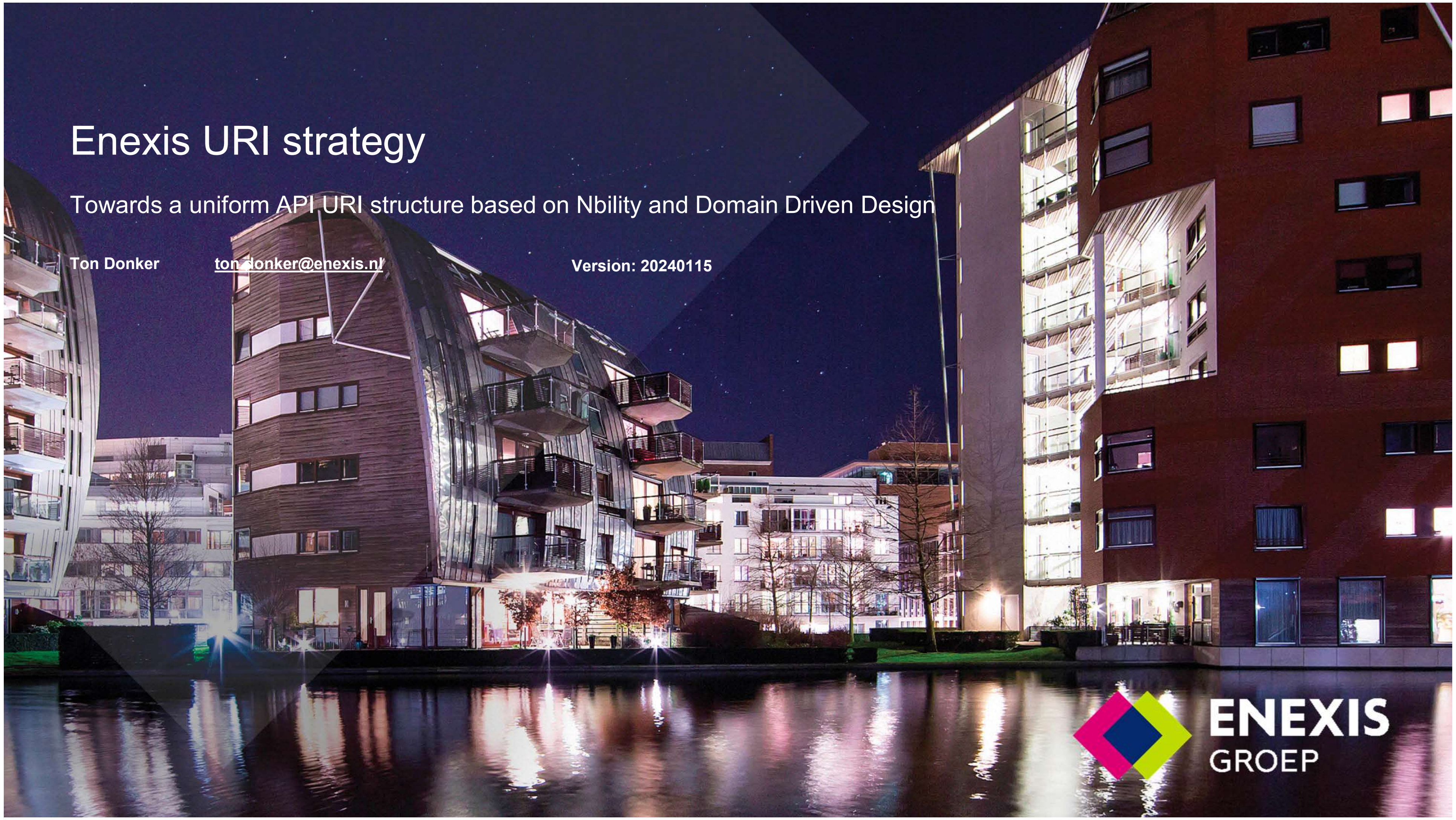


Enexis URI strategy

Towards a uniform API URI structure based on Nbility and Domain Driven Design

Ton Donker **ton.donker@enexis.nl**

Version: 20240115

Change history

Version	Changes	Document
20230618	Initial release	Enexis_URI_strategie_20230618.pptx
20230706	Review System Architect	Enexis_URI_strategie_20230706.pptx
20230714	Review Information Analyst, Engineer	Enexis_URI_strategie_20230714.pptx
20230721	Review Information Analyst, Engineer, Product Owner	Enexis_URI_strategie_20230721.pptx
20231013	Review Enterprise Architect, Integration Architect	Enexis_URI_strategie_20231013.pptx
20231111	Review API developers	Enexis_URI_strategie_20231111.pptx
20240115	Final review and approved	Enexis_URI_strategie_20240115.pptx

Related documents

	Title	Source
	Enexis APIs Standaarden en Richtlijnen	Bijlage 2 Enexis APIs - Standaarden & Richtlijnen - Integratie & Data Autoriteit (IDA) - Confluence (atlassian.net)
	Nbility	NBility > EDSN Corporate
	HAL-Implementatie-ECDM	*To be defined*
	Enexis Integration Terminology	*To be defined*
	Sector community URI Strategy	*To be defined*

Contents

Preface

This document aims to establish guidelines and design patterns for creating, managing, and designing URIs effectively within the Enexis API ecosystem. A well-defined URI strategy ensures that URIs are consistent, meaningful and scalable, facilitating efficient resource identification and retrieval.

The Nbility model and the Domain-Driven Design (DDD) approach lay the foundation for this URI strategy. Where Nbility describes the WHAT: *business capabilities in the problem space*, Domain-Driven Design concentrates on the HOW: *bounded contexts in the solution space*. Both methodologies help to structure and shape the URI endpoint design process, by identifying boundaries and providing patterns for clear and well-designed URIs.

This presented strategy primarily aims at architects, API designers, and developers and anyone involved in the URI design process. The most important part of this document is a set of ten design requirements - *the Ten Enexis URI Commandments* - that must be followed in the URI definition process. Each requirement is theoretically justified and accompanied by clear examples. All design requirements MUST be fulfilled in order to achieve consistency, uniformity and clarity in API URI endpoints.

For substantive questions, you are welcome to contact Integration Business Support team for further assistance.

Reading guide

This document is prepared for architects, designers, and (API) developers and contains the following information for each target audience:

	Architects	Designers	(API) Developers
1. URI Strategy	X	X	X
2. URI Implementation		X	X
3. Nbility Requirements	X	X	
4. Domain-Driven Design	X		
5. Privacy Aspects	X	X	X

Reading guide

Target audience:

This URI strategy is based on the Nbility sector model and the Bounded Context concept of Domain-Driven Design. Knowledge of Nbility and Domain-Driven Design is therefore essential when designing the URI endpoints of an API. Architects and designers are the primary knowledge holders of the Nbility model, while architects also possess expertise in Domain-Driven Design. The URI implementation is relevant for designers and developers. Information about privacy aspects is important for all target audiences.

Document structure:

This document is structured as a top-down hierarchy: first, the *Ten Enexis URI commandments* are presented in a clear overview. Subsequently, the reader can gain insight into the design choices and references to literature to read the underpinning of the design rule in detail.

All the requirements imposed on URIs are outlined in this document. The requirements for the path components are structured as follows: a brief introduction is followed by the presentation of the *design rule*. The rule is then explained and substantiated in the *rationale*. Subsequently, references to *literature* are provided, and a conclusion is offered, confirming and supporting the design rationale.

This document concludes with appendices that provide additional background information.

Introduction

URI strategy

What’s the problem?

A URI design strategy is crucial for creating a consistent and intuitive API endpoint structure, making it easier for consumers and developers to understand and use the API. The problem is how to develop a well-defined URI strategy, to ensure that URIs are consistent, meaningful and predictable, facilitating efficient resource identification and retrieval.

What’s the solution?

Since meaning is determined by context, the Domain-Driven Design methodology can be applied to identify bounded contexts. Bounded contexts represent linguistic boundaries for domain objects and help eliminate ambiguity. Bounded contexts have influence on the design, ownership, scope and purpose of an API. Therefore, the solution is to include the bounded context in the URI endpoint of an API to unambiguously represent domain objects (meaning) and expose clear business capabilities (purpose).

What’s the result?

Consistency, uniformity and clarity in API URI endpoint naming, combined with the concept of bounded context as business context boundary, result in the exposure of meaningfull business capabilities through APIs with clear responsibilities, objectives and optimal usability.

Introduction

Problem example

Enexis has used the following pattern to structure the URI of an API endpoint:

https://apigateway.enexis.nl/{APIname}/{version}/{resource} e.g.:
https://apigateway.enexis.nl/meter-register/v1/meters.

However, the domain object *meter* can have different meanings depending on the relevant context. For instance *meter* in de context of 'technical infrastructure' may have different attributes and semantics compared to the context of 'customer and market,' or in the context of 'asset management.'

Solution example

The context dependency of a domain object is addressed in Domain Driven Design by the concept of bounded context. A classical example is given by Martin Fowler in which bounded contexts for polyseme *meter* are suggested. The solution is to reserve a dedicated component in the URI for the bounded context annnotation, e.g:

https://apigateway.enexis.nl/asset-management/meter-registration/v1/meters

where *asset-management* represents the bounded context and functions as a linguistic boundary for the *meter* domain object. An additional benefit of this approach is that it allows for the identification of the responsible team through context. According to DDD principles, a bounded context is managed by one team only, and bounded contexts also define team boundaries.

Introduction

URI strategy

Route to solution

Since there is an overlap and strong relationship between bounded contexts and business capabilties or subdomains, the Nbility model can be utilized as basis for the identification of bounded contexts. Therefore the route to solution is: identify bounded context based on business capabilities and terminologies provided by the Nbility paradigm and include context annotation in the path component of the URI. In the following slides the new URI pattern is explained in detail.

