AGENDA

What We'll Cover

Part 1: Foundations

Starting from zero: what problem are we solving, and what is PAGI?

Part 2: Meet PAGI::FastAPI

Routing, validation, dependency injection, WebSocket, SSE, rate limiting, CSRF, bot protection, and automatic docs.

Part 3: The Security Gap

Why authentication is not built in, and the building blocks that are.

Part 4: PAGI::FastAPI::Security

Ready-made schemes: Bearer, Basic, API Key, and OAuth2.

Zero assumed knowledge, every new idea gets a plain-language reference point as we go.

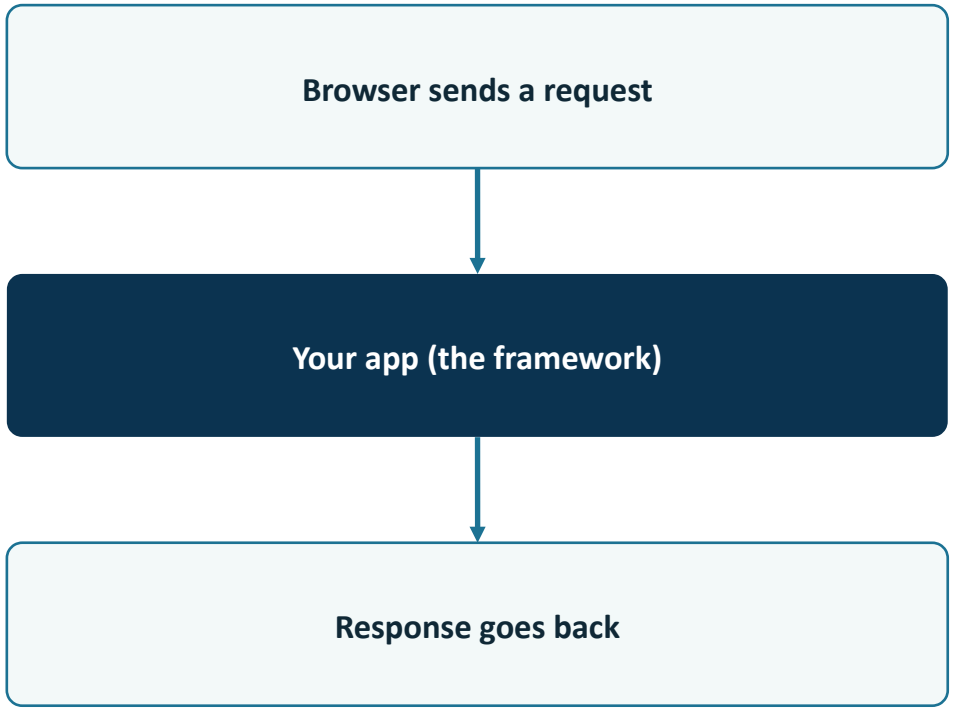
PART 1

Foundations

Starting from zero, what problem are we even solving?

What Is a Web Framework?

- A browser (or app) sends a request over the internet, "give me this page" or "save this order."
- Something has to receive that request, figure out what to do, and send back a response.
- A web framework is the toolkit that makes writing that "something" easy.



Restaurant analogy:
The customer (browser) places an order (request). The waiter (your app) takes it, maybe checks with the kitchen (database), and brings back a plate (response).

THE BIG PICTURE

The Trouble With "One Thing at a Time"

- A "blocking" waiter won't start a new table until the last one has completely finished eating.
- If one request is slow (waiting on a database, another API, a file), everyone else waits too.
- "Async" lets one waiter juggle many tables, switching to a new one exactly while waiting on something slow.

PAGI::FastAPI is async natively, handlers are async subs that can pause on slow work without blocking everyone else.

Table 1
(waiting on DB)

Table 2
queued...

Table 3
queued...

Table 4
queued...

THE FOUNDATION

What Is PAGI?

PAGI = Perl Asynchronous Gateway Interface

- A standard "socket": any PAGI-speaking app can plug into any PAGI-speaking server.
- It defines how a server hands a request to your app, and how your app hands a response back.
- You almost never write PAGI directly, frameworks like PAGI::FastAPI speak it for you.

