



DELPHIDAY

italian conference

DELPHI BEYOND BOUNDARIES

Microservizi REST scalabili con Object Pascal moderno



ANTONELLO CARLOMAGNO

WU73 SRL

@ a.carlomagno@wu73.com

in linkedin.com/in/antonellocarlomagno

DELPHIDAY

italian conference

9-10 Giugno 2026
Piacenza



PROGETTI

Retrocomputing
www.msxitalia.org

Turismo
www.ilturista.info

DELPHIDAY
italian conference

9-10 Giugno 2026
Piacenza

 **wintech**
italia



AGENDA

1. **Architettura a microservizi** con Delphi
2. **WiRL framework** per REST API
3. **PostgreSQL e connection** pool thread-safe
4. **NATS JetStream** per messaggistica asincrona
5. **Autenticazione JWT** e sicurezza API
6. **Integrazione servizi esterni** e webhook
7. **Performance tuning** con Indy HTTP
8. **Deploy Docker** su cloud DigitalOcean
9. **Infrastruttura DigitalOcean** managed (VPC, LB, Spaces, firewall)
10. **Dashboard HTMX** senza frontend framework
11. **Monitoring e alerting** con un microservizio gateway
12. **Stack tecnologico** e strumenti API (Bruno, OpenAPI)
13. **Sicurezza** e difesa in profondita'
14. **Lezioni apprese** e best practice



Architettura a microservizi con Delphi

1



Architettura a microservizi

Perche' microservizi in Delphi?

Delphi produce eseguibili nativi, leggeri e veloci. Ogni servizio e' un singolo .exe autonomo, senza runtime o dipendenze esterne.

La nostra architettura

- 10 microservizi core indipendenti (+1 un gateway di alerting sms-bridge), ciascuno con una responsabilita' specifica
- DigitalOcean Load Balancer: SSL termination (Let's Encrypt managed) + bilanciamento
- NGINX come API Gateway interno: routing path-based verso i servizi
- Ogni servizio ascolta su una porta dedicata
- Comunicazione sincrona via REST, asincrona via NATS JetStream

Vantaggi chiave

- Deploy indipendente di ogni servizio; scaling orizzontale su print e notification
- Fault isolation e footprint minimo (ogni .exe pesa pochi MB)



Architettura a microservizi

Servizio	Responsabilita'
Auth	OTP via SMS + JWT
User	Profili e rubrica
Credit	Crediti e pagamenti
Postcard	Creazione e invio
Payment	Gateway NEXI
Notification	Push FCM + Email
Print	Stampa
Log	Logging centralizzato + export CSV
Dashboard	Pannello B2B
Admin	Monitor, template, business, log



WiRL framework per REST API

2



WiRL framework per REST API

Cos'e' WiRL?

WiRL e' un framework REST open source per Delphi. Permette di creare API RESTful con attributi e convention-over-configuration.

```
[Path('/postcards')]
TPostcardResource = class
  [GET, Path('/{id}')]
  [Produces(TMediaType.APPLICATION_JSON)]
  function GetPostcard([PathParam] id: string): TPostcard;

  [POST]
  [Consumes(TMediaType.APPLICATION_JSON)]
  function CreatePostcard([BodyParam] AData: TJSONObject): TJSONObject;
end;
```



WiRL framework per REST API

Componenti principali

- **Engine + Application:** configurazione server e routing
- **Resource classes:** endpoint REST con attributi [Path], [GET], [POST]
- **Filters:** middleware per autenticazione, logging, encoding
- **Neon serializer:** serializzazione automatica oggetti Delphi in JSON

Pattern adottati

- Un resource class per dominio (Users, Postcards, Credits...)
- Filtri [PreMatching] per JWT validation e UTF-8 encoding
- Separazione netta: Resource → Service → Repository
- Ogni servizio espone il proprio set di endpoint REST
- 79 end point divisi per servizio



PostgreSQL e connection
pool thread-safe

3



PostgreSQL e connection pool

Il problema

Un server HTTP multi-thread non puo' condividere una singola connessione database. Serve un pool che gestisca connessioni concorrenti in modo sicuro.