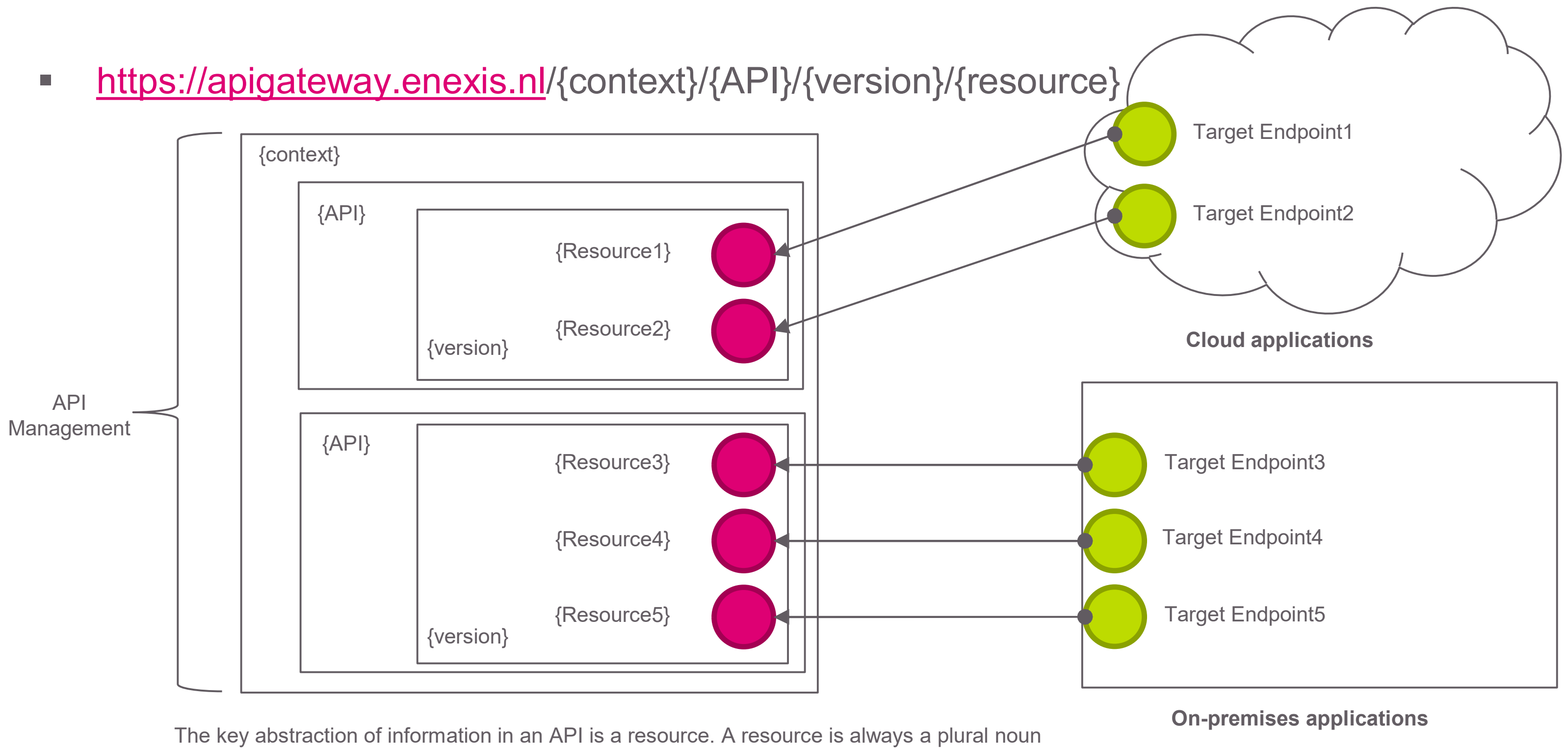
Overview design rules

URI strategy

Rule	Description
ENX-URI-001	Define the path component {context} based on the relevant bounded context. If no bounded context has been determined yet, choose Nbility business capability level 3
ENX-URI-002	Define {context} name based on the noun-verb format, like ‘customer-support, ‘debts-management’, grid-components-development’
ENX-URI-003	Define {API} name based on the format noun-verb, like ‘planned-outages-management, ‘connection-details-retrieval’, ‘product-installations-registration’
ENX-URI-004	Define verb part of {API} name based on the predetermined keywords to express its purpose and objective
ENX-URI-005	Define noun part of {API} name on toplevel (root) resource – align with Nbility business objects
ENX-URI-006	Define the {resource} component of the URI based on the Nbility business objects and their associations
ENX-URI-007	The use of controllers is allowed sparingly, and they appear as the last segment in a URI path, serving as an imperative verb with no child resources
ENX-URI-008	Query parameters must be used for filtering, sorting and hypermedia and prefixed with an underscore in case of meta functionality
ENX-URI-009	The URI must follow general syntax principles and should comply with recommended syntax principles
ENX-URI-010	In the URI, a limited set of regular personal data is allowed, but special personal data or combinations of regular personal data are never permitted

1. URI strategy – endpoint structure

- https://apigateway.enexis.nl/{context}/{API}/{version}/{resource}



The key abstraction of information in an API is a resource. A resource is always a plural noun expressing a functional business object, collection or register. A resource represents a physical endpoint, either operational on-premise or in cloud environments. Context is used to classify groups of APIs on a functional and organizational level and provides namespaces in which every API is uniquely identified and named. This means that an identical API name can reoccur in different contexts.

1. URI strategy – endpoint structure

- https://apigateway.enexis.nl (base URI of all APIs)
  - /{context} - (unique annotation of the context)
    - /{API} - (API name)
      - /{version} - (major version of API)
        - /{resource} - (resource, collection or register)

Examples:

- https://apigateway.enexis.nl/customer-support/customer-tickets-registration/v1/tickets
- https://apigateway.enexis.nl/outage-assessment/planned-outages-management/v2/planned-outages
- https://apigateway.enexis.nl/asset-management/meter-details-retrieval/v3/meters

Base URL {context} {API name} {version}{resource}

The base URI of the test- and acceptance environement are respectively https://apigateway-tst.enexis.nl and https://apigateway-acc.enexis.nl

2. URI design – {context}

- https://apigateway.enexis.nl (base URL of all APIs)
  - /{context} – (unique annotation of the context)

Key message is that APIs are based on bounded contexts. By identifying bounded contexts, the scope, purpose and team ownership of an API representing a business capability with unambiguous domain objects (aka resources) becomes clear. There are two aspects that can be used as guidance to identify bounded contexts — terminology and business capabilities[1].

Defining clear boundaries is an important factor for APIs, as it helps to accelerate the API design and development process by scoping APIs to a specific set of responsibilities[2].

The Nbility model[3] with categorized business capabilities level 3 can be used as a starting point for the identification process of bounded contexts. Identifying the bounded context precedes the design of the API[3]. To provide contexts with a uniform naming, the format *noun-verb* is required for naming a bounded context.

1. [Millett 2015a] p128
2. [Higginbotham 2022] p72
3. https://www.edsn.nl/nbility-model/

2. {context} – design rules

URI strategy

ENX-URI-001

Define the path component {context} based on the relevant bounded context. If no bounded context has been determined yet, choose Nbility business capability level 3

ENX-URI-002

Define {context} name based on the noun-verb format, like 'customer-support, 'debts-management', 'grid-components-developement'

Example:

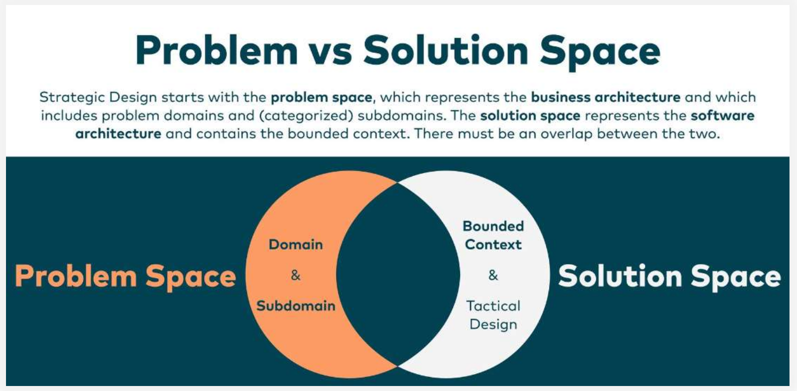
- https://apigateway.enexis.nl/**outage-assessment**/planned-outages-management/v2/planned-outages

{context} in noun-verb format and based on Nbility (reference: C.2.3.2)

2. {context} – design rationale

According to the Domain-Driven Design literature, business capabilities can be associated with (sub)domains as part of the problem space describing the WHAT, whereas bounded contexts belong to the solution space, describing the HOW[2]. Conceptually, they are different things, but there is a strong relationship and even overlap.

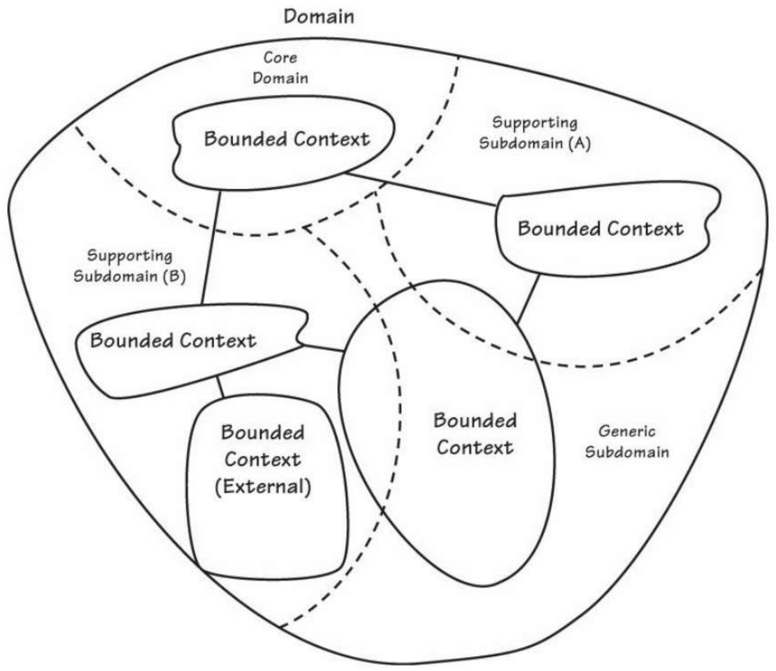
It is a desirable goal to align subdomains one-to-one with bounded contexts[3].


Problem vs Solution Space
Strategic Design starts with the problem space, which represents the business architecture and which includes problem domains and (categorized) subdomains. The solution space represents the software architecture and contains the bounded context. There must be an overlap between the two.
Problem Space
Domain & Subdomain
Bounded Context & Tactical Design
Solution Space

While (micro)services are aligned with business capabilities[4], bounded contexts are aligned with (sub)domains. To draw this comparison further: (micro)services can be seen as the implementation or realization of business capabilities. They are typically designed, developed, and deployed to enable and support specific business processes or functions. Same holds for bounded contexts as the implementation of (sub)domains.

1. Figure from: https://speakerdeck.com/mploed/riding-the-elevator-domain-driven-design-in-the-penthouse?slide=47
2. https://medium.com/nick-tune-tech-strategy-blog/domains-subdomain-problem-solution-space-in-ddd-clearly-defined-e0b49c7b586c
   [Khononov 2022] p298: … a subdomain bounds a business capability…;
   [Vaughn 2013] p76
   https://towardsdatascience.com/data-domains-where-do-i-start-a6d52fef95d1
3. [Vaughn 2013] p77
4. https://martinfowler.com/articles/microservices.html: …services are built around business capabilities

2. {context} – design rationale



As depicted in the figure on the left, it is possible for bounded contexts to have an n;n relationship with subdomains[2]. A bounded context can implement 1..* subdomains (or parts of subdomains) and a subdomain can be implemented by 1..* bounded contexts (or parts of bounded contexts).

