Ralph Hees

Managing APIs Effectively

API Management

Ralph Hees

Consultant & Architect
Software, Cloud, CICD, Enterprise
Hobby: Zeilen

Hoeveel API’s gebruikt een developer gemiddeld op een dag?

Vraag

15 tot 50 API’s Dagelijks

Vraag

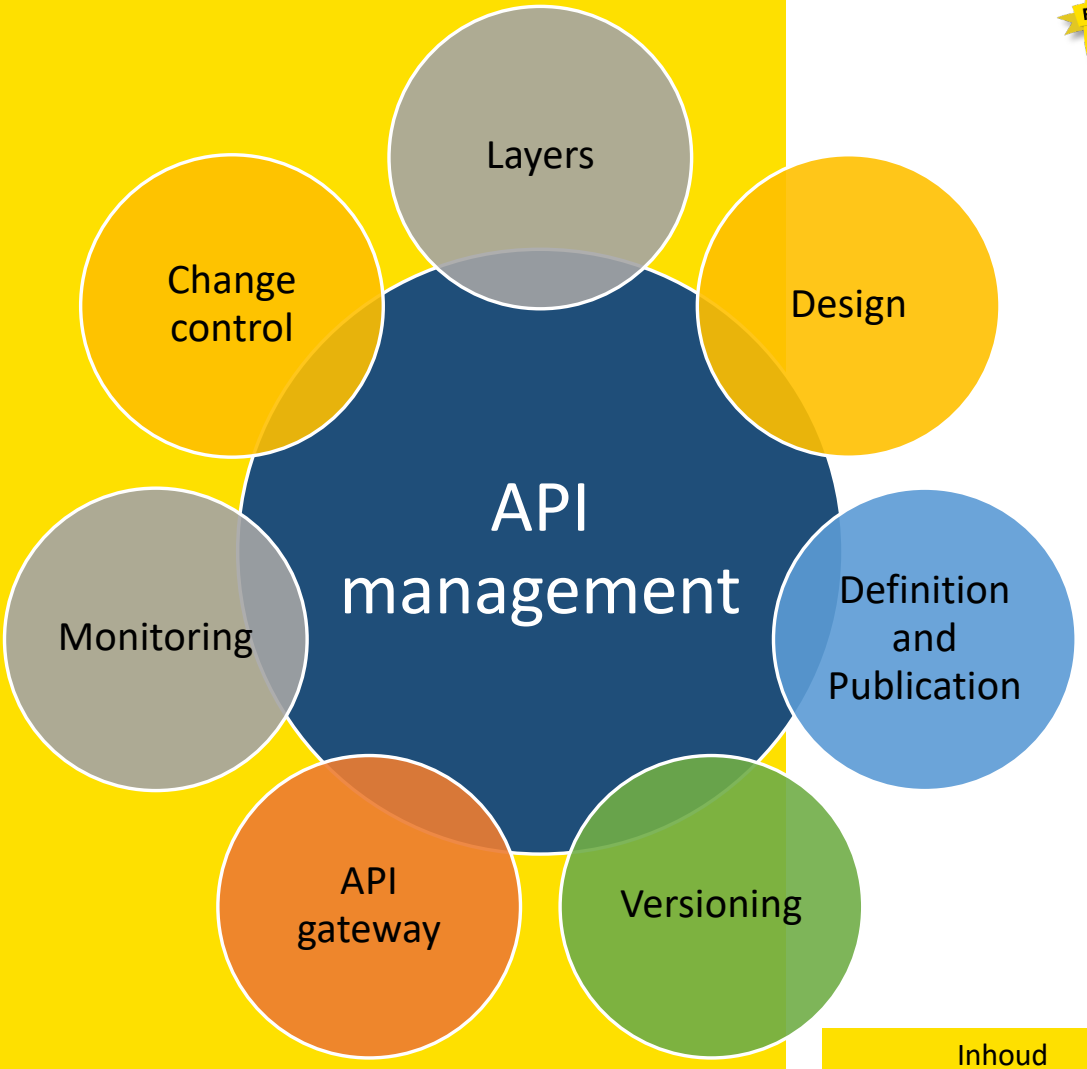
INHOUD

1.API Lagen

2.Management technieken

3.Gebruik Best Practices

4.Test methoden



API soorten

Typen van API Protocollen and Interfaces

1. Web-Based API Protocols
2. Library APIs
3. Platform APIs
4. Hardware APIs
5. System APIs

API soorten

Web-Based API Protocollen

a) REST (Representational State Transfer)

b) SOAP (Simple Object Access Protocol)

c) GraphQL

d) gRPC (Google Remote Procedure Call)

e) WebSockets

f) MQTT (Message Queuing Telemetry Transport)

g) AMQP (Advanced Message Queuing Protocol)

h) OData (Open Data Protocol)

i) XML-RPC

API soorten

Platform & Library APIs

Herbruikbare functies en libraries binnen specifieke programmeer omgevingen.