La soluzione: connection pool custom

```
TDatabasePool = class
  FPool: TThreadList<TPgConnection>;
  FSemaphore: THandle; // limita connessioni simultanee
  function Acquire: TPgConnection;
  procedure Release(AConn: TPgConnection);
end;
```



PostgreSQL e connection pool

Caratteristiche

- **Thread-safe:** ogni thread ottiene la propria connessione
- **Semaforo:** limita il numero massimo di connessioni attive
- **Lazy creation:** le connessioni vengono create on-demand
- **Health check:** validazione connessione prima del riutilizzo
- **pgDAC (Devart):** driver PostgreSQL nativo, no ODBC, no dbExpress

Database gestito

- PostgreSQL su DigitalOcean Managed Database
- 27 tabelle, 12 funzioni SQL custom; migrazioni numerate (001-024)
- UseUnicode := True per supporto completo UTF-8 e emoji
- 1 Nodo in stand-by
- 1 Nodo read-only



NATS JetStream per messaggistica asincrona

4



NATS JetStream

Perche' messaggistica asincrona?

Stampare e spedire una cartolina dipende da servizi esterni e richiede secondi. Il backend accoda il lavoro e risponde subito; worker indipendenti lo eseguono in background.

Core NATS e JetStream: due meccanismi

- **Core NATS (pub/sub + queue group):** consegna live, fire-and-forget. Job di stampa, notifiche, campagne B2B.
- **JetStream (stream + consumer pull):** persistenza su disco, consumo in batch. Logging centralizzato.
- *Persistenza dove serve (i log), velocita' dove basta (i job): una scelta, non una dimenticanza.*

Perche' NATS, non Kafka o RabbitMQ

- Singolo binario, parte in millisecondi, pochi MB di RAM
- Protocollo testuale: client NATS scritto in Delphi, senza wrapper C
- Queue Groups integrati: load balancing senza infrastruttura esterna



NATS JetStream

Queue Groups: lo scaling e' un parametro

- Stesso subject + stesso gruppo: NATS consegna ogni messaggio a un solo worker, a rotazione
- **print-workers, notification-workers, b2b-campaign-workers**
- docker compose --scale print-service=3 -- zero codice di load balancing

Client NATS custom (WauCard.NATS.pas)

- Client TCP nativo in Delphi, non un binding C: protocollo testuale, handshake PING/PONG
- Publish core e JetStream con ACK, QueueSubscribe, pull; riconnessione 5s su thread dedicato
- Gia' cluster-aware: failover su piu' nodi senza toccare il codice applicativo

E se un job si perde?

- **Job (core NATS):** rigenerabili -- lo stato resta nel DB, il job si ripubblica
- **Log (JetStream):** persistiti su disco, recuperati al riavvio del log-service



Autenticazione JWT e sicurezza API

5



Autenticazione JWT e sicurezza

Flusso di autenticazione

1. Utente invia numero di telefono
2. Backend genera OTP e lo invia via SMS (SMSAPI)
3. Utente conferma OTP
4. Backend genera coppia JWT: `access_token` + `refresh_token`
5. Ogni richiesta successiva include il token nell'header

JWT in Delphi

- **Libreria JOSE-JWT** (integrata in WiRL)
- Access token: durata 1 ora (configurabile via env)
- Refresh token: durata 7 giorni
- Claims custom: `user_id`, ruolo, scadenza



Autenticazione JWT e sicurezza

Filtro di autenticazione

```
[PreMatching]  
[Priority(TWiRLPriorities.AUTHENTICATION)]  
TJWTAuthFilter = class(TInterfacedObject,  
    IWiRLContainerRequestFilter)  
end;
```

- Intercetta ogni richiesta prima del routing; valida token, estrae claims
- Rotte pubbliche escluse via whitelist (login, OTP, webhook)

Sicurezza aggiuntiva

- NGINX: rate limiting, HTTPS, header security
- Webhook NEXI: validazione MAC SHA1; Print: HMAC-SHA256
- Google Cloud Vision: moderazione immagini prima della stampa



