XUA

A PHP CODE GENERATOR

WINTER 1399

KAMYAR MIRZAVAZIRI

TABLE OF CONTENTS

PREFACE;

AN INTRODUCTION TO XUA

Introduction

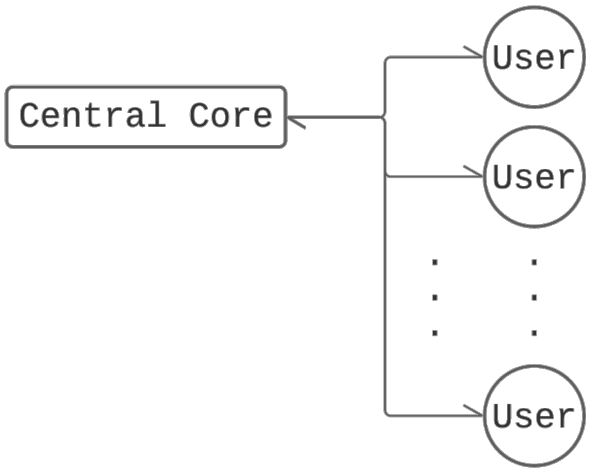
Xua is shortened form of Xuarizmi, named after al-Khwarizmi (Persian: خوارزمی), a Persian mathematician, astronomer, geographer, founder of algebra (The term algebra itself comes from the title of his book "al-jabr" meaning completion or rejoining), and head of the library of the House of Wisdom in Baghdad. His name was formerly Latinized as Algoritmi (and yes, that's where the word algorithm comes from).

Xua is a Code Generator Tool, mainly applicable for creating Web-Services and, in some cases, for creating complete Web-Sites (including back-end and front-end). Xua aims to generate all trivial and technical codes for us, so we only write codes that require human intelligence. In other words, the only thing that Xua does not do is its name. We need to give it algorithms, and Xua does the rest by itself.

Xua generates server codes in PHP language. While a full and complete PHP Project can be generated by Xua, it is not possible to get a complete front-end project; instead, Xua can generate a front-end library (Marshal Library) that can make the connection between client and Xua server faster, more secure, more reliable, and easier to implement.

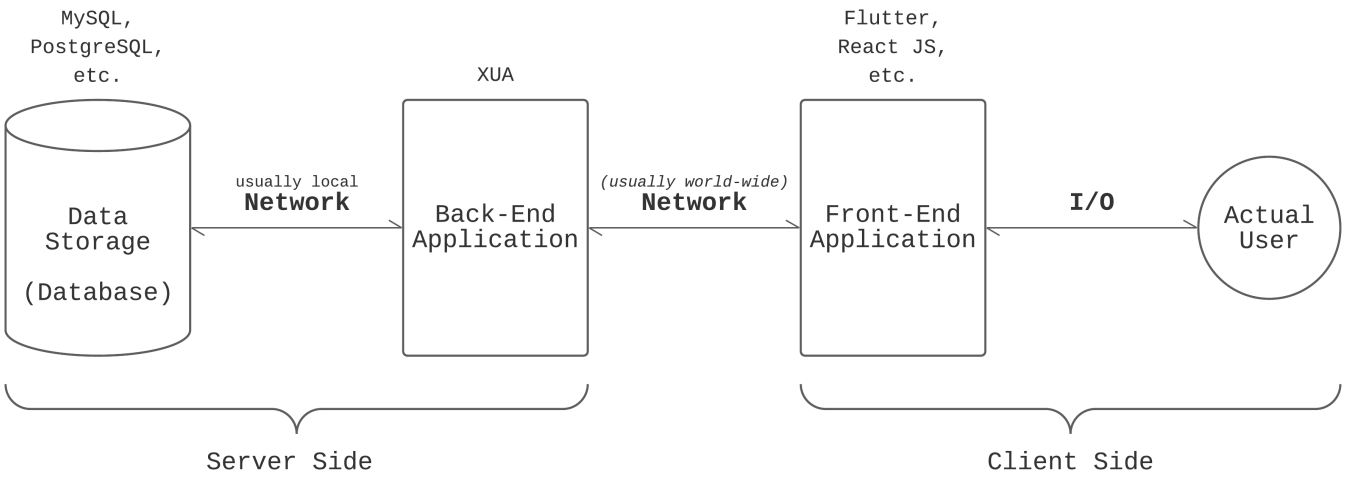
Architecture

Generally, most multi-user applications (such as social networks, messengers, online games, websites) have the following architecture (except for those which are peer-to-peer or using chains and other similar concepts).



