OCaml Meeting 2026

モジュールシステムで組み立てる

Cloud Run アプリケーション

クラウドのコンテナ基盤で OCaml を動かしてみた

芳賀 雅樹 / @silasolla

自己紹介

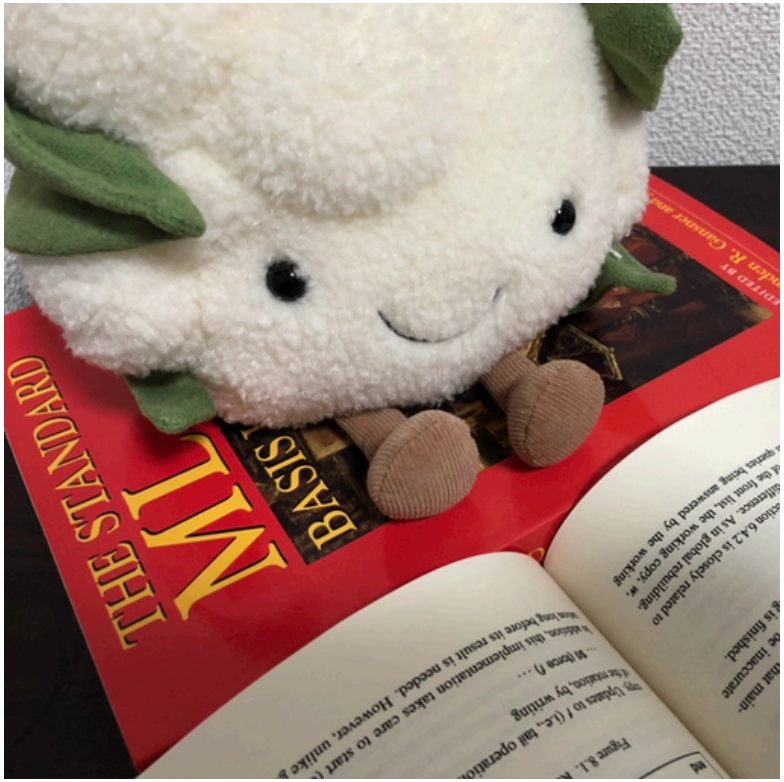
芳賀 雅樹 / @silasolla

Standard ML がすきです！

- ちゃんと OCaml の話をします！
- レギュレーション守ります！

3-shake Inc. で働いています！

- アプリ開発のモダナイゼーション支援をやっている
- トイルを無くすのが生業



インターネット近影

作ってみたもの


```
POST /api/chat  →  { "text": "...", "finishReason": "STOP" }
```

Cloud Run アプリが取得した認証情報で `generateContent` を利用

OCaml にした理由

本当は **Standard ML** で書きたかった

Basis Library の生 TCP ソケットから HTTP を自分でパースする気にはなれず

同じ ML でも **OCaml** はよく知らない

実践的な部分に触れながらキャッチアップする機会になればと思った

技術スタック

OCaml 5.4	現時点では 5.5 だと開発ツールが揃わない (e.g. `ocamlformat`)
dune 3.23	Dune Package Management で依存を固定してビルド
Nix flakes	dune や C ライブラリ (e.g. `pkg-config`, `gmp`) を入れる
Eio + cohttp-eio	Effect Handler の雑な興味 (Dream などは利用できない)
Cloud Run	コンテナとして scratch イメージで動作させたい

ライブラリの一覧

宣言	**port** インタフェースの提供	**domain** アプリとして扱う型	
実装	**adapter** 外部リソースの呼び出し	**handler** API の公開とハンドリング	**server** HTTP サーバの自前実装
補助	**config** 環境変数の読み取り	**log** ログ出力の設定	

これらを `main` で組み立てる

Cloud Run を想定して実装するには

環境変数で Listen するポートを指定される

構造化ログのために JSON ログを標準出力する

メタデータサーバから認証トークンを取得する

SIGTERM を捕捉して終了する (Graceful Shutdown)

これらをどう実装するか

Cloud Run によるポート指定

```
port = int_of_string (getenv "PORT" ~default:"8080")
```

getenv は Sys.getenv_opt のラッパー

```
Eio.Net.listen (Eio.Stdenv.net env)
  ~sw ~backlog:128 ~reuse_addr:true
  (`Tcp (Eio.Net.Ipaddr.V4.any, port))
```

環境変数 PORT で指定されるので Eio で Listen すればよい

実行環境での分岐

```
Logs.set_reporter
  (if Config.on_cloud_run ()
     then reporter json_emit
     else Logs_fmt.reporter ())

let auth : (module Port.AUTH) =
  if Config.on_cloud_run ()
    then (module Metadata_auth.Make (Http))
    else (module Local_auth.Make (Http))
```