a) Java APIs & Libraries
b) .NET APIs & Libraries
c) Python APIs & Libraries
d) JavaScript & Node.js APIs
e) Mobile & Cloud APIs

API soorten

Hardware and Oprating Systems

a) Operating System APIs
b) Graphics & Multimedia APIs
c) AI & Machine Learning APIs

API soorten

Conclusie

Wide range

Web based

Programming

Hardware

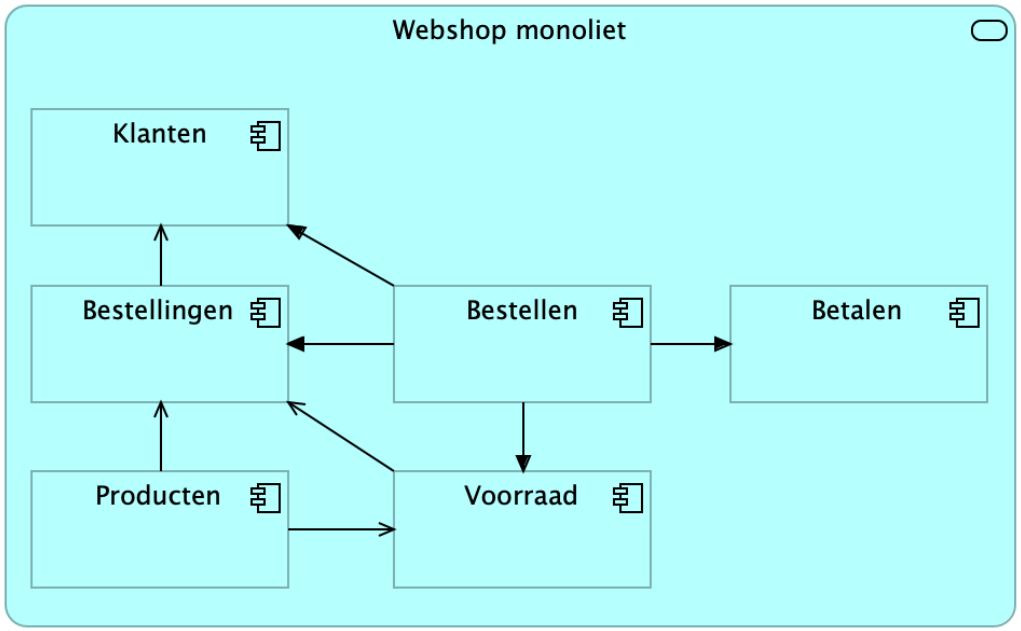
Cloud computing

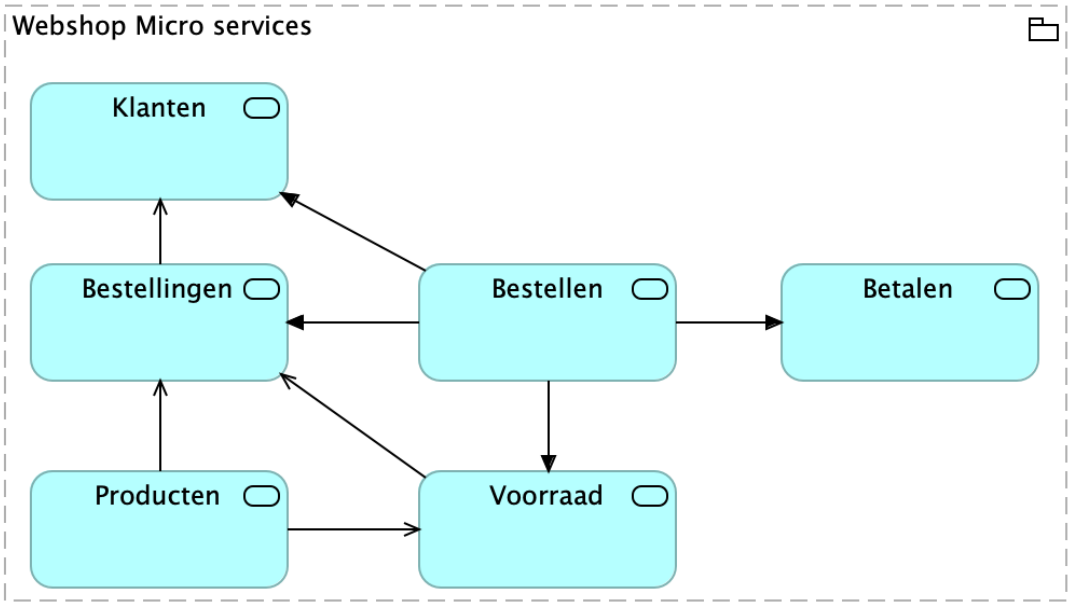
Mobile

AI

Structuren

Monoliet vs Micro services





Kenmerk	Monolithische Architectuur	Microservices Architectuur
Definitie	Enkele codebase, nauw verbonden	Kleine applicaties met API’s
Complexiteit	In begin eenvoudig, na groei moeilijker.	Complexer in ontwikkeling en beheer
Schaalbaarheid	Enkel hele systeem	Elke service
Deployment	Hele applicatie	Los, Continues Delivery
Fouttolerantie	Een enkele fout kan de hele applicatie laten uitvallen.	Fout in één service heeft geen invloed op het hele systeem.
Prestaties	Sneller in een enkel-serveromgeving, maar kan trager worden naarmate de applicatie groeit.	Overhead inter service communicatie, maar efficiënter op grote schaal.
Ontwikkelingssnelheid	Snelle start, maar vertraagt naarmate de applicatie groter wordt.	Langzamere initiële ontwikkeling, maar snellere updates.
Technologische flexibiliteit	Beperkt tot één technologie-stack.	Verschillende services kunnen verschillende technologieën gebruiken.
Onderhoud	Kan moeilijk te beheren worden naarmate de codebase groeit.	Eenvoudiger te onderhouden.

Structuur

API lagen

1. Presentation Layer APIs
2. Business Logic Layer APIs
3. Data Layer APIs
4. Infrastructure Layer APIs
5. Library and Interface APIs

API Lagen