Integrazione servizi esterni e webhook

6



Integrazione servizi esterni

Servizio	Scopo	Protocollo
SMSAPI	Invio OTP via SMS	REST API
TurboSMTP	Email transazionali	REST API
Print	Stampa e spedizione cartoline	REST + Webhook
NEXI XPay	Pagamenti Pay-by-Link	REST + Webhook
Firebase FCM	Push notification	REST API
DigitalOcean Spaces	Storage immagini (S3)	AWS S3 API
Google Cloud Vision	Moderazione immagini	REST API



Integrazione servizi esterni

Pattern di integrazione

- RESTRequest4Delphi come client HTTP unificato (modalita' NetHTTP, no OpenSSL)
- Ogni integrazione isolata nella propria unit Service
- Retry con backoff per chiamate esterne
- Logging di ogni chiamata esterna per debug

Webhook: il backend come ricevitore

- NEXI → POST /webhook/nexi → Valida MAC → Accredita crediti
- Print → POST /webhook/print → Valida HMAC → Aggiorna stato
- Validazione crittografica e idempotenza di ogni webhook in ingresso
- Risposta immediata 200 OK, elaborazione nel contesto della richiesta



Performance tuning con Indy HTTP

7



Performance tuning con Indy

Indy come server HTTP

Delphi + WiRL usa Indy (TIdHTTPServer) come server HTTP embedded. Nessun web server esterno necessario per il servizio stesso.

Parametri di tuning

```
FServer.SetPort(8004).SetThreadPoolSize(75);  
IdHTTPServer.ListenQueue := 200;  
IdHTTPServer.KeepAlive := True;  
IdHTTPServer.MaxConnections := 200;  
IdHTTPServer.TerminateWaitTime := 5000;
```

Risultati stress test

Tool: bombardier -n 2000 -c 100

Risultato: 100% success rate, nessuna connessione rifiutata

Latenza stabile sotto carico



Performance tuning con Indy

Ottimizzazioni applicate

- Thread pool dimensionato: 75 thread per servizio
- ListenQueue alta: 200 connessioni in coda accettate
- KeepAlive attivo: riutilizzo connessioni TCP
- Connection pool DB: evita colli di bottiglia sul database
- NGINX upstream: distribuzione tra istanze multiple

Perche' Delphi performa bene

- Codice compilato nativo, no garbage collector
- Memoria gestita manualmente: predicibile e veloce
- Footprint ridotto (poche decine di MB) e startup istantaneo



Deploy Docker su cloud DigitalOcean

8



Deploy Docker su DigitalOcean

Containerizzazione

Ogni microservizio ha il proprio Dockerfile. Il binario Delphi compilato viene copiato in un container minimale.

Stack di deploy

DigitalOcean Load Balancer (SSL Let's Encrypt managed)

Droplet (Ubuntu) - Docker Compose

nginx (routing HTTP path-based interno)

auth / user / ... (10 servizi core)

nats (JetStream)

[PostgreSQL su Managed DB]

12 container = 10 microservizi + NGINX (gateway) + NATS (broker). PostgreSQL e' esterno (Managed Database, non containerizzato).



Deploy Docker su DigitalOcean

Docker Compose

- Un file docker-compose.yml orchestra tutti i servizi
- Variabili d'ambiente da file .env
- Volumi per persistenza log e configurazioni
- Health check per restart automatico
- Network isolata per comunicazione inter-servizio

CI/CD

- Compilazione locale (Delphi richiede Windows)
- Upload binari e deploy via script
- Rollback rapido: riavvia il container con la versione precedente

Registry DigitalOcean

- Container Registry privato



Infrastruttura DigitalOcean

9



Infrastruttura DigitalOcean

Componente	Specifiche	Ruolo
Droplet app	4 vCPU / 8 GB	esegue i 12 container
Droplet monitor	1 vCPU / 1 GB	stack monitoring isolato
Managed PostgreSQL 17	db-s-1vcpu-2gb + pool	dati + connection pooling
Load Balancer	Small, SSL managed	HTTPS termination + HA
VPC	10.10.0.0/16	rete privata interna
Cloud Firewall x2	regole per-tag	sicurezza perimetrale
Reserved IP	statico	indirizzo stabile
Spaces + CDN	object storage S3	immagini cartoline
Let's Encrypt	managed	TLS auto-rinnovato
Monitoring + Backups	alert + snapshot	osservabilita' nativa