This architecture is called client-server. The shared data is stored in a central core (usually called the server), and users (usually called clients) can read, modify, and delete specific data based on their permissions. From Xua's perspective, we can think of the connection between the data

storage and each user as a series of interfaces, each one communicating with the next and previous interfaces on a specific platform.



According to this principle, both back-end and front-end applications are nothing more than interfaces between the actual user and the data storage.

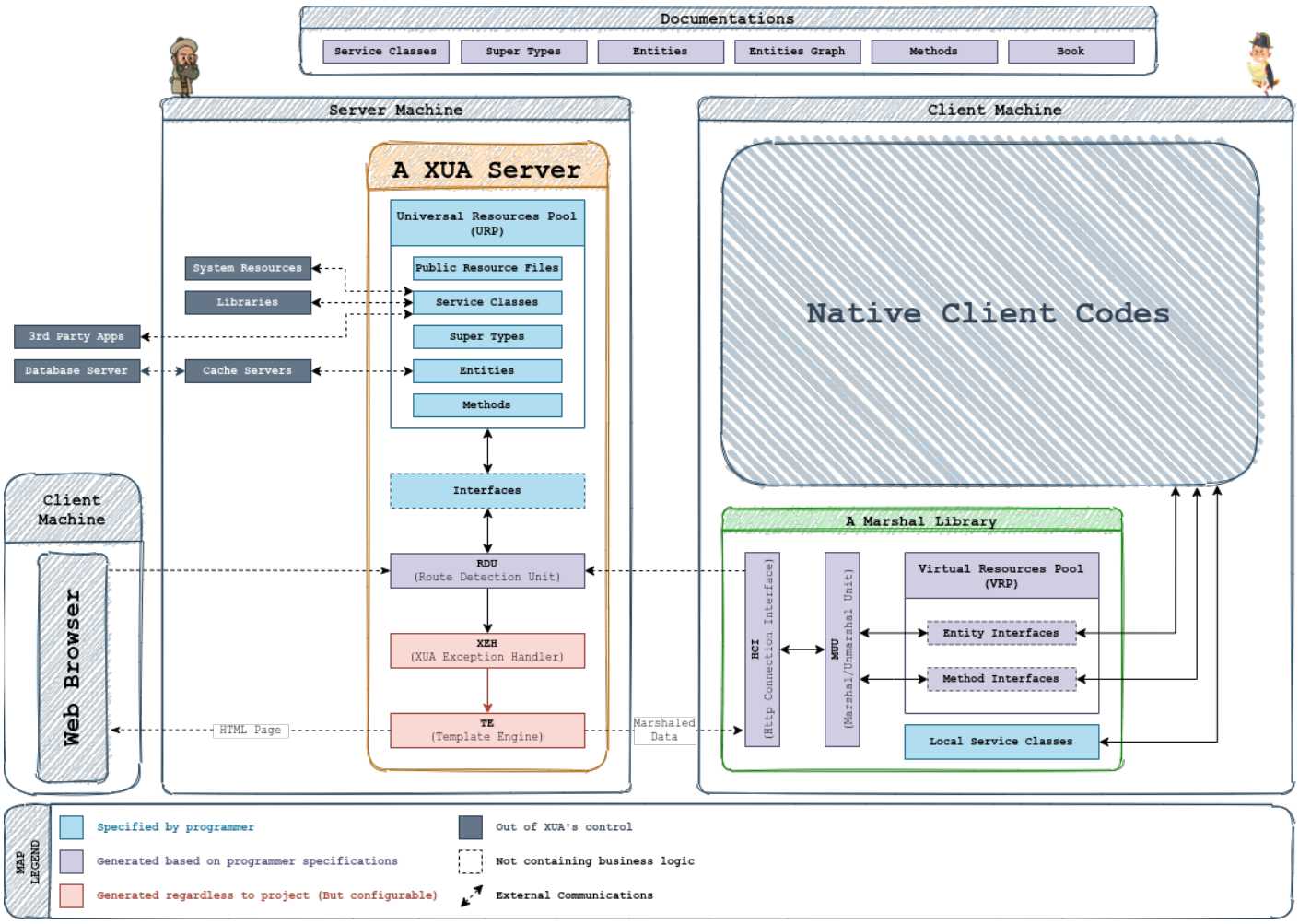
The Triplet of Xua is three blocks: Method, Entity, and Super.The Method blocks are the ones that handle communication with the front-end application, and the Entity blocks are the ones that handle communication with the Database. Super blocks (actually Super-Type blocks) are one level lower than Method and Entity blocks; Supers are not explicitly used in communications but instead are used in defining Methods and Entities and are a fundamental part of Xua as they do all of the type-checkings, validation, and marshaling and unmarshaling the data.