Generiek door alle lagen

1. Design
2. Documentatie
3. Versionering & Compatibaliteit
4. Authententicatie & Authorisatie
5. Monitoring & Performance analytics
6. Error Handling & Logging
7. Testing & Validatie
8. Lifecycle Management & Deprecatie

Design

Generiek door alle lagen

Heldere design principes

Best Practices:

1. Design first approach
2. Domain Driven Design
3. Event storming
4. API reference
5. Ensure backwards compatibility

Versionering & Compatibaliteit

Generiek door alle lagen

API’s evolve

Version management verschilt per laag

Best Practices:

1. Semantic versioning.
2. Ondersteun oude versies een tijd.
3. Ondersteun meerder versies, indien nodig.

Authenticatie & Autorisatie

Generiek door alle lagen

Controleer toegang

Web API’s Oauth, WebauthN, JWT, API Keys

Best Practices:

1. Gebruik RBAC of Permissie gebaseerde beveiliging
2. Gebruik IAM (Identity and Access Management)
3. Implementeer Oauth 2.0 of WebAuthN

Monitoring & Performance Analytics

Generiek door alle lagen

Monitor uptime, latency, error waardes & gebruiks patronen.

Best Practices:

1. Observability platform (logging & tracing).
2. SIEM (Security Information en Event Management).
3. Rate limiting.
4. API Gateway
5. Optimaliseer performance gebaseerd op statistieken.

API begrip

Error Handling & Logging

Generiek door alle lagen

Consistent en duidelijke foutmeldingen

Best Practices:

1. Gebruik gestrictureerde error meldingen
2. Implementeer resilency (retry mechanismen)
3. Sla logs central op (debugging & auditing)

Testen & Validatie

Generiek door alle lagen

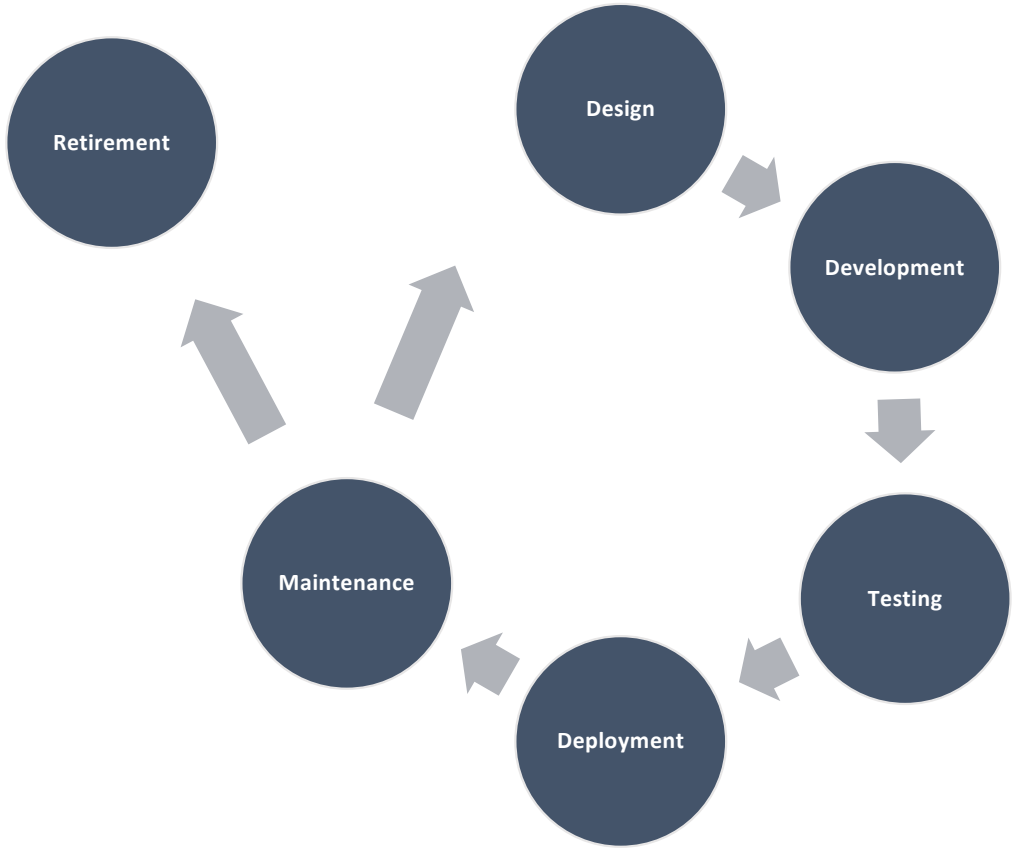
API’s moeten getest worden.