Ideally, there is a 1:1 relationship between bounded contexts and subdomains. However, in practice, this relationship can vary depending on the specific domain's structure and complexity. The key message of this design rationale is that the Nbility model with categorized business capabilities level 3 can be used as a starting point[3] for the identification process of bounded contexts. This can lead to a many-to-many alignment between bounded contexts and business capabilities or subdomains.

1. Figure from [Vernon 2013] p70
2. https://contextmapper.org/docs/language-model/
3. https://www.edsn.nl/nbility-model/

2. {context} – design rationale

There is no strict naming convention for bounded contexts in Domain-Driven Design, as the naming of bounded contexts is subjective and depends on the specific domain and the understanding of the development team. However, it is generally recommended to use clear and descriptive names that reflect the purpose and responsibilities of the bounded context[1].

The TOGAF standard[2] provides a naming convention for a business capability in a *noun-verb*[3] format, e.g.: *project-management* or *strategy-planning*. The first noun part of the naming convention is a business object, the second part is the activity associated with the object. Since bounded contexts are related to business capabilities/subdomains, the TOGAF naming convention can be used as guidance for naming the context in the path component of the API URI endpoint[4].

This design rationale is further motivated and justified in {context} - design literature

1. Appendix D overviews an extensive list of bounded context naming examples as used in the DDD literature
2. https://pubs.opengroup.org/togaf-standard/business-architecture/business-capabilities.html#_Toc95135880
3. While 'management' is a noun and not a verb, TOGAF uses the term 'noun-verb' to identify this compound combination of nouns See also: https://enterprise-architecture.org/university/business-capability-modelling/
4. Bounded context naming examples that match this pattern are (source: https://stefan.kapferer.ch/2020/09/14/domain-driven-service-design/):
   - Customer Management
   - Debt Collection
   - Risk Management
   - Account Management [Millett] p270

3. URI design – {API}

- https://apigateway.enexis.nl (base URL of all APIs)
  - /{context} – (unique annotation of the context)
    - /{API} - (API name)

The API name conveys the purpose and objectives of the API. To maintain a consistent naming convention for APIs, it is required to use the *noun-verb* format, with predetermined values for the verb part:

	Keyword	Description
	registration	… if the main purpose of the API is to register information (write)
	retrieval	… if the main purpose of the API is to retrieve information (read)
	management	… if the main purpose of the API is both read/write and it impacts the total lifecycle of a resource
	computation	… if the main purpose of the API is to perform calculations or determine numerical values
	notification	… if the main purpose of the API is to handle events and notifications
	transfer	… if the main purpose of the API is to exchange and transfer data

The noun part in the API name should be derived from the Nbility business objects, aligning with the use cases and problems the API aims to address.

3. {API} – design rules

ENX-URI-003

Define {API} name based on the format noun-verb, like ‘planned-outages-management, ‘connection-details-retrieval’, ‘product-installations-registration’

ENX-URI-004

Define verb part of {API} name based on the predetermined keywords to express its purpose and objective

ENX-URI-005

Define noun part of {API} name on toplevel (root) resource – align with Nbility business objects

Example:

- https://apigateway.enexis.nl/outage-assessment/**planned-outages-management**/v2/**planned-outages**

{API name} is based on predetermined keyword and root resource

3. {API} – design rationale

There are best practices for naming APIs:

- Choose names that accurately reflect the purpose and functionality of the API. Names should be intuitive, concise, and self-explanatory;
- APIs typically represent resources or actions. Use noun phrases to describe the resources and actions the API provides. For example: *user-profile* or *order-creation*[1];
- The name of the API is typically indistinguishable from the name of the resource that can be manipulated by the API[2];
- An API name / endpoint is the location where a service can be accessed[3];
- The URI of a REST API identifies the service within the system component[4];

REST advocates a resource-oriented approach: URIs should reflect the resources themselves rather than the actions performed on them. This applies to both the resource path (after the version number) and the Base URL, which is the URL prefix for all APIs[5].

.

1. In Higginbotham [2021] p112 APIs are named: ‘Shopping API’, ‘Order Creation API’ and ‘Payment Processing API’
2. Biehl [2015] p52
3. Zimmermann [2021]
4. Sloos [2020]
5. https://swagger.io/docs/specification/api-host-and-base-path/

3. {API} – design rationale

Including the name of the API in the Base URL of the URI is common practice in both the Dutch[1] and energy sector community URI strategy.

The base URL is composed of the {context} and {API} components. In {API} - design literature it is explained that the {API} component represents an interface bounded context, as a separate bounded context nested hierarchically within the parent {context}.

By defining the API name in the noun-verb format as described in the previous slides, the URI guidelines and design rules of the Dutch Energy sector are followed, aiming to establish a uniform naming strategy in the energy sector.

Note that the Dutch government (DSO) allows verbs in the API names[2]. In the Dutch energy sector community, we have decided not to adopt this as this is not in line with common best practices, conflicts with the uniform interface and leads to poor HTTP resource design.

1. Examples: https://developer.overheid.nl/apis and Appendix F
2. Example: The API ‘Ruimtelijke plannen opvragen’ has the following URI endpoint: https://ruimte.omgevingswet.overheid.nl/ruimtelijke-plannen/api/opvragen/v4

3. {API} – design rationale

Using a predetermined keyword in the API name adds semantics and clarity to the purpose and goal of the API and carves out the architectural role API endpoints (emphasis on data or activity)[1]. The API name expressed as a combination verb-noun gives answer to the consumer question *'…what can I do with this API ….?'*. When choosing an API name is too hard, it can be a sign of design smell[2].

Best practices for naming command messages and events are also based on noun verb combinations. For command messages, examples include: 'PlaceOrder', 'CheckOffer' while for events examples are: 'OrderProcessed', 'OfferUpdated'[3].

This design rationale is further motivated and justified in {API} - design literature

Examples:

▪ https://apigateway.enexis.nl/installation-management/facilities-**management**/v1/facilities	Nbility C.6.2.2
▪ https://apigateway.enexis.nl/measurement-collection/measurements-**retrieval**/v2/measurements	Nbility C.4.1.2
▪ https://apigateway.enexis.nl/customer-support/contact-moments-**registration**/v2/customer-interactions	Nbility C.1.1.1

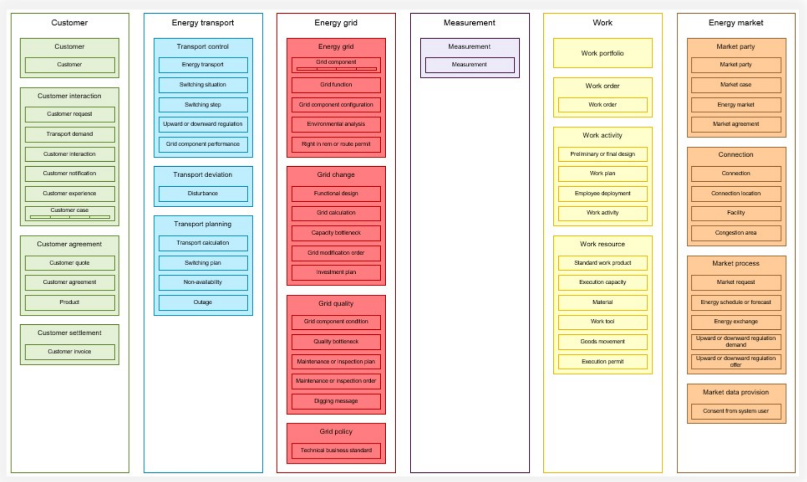
1. [Zimmermann 2022] p162
2. https://apihandyman.io/pink-fluffy-unicorn-api-wtf-or-3-reasons-why-choosing-a-not-meaningful-API-name-can-be-a-problem/
3. [Zimmermann 2022] p194

4. URI design – {resource}

URI strategy

- https://apigateway.enexis.nl
  - /{context} – (unique annotation of the context)
    - /{API} - (API name)
      - /{version} - (major version of API)
        - /{resource} - (resource, collection or register)

Resources are based on and aligned with the Nbility business objects. The business objects can be related with each other, either associatively or hiërarchically. To design cohesive APIs, related business objects can be combined within the same API as long it does not conflict with the scope boundary of the bounded context ownership or conflicting Nbility information domains.



Nbility information domains

4. {resource} – design rules

URI strategy

ENX-URI-006

Define the {resource} component of the URI based on the Nbility business objects and their associations

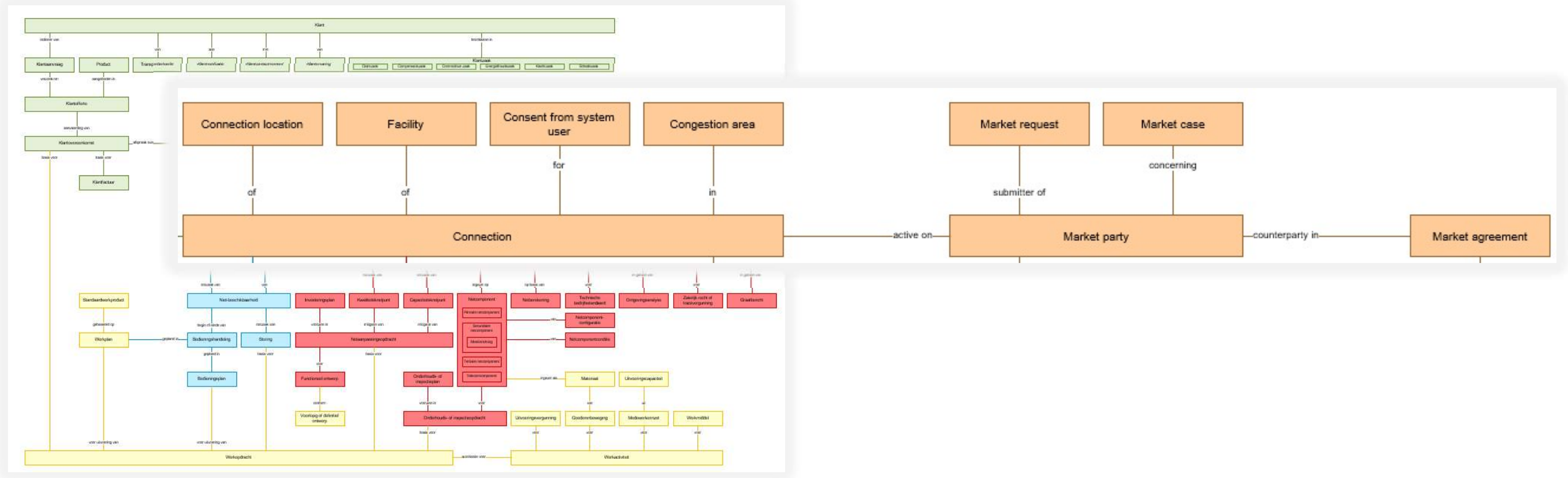
Examples:

- https://apigateway.enexis.nl/market-contract-management/contract-management/v1/**market-agreements**
- https://apigateway.enexis.nl/market-contract-management/contract-management/v1/**accounting-points**
- https://apigateway.enexis.nl/customer-support/contact-moments-registration/v2/**customer-interactions**
- https://apigateway.enexis.nl/customer-management/customers-registration/v2/**customers**

{resource} based on Nbility business objects

4. {resource} – design rationale

To give an example of associated business objects: the Nbility model describes a strong relationship between the objects ‘Market party’, ‘Market agreement’ and ‘Connection’. A ‘Market party’ *is active on* a ‘Connection’ and *a counterparty in* a ‘Market agreement, as visualized’ in below picture:



4. {resource} – design rationale

A *contracts-management* API exposes the associated resources as endpoints (Nbility reference: C.6.1.3):

- https://apigateway.enexis.nl/market-contract-management/contracts-management/v1/**market-agreements**
- https://apigateway.enexis.nl/market-contract-management/contracts-management/v1/**accounting-points**

where:

- *market-agreements*: represents the collection of agreements recorded on a given connection
- *accounting-points*: the collection of allocation points for which agreement information can be retrieved. The *accounting-point* is the CIM equivalent of Nbility object *connections*

This is possible because of the strong relationship between the two business objects as outlined by the Nbility model.

To design cohesive APIs and to be clear about the scope and responsibility only related objects exposed as resources should be represented in the API. See the Enexis API design guidelines for further information.

This design rationale is further motivated and justified in {resource} - design literature

5. Processing resource

URI strategy

Processing resources represent activity-oriented API designs and endpoints[1]. A common situation in API design is the need to go beyond the typical CRUD interaction model. For example, some APIs may need to support actions such as submit, approve, and decline. With our somewhat limited action verbs in HTTP, how can we add this to our API?[2]

- A verb is allowed as an action (controller) on a resource in certain cases
- A controller resource or processing resource is an executable function on a resource
- Only allowed if the use of standard HTTP methods results in a situation that compromises the flexibility, adoption, and/or usability of the API
- A controller resource name is a verb instead of a noun. Verb in imperative mode
- Controller names typically appear as the last segment in a URI path, with no child resources to follow them in the hierarchy
- The HTTP method on a controller SHOULD be:
  - **POST** for actions with side effects (state change) or actions without side effects but requiring a request body
  - **GET** for idempotent actions without side effects

1. https://api-patterns.org/patterns/responsibility/endpointRoles/ProcessingResource
2. https://tyk.io/blog/rest-never-crud/

5. Processing resource

- In certain cases, it can be highly desirable to perform a complex search based on a large dataset, wildcards, or arrays. The use of query parameters is not sufficient in this case and can lead to unnecessary complexity. When dealing with 5 (~) query parameters or more, the application of a complex search with a POST request is required
- The OpenAPI Specification does not allow a JSON payload in combination with a GET request
- For an extensive search (complex search), a POST request is used. There is market consensus for its usage
- Explicit use of a 'search' resource can be made to avoid confusion with creating a new resource

Examples:

- POST https://apigateway.enexis.nl/capacity-planning/investment-management/v1/investment-plans/123/**approve**
- POST https://apigateway.enexis.nl/market-connections-management/connection-details-retrieval/v1/connections/**search**

ENX-URI-007

The use of controllers is allowed sparingly, and they appear as the last segment in a URI path serving as an imperative verb with no child resources

More information:
- https://restfulapi.net/resource-naming/
- https://cloud.google.com/apis/design/custom_methods
- https://gitdocumentatie.logius.nl/publicatie/api/adr/#operations

6. Query parameters

- Query parameters are only allowed for the following applications:
  - Filtering;
  - Hypermedia controls;
  - Pagination;
  - Sorting;
- Query parameters should not be used for the identification of unique objects (resources)
- The names of query parameters are always case insensitive for usability purposes. However, if resource names are used as the 'value' of a query parameter, those resource names are case sensitive
- To distinguish between ‘real’ query parameters and the more ‘steering’ or ‘meta’ parameters like expand, fields, limit and page, a prefix should be used to prevent misinterpretation of the function of the query parameters: _expand, _field, _limit, _sort and _page

Examples:

- https://apigateway.enexis.nl/customer-settlement/sales-orders-management/v1/sales-orders?**type=unblocked**
- https://apigateway.enexis.nl/market-connections-management/connection-details-retrieval/v1/connections?**_limit=100&_page=1**

ENX-URI-008

Query parameters must be used for filtering, sorting and hypermedia and prefixed with an underscore in case of meta functionality

7. Syntax principles (general)

- The URI MUST
  - comply with internet standard RFC3986;
  - not support file extensions;
  - not end with trailing slashes;
  - use lowercase notation. Exceptions are query parameters that follow the lowerCamelCase notation cf JSON payload notation;
  - not contain empty path segments;
  - use hyphens in case of composite terms. Also known as ‘kebab’ or ‘spinal case’;
  - use underscores for meta functional query parameters;
  - not use spaces: use hyphens instead;
  - Not use diacritic signs for path parameters. Characters like é should be converted to e (etc.)
  - not use non-standard abbreviations;
  - not use unknown characters: (!, @, #, $, %, ^, &, *, etc.) should be replaced with a hyphen;
  - identify resources and subresources via path parameters;
  - predetermined query parameters for searching, sorting, filtering and hypermedia;
  - not contain the word ‘api’ in API name or resource name;
  - not contain the name of an organization, project or application in API name of resource name;

7. Syntax principles (recommended)

- The URI SHOULD
  - be durable and long-term stable: Cool URIs don't change;
  - restrict the nesting of subresources to a maximum of three levels. Deep nesting complicates readability and best practice is to keep the URI length under a limit of 2048 characters;
  - be 'intuïtive' and 'human readable';
  - pluralize resource names;
  - handle controller resources and processing resources as exceptions;
  - use path parameters instead of query parameters to identify a unique resource;

ENX-URI-009

The URI must follow general syntax principles and should comply with recommended syntax principles

Sources:

https://nordicapis.com/10-best-practices-for-naming-api-endpoints/
https://opensource.zalando.com/restful-api-guidelines/index.html#urls
[Higginbotham 2021]
[Sloos 2020]

8. Security and privacy

An API makes resources addressable within a URI path using resource keys. The resource key serves as a unique identification of a single resource within a (potentially large) resource collection.

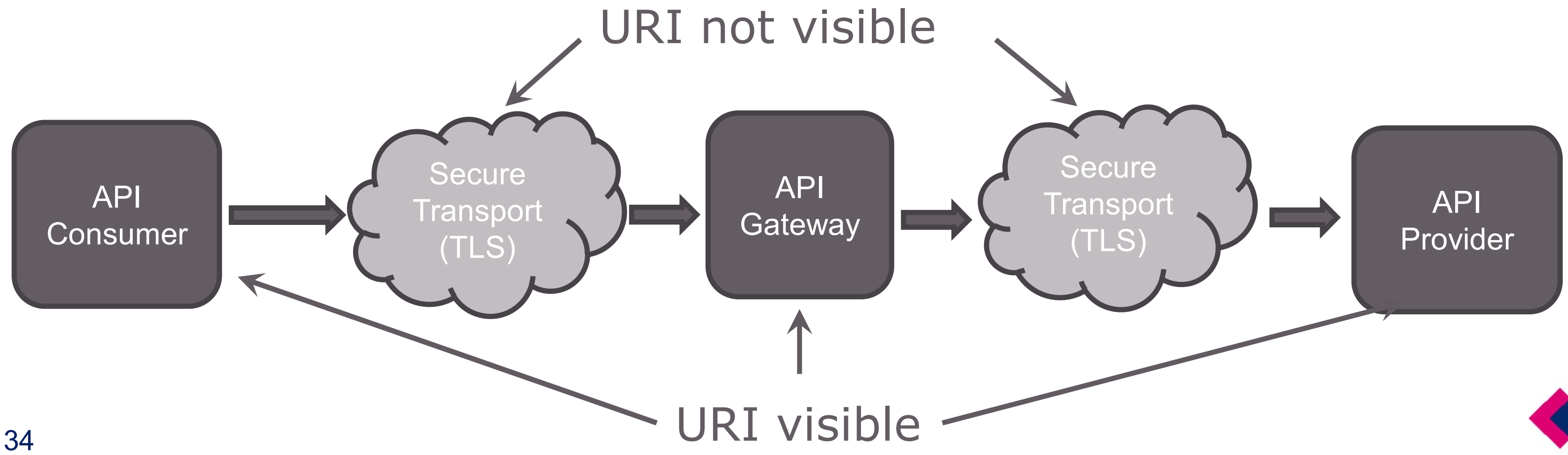
From a privacy perspective, careful handling of resource keys is necessary if they can be directly linked to individuals see Privacy Statement:

Description	
When designing an API, it is important to ensure that the use of one or more resource keys in a URI of that API never leads to the leakage of sensitive information[1]	
Rationale	**Implication**
There are examples where the key (or combination of keys) of one or more resources could potentially lead to the leakage of sensitive information (e.g., a social security number used as a key for a person). This is particularly relevant since URIs on the endpoints of a secure connection can be stored in server logs	• The structure of resources as provided by an API must comply with the Enexis URI conventions • The Privacy Statement describe how certain combinations of resource keys are allowed or not allowed, and how replacement keys and/or encryption of resource keys can be used (or should be used) to avoid data leaks

1. https://danielmiessler.com/p/sensitive-information-sent-in-the-url-over-https/

8. Security and privacy – URI visibility

- The mandatory use of TLS at all times ensures that URIs in the *consumer-gateway-provider* chain are **not visible during transit** as there is an encrypted end-to-end network connection. However, the URLs are still **visible** at the endpoints.
- The handling of URIs at the endpoints for internal APIs is governed by the current Enexis policy regarding application development (reference architecture, privacy, and security policy). For external parties, the proper handling of URIs should be documented in the relevant legal agreement.



9. Privacy Statement

The following slides contain a statement with an addition, issued on May 10, 2021, by the Privacy Officer of Governance and Legal Affairs at Enexis, in response to the question:

Under what conditions, based on the security aspects presented in the Privacy and Security slides, can personal data be exchanged via the URI of APIs?