Reference point: this is the same idea as PSGI/Plack in classic Perl, or ASGI/Uvicorn in Python, PAGI is that, for modern async Perl.

THE FOUNDATION

Where PAGI::FastAPI Fits In

Your App	The routes and business logic you actually write.
PAGI::FastAPI	Routing, type validation, dependency injection, automatic docs.
PAGI protocol	The standard "socket" connecting app and server.
pagi-server	The actual process that accepts connections and runs the loop.

You almost never touch PAGI directly. PAGI::FastAPI is the friendly layer you write against, inspired by Python's FastAPI.

PART 2

Meet PAGI::FastAPI

From zero to a running API in a few lines

Install & Hello World

terminal

```
cpanm PAGI::FastAPI
```

app.pl

```
use PAGI::FastAPI;
my $app = PAGI::FastAPI->new(
    title => 'My API',
);
$app->get('/hello',
    handler => async sub ($c) {
        return { message => 'Hello!' };
    }
);
$app->to_app;
```

run it

```
pagi-server app.pl
```

try it (another terminal)

```
curl http://localhost:5000/hello
```

Reference point: if you know Python's FastAPI, this will feel immediately familiar, same shape, Perl syntax.

Routing & Path Parameters

app.pl

```
$app->get('/items/{id}',
    handler => async sub ($c) {
        return {
            item_id => $c->param('id'),
        };
    }
);
```

- GET, POST, PUT, PATCH, and DELETE are all supported the same way.
- {id} in the path becomes a named path parameter.
- Every handler is an async sub, non-blocking by default.

$c is a PAGI::FastAPI::Context object, your window into the request (params, headers, body) and the response (status, headers).

Automatic Validation, Built In

app.pl

```
$app->post('/items',
    body => { name => Str, price => Int },
    handler => async sub ($c) { ... }
);
```

bad request -> 422 automatically

```
{
  "detail": "Body param 'price'
   invalid: not an Int"
}
```

- Query and JSON body values are checked against Type::Tiny constraints.
- Bad or missing data never reaches your handler code.
- Reference point: like Python FastAPI's Pydantic models, using Type::Tiny instead.

Dependency Injection with Depends()

- A dependency is a reusable piece of setup logic, a DB connection, the current user, anything.
- Declare it once, inject it into any route that needs it.
- A dependency can also act purely as a guard, more on this in Part 3.

app.pl

```
my $get_db = async sub ($c) {
    return My::DB->connect;
};

$app->get('/items',
    dependencies => [
        Depends($get_db, key => 'db'),
    ],
    handler => async sub ($c) {
        my $db = $c->stash->{db};
        return { status => "ok" };
    }
);
```

Middleware & CORS

runs on every request

```
$app->add_middleware(
    async sub ($c, $next) {
        ...
        return await $next->($c);
    }
);
```

for browser-based frontends

```
$app->add_cors(
    origins => ['https://example.com'],
);
```

Middleware wraps every request; dependencies target specific routes. CORS now delegates to PAGI::Middleware::CORS (PAGI::Tools) rather than a hand-rolled version.

Rate Limiting, Built In

app.pl

```
$app->add_rate_limit(
    requests => 100,
    window   => 60,
);
# Adds x-ratelimit-limit,
# -remaining, and -reset
# headers automatically
```

- Exceeding the quota returns HTTP 429 with a retry-after header, automatically.
- Per-route limits too: $app->get($path, rate_limit => {...}, handler => ...).
- Pluggable storage via PAGI::FastAPI::RateLimit::Driver, in-memory by default.
- Two external distributions available for Rate Limit: PAGI::FastAPI::RateLimit::Driver::CHI and PAGI::FastAPI::RateLimit::Driver::Redis

Keying defaults to X-API-Key, then X-Forwarded-For, then client IP, override with key_cb for per-user limits.