Best Practices:

1. Contract testing.
2. API testing.
3. Schema validatie.
4. Automatische test in CI/CD pipelines.

Lifecycle Management & Deprecatie

Generiek door alle lagen



API begrip

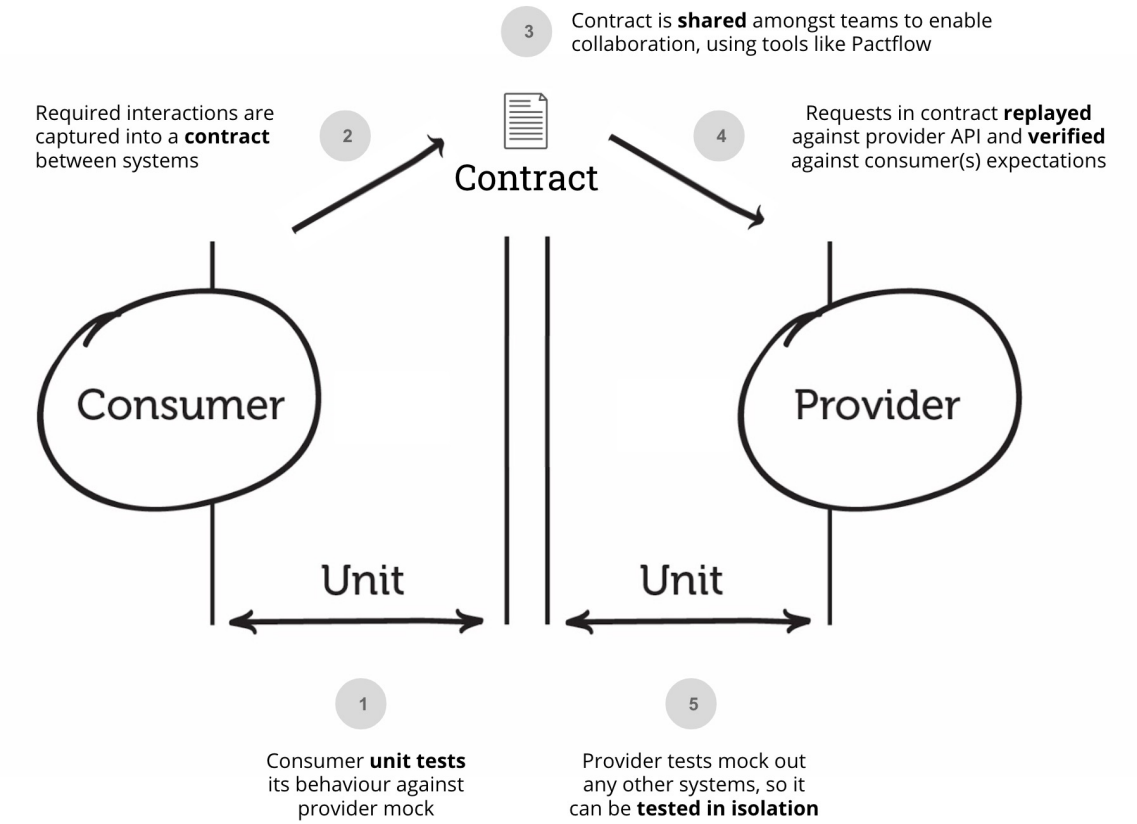
Lifecycle Management & Deprecatie

Generiek door alle lagen

Best Practices:

1. **Onderhoud API registries.**
2. **Track binnenkomende en uitgaande dependencies.**
3. **Plan changes.**
4. **Publiseer deprecatie tijdlijnen.**
5. **Automatiseer deprecatie process**
    1. **Feature flags**
    2. **API gateways**

Contract testing



Contracts

Contract testing

Voordelen

1. Consistentie controle.
2. Verminderen risico op integratie problemen.
3. Verminderen cloud Kosten.
4. Sneller valideren.
5. Inzicht in gebruik.