Cloud Run が自動で設定した環境変数 K_SERVICE の有無で判定する

構造化ログの出力

```
let json_emit level msg =
  print_endline
    (Yojson.Safe.to_string
       (`Assoc
          [ ("severity", `String (severity_of_level level));
            ("message", `String msg) ]))
```

Cloud Logging に構造化ログとして解釈させたい

ログフィールドを含めて JSON 出力する

Logs ライブラリの reporter を実装

```
let report _src level ~over k msgf =
  let buf = Buffer.create 128 in
  let ppf = Format.formatter_of_buffer buf in
  let finish _ =
    Format.pp_print_flush ppf ();
    emit level (Buffer.contents buf);
    over (); k ()   (* ログ出力を報告し，継続を呼んで復帰 *)
  in
  msgf @@ fun ?header:_ ?tags:_ fmt -> Format.kfprintf finish ppf fmt
```

Buffer から読み出した内容を先ほどの Emitter で出力

Formatter を経由する仕組みなのでやや面倒

ログの書き込みと出力

```
Log.L.info (fun m ->
  m "%s %s" (Http.Method.to_string meth) path)
```

ローカルでの出力：

```
chat-server: [INFO] GET /api/chat
```

Cloud Run での出力：

```
{"severity":"INFO","message":"GET /api/chat"}
```

認証部分の抽象化

```
module type AUTH = sig
  val access_token : unit -> (string, error) result
end
```

トークンの取りかたが実行環境によって変わるので固定

利用する側はトークンをどうやって取ったか考えずに書ける

認証部分の実装

```
module Make (Http : Port.HTTP_CLIENT) : Port.AUTH = struct
  let cache = Token_cache.create ()
  let fetch () = (* ADC のリフレッシュトークンをアクセストークンと交換 *)
  let access_token () = Token_cache.get cache ~fetch
end
```

```
module Make (Http : Port.HTTP_CLIENT) : Port.AUTH = struct
  let cache = Token_cache.create ()
  let fetch () = (* Http.get でメタデータサーバから認証情報を取得 *)
  let access_token () = Token_cache.get cache ~fetch
end
```

TLS の設定はファンクタ適用したクライアントから

認証情報のキャッシュ

```
let get t ~fetch =
  let now = Unix.time () in
  match t.entry with
  | Some (token, expires_at) when now < expires_at -> Ok token
  | _ -> let* token, ttl = fetch () in  (* let* は Result.bind *)
         t.entry <- Some (token, now +. ttl);
         Ok token
```

トークンを取得して有効期限まで保持する

期限切れになったら再度 fetch する

SIGTERM を捕捉して終了する

```
let on_signals () =
  let promise, resolve = Eio.Promise.create () in
  let fired = ref false in
  let handle (_ : int) =
    if not !fired then (fired := true; Eio.Promise.resolve resolve ())
  in
  Sys.set_signal Sys.sigterm (Sys.Signal_handle handle);
  Sys.set_signal Sys.sigint (Sys.Signal_handle handle);
  promise
```

scratch で動かすならシグナルの処理が必要 (PID 1 問題)

SIGTERM や SIGINT が来たら promise を解決

停止条件として HTTP サーバに渡す

```
let run ?stop env ~port routes =
  let stop =
    match stop with
    | Some s -> s
    | None -> Shutdown.on_signals ()
  in
  Cohttp_eio.Server.run ~stop ~max_connections socket server
```

シグナルを捕捉したら HTTP サーバを止める

処理中のリクエストを待ってから終了する (Graceful Shutdown)

ルーティングの実装

```
type dispatch_result =
  No_path | Wrong_method | Found of Port.handler

let handle routes ~meth ~path ~body =
  match (dispatch routes meth path, meth) with
  | No_path, _ -> Response.error `Not_found
  | Wrong_method, `OPTIONS -> Response.preflight
  | Wrong_method, _ -> Response.error `Method_not_allowed
  | Found handler, _ -> run_handler handler body
```

パスがなければ 404 に，メソッドがなければ 405 にする

JSON のスキーマ変換

```
api.atd      type chat_request = { messages : message list }
gemini.atd   type gen_request  = { contents : gen_content list }
types.atd    type role = [ User | Model | System ]
```

公開する API や Gemini のやりとりする JSON を `Domain` の型に変換

ATD で記述すると `atdgen` で変換関数を生成できる

Handler の実装

```
let handle_chat body =
  let* req = parse body in
  let* resp = Llm.generate (to_domain req) in
  Ok (Api_j.string_of_chat_response
        { text = resp.text; finish_reason =
resp.finish_reason })
```

JSON を受け取って JSON を返す (HTTP に依存しない)