There is also a fourth block called Interface, which is in the frontline of the back-end application. There is a specific interface called URPI (Universal Resource Pool Interface) responsible for connecting its requests to Methods, but the programmer can define other interfaces as well and fill them with pure PHP codes. There is also a file called `routes.xrml` (`xrml` stands for Xua's routes minimal language) which defines that what Interface should be called based on the request method (such as `POST`, `GET`, `OPTIONS`) and the request URI, which are HTTP protocol concepts (which is the protocol Xua and front-end communicate with).

Xua advocates the single block policy. A `.xua` file can contain one block at most, and the block can be Method, Entity, Super, or Interface. However, having more than one block in a file is permitted only for documentation purposes, and it should not affect the server project. So there are only four types of `.xua` files. One can also inject native scripts under the name of Services and assets and other files (under the name of Resource Files) into the project.

So a Xua project is a combination of specifications (**Methods**, **Entities**, **Supers**, **Interfaces**, **Services**, and **Resource Files** to be specific) that will result in generating a server project (known as Xua Server), a front-end library (known as Marshal Library), and a complete set of documentations (known as Documentations).

The detailed Xua architecture is a little frustrating compared to the simplified version we have just provided, but we only mention it here. There is no need to master it entirely for now.



The blue units are the ones mentioned above, and the programmer needs to specify them directly. The RDU is generated based on the file `routes.xrml`, and MUU is generated based on Super blocks. So a Xua project is simply a set of files specifying this stuff. All these lead us to the following structure of a Xua project.

```
Xua Project Root
├─ Methods
│  └─ (.xua files)
├─ Entities
│  └─ (.xua files)
├─ Supers
│  └─ (.xua files)
├─ Interfaces
│  └─ (.xua files including HomeInterface.xua, TestInterface.xua,
│      and UniversalResourcePoolInterface.xua)
├─ Services
│  ├─ Server
│  │  ├─ PHP
│  │  └─ └─ (.php files)
│  ├─ Marshal
│  │  ├─ JavaScript
│  │  └─ └─ (.js files)
│  └─ (...)
├─ (Resource Files ...)
```

```
├─ routes.xrml
└─ config.xml
```

But this is the default structure; the programmer can change it by modifying `config.xml`.

In this documentation, we try to cover all of these units. By visiting the page Architecture and clicking on a unit, you will be redirected to the documentation section of that unit. Note that the Units documentation mainly contains information about why a unit exists and what it does in a more theoretical way. For details on how (instead of why) to create a Xua block (syntax), read the corresponding chapter.

Practicalities

System Requirements

Currently, Xua is only available on Linux.

- `python3` along with `pip` are required to install `xua` package.

- The PHP Engine and a PHP Server is recommended for test and debug on the local server.

- MySQL Server is recommended for test and debug on the local server.

- DataGrip is recommended for monitoring data to test and debug.

- Visual Studio Code is recommended with Xua extension installed to code in Xua language. Although it is unnecessary, the extension will help with code correction and autocomplete and make the build processes easier.

Getting Started

Installation

Installing the VSCode Xua extension will automatically install requirements and the Xua CLI tool. Still, if you're not willing to use VSCode, you can install `xua` using `pip` from that.

- Install `pip`

```
sudo apt install python3-pip
```

- Add `pip` packages directory to `PATH`

```
if [[ ":$PATH:" != *":/home/ubuntu/.local/bin:"* ]]; then PATH="/home/ubuntu/.local/bin${PATH:+":$PATH"}"; fi
```

- Install `xua`

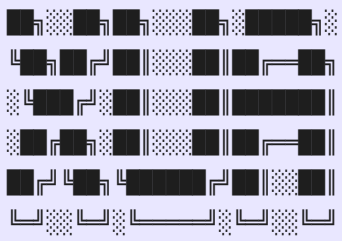
```
pip install -e git+https://github.com/kmirzavaziri/xua-cli/#egg=xua
```

You can check the `xua` is installed correctly by running the command

```
xua --version
```

it should output something like this

```
      Xua: A PHP Code Generator



               1.0-β
```

Hello World!