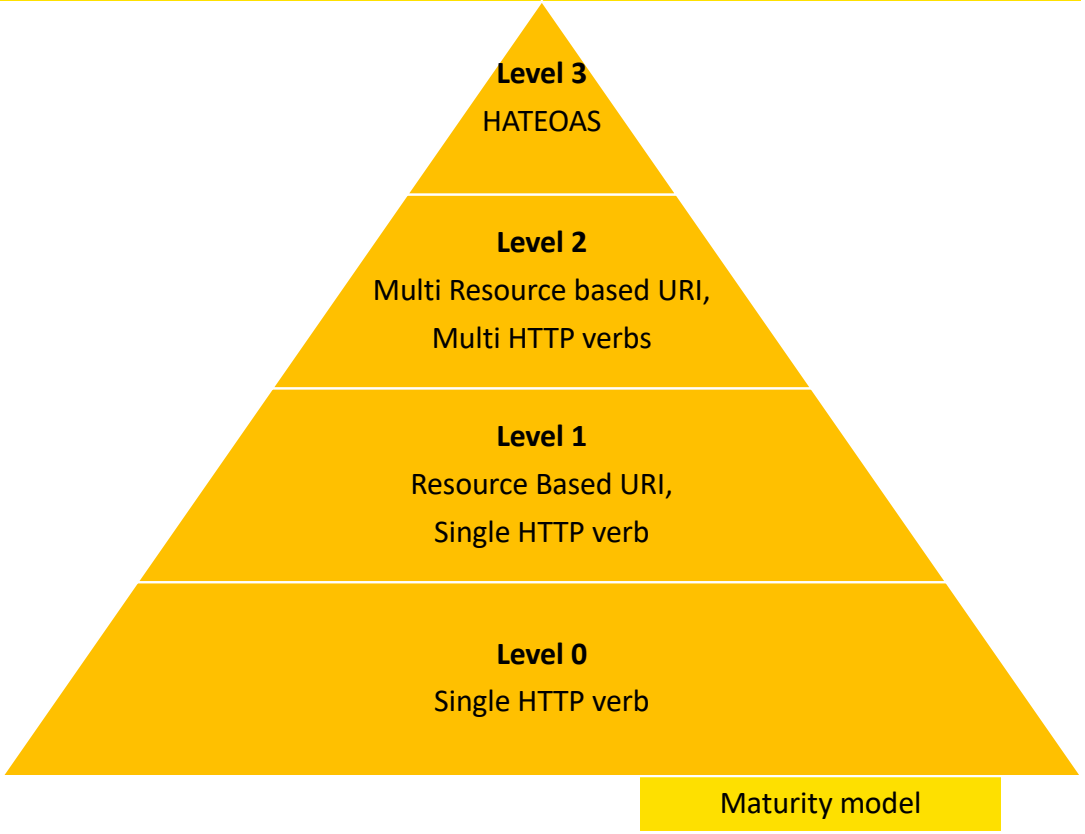
Contract testing

Aanpak

1. Consumer driven (CDCT) Assertible (OpenAPI specs)
2. Producer driven (PDCT)
    1. OpenAPI
    2. gRPC with Protobuf contracts
3. Usable in both or bidirectional
    1. Pact (wide range of languages supported)
    2. Spring Cloud Contract (YAML/Groovy, can interact with Pact)
    3. Specsmatic (OpenAPI specs, HTTP & Event-Driven)
    4. Contract Testing in Postman (OpenAPI specs)
    5. Dredd (OpenAPI specs)

REST Maturity model

Richardson Maturity Model (RMM)