Infrastruttura DigitalOcean

Scelte chiave

- **VPC:** traffico tra microservizi e verso il DB resta nella rete privata
- **Cloud Firewall come unica sorgente di verita':** niente firewall locale duplicato sui droplet
- **Load Balancer gestisce il TLS:** i servizi parlano solo HTTP interno

Il costo? ~115€/mese

Infrastruttura production-grade completa - droplet, DB gestito, load balancer, storage, monitoring.



Dashboard HTMX senza frontend framework

10



Dashboard HTMX

Il problema dei framework frontend

React, Angular, Vue richiedono build pipeline, node_modules, bundler e competenze JavaScript avanzate. Per un pannello di gestione B2B, e' overengineering.

HTMX: hypermedia come alternativa

```
<button hx-post="/campaigns/start"
        hx-target="#status"
        hx-swap="innerHTML">
```

Avvia Campagna

```
</button>
```

- Il server risponde con frammenti HTML, non JSON
- Il browser sostituisce pezzi di pagina senza reload; zero JS custom

Stack frontend minimale

- HTMX (interattività senza JS) + Tailwind CSS via CDN
- Template HTML generati lato server in Delphi



Dashboard HTMX

Funzionalità' della Dashboard B2B

- Gestione liste contatti (upload CSV, CRUD)
- Creazione e monitoraggio campagne bulk mailing
- Invio cartoline singole con preview
- Upload immagini con anteprima real-time
- Gestione codici redeem con banner personalizzati
- Monitoraggio stato ordini di stampa

Vantaggi

- Una sola codebase: backend Delphi genera tutto, nessuna API JSON separata
- SEO-friendly e accessibile di default
- Sviluppo rapidissimo: aggiungi un endpoint, restituisci HTML



Monitoring e alerting in produzione

11



Monitoring e alerting

Il problema

Il monitor interno cade insieme al droplet che dovrebbe sorvegliare. Serve un osservatore esterno e indipendente, su un droplet dedicato.

Stack di monitoring

- **Uptime Kuma:** ~15 monitor (servizi interni VPC, endpoint HTTPS pubblici, TCP NATS)
- **Apprise:** gateway notifiche, fan-out a Telegram + Email + SMS
- **Caddy:** reverse proxy con IP whitelist (admin) e status page pubblica per i clienti



Monitoring e alerting

sms-bridge: l'11° microservizio Delphi

Apprise non ha un plugin nativo per SMSAPI (provider IT). Soluzione: un microservizio WiRL minimale che fa da gateway.

Apprise (json://) → sms-bridge (Delphi/WiRL) → SMSAPI

- Stesso pattern Resource/Service degli altri servizi, binario ~9 MB
- Sanitizza i link (anti-phishing SMS), forza IPv4 verso il provider

Alerting dal backend: WauCard.Alerts.pas

Alerts.Notify(atOpsCritical, 'Print job fallito', ABody);

- Helper shared fire-and-forget (worker thread + coda interna)
- Routing per gravita': solo failure → SMS; tutto → Telegram + Email



Stack tecnologico

12



Stack tecnologico

REST / HTTP

- **WiRL**: framework REST stile JAX-RS (resource con attributi, filtri, content negotiation)
- **Indy**: server HTTP embedded in ogni servizio. **RESTRequest4Delphi**: client verso le API esterne (NetHTTP, no OpenSSL)

Database

- **pgDAC (Devart)**: driver PostgreSQL nativo, usato nel connection pool thread-safe
- **SecureBridge (Devart)**: SSL/TLS per pgDAC senza OpenSSL di sistema

JSON, Auth e Cloud

- **Neon**: JSON e oggetti Delphi. **JOSE-JWT**: access + refresh token
- **AWS SDK for Delphi**: client S3 per DigitalOcean Spaces (upload immagini, presigned URL)

Nats.Delphi (esteso x JetStream)