To create a new project, go to a directory you want to create the project and run

```
xua new project PROJECT_NAME
```

Now go to the project directory and build the template project you just created by running

```
cd PROJECT_NAME
xua build server/ph
```

or alternatively, click on the build button that appears in VSCode after installing the plugin.

Set up a PHP server on the build destination location (`build/php/` by default)

```
cd build/php
php -S localhost:8000 main.php
```

or alternatively, click on the run server button that appears in VSCode after installing the plugin.

Now, you should be able to see the Hello World! page by visiting the localhost:8000 page on your browser.

UNITS;

THE FUNDAMENTALS

Introduction

In this chapter, we try to focus on concepts and theories. Reading this chapter helps with understanding the concepts and fundamentals, but turning these theories into practice, needs reading the main chapter of the unit. Only a few units are directly defined by the programmer (namely: Methods, Entities, Supers, Interfaces, Services, Resource Files), and there is a special chapter for each, which you can reach by clicking on them.

Xua Server

Xua generates three projects. Xua Server, Marshal Library, and Documentations. Among these projects, the server is the most important one, which contains all the units that serve logic to the whole project. The role of the server is to be an interface between the front-end application and the database. The server is responsible for reading and modifying the database based on the requests it receives.

Xua Server Directly Defined Units

There are some units in the server project that need a direct definition from the programmer. These units are shown in blue on the architecture page.

Universal Resources Pool

A Xua Server contains a pool of resources that are available for use. These resources can be available universally, i.e., foreign parties can use resources, and internally, i.e., available for internal resources and units. The idea is similar to a class's private and public visibility, but a public class is only accessible from other codes on the same project, while a universal resource is accessible from other projects, even on other machines other than where the server project is stored. Xua resources are divided to Methods, Entities, Supers, Services, and Public Resource Files.

Methods

(main chapter)

Methods are the central part of a Xua Project. A method is a minimal unit that does a particular job when called. One can think of methods as just functions, but there are two differences. A method can be called from outside of the project and can return multiple values.

Methods are simply triplets $M = (Q, R, B)$. The first member, `Request`, shown by Q, defines the structure of request that needs to be prepared and given to the method, and the second one, `Response`, shown by R, defines the structure of response that the method returns. The third member, `Body`, shown by B, is a script executed when the method is called.

Xua will generate a PHP class indirectly extending a Xua abstract class called `MethodEve` for each Entity Block the programmer creates. One can call these Method classes using their constructor. For example

```
$response = new SomeMethod($request);
```

is a PHP script that calls `$SomeMethod` with request `$request` provided and stores the result in `$response`.

Entities

(main chapter)

A Xua project can communicate with a database server (usually MySQL), but the programmer must specify the database structure. To do so, Xua offers Entity Blocks. Xua will generate a PHP class extending a Xua abstract class called `Entity` for each Entity Block the programmer creates. These Entity classes are in a one-to-one correspondence with database tables. Each row of the table then can be corresponded by an instance of the table's corresponding class.

To define an Entity, the programmer needs to define a list called `Fields` containing Entity columns with their types. Setting the `Fields` is mandatory, but there are some optional members an Entity may have.

One member is a list called `Indexes`, which defines the database indexes, that would make select queries faster or force unique values (Read more about database indexes by googling).

Another is a boolean function (predicate) called `Validation`, which checks if an Entity instance (a row) is valid according to business logic (part of the program that encodes the real-world business rules that determine how data can be created, stored, and changed). For example, assume we have two fields, `country` and `city` in the Entity `User`. The Validation function is responsible for checking if the city is inside the country and avoid invalid or inconsistent data.

Therefore, an Entity is theoretically a triplet $E = (F, I, V)$. Although one can override database communication methods (functions responsible for reading or modifying data), that is a technical detail, and we do not cover it in this section.

Super Types

(main chapter)

Super Types or simply Supers are parametric types.

Let us talk about types first. One can think of a type as a set, for example the type `Integer` is actually the set $\mathbb{Z} = \{0, 1, -1, 2, -2, \dots \}$ and when we say that 42 is of type `Integer`, this means that $42 \in \mathbb{Z}$, which is written $42 : \mathbb{Z}$ in Xua's syntax. We know that there is a one-to-one correspondence between sets and predicates. Having a predicate P, we can define the corresponding set to be $S = \{ x | P(x) \}$, and having a set S, we can define the corresponding predicate to be $P(x) := x \in S$. Since implementing predicates as boolean functions is more practical in computers, we use predicates to define types.