REST Maturity model

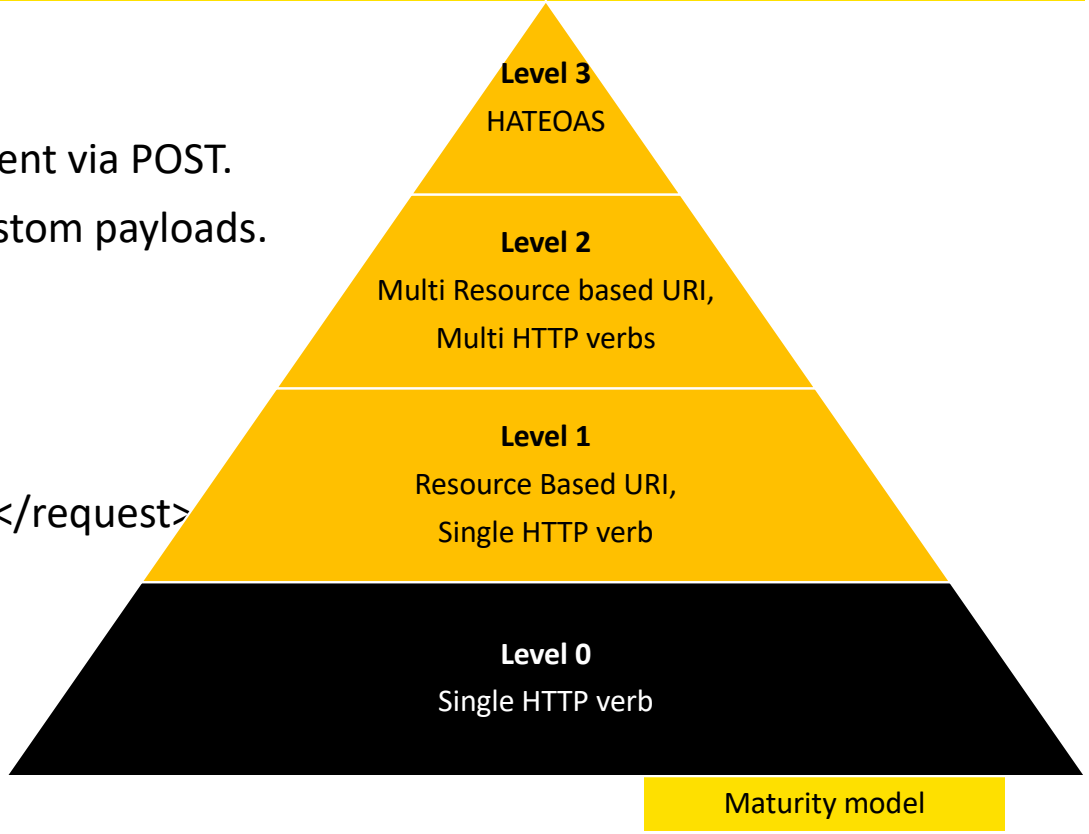
Level 0 The Swamp of POX

- Uses single endpoint (e.g., /api or /service).
- No proper use of HTTP methods; everything is sent via POST.
- No standard response formats, often XML or custom payloads.

POST /api

Content-Type: application/xml

<request><action>getUser</action><id>123</id></request>

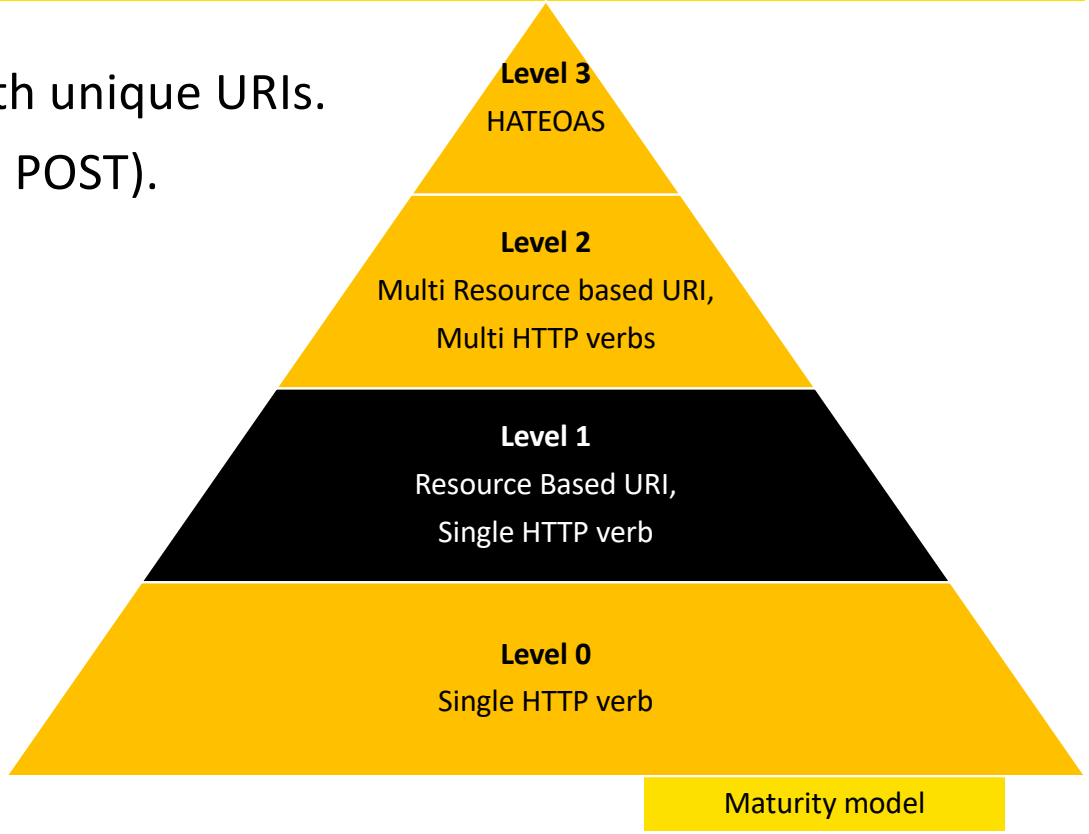


REST Maturity model

Level 1 Resource URI's, only GET or POST

- Introduces the concept of resources, with unique URIs.
- Still relies on single HTTP method (often POST).

