct UserCreate {
    #[validate(length(min = 3))]
    name: String,
    #[validate(email)]
    email: String,
}
```

and their handler `fn create_user(body: UserCreate) -> Result<User, HttpError>`. The framework will deserialize JSON into `UserCreate`, run `validate()`, and if okay, pass it. On error, it could return an automatic 422 Unprocessable Entity with details, similar to FastAPI's error for invalid input.

**DX & Maintainability**: Using macros to declaratively define routes and models reduces boilerplate and makes the **intention** clear (just like FastAPI's decorator and Pydantic usage shows exactly what the endpoint expects and returns). However, writing custom procedural macros is non-trivial – ensure to test them thoroughly and provide good compiler error messages when misuse occurs. If implementing from scratch is too large an effort initially, Photon could leverage existing solutions: for example, use Utoipa for documentation, use `serde_qs` for query string parsing to structs, use `validator` for validation, etc., glueing them in userland code rather than a single macro. The first iteration could even require slightly more explicit code (e.g. the user calls `body.validate()?;` themselves in the handler), but the end goal should be to streamline this with macros or framework support.

**Alternatives**: If BRRTRouter's design (spec-driven) ever feels too restrictive for dynamic route definition, one could consider more conventional code-driven routers (like an Axum or Actix style) but with `may` runtime. Currently, since BRRTRouter is already coroutine-based and OpenAPI-centric, sticking with it provides consistency and performance [10]. The use of macros and codegen should complement it by generating the spec and registration calls needed. This hybrid approach (code generation of spec → driving router) is unusual but can yield a very powerful DX: **one source of truth** (the Rust code) for both routing behavior and API documentation, akin to FastAPI.

## Developer Experience (DX) and Compatibility Summary

Bringing it all together, Photon's design should emphasize *developer joy* and modern best practices, while navigating the constraints of a coroutine model. Key architectural choices – like using `may` for sync-like coding, BRRTRouter for OpenAPI-driven routing, and rich code generation – all serve to make the framework approachable yet robust. The coroutine model, while less common in Rust, is fully capable of high performance networking [8], and it simplifies the mental model (no `Future` state machines for the user). Ensure to document any gotchas (for example, that certain async-only libraries won't work unless wrapped, or how to integrate with ecosystem tools). By leveraging existing libraries (MiniJinja, SeaORM via Lifeguard, Tungstenite, etc.), Photon can stand on the shoulders of giants and avoid reinventing the wheel, focusing on the glue that delivers a Rails/FastAPI-like experience.

Below is a summary of each major Photon feature with recommended approaches and tools:

| Feature | Recommended Approach & Tools | Notes / Alternatives |
|---|---|---|
| **Coroutine Runtime** | Use `may` for stackful coroutines (M:N threading). Write handlers as normal functions (no async) and let `may` schedule them [8]. | Ensures simpler code. If `may` faces issues, consider `glommio` or other async-alternatives, but `may` is proven for coroutine I/O. |

| Feature | Recommended Approach & Tools | Notes / Alternatives |
|---|---|---|
| **Routing & HTTP Server** | Integrate **BRRTRouter** for routing, driven by an OpenAPI 3.1 spec [9] [10]. Generate the spec from code, load it at startup, use `Dispatcher` to register handlers. | BRRTRouter + OpenAPI ensures docs and routing are in sync. If spec-first is cumbersome, could allow code-only routes and internally build the spec object. |
| **OpenAPI Generation** | **Utoipa** crate for code-first OpenAPI generation [32], or custom macros. Derive schemas on data models (`serde` + `schemars`). Write out `openapi.json` on build or startup for user. | If Utoipa doesn't fit, consider **Okapi** (Rocket-specific) or **Paperclip** (Actix) for inspiration. In future, could allow reading an external spec for design-first workflows. |
| **Database (ORM) & Pool** | Use **SeaORM** for ORM (ActiveRecord-like). Since SeaORM is async, run DB ops in a dedicated thread or via `may` blocking. **Lifeguard** can manage a pool of DB connections (wrap SeaORM's DatabaseConnection). Possibly spawn a lightweight Tokio runtime for DB queries (if needed) and use blocking calls in handlers. | **Alternative**: Use Diesel (synchronous ORM) for simpler integration (at cost of compile-time schema binding). Lifeguard's pool ensures we reuse DB connections safely. Document that DB calls in Photon are synchronous from the handler perspective. |
