Providing APIs (Application programming interface)

> ***Warning*** *This file is maintained at Conduction's Google Drive Please make any suggestions of alterations there.*

The gateway provides an API for other applications to use and consume APIs from sources in a way that gateway acts both as a provider and consumer of APIs. How to consume APIs from the gateway is further detailed under the Sources chapters. This chapter deals with providing APIs from the gateway to other applications

Endpoints

Each api consists of a collection of endpoints. These provide the basis location that a call can be made to e.g. `/api/pets`.

Contex

The gateway always views each call to an endpoint in its own context determined by three main aspects.

- (User) Who is making the call? e.g. User John
- ([Application]Applications.md) How is he making the call? e.g. from the front desk applications
- (Process) For which process is he making the call? e.g. client registration

The call is then handled by the request service.

Generic API Functionality

A normal filter: {propertyName}={searchValue} e.g. `firstname=john`

A property is IN array filter: {propertyName}[]={searchValue1} e.g. 'firstname[]=john' {propertyName}[]={searchValue2} e.g. 'firstname[]=harry'

or

A normal filter with method: {propertyNema}[method]={searchValue} e.g. 'firstname[case_insensitive]=john'

A property is IN array filter with method: {propertyNema}[method][]={searchValue1} e.g. 'number[int_compare][]=2' {propertyNema}[method][]={searchValue2} e.g. 'number[int_compare][]=5' Note that not every method can be used like this

All functional query parameters always start with an _ to prevent collisions with property names e.g. _order

Methods

method less queries (e.g. `firstname=john`) are treated as exact methods `firstname[exact]=john`

- **[exact] (default) exact match** Only usable on properties of the type `text`, `integer` or `datetime`. Seea
- **[case_insensitive] (default) case insensitive searching** Only usable on properties of the type `text`, uses the regex function under the hood in a case insensitive way.
- **[case_sensitive] case sensitive searching** Only usable on properties of the type `text`, uses the regex function under the hood in a case sensitive way.

- **[like] wildcard search** Only usable on properties of the following types: `text` , `integer` `datetime` These types work the same as a regex search, but wraps the value in `.*` creating `.*$value.*` and sets the matching pattern to case insensitive and multi-line.This means you can search for single words in sentences or text. Keep in mind that `like` will search for complete occurrences zo $name[like] =John Doe will only return hits on “John Doe” and not records containing either John OR Doe.
- **[>=] equal or greater than** Only usable on properties of the type `integer` , will automatically cast the searched value to integer to make the comparison
- **[>] greater than** Only usable on properties of the type `integer` , will automatically cast the searched value to an integer to make the comparison
- **[<=] equal or smaller than** Only usable on properties of the type `integer` , will automatically cast the searched value to an integer to make the comparison
- **[<] smaller than** Only usable on properties of the type `integer` , will automatically cast the searched value to an integer to make the comparison
- **[after] equal or greater than** Only usable on properties of the type `date` or `datetime`
- **[strictly_after] greater than** Only usable on properties of the type `date` or `datetime`
- **[before] equal or smaller than** Only usable on properties of the type `date` or `datetime`
- **[strictly_before] smaller than** Only usable on properties of the type `date` or `datetime`
- **[regex] compare the values based on regex** Only usable on properties of the type `string`
- **[int_compare] will cast the value of your filter to an integer before we filter with it.** Useful when the stored value in the gateway cache is an integer, but by default you are searching in your query with a string “1012”. Works with the property IN array filter like this: {propertyNema} [int_compare][]={searchValue1}
- **[bool_compare] will cast the value of your filter to a boolean before filtering.** Useful when the stored value in the gateway cache is a boolean, but by default you are searching in your query with a string “true”. Works with the property IN array filter like this: {propertyNema}[bool_compare] []={searchValue1}

> ***Note*** *When comparing dates we use the PHP [dateTime($value)] function to cast the strings to dates. That means that you can also input strings like* `now` . `yesterday` *see the full list of [relative formats].*

Ordering the results

_order[propertyName] = desc/asc

> ***Note*** *The* `_search` *order property currently also supports* `order` *for backwards compatibility*

Working with pagination

Requests to collections (e.g. more then one object) are encapsulated in an response object, the gateway automatically paginates results on 30. You can set the amount of items per page through the `_limit` query parameter. There is no upper limit to this parameter, so if desired, you could request 10000 objects in one go. This does however come with a performance drain because of the size of the returned response in bytes where the main throttle is the internet connection speed of the transfer combined with the size of individual objects. We therefore suggest not to user limits greater than 100 in frontend applications.

```
{
  "total":100,
  "limit":30,
  "pages":4,
  "page":1,
  "results":[]
}
```


- **_limit**
- **_page**
- **_start**

> ***Note*** *The pagination properties currently also support backwards compatibility by removing the _ part. Meaning that they may also be used as* `limit` *,* `page` *and* `start`

The search index

The Common Gateway automatically creates a search index of all objects based on the text value of their properties (non-text values are ignored). This search index can be used when approaching API endpoints through the special `_search` query parameter. Search functions as a wildcard.

e.g. `_search=keyword`

By default the search query searches in all fields. If you want to search specific properties you can do so by defining them as methods. You can search properties fields (in an OR configuration) by separating them through a comma, and supplying them in the method.You can also search in sub properties e.g. `_search[property1,property2.subProperty]=keyword` .

> ***Note*** *The* `_search` *property currently also supports* `search` *for backwards compatibility*

Limiting the return data

In some cases you either don’t need or want a complete object. In those cases it's good practice for the consuming application to limit the field in its return call. This makes the return messages smaller (and therefore faster), but it is also more secure, because it prevents the sending and retention of unnecessary data.

The returned data can be limited using the _fields query parameter. This parameter is expected to be an array containing the name of the requested properties. It’s possible to include nested properties using dot notation. Let’s take a look at the following example. We have a person object containing the following data:


```
{
  "firstname":"John",
  "lastname":"Doe",
  "born":{
    "city":"Amsterdam",
    "country":"Netherlands",
    "date":"1985-07-27"
  }
}
```


Of we then query using `_fields[]=firstname&_fields[]=born.date` we would expect the following object to be returned:

```
{
  "firstname":"John",
  "born":{
    "date":"1985-07-27"
  }
}
```

> ***Note*** *The _fields property may be aliased as _properties*

_remove is specific unset

Specifying the data format

The gateway can deliver data from its data layer in several formats. Those formats are independent from their original source, e.g. A source where the gateway consumes information from might be XML based, but the gateway could then provide that information in JSON format to a different application.

The data format is defined by the application requesting the data through the `Accept` header.

Mapping the data (transformation)

It is also possible to transform incoming data by providing a mapping object, more information about creating mapping objects can be found under mappings.

Mappings can be passed through the gateway by url encoding the desired mapping and passing it trough the _mappings query parameter.

> ***Note*** *It is discouraged to use mappings in this context since it makes the API restFull.*

> ***Warning*** *This file is maintained at Conduction's Google Drive Please make any suggestions of alterations there.*

Action Handlers

...

Architecture

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions of alterations there.*

Main Process

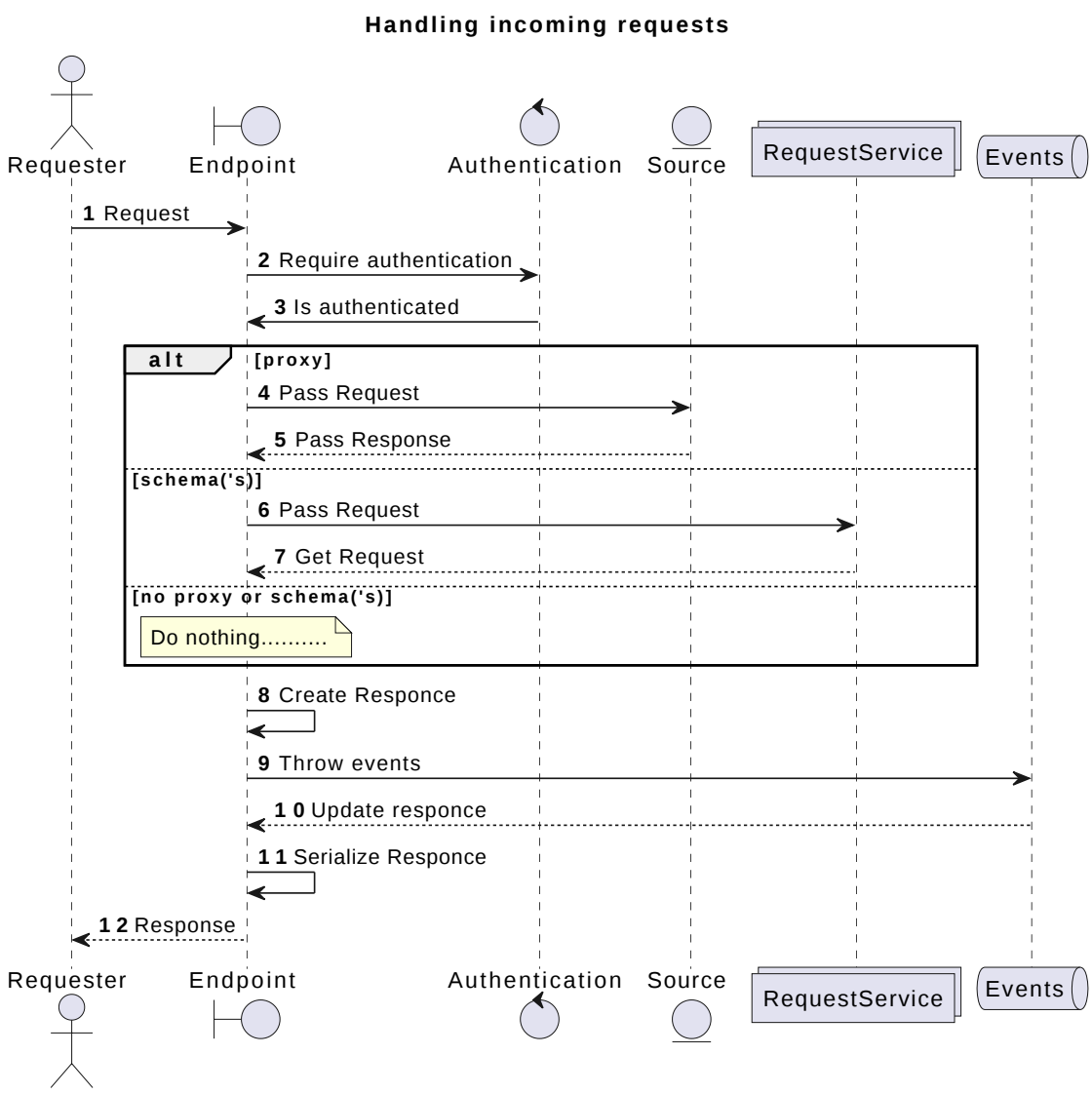
The Common Gateway is designed to handle a multitude of request types. Upon receiving a request, the Gateway commences a series of steps to handle the request and provide an appropriate response. The main process is as follows:

1. **Receipt of Request:** The Gateway can receive a wide range of request types. These include: HTTP GET, PUT, POST, etc., containing a JSON, XML, or SOAP object A browser GET request (HTML) A user posting a form A user downloading a file
2. **Endpoint Identification:** Once the Gateway receives a request, the first task is to identify the endpoint. The endpoint is a primary determinant of how the request will be processed. Endpoints are mapped to specific functions or services in the Gateway, and the identified endpoint dictates the necessary operations to be performed.

3. **Response Generation:** After the request has been processed according to the rules of the identified endpoint, the Gateway generates an appropriate response. This could be a JSON or XML response for API requests, an HTML page for browser requests, or the requested file for download requests.

This main process forms the backbone of the Common Gateway's operation. It ensures that any incoming request can be accurately interpreted and handled, and that an appropriate response can be generated and returned to the user.

The use of endpoints allows the Gateway to be highly flexible and adaptable, capable of handling a wide variety of requests and responses, making it an ideal solution for various use cases.



Main request components

1. Endpoint
2. Source
3. Datalayer
4. Request Service

5. Events

Two codebases to rule them all

As part of our ongoing efforts to improve and streamline our development process, we're transitioning from a single codebase setup to a library-based one. This shift involves migrating code from the existing repository at ConductionNL/commonground-gateway to a new repository at CommonGateway/CoreBundle.

This transition is not just about moving code; it's about enhancing the quality of our codebase. We're taking this opportunity to clean up both our code and documentation and to increase the coverage of our unit tests. However, this is a significant undertaking, and we're not finished yet. As of now, Entities, Controllers, and some workflows, including unit testing, are still located in the old repository. We expect the migration to be completed by summer 2023.

An integral part of this transition is the decoupling of client-specific code from the core codebase. This code will now reside in separate plugin repositories. This separation of concerns ensures a cleaner, more maintainable codebase, and allows for more customizable client implementations.

Please bear with us during this period of transition. We're confident that these changes will result in a more robust and efficient gateway, and we appreciate your understanding and patience during this time.

Design Decisions

API First

An API-first approach means that for any given development project, your APIs are treated as "first-class citizens." An API-first approach involves developing APIs that are consistent and reusable, which can be accomplished by using an API description language to establish a contract for how the API is supposed to behave. The specification we use is the OpenAPI Specification. You can view the latest version of this specification (3.0.1) on GitHub.

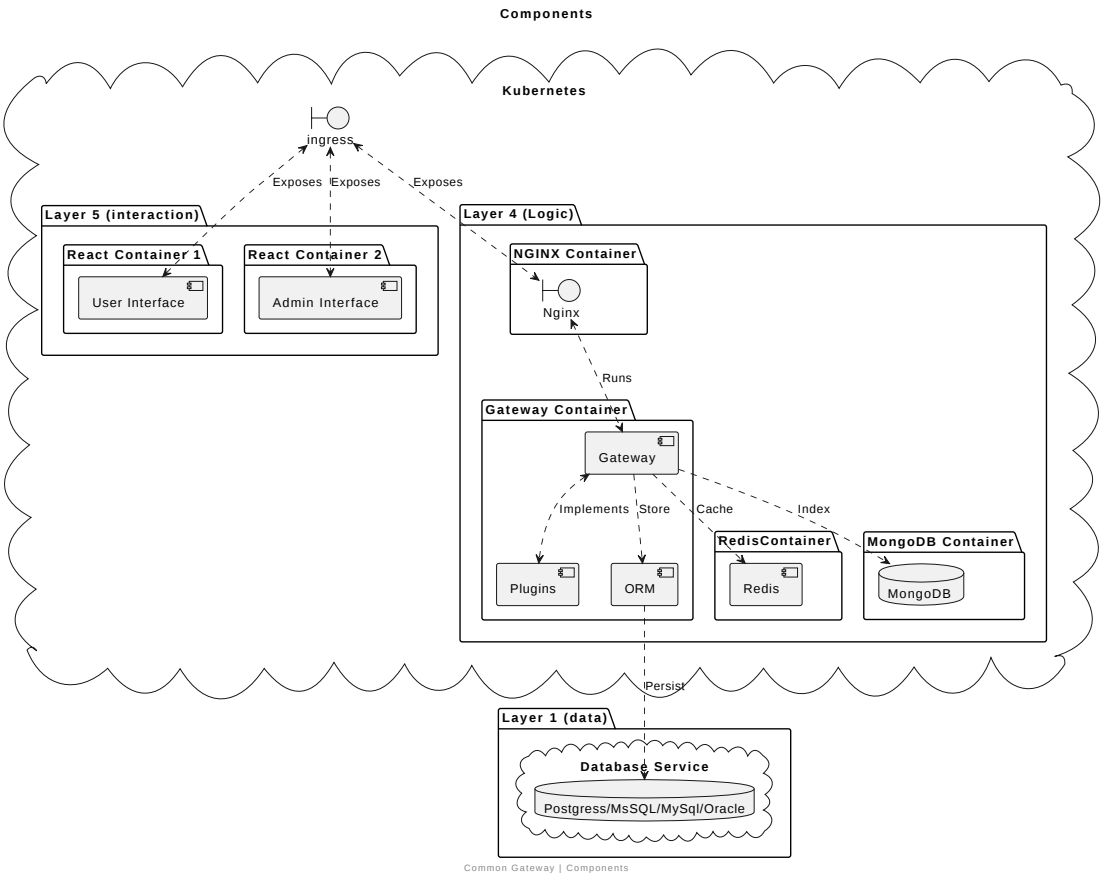
Documentation

We host technical documentation on Read the Doc's and general user information on GitHub pages, to make the documentation compatible with GitHub we document in markdown (instead of reStructuredText). Documentation is part of the project and contained within the /docs folder.

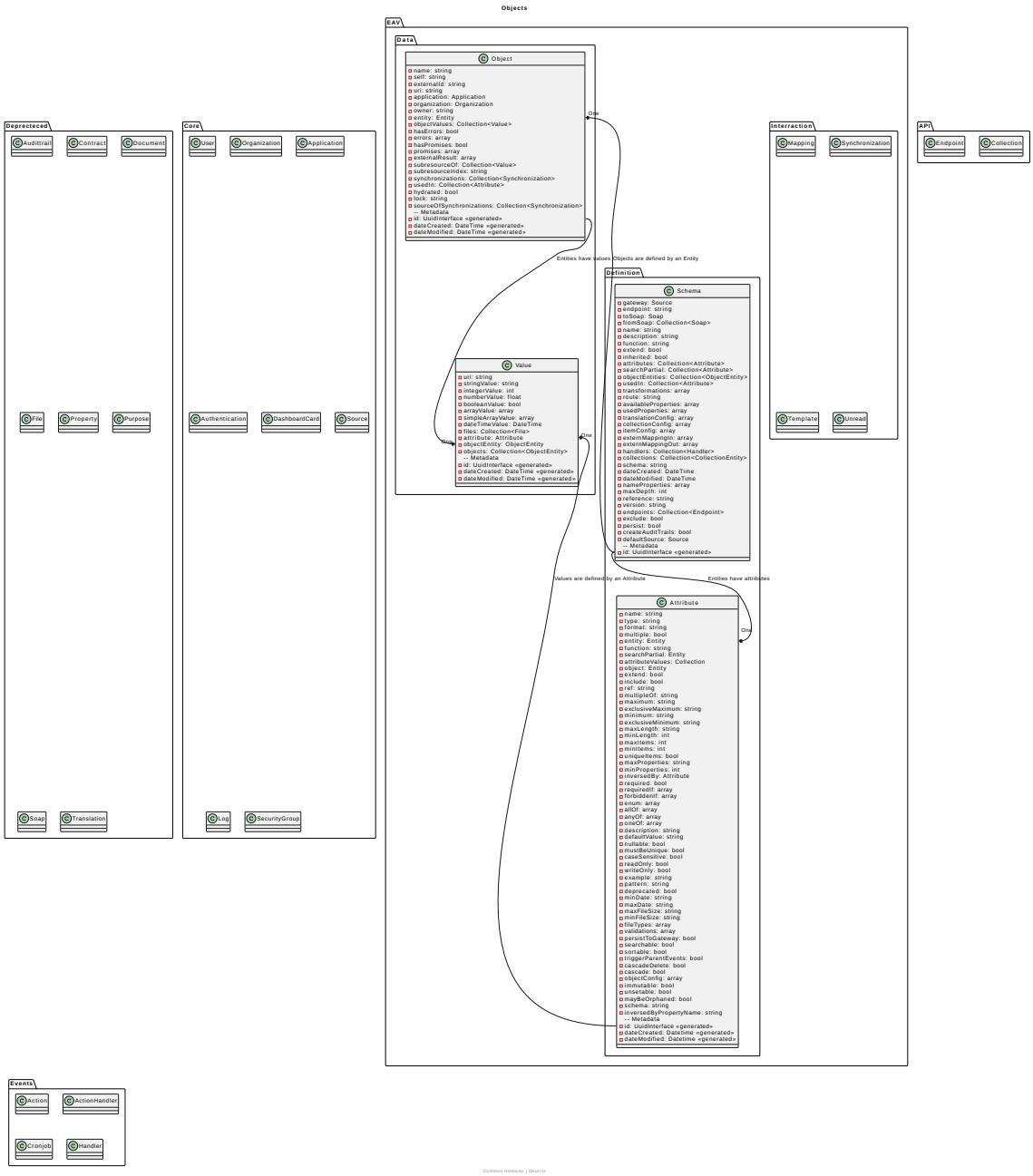
Common Ground

All applications are developed following the Common Ground standards on how a data exchange system should be: modular and open-source. More information on Common Ground can be found here

Kubernetes set up


Components
Kubernetes
ingress
Exposes
Exposes
Exposes
Layer 5 (interaction)
React Container 1
User Interface
React Container 2
Admin Interface
Layer 4 (Logic)
NGINX Container
Nginx
Runs
Gateway Container
Gateway
Implements
Store
Cache
Index
Plugins
ORM
RedisContainer
Redis
MongoDB Container
MongoDB
Persist
Layer 1 (data)
Database Service
Postgress/MsSQL/MySql/Oracle
Common Gateway | Components

Objects



Authentication

> ***Warning*** *This file is maintained at the Conduction [Google Drive](). Please make any suggestions of alterations there.*

Authentication is an essential part of securing the gateway. It involves validating the identity of a client—whether that's a user, a device, or another application—before allowing it to access the system's resources. There are several ways for applications to authenticate themselves to the gateway, each with its own use cases and security considerations.