9. Privacy statement

Assuming that:

- when using regular personal data in the URI, the entire message (body and URI) is encrypted during transmission;
- these data can only be viewed by the API Consumer, API Gateway, and API Provider;
- the use of personal data in the URL is only permitted if it is necessary for both the sending and receiving parties to have access to this personal data for the performance of their tasks;
- no sensitive or special categories of personal data can be inferred from the URI; (category 2);

There is an acceptable risk for the Business in using a certain number of regular personal data in the URIs of APIs, particularly to make the construction of APIs manageable and controllable. This includes the use of identifying data such as:

- EAN-code;
- Customer number;
- Meter number;
- ZIP code housenumber;
- BAG (Basisregistratie Adressen en Gebouwen)

9. Privacy Statement – addition

Additionally, there are certain sensitive or special categories of personal data that, if disclosed, could cause harm to the customer. These data **should never appear in the URI**. If it is necessary to exchange such data, they will be transmitted exclusively in the body of the request, ensuring that they cannot be inferred from the logging either. The list of special categories of personal data is exhaustive, namely:

- Race or ethnic origin;
- Political opinions;
- Religious or ideological beliefs;
- Membership of a trade union;
- Genetic or biometric data
- Health data*;
- Data about sexual orientation;
- BSN-number;
- Criminal data or criminal records trafrechtelijke gegevens;

* The underscored data categories definitely occur within Enexis

9. Privacy Statement – addition

URI strategy

When it comes to sensitive data, examples (non-exhaustive) include

- Fraud data;
- Financial data (both bank account number/credit card and payment data);
- Consumption data;
- Disconnection data;

All of these data categories definitely occur within Enexis.

It is not possible for Privacy to provide an exhaustive list of personal data and their respective categories. Whether something qualifies as personal data and its sensitivity strongly depends on the context in which the data is used. In this example, I assumed customer/connection data. For employee data, the list of sensitive data may have a different composition. **Additionally**, there is an **important exception**. By combining certain data, new sensitive information can be derived. The **combination** of an EAN code with a meter number or a postcode with a house number and meter number **should never appear in the URI**. The meter number is widely used as an identifier in many processes to verify that the customer has access to the connection. By using a combination of connection data and meter number, it would facilitate fraud.

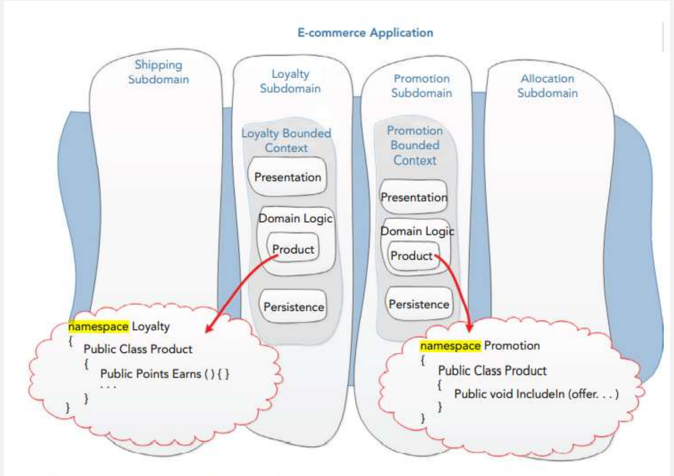
ENX-URI-010

In the URI, a limited set of regular personal data is allowed, but special personal data or combinations of regular personal data are never permitted

{context} – design literature

It is interesting to explore in the DDD literature where examples are provided of endpoints[1] with a bounded context as part of the namespace or URI path. Below are some examples and quotes:

- [Evans 2003] p419: Of course, because **each BOUNDED CONTEXT is its own name space**, one structure could be used to organize the model within one CONTEXT, while another was used in a neighboring CONTEXT, and still another organized the CONTEXT MAP[2].
- [Millett 2015] p86: A closer inspection, as shown in Figure 6-11, shows the product concept existing in two models but defined by the context that it is within.
- [Millett 2015] p274: To build the API that produces the Account Management entry point, you need to start by adding a new ASP.NET application to the solution called AccountManagement.EntryPoint. This follows the convention of naming API projects based on the format {bounded context}.{Resource}.



```
namespace Loyalty
{
  Public Class Product
  {
    Public Points Earns ( ) { }
    . . .
  }
}
```

```
namespace Promotion
{
  Public Class Product
  {
    Public void IncludeIn (offer. . . )
  }
}
```

FIGURE 6-11: The Product class in different contexts.

1. An API endpoint is the provider-side end of a communication channel and a specification of where the API endpoints are located so that APIs can be accessed by API clients (also called API consumers). Each endpoint thus must have a unique address such as a Uniform Resource Locator (URL), as commonly used on the World-Wide Web (WWW), in HTTP-based SOAP, or in RESTful HTTP. Each API endpoint belongs to an API; one API can have different endpoints: http://eprints.cs.univie.ac.at/7194/1/patterns-on-designing-api-endpoint-operations-proceeding-version5.pdf
2. In this case, 'namespace' refers to a C# namespace or a package in Java

{context} – design literature

- [Millett 2015] p273: When clients want to interact with a REST API, they start by requesting the entry point resource. From then on, they mostly just follow links in the hypermedia that is returned. Choosing where to locate your entry point(s) has a number of considerations that you should take into account. For instance, the design in Figure 13-10 chooses to have a **single entry** point per-bounded context. But you could have a single entry point for the entire system or go more fine-grained and have an entry point per top-level resource. Ultimately, you have to decide on a per-project basis how much of the system you want to expose via entry point resources[1].

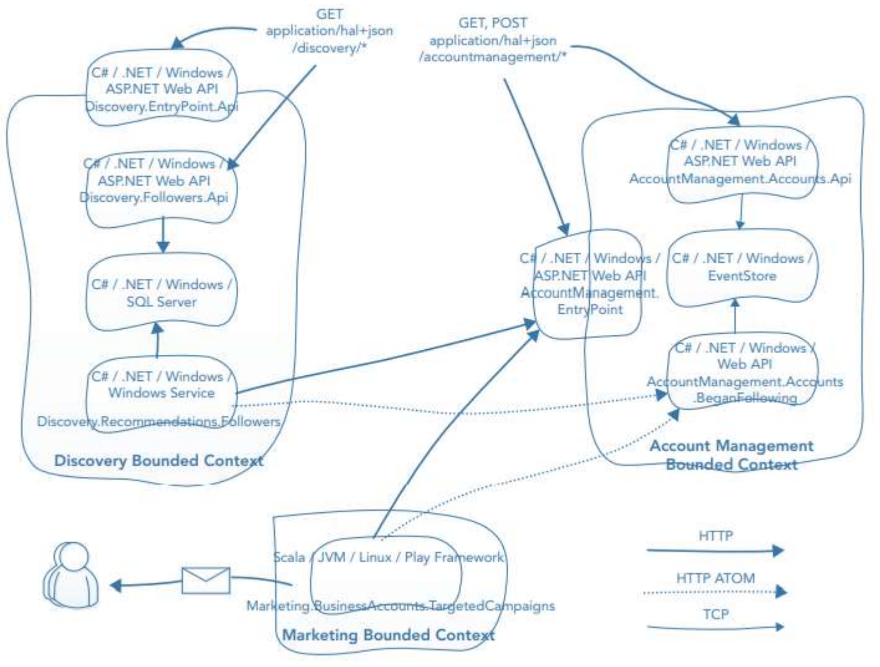

GET application/hal+json /discovery/*
GET, POST application/hal+json /accountmanagement/*
C# / .NET / Windows / ASP.NET Web API Discovery.EntryPoint.Api
C# / .NET / Windows / ASP.NET Web API Discovery.Followers.Api
C# / .NET / Windows / SQL Server
C# / .NET / Windows / Windows Service Discovery.Recommendations.Followers
Discovery Bounded Context
C# / .NET / Windows / ASP.NET Web API AccountManagement.Accounts.Api
C# / .NET / Windows / ASP.NET Web API AccountManagement.EntryPoint
C# / .NET / Windows / EventStore
C# / .NET / Windows / Web API AccountManagement.Accounts.BeganFollowing
Account Management Bounded Context
Scala / JVM / Linux / Play Framework
Marketing.BusinessAccounts.TargetedCampaigns
Marketing Bounded Context
HTTP
HTTP ATOM
TCP

FIGURE 13-10: Containers diagram of Discovery, Account Management, and Marketing bounded contexts.

{context} – design literature

- [Millett 2015] p274: Designing an entry point involves identifying the initial resources and transitions that should be available to consumers of the API. **The Account Management bounded context's entry point will be the list of top-level resources**. In this example, that is just the Accounts resource. From the Accounts resource, hypermedia will link to individual accounts, and from individual accounts to details of those accounts, exposed as child resources, such as its followers[1].

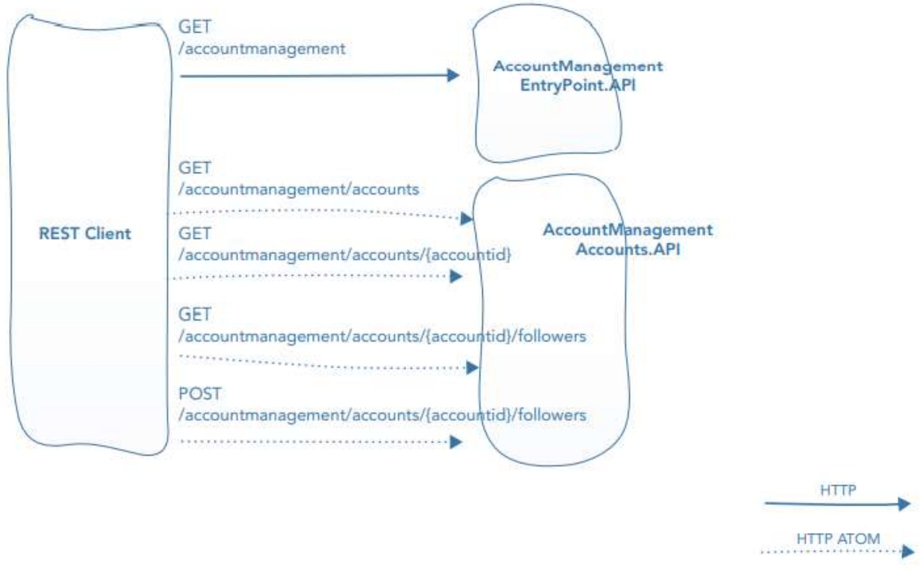

REST Client
GET /accountmanagement
AccountManagement EntryPoint.API
GET /accountmanagement/accounts
GET /accountmanagement/accounts/{accountid}
GET /accountmanagement/accounts/{accountid}/followers
POST /accountmanagement/accounts/{accountid}/followers
AccountManagement Accounts.API
HTTP
HTTP ATOM

FIGURE 13-11: Flow of HTTP requests for the Add Follower use case.