| **Templating** | **MiniJinja** for HTML templates (Jinja2 syntax, runtime rendering) [20]. Load templates at startup into an `Environment`. Provide helper to render templates with context (any `serde::Serialize`). | Alternatively, **Askama** (compile-time templates) offers faster rendering and type checking, but less dynamic (requires rebuild to update templates). MiniJinja is flexible and easy for a Rails-like experience (where templates are just files). |
| **Project Scaffolding CLI** | Build `photon` CLI (or `cargo-photon`) with **Clap** for commands. Use embedded templates (possibly with MiniJinja or simple substitutions) to generate new projects and components [19]. Provide generators for models, controllers, etc., injecting code as needed. | Can integrate with **sea-orm-cli** to generate DB entity files if using SeaORM (like Loco does [33]). For initial template, consider hosting a Git repo and using `cargo generate` as an option, but a built-in approach offers more control. |

| Feature | Recommended Approach & Tools | Notes / Alternatives |
|---|---|---|
| **WebSocket Support** | Use **Tungstenite** for WebSocket handshake & framing (synchronous API). Designate WS routes separately; on upgrade request, use `tungstenite::accept()` on the `may` TCP stream to get a WebSocket object [24]. Handle send/recv in a coroutine loop per connection. Provide a friendly API (e.g., a `PhotonWebSocket` object with high-level methods). | Ensure the HTTP server can hand off sockets for WS. If integrating is complex, a fallback is running a parallel WS server (on another thread/port). Alternative crate: `ws` (which is callback-based) but it's older; tungstenite is lightweight and widely used. |
| **Background Tasks** | In-process tasks via `may::coroutine::spawn`. Expose as `photon::spawn_background`. Use `may`'s timer for scheduling delayed tasks. Possibly provide a task runner that executes tasks after response (similar to FastAPI's `BackgroundTasks`). | Future alternative: integrate **Sidekiq-rs** or a RabbitMQ/Redis queue for out-of-process tasks (Photon could push tasks to a queue service). But initially, coroutine-based tasks (like Loco's Tokio workers [25] analog) suffice. |
| **Routes & Validation Macros** | Procedural macros for route definitions (e.g., `#[photon_route]`), to auto-register routes and generate OpenAPI info. Utilize traits for extracting params (path/query/body) into types, leveraging `serde` for JSON and `FromStr` for path params. Use `validator` crate for model validation (call `Validate` on deserialized input) to emulate Pydantic behavior. | Draw inspiration from frameworks: Actix extractors, Rocket's attribute routes, FastAPI's decorator style. Ensure error handling (e.g., parse errors return 400) is done in framework code. This keeps user handlers clean. If writing custom macros is too heavy initially, require the user to call helper functions for extraction and later iterate towards full macro sugar. |

By following these recommendations, Photon can deliver a powerful yet ergonomic web framework. The combination of a coroutine runtime and high-level tooling aims to give developers the "FastAPI feeling" – quick development, intuitive syntax, and automatic documentation – with Rust's performance and reliability. The architecture choices also leave room for growth: for example, one could add middleware hooks (authentication, logging) via wrapper macros or integrate more libraries as the ecosystem evolves. In summary, **Photon's design should marry Rust's strengths (type safety, speed, modern libraries) with the developer-centric approach of frameworks like FastAPI and Rails**, all while using `may` to keep the programming model straightforward and synchronous-like [34] [35]. With careful implementation of the above components, Photon can achieve this balance, providing an enjoyable DX and a maintainable, scalable core.

[1] [2] [3] [27] [30] may - Rust
https://docs.rs/may

[4] [5] [6] [7] [8] [9] [10] [15] [16] [31] 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] [12] [13] [14] [32] Auto-Generating & Validating OpenAPI Docs in Rust: A Streamlined Approach with Utoipa and Schemathesis – Identeco
https://identeco.de/en/blog/generating_and_validating_openapi_docs_in_rust/

[17] [18] ali_aliev (u/ali_aliev) - Reddit
https://www.reddit.com/user/ali_aliev/

[19] Loco.rs - Productivity-first Rust Fullstack Web Framework
https://loco.rs/

[20] [21] [22] [23] minijinja - Rust
https://docs.rs/minijinja

[24] accept in tungstenite - Rust
https://docs.rs/tungstenite/latest/tungstenite/fn.accept.html

[25] [26] [28] [29] Workers - Loco.rs
https://loco.rs/docs/processing/workers/

[33] A Quick Tour - Loco.rs
https://loco.rs/docs/getting-started/tour/

[34] [35] Rust Alternative to FastAPI - The Rust Programming Language Forum
https://users.rust-lang.org/t/rust-alternative-to-fastapi/129619