Uses Classes

- AuthenticationService

Applications

Application Key

Application keys provide a simple and straightforward method for authenticating an application. They are unique identifiers that an application presents when making a request, acting as a sort of password. However, application keys should be safeguarded as they can potentially provide access to sensitive data and services if compromised.

JWT Token (preferred)

JWT (JSON Web Token) is a compact and URL-safe means of representing claims to be transferred between two parties1. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity-protected with a Message Authentication Code (MAC) and/or encrypted.

JWT tokens should be included calls using the “Authorization” header en prefixed with bearer.

ZGW JWT Token

ZGW (Zaakgericht Werken) JWT Tokens are a specific type of JWT token, commonly used in the Netherlands for government-related APIs. They follow a specific standard and carry additional information that is pertinent to the ZGW context.

Two-Way SSL

Two-way SSL, also known as mutual SSL, is a process in which both the client and the server authenticate each other through the verification of each other's digital certificates. This method ensures that both parties are who they claim to be and can trust each other, thereby providing an additional layer of security.

IP Whitelisting

IP whitelisting is a security feature that restricts access to a network or a system only to trusted users. If you are using the gateway in an API Gateway setup, you can set up IP whitelisting to only accept calls from an application if they originate from either a specific IP address or an IP address range.

> ***Warning***
>
> *IP Whitelisting should never be used alone as IP addresses can be easily spoofed. Instead, it should be used as an additional authentication requirement in a machine-to-machine context. In a Web gateway context, IP Whitelisting can lead to undesired results due to the dynamic nature of client IP addresses in such contexts.*

Domain Whitelisting

Domain whitelisting is a security feature that allows access to a system only from specific domain names. If you are using the gateway in a Web Gateway setup, you probably want to ensure that it only serves your own site. This helps prevent cross-site scripting attacks and ensures that other sites don't misuse your services.

> ***Warning***
>
> *Domain whitelisting cannot be used in a machine-to-machine context because in most cases, the requesting machine won't have a domain. Use IP whitelisting in those contexts instead.*

Users

Integrated Identity Provider

An Integrated Identity Provider (IdP) is a system entity that creates, maintains, and manages identity information for principals and provides principal authentication to other service providers within a federation. This authentication process involves validating user credentials and providing identity data to applications for authorization decisions.

External Identity Provider

An External Identity Provider is an authentication service that is built, hosted, and managed by a third-party service provider. It allows users to authenticate using a single set of credentials stored externally, without the need for additional passwords or usernames.

One common protocol used for this purpose is OAuth 2.0. OAuth 2.0 is an authorization protocol that enables applications to obtain limited access to user accounts on an HTTP service, such as Facebook, GitHub, and DigitalOcean. It works by delegating user authentication to the service that hosts authentication data.

Authorization

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions of alterations there.*

This document explains how authorization is managed in the Common Gateway project, following the principles of Role-Based Access Control (RBAC).

Uses Classes

- RequestService

Role-Based Access Control (RBAC)

In the Common Gateway, both users and applications are granted permissions based on RBAC. For information on how users and applications are authenticated, please refer to the Authentication page.

In our system, a user or an application can be assigned one or more roles. We refer to these roles as "groups" in our applications. Each group is associated with a set of permissions, which we call "scopes".

A key feature of our RBAC implementation is that groups can inherit scopes from other groups. This allows us to create a hierarchy of groups with progressively broader scopes.

Example

Here is an example to illustrate the inheritance of scopes in our RBAC system:

- **Anonymous**: This group has the most basic set of scopes. It represents users or applications that have not been authenticated.
- **User**: This group inherits all scopes from the Anonymous group. In addition, it has additional scopes that are specific to authenticated users.
- **Manager**: This group inherits all scopes from the User group. It has additional scopes that enable management functions.
- **Administrator**: This group inherits all scopes from the Manager group. As the group with the highest level of access, it has all available scopes.

This hierarchy allows for a clear and manageable organization of scopes. It ensures that each user or application has only the permissions it needs to perform its tasks, in line with the principle of least privilege.

Scope Inheritance

In the Common Gateway project, we establish a hierarchy of groups, each inheriting scopes from the group that lies below it. This "bottom-up" inheritance mechanism allows for efficient scope management, as each group inherits all the scopes of its subordinates, with any additional scopes assigned manually. For example, if there exists a 'Manager' group that sits above a 'User' group, the 'Manager' will inherit all scopes from the 'User'. The 'User' group may, in turn, inherit scopes from an 'Anonymous' group, leading to a situation where a 'Manager' possesses the scopes of both 'User' and 'Anonymous' groups.

Scopes Definition

Admin scopes

Scopes within the Common Gateway are defined based on the CRUD (Create, Read, Update, Delete) operations. A scope can be applied to an entire system aspect (e.g., 'cronjobs.READ') or to a specific object (e.g., '[uuid].read').

Below is a breakdown of system aspects and their possible scopes:

System Aspect	CREATE	READ	UPDATE	DELETE	SPECIAL
Actions	Yes	Yes	Yes	Yes	RUN
Sources	Yes	Yes	Yes	Yes	-
Cronjobs	Yes	Yes	Yes	Yes	RUN
Endpoints	Yes	Yes	Yes	Yes	-
Objects	Yes	Yes	Yes	Yes	REVERT
Schemas	Yes	Yes	Yes	Yes	-
Logs	Yes	Yes	Yes	Yes	-
Plugins	Yes	Yes	-	Yes	UPDATE
Collections	Yes	Yes	Yes	Yes	-
Mappings	Yes	Yes	Yes	Yes	-
Templates	Yes	Yes	Yes	Yes	-
Users	Yes	Yes	Yes	Yes	-
Groups	Yes	Yes	Yes	Yes	-
Applications	Yes	Yes	Yes	Yes	-
Organizations	Yes	Yes	Yes	Yes	-

Note: In the table above, a '-' indicates that the scope is not applicable for that aspect. For example, the 'CREATE' operation is not applicable to 'Objects', and the 'UPDATE' operation is not applicable to 'Plugins' but they have a special 'UPDATE' scope.

This scope inheritance and definition mechanism provides a flexible and robust system for managing access and operations within the Common Gateway project.

There are also some special scopes **Cronjobs/Actions:RUN** The ability to manually run an action or cronjob **Objects :REVERT ** The ability to manually revert an object to an earlier version **Plugins :UPDATE ** The

ability to manually update a plugin to a newer version **admin.method ** The ability to access all /api endpoints based on request method.

The default admin user has all admin.method scopes (admin.GET, admin.POST etc) by default.

API scopes

API scopes should be assigned to users through their SecurityGroups which can be added through the admin user interface. If you would like to work with an /api endpoint for a specific Schema the scope format is: `schemas.referenceOfSchema.GET` and an actual example is: `schemas.https://example.com/schema/example.schema.json.GET`.

If you have an endpoint with a configured proxy you need the scope of the source to work with it, the format is: `sources.referenceOfSource.GET` and an example scope is: `sources.https://example.com/source/api.petstore.source.json.GET`.

Common Gateway: Ownership and Creation

In the context of the Common Gateway project, it's crucial to understand the difference between the roles of an 'Owner' and a 'Creator'. These two roles possess different levels of control over objects within the system, and each has specific rights and limitations.

Owner vs. Creator

Owner

The 'Owner' of an object in the system has full control over it. This means that they can perform all CRUD (Create, Read, Update, Delete) operations on the object, including changing its properties, modifying its functionality, and even deleting it. Essentially, the owner has all rights to any object they own.

In addition to these rights, the owner also has the unique authority to transfer ownership to another user, application, or organization. This allows for flexibility in management and control, as the ownership can be shifted according to the needs of the project or team.

Creator

The 'Creator' of an object, on the other hand, has no rights. It is merely a transactional log of the user or application that created an object in the first place. In most cases the creator becomes the owner when creating an object.

Multitenancy in the Common Gateway

Multitenancy is a key concept in the Common Gateway project. It allows multiple independent instances of users and applications to operate within the same environment, while maintaining distinct, secure access to their respective objects. In this project, multitenancy is implemented at the organization level, meaning that all objects are tied to an organization, and users and applications can only interact with objects that belong to the same organization.

Object Ownership and CRUD Rights

Within an organization, users and applications have CRUD (Create, Read, Update, Delete) rights to objects. However, these rights are restricted to the scope of their respective organization. This means that a user or application from one organization cannot access or manipulate the objects of another organization.

For instance, if a user belongs to Organization A, they cannot read or modify the data of Organization B unless they have been granted specific access to Organization B.

Switching Between Organizations

If a user is a part of multiple organizations, they must manually switch between these organizations to exercise their rights within each. When a user switches organizations, their context changes, and they can now interact with the objects of the newly selected organization.

For example, if a user belongs to both Organization A and Organization B, they would initially have access to the objects of Organization A. If they want to access the objects of Organization B, they would need to switch their active organization to Organization B. This ensures that data is securely partitioned between organizations, preserving the integrity and security of each organization's data.

Maintaining Multitenancy

Multitenancy is maintained through one of two methods:

- **Single Database Setup:** In a single database setup, the organization is always added as a query parameter in the database operations. This ensures that only the data corresponding to the active organization is fetched, maintaining data separation between different organizations.
- **Multiple Database Setup:** In a multiple database setup, each organization has its own separate database. Traffic is routed to the specific database that corresponds to the active organization. This is the preferred setup because it provides a higher level of data isolation and can better handle the scale of large organizations.

In both cases, the principle of multi tenancy is preserved. Users and applications only have access to their own organization's data, ensuring data security and privacy across all organizations in the Common Gateway environment.

Code Quality

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions of alterations there.*

Code quality is a central pillar of our development philosophy. For us, it signifies easy-to-interpret, well-documented, and maintainable code. This means that we aim to reduce cyclomatic complexity and cognitive strain, while clearly defining units of code for specific tasks and nothing more. High-quality code is essential for rapid development, especially when code needs to be revisited months later.

Writing good code

Writing good, maintainable code is a crucial part of any software development process. It ensures that the codebase is easy to understand, modify, and extend, thereby increasing the efficiency of the developers and the overall quality of the software.

- **No Dead Code:** Dead code is code that is no longer in use or never used, including unused variables, functions, or even modules. It should be removed as it creates unnecessary complexity and can cause confusion and mistakes.
- **No Code Duplication**: Code duplication is a known code smell and should be avoided. It makes the codebase harder to maintain and increases the likelihood of bugs. Use principles of DRY (Don't Repeat Yourself) to avoid duplication.
- **Keep Units of Code Short and Simple**: Long methods or functions can be difficult to understand, test, and debug. It is advisable to break them down into smaller, simpler units that do one thing well.
- **Separation of Concerns**: Each module or component of the software should have a single responsibility. This design principle, known as the Single Responsibility Principle (SRP), makes the software easier to maintain and understand.

- **Loosely Coupled Architecture**: Coupling refers to the degree to which one module depends on other modules. High coupling leads to a fragile system that is hard to change and understand. It's better to have a loosely coupled architecture where modules interact through well-defined interfaces.
- **Keep the Codebase as Small as Possible**: A smaller codebase is easier to understand, test, and maintain. It also reduces the risk of bugs. Remove unused code, avoid unnecessary complexity, and strive for simplicity.
- **Tested Software**: Tests are crucial to ensure that the software works as expected and to prevent the introduction of bugs. Automated tests are particularly useful as they can be run frequently and catch regressions early.
- **Refactoring**: Code smells are indicators of potential problems in the code. They might not be causing a problem now, but they increase the risk of bugs in the future. Regular refactoring helps keep the codebase clean and maintainable.

One of the good resources for writing maintainable software is "Building Maintainable Software" by Joost Visser from the Software Improvement Group. This book provides practical guidelines for writing clean, maintainable software and should be a part of every developer's toolkit.

CI/CD Pipeline

Our continuous integration/continuous delivery (CI/CD) pipeline is a crucial part of our software development process.

CI/CD is a method to frequently deliver apps to customers by introducing automation into the stages of app development. The main concepts attributed to CI/CD are continuous integration, continuous delivery, and continuous deployment.

CI/CD bridges the gaps between development and operation activities and teams by enforcing automation in building, testing, and deployment.

Codacy

Codacy is an automated code review tool that helps developers to save time in code reviews and to tackle technical debt. It uses static code analysis to identify new static analysis issues, code coverage, code duplication, and code complexity in every commit and pull request, directly from your Git workflow.

PHPCBF and PHPCS

PHPCBF (PHP Code Beautifier and Fixer) and PHPCS (PHP Code Sniffer) are tools that we use to ensure that our code adheres to our chosen coding standards.

PHPCS is a tool that checks your PHP code to see if it adheres to the specified coding standards. It can even check your CSS and JavaScript.

PHPCBF is the companion tool to PHPCS and can be used to automatically correct coding standard violations. PHPCBF works by taking the PHP Code Sniffer tokenized version of a file and making modifications directly to that, then writing out the changes.

PHPUnit

PHPUnit is a framework that we use for unit testing our PHP code. Unit testing is a method where individual units of source code are tested to determine if they are suitable for use. It helps us to verify if the logic of individual units of our source code is working correctly.

Postman

Postman is a collaboration platform for API development. It's used for building, testing, and modifying APIs. Postman helps us ensure that any APIs we create are functioning as intended, and allows us to create mock servers, document our APIs, and more.

Deployment Process

Despite having a CI/CD pipeline, we do not deploy from it. Instead, we use Kubernetes and Harbor for defining deployments on the client side.

Kubernetes is an open-source platform designed to automate deploying, scaling, and operating application containers. Harbor, on the other hand, is an open-source cloud native registry that stores, signs, and scans content.

By leveraging the combination of Kubernetes and Harbor, we ensure our deployment process is robust, scalable, and secure. This setup gives us the flexibility to manage our deployments effectively according to each client's specific requirements and infrastructure.

Code Formatting

PHP Standards Recommendations (PSR) We adhere to the PSR-1 (Basic Coding Standard) and PSR-12 (Extended Coding Standard), which have been established by the PHP Framework Interop Group (PHP-FIG). These standards provide rules about how PHP code should be formatted and are generally accepted across the PHP community.

Symfony Coding Standards

In addition to the PSR-1 and PSR-12 standards, we follow the Symfony Coding Standards, which include several additional conventions such as using Yoda conditions and prefixing abstract classes with Abstract.

Doc blocks and inline comments

In the journey towards maintaining high-quality code, documentation plays a crucial role. It provides a clearer understanding of the functionality and purpose of different parts of the codebase, enhancing readability and maintainability.

Rich DocBlocks and inline comments are two powerful tools in this regard.

Code Reviews

Four-Eye Principle

For all normal pull requests, we follow the Four-Eye Principle, meaning that at least two team members must review and approve the changes before they can be merged into the main or development branch. This ensures that at least four eyes have seen the code, minimizing the chances of bugs or issues going unnoticed.

Six-Eye Principle

For pull requests that contain core changes—significant modifications that affect the fundamental operation of our application—we follow the Six-Eye Principle. This means that at least three team members must review and approve the changes. Increasing the number of reviewers for these critical changes reduces the risk of introducing bugs or instability into our application.

Branching Strategy

Main Branch

The main branch contains the code of the current production version. Only fully tested, stable code should be merged into this branch.

Development Branch

The development branch serves as an integration branch for features and fixes. It contains the code for the next release. Once the code in the development branch is stable and thoroughly tested, it can be merged into the main branch.

Feature Branches

Feature branches are created for new features or bug fixes. They branch off from the development branch and should be merged back into it once the feature is completed or the bug is fixed. Each feature branch should have a clear scope and contain changes related to only one specific feature or bug fix.

Remember, each commit should make clear, concise changes, and the commit message should accurately describe those changes.

Following these standards, we can maintain a high level of code quality, ensure the stability of our application, and foster effective collaboration amongst our team.

Semantic Versioning

Semantic Versioning ((SemVer) is a versioning scheme for software that aims to convey meaning about the underlying changes in a release. It is a widely used standard that helps developers and users understand what kind of changes they can expect when moving from one version of a software package to another.

A semantic version number consists of three parts: MAJOR.MINOR.PATCH, each separated by a dot.

- **MAJOR** version increment indicates that there are incompatible changes in the API, and users may need to change their code to ensure compatibility with the new version.
- **MINOR** version increment indicates that new features have been added in a backwards-compatible manner. Users can benefit from the new features without making any changes to their existing code.
- **PATCH** version increment indicates that backwards-compatible bug fixes have been introduced. These changes are meant to improve the performance, stability, or accuracy of the software without adding any new features.

For example, in version 2.3.4:

'2' is the Major version '3' is the Minor version '4' is the Patch version

By adopting Semantic Versioning, developers can make their upgrade paths clearer and package users can have better expectations about compatibility between different versions of the software.

Commands

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions of alterations there.*

In the Common Gateway, which is a critical component of the architecture, Symfony command-line commands play an integral role. These commands, which are executed in a container and require command access, are powerful tools that can be used for various tasks such as manipulating the database, handling migrations, managing plugins, and many other critical tasks.

However, it's important to note that in normal operations, installations, and implementations of the Common Gateway, direct usage of these commands should be avoided. These commands are generally utilized internally by the gateway's functionalities and are designed to assist in extreme situations where manual intervention is required to troubleshoot or resolve a complex issue.

These commands are primarily designed to interact with the underlying Symfony framework, which powers the Common Gateway. They are a part of Symfony's console component which provides a simple API for creating command-line commands.

Please be aware that these commands, while powerful, should be used with caution. They have the ability to directly manipulate the state of your application and should only be used when necessary and by individuals who have a deep understanding of the Common Gateway and its architecture.