1. In this example, the bounded context is annotated as top-level resource in the API endpoint. This might be too course-grained, since a bounded context is not a resource by itself, but more a linguistic boundary of a resource. The overall bounded context as top-level resource instead of the aggregate root might result in complex and large endpoints. [Zimmermann 2021a]: 'Instead of an aggregate, Bounded Context as API Endpoints is a practice that offers bounded contexts as composite units offered as API endpoints. But this often is much too coarse-grained leading to too large and complex API endpoints'. Source: https://eprints.cs.univie.ac.at/6948/1/europlop21-s16-camera-ready2.pdf

{context} – design literature

Except in the canonical DDD literature, there are internet resources in which the inclusion of the bounded context in the URI path of the API endpoint is discussed:

- If we also build our microservices to match these bounded contexts, we will inevitably end up with duplicated aggregate roots, which is bad REST juju as you can't easily shadow RESTful endpoints without making things confusing or making your API Gateway management software angry. So, how do we break up microservices with similarly named aggregate roots? The answer is fairly simple, you use a similar method to how you may implement versioning a RESTful API e.g.:

  sales.mybusiness.com/v1/Customer
  customer-care.mybusiness.com/v1/Customer
  or
  mybusiness.com/**sales**/v1/Customer
  mybusiness.com/**customer-care**/v1/Customer

- Modern API Management Gateway software should be fine with adopting this model, but as a business, you'll have to agree on how you manage these "Multiple Canonical Models", which is another topic entirely[1].

1. https://www.linkedin.com/pulse/architectural-mistakes-i-have-made-microservices-apis-jeff-watkins this is not only theoretically well defined, but also workable and executable in a 'disciplined environment': https://www.linkedin.com/feed/update/urn:li:article:7478996722983439236/?commentUrn=urn%3Ali%3Acomment%3A%28article%3A7478996722983439236%2C6928992793684025344%29

{context} – design literature

More internet resources in which bounded contexts as part of the URI is being discussed:

- https://softwareengineering.stackexchange.com/questions/279833/partitioning-rest-api-resources-into-areas-based-on-business-domains
- https://microservice-api-patterns.org/cheatsheet (identify endpoint candidates)
- https://itlibrium.com/en/blog/how-to-use-csharp-projects-and-namespaces-in-a-ddd-project (company -> bounded context -> modules)
- https://softwareengineering.stackexchange.com/questions/317423/ubiquitous-language-and-maturity-level-in-rest-api (example http://yourdomain/<BoundedContextName>/<ApplicationServiceName>)
- https://eprints.cs.univie.ac.at/6780/2/api_ddd_grey_literature_based_gt.pdf (table 1)
- https://www.slideshare.net/launchany/designing-apis-and-microservices-using-domaindriven-design (slide 31)
- https://alok-mishra.com/2021/06/30/domain-driven-design-ddd-core-concepts-explained/ (Namespace /api/{my-domain}/{context})

➢ ***Conclusion: bounded context as part of the path component in the URI is a recognizable concept in both DDD literature and in DDD-related discussions on the internet***

{resource} – design literature

URI strategy

Aggregates, along with Entities, Value Objects, Services, and Repositories, are the tactical building blocks used to design a domain model of a bounded context. The applications and properties of these building blocks are extensively described in the DDD literature. Vaughn Vernon devotes almost 200 pages to the description of Entities, Value Objects, and Aggregates. A recent explanation of Aggregates can be found in Thomas Ploch's presentation (DDD Europe, Amstelveen 2022)[1].

In the DDD literature and on the internet, there is consensus regarding the correlation between REST resources and bounded context's aggregates. We use the term *correlation* here because aggregates are part of the implementation domain model of the bounded context, formulated in the *Ubiquitous Language*, while REST resources are part of the integration model, formulated in the *Published Language*. Aggregates belong to the business domain, and REST resources belong to the integration domain, and by definition are decoupled to ensure flexibility in domain models and stability in REST interfaces. Here are some quotes about aggregates and resources:

- As an Aggregate abstracts the implementation details of a number of related Entities and other DDD model elements, it naturally serves as an Identified Interface Element. Practitioners agree that Aggregate Roots are a good starting point for API Resources[2];
- Another thing to keep in mind is that Aggregates usually either succeed or fail as a whole. They often form the boundaries of transactionality. That is why Aggregates in particular are good first candidates for microservices[3];
- Aggregations are naturally assigned to resources in REST[4];

1. https://slides.com/tploch/what-is-a-ddd-aggregate
2. https://zenodo.org/record/4493865/files/api_ddd_grey_literature_based_gt.pdf
3. https://kgb1001001.github.io/cloudadoptionpatterns/Cloud-Native-Architecture/Identify-Entities-And-Aggregates/
4. https://docs.microsoft.com/nl-nl/azure/architecture/microservices/design/api-design

{resource} – design literature

URI strategy

- RESTful HTTP provides methods, primarily GET, PUT, POST, and DELETE. Even though these may seem to support only CRUD operations, using a little imagination allows us to actually categorize operations with explicit intent within one of the four method categories. For example, GET can be used to categorize various kinds of query operations, and PUT can be used to encapsulate a command operation that executes on an Aggregate (10)[1];
- Yet each resource is built from, for example, one or more Aggregates belonging to the Core Domain[2];
- Aggregates map nicely to REST resources[3];
- Aggregates and Entities become Resource APIs[4];
- Aggregates supply API resources (or responsibilities) of service endpoints[5];
- Do not mechanically turn all application domain (micro-)layer artifacts such as Domain-Driven Design (DDD) pattern instances or facades into (micro-)services, but follow a recognized or homegrown identification method. For instance, the DDD Bounded Context is seen to form an upper boundary for microservice size, while Aggregates serve as lower boundary[6];
- Aggregates supply API resources (or responsibilities) of service endpoints[6];
- According to our practitioner sources, Entities as API Endpoints can lead to problems related to API complexity, data consistency, chatty APIs, performance and scalability isses, API understandability, API maintainability, and API evolvability[7];
- So a ‘service api’, can be thought of as a set of operations, each being a command to an aggregate. See the /reserve-seats example[8];

1. [Vernon 2013] p405
2. [Vernon 2013] p145
3. https://youtu.be/NdZqeAAIHzc?t=2053
4. https://www.ibm.com/garage/method/practices/code/domain-driven-design/
5. https://www.microservice-api-patterns.org/ZIO-FromDDDToMAPIsQS2020v10p.pdf (slide 36)
6. https://github.com/socadk/design-practice-repository/blob/master/artifact-templates/SDPR-RefinedEndpointList.md
7. https://eprints.cs.univie.ac.at/6948/1/europlop21-s16-camera-ready2.pdf
8. https://medium.com/@unmeshvjoshi/aggregate-oriented-microservices-d314eb04f2b1

{resource} – design literature

Next to REST resources, also business objects (as presented by Nbility) can be compared to aggregates in Domain-Driven Design (DDD). While the terminology may vary, the concept of grouping related objects together into cohesive units with defined boundaries exists in both cases.

Business objects align with aggregates and are objects that represent concepts, entities, or elements within the business domain. These objects encapsulate behavior, maintain state, and collaborate to fulfill business requirements. Business objects are often organized into higher-level structures or units to handle complex business operations or processes[1].

While the term *business object* is more generic and can encompass a wider range of objects within a business domain, the concept aligns with the idea of aggregates in DDD. Both concepts aim to group related objects together, define boundaries, and encapsulate behavior to achieve business goals and maintain consistency.

- ***Conclusion: the alignment of business objects, DDD aggregates and REST resources is a recognizable concept in both DDD literature and in DDD-related discussions on the internet***

1. [Millett 2015] p53, p465

{API} – design literature

The concept of the *Interface Bounded Context*, which refers to a separate context with own interface model, dedicated team, and language with integration capabilities, is already introduced in Vaughn Vernon's book[1]:

- Tempting though it may be, it is not advisable to directly expose a domain model via RESTful HTTP. This approach often leads to system interfaces that are more brittle than they need to be, as each change in the domain model is directly reflected in the system interface. There are two alternative approaches for combining DDD and RESTful HTTP. The first approach is to create a **separate Bounded Context for the system's interface layer** and use appropriate strategies to access the actual Core Domain from the system's interface model. This can be deemed a classic approach, as it views the system's interface as a cohesive whole that is simply exposed using resource abstractions instead of services or remote interfaces

An independent and separate integration model, decoupled from a bounded context implementation model, provides the advantage of stability in the interface layer and flexibility of the bounded context domain model:

- Decoupling the bounded context's implementation and integration models gives the upstream bounded context the freedom to evolve its implementation without affecting the downstream contexts[2]
- When using DDD, the REST API should always be separated from the domain model. The main reason for this is simplification - you don't want to leak the complexity of the domain model through the API to the clients[3]
- It is very important to distinguish between resources in REST API and domain entities in a domain driven design. Domain driven design applies to the implementation side of things (including API implementation) while resources in REST API drive the API design and contract[4]

1. [Vernon 2013] p145
2. [Khononov 2022] p55
3. https://stackoverflow.com/questions/33970716/why-the-domain-model-should-not-be-used-as-resources-in-rest-api
4. https://nhpatt.com/bounded-context-in-apis/

{API} – design literature

In recent literature, this concept is further developed, where not only model decoupling is advocated but also language decoupling: changes in the Ubiquitous Language of a bounded context will not impact the Open Host Service being offered and formulated in the Published Language:

- The supplier's public interface is not intended to conform to its ubiquitous language. Instead, it is intended to expose a protocol convenient for the consumers, expressed in an integration-oriented language. As such, the public protocol is called the published language[1]

It is important to mention that the Open Host service represents an integration model that simplifies the underlying bounded context domain model, intended for contexts that do not want to be hindered by the complexity of the underlying bounded context domain model. Here are some quotes:

- A public API is exposed via a contract with a simplified model whose language is published for all to understand[2];
- Second, the published language exposes a more restrained model. It is designed around integration needs. It encapsulates the complexity of the implementation that is not relevant to the service's consumers. For example, it can expose less data and in a more convenient model for consumers[3]

A variant of the Interface Bounded Context is Interchange Context[4] in which the anticorruption layer acts as an integration-specific bounded context. Team ownership however is not entirely clear[5].