CSRF Protection, Built In

app.pl

```
$app->enable_csrf(
    secret => $ENV{CSRF_SECRET},
);

# Sets a csrf_token cookie;
# expects it echoed back via
# the X-CSRF-Token header
```

- Defaults to enforce => 'header' and secure => 0, override both before shipping to production.
- No secret passed here? Falls back to the secret given to PAGI::FastAPI->new(secret => ...).
- $c->csrf_token and $c->csrf_verify($token) are available on every request via Context.

Double-submit cookie pattern: the browser must mirror the cookie into the header itself, a plain <form> post can't do that without JS.

Bot Protection: Proof-of-Work

app.pl

```
$app->add_bot_protection(
    difficulty => 3,
    secret     => $ENV{BOT_SECRET},
);
# 401 + signed challenge until
# the client solves the PoW
# puzzle, client-side, in JS
```

- Unsolved requests get a 401 with a signed challenge, browsers solve it automatically and retry.
- difficulty controls how many leading zero bits the SHA-256 proof must have.
- Challenges are HMAC-signed and stateless, no shared cache or database required.

Binds each challenge to the client IP for defense in depth, put a trusted reverse proxy in front so it can't be spoofed.

Lifespan Events: Startup & Shutdown

app.pl

```
$app->on_startup(async sub {
    ... connect DB pool ...
});

$app->on_shutdown(async sub {
    ... close DB pool ...
});
```

Open the shop once in the morning, close it once at night, not per customer. Perfect for connection pools.

- on_startup runs once, before the app accepts any requests.
- on_shutdown runs once, as the server is stopping.
- We'll see this exact pattern again with DBIx::Class::Async.

WebSocket Support

app.pl

```
$app->websocket('/chat',
    handler => async sub ($ws, $deps) {
        await $ws->accept;
        await $ws->each_json(async sub {
            await $ws->send_json($_[0]);
        });
    }
);
```

- $ws is a PAGI::WebSocket (from PAGI::Tools), not a separate PAGI::FastAPI class.
- on_close, each_json, try_send_json, and keepalive are all built in, no hand-rolled receive loop needed.
- Same non-blocking model as HTTP routes, one await never blocks another connection.

Path params work the same way as HTTP routes, $ws->path_params->{room} extracts {room} from the URL.

Server-Sent Events (SSE)

app.pl

```
$app->get('/stream',
    handler => async sub ($c) {
        return $c->sse(async sub ($sse) {
            await $sse->keepalive(15);
            await $sse->send_json({ tick => 1 });
            await $sse->close;
        });
    });
```

- $sse is a PAGI::SSE (from PAGI::Tools), handed to your generator sub, not a bespoke stream class.
- send(), send_json(), send_event(), keepalive(), and close() are all built in.
- Same non-blocking model as everything else, one slow stream never blocks another connection.

$c->sse(...) works the same way, Context offers it directly, no separate response class to import.