Remember, while these commands exist as a helpful tool in extreme circumstances, in most situations, the Common Gateway is designed to operate and manage its tasks without needing direct command-line intervention. Please always refer to the official documentation and guidelines before running these commands.

comongateway:composer:update

options –bundle {required bundle name} only run the updater for a specific bundle –data {optional bundle name} force the loading of test data –skip-schema {optional bundle name} –skip-script {optional bundle name} –unsafe

Cronjobs

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions of alterations there.*

Cronjobs are a central part of the Common Gateway's event system, enabling scheduling of tasks and automating a myriad of routine procedures. By utilizing the crontab (cron table) file, you can schedule scripts or commands to run at a fixed time, date, or interval. This makes cron jobs a powerful tool for system administration, task automation, data manipulation, and more.

These cron jobs throw events, which may carry an optional data set. The data set can comprise any information relevant to the task - from simple identifiers to complex data structures, depending on the event's nature and the task at hand. This allows for great flexibility, facilitating tasks such as data backup, sending emails, or system maintenance.

The real power of this system comes into play when other components subscribe to these events. Action handlers, or functions that determine how an application responds to a certain event, can subscribe to cron job-generated events. When an event occurs, these action handlers are notified and can then process or interact with the associated data.

For instance, a handler might subscribe to a "BackupComplete" event. When a cron job finishes backing up a database, it throws this event with data about the backup's result. The handler, having subscribed to this event, will then receive this data, enabling it to perform subsequent actions - perhaps logging the backup's success, notifying a system administrator, or initiating another dependent task.

In conclusion, cron jobs in the Common Gateway are a versatile tool, enabling task scheduling and data transmission through events. The true potential is unlocked when other elements, such as action handlers, subscribe to these events, providing a reliable way to automate complex tasks and procedures.

To further explore and experiment with cron jobs, you can use an online crontab editor.

Datalayer

> ***Warning*** *This file is maintained at Conduction's Google Drive Please make any suggestions of alterations there.*

The data layer in the Common Gateway project is an innovative feature designed to act as a cross between an index (akin to Elasticsearch) and a data lake. It normalizes data from diverse sources and enables sophisticated searching through various query languages. Its purpose is to simplify cross-source questioning across databases, APIs, and files such as Excel spreadsheets.

Data Normalization

Our data layer accomplishes this functionality by using schemas as Entity-Attribute-Value (EAV) objects, and normalizing data from different sources within these structures. The advantage of this method is that it provides a uniform view of data regardless of its original source or format, making it easier to search and analyze.

Source of Truth

Despite its powerful capabilities, it's essential to understand that the data layer is not a source of truth. Instead, it serves as a facilitator, helping us search through the underlying sources. It accomplishes this through a mechanism called "smart caching."

Smart Caching

Smart caching works by taking a subscription notification from the source (thereby instantly updating the cache if the source changes) or regularly checking the source. This design ensures that the data layer always provides the most current data available, optimizing accuracy and performance.

There is also the option to bypass the cache entirely and query the source directly in an asynchronous manner. However, this approach can result in a performance penalty due to the lack of caching.

Extending Data Models

The data layer is not just about data normalization and searching. It also allows us to attach extra properties to objects that don't originally have them, effectively extending data models. This feature enables greater flexibility and versatility in how we use and analyze our data.

Overall, the Common Gateway data layer is a crucial component of our architecture, enabling seamless integration and querying across various data sources, while ensuring up-to-date information and the flexibility to extend data models as needed.

Design Decisions

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions of alterations there.*

API First

An API-first approach means that for any given development project, your APIs are treated as "first-class citizens." An API-first approach involves developing APIs that are consistent and reusable, which can be accomplished by using an API description language to establish a contract for how the API is supposed to behave. The specification we use is the OpenAPI Specification. You can view the latest version of this specification (3.0.1) on GitHub.

Documentation

We host technical documentation on Read the Doc's and general user information on GitHub pages, to make the documentation compatible with GitHub we document in markdown (instead of reStructuredText). Documentation is part of the project and contained within the /docs folder.

Common Ground

All applications are developed following the Common Ground standards on how a data exchange system should be: modular and open-source. More information on Common Ground can be found here

Endpoints

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions or alterations there.*

Endpoints are locations (with addresses) where applications can send and receive messages. The generally consist of a domain and path part together forming an URL. For example in case of the `demo.commongateway.nl/api/pets` url, the `/api/pets` would be the path and and `demo.commongateway.nl/` the domain.

The common gateway uses endpoint to allow applications to access it, it splits al endpoints into two categories wich are separated by their first path part.

/admin for endpoints that are part of the gateways internal workings /api for user created endpoints

Index

1. Defining an endpoint

Defining an endpoint

The Common Gateway stores,imports and exports endpoints as JSON mapping objects. Bellow you can find an example endpoint object

Endpoint objects MUST follow the bellow specifications

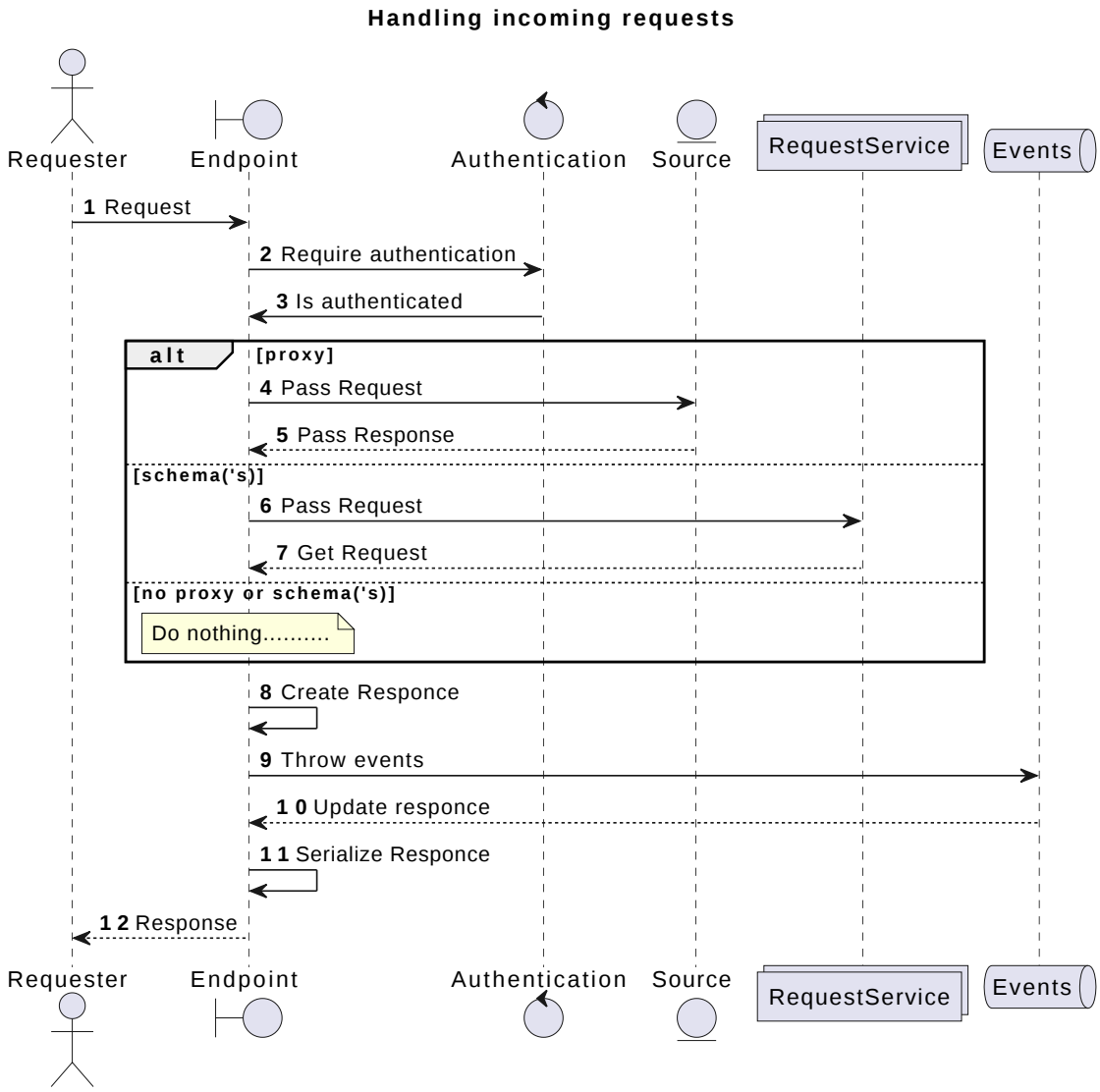
Property	Required	Usage	Allowed Value
title	Yes	User friendly single sentence describing of the endpoint used for identification	string, max 255 characters
description	No	User friendly multi line description of the endpoint used for explaining purpose and workings of the mapping	string, max 2555 characters
$id	No	Used during the import of endpoint to see if a	string, max 255 characters

		endpoint is already present	
$schema	Yes	Tells the common gatway that this object is a endpoint	Always: 'https://docs.commongateway.nl/schemas/Endpoint.schema.j
version	no	Used during the import of endpoint to see if endpoint should be overwritten (updated)	A valid semantic version number
pathRegex	no	The regex used by the Gateway to find the endpoint. For the above example, that would be `^pets`, but the regex could also allow for variable parts like `^pets/?([a-z0-9-]+)?$`. The pathRegex **MUST** be unique within a Gateway installation	
path	no	An array of the items in the path that are separated by `/` in the endpoint. For the above example that would be `['pets','id']`.	

		Path parts **MUST** exist of letters and numbers.	
pathParts	no	An array containing the parts of the path for setting variables for later processes to use later on. Based on their index in the path array and the variable's name that should be created. For the above example, that would be `['1'=>'id']`	
methods	no	Determins the HTTP methods suported by this endpoint	An array of the methods that are allowed on this path, for example ['GET','PUT','POST','PATCH','DELETE']. An endpoint **must** have at least one method
source	no	Turns the endpoint into a proxy for a differend API	ONE external source that is proxied by the endpoint (see Prox
schemas	no	Any schemas provided by the endpoint (see schema's)	Array of schema's
throws	no	Any events thrown by the endpoint (see event-driven)	Array of events thrown when the endpoint is called

Handling incoming traffic

Once an endpoint is called by an external application calls an endpoint(or user, a browser is also an application), the endpoint will handle the following actions in the following order.



As you can see can see there are three basic alternatives

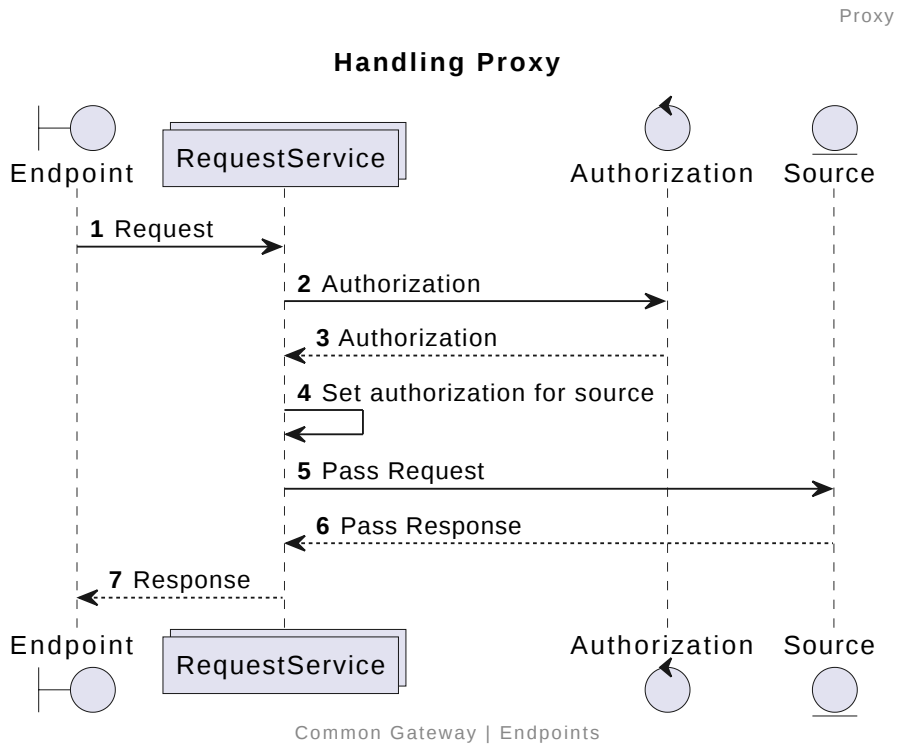
1. An endpoint is linked to a source (becoming a proxy for that source)
2. An endpoint is linked to one or more schema's
3. An endpoint is not linked to sources or schema's

> ***Note***
>
> - *When a proxy (source) is set on an endpoint the schema fase wil be skipped*
> - *Events are always thrown (in all three cases)*
> - *If an endpoint is not linked to a source, schema's AND dosn't contain event it wil always return an error*

Proxy

An endpoint MAY be a proxy for another (external) source. In this case, the endpoint will forward all traffic to the external source (and attach authorization if required by that source). It will also pass along: any headers , query parameters , body , and methods (e.g., GET) sent to the endpoint. Keep in mind that it will only do so if the method used is activeated on the endpoint (in practice, it is quite common not to expose delete through an endpoint)

Suppose the endpoint path contains an endpoint parameter in the path regex e.g. `example` . In that case, it will also forward that message to that specific endpoint on the external source. So `gateway.com/api/petstore/pets` would be forwarded to `petstore.com/pets` .



Keep in mind that a proxy does not transform or translate data, it simply forwards the received request to a source and then returns the response of that source. If you require more functionality (e.g. data transformation or translations) you should setup a schema.

Schema's

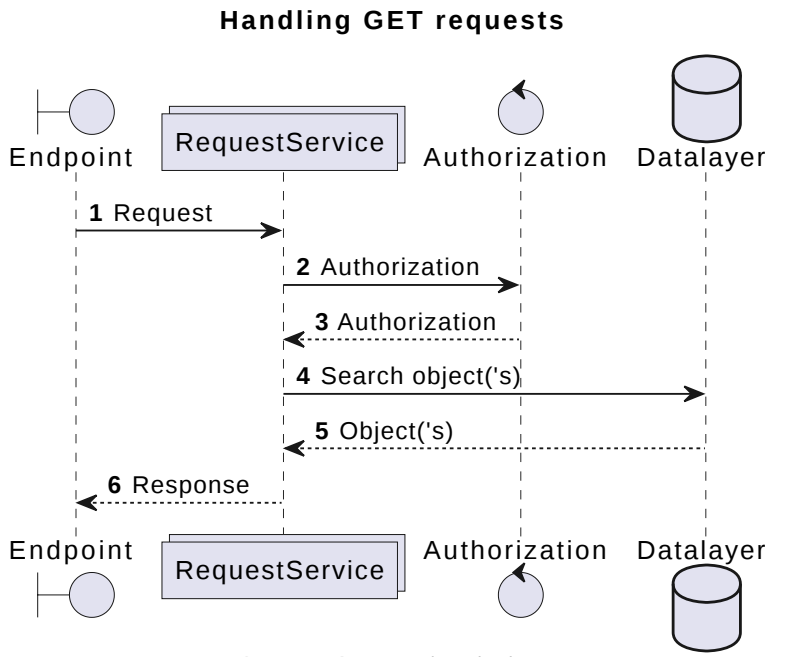
If an endpoint connects is connected to one or more schemas, it will try to handle the traffic based on the requested service.

> ***Note***
>
> - *If an endpoint is hooked to schema('s) it will automatically create an API and appropriate Redoc based on its settings. See API for more information on the API.*
> - *It is possible to hook an endpoint to multiple schemas. When hooked to multiple schemas, the endpoint can still handle POST requests, BUT a POST request must include a valid _self.scheme.id or _self. scheme.ref that refers to the appropriate schema so that the endpoint understands what schema you are trying to create.*

> - *If an endpoint is hooked to more than one entity, it will render search results from all linked entities based on supplied queries.*

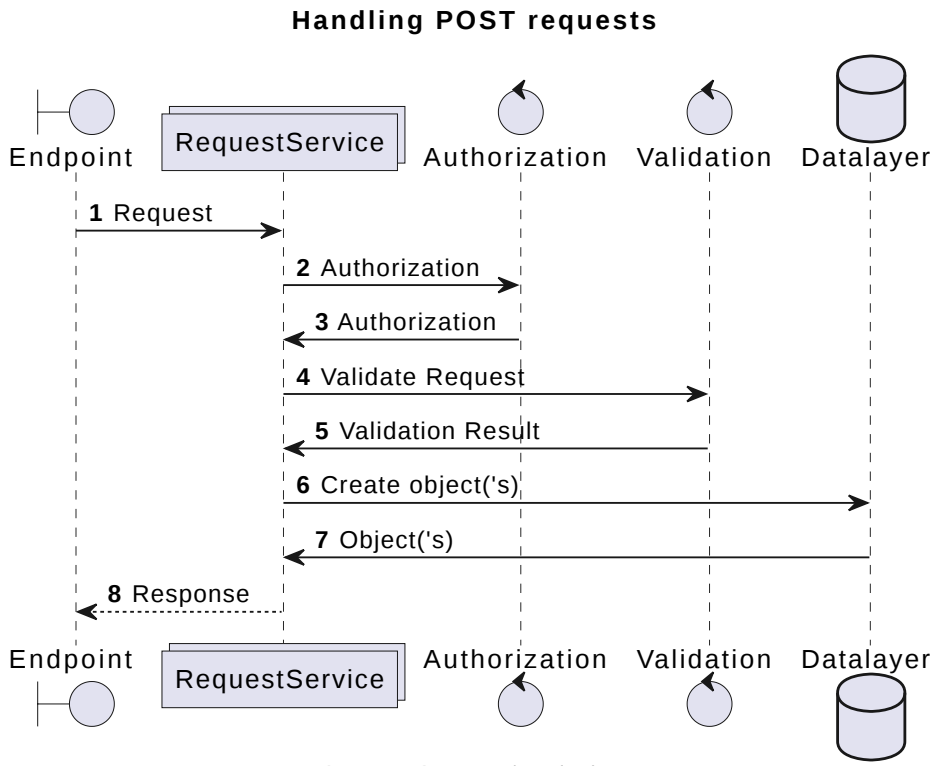
GET

GET Request



Common Gateway | Endpoints

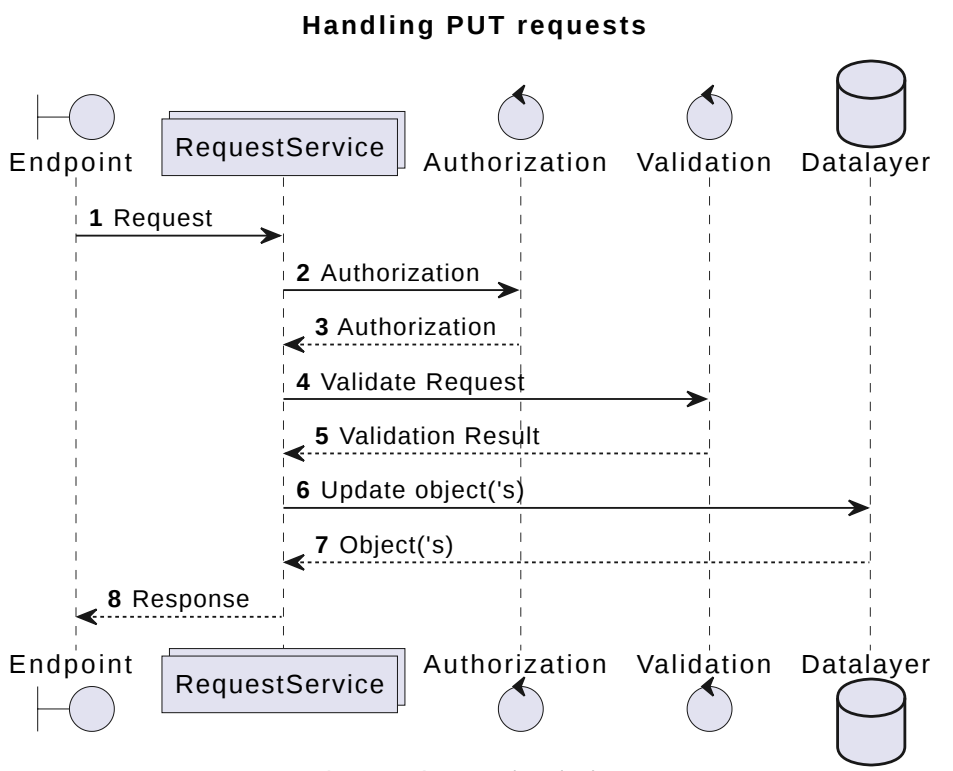
POST



PUT

Put request are handled roughly the same as an POST request, with one exception.