1. [Khononov 2022] p88
2. [Millett 2015] p98
3. [Khononov 2022] p229
4. [Khononov 2022] p193
5. https://medium.com/nick-tune-tech-strategy-blog/gateway-interchange-contexts-899696e67848

{API} – design literature

The term 'Interface Bounded Context' is explicitly mentioned in publications by Olaf Zimmermann[1]:

- …as some APIs are based on bounded contexts, especially if dedicated interface bounded contexts are designed with the purpose of creating APIs from them
- Interface Bounded Context: Another option is to design an explicit bounded context that contains the interface to be exposed in the API. This bounded context is designated as an Interface Bounded Context in the domain model design
- Introduce a dedicated API view on selected parts of the domain model to establish a published language that exposes parts of the ubiqitous language of the domain in a controlled, managed fashion. Mark the domain model elements exposed to the API clearly as API elements

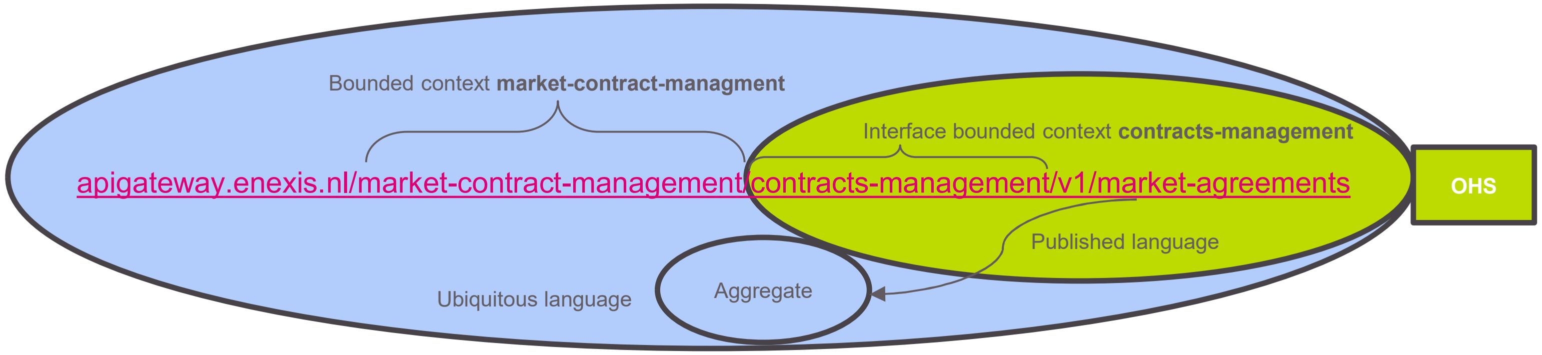
The interface bounded context represents a nested child bounded context[2] as part of the top-level parent bounded context. This relationship can be seen as a hierarchical parent - child construction. The parent bounded context defines the 'outer' scope of the context and defines team ownership and is formulated in the Ubiquitous Language. The 'inner' child interface bounded context scopes a subset of the 'outer' scope of the parent context for interfacing purposes and is formulated in the Published Language.

The {API} path of this URI strategy maps to the interface bounded context concept. It is named in the published language because it needs to be understood by other bounded contexts.

1. https://eprints.cs.univie.ac.at/6948/1/europlop21-s16-camera-ready2.pdf
2. [Newman 2015] p73

{API} – design literature

Furthermore the API name is part of the path component of the URI and since the path component should contain data organized in a hiërarchical form, the parent – child construction of top-level bounded context and interface bounded context seems to meet this requirement quite well. To depict the anatomy visually:


Bounded context **market-contract-managment**
Interface bounded context **contracts-management**
apigateway.enexis.nl/market-contract-management/contracts-management/v1/market-agreements
OHS
Published language
Ubiquitous language
Aggregate

- ***Conclusion: the API name identifies the interface bounded context as a separate bounded context nested hiërarchically within the parent top-level bounded context***

Appendix A - DDD: Bounded Context examples

Bounded Context	Bron (Object)	
Booking	• [Evans 2003]: 16 (Passenger) • [Millett 2015]: 86 • [Evans 2003]: 307	
Billing	• [Khononov 2022]: 91 (with Context map) • [Khononov 2022]: 238	
Catalogue	• [Millett 2015]: 227 • [Khononov 2022]: 238	
CRM	• [Millett 2015]: 100 (Customer model shared kernel with BC Commerce) • [Vernon 2013]: 190 • [Zimarev 2019]: 410 • [Khononov 2022]: 91 (with context map) • [Khononov 2022]: 238 • [Khononov 2022]: 321 • [Khononov 2022]: 70	
Customer Service	• [Khononov 2022]: 76 (Ticket) • Fowler: https://martinfowler.com/bliki/BoundedContext.html (Product, Customer)	
HR	• [Khononov 2022]: 96 (Employee model as shared kernel with BC Payroll)	
Identity & Access	• [Khononov 2022]: 91 (with Context map) • [Vernon 2013]: 74	
Inventory	• [Millett 2015]: 81 (Product) • [Vernon 2013]: 69 • [Zimarev 2019]: 410	

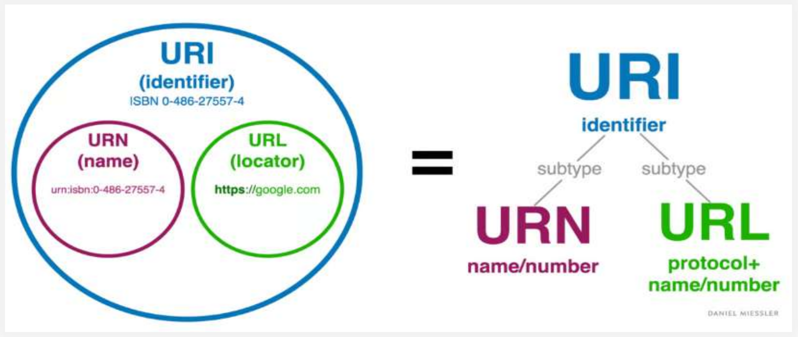
Appendix A - DDD: Bounded Context examples

Bounded Context	Bron (Object)	
Marketing	• TADD: 16 (Subscriber) • [Millett 2015]: 81 (Product) • [Khononov 2022]: 64 (Lead) • [Khononov 2022]: 91 (with Context map)	
Ordering	• [Millett 2015]: 88	
Payroll	• [Millett 2017]: 20 (Employee model as shared kernel with BC Empoyee Management) • [Millett 2015]: 96 (Employee model as shared kernel with BC Payroll)	
Procurement	• [Millett 2015]: 98 • [Millett 2015]: 81 (Product)	
Promotions	• [Millett 2015]: 660 • [Millett 2015]: 88 • [Khononov 2022]: 300 (Lead, Campaign)	
Pricing	• [Millett 2015]: 81 (Product) • [Millett 2015]: 88 • [Millett 2015]: 227	
Purchasing	• [Vernon 2017]: 79	
Sales	• [Millett 2017]: (Lead) • [Millett 2015]: 76 (Ticket) • [Zimarev 2019]: 410 • [Khononov 2022]: 64 (Lead) • [Khononov 2022]: 300 (Lead, Campaign) • Fowler: https://martinfowler.com/bliki/BoundedContext.html (Product, Customer)	
Shipping	• [Millett 2015]: 86	

Appendix B - URIs, URLs and URNs

This document is called ‘URI strategy’ and it’s good to explain the differences between URIs, URLs and URNs[1]. These concepts can be defined as follows:

- **A Uniform Resource Identifier** (URI) is a string of characters that uniquely identify a name or a resource on the internet. A URI identifies a resource by name, location, or both. URIs have two specializations known as Uniform Resource Locator (URL), and Uniform Resource Name (URN);
- **A Uniform Resource Locator** (URL) is a type of URI that specifies not only a resource, but how to reach it on the internet—like http://, ftp://, or mailto://;
- **A Uniform Resource Name** (URN) is a type of URI that uses the specific naming scheme of urn: — like urn:isbn:0-486-27557-4 or urn:isbn:0-395-36341-1;



1. https://danielmiessler.com/p/difference-between-uri-url

Appendix B - URIs, URLs and URNs

So all URLs are URIs, but not all URIs are URLs. An URL is a type of URI.

An XML namespace can be identified by some Uniform Resource Identifier (URI), either a Uniform Resource Locator (URL), or a Uniform Resource Name (URN). At Enexis we have chosen for the URN notation in the ECDM framework:

targetNamespace="urn:xenexis:ecdm:3010.01.01:DocumentAdministration:FraudDossierManagement:1:Standard“

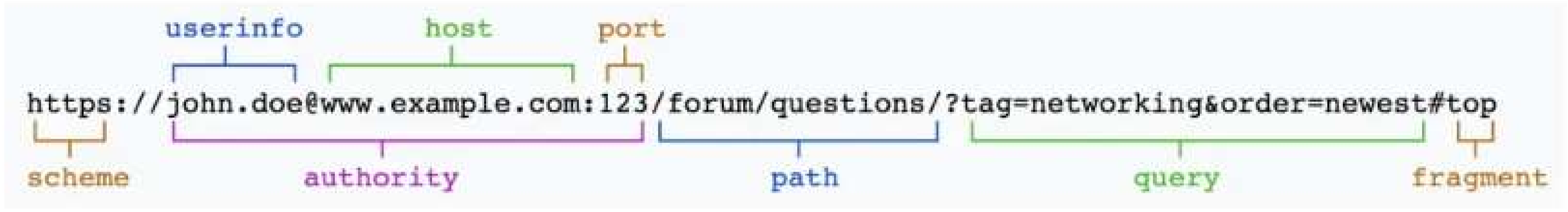
A message broker topic is typically represented as an URI, because the focus is on the identification and categorization of the topic rather than its location on the web[1].

Since this document aims at a design strategy for both URLs and URNs the overall term URI is used[2].

1. [Higginbotham 2021] p178
2. In REST literature the URI is the very often used term in association with endpoint design strategy

Appendix C - RFC3986

RFC3986 serves as the primary reference for the generic URI syntax[1]. It defines the basic components of a URI, including the scheme, authority, path, query, and fragment. It also covers the rules for character encoding, escaping reserved characters, and resolving relative references within an URI:


userinfo
host
port
https://john.doe@www.example.com:123/forum/questions/?tag=networking&order=newest#top
scheme
authority
path
query
fragment

The **path** component in RFC3986 represents the location or hierarchy of the resource within the URI's scheme and authority, providing the necessary context to identify and access the specific resource. Including an API name as part of the path component is a common practice in RESTful API design.

1. https://datatracker.ietf.org/doc/html/rfc3986

Appendix C - RFC 3986

The RFC provides guidelines for URI syntax, and it allows for flexibility in structuring the path component. For example, consider the following URI:

https://example.com/api/v1/users

In this URI, the *api* segment is included as part of the path to indicate the API, followed by the version (v1) and the resource being accessed (users). This approach helps in organizing the API endpoints and differentiating them from other resources on the server.