Event Loops: Future::IO Is the Goal

app.pl

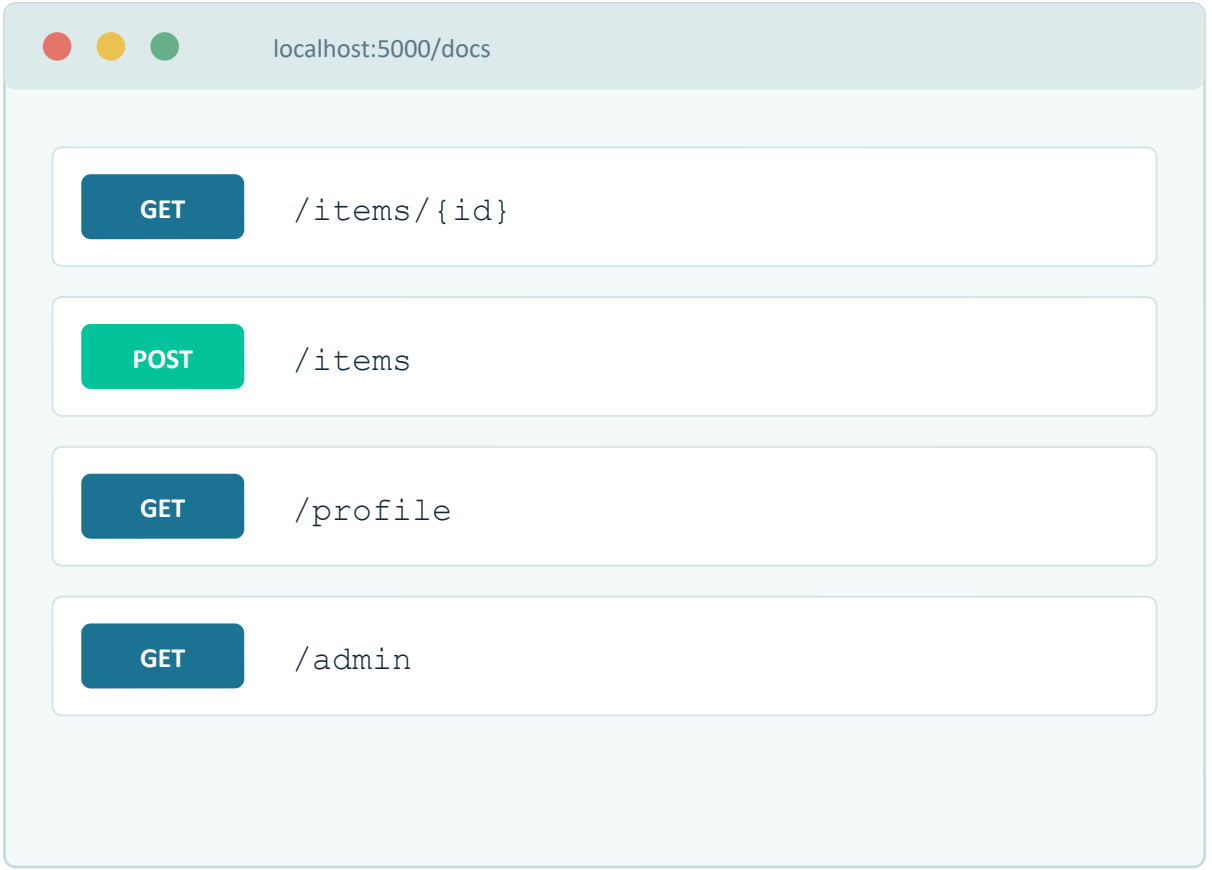
```
use Future::IO;
async sub heartbeat {
    while (1) {
        await Future::IO->sleep(30);
    }
}
heartbeat()->retain;
```

- PAGI is deliberately silent on the event loop, that's the server's business, not your app's.
- pagi-server happens to use IO::Async today. Treat that as an implementation detail, not a dependency.
- No loop object of your own to build, Future::IO->sleep delegates to whatever backend is already running.

Mixing in a Mojo::IOLoop library like Mojo::Pg? See the docs for the IO::Async::Loop::EV bridge; a fallback, not the default path.

Docs You Don't Have to Write

- GET /docs, an interactive Swagger UI, generated automatically.
- GET /openapi.json, the machine-readable spec behind it.
- Both are built from your actual routes and Type::Tiny constraints, always in sync with your code.



CHECKPOINT

Recap: Why PAGI::FastAPI?

Async & Non-Blocking

Built on the PAGI protocol from the ground up.

Type-Safe Validation

Type::Tiny checks query params and JSON bodies for you.

Dependency Injection

Reusable, composable pieces via Depends().

Automatic Docs

Swagger UI and OpenAPI JSON, always in sync.

One thing is conspicuously missing so far: authentication. Let's talk about that next.

PART 3

The Security Gap

Why isn't there a built-in @login_required?

SECURITY

No Built-In Auth, On Purpose

- Auth needs vary too much to bake in one approach: sessions? JWTs? API keys? OAuth2?
- Instead, you get two general building blocks: middleware (every request) and dependencies (per route).
- A dependency signals failure the same way any dependency does: set $c->status(code >= 400) and return a body.

Gotcha worth remembering

A dependency must ***not*** die() to signal failure that crashes the request with an uncaught exception instead of returning a clean 401.