先ほどの `atdgen` で生成した `Api_j` の変換関数を利用できる

HTTP のエラーハンドリング

```
type error =
  | Auth_error of string
  | Parse_error of string
  | Upstream_error of string

let status_of_error (e : Port.error) : Http.Status.t =
  match e with
  | Port.Parse_error _ -> `Bad_request
  | Port.Auth_error _ -> `Internal_server_error
  | Port.Upstream_error _ -> `Bad_gateway
```

Handler が返した error を HTTP status に対応させる

リクエストボディと接続数の上限設定

```
let max_body_bytes  = 64 * 1024
let request_timeout = 30.
let max_connections = 128
```

cohttp-eio にデフォルトの上限値がないので指定する

body をメモリに展開して爆死しないよう (64KB × 128 接続で 8MB 程度)

リクエストボディの読み取り

```
let read_body req flow =
  match Http.Request.has_body req with
  | `No | `Unknown -> Ok ""  (* Handler のほうで落ちる *)
  | `Yes -> (
      let buf = Eio.Buf_read.of_flow flow ~max_size:max_body_bytes in
      match Eio.Buf_read.take_all buf with
      | s -> Ok s  (* バッファ超過など 413 や 400 を例外で返す *))
```

has_body で Content-Length と Transfer-Encoding があるかを確認する

take_all の前に body の終了位置がわかっていないといけない

接続が閉じて EOF が返るまで待つわけにはいかない

HTTPS クライアントの用意

```
let tls_config = (* システムの CA 証明書を読んで TLS 設定を作る *)

let https uri raw =
  let host = (* uri からホスト名を取り出す *) in
  Tls_eio.client_of_flow ?host tls_config raw

let client =
  Cohttp_eio.Client.make
    ~https:(Some https) (Eio.Stdenv.net Env.env)
```

Vertex AI の LLM と HTTPS で通信するのに TLS が必要

TLS 接続を `tls-eio` で作成して cohttp-eio に渡す

LLM 呼び出しの抽象化

```
module type LLM = sig
  val generate :
    Domain.request -> (Domain.response, Port.error) result
end
```

エンドポイントを組み立てたり JSON を変換したりする部分

外部と通信しない `Auth` や `Http` の実装を渡せばテストできる

アプリケーションの組み立て

```
let module Auth = (val auth) in
let module Llm =
  Adapter.Gemini_client.Make
    (struct let t = config end) (Auth) (Http)
in
let module Chat_feature = Handler.Chat.Make (Llm) in
let routes = Chat_feature.routes @ Handler.Health.routes in
Server.Listener.run env ~port:config.Config.port routes
```

module type を満たすモジュールをファンクタの適用で組み上げる

scratch イメージの構築

```
bin=/app/dist/bin/chat-server
roots=$(grep -aoE '/nix/store/[a-z0-9]{32}-[^/[:space:]"]*' "$bin" \
  | sort -u | grep -vE -- '-dev$|-gcc-[0-9.]+$')
closure=$(for r in $roots; do nix-store -qR "$r"; done | sort -u)
for p in $closure; do cp -a "$p" /rootfs/nix/store/; done
```

OCaml バイナリから store のパスを辿って共有ライブラリ依存を集める

`ldd` は `dlopen` で読み込むライブラリを返さないっぽい

Nix の成果物であれば `nix-store -qR` が使えそうだが (後述)

アプリ自体は 19MB だが glibc など 56MB ほど同梱 (musl は試していない)

起動時の警告

```
chat-server: [WARNING] Ignored 180 trust anchors.
```

ca-certs は OpenSSL 独自の TRUSTED CERTIFICATE を読まない

nixpkgs のバンドルでは S/MIME 専用のルートと 2004 年発行の 2 件が該当した

TLS で利用する証明書は読まれているので問題ない

依存の管理をどうしようか

```
dune.lock/logs.0.10.0.pkg                  （プロジェクト）
_build/.dev-tools.locks/utop/logs.pkg  （開発ツール）
```

Logs が二重定義になってリンカーまわりがバグる (dune utop が起動しない)

```
RUN nix develop --command sh -c 'dune build @install'
```

derivation の中の dune build はインターネットからソースを取得できない

ハッシュ取れば良いのかしら (fixed output derivation とかいうやつ)

dune と Nix のどちらに寄せるのが正解なのかあんましわかんない

所感

モジュールシステム，たのしい！！

継続渡しのライブラリに触れると，へんな筋肉を使う

SemVer じゃないから，コンパイラやライブラリの互換性が面倒かも

HTTP サーバやクラウドの認証まわりを自前で実装するのも

キャッシュが無いとビルドが長い (`dune tools` とかも)

実務で再現性あるナレッジ蓄積するのには難しい印象がある

最近 Go が 1.27 になったけど後方互換性あるし `CGO_ENABLED=0` で単一バイナリになるし