As long as the API name and other segments in the path component adhere to the character restrictions defined in RFC3986 and do not conflict with reserved characters or reserved meanings, it is generally considered compliant with the RFC.
In {API} - design literature is stated that the API name component is in fact a separate interface bounded context, nested in the parent bounded context, persevering the hierarchy constraint of the path component in RFC3986.

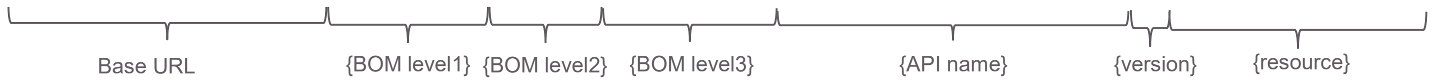
Appendix D - Sector community URI strategy

In the energy sector (EDSN) the URI structure of community APIs has been determined on the hiërarchial levels of the Nbility business objects (BOMs):

- https://api.cmf.energysector.nl (base URL of all cmf APIs)
  - /{toplevel} - (Nbility BOM level1)
    - /{sublevel1} - (Nbility BOM level2)
      - /{sublevel2} - (Nbility BOM level3)
        - /{API} - (API name)
          - /{version} - (major version of API)
            - /{resource} - (resource, collectie or register)

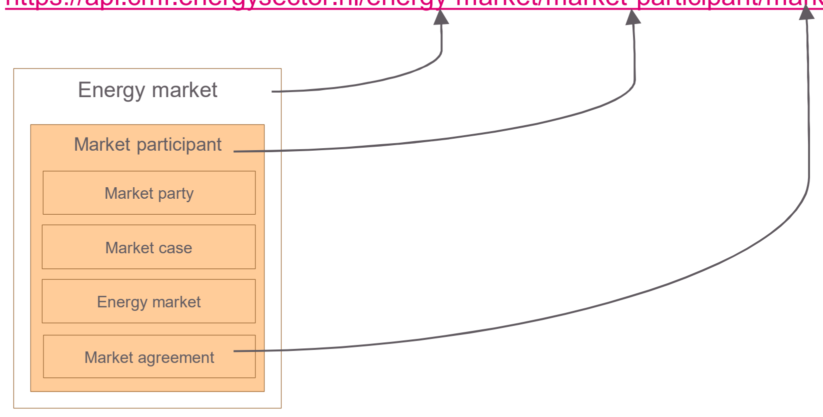
Examples:

- https://api.cmf.energysector.nl/energy-market/market-participant/market-agreement/contract-management/v1/market-agreements
- https://api.cmf.energysector.nl/energy-market/connection/facility/power-generating-modules-management/v1/powergeneratingmodules



Appendix D - Sector community URI strategy

https://api.cmf.energysector.nl/energy-market/market-participant/market-agreement/contract-management/v1/market-agreements



This approach is compliant with RFC3986 and supports the hiërarchy definition of the path component of the URI[1], however it does not apply the concept of bounded contexts as linguistic boundaries for business objectsmodels (resources) and ownership boundaries for teams.

Furthermore it restricts the API design in general, as it is focused on one particular business object model (resource) and leaves little room for additional associated resources in the API, which might be required depending on the use case.

1. https://www.tech-invite.com/y35/tinv-ietf-rfc-3986-2.html#e-3-3

Appendix D - Sector community URI strategy

The name of an API is a combination of BOM level 3 and Nbility business level 3 where possible. In the overview below a fragment of the API namings for the market domain are provided:

https://api.cmf.energysector.nl/energy-market/market-participant/market-agreement/contract-management/v1/market-agreements

{API name}

Level	Business capability level3	BOM level 3	<name API> = BOM level3 + business capability level3
6.1.1	Maintain relations with market parties	market-participant	market-participant-management
6.1.2	Handle questions of market parties	market-case	market-case-management
6.1.3	Close contracts for market services	market-agreement	contract-management
6.1.4	Manage energy markets	energy-market	energy-market-management
6.2.1	Manage energy grid connections	connection	connection-management
6.2.2	Manage installations behind connections	facility	facility-management
6.2.3	Manage congestion areas	congestion-area	congestion-area-management
6.3.1	Execute energy market procedures	market-request	market-request-management
6.3.2	Exchange energy predictions	energy-schedule	energy-schedule-communication
6.3.3	Determine energy exchange	energy-exchange	energy-exchange-quantification
6.3.4	Allocate energy exchange to market parties	market-invoice	market-invoice-assignment
6.3.5	Communicate upward or downward regulation demand	regulation-demand	regulation-demand-notification
6.3.6	Manage upward or downward regulation bids	regulation-offer	regulation-offer-management

Appendix D - Sector community URI strategy

The API name is structured according to the naming covention overview below. Please note that this is still in an expirimental phase. Motivation however is clear: by using predetermined keywords in the API naming, the discoverablity, uniformity and architecure role of the API becomes clear:

	Keyword	Description
	management	… if the API impacts the total life cycle of a resource (all CRUD operations)
	retrieval	… if the API only retrieves information (GET, POST query)
	assignment	… if the API assigns something to a party (POST)
	quantification	… if the API performs calculations or determines something numerically
	notification	… if the API publishes an event or information
	communication	… if the API exchanges information bi-directional, without mutations (question / answer)

Examples:

- https://api.cmf.energysector.nl/energy-market/market-participant/market-agreement/**contract-management**/v1/market-agreements
- https://api.cmf.energysector.nl/energy-market/connection/facility/**power-generating-modules-management**/v1/powergeneratingmodules
- https://api.cmf.energysector.nl/energy-market/ market-process/energy-exchange/**measurement-series-notification**/v1/measurement-series

{API name}

Appendix E: Message Broker Topics

Message broker topics[1] can be named after bounded contexts in an event-driven architecture. Naming streaming topics and message queues after bounded contexts can help maintain a clear separation and organization of events based on the different parts of the system they belong to[2].

This URI strategy can very well be applied to a general naming convention for queues and topics. When we consider the current Enexis naming conventions:

- AMQ queue[3]: pattern **<team>.<ECDM business function>.<version>** - example: **csti.document-managent.getDocument.v1**
- Kafka topic[4]: pattern **<app/dis-number-service-now>_<stream-name>** - example: **APP7699_MeterReadingValues**

we observe that teams, applications and functional message names are part of the naming pattern. Based on the URI strategy of this document the naming convention could be:

pattern: **<context>.<resource>**

Where **<context>** adheres to design rule ENX-URI-001 and represents teams and applications, whereas **<resource>** adheres to ENX-URI-006 and denotes events, commands, or other types of data relevant to the downstream bounded contexts.

1. [Higginbotham 2021] p168 for further explanation on the terminology
2. https://levelup.gitconnected.com/kafka-for-engineers-975feaea6067
3. https://enexis.sharepoint.com/sites/WIKI-010520183/Wiki%20Paginas/AMQ%20Queues%20Topic%20Destinations%20and%20Authorization.aspx
4. https://enexis.sharepoint.com/sites/WIKI-010520183/Wiki%20Paginas/035.%20Naming%20Conventions%20(Axual%20Streaming).aspx

Appendix F: Dutch government URI examples

API	Org	URI
BAG API Huidige Bevragingen[1]	Kadaster	https://api.bag.kadaster.nl/esd/huidigebevragingen/v1/
BAG API Individuele Bevragingen[2]	Kadaster	https://api.bag.kadaster.nl/lvbag/individuelebevragingen/v2
Terugmelding API[3]	Kadaster	https://api.kadaster.nl/tms/v1/terugmeldingen
BRT API[4]	Kadaster	https://brt.basisregistraties.overheid.nl/api/v2
Bevraging Ingeschreven Persoon [5]	VNG	https://www.haalcentraal.nl/haalcentraal/api/brp
Authorisatie component[6]	VNG	https://autorisaties-api.vng.cloud/api/v1
Omgevingsdocument downloaden[7]	DSO	https://service.omgevingswet.overheid.nl/publiek/omgevingsdocumenten/api/downloaden/v1
Omgevingsdocument presenteren[8]	DSO	https://service.pre.omgevingswet.overheid.nl/publiek/omgevingsdocumenten/api/presenteren/v7
Centrale OIN Raadpleegvoorziening[9]	Logius	https://oinregister.logius.nl/api/v1

1. Basisregistratie Adressen en Gebouwen: https://developer.overheid.nl/apis/kadaster-bag-current
2. Basisregistratie Adressen en Gebouwen: https://developer.overheid.nl/apis/kadaster-bag-individual
3. Terugmelding: https://developer.overheid.nl/apis/kadaster-terugmeldingen
4. Basisregistratie Topografie: https://developer.overheid.nl/apis/kadaster-brt
5. https://developer.overheid.nl/apis/vng-bevraging-ingeschreven-persoon
6. https://developer.overheid.nl/apis/vng-ac
7. https://developer.overheid.nl/apis/dso-omgevingsdocument-downloaden
8. https://developer.overheid.nl/apis/dso-omgevingsdocument-presenteren
9. https://developer.overheid.nl/apis/logius-cor

Bibliography

- [Biehl 2015] M. Biehl, API Architecture: The Big Picture for Building APIs. API University Series, 2015
- [Evans 2003] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003;
- [Khononov 2022] V. Khononov, Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy. O'Reilly 2022
- [Higginbotham 2021] J. Higginbotham, Principles of Web API Design: Delivering Value with APIs and Microservices. Addison-Wesley, 2021
- [Millett 2015] S. Millett & N. Tune, Patterns, Principles, and Practices of Domain-Driven Design. John Wiley & Sons, Inc, 2015
- [Millett 2017] S. Millett, The Anatomy of of Domain-Driven Design. DDD Europe 2022
- [Newman 2015] S.Newman, Building Microservices: Designing Fine-Grained Systems. O'Reilly 2015
- [Sloos 2020] T. Sloos, DSO URI strategie. Digitaal Stelsel Omgevingswet, 2020
- [Vernon 2013] V. Vernon, Implementing Domain-Driven Design. Addison-Wesley, 2013
- [Zimarev 2019] A. Zimarev, Hands-On Domain-Driven Design with .NET Core. Packt Publishing 2019
- [Zimmermann 2021] O. Zimmermann, Patterns on Deriving APIs and their Endpoints from Domain Models. EuroPLoP '21, Irsee, Germany July, 2021
- [Zimmermann 2022] O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, C. Pautasso, Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges. Addison-Wesley, 2022

SAMEN WERKEN WE AAN EEN **BETROUWBARE EN DUURZAME ENERGIEVOORZIENING** VOOR VANDAAG ÉN VOOR DE TOEKOMST.

For more information, please contact: CST-Integratie@enexis.nl

ENEXIS GROEP