Always: `$c->status(401); return { detail => "..." };`

A 30-MINUTE INTRODUCTION · BEGINNER FRIENDLY

PAGI::FastAPI & PAGI::FastAPI::Security

Building modern, async, type-safe, and secure; APIs in Perl

github.com/manwar/PAGI-FastAPI · github.com/manwar/PAGI-FastAPI-Security

PART 4

Enter PAGI::FastAPI::Security

A companion distribution of ready-made, Depends()-compatible authentication schemes, modelled on Python FastAPI's fastapi.security module.

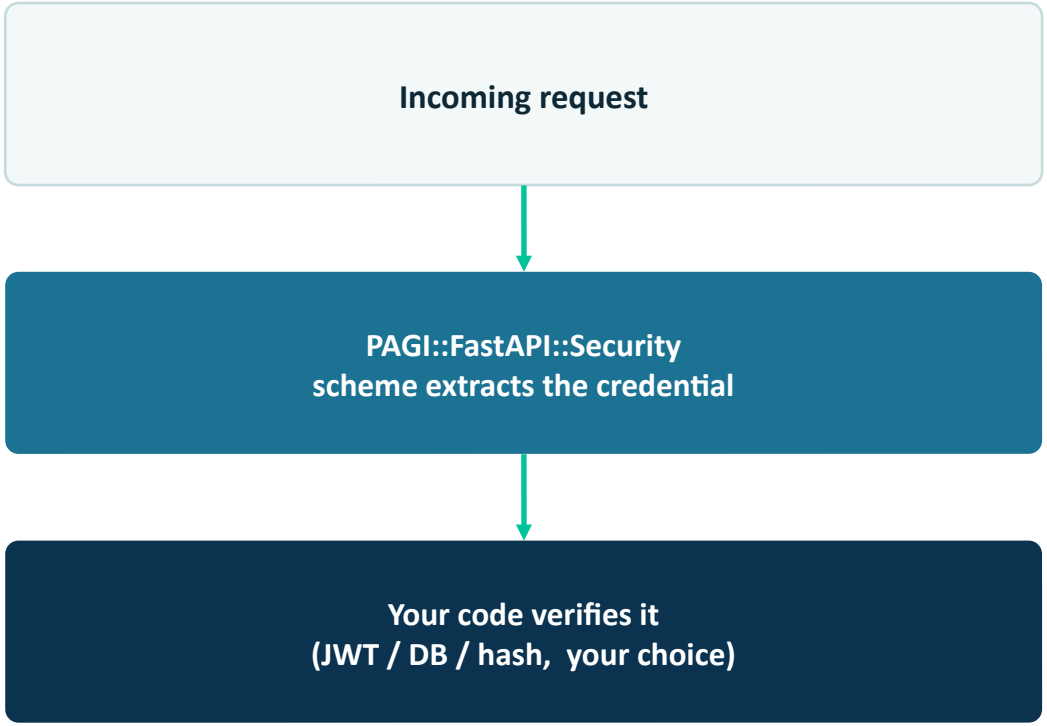
The Four Schemes at a Glance

Scheme	Extracts	On Failure
HTTPBearer	`Authorization: Bearer <token>`	`401 + WWW-Authenticate: Bearer`
HTTPBasic	`Authorization: Basic <base64>`	`401 + WWW-Authenticate: Basic`
APIKey	`header / query / cookie`	`403 Forbidden`
OAuth2::PasswordBearer	`Bearer token + token_url/scopes`	`401 + WWW-Authenticate: Bearer`

Same shape every time: extract the credential, and on failure short-circuit with the right HTTP status and challenge header.

Extraction, Not Verification

- Every scheme only pulls the credential out of the request, it never verifies it.
- You decide how to verify: JWT signature check, database lookup, password hash comparison.
- Reference point: exactly like fastapi.security in Python, it doesn't ship a JWT library either.