POST /users/123



REST Maturity model

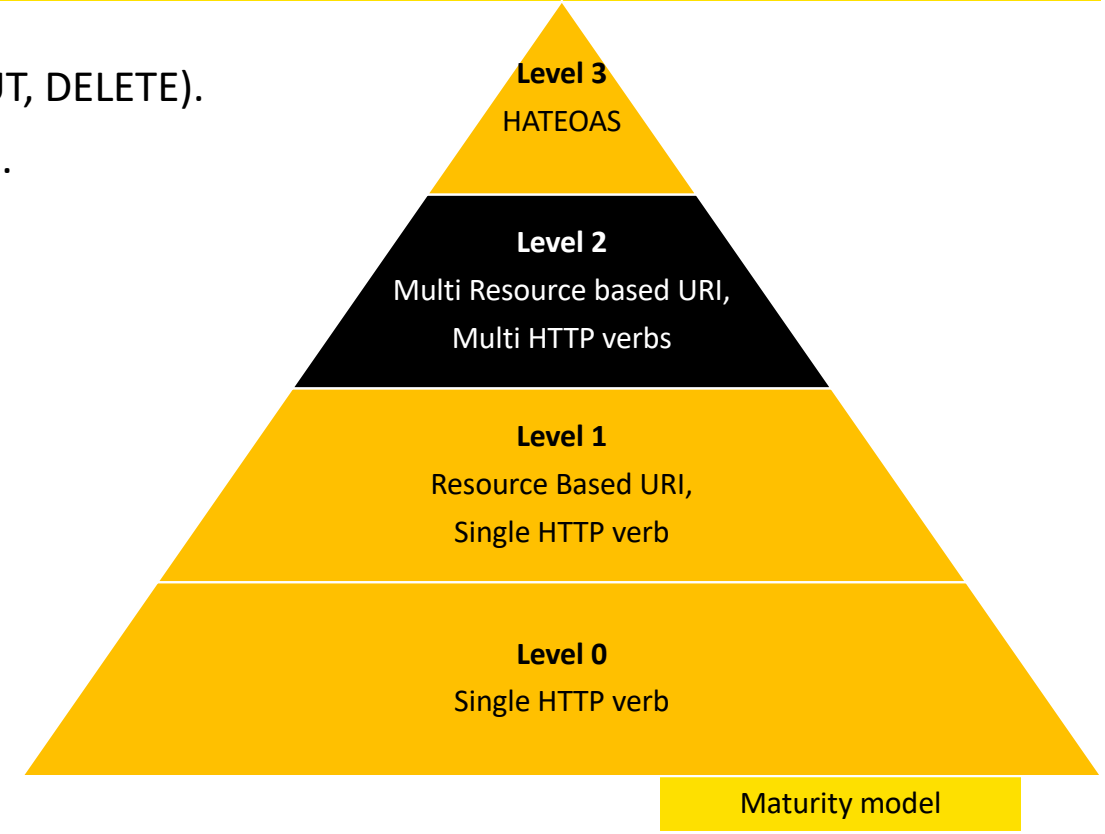
Level 2 Resource URI's and standard HTTP verbs

- Utilizes standard **HTTP methods** (GET, POST, PUT, DELETE).
- Uses proper **status codes** to indicate responses.