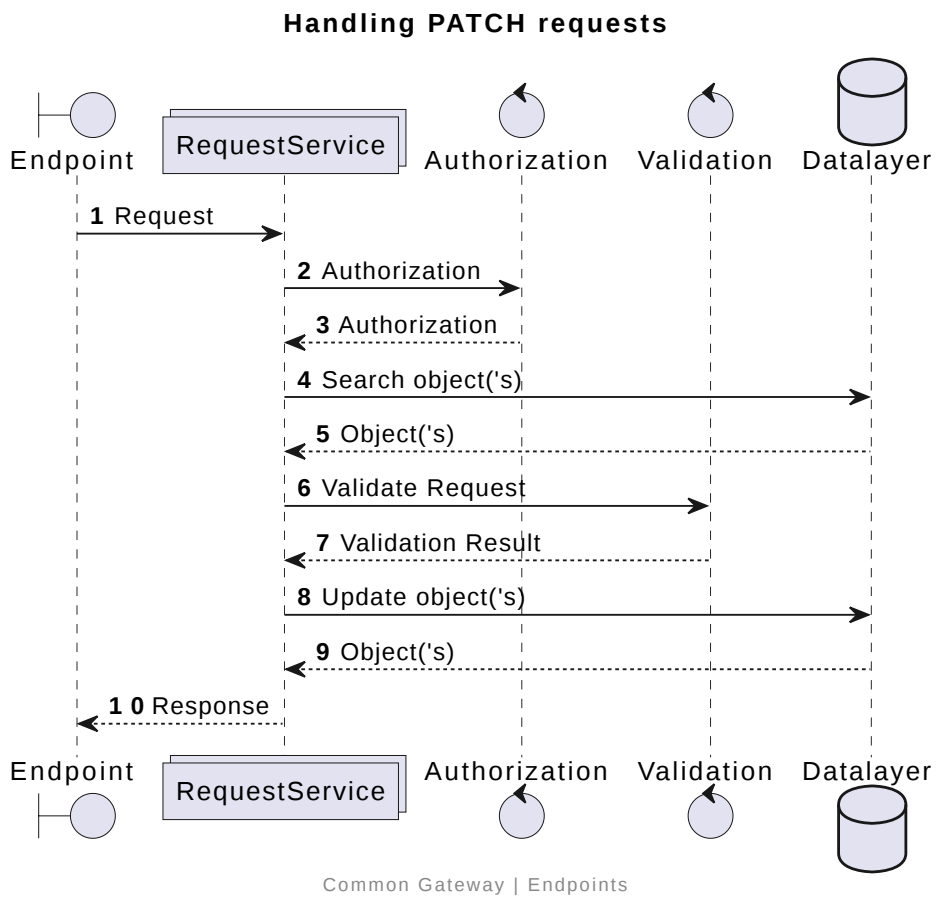
1. On a PUT request an exisiting object wil be entirely replaced by the new object. Values that where present in the original object but are not present in the new object wil be DELETED.



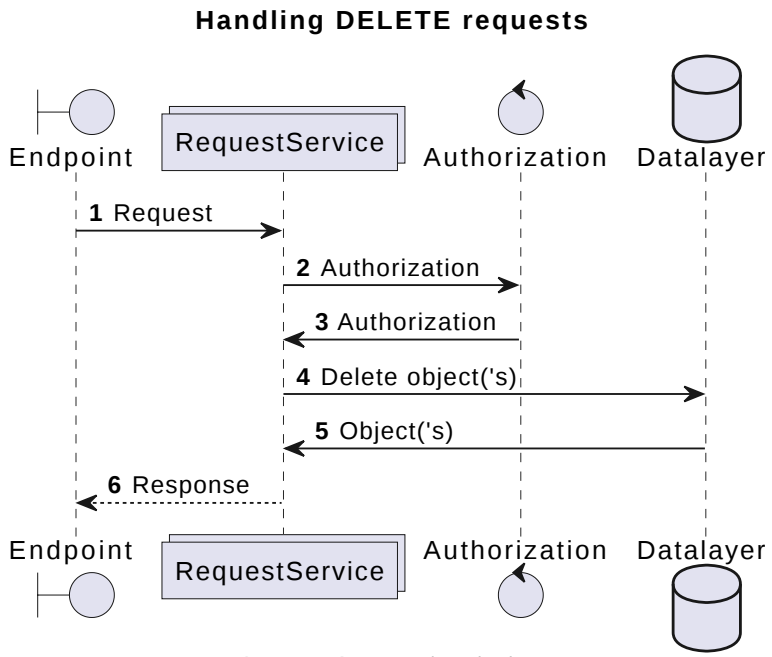
PATCH

Put request are handled roughly the same as an PUT request, with two exception.

1. On a PATCH request an exisiting object wil be updated by the new object. Values that where present in the original object but are not present in the new object wil be KEPT.
2. Validity will of the request will be determend afther merging the original and new object. e.g. a required value dosn't need to be pressent in an patch request if it was already present in the original object. Assuming that the orignal object already had al required values (or else it could note have been created) a patch requests only required value should be it's id (excpetion being that the object definition could have been changed afther the original object was created)

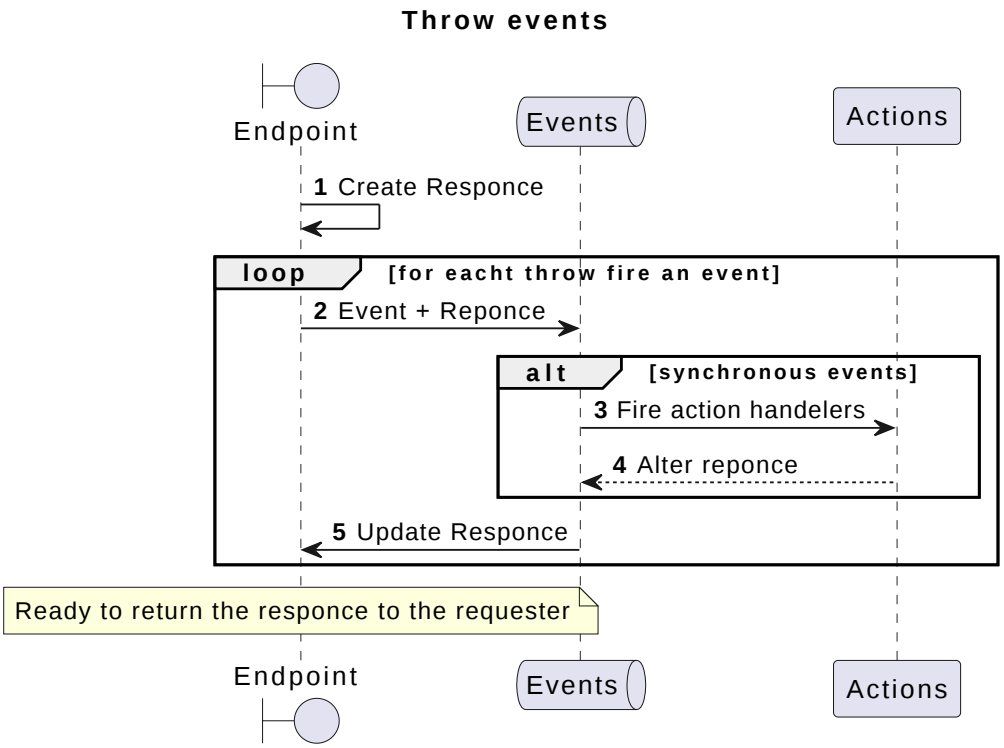

Handling PATCH requests
Endpoint
RequestService
Authorization
Validation
Datalayer
1 Request
2 Authorization
3 Authorization
4 Search object('s)
5 Object('s)
6 Validate Request
7 Validation Result
8 Update object('s)
9 Object('s)
1 0 Response
Endpoint
RequestService
Authorization
Validation
Datalayer

DELETE



Throwing Events

As a final step the endpoint will ALWAYS fire any events that are defined under throws. You can read more about events under events.



When the endpoint throws events, it generates a response in the call cache. After handling all the throws ,are handled it will return the response to the user. The response starts as a `200 OK` “your request has been processed”, but may be altered by any action that subscribed to a thrown event may alter it.

> ***Note***
>
> - *Any action can subscribe to an event thrown by an endpoint, but common examples are proxy en request actions. These fulfill the same functionality as the direct proxy or event link but allow additional configuration, such as mapping.*
> - *It recomended to ALWAYS fire event asynchronysly*

Serialization

Return an error

The endpoint will return an error if no proxy, entities, or throws are defined.

Security

Endpoints MAY be secured by assigning them to user groups. This is done on the basis of methods.

Events

> ***Warning*** *This file is maintained at the Conduction [Google Drive](). Please make any suggestions or alterations there.*

The Common Gateway is based on event driven architecture, meaning that all code and functionality is loosely coupled (see booth architecture and code quality). That means that at no point during execution of business logic a functionality should directly call a different functionality. This might seem complicated (and at time it is) but it provides two important benefits:

- It allows us to divide the work of executing code among several “worker” containers (read more). Providing an extreme performance boost on production environments on heavy load business logic.
- It allows all interested parties to develop plugins for the Common Gateway that directly hook into and extend the core functionality.

Triggers

Event-driven architecture uses events to trigger and communicate between services (some functionality from the codebase). A good example if this is an endpoint. The gateway detects if a user or application approaches an endpoint (e.g., `api/pets`) and sets an event on the stack. Events always consist of an unique trigger of the type `string` In this case `commongateway.endpoint` and an array of data (in this case the request information like method en body). We call this throwing an event. Other good examples of triggers are : cronjobs, Object changes(e.g., CRUD actions)changes in objects (e.g. CRUD actions).

Actions

Actions are preconfigured sets of business logic that “listen” for one or more events to be thrown and then execute code.The ActionHandler contains the executable code.

Actions primarily consists of three things: The events it listens to The action handler that should be used to handle the action Configuration for that action handler

Storing the configuration for the action handler in the actual action means that actionHandlers can be reused. An example would be the mail actionHandler provided by the core bundle. It can be used by actions hooking into the new user event to send an welcome email to new users AND by actions hooking into the logger event to send an email to the gateway admin whenever errors occur.

Chaining actions

Additionally, Actions can throw events themselves. You can build simple flows using this typical pattern (called chaining).. Currently, the gateway isn’t a full-blown BPMN engine and should not be used that way. It is however possible to integrate the BPMN engine into gateway flows using custom plugins (we are still looking for a sponsor for a Camunda or Flowable plugin).

Event list

Events can't be pre-defined as they come into existence once a service throws them. You can define your own events through either: the gateway UI admin API. Events should logically be namespaces and use dot notation. The namespace commongateway is reserved for core functionality

The gateway subscribes to the following events by default.

Name	When	Data
commongateway.object.pre.create	Before an object is created in the database	["object"=>{array representation of object},"entity"=>{uuid of the objects entity}]
commongateway.object.post.create	After an object is created in the database	["object"=>{array representation of object},"entity"=>{uuid of the objects entity}]

commongateway.object.post.read	After an object is read from the database	["object"=>{array representation of object},"entity"=>{uuid of the objects entity}]
commongateway.object.pre.update	Before an object is updated in the database	["object"=>{array representation of object},"entity"=>{uuid of the objects entity}]
commongateway.object.post.update	After an object is updated in the database	["object"=>{array representation of object},"entity"=>{uuid of the objects entity}]
commongateway.object.pre.delete	Before an object is deleted in the database	["object"=>{array representation of object},"entity"=>{uuid of the objects entity}]
commongateway.object.post.delete	After an object is deleted in the database	[]
commongateway.object.pre.flush	Before the work of the entity manager is transferred to the database	[]
commongateway.object.post.flush	After the work of the entity manager is transferred to the database	[]
commongateway.installer.pre.upgrade	Before the installer upgrades	[]
commongateway.installer.post.upgrade	After the installer upgrades	[]
commongateway.initilizer.pre.upgrade	Before the initializer upgrades	[]
commongateway.initilizer.post.upgrade	After the initializer upgrades	[]
commongateway.plugin.pre.install	Before the plugin is installed	[]
commongateway.plugin.post.install	After the plugin is installed	[]
commongateway.plugin.pre.upgrade	Before the plugin is upgraded	[]
commongateway.plugin.post.upgrade	After the plugin is upgraded	[]
commongateway.plugin.pre.remove	Before the plugin is removed	[]
commongateway.plugin.post.remove	After the plugin is removed	[]

Design your own triggers, events, actions and action handlers

When adding your customizations to the Common Gateway, you should always follow the separation of concerns:

keep flows small (don’t try to do too much in one flow) keep functionality (actionHandlers) minimal

For complex scenarios, consider using several chained actionHandlers. When adding your own flavor to the common gateway you should always follow separation of concerns.

In other words keep flows small, don’t try to do too much from a single flow and keep your actionHandles minimal. If things get more complex consider using several chained action handlers.

*__ALWAYS__ use the [vendor].[plugin].[action].[sub action] naming pattern for your events to prevent conflicts them conflicting with other events. When adding events on an installation or app basis use: either the app (e.g app..[action].[sub action]) or cron (e.g. cron.[action].[sub action])

namespace patterns to keep your events recognisable.

Features

Welcome to the feature page for the Common Gateway, a multi-faceted platform designed with flexibility and interoperability in mind. The Common Gateway caters to four main use cases, each enhancing the other to provide a comprehensive, unified solution for diverse data management needs. Here's a brief overview:

Usse cages

1. API Gateway

The Common Gateway can function as an API Gateway, acting as a single entry point for multiple APIs. This streamlines the management of APIs, providing consistent routing, security, and other necessary features. This simplifies client-side interactions and consolidates all your API requirements under one roof.

2. Web Gateway

Beyond forwarding APIs, the Common Gateway can serve as a robust Web Gateway. It provides API support for applications while also handling user onboarding, management, and authentication. This dual functionality ensures seamless user experience and secure data access, enhancing application reliability and performance.

3. Integration Platform

The Common Gateway shines as an Integration Platform, harmonizing multiple data sources and transforming non-API sources, such as Excel files, into APIs. This capability enables easy data integration, simplifying the creation of a unified view of data from various sources. It paves the way for more efficient data processing and analysis, fostering data-driven decision-making.

4. Federated Network Provider

Finally, the Common Gateway facilitates a federated network, enabling cross-organizational querying of data sources. This feature allows different organizations to share access to specific data while maintaining control over their own systems. It boosts collaboration, data sharing, and multi-organizational integration without compromising system autonomy.

By combining these use cases, the Common Gateway provides a versatile solution for managing, integrating, and utilizing data across various sources and platforms. It's the perfect tool to elevate your data strategy and

drive your business towards a more connected, data-informed future. Stay tuned to explore each of these features in depth.

Functionality

1. Authentication

Authentication is a crucial aspect of securing the gateway, validating the identity of clients such as users, devices, or other applications before granting access to system resources. There are numerous methods for applications to authenticate themselves, including application keys, JWT tokens, ZGW JWT tokens, two-way SSL, IP and domain whitelisting. However, each method carries its own use cases and security considerations. For user authentication, the system can either utilize an Integrated Identity Provider (IdP), which manages identity information within a federation, or an External Identity Provider, a third-party service that allows users to authenticate with a single set of externally stored credentials. The choice of method depends on the specific requirements and contexts of use.

2. Authorization

Authorization in the Common Gateway project is managed based on Role-Based Access Control (RBAC). Users and applications are assigned roles, or "groups", each associated with a set of permissions, or "scopes". These groups can inherit scopes from other groups, creating a hierarchical organization of scopes. Scopes are defined based on the CRUD (Create, Read, Update, Delete) operations and can be applied to entire system aspects or specific objects. The system also distinguishes between the roles of an 'Owner' and a 'Creator', each having specific rights and limitations. Multitenancy is a key concept in the Common Gateway project, allowing multiple independent instances of users and applications to operate within the same environment while maintaining secure access to their respective objects. This is implemented at the organization level, and users and applications can only interact with objects that belong to the same organization. Multitenancy is maintained through either a single database setup or a multiple database setup.

3. Datalayer

The data layer in the Common Gateway project acts as a hybrid of an index and a data lake, normalizing data from various sources and facilitating sophisticated searches across databases, APIs, and files. It uses schemas as Entity-Attribute-Value (EAV) objects to provide a uniform view of data, regardless of its original source or format. However, it is not a source of truth but serves as a facilitator through a mechanism called "smart caching," which ensures the most current data is provided. The data layer also allows for the extension of data models by attaching additional properties to objects, enabling greater flexibility and versatility in data use and analysis.

4. Logging

The Common Gateway project uses Symfony's Monolog bundle for logging, offering several channels for logs and multiple error levels following the RFC 54240 standard. Channels categorize logs, and plugins can add their own channels. Logs can also contain additional data like session ID, user, application, and more, provided through a dataprocessor. Plugins are required to add their identifier to logs for traceability. Logs are stored in the gateway's log directory, printed to standard output, and saved in a MongoDB database for easy searchability. Log retrieval is possible through the admin/logs endpoint. Plugins can create additional logs, and they're advised to use existing channels or the plugin channel, or, if necessary, create their own channel. Logs should be made from services following separation of concern, and should include the plugin package name for findability and accessibility.

5. Mappings

The Mappings feature in the CommonGateway/CoreBundle project supports the process of transforming the structure of an object when the source data doesn't match the desired data model. This transformation is

accomplished by a series of mapping rules in a "To <- From" style. In simple terms, mapping changes the position of a value within an object1.

6. Notifications

7. Plugins

The Common Gateway's plugin system provides a method of keeping client-specific code separated from the core functionality. This structure is based on the Symfony bundle system, making the Common Gateway easily extensible. Essentially, all Common Gateway plugins are Symfony bundles, and vice versa, allowing for a configuration set that can extend a base Gateway's functionality1.

8. Schema's

Schemas are central to the Common Gateway's data layer. They define and model objects, setting the conditions for these objects. Each object in the gateway is associated with a single schema. These schemas follow the JSON schema standard, making them interchangeable with OAS3 schemas. Schemas are akin to "tables" in traditional databases as they store data in a structured manner. However, unlike tables, data is stored as objects, each akin to a dataset or row in a traditional table, but capable of multidimensional storage, such as containing arrays or other objects. This object-oriented approach provides a much more flexible way of serving data. Furthermore, schemas define the properties of objects and also set conditional validations for the value of each property1.

9. Security

The Common Gateway integrates security into the core of its development process. As part of its Continuous Integration and Continuous Deployment (CI/CD) pipeline, the platform employs automated penetration testing and scanning. This approach allows potential security vulnerabilities to be identified and addressed during the early stages of development, rather than later in the production phase1.

10. Sources

The "Sources" feature in Common Gateway represents the locations where the Gateway obtains its data. These sources are typically other APIs, but the gateway can also connect to file servers through (S)FTP, directly connect to databases (like MongoDB, PostgreSQL, MySQL, Oracle, and MSSQL), blockchain solutions, or networks of trust such as NLX/FCS. The source object provides information about the source, including its status, transaction logs, and the current connection settings. Sources can be found through the sources menu item on the left side of the Admin UI or the admin/sources endpoint on the gateway API1.

11. Synchronization

The data synchronization process in Common Gateway is an essential component of its data layer, ensuring data consistency between the data layer and an external source. The process starts with determining the current state of data where three main scenarios exist: the object exists on the Gateway but not on the source, the object exists on the source but not on the Gateway, and the object exists on both the source and Gateway. Appropriate actions are taken depending on the scenario, such as deleting, adding, or updating the object. In situations where it's unclear which version is newer, a source of truth is determined, typically the source, but it can be configured to be the Gateway. The next step is to create a synchronization object that describes the relationship between the two objects (Data Layer and Source), containing details like the IDs on both ends, the dates of changes, and a hash of the source object. This object is essential for detecting changes in the source object and ensuring data consistency across the Gateway and the source, enhancing the reliability and integrity of the system1.

12. Twig

12. Federialization

Federalization

> ***Warning*** *This file is maintained at Conduction's Google Drive Please make any suggestions of alterations there.*

Installation

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions for alterations there.*

We differ in the installation of the gateway between local en server installations, keep in mind that local installations are meant for development, testing en demo purposes and are (by their nature) not suited for production environments. When installing the gateway for production purposes ALWAYS follow the steps as set out under the server installation manual.

Local installation

For our local environment, we use docker desktop. This allows us to spin up virtual machines that mimic production servers easily. In other words, it helps us ensure that the code we test/develop locally will also work online. The same can also be said for configurations.

To spin up the gateway for local use, you will need both Docker Desktop and a git client (we like to use git kraken but any other will suffice, you can also install git on your local machine)

Steps

1. Install docker desktop, git client and a browser like google chrome.
2. Create a folder where you want to install the gateway, for example: documents/gateway
3. Open a command line interface e.g. windows key + cmd + enter
4. Navigate to the folder you just created with our examples. This would be: `$ cd documents/gateway`
5. Git clone the Common Gateway repository to a folder on your machine (if you like to use the command line interface of git that's `git clone https://github.com/ConductionNL/commonground-gateway.git`
6. (optional) Check out a specific version of the gateway e.g. `git checkout feature/oc-ui`
7. Change the directory into the gateway folder (The folder where you also find the docker-compose.yml file) for example: `cd commonground-gateway`
8. Startup the gateway through `$ docker compose up` .
9. You should now see the gateway initiating the virtual machines it needs on your command line tool. (this might take some time on the first run, you will see the text 'Ready to handle connection' when it is ready to connect)
10. Additionally, you should now see the containers come up in your docker desktop tool (that you can use from here on)
11. When it is done you can find the Gateway API in your browser under localhost, the Gateway UI under localhost:8000, and any additional web apps that are part of your ecosystem under localhost:81.

    > ***Note****: Read more about command line tools here and how to navigate*

Troubleshooting

If during the steps above you run into any problems the following tips might help:

- If during step 9 the text 'Ready to handle connection' does not appear (keep in mind, this might take a while!) and your docker desktop shows that the php-1 container is not in status running you could be running into an error. If this is the case check your command line tool for any error messages.

- For more general troubleshooting (relevant for local and server installation) please take a look at troubleshooting

Server Installation

There are three main routes to install the Common gateway, but we advise using the helm route for the Kubernetes environment. However, if you want, you can install the gateway on a Linux machine or use a docker-compose installation.

Haven / Kubernetes

The Common Gateway is a Common Ground application built from separate components. To make these components optional, they are housed in separate Kubernetes containers. This means that a total installation of The Common Gateway requires several Containers. You can find which containers these are under architecture.

Installation through Helm charts (recommended method)

Before you continue, make sure you use Helm V3 for all the following steps.

And before installing the gateway through Helm charts, make sure you configure cert-manager on your cluster. For more information please visit the cert-manager documentation.

Also, make sure to also install the nfs provisioner, see: ArtifactHub. When installing nfs make sure you install with persistence enabled. And make sure to configure a persistence size higher than 1Gi (8Gi or even higher is recommended). The persistence size must be higher than the size of the gateway vendor Persistent Volume Claim(s combined) on your cluster.

Helm charts for the Common Gateway are also provided through ArtifactHub. Remember that these Helm files only install the gateway and not any attached applications.

You can add the predefined repository via cli:

```
`$ helm repo add helm repo add commonground-gateway-frontend
https://raw.githubusercontent.com/ConductionNL/commonground-gateway-frontend/master/helm
```

And installed with the chart:

```
`$ helm install my-commonground-gateway commonground-gateway-frontend/commonground-gateway
--version 0.1.5
```

More information about installing via Helm can be found on the Helm Documentation. Further information about installation options can be found on ArtifactHub.

> *Note: With Helm, the difficulty often lies in finding all possible configuration options. To facilitate this, we have included all options in a so-called values file, which you can find here. One very common value used when installing the gateway through helm is the value --set global.domain={{your domain here}}.*

Installation through docker compose

The gateway repository contains a docker compose, and a .env file containing all setting options. These are the same files that are used for the local development environment. However when using this route to install the gateway for production you **MUST** set the `APP_ENV` variable to 'PROD` (enabling caching and security features) and you must change all passwords (conveniently labeled *!ChangeMe!*) **NEVER** run your database from docker compose, docker compose is non-persistent and you will lose your data. Always use a separately managed database solution.

Installation through composer (Linux / Lamp)

Before starting a Linux installation make sure you have a basic LAMP setup, you can read more about that here. Keep in mind that the gateway also has the following requirements that need to be met before installation.

Linux extensions Composer PHP extensions A message queue in the form of RabbitMQ. A caching mechanism in the form of Redis

After making sure you meet the requirement you can install the gateway through the following steps.

In your Linux environment create a folder for the gateway (`md /var/www/gateway`) and navigate to that folder (`cd /var/www/gateway`). Then run either `$ composer require common-gateway/core-bundle` or the composer require command for a specific plugin.

Troubleshooting

During installation, it is possible that you run into problems, below you will find common problems and how to deal with them. If you are still running into problems after reading this or if you have any constructive criticism please seek contact with our development team (info@conduction.nl).

> *Note: when troubleshooting you will, in most cases, need to run some commands. Unless stated otherwise, these commands should always be executed in the php container.*

A very common way to check why the Common Ground Gateway is not functioning as expected is by running the `bin/console doctrine:schema:validate` command. This command validates the mappings of the Commonground Gateway database, but will often return insightful error messages when running into other problems.

You have requested a non-existent service

This is an error message you will only get when you are trying to install & initialize plugins on the CommonGround Gateway. If you get the error message "You have requested a non-existent service" then this indicates in most cases that you are missing a specific plugin, if you have already installed this plugin (or tried to) however, your config/bundles.php file is most likely not up-to-date. This bundles.php file should contain all installed bundles. If you, for example, get a message that `OpenCatalogi\OpenCatalogiBundle\ActionHandler\ComponentenCatalogusApplicationToGatewayHandler` does not exist, you most likely need to add `OpenCatalogi\OpenCatalogiBundle` to the bundles.php file like this: `OpenCatalogi\OpenCatalogiBundle\OpenCatalogiBundle::class => ['all' => true],` locally you can just edit this file, with a server installation you might want to use something like vi editor for this.

Adding the gateway to your existing Symfony project (Beta)

The gateway is a Symfony bundle and can also be added directly to an existing Symfony project through composer. The basic composer command is `composer require commongateway/corebundle` and you can read more about the installation process on packagist.

Applications

There are several applications that make use of Common Gateway installation as a backend, best known are huwelijksplanner (HP), Klantinteractie Service Systeem(KISS) en Open Catalogi (OC).

By design front ends are run as separate components or containers (see Common Ground layer architecture). That means that any frontend application using the gateway as a backend for frontend (BFF) should be installed separately.

Production

The gateway is designed to operate differently in a production, than in a development environment. The main difference is the amount of caching and some security settings.

Cronjobs and the cronrunner

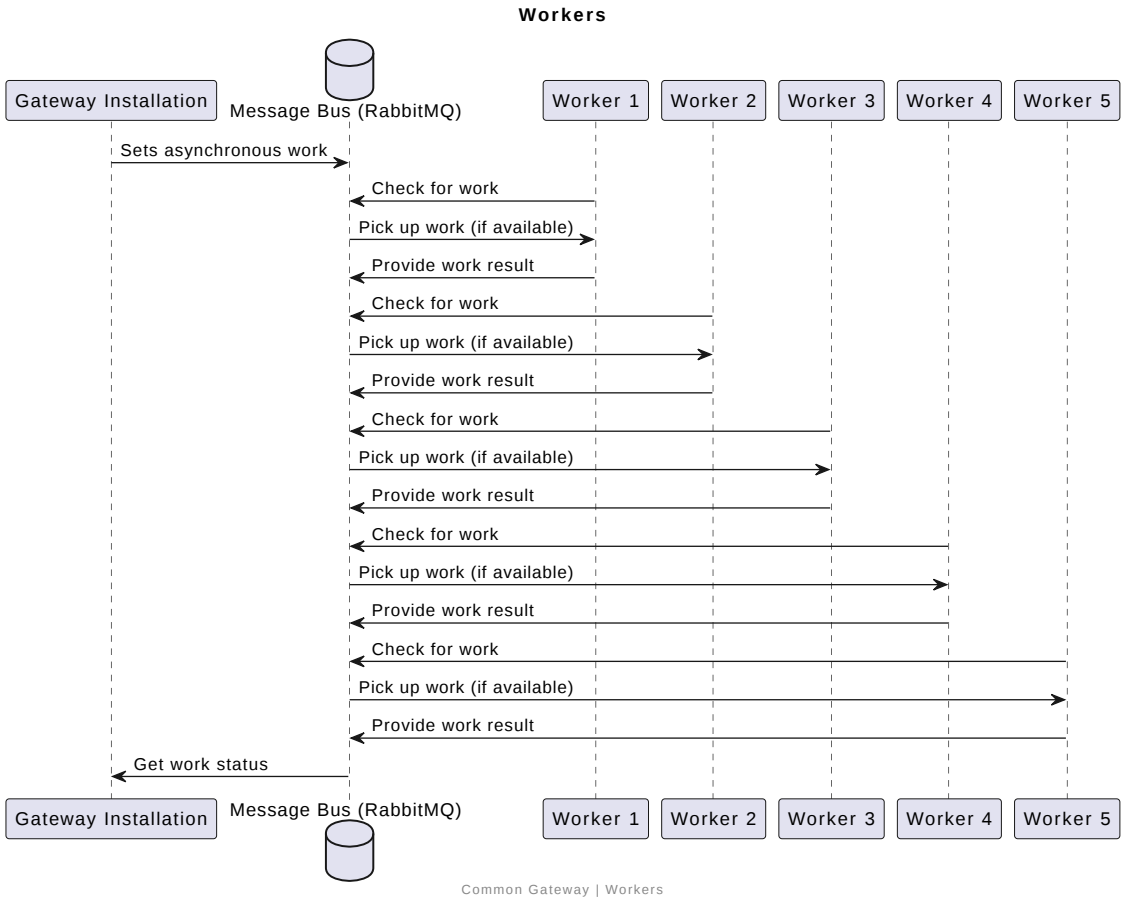
The gateway uses cronjobs to fire repeating events (like synchronizations) at certain intervals. Users can set them up and maintain them through the admin UI. However, cronjobs themselves are fired through a cron runner, meaning that there is a script running that checks every x minutes (5 by default) whether there are cronjobs that need to be fired. That means that the execution of cronjob is limited by the rate set in the cronrunner .e.g if the cronrunner runs every 5 minutes it's impossible to run cronjobs every 2 minutes.

For docker compose and helm installation the cron runner is based on the Linux crontab demon and included in the installation scripts. If you are installing the gateway manually you will need to set up your own crontab to fire every x minutes.

For your crontab, you need to execute the `bin/console cronjob:command` cli command in the folder where you installed the Common Gateway. e.g. `*/5 * * * * /srv/api bin/console cronjob:command`. If you need help defining your crontab we advise crontab.guru.

Workers

The gateway uses workers to asynchronously handle workload, the concept is quite simple the gateway installation sets asynchronous works on a massage queue (RabbitMQ) other gateway installations then look into the message queue to see if there is any work that they can pick up. In the default helm installation, we use 5 gateway containers for this. The amount of workers is however configurable through the `change.me` parameter.



If you are installing the gateway on a Linux setup you will need to manually install workers (preferably on other machines than your main gateway installation) and put them into worker mode by running the command 'bin/console messenger:consume async', and point them to the RabbitMQ message queue. The RabbitMQ location is defined in the file api/config/messenger.yaml in the variable parameters.env(MESSENGER_TRANSPORT_DSN).

Setting up plugins

After you install the Commonground Gateway, you can use the Commonground Gateway as it is, or take a look at plugins. More about plugins can be found here

#Logging

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions or alterations there.*

The gateway uses symfony's monolog bundle for logging, and provides several channels for logs.

Channels

By default, the following channels are provided by the gateway, but plugins might add their own channels by including a `monolog.yaml` in their configuration. Channels represent the part of the gateway that has created the log and are used to separate logs by category. endpoint request schema cronjob action object synchronization plugin composer installation mapping call

Error levels

The gateway uses the following error levels conform [RFC 54240](https://www.rfc-editor.org/rfc/rfc54240 DEBUG: Detailed debugging information. INFO: Handles normal events. Example: SQL logs NOTICE: Handles normal events, but with more important events WARNING: Warning status, where you should take an action before it will become an error. ERROR: Error status, where something is wrong and needs your immediate action CRITICAL: Critical status. Example: System component is not available ALERT: Immediate action should be exercised. This should trigger some alerts and wake you up during night time. EMERGENCY: It is used when the system is unusable.

Data

The gateway uses a dataprocessor to add additional data to a log. The following data is automatically added (if available through the session object). session: The session id user: The user that made the request application: The application that made the request organization: The organization of that application source: The source that was accessed by the process endpoint: The id of the endpoint where the call landed schema: The schema that was used during the session action: The currently running action object event: synchronization cronjob: The cronjob that triggered the call command: The command options and input that triggered the call, based on

The plugin identifier is not automatically added to logs, but plugins are required to add that value themself so that logs are easily traced back to an specific plugin

Normally speaking a call should be started by either a cronjob,command or endpoint,

Storage

By default, the gateway stores logs to `{channel name}.log` files in the log directory of the gateway, prints logs out to STD out and saves them to the Mongo database. When storing the logs to mongoDB all logs are stored to a single collection to make them easily searchable.

Retrieving logs

Logs can be retrieved through the `admin/logs` endpoint that provides them from the mongo database.

Creating logs from your plugin

You might feel the need to create additional logging for your plugin because it does some extremely important stuff that just *might* go wrong and then needs to be fixed. There are basically 3 ways of going about this

1 - There is an appropriate channel (for example actions for an action handler that you have built). Use that! Conforming yourself to existing logging means that your logs will be automatically available through the gateways tool like the admin ui and grafana. 2 - Use the plugin channel, that is what it is for, anny undefined logs can be stored there. 3 - Create your own channel by expanding monolog.yaml from your plugin, this is almost never the preferred option. It gives you great flexibility in how to use the logs, but it leaves the context of the gateway making your logs unpredictable for those who need them the most, your users.

You **SHOULD** always log from services, not just because it is easter, but because following separation of concern your business logic(and therefore the stuff you want to log) should be contained there anyway. It does however also make it very easy to add logging to your service trough autowiring. When adding a logger to your action handlers you can for example include the action channel like:

```
…
use Psr\Log\LoggerInterface
```

```
…

public function __construct(LoggerInterface $ActionLogger)
{
    $this->logger = $ActionLogger;
}
```

Or when using the generic plugin channel

```
…
use Psr\Log\LoggerInterface
…

public function __construct(LoggerInterface $PluginLogger)
{
    $this->logger = $PluginLogger;
}
```

After this creating a log is rather easy, just use `$this->logger->{level}({message}) e.g.

```
$this->logger->info('I just got the logger');
$this->logger->error('An error occurred');
```

Keep in mind that if you want your logs to be findable and accessible through the admin ui you should also include your plugin package name as an extra value e.g.

```
$this->logger->info('I just got the logger',[“plugin”=>”common-gateway/pet-store-
bundle”]);
```

note: It is actually possible to register your logs under the wrong plugin, don’t. First of all it will confuse your users and be traceable through the process id. Secondly it will be flagged as a code injection attract and reported.

Mappings

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions or alterations there.*

The mapping service supports the process of changing the structure of an object. It's used to transform data when the source doesn't match the desired data model. Mapping is done by a series of mapping rules in a To <- From style. In simple mapping, the position of a value within an object is changed.

Index

1. Defining a mapping
2. Usage
3. Advanced (Twig) mapping and/or adding keys
4. Pass Through and/or drop keys
5. Working with conditional data
6. Sub mappings
7. Casting (Forcing) the type/format of values
8. Special Casting (Forcing) change of values
9. Translating values

10. Renaming Keys
11. Order of mapping
12. What if I can't map?

Defining a mapping

The Common Gateway stores, imports, and exports mappings as JSON mapping objects. Below you can find an example mapping object

```
{
  "title": "MyMapping",
  "description": "MyMapping",
  "$id":
"https://development.zaaksysteem.nl/mapping/xxllnc.XxllncCaseToZGWZaak.mapping.json",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "version": "0.0.1",
  "passTrough": false,
  "mapping": {
    "{{to_key}}": "{{from_key}}"
  },
  "unset": ["{{from_key}}"],
  "cast": {
    "{{to_key}}": ["{{type}}"]
  }
}
```

Mapping objects MUST follow the below specifications

Property	Required	Usage	Allowed Value
title	Yes	User-friendly single sentence describing of the mappings used for identification	string, max 255 characters
description	No	User-friendly multi-line description of the mapping used for explaining the purpose and workings of the mapping	string, max 2555 characters
$id	No	Used during the import of mappings to see if a	string, max 255 characters

		mapping is already present	
$schema	Yes	Tells the common gateway that this object is a mapping	Always: 'https://docs.commongateway.nl/schemas/Mapping.schema.j
version	no	Used during the import of mappings to see if mapping should be overwritten (updated)	A valid semantic version number
passTrough	no	Determines whether to copy the old object to the new object	A boolean, default to false
mapping	no	Moves property positions in an object	An array where the key is the new property location(in dot notation) and the value the current property location (in dot notation)
unset	no	Unset unused properties	A valid JSON object, read more about using unset
cast	no	Casts properties to a specific type	A valid JSON object, read more about using cast

Usage

Okay, let's take a look at the most commonly used example API (petstore) and a basic original object.

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "status": "available"
}
```

Now let's say we want to move the status into a new object that has a sub-object called metadata, like this:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
```

```
  "metadata":{
    "status": "available"
  }
}
```

Then we need to create a mapping that copies the property to a new location through mapping, like this:

```
{
  "title": "A simple mapping for animals",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "mapping": {
    "id": "id",
    "name": "name",
    "metadata.status": "status"
  }
}
```

So what happened under the hood? How is de status moved? Let's take a look at the first mapping set

```
{
  "mapping": {
    "id": "id",
    "name": "name",
    "metadata.status": "status"
  }
}
```

Rules are carried out as a `To <- From` pair. In this case, the `metadata.status` key has a `status` value. When interpreting what the description is, the mapping service has two options:

- The value is either a dot notation array pointing to another position in the object (see dot notation). If so, then the value of that position is copied to the new position. (Under the hood the gateway uses PHP dot notation to achieve this result)
- The value is not a dot notation array to another position in the object (see dot notation), then the value is rendered as a twig template.

> ***Note***
>
> - *The key is ALWAYS treated as a dot notation telling the service where to move the properties content to.*
> - *Mapping object MUST have a title and $schema definition and SHOULD have a description.*
> - *It is not necessary to declare every step of the array (e.g. metadata, metadata.status, metadata.status.name) just declaring the property where you want it will create the in-between array keys*

Keep in mind that dot notations have no maximum depth, so an original object like:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "status": "available"
}
```

Could be mapped like this:

```
{
  "title": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "mapping": {
    "id": "id",
    "name": "name",
    "metadata.status.name": "status"
  }
}
```

To a new object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "metadata":{
    "status":{
      "name": "available"
    }
  }
}
```

> ***Note***
>
> - *Using dot notation to move values around within an object will NOT cause the value to change or be converted. In other words, you can move an entire array or sub-object around by simply moving the property that it is in. Also, booleans will remain booleans, integers remain integers, etc.*
> - *In the case that a key has a dot in it, and you don't want it to trigger the array pointing with dot notation you can use the ASCII code for a dot instead. Example: "location.first.name" If you want first.name to be a string (just to show what I mean: "location.'first.name'") it is possible to do this: "location.first.name". For more options like this, see: https://www.freeformatter.com/html-entities.html.*

Advanced (Twig) mapping and/or adding keys

Another means of mapping is Twig mapping. Let's look at a more complex mapping example to transform or map out data. The pet store decided that we would like to assign pets to an aisle, there are three aisles (green, blue, and red) and every pet needs to be assigned randomly. That means that we need business logic in our mapping. fortunately we can use twig logic in our mapping by placing it in `{{}}` braces. that means that we can do this in our mapping

```
{
  "name": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "mapping": {
    "id": "id",
    "name": "name",
    "status": "status",
    "aisle": "{{ random([green, blue , red]) }}"
  }
}
```

To turn this original object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "status": "available"
}
```

Into this new object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "status": "available",
  "aisle": "red"
}
```

As you might have noticed we have now added a key that wasn't present in the old object. That is because the mappings simply copy values into the new object. These values MAY be created on the fly through the twig extension.

> ***Note***
>
> - *Both dot-notation and twig-based mapping are valid to move value's around in an object. BUT Dot-notation is preferred performance-wise.*
> - *It is possible to add keys by just declaring them*

Pass Through and/or drop keys

In the above examples, we are mapping a lot of properties into our new object that stays in the same location as the where in hour old object. e.g. `id`, `name`, `status`. You can spot these in our mapping:

```
{
  "name": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "mapping": {
    "id": "id",
    "name": "name",
    "status": "status"
  }
}
```

If we have large objects this might be a lot of work (we would need to map EVERY value). This is where `passTrough` comes to our rescue. When setting `passTrough` to `true` in our mapping all the data from the original object is copied to our new objects (passed through the mapper). So if we want our object to stay exactly the same we can simply do the following mapping.

```
{
  "name": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": true
}
```

Now that's just going to give us exactly the same object, so let's add a simple bit of mapping. And we should see something interesting happening.

```
{
  "title": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": true,
  "mapping": {
    "metadata.status.name": "status"
  }
}
```

Will turn this original object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "status": "available"
}
```

Into this new object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "status": "available",
  "metadata":{
    "status":{
      "name": "available"
    }
  }
}
```

Okay, so we now have a double `status` that is because the mapper always copies a value from the old key position to the new key position. So if we are using passTrough we will copy that value twice (once through the mapper and once through passthrough). To solve this we will need to manually unset the undesired key. Which we can do with a mapping like:

```
{
  "title": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": true,
  "mapping": {
    "metadata.status.name": "status"
  },
  "unset":["status"]
}
```

Which will turn this original object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
```

```
  "status": "available"
}
```

Into this new object

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "metadata":{
    "status":{
      "name": "available"
    }
  }
}
```

> ***Note***
>
> - *Using passthrough represents a security risk. All values make it to the new object, so it should only be used on trusted or internal objects*
> - *passthrough is applied BEFORE mapping, so a mapping can be used to 'overwrite' values that were passed through*
> - *Normally when using passthrough we would like to clean up the result because we tend to end up with double data.*
> - *Dropping keys is always the second last action performed in the mapping process (before casting).*
> - *Unset should contain an `array` of keys, keys are defined in dot notation. So its possible to remove properties from any place within an object.*

Working with conditional data

Twig natively supports many logical operators, but a few of those are exceptionally handy when dealing with mappings. For example, concatenating strings like {{ 'string 1' ~ 'string 2' }} which can be used as the source data inside the mapping

```
{
  "title": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": true,
  "mapping": {
    "metadata.color": "{{ \"The color is \" ~ color }}"
  },
  "unset": ["color"]
}
```

The same is achieved with string interpolation via a mapping of:

```
{
  "title": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": true,
  "mapping": {
    "metadata.color": "{{ \"The color is #{color}\" }}"
  },
```

```
  "unset": ["color"]
}
```

Both turn this original object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "color": "blue"
}
```

Into this new object

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "metadata": {
    "color": "The color is blue"
  }
}
```

Another useful twig take is the if statement. This can be used to check if a value exists in the first place in our mapping

```
{
  "title": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": true,
  "mapping": {
    "metadata.color": "{% if color %} {{color}} {% else %} unknown {% endif %}"
  },
  "unset": ["color"]
}
```

or to check for specific values in our mapping

```
{
  "title": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": true,
  "mapping": {
    "metadata.color": "{% if color == \"violet\" %} pink {% endif %}"
  },
  "unset": ["color"]
}
```

Sub mappings

In some cases, you might want to make use of mappings that you have created before with the mapping you are currently defining. Common cases include mapping an array of sub-objects or dividing your mapping into smaller files for stability and maintenance purposes.

To do this you can access the mapping service from within a mapping through twig like:

```
{
  "title": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": true,
  "mapping": {
    "color": "{{ color|map('{reference}', {array}) }}"
  }
}
```

The mapping service takes three arguments:

- reference [required]: Either the reference of the mapping that you want to use
- array [required]: The actual data that you want to map
- list [optional, defaults to false]: Set this to true if you want to map as a list (of objects) instead of its entirety (as one object).

Casting (Forcing) the type/format of values

In some cases, you might want to change the properties variable type or if you are using twig rendering, mapping output will always change all the values to `string`. For internal gateway traffic, this isn't problematic, as the data layer will cast values to the appropriate outputs. When sending data to an external source, having all Booleans cast to strings might be bothersome. To avoid this predicament, we can force the datatype of your values by 'casting' them.

We can cast values by including a cast property in our mapping, the following type casts are currently available:

Cast	Function (php docs)	Twig	Notes
string	php Type Juggling	No	x
bool / boolean	php Type Juggling	No	x
int / integer	php Type Juggling	No	x
float	php Type Juggling	No	x
array	php Type Juggling	No	x
date	php date function	No	x
url	php urlencode function	Yes	x
urlDecode	php urldecode function	Yes	x
rawurl	php rawurlencode function	Yes	x
rawurlDecode	php rawurldecode function	Yes	x
html	php htmlentities function	Yes	x
htmlDecode	php html_entity_decode function	Yes	x
base64	php base64-encode function	Yes	x
base64Decode	php base64-decode function	Yes	x
json	php json-encode function	Yes	x

jsonToArray	php json-decode function	Yes	The htmlDecode cast is always done before this cast

That means that we can write a mapping like

```
{
  "title": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": true,
  "cast": {
    "age": ["int"],
    "available": ["bool"]
  }
}
```

To turn this original object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "age": "2",
  "available": "yes"
}
```

Into the new object

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "age": 2,
  "available": true
}
```

> ***Note***
>
> - *Beware what functions PHP uses to map these values and if the cast should be possible (or else an error is thrown).*
> - *Casting is always the last action performed by the mapping service*

Special casting (Forcing) change of values

In some rarer cases you might want to not 'just cast to a different type' but change a value entirely, these casts do not match with just one specific php cast or php function but contain more than one line of code (or need some extra explanation on how they work). In most normal cases when you want to cast to (for example) an integer you would only use one cast `["integer"]` but with these casts it isn't unusual to combine multiple casts such as `["jsonToArray", "unsetIfValue=="]`.

We can change values by including a cast property in our mapping, the following special casts are currently available:

Cast	Description
nullStringToNull	This cast checks if the value equals string = 'null' and casts it to actual null.

coordinateStringToArray	This cast converts a coordinate string to an array of coordinates.
keyCantBeValue	This cast checks if the value equals the property name and if so, unsets the property.
unsetIfValue	This cast checks if the value equals a specific value and if so, unsets the property. An example: `"unsetIfValue==example`. This can also be used to check if the value is empty with `"unsetIfValue==`".
countValue	This cast uses the php count function to count another value, if that other value is countable, sets the property to the count of that other value. An example: `"countValue:example`. This is equal to php `count(example)`.

That means that we can write a mapping like

```
{
  "title": "A more complex mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": false,
  "mapping": {
    "doggies": "{% if subMapping is defined and subMapping is not empty %}{{
map(‘{reference}’, subMapping, true)|json_encode }}{% else %}\"\"{% endif %}"
  },
  "cast": {
    "doggies": ["jsonToArray", "unsetIfValue=="]
  }
}
```

To turn this original object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "example",
  "doggies": null
}
```

Into the new object

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "example"
}
```

Or to turn this original object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "example2",
  "doggies": [
    {
      "name": "doggie",
      "description": "<- renamed to note by the subMapping",
      "age": 2
    }
```

```
  ]
}
```

Into the new object

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "example2",
  "doggies": [
    {
      "name": "doggie",
      "note": "<- renamed to note by the subMapping",
      "age": 2
    }
  ]
}
```

Translating values

Twig natively supports translations, but remember that translations are an active filter `|trans`. And thus should be specifically called on values you want to translate. Translations are performed against a translation table. You can read more about configuring your translation table here.

The base for translations is the locale, as provided in the localization header of a request. When sending data, the base is in the default setting of a gateway environment. You can also translate from a specific table and language by configuring the translation filter e.g. `{{ 'greeting' | trans({}, [table_name], [language] ) }}`

The following mapping:

```
{
  "title": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": true,
  "mapping": {
    "color": "{{source.color|trans({},\"colors\") }}"
  }
}
```

Wil turn this original object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "color": "blue"
}
```

Into this new object (on locale nl):

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
```

```
  "color": "blauw"
}
```

If we want to force German (even if the requester asked for a different language), we'd map like

```
{
  "title": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": true,
  "mapping": {
    "color": "{{source.color|trans({},\"colors\".\"de\") }}"
  }
}
```

And get the following new object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "color": "Blau"
}
```

> ***Note***
>
> - *In most cases request won't be originating from a browser, so its best to ALWAYS define the language that you would like to use*

Renaming Keys

The mapping doesn't support the renaming of keys directly but can rename keys indirectly by moving the data to a new position and dropping the old position (if we are using passThrough).

For example, we could write a mapping like this:

```
{
  "title": "A simple mapping",
  "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
  "passthrough": true,
  "mapping": {
    "title": "name"
  },
  "unset": [
    "name"
  ]
}
```

To turn this original object:

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "color": "blue"
}
```

Into this new object

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "title": "doggie",
  "color": "blue"
}
```

Order of mapping

The mapping service always handles all mappings in the following order

1. passTrough
2. mapping
3. unset
4. cast

What if I can't map?

Even with all the above options, it might be possible that the objects you are looking at are too different to map. In that case, look for something other than mapping solutions. If the old and new object are too different, add them to the data layer and write a plugin to keep them in sync based on actions.

Using ChatGDP

Mappings are mostly based on comparing the original object you have and the new object you require the actual, or in other words comparing statuses and writing something to get from original to new. In practice, this is a pretty straightforward process that can easily be left to online AIs (like ChatGDP) to write the first version for you. Simply head over to https://chat.openai.com/ and start a prompt to let chat GDP know what you want to do.

We generally ask it `Can you provide a new example based on the mapping service? [copy past this readme from start to passTrough]` , like:

Can you provide an new example basied on the mapping service?

Mapping Service
The mapping service supports the process of changing the structure of an object. It's used to transform data when the source doesn't match the desired data model. Mapping is done by a series of mapping rules in a To <- From style. In simple mapping, the position of a value within an object is changed.

Index

Defining a mapping
Usage
Advanced (Twig) mapping and/or adding key's
Pass Through and/or dropping key's
Working with conditional data
Sub mappings
Casting (Forcing) the type/format of values
Translating values
Renaming Keys
Order of mapping
What if I can't map?
Defining a mapping
The Common Gateway stores,imports and exports mappings as JSON mapping objects.
Bellow you can find an example mapping object

```
{
 "title": "MyMapping",
 "description": "MyMapping",
 "$id":
"https://development.zaaksysteem.nl/mapping/xxllnc.XxllncCaseToZGWZaak.mapping.json",
 "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
 "version": "0.0.1",
 "passTrough": false,
 "mapping": {
  "{{to_key}}": "{{from_key}}"
 },
 "unset": ["{{from_key}}"],
 "cast": {
  "{{to_key}}":"{{type}}"
 }
}
```

ChatGDP should now explain to you how it would create mappings.

Yes, I can provide an example to show that I understand the instructions.

Let's say we have an object with the following structure:

```json
{
    "name": "John",
    "age": 30,
    "city": "New York",
    "country": "USA",
    "interests": ["music", "sports", "movies"]
}
```

We want to transform this object into a new structure where the name, age, and city are mapped as is, and the country and interests are moved into a new sub-object called "details". To achieve this, we can define the following mapping object:

```json
{
    "title": "User mapping",
    "description": "Map user object to new structure",
    "$schema": "https://docs.commongateway.nl/schemas/Mapping.schema.json",
    "version": "1.0.0",
    "mapping": {
        "name": "name",
        "age": "age",
        "city": "city",
        "details.country": "country",
        "details.interests": "interests"
    }
}
```

In this mapping object, we have defined the new structure by using dot notation to specify the sub-object "details". The mapping rules are defined in a `To <- From` style, where the key on the left side of the arrow is the new key, and the key on the right side of the arrow is the original key.

Oke so now we have ChatGDP setup, we can as it to create mappings for us. Let's ask it to create a simple mapping for us

Normally we would ask something like this:

Can you create a mapping from this original object

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "color": "blue"
}
```

Into the following new object

```
{
  "id": "0d671e30-04af-479a-926a-5e7044484171",
  "name": "doggie",
  "metadata":{
    "color": "blue",
    "dateCreated": "use twig to create a timestamp"
  }
}
```

based on the mapping service

[copy past this readme from start to passTrough]

Can you create a mapping from this original object

```json
{
 "id":"0d671e30-04af-479a-926a-5e7044484171",
 "name":"doggie",
 "color": "blue"
}
```

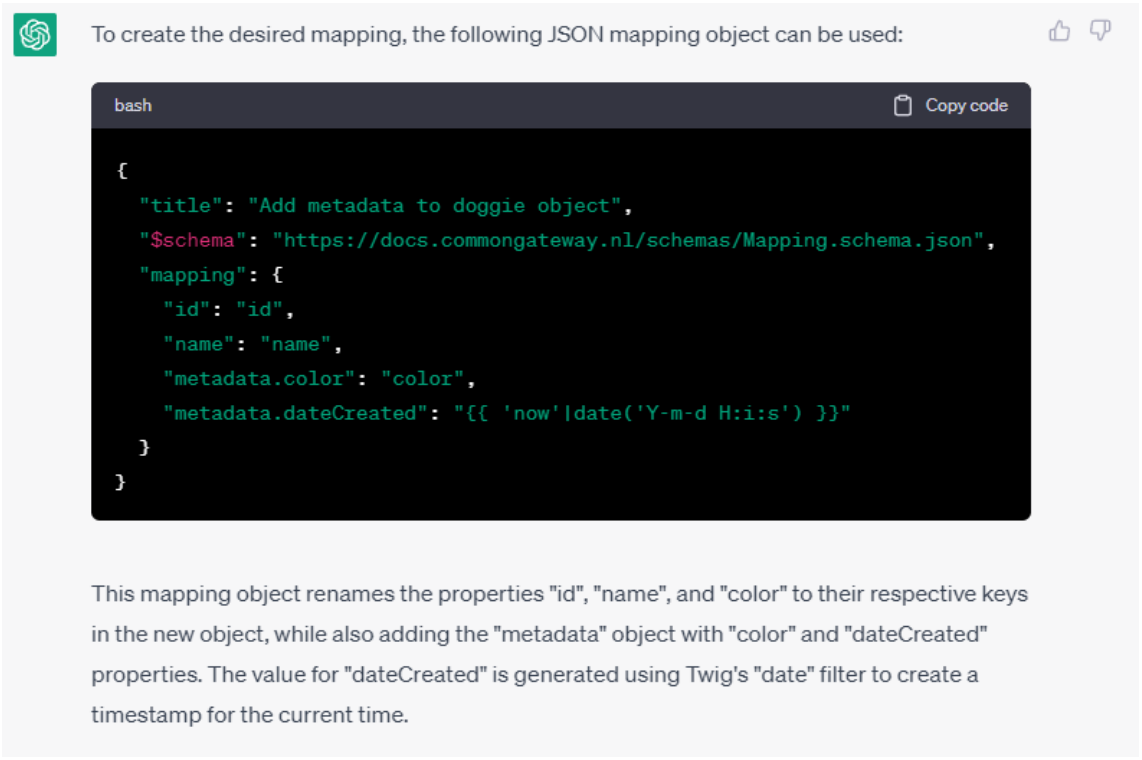
Into the following new object

```json
{
 "id":"0d671e30-04af-479a-926a-5e7044484171",
 "name":"doggie",
 "metadata":{
  "color": "blue",
  "dateCreated":"use twig to create a time stamp"
 }
}
```

based on the mapping service

Mapping
Mapping is the process of changing the structure of an object. It's used to transform data when the source doesn't match the desired data model. Mapping is done by a series of mapping rules in a To <- From style. In simple mapping, the position of a value within an object is changed.

And, presto! ChatGDP writes a basic mapping for us



> ***Note***
>
> - *Conveniently ChatGDP provides a copy code button at the right top of the coding example that allows us to simply download the provided mapping and import it into the gateway.*
> - *We cut the above screenshots short for layout reasons but be sure to include as much from the mapping readme as you can*
> - *Always check the code that ChatGDP provided! It is known to make errors ;)*

Monitoring

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions or alterations there.*

As a gateway, the Common Gateway sits at the front of your application landscape, acting as the main entry point for all incoming requests. This central position makes it a prime location for collecting valuable data about the health, performance, and behavior of your application ecosystem.

Monitoring is vital to the Common Gateway for several reasons:

1. **Performance Monitoring**: By tracking metrics like request duration, rate, and error rates, you can gain insights into the performance of your application. This can help you identify bottlenecks, understand capacity needs, and ensure that your application is performing optimally.
2. **Health Checks**: Health checks provide a way to quickly detect and respond to issues that could impact the availability or performance of your services. These checks can be used to trigger alerts, ensuring that you can respond quickly when problems arise.
3. **Debugging and Troubleshooting**: When issues do arise, having detailed metrics at your disposal can be invaluable for diagnosing and resolving the problem. Prometheus's querying language PromQL

can be used to slice and dice the data in various ways, making it easier to understand the root cause of an issue.
4. **Capacity Planning and Scaling**: By tracking the load on your services, you can make informed decisions about when to scale up or down, helping you manage costs and ensure sufficient capacity to handle your traffic.

Prometheus

The Common Gateway supports Prometheus monitoring through its MetricsService. This service exposes a /metrics endpoint that Prometheus can scrape to collect metrics about the operation of the Common Gateway. This makes it easy to integrate the Common Gateway with Prometheus, allowing you to benefit from the rich insights that Prometheus can provide about the health and performance of your gateway and the services behind it.

Prometheus is an open-source systems monitoring and alerting toolkit that is widely adopted for its simplicity and effectiveness. It collects metrics from monitored targets by scraping metrics HTTP endpoints on these targets.

Supported Metrics

The common gateway supports several metrics

General information

Name	Type	Help
app_version	gauge	The current version of the application.
app_name	gauge	The name of the current version of the application.
app_description	gauge	The description of the current version of the application.
app_users	gauge	The current amount of users
app_organisations	gauge	The current amount of organisations
app_applications	gauge	The current amount of applications
app_requests	counter	The total amount of incoming requests handled by this gateway
app_calls	counter	The total amount of outgoing calls handled by this gateway

Errors

Name	Type	Help
app_error_count	counter	The amount of errors, this only counts logs with level_name 'EMERGENCY', 'ALERT', 'CRITICAL' or 'ERROR'.
app_error_list	counter	The list of errors and their error level/type.

Objects

Name	Type	Help
app_objects_count	gauge	The amount objects in the data layer

app_cached_objects_count	gauge	The amount objects in the data layer that are stored in the MongoDB cache
app_schemas_count	gauge	The amount defined schemas
app_schemas	gauge	The list of defined schemas and the amount of objects.

Plugins

Name	Type	Help
app_plugins_count	gauge	The amount of installed plugins
app_installed_plugins	gauge	The list of installed plugins.

Notifications

> ***Warning*** *This file is maintained at Conduction's Google Drive Please make any suggestions of alterations there.*

CloudEvents is a specification that enables consistent description of event data across diverse services, platforms, and systems. This uniformity addresses the challenge of different event producers describing events in varying ways, which often requires developers to continuously learn how to consume these events. The existence of a common event description can enhance the utility of libraries, tools, and infrastructure designed to facilitate the delivery of event data across different environments, such as SDKs, event routers, and tracing systems. By promoting interoperability, CloudEvents increases the potential for data portability and productivity. The specification has attracted considerable interest from various industry players, including major cloud providers and popular SaaS companies. Hosted by the Cloud Native Computing Foundation (CNCF), CloudEvents was approved as a Cloud Native sandbox level project on May 15, 2018, and as an incubator project on October 24, 20191.

Plugins

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions or alterations there.*

Plugins are a neat way of separating concerns and making sure that client specific code doesn't get into the core. You can read a bit more about why we use plugins under code quality.

The Common Gateway is easily extendable through a plugin structure. The structure is based on the Symfony bundle system in other words, all Common Gateway plugins are Symfony bundles, and Symfony bundles can be Common Gateway plugins. You can consider a plugin for the Common Gateway as a configuration set to extend a base Gateway's functionality. The plugin structure is based on the Symfony bundle system. In other words, all Common Gateway plugins are Symfony bundles, and Symfony bundles can be Common Gateway plugins.

If you want to develop your own plugin, we suggest using the Pet store plugin as a starting point.

Finding and installing plugins

If you start from a brand new Gateway installation and head over to your Dashboard, you can find the plugin section on the left side panel. You can search for the plugins you want to add from this tab by selecting Search for plugins. Find the plugin you wish to install and view its details page. You should see an install button in the top right corner if the plugin is not installed.

The Common Gateway finds plugins to install with packagist. It does this entirely under the hood, and the only requirement is that plugins need a ‘common-gateway-plugin” tag. Packagist functions as a plugin store as well in this regard.

The plugins are installed, updated, and removed with the composer CLI. While this feature still exists for developers, we recommend using the user interface see plugins for installing plugins.

Creating plugins

If you want to develop your plugin, we recommend using the PetStoreBundle. This method ensures all necessary steps are taken, and the plugin will be found and installable through the method described above.

Updating and removing plugins

In case you want to update or remove a plugin, go to “Plugins” in the Gateway UI main menu and select “Installed”. Click on the plugin that you want to update or remove and press the Update or Remove button in the top right of the screen.

Adding Actions, Sources, Cronjobs, to your plugin

You can include an installation folder in the root of your plugin repository containing schema.json files or other files. Whenever the Gateway installs or updates a plugin, it looks for the schema map and handles all schema.json files in that folder as a schema upload.

Keep in mind that you will need to properly set the $schema of the object in order for the gateway to understand what schema you are trying to create. The core schema’s of the gateway are defined as

- 'https://docs.commongateway.nl/schemas/Action.schema.json',
- 'https://docs.commongateway.nl/schemas/Application.schema.json',
- 'https://docs.commongateway.nl/schemas/CollectionEntity.schema.json,
- 'https://docs.commongateway.nl/schemas/Cronjob.schema.json',
- 'https://docs.commongateway.nl/schemas/DashboardCard.schema.json',
- 'https://docs.commongateway.nl/schemas/Endpoint.schema.json',
- 'https://docs.commongateway.nl/schemas/Entity.schema.json',
- 'https://docs.commongateway.nl/schemas/Gateway.schema.json',
- 'https://docs.commongateway.nl/schemas/Mapping.schema.json',
- 'https://docs.commongateway.nl/schemas/Organization.schema.json',
- 'https://docs.commongateway.nl/schemas/SecurityGroup.schema.json',

> *Note: While adding SecurityGroups through core schema's is allowed, adding (or changing) Users is not, because of security reasons, if you would like to add users (in a more secure way) take a look at how to configure an installation.json file.*
>
> - '*https://docs.commongateway.nl/schemas/User.schema.json*',

Here is an example. The $id and $schema properties are needed for the Gateway to find the plugin. The version property's value helps the Gateway decide whether an update is required and will update automatically.

Installation

The gateway supports installations, update and remove actions for plugins. Allowing them to change configurations and alter data, this is done through stallation.json file. The installation.json file is a fundamental part of the plugin installation process. It provides the necessary configuration for a plugin to integrate smoothly with the platform. This file should be located in the `/Installation` folder of the plugin's directory.

Here is an explanation of the various sections in the installation.json file:

InstallationService

- **installationService**: This specifies the service that handles the installation process. For example: "installationService": "CommonGateway\PetStoreBundle\Service\InstallationService"

The installation service allows you to run code during changes to the plugins livecycle. It **MUST** always implement the `CommonGateway\CoreBundle\Installer\InstallerInterface`. And it **CAN** provide functions that are called during changes to the plugin from the gateways installer.

An example installationService could like like

```php
<?php
/**
 * The installation service
 *
 * @author  Conduction.nl <info@conduction.nl>
 * @license EUPL-1.2 https://joinup.ec.europa.eu/collection/eupl/eupl-text-eupl-12
 */

namespace CommonGateway\PetStoreBundle\Service;

use CommonGateway\CoreBundle\Installer\InstallerInterface;
use Doctrine\ORM\EntityManagerInterface;
use Psr\Log\LoggerInterface;


class InstallationService implements InstallerInterface
{

    /**
     * The entity manager
     *
     * @var EntityManagerInterface
     */
    private EntityManagerInterface $entityManager;

    /**
     * The installation logger.
     *
     * @var LoggerInterface
     */
    private LoggerInterface $logger;


    /**
     * The constructor
     *
     * @param EntityManagerInterface $entityManager      The entity manager.
     * @param LoggerInterface        $installationLogger The installation logger.
     */
    public function __construct(
        EntityManagerInterface $entityManager,
        LoggerInterface $installationLogger
    ) {
```

```
        $this->entityManager = $entityManager;
        $this->logger        = $installationLogger;

    }//end __construct()


    /**
     * Every installation service should implement an install function
     *
     * @return void
     */
    public function install()
    {
        $this->logger->debug("PetStoreBundle -> Install()");

        $this->checkDataConsistency();

    }//end install()


    /**
     * Every installation service should implement an update function
     *
     * @return void
     */
    public function update()
    {
        $this->logger->debug("PetStoreBundle -> Update()");

        $this->checkDataConsistency();

    }//end update()


    /**
     * Every installation service should implement an uninstall function
     *
     * @return void
     */
    public function uninstall()
    {
        $this->logger->debug("PetStoreBundle -> Uninstall()");

        // Do some cleanup to uninstall correctly...

    }//end uninstall()


    /**
     * The actual code run on update and installation of this bundle
     *
     * @return void
     */
    public function checkDataConsistency()
    {
```

```
        //This is the place where you can add or change Installation data from/for this
bundle or other required bundles.
        //Note that in most cases it is recommended to use .json files in the Installation
folder instead, if possible.

        $this->entityManager->flush();

    }//end checkDataConsistency()


}//end class
```

> *Note: In most cases it isn't actually necessary to write an installation service, you can just load configurations by supplying the necessary objects.*

Configuration

There are two routes to include configuration (objects) in your plugin. The first and easiest one is to include them directly into your installation.json. This is possible for applications, users, cards, actions, collections, endpoints and cronjobs. You can also supply the object separately, in that case they **MUST** be contained in de the `/Installation` folder of the plugin's directory and **SHOULD** be in a sub folder labeled after the type of object that you want to create e.g. `/Cards`. This is the preferred way (especially with larger plugins) because it keeps a repository more readable.

If however you want to create objects from the `installation.json` you can use the following properties:

- **applications**: This is an array of applications related to the plugin. Each application should have properties like title, $id, $schema, version, description, and domains.
- **users**: This section defines the users that have access to the plugin. Each user should have properties like $id, version, description, email, locale, and securityGroups.
- **cards**: This section includes properties like schemas, collections, and applications.
- **actions**: This section defines the handlers and the actions associated with them. Each handler should have properties like reference, actionHandler, listens, and configuration. The configuration section includes specific parameters that the handler uses.
- **collections**: This is an array of collections that the plugin should have access to. Each collection should have properties like reference and schemaPrefix.
- **endpoints**: This section defines the API endpoints that the plugin exposes. This section is divided into multipleSchemas and schemas. The multipleSchemas section allows defining endpoints that use multiple schemas. Each endpoint should have properties like $id, version, name, description, schemas, path, pathRegex, and methods. The schemas section allows defining endpoints specific to a schema.
- **cronjobs**:

Below is an example of the structure of an installation.json file:

```
{
  "installationService": "CommonGateway\\PetStoreBundle\\Service\\InstallationService",
  "applications": [
    {
      "title": "Example Front-end Application",
      "$id": "https://example.com/application/ps.frontend.application.json",
```

```
      "$schema": "https://docs.commongateway.nl/schemas/Application.schema.json",
      "version": "0.0.1",
      "description": "An example Front-end Application. This is not required for the
gateway to work, there is a default Application created on init. Applications can be used
to allow the given domains to use the gateway. And can be used by plugin services to get a
domain of an application.",
      "domains": [
        "frontend.example.com"
      ]
    }
  ],
  "users": [
    {
      "$id": "https://example.com/user/johnDoe.user.json",
      "version": "0.0.1",
      "description": "An example User with an example SecurityGroup. It is not allowed to
set a User password or change/create Admin Users this way.",
      "email": "johnDoe@username.com",
      "locale": "en",
      "securityGroups": [
        "https://example.com/securityGroup/example.securityGroup.json"
      ]
    }
  ],
  ...
}
```

The above configuration represents a part of the installation.json for a hypothetical Pet Store plugin. It specifies the installation service, an application, and a user. Additional sections should be added as needed, following the structure outlined above.

Please ensure that your installation.json file follows this structure and includes all required sections for your plugin. This will ensure a smooth installation process and correct integration of your plugin with the system

> *Note: The installation.json should not contain any descriptionarry information about the plugin. That should be provided trough the* `composer.json` *in the plugins route.*

Adding test data or fixtures to your plugin

You can include both fixtures and test data in your plugin. The difference is that fixtures are required for your plugin to work, and test data is optional. You can include both data sets as .json files in the folder at the root of your plugin repository. An example is shown here.

Datasets are categorized by name, e.g., data.json in the data folder will be considered a fixture, whereas [anything else].json will be regarded as test or optional data (and not loaded by default).

As a fixture, anything in data.json is always loaded on a plugin installation or update. The other files are never loaded on a plugin install or update. However, the user can load the files manually from the plugin details page in the gateway UI.

All files should follow the following convention in their structure 1 - A primary array indexes on schema refs, 2 - Secondary array within each primary array containing the objects that you want to upload or update for that specific schema.

```
{
  "ref1": [
```

```
    {"object1"},
    {"object2"},
    {"object3"}
  ],
  "ref2":[
    {"object1"},
    {"object2"},
    {"object3"}
  ],
  "ref3":[
    {"object1"},
    {"object2"},
    {"object3"}
  ]
}
```

Keep in mind that the `ref` here could either be a schema you defined yourself or a gateway core schema. It is therefore also possible to upload several configuration files through a data.json.

When handling data uploads (be it fixtures or others), the Gateway will loop through the primary array, trying to find the appropriate schema for your object. If it finds this schema, it creates or updates the object for the schema.

For fixtures, this is done in an unsafe manner, meaning that When the Gateway can't find a schema for a reference, it will ignore the reference and continue The Gateway won't validate the objects (meaning that you can ignore property requirements, but you can't add values for non-existent properties)

When handling other uploads with optional ore test data the Gateway does so in a safe manner, meaning: When the Gateway can't find a schema for a reference, it will throw an error The Gateway will validate the objects and throw an error when it reaches an invalid object

Describing your plugin

Plugins are described trough a `composer.json` file in the plugin root. In order for a plugin to findable and installable for common gateway installations it **MUST** meet the following criteria:

- Have a name
- Type is set to `symfony-bundle`
- `common-gateway-plugin` is added to the key-words `commongateway/corebundle` is a requirement

Other fields are optional but highly recommended, keep in mind that both the gateway plugin store and common-gateway website look for a readme.md in the repository root to display as personal page for the plugin.

An example composer file would look like

```
{
    "name": "common-gateway/pet-store-bundle",
    "description": "An example package for creating symfony flex bundles as plugins",
    "type": "symfony-bundle",
    "keywords": [
        "commongateway",
        "common",
        "gateway",
        "conduction",
        "symfony",
        "common-gateway-plugin",
```

```
        "pet store"
    ],
    "homepage": "https://commongateway.nl",
    "license": "EUPL-1.2",
    "minimum-stability": "dev",
    "require": {
        "php": ">=7.4",
        "commongateway/corebundle": "^1.0.51"
    },
    "require-dev": {
        "symfony/dependency-injection": "~3.4|~4.1|~5.0"
    },
    "autoload": {
        "psr-4": {
            "CommonGateway\\PetStoreBundle\\": "src/"
        }
    }
}
```

Publishing your plugin

The gateway used the packagist netwerk for plugin discovery, that means that you do not need to upload your plugin to an appstore etc. You simply submit your plugin repository to packagist.

> *Note: It is also possible to keep your plugins private, read more about that under {private packages] (https://packagist.com/)*

> *Warning: Plugins are only finable if they adhere to all of the description requirements.*

Requiring other plugins

Features

Schemas

> ***Warning** This file is maintained at the Conduction Google Drive. Please make any suggestions or alterations there.*

Schemas are the core of the Common Gateway's data layer. They define and model objects and set the conditions for objects. Each object in the gateway always belongs to ONE schema. Schemas follow the JSON schema standard and are therefore interchangeable with OAS3 schemas. For the Dutch governmental ecosystem this means that a schema adheres to the `overige objecten standaard` description of an `object type`. In action to this we extend schema's with metadata.

In a more traditional way, schema's can be viewed as the "tables" of the data layer as they store data in a predefined way. However, unlike tables, the data is stored as objects. Where each data set, that would normally be a row, becomes an object. The main difference between tables and objects is that objects are multidimensional(a value can be another object) and table rows are flat (each column containing one value). An object can contain an array, objects or arrays of objects. Objects present us with a vastly superior way of serving data.

An example object could be

```
{
  "id": 1,
  "name": "doggie",
  "status": "available"
}
```

Schema's define objects by giving us the properties they contain and conditional validations for the value of each property.

Creating or updating a schema

Schema’s can be modeled from the schema page in the Admin UI or through the `/admin/schema` endpoint. To create a new schema go to “Schemas” in the menu, and press “Add schema”. Just fill in the name “PET and hit save (a schema needs to be created before you can add properties). After the schema is created you are automatically redirected to the edit page of that schema.

Adding properties to a schema

Go to the properties tab and press “Add property” to add a property to your schema. When adding a property

Adding objects

After adding properties to a schema

Downloading schema’s

Uploading schema’s

You can upload a schema in the Gateway UI by pressing the upload button in the top right corner. Schemas might also be uploaded by plugins or collections. When a schema is uploaded the following things will happen:

The Gateway will look in its schema library if a version of that schema is already present. It does so based on the schema ID. Based on that result the Gateway will handle the schema accordingly: If no matching schema is found the gateway will create a new schema If a matching schema is found the gateway wil compare versions and decide what to do: The old schema has no set version and the new schema has no set version -> The gateway will update the old schema with the new schema The old schema has no set version and the new schema has a set version -> The gateway will update the old schema with the new schema The old schema has a set version number and the new schema has a higher set version number -> The gateway will update the old schema with the new schema The old schema has a set version number and the new schema has a lower set version number -> The gateway does not update the schema The old schema has a set version number and the new schema does not have a set version number -> The gateway does not update the schema

Let's take a look at an example. We have a weather plugin that contains a weather schema.

Properties

An entity consists of the following properties that can be configured

Property	Required	Description
name	yes	An unique name for this entity

description	no	The description for this entity that will be shown in de API documentation
source	no	The source where this entity resides
endpoint	yes if an source is provided	The endpoint within the source that this entity should be posted to as an object
route	no	The route this entity can be found easier, should be a path
extend	no	Whether or not the properties of the original object are automatically included

Properties

Properties represent variables on objects. In the following object from the petstore api id, name, and status are properties.

```
{
  "id": 1,
  "name": "doggie",
  "status": "available"
}
```

that you want to communicate to underlying sources. In a normal setup and attribute should at least apply the same restrictions as the underlying property (e.g. required) to prevent errors when pushing the entity to its source. It can however provide additional validations to a property, for example the source AIU might simply require the property ‘email’ to be a unique string, but you could set the form to ‘email’ causing the input to be validated as an ISO compatible email address.

Properties

Property	Required	Description
name	yes	`string` An name for this attribute. MUST be unique on an entity level and MAY NOT be ‘id’,’file’,‘files’, ’search’,’fields’,’start’,’page’,’limit’,’extend’ or ’organization’
description	no	The description for this attribute that will be shown in de API documentation
type	yes	`string` See types
format	no	`string` See formats
validations	no	`array of strings` See validations
multiple	no	`boolean` if this attribute expects an array of the given type
defaultValue	no	`string` An default value for this value that will be used if a user doesn't supply a value
deprecated	no	`boolean` Whether or not this property has been deprecated and will be removed in the future
required	no	`boolean` whether or not this property is required to be in a POST or UPDATE

requiredIf	no	`array` a nested set of validations that will cause this attribute to become required
forbidden	no	`boolean` whether or not this property is forbidden to be in a POST or UPDATE
forbiddenIf	no	`array` a nested set of validations that will cause this attribute to become forbidden
example	no	`string` An example of the value that should be supplied
persistToSource	no	`boolean` Setting this property to true will force the property to be saved in the gateway endpoint (default behavior is saving in the EAV)
searchable	no	`boolean` Whether or not this property is searchable
cascade	no	`boolean` Whether or not this property can be used to create new entities (versus when it can only be used to link existing entities)

> ***Warning*** *To prevent collisions with json-ld, json-hall, graphql and inner gateway workings property names aren't allowed to start with the following characters* `_` , `@` , `$` *additionally you can't add a property called* `id` *to your schema's. When importing schema's all properties in violation of the above will be ignored without warning.*

####Types The type of attribute provides basic validations and a way for the gateway to store and cash values in an efficient manner. Types are derived from the OAS3 specification. Current available types are:

Format	Description
string	a text
integer	a full number without decimals
decimal	a number including decimals
boolean	a true/false
date	an ISO-??? date
date-time	an ISO-??? date
array	an array or list of values
object	Used to nest a Entity as attribute of another Entity, read more about nesting
file	Used to handle file uploads, an Entity SHOULD only contain one attribute of the type file, read more about handling file uploads

- you are allowed to use integer instead of int, boolean instead of bool, date-time or dateTime instead of datetime,

####Formats A format defines a way a value should be formatted, and is directly connected to a type, for example a string MAY BE a format of email, but an integer cannot be a valid email. Formats are derived from the OAS3 specification, but supplemented with formats that are generally needed in governmental applications (like BSN) . Current available formats are:

General formats

Format	Type(s)	Description

alnum		Validates whether the input is alphanumeric or not. Alphanumeric is a combination of alphabetic and numeric characters
alpha		Validates whether the input contains only alphabetic characters
numeric		Validates whether the input contains only numeric characters
uuid	string	
base		Validate numbers in any base, even with non regular bases.
base64		Validate if a string is Base64-encoded.
countryCode	string	Validates whether the input is a country code in ISO 3166-1 standard.
creditCard	string	Validates a credit card number.
currencyCode	string	Validates an ISO 4217 currency code like GBP or EUR.
digit	string	Validates whether the input contains only digits.
directory	string	Validates if the given path is a directory.
domain	string	Validates whether the input is a valid domain name or not.
url	string	Validates whether the input is a valid url or not.
email	string	Validates an email address.
phone	string	Validates a phone number.
fibonacci	integer	Validates whether the input follows the Fibonacci integer sequence.
file	string	Validates whether file input is as a regular filename.
hexRgbColor	string	Validates whether the input is a hex RGB color or not.
iban	string	Validates whether the input is a valid IBAN (International Bank Account Number) or not.
imei	string	Validates if the input is a valid IMEI.
ip	string	Validates whether the input is a valid IP address.
isbn	string	Validates whether the input is a valid ISBN or not.
json	string	Validates if the given input is a valid JSON.
xml	string	Validates if the given input is a valid XML.
languageCode	string	Validates whether the input is language code based on ISO 639.
luhn	string	Validate whether a given input is a Luhn number.
macAddress	string	Validates whether the input is a valid MAC address.
nfeAccessKey	string	Validates the access key of the Brazilian electronic invoice (NFe).

- Phone numbers should ALWAYS be treated as a string since they MAY contain a leading zero.

Country specific formats

Format	Type(s)	Description

bsn	string	Dutch social security number (BSN)
nip	string, integer	Polish VAT identification number (NIP)
nif	string, integer	Spanish fiscal identification number (NIF)
cnh	string, integer	Brazilian driver’s license
cpf	string, integer	Validates a Brazilian CPF number
cnpj	string, integer	Validates if the input is a Brazilian National Registry of Legal Entities (CNPJ) number

- Dutch BSN numbers should ALWAYS be treated as a string since they MAY contain a leading zero.

####Validations Besides validations on type and string you can also use specific validations, these are contained in the validation array. Validation might be specific to certain types or formats e.g. minValue can only be applied to values that can be turned into numeric values. And other validations might be of a more general nature e.g. required.

Validation	value	Description
between		Validates whether the input is between two other values.
boolType		Validates whether the type of the input is boolean.
boolVal		Validates if the input results in a boolean value.
call		Validates the return of a [callable][] for a given input.
callableType		Validates whether the pseudo-type of the input is callable.
callback		Validates the input using the return of a given callable.
charset		Validates if a string is in a specific charset.
alwaysInvalid		Validates any input as invalid
alwaysValid		Validates any input as valid
anyOf		This is a group validator that acts as an OR operator. AnyOf returns true if at least one inner validator passes.
arrayType		Validates whether the type of an input is array
arrayVal		Validates if the input is an array or if the input can be used as an array (instance of ArrayAcces or SimpleXMLElement.
attribute		Validates an object attribute, even private ones.
consonant		Validates if the input contains only consonants.
contains		Validates if the input contains some value.
containsAny		Validates if the input contains at least one of defined values.

control		Validates if all of the characters in the provided string, are control characters.
countable		Validates if the input is countable, in other words, if you’re allowed to use count() function on it.
decimal		Validates whether the input matches the expected number or decimals.
each		Validates whether each value in the input is valid according to another rule.
endsWith		This validator is similar to Contains(), but validates only if the value is at the end of the input.
equals		Validates if the input is equal to some value.
equivalent		Validates if the input is equivalent to some value.
even		Validates whether the input is an even number or not.
executable		Validates if a file is an executable.
exists		Validates files or directories.
extension		Validates if the file extension matches the expected one. This rule is case-sensitive.
factor		Validates if the input is a factor of the defined dividend.
falseVal		Validates if a value is considered as false.
file		Validates whether file input is as a regular filename.
image		Validates if the file is a valid image by checking its MIME type.
filterVar		Validates the input with the PHP’s filter_var() function.
finite		Validates if the input is a firtine number.
floatType		Validates whether the type of the input is float.
floatVal		Validate whether the input value is float.
graph		Validates if all characters in the input are printable and actually creates visible output (no white space).
greaterThen		Validates whether the input is greater than a value.
identical		Validates if the input is identical to some value.
in		Validates if the input is contained in a specific haystack.
infinite		Validates if the input is an infinite number.
instance		Validates if the input is an instance of the given class or interface.
iterableType		Validates whether the pseudo-type of the input is iterable or not, in other words, if you're able to iterate over it with foreach language construct.
key		Validates an array key.
keyNested		Validates an array key or an object property using . to represent nested data.
keySet		Validates keys in a defined structure.

keyValue		
leapDate		Validates if a date is leap.
leapYear		Validates if a year is leap.
length		
lessThan		Validates whether the input is less than a value.
lowercase		Validates whether the characters in the input are lowercase.
not		
notBlank		Validates if the given input is not a blank value (null, zeros, empty strings or empty arrays, recursively).
notEmoji		Validates if the input does not contain an emoji.
no		Validates if value is considered as “No”.
noWhitespace		Validates if a string contains no whitespace (spaces, tabs and line breaks).
noneOf		Validates if NONE of the given validators validate.
max		Validates whether the input is less than or equal to a value.
maxAge		Validates a maximum age for a given date. The $format argument should be in accordance with PHP's date() function.
mimetype		Validates if the input is a file and if its MIME type matches the expected one.
min		Validates whether the input is greater than or equal to a value.
minAge		Validates a minimum age for a given date. The $format argument should be in accordance with PHP's date() function.
multiple		Validates if the input is a multiple of the given parameter.
negative		Validates whether the input is a negative number.

Objects

An object is a data set conforming to schema, e.g for the schema pet we might have an object pluto.

Metadata

Hydration

The process of transforming incoming data to objects is called hydration.

Security

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions or alterations there.*

We believe in integrating security into the core of our development process. We employ automated penetration testing and scanning as part of our Continuous Integration and Continuous Deployment (CI/CD) pipeline. This approach allows us to identify and address potential security vulnerabilities early, during the development phase, rather than later in the production phase.

Automated Penetration Testing

Automated penetration testing tools are integrated into our CI/CD pipeline to simulate attacks on our systems and identify security weaknesses. These tools conduct a series of tests to check for common vulnerabilities, including those listed in the OWASP Top 10.

The results from these tests are then used to inform our development and security teams about potential vulnerabilities. This process enables us to address these vulnerabilities before the software is deployed to production.

Scanning

Our CI/CD pipeline also includes automated scanning tools that check our source code, containers, and cloud infrastructure for security issues.

Source code scanners analyze our code to find security weaknesses such as those in the OWASP Top 10 list of common security risks. Container scanners inspect our Docker and other container images for vulnerabilities, misconfigurations, and compliance with best practices. This is in line with our commitment to adhere to the top ten containerization security tips. Cloud security scanners ensure that our cloud infrastructure is configured securely, following the principle of least privilege and other cloud security best practices. Adhering to the Top Ten Containerization Security Tips In our commitment to maintain robust security, we adhere to the top ten containerization security tips. Here are some of the practices we follow:

- **Use minimal base images:** We only include the necessary services and components in our container images to reduce the attack surface.
- **Manage secrets securely:** We don't store sensitive information like passwords, API keys, or secret tokens in our container images. Instead, we use secure secrets management tools.
- **Use containers with non-root privileges:** We run our containers as non-root users whenever possible to limit the potential damage if a container is compromised.
- **Regularly update and patch containers:** We keep our containers up to date with the latest security patches.
- **Scan images for vulnerabilities:** As mentioned above, we use automated tools to scan our container images for known vulnerabilities.
- **Limit resource usage:** We use container runtime security features to limit the amount of system resources a container can use.
- **Use network segmentation:** We isolate our containers in separate network segments to limit lateral movement in case of a breach.
- **Implement strong authentication and authorization controls:** We ensure that only authorized individuals can access our containers and the data within them.
- **Monitor and log container activity:** We collect and analyze logs from our containers to detect any suspicious activity.
- **Ensure immutability and maintain an effective CI/CD pipeline:** Our containers are designed to be immutable, meaning they are not updated or patched once they are deployed. Instead, changes are made to the container image and a new version of the container is deployed through our CI/CD pipeline.

By integrating security into our development process, we aim to create a secure, reliable environment for our software and services.

User Authentication

We implement user authentication through oAuth or Active Directory Federation Services (ADFS). ADFS is a software component developed by Microsoft that provides users with single-sign-on access to systems and applications located across organizational boundaries.

Users first authenticate through oAuth/ADFS, which then produces a series of claims identifying the user. These claims are then used by the Open Catalogi application, which uses them to decide whether to grant the user access and roles (See RBAC). This system simplifies the login process for users and allows for secure authentication across different systems and applications.

Identification Based on Two-Way SSL

Identification of other catalogs in our federated network is based on two-way SSL (Secure Sockets Layer) certificates, specifically adhering to the Dutch PKI (Public Key Infrastructure) system. This approach ensures a secure and trusted communication channel between the software and the catalog.

The two-way SSL authentication mechanism requires both the client and the server to present and accept each other's public certificates before any communication can take place. This process guarantees the identity of both the client and server, ensuring a high level of security and trust in the communication.

Role-Based Access Control (RBAC)

Our system implements Role-Based Access Control (RBAC) to manage both user and application rights. RBAC is a method of regulating access to computer or network resources based on the roles of individual users within the organization.

In RBAC, permissions are associated with roles (and configured in our software), and users and other applications are assigned appropriate roles. This setup simplifies managing user privileges and helps to ensure that only authorized users and applications can access certain resources or perform certain operations.

Data Security Levels

Our system handles various types of data, each requiring different levels of security:

- **Public Data:** This data is available to all users and doesn't contain any sensitive information. Even though it's public, we still take measures to ensure its integrity and availability.
- **Data Available to Specified Organizations:** Some data is only accessible to certain organizations. We implement strict access controls and authentication methods to ensure that only authorized organizations can access this data.
- **Data Available Only to the Own Organization:** Certain data is strictly internal and only accessible by our organization. This data is protected by multiple layers of security and can only be accessed by authenticated and authorized personnel within our organization.
- **User-Specific Data:** Some data is personalized and only available to specific users. This data is protected by strong access controls and encryption. Only the specific user and authorized personnel within our organization can access this data.

We take data security very seriously and have implemented robust measures to ensure the safety, confidentiality, integrity, and availability of all data in our system.

Seperating Landing Zone, Execution Zone and Data

In our setup, we utilize NGINX and PHP containers to ensure a clean separation of concerns between internet/network access, code execution, and data storage. This design facilitates robust security and improved manageability of our applications and services.

- **NGINX Containers as Landing Zone:** The first layer of our architecture involves NGINX containers serving as a landing zone. NGINX is a popular open-source software used for web serving, reverse proxying, caching, load balancing, and media streaming, among other things. In our context, we use it primarily as a reverse proxy and load balancer. When a request arrives from the internet, it first hits the NGINX container. The role of this container is to handle network traffic from the internet, perform necessary load balancing, and forward requests to appropriate application containers in a secure manner. This arrangement shields our application containers from direct exposure to the internet, enhancing our security posture.
- **PHP Containers as Execution Zone:** Once a request has been forwarded by the NGINX container, it lands in the appropriate PHP container for processing. These containers serve as our execution zone, where application logic is executed. Each PHP container runs an instance of our application. By isolating the execution environment in this way, we can ensure that any issues or vulnerabilities within one container don't affect others. This encapsulation provides a significant security advantage and makes it easier to manage and scale individual components of our application.
- **Data Storage Outside the Cluster:** For data storage, we follow a strategy of keeping data outside the cluster. This approach separates data from the execution environment and the network access layer, providing an additional layer of security. Data stored outside the cluster can be thoroughly protected with specific security controls, encryption, and backup procedures, independent of the application and network layers.

This three-tiered approach – NGINX containers for network access, PHP containers for code execution, and external storage for data – provides us with a secure, scalable, and resilient architecture. It allows us to isolate potential issues and manage each layer independently, thereby enhancing our ability to maintain and secure our services.

#Sources

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions or alterations there.*

Sources represent places where the Gateway can get its information, typically these would be other APIs, but the gateway can also connect the files servers through (S)FTP and databases directly (MongoDB, PostgreSQL, MySQL, Oracle, and MSSQL), blockchain solutions, or networks of trust (NLX/FCS). The source object represents information about the source (status and transaction logs) as well as the current connection setting. You can find sources through the sources menu item on the left side of the Admin UI or the admin/sources endpoint on the gateway API

.

Adding and testing a source

Sources can be added through the add source button at the top right of the sources overview page.

Let's take a look at configuring a source for the Swagger Pet store. Head over to the sources page and press “add source” on the following page and set the source `location` to https://petstore.swagger.io/v2/pet and pick a name. Since the Swagger Petstore is unconnected, we don’t need additional details and can save our connection by pressing the save button.

After saving our connection the test tab appears below the connection detail page. We can now test the connection to our source. Let's try a request with the GET method to the endpoint `/findByStatus?status=available`. Enter the details in the form and press `test connection`. If the connection test is successful, we should see the result of our test in the right bottom corner Additionally, the status of the connection should update to the last call, and a new call log should be available under the logs tab with the results of our test described.

Now that we tested our connection, we can add the connection to our dashboard by pressing “+ Add to dashboard” Head back over to the dashboard to see the status card of our new connection.

Syncing a source

For this part we assume that you already have made a schema called pet containing the properties `name` and `status`. If you haven't yet done this, follow the steps under schema to create a schema resource accordingly

Exposing a source to applications

todo

(Reverse) Proxy

The easiest way to expose a source to applications is setting up a reverse proxy. A reverse proxy means that all the requests sent to a specific endpoint on the gateway are forwarded to the source, and the response of the source is forwarded to the asking applications.

Setting up a reverse proxy shields the underlying source from the application. The application authenticates itself to the gateway and the gateway then contacts the source. This means that the application doesn’t need authentication or access to the source itself and allows the gateway to monitor the traffic. You can read more about setting up proxies under endpoints.

Datalayer

A different approach to exposing the data (directly) within a source is creating a schema that maps to the source. This method gives a bit more flexibility in the transformer

Synchronizations

> ***Warning*** *This file is maintained at Conduction’s Google Drive Please make any suggestions of alterations there.*

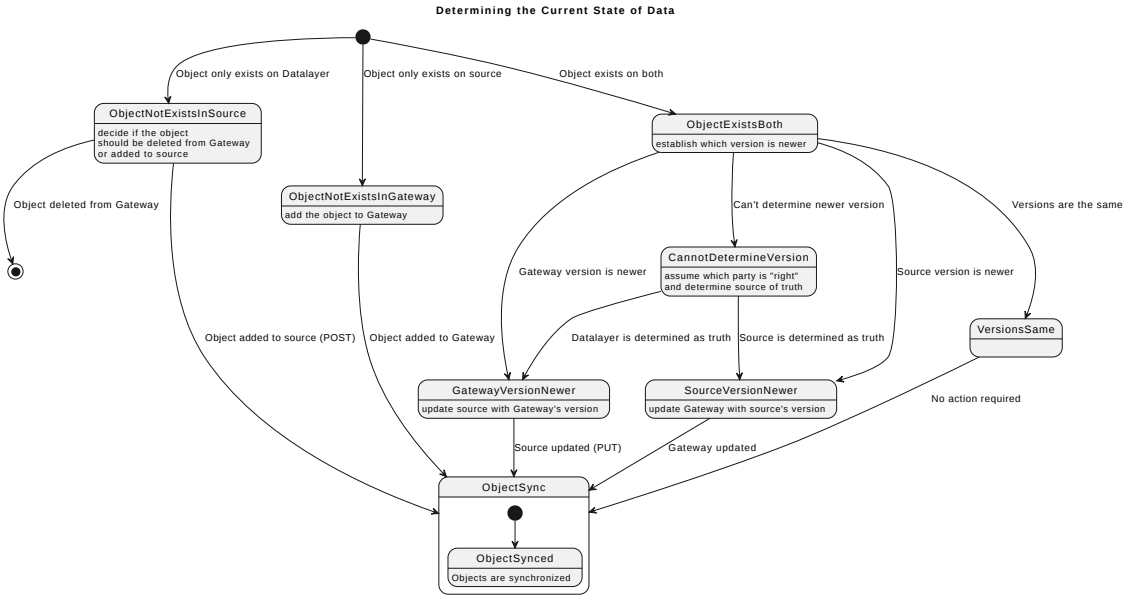
The data synchronization process is an essential part of the Common Gateway data layer. This process ensures that data between the data layer and an external source remains consistent. While it's possible to sync directly between sources, this would necessitate a tripartite setup involving the Gateway.

Determining the Current State of Data

The initial step in the synchronization process involves determining the current state of data. There are three main scenarios:

1. The object exists on the Gateway but not on the source: In this case, we need to decide if the object should be deleted from the Gateway or added to the source, based on the specific rules and constraints of the system.
2. The object exists on the source but not on the Gateway: Here, we usually need to add the object to the Gateway to maintain synchronization.
3. The object exists on both the source and Gateway: In this situation, we need to establish which version is newer and update accordingly. There are three sub-scenarios:
    - The Gateway version is newer: The source should be updated with the Gateway's version.
    - The source version is newer: The Gateway should be updated with the source's version.
    - The versions are the same: No action is required.
    - In instances where we can't establish which version is newer, we must assume which party is "right" and determine a source of truth. Typically, this would be the source, but it can be

configured to be the Gateway.



Creation of a Synchronization Object

The next step is to create a synchronization object that holds and describes the relationship between the two objects (Data Layer and Source). This object includes details such as the IDs on both ends, the dates of changes, and a hash of the source object.

The synchronization object is crucial as it allows us to determine if the source object has changed, even if the object doesn't have a property that enables this detection. By comparing the current hash with the previous hash stored in the synchronization object, we can detect any changes and trigger the necessary updates. This method ensures the consistency of data across the Gateway and the source, enhancing the reliability and integrity of our system.

Twig

> ***Warning*** *This file is maintained at the Conduction Google Drive. Please make any suggestions or alterations there.*