Code in Action: Bearer + Real Verification

app.pl

```
my $bearer = PAGI::FastAPI::Security::HTTPBearer->new;
$app->get('/items',
    dependencies => [
        $bearer->depends(key => 'token'),
        Depends(async sub ($c) {
            my $claims = eval { verify_jwt($c->stash->{token}) };
            unless ($claims) {
                $c->status(401);
                return { detail => 'Invalid token' };
            }
            return $claims;
        }, key => 'claims'),
    ],
    handler => async sub ($c) {
        return { user_id => $c->stash->{claims}{sub} };
    },
);
```

Every scheme also accepts auto_error => 0, for routes that behave differently for authenticated vs. anonymous requests.

DEVELOPMENT SNAPSHOT

Six New Modules in v1.2.0

TypedPath

Validates (and coerces) a path parameter the same way query/body validation already works, via Depends().

ResponseModel

with_response_model() checks a handler's return value against a Type::Tiny schema and filters extra fields.

Middleware::ExceptionHandler

Typed exception -> handler dispatch by blessed()/isa(), with a configurable default_handler fallback.

Redirect, File & Cookies

redirect_to(), file_response(), and cookie() round out the response and request-parsing helpers.

The last published release is v1.7.3.

Code in Action: TypedPath + ResponseModel

app.pl

```
use PAGI::FastAPI::TypedPath qw(TypedPath);
use PAGI::FastAPI::ResponseModel qw(with_response_model);
$app->get('/products/{item_id}',
    dependencies => [
        Depends(TypedPath('item_id', Int), key => 'item_id'),
    ],
    handler => with_response_model(
        { id => Int, name => Str },
        async sub ($c) {
            return My::DB->find_product($c->stash->{item_id});
        },
    ),
);
```

A schema mismatch here is a 500, not a 422, it's the handler's bug (e.g. a stray column from the DB row), not the caller's.

WRAP-UP

The Full Picture

Your App	Routes and business logic.
PAGI::FastAPI::Security	Plugs in via Depends(), ready-made auth extraction.
PAGI::FastAPI	Routing, validation, dependency injection, automatic docs.
PAGI protocol + pagi-server	The standard socket, and the process that runs it.

PAGI::FastAPI::Security doesn't change PAGI::FastAPI at all, it plugs into the same Depends() mechanism every other dependency uses.

FAQ

Common Questions

How does this compare to Python's FastAPI?

Same shape; routing, DI, type validation, auto docs, ported to async, type-safe Perl. PAGI is this stack's ASGI.

Do I need to learn IO::Async or Future::AsyncAwait deeply?

No, write async subs and await Futures. pagi-server owns the event loop; that's underneath your app, not part of it.

Is authentication built in?

No, by design, middleware and Depends() are the primitives. PAGI::FastAPI::Security adds ready-made schemes on top.

More questions? github.com/manwar/PAGI-FastAPI/issues, same repo linked on the last slide.

FAQ

A Few More Questions

What Perl version do I need?

5.38 or newer, the class feature is core to the framework. (Earlier docs said 5.36; fixed in v1.0.0.)

Can I swap out the in-memory rate-limit store?

Yes, implement RateLimit::Driver's two-method interface for Redis or CHI. Memory is just the shipped default.

Where do I report a bug or ask a question?

github.com/manwar/PAGI-FastAPI/issues, same repo linked on the last slide.

Still stuck? metacpan.org/pod/PAGI::FastAPI has the full API reference, including everything on these slides.

FAQ

Even More Questions (v1.7.0)

Is v1.7.0 on CPAN yet?

Yes, the latest published release is v1.7.3.

Does TypedPath replace query/body validation?

No, it's the path-parameter analog. Query and body validation via Type::Tiny, from Part 2, still work exactly as before.

Does ExceptionHandler replace $c->status()-and-return?

No, it complements it, for a real die() escaping a handler. It re-throws if no handler class matches and no default_handler is set.

Full v1.7.0 module docs, plus the Changes file, live at github.com/manwar/PAGI-FastAPI on the main branch.

THANK YOU

Questions?

- github.com/manwar/PAGI-FastAPI
- github.com/manwar/PAGI-FastAPI-Security
- metacpan.org/pod/PAGI::FastAPI
- metacpan.org/pod/PAGI::FastAPI::Security