GET /users/123 → Retrieves user

PUT /users/123 → Updates user

DELETE /users/123 → Deletes user

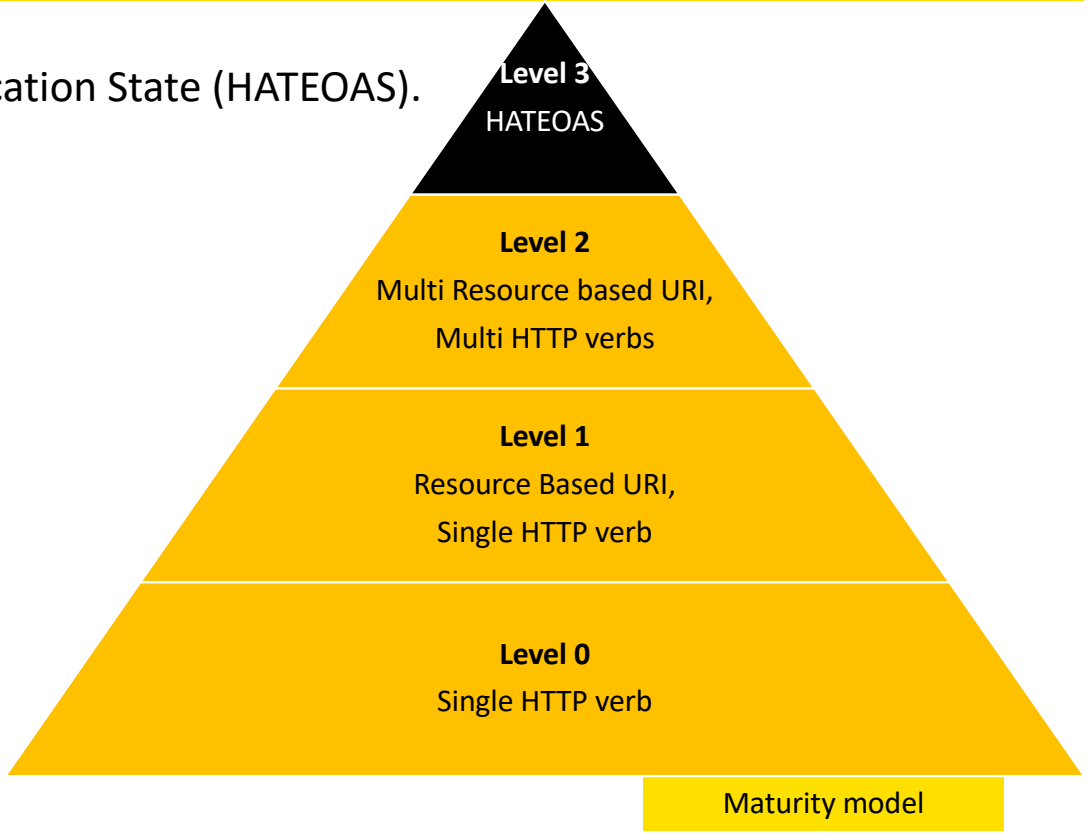


REST Maturity model

Level 3 Hypertext As The Engine Of Application State

- Implements Hypermedia as the Engine of Application State (HATEOAS).
- Responses include links to related resources, guiding clients dynamically.

```
{
 "id": 123,
 "name": "John Doe",
 "links": [
  { "rel": "self", "href": "/users/123" },
  { "rel": "orders", "href": "/users/123/orders" }
 ]
}
```



Real world takeaways

Level 1 APIs (Twitter's early API)

suffer from lack of structure and standardization.

Level 2 APIs (modern Twitter, PayPal)

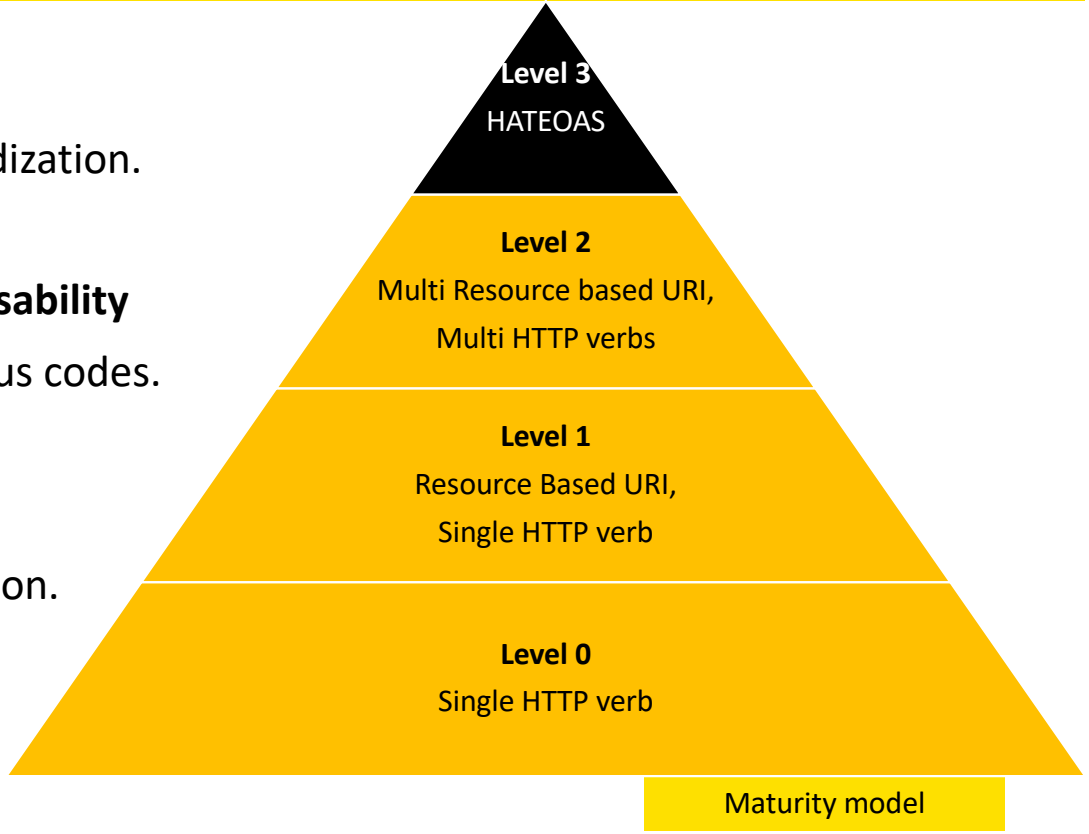
provide **better caching, scalability, and usability**

by implementing HTTP methods and status codes.

Level 3 APIs (GitHub)

enhance **discoverability and flexibility**

with HATEOAS, allowing dynamic navigation.



Summary

Overzie API soorten

Zelfde onderdelen in architectuur

Best practices:

- API design first
- Lifecycle management
- Secure, Stable and Scaleble API’s
- Monitoring
- Testing

Github slides

AND NOW…

Let’s build some API’s!