- **WauCard.NATS.pas**: *client NATS nativo in Object Pascal, non un binding C*



Stack tecnologico

Bruno -- collezione API versionata in git

- Client API open source (alternativo a Postman): le richieste sono file .bru versionati nel repo
- **79 richieste** organizzate per i 10 microservizi, in tools/bruno/waucards/
- Due environment (local / production) -- da locale a produzione con un click
- Auth supportate: Bearer JWT, X-API-Key, cookie admin_session, firma webhook

Pipeline Bruno -> OpenAPI -> Redoc

- Le collezioni .bru generano una spec OpenAPI 3.0, poi doc HTML statica con Redoc
- Output: backend.html (79 endpoint), mobile.html (36 endpoint)
- ***Una sola fonte di verita': la documentazione non diverge mai dal codice***



Sicurezza: difesa in
profondita'

13



Sicurezza

Strati concentrici: se un livello cede, quello sotto regge

Rete > TLS > Gateway > Auth > Applicazione > Database > Dato

I controlli, strato per strato

- **Rete:** Cloud Firewall default-deny (per tag) + VPC privata 10.10.0.0/16
- **TLS ovunque:** Let's Encrypt gestito sul Load Balancer + connessioni DB in SSL
- **Gateway NGINX:** rate limiting per zona (auth 10 r/s anti brute-force) + security headers
- **Autenticazione:** niente password -- OTP via SMS + JWT (access 1h / refresh 7g)
- **Webhook firmati:** MAC SHA1 (NEXI), HMAC-SHA256 (Print), validati prima di agire
- **Database:** Managed, mai pubblico; Trusted Sources per tag + pool centralizzato
- **Bakcup:** Snapshot giornalieri a caldo



Sicurezza

Dato, privacy e segreti

- **Dati personali fuori dai log:** solo UUID nei log, mai nomi, email o indirizzi
- **Immagini** su Spaces via presigned URL a scadenza (no bucket pubblico);
- **Moderazione** AI con Google Vision prima della stampa
- **Segreti in .env non versionato:** nessun token nel codice, nei log o nella documentazione

Lezioni dal campo (incident real)

- **Doppio firewall = trappola:** UFW locale con IP obsoleti ci ha chiusi fuori dall'SSH. Soluzione: una sola sorgente di verità
- **Default-deny verificato:** porte non whitelisted irraggiungibili anche se in ascolto
- *La sicurezza non è configurata una volta: è mantenuta, e si verifica.*



Lezioni apprese e best practice

14



Lezioni apprese e best practice

Cosa ha funzionato

- Separazione netta Resource/Service/Repository per ogni microservizio
- Unit condivise (server/shared/src/): config, database, JWT, logger, NATS
- NATS per disaccoppiamento: print e notification scalano indipendentemente
- HTMX per dashboard: velocita' di sviluppo imbattibile
- Migrazioni SQL numerate: evoluzione schema controllata e ripetibile

Cosa ho imparato (a volte nel modo difficile)

- Encoding UTF-8: WiRL + Indy richiedono filtri [PreMatching] dedicati per emoji
- Connection pool: dimenticare UseUnicode causa corruzione dati silenziosa
- Webhook security: mai fidarsi, sempre validare la firma crittografica



Lezioni apprese e best practice

Delphi nel 2026: ha senso?

- Compilazione nativa, performance eccellente
- Ecosistema maturo: WiRL, pgDAC, Indy, JOSE-JWT
- Comunita' attiva e librerie open source in crescita
- Ideale per backend dove servono velocita', basso consumo risorse, stabilita'
- Il punto debole: tooling CI/CD (compilazione richiede Windows + IDE)

Delphi non e' legacy. E' una scelta consapevole.

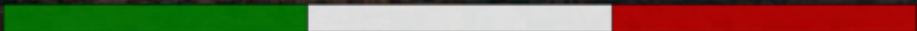


Q&A

Avete qualche domanda?

An aerial photograph of a city, likely Rome, showing a river (the Tiber) flowing through the center. The city is densely packed with buildings, and a large square with a prominent building is visible in the lower center. The image is overlaid with a semi-transparent dark layer.

DELPHIDAY


italian conference

THANK YOU