Sage

May 2021 - Working Draft

`Author`
Doruk Eray
Founder and Chief @ Dorkodu.
Self-taught Software Engineer.

Website : doruk.dorkodu.com
Email : doruk@dorkodu.com

Contents

* **WIP :** Work in progress.

1 Introduction

This is the proposal for Sage; a query-based, entity-focused data exchange approach originally created at Dorkodu to simplify the communication for data interactions between different layers of software, especially built for client-server applications. The development of this open standard started in 2020.

The latest working draft release can be found on Sage's website on Dorkodu Libre.

Copyright Notice

Disclaimer

Conformance

A conforming implementation of Sage must fulfill all normative requirements. Conformance requirements are described in this document via both descriptive assertions and key words with clearly defined meanings.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative portions of this document are to be interpreted as described in IETF RFC 2119. These key words may appear in lowercase and still retain their meaning unless explicitly declared as non-normative.

A conforming implementation of Sage may provide additional functionality, but must not where explicitly disallowed or would otherwise result in non-conformance.

Non-normative Portions

All contents of this document are normative except portions explicitly declared as non-normative.

> This is an example of a non-normative explanation, or author's opinion.

Examples in this document are non-normative, and are presented to help understanding of introduced concepts and the behavior of normative portions of the specification. Examples are either introduced explicitly in prose (e.g. “for example”) or are set apart in example or counter-example blocks, like these :

```
// This is an example of a non-normative code sample.
console.log("Hello, World!");
```

Code examples in this document are for providing real-life samples, but does not have to be from a real implementation. We created reference implementations, and recommend you checking out them.

> **Example**
>
> This is an example of a non-normative example.

Notes in this document are non-normative, and are presented to clarify intent, draw attention to potential edge-cases and pit-falls, and answer common questions that arise during implementation. Notes are either introduced explicitly in prose (e.g. “**Note :**“) or are set apart in a note block, like this:

> **Note**
>
> This is an example of a non-normative note.

2 Overview

Sage is a query-based, entity-focused data exchange *(or retrieval)* approach designed to simplify the communication for data interactions between different layers of applications by providing an expressive, intuitive and lightweight way.

The primary goal was to develop a simpler way for inter-layer data interactions, but Sage is designed to be implemented as an isolated data exchange layer, which can also play an **API** role in your architecture.

For example, here is a sample Sage transaction *(request and response for a query)* ***:***

— Query :

```
{
  "forrest": {
    "type": "Movie",
    "attr": ["name", "starring", "releaseYear"],
    "args": {
      "id": 5
    }
  }
}
```

— Result :

```
{
  "data": {
    "forrest": {
      "name": "Forrest Gump",
      "starring": "Tom Hanks",
      "relaseYear": 1994
    }
  }
}
```

In this example, we requested for a **Movie** entity with the argument **id: 5** and wanted attributes of **name**, **starring** and **releaseYear**. And as a result you got an object which contains only what you wanted.

3 Principles

Our priority is to keep Sage simple, approachable, easy-to-use and lightweight while solving data exchange problems efficiently. Here are some of our design principles :

Some of Our Design Principles

- ### Agnostic About Your Application and Technology Stack

  Sage **never dictates** the use of *any programming language, platform, storage technique or even a query language like SQL or GraphQL*. It should be independent from other layers, also directly based on your business logic.

  If we simplify how Sage works, it is like a middleware for data exchange between the layers of consumer (client) and service (server).

- ### Query-based Data Exchange

  Most problems in data exchange is experienced in the retrieval process between layers. *(e.g. client-server applications & APIs)* In order to solve this, Sage is ***query-based***, which is the ideal way. You can query your data, by *giving the desired attributes, and arguments for conditions*, and get only what you want. Even you can call remotely your Sage instance to do some work, by adding an **act** to your query item.

  Any data fetching and modification process imaginable can be developed with Sage. Especially in the product-side client applications, it would be a mindful choice to consume a Sage based API.

- ### Entity-focused Type System

  In modern software world, using data structurally as an "object" is the most common way, so Sage performs operations (e.g. data retrieval or modification) on a specific *"entity type"* . This entity type can be thought as a kind of "class" in OOP, with attributes (= properties) and acts (= methods) are defined and used on this entity type. You query for an object

- **Product-centric**

  Sage was primarily designed to solve the problems of the data consumer, front-end application developers. The main goal was to provide an *intuitive, neat, lightweight and easily applicable framework for simplifying the data exchange process* and ease the burden on developers and their software architecture. For this reason, Sage offers a naturally appropriate way for both client and server sides of your application.

- **Client-first**

  Through its simple but effective type system, a Sage instance publishes its data capabilities in an entity-focused way, which determines what its clients are allowed to consume. It is the client that should be able to specify exactly how it will consume that data. These queries are specified at attribute-level granularity.

  In the majority of client-server applications written without a similar way to Sage, most of which use REST, the server itself determines the data returned in its endpoints. But a Sage API, on the other hand, returns exactly what a client asks for and no more.

  An addition to that, with Sage, you can also define acts on an entity type, which can be called in a query. This is useful when you want to make changes on your data, or trigger your API to do something.

So...

Because of these principles, Sage is a simple-to-use, flexible, lightweight but also powerful and productive way for application-centric data exchange. Product developers and designers can create applications a lot more effectively by working with a Sage API. Sage can quickly make your application stack enjoyable to work with. To enable that experience, there must be those that build those APIs and tools for the rest to use.

> This paper *(or proposal)* serves as a reference for engineers who will develop Sage implementations. It describes the approach, concepts, rules and components. The goal of this document is to provide a foundation and framework for Sage. We look forward to work with the community to improve this approach.

4 Concepts

Here we introduce some concepts and terms which you will need to understand well in order to understand and learn more deeply about Sage. Actually we do this because you may fall in love with Sage in first sight ;)

4.1 Entity

Entities are at the core of the type system in Sage. You define your data as entities. You can think of entities like objects in OOP.

Each entity can have any number of attributes and acts. Attributes are **key-value pairs**, while acts are **remote-callable methods** (which you can trigger them to do something, like updating the database).

- ## Attribute

  An attribute describes a property of this entity type. An attribute can be *any type* which **must be JSON serializable** *(a scalar, or an array/object following this rule)*.

  An attribute will be retrieved by a resolver/retriever function defined by user, which Sage will pass the query object as a parameter.

  > **Note**
  >
  > Sage **does not** dictate that a resolver function will receive the query object as the *ONLY* parameter. If your implementation has additional functionality which requires to pass another parameter, you can do it.
  >
  > **For example:** In our PHP implementation, you can define an optional 'context' array which you can use as a simple dependency container. Sage will pass that context value to resolver and act functions.

  — Here is a pseudo example for defining attributes, in PHP :

```php
/*
 * A psuedo attribute definition, with a string type constraint.
 *
 * Description, constraints and other optional settings should be expressed as a
 * map if possible. Like object literals in JS, or assoc arrays in PHP.
 */

// Attribute(<name>, <resolver>, <options>)
$attribute = new Attribute(
  'name',
  function ($query) {
    $id = $query->argument('id');
    $person = DataSource::getPersonById($id);
    return $person->name;
  },
  [
    'type'        => Type::string(),
    'description' => "Name of a person."
    'nonNull'     => true
  ]
);
```

- ## Act

  An act is a function *(like a method)* that can be added to an entity type. An act is called in a query with *(optional)* arguments.

  An act is identified by a string name. It takes the query as a parameter. You can write your business logic code as acts and run any of them by calling it from a query.

  — Here is a pseudo example for defining acts, in PHP :

```php
/*
 * A psuedo act definition.
 * Options should be expressed as a map, if possible.
 */

// Act(<name>, <closure>, <options>)
$act = new Act(
  "greet",
  function($query) {
    $name = $query->argument('name');
    say("Hi, " . $name);
  },
  [
    'description' => "Greets someone, takes 'name' as argument."
  ]
);
```

4.2 Query

Sage is a query-based approach for data exchange.

The client requests the data it needs from a Sage service with a structured query document that is created according to certain rules. Basically, a query document contains a list of query items all of which describes an entity instance needed, with no limit on the number of query items.

But how to query?

> **Note**
>
> Sage **does not** have a *special query language*.
>
> We think it is *unnecessary* to add this burden while being able to use one of the common formats. So we decided to use ***JSON***, which has advantages such as being *universal, lightweight and easy to use*. Also, JSON is commonly used in the software ecosystem, which means it has wide support *(such as tools, helper libraries)* on different languages and platforms.
>
> Thus, the query and the result are expressed using the JSON format. By using a universal format and not inventing a new query language, we keep Sage lightweight, easily approachable and implementable.

Each query item must be identified with a string name which must be unique within the query document, and expressed as a JSON object which the name points to.

— Below here is an example of a transaction (query and response) requesting a single entity :

```
{
  "doruk": {
    "type": "User",
    "attr": ["name", "email", "age"],
    "args": {
      "handle": "@doruk"
    }
  }
}
```

```
{
  "data": {
    "doruk": {
      "name": "Doruk Eray",
      "email": "doruk@dorkodu.com",
      "age": 16
    }
  }
}
```

Query Item Structure

A query item contains at most 4 attributes. To have lightweight, compact query documents, attribute names are used in their shortened forms. **Type, Attributes, Act and Arguments.**

- `type`

  Entity type. A query item will be executed on its given type. This is the only required attribute.

- `attr`

  (optional) The string array of attribute names you want Sage to return. Each attribute is identified with its own *string name*.

  You can set *"attr"* to an empty array, or even don't define *"attr"* attribute.

  - Empty array means an **empty result** object will be returned.
  - Not setting the attribute means **no result object** will be returned.

- `act`

  (optional) In addition to an entity's attribute getters, Sage also can call for an *act* you defined, if you give its name in the query.

  This is an example query of adding a to-do, for the sake of simplicity :

```
{
  "AddToDo:101": {
    "type": "ToDo",
    "act": "addToDo",
    "args": {
      "userId": 101,
      "title": "Finish Sage's Whitepaper.",
      "deadline": "2021-05-20"
    },
    "attr": ["id", "user", "title", "isCompleted", "deadline"]
  }
}
```

This sample will add a to-do with given arguments, then return the desired attributes. Here is the result :

```
{  "AddToDo:101": {    "id": 12345,    "user": {      "id": "101",
 "username": "doruk",      "name": "Doruk Eray"    },    "title": "Finish
Sage's Whitepaper.",    "isCompleted": false,    "deadline": "2021-05-20"
}}
```

> **Note**
>
> Sage does not handle these steps automatically. It's the developer who must write the code required to add this to-do to data storage, also how to retrieve these attributes.

- `args`

  (optional) Arguments, a list of key-value pairs. They are no different than passing parameters to a function. You are providing arguments to your Sage API to specify "how" you want the data. They will be passed to all attribute getters *(and to the act if given in the query.)*

  > For example, let's say you want the `ToDo` with the `id` of `1234`. If so, while you are querying you can give an `id` argument and set it to `1234`. On the other hand, in the resolver you would look for an id argument and fetch the User with the given id from the data source.

4.3 Schema

Your Sage server's data capability is defined by its data schema, which is just a list of entity types. That's it. A list/array of all entity types you want to be available. Schema will be passed to Sage's query executor. Any query document given to the execution engine will be run on this schema you define.

In this section we only introduced some concepts. You can find more details about components of Sage in the following sections of this document.

5 Components

Sage consists from following components :

- Type System
- Introspection
- Validation
- Execution
- Response

5.1 Type System

The Sage Type system describes the capabilities of a Sage server and is used to determine if a query is valid. The type system also can be used to add strict type constraints on attributes to determine that values provided at runtime are valid an of a desired.

Schema

A Sage service's data capabilities are referred to as that service's "*schema*".

A schema is defined as a list of entity types it supports.

A Sage schema must itself be internally valid.

All entity types within a Sage schema must have unique, string names. No two provided entity types may have the same name. No provided type may have a name which conflicts with any built in types (including Scalar and Introspection types).

All items *(entities, their attributes and acts)* defined within a schema must not have a name which begins with '@' *(at symbol)*, as this is used exclusively by Sage's introspection system.

Types

The fundamental unit of any Sage schema is the *type*.

The most basic type is a `Scalar`. A scalar represents a primitive value, like a string or an integer.

However, we have a concept called **"constraints".** The most important constraint is **strict-types**. Oftentimes it is useful to add some constraints to attributes, like **strict-types**. For example, strict-type constraint allows the schema to specify exactly which data type is expected from a specific attribute. With Sage; you can do that, too.

Scalar Types

Scalar types represent primitive values in the Sage type system.

All Sage scalars are representable as strings, though depending on the response format being used, there may be a more appropriate primitive for the given scalar type, and server should use those types when appropriate.

> We prefer **JSON** and suggest you to use JSON if possible, but you can also use another format for query and/or result, *in the same way we use JSON*.
>
> In the **"Response"** section, we will mention this topic.

Result Coercion

A Sage server, when retrieving an attribute of a given scalar type, must uphold the contract the scalar type describes, either by coercing the value or producing an **attribute error** if a value cannot be coerced or if coercion may result in data loss.

A Sage service may decide to allow coercing different internal types to the expected return type. For example when coercing a attribute of type `int` or a `boolean` true value may produce `1` or a string value `"123"` may be parsed as base-10 `123`. However if internal type coercion cannot be reasonably performed without losing information, then it must raise an **attribute error**.

Since this coercion behavior is not observable to clients of a Sage service, the precise rules of coercion are left to the implementation. The only requirement is that a Sage server must yield values which adhere to the expected Scalar type.

Default Scalar Types in Sage

Sage supports a basic set of well-defined Scalar types. A Sage server should support all of these types, and a Sage server which provide a type by these names must adhere to the behavior described below.

> These scalar types are used for strong-type constraints on attributes. Normally attributes are all weak-typed, which means can be any type, provided that they must be JSON serializable.

Integer

The integer scalar type represents a signed 32-bit numeric non-fractional value. Response formats that support a 32-bit integer or a number type should use that type to represent this scalar.

Result Coercion

Attributes returning the **integer** type expect to encounter **32-bit** integer internal values.

Sage servers may coerce non-integer internal values to integers when reasonable without losing information, otherwise they must raise an **attribute error**. Examples of this may include returning `1` for the floating-point number `1.0`, or returning `123` for the string `"123"`. In scenarios where coercion may lose data, raising an attribute error is more appropriate. For example, a floating-point number `1.2` should raise an attribute error instead of being truncated to `1`.

If the integer internal value represents a value less than -2^{31} or greater than or equal to 2^{31}, an attribute error should be raised.

> **Note**
>
> Numeric integer values larger than 32-bit can use string type, as not all platforms and transports support encoding integer numbers larger than 32-bit.

Float

The Float scalar type represents signed double-precision fractional values as specified by IEEE 754. Response formats that support an appropriate double-precision number type should use that type to represent this scalar.

Result Coercion

Attributes returning the **float** type expect to encounter double-precision floating-point internal values.

Sage servers may coerce non-floating-point internal values to **float** when reasonable without losing information, otherwise they must raise an *attribute error*. Examples of this may include returning `1.0` for the integer number `1`, or `123.0` for the string `"123"`.

String

The string scalar type represents textual data, represented as UTF-8 character sequences. The string type is generally used by Sage to represent free-form human-readable text. All response formats must support string representations, and that representation must be used here.

Result Coercion

Attributes returning the string type expect to encounter UTF-8 string internal values.

Sage servers may coerce non-string raw values to string when reasonable without losing information, otherwise they must raise an attribute error. Examples of this may include returning the string `"true"` for a boolean true value, or the string `"1"` for the integer `1`.

Boolean

The Boolean scalar type represents `true` or `false`. Response formats should use a built-in boolean type if supported; otherwise, they should use their representation of the integers `1` and `0`.

Result Coercion

Attributes returning the **boolean** type expect to encounter boolean internal values.

Sage servers may coerce non-boolean raw values to `boolean` when reasonable without losing information, otherwise they must raise an attribute error. Examples of this may include returning `true` for non-zero numbers.

Entity

Sage Entities represent…

- a list of named attributes *(= fields, properties)*, each of which yield a value *(optionally, a value of a specific type)*
- a list of named acts (= methods) each of which are functions that you can call from your query item *(and optionally with the arguments you give)*.

Entity values should be serialized as maps, where the queried attribute names are the keys and the result of evaluating the attribute is the value.

All attributes and acts defined within an Entity type must not have a name which begins with "@" (at symbol), as this is used exclusively by Sage's introspection system.

> **Note**
>
> Sage queries are not hierarchal. You request for entities individually, but you can also compose different entity types in one manually. It's completely up to the end user who will develop a Sage service.
>
> We wanted to handle every single entity separately. By doing so we try to provide as much granularity as possible. This becomes very useful if you think in terms of a *"knowledge graph"*, where you don't embed relationships with other entities, instead you just link to them.
>
> We develop Sage with the future of Web in mind, not just for today's hot fashions. As Dorkodu our primary interests are *Web 3.0 (Semantic Web), Information Science and Linked Data*.
>
> We want Sage to be the data exchange protocol of future.

For example, a `Person` entity type could be described as **:**

— just as an example, in a hypothetical format **:**

```
entity Person { id: @integer  name: @string age: @integer}
```

Where `name` is an attribute that will yield a **string** value, while `age` and `id` are attributes that each will yield an **integer** value.

> Do not forget that strict-types are optional. You don't need to set a type constraint for each attribute you define.

Only attributes and acts which are declared on that entity type may validly be queried on that entity.

For example, selecting all the attributes of a `Person` **:**

```
{ "someone": {    "type": "Person",    "attr": ["name", "age"],    "args": {
 "id": 10    }  } }
```

Would yield the object:

```
{  "someone": {   "name": "Doruk Eray",   "age": 17,  }}
```

While selecting a subset of attributes:

```
{ "someone": {    "type": "Person",    "attr": ["name"],    "args": {      "id": 10    } } }
```

Must only yield exactly that subset:

```
{  "someone": {   "name": "Doruk Eray", }}
```

We see that an attribute of an entity type may be a scalar type, but it can also be an **list** or **entity**.

For example, the `Person` type might include an `occupation` attribute with the type *object* :

```
entity Person {  id: @integer  name: @string  age: @integer  occupation: @entity(User)}
```

And let's say we only requested for `occupation` attribute. Here it returns an *object* :

```
{  "someone": {   "occupation": {     "company": "Dorkodu",     "role": "Founder",      "startYear": 2017     } }}
```

Type Validation

Entity types can be invalid if incorrectly defined. These set of rules must be adhered to by every Entity type in a Sage schema.

1. An Entity type must define one or more attributes.
2. For each attribute of an Entity type :
    1. The attribute must have a unique name within that Entity type; no two attributes may share the same name.
    2. The attribute must not have a name which begins with the character "@" *(at)*.
    3. The attribute must return a type which must be **output-able**. We will talk about this later.
3. For each act of an Entity type :
    1. The act must have a unique name within that Entity type; no two acts may share the same name.
    2. The act must not have a name which begins with the character "@" *(at)*.
    3. The act must be a callable *(function, method, closure etc.)*, and accept at least one parameter, which is the query object.

List

A Sage list is a special collection type which declares the type of each item in the List (referred to as the *item type* of the list). List values are serialized as ordered lists, where each item in the list is serialized as per the item type.

To denote that a field uses a List type, the item type also must be declared as a type constraint.

— For example, here we define an attribute which is a string list :

```
/*
 * A psuedo attribute definition, with a list type constraint.
 */
// Attribute(<name>, <resolver>, <options>)
$attribute = new Attribute(
  'names',
  function ($query) {
    // This will return an array of strings
    return DataSource::getNames();
  },
  [
    'type'        => Type::listOf( Type::string() ),
    'description' => "Names list."
  ]
);
```

Result Coercion
Sage servers must return an ordered list as the result of a list type. Each item in the list must be the result of a result coercion of the item type. If a reasonable coercion is not possible it must raise an attribute error. In particular, if a non-list is returned, the coercion should fail, as this indicates a mismatch in expectations between the type system and the implementation.

If a list's item type is nullable, then errors occuring during preparation or coercion of an individual item in the list must result in a the value **null** at that position in the list along with an error added to the response. If a list's item type is non-null, an error occuring at an individual item in the list must result in an attribute error for the entire list.

For more information on the error handling process, see “Errors and Non-Nullability” within the Execution section.

Constraints

Strict-type

All attributes are weak-typed by default. This means they can be any type which must be JSON serializable.

But optionally you can set strict-type constraints for an attribute. This means, the resolver function of that attribute must return a value of that specific type you want.

Anyway, Sage will try to coerce the returned value to the desired type if possible.

These are all possible types which you can set as a strict-type constraint **:**

- **boolean**
- **integer**
- **string**
- **float**

- **entity** (must set a specific entity type)
- **list** (must set an item type)

Non-null

By default, all types in Sage are **nullable**; which means the **null** value is a valid response for all of the above types. To declare a type that disallows null, the Sage Non-null constraint can be used. This constraint wraps an underlying type, and acts identically to that wrapped type, with the exception that **null** is not a valid response for the wrapping type.

> **Example**
>
> Think about the '**age**' attribute of a '**Person**'. In real life; it is an *integer*, and *non-null*.
>
> If you set these constraints for *'age'* attribute, it must return a non-null, integer value.

Descriptions

Documentation is a boring part of API development. But it turned about to be a killer feature when we decided that any Sage service should be able to publish a documentation easily.

To allow Sage service designers easily write documentation alongside the capabilities of a Sage API, descriptions of Sage definitions are provided alongside their definitions and made available via introspection. Although descriptions are completely optional, we think it is really useful.

All Sage types, attributes, acts and other definitions which can be described should provide a description unless they are considered self descriptive.

Deprecation

Attributes or acts in an entity may be marked as *"deprecated"* as deemed necessary by the application. It is still legal to query for these attributes or acts (to ensure existing clients are not broken by the change), but they should be appropriately treated in documentation and code.

This must be handled just like setting optional constraints on attributes. As simple as declaring the '**deprecated**' setting as **true**.

5.2 Introspection

A Sage server supports introspection over its schema. The schema is queried using Sage itself, which makes it easy-to-use and flexible.

Take an example query, there is a User entity with three fields: id, name, and age.

— for example, given a server with the following type definition :

```
entity User {
  id: @integer @nonNull
  name: @string @nonNull
  age: @string
}
```

The query

```
{
  "introspectionSample": {
    "type": "@entity",
    "attr": [ "name", "attributes", "description", "isDeprecated" ],
    "args": {
      "name": "User"
    }
  }
}
```

would return

```
{
  "introspectionSample": {
    "name": "User",
    "attributes": [
      {
        "name": "id",
        "type": "@string",
        "nonNull": true
      }
    ],
    "description": "The user entity type.",
    "isDeprecated": false
  }
}
```

Reserved Names

Entity types, attributes and acts required by the Sage introspection system are prefixed with "@" *(at symbol)*. We do this in order to avoid naming collisions with user-defined Sage types. Conversely, type system authors must not define any entity types, attributes, acts, arguments, or any other type system artifact with a leading '@' *(at symbol)*.

Documentation

All types in the introspection system provide a `description` attribute of type **string** to allow type designers to publish documentation in addition to data capabilities.

Deprecation

To support the effort for backwards compatibility, any piece of Sage type system (entity type, attribute and act) can indicate whether or not they are deprecated (**isDeprecated :** *boolean*) and a description of why it is deprecated (**deprecationReason :** *string*).

Tools built using Sage introspection should respect deprecation by discouraging deprecated use through information hiding or developer-facing warnings.

Schema Introspection

The schema introspection system can be queried using its schema. The user of a Sage implementation doesn't have to write this schema. It must be built-in and available.

— The schema of the Sage introspection system, written in our “fictitious" pseudo schema definition language :

```
entity @schema {
  entities: @list("@entity")
}

entity @entity {
  name: @string @nonNull
  description: @string
  attributes: @list("@attribute") @nonNull
  isDeprecated: @boolean
  deprecationReason: @string
}

entity @attribute {
  name: @string @nonNull
  description: @string
  type: @typekind
  nonNull: @boolean
  isDeprecated: @boolean
  deprecationReason: @string
}

entity @act {
  name: @string @nonNull
  description: @string
  isDeprecated: @boolean
  deprecationReason: @string
}
```

6 Validation

— Work in progress.

7 Execution

— Work in progress.

8 Response

— Work in progress.

9 Notes

This is the notes section for authors' opinion about non-normative parts of Sage. Some “must” or “must not” have features or the philosophy behind Sage are mentioned here. We will publish a separate notes document once publish this paper.

Sage is not a wrapper, but a middle layer.

Sage should only be positioned as a “middleware” data exchange layer, not a wrapper around a whole service stack.

Some Features We Want

- **Declarative :** It must be the data consumer who declares what will they get. It must be the data service (or API) who declares its capability with a data schema.
- **Graph-like :** Data should be described like a graph. Instead of behaving like a giant data document, Sage behaves like a universal entity graph. Objects *(with properties and methods)* are at the core. You can interact with a Sage service like querying a virtual object data store.
- **Weak-typed :** Type restrictions on attributes should be optional, and a decision up to the developer. Not every definition needs strict types. With Sage, all attributes are weak-typed by default.

Nullable Attributes

Any attribute is nullable by default. This is a golden rule which gives Sage one of its key strengths. When something goes wrong while retrieving an attribute, just return null and explain what happened in an additional section in response document. It's not useful to abort and ignore the whole progress.

10 Reference Implementations

To clarify the desired and ideal outcome of this proposal, we built reference server and client implementations. Both of them should be ready-to-go and will be (or is) used on the production at Dorkodu.

- **Sage Server**

  You can see here the reference server implementation written in PHP.

  `GitHub` dorkodu/sage.php

- **Sage Client**

  You can see here the reference client implementation written in JavaScript.

  `GitHub` dorkodu/sage.js

  > **Note**
  >
  > The Sage Proposal does not focus on the client, and dictates no certain rules. However, we have built a web client with JavaScript, for our own needs. And it can be considered as a *"reference"* for the community.

11 Conclusion

In this paper, we present Sage, which is in simple terms, a new way for data retrieval. We first give an overview, then introduce our design principles, concepts behind Sage; after that we dive deeply into the components, then share some notes you must read and understand before implementing specific versions of Sage.

This paper can be interpreted as a whitepaper, or a proposal for the idea, or a not-officially-standardized specification.

12 References

- **Sage**

  Sage's website on Dorkodu Libre. There you can find more resources about Sage, and also the latest working draft of this paper.

- **Dorkodu**

  Who we are? We want to make human knowledge open, useful and meaningful for everyone. We hope you will hear more about us soon. If you are interested in our work, you should see our website.

- **Dorkodu Libre**

  Dorkodu Libre, the website for our contributions we make to the software community. You can find other awesome projects we work on. We try to do useful and meaningful things and make them open; then share our humble contributions with the community through open source software.

- **Dorkodu on GitHub**

  You can find our open source project repositories on GitHub.

- **GraphQL**

  I think I owe a thank to the GraphQL community. What they did was really exciting and changed the mindset of the industry about approaching to a fresh way of doing things.

  Also while writing this document, I was *heavily inspired by* their specification, especially for the document structure. I am *literally* just a kid, so it was really hard for me to do these boring paperwork before writing an actual implementation.