However, there are some features that types usually have that sets do not. In Xua's definition of type, a type is a set equipped with three functions `NativeType`, `Marshal`, and `Unmarshal`. We express a type by the 4-tuple (P, Y, M, U), where P is the characteristic function of the type's set. These functions make a type dependent on the target language. For example, if we are willing to use the integer type in Java language, we need to write these functions in Java language.

Note that predicate can only be defined in the PHP language. But others are stack-based, meaning that they can be written in any of Xua's supported languages and also for the database.

The Native Type function must return a string literal declaring the value type in the native language's syntax. For example, if the type accepts date-times and null value, it must return `?DateTime` for PHP language and `DATETIME NULL` for MySQL.

Note. The database functions are written on PHP syntax, for example, for the MySQL Native Type one can write

```
Type<Database>{
    return "DATETIME NULL";
}
```

The Marshal and Unmarshal functions are responsible for preparing the values for transmitting or storing. To store a value in the database and read from it, one must write marshaling functions with the database as the target, and for transmitting over the HTTP, PHP and front-end languages are used as the target.

As an example (that is not quite practical and efficient), let T be the type of integers defined for the MySQL stack. We know $P(42)$ is true; therefore, we can say $42 : T$. It seems that M must be U^{-1}, but it is not a necessity. Let us say that M takes an integer and returns its binary representation. Therefore $M(42) = '101010'$. The U must be the opposite, so $U('101010') = 42$, but what is $U('42')$, the decimal representation in a string. Can we say this is undefined and the domain of U only accepts binary strings? It is up to the programmer that $U = M^{-1}$ or not, so the programmer needs to define both M and U functions separately. Finally, we can say T must return the string `VARCHAR(100)`. Because the result of our marshal function is a string, and Xua uses that to store data in the database.

Now, what is a Super Type? A Super Type is a function that takes some parameters and returns a type (Actually returns a list of types for different stacks). A Super Type is an object a whole level above a type. Each Super Type eventually results in a PHP class extending a Xua abstract class called `Super`. Each instance of this class is a type (possible to use different stacks on it), and the class constructor is the function that takes some parameters and creates a type. The parameters are then accessible as attributes. For example, take a look at `Enum` that takes one parameter called `values`.

```
$genderType = new Enum([
    'values' => ['male', 'female']
]);
```

The parameter `values` is set to array `['male', 'female']` and the variable `$genderType` is now a type that has the 4 discussed members.

P, the first member, is called `accepts`.

```
var_dump($genderType->accepts('male')); \\ dumps true
```

Since $\mathsf{male} \in \{\mathsf{male}, \mathsf{female}\}$.

Now note that for the stack-based functions, the server project includes only two stacks, database, and PHP. Functions for other stacks are included in the corresponding Marshal Libraries.

So there must be two Y's, in the PHP generated class. The database-targeted function is called `databaseType`, and the PHP-targeted function is called `phpType`.

```
var_dump($genderType->databaseType()); \\ dumps 'INT'
var_dump($genderType->phpType()); \\ dumps 'int'
```

Similar to Native Type, there are two Ms `marshal` and `marshalDatabase`, but for simplicity assume they obey the same algorithm, which is returning the index of the element in array `values`.

```
var_dump($genderType->marshal('male')); \\ dumps 0
var_dump($genderType->marshal('female')); \\ dumps 1
```

At last, U functions `unmarshal` and `unmarshalDatabase`.

```
var_dump($genderType->unmarshal(1)); \\ dumps 'female'
```

Note. The marshal and unmarshal functions of Xua's official Enum are both the identity function; this is just an example.

There is also another member of a type that returns the parameters given to it when constructed.

```
$genderType->parameters();
```

Will return the following map.

```
[
    'value' => ['male', 'female']
]
```

A Little Formalism on Super Types

The set of boolean values, i.e., $\{\mathsf{0}, \mathsf{1}\}$ is shown by $\mathbb{B}$.

The set of all single-argument predicates is shown by $\mathbb{P}$, all Native Type functions by $\mathbb{Y}$, all marshal functions by $\mathbb{M}$, and all unmarshal functions by $\mathbb{U}$.

The set of all types (the so-called 4-tuples) is shown by $\mathbb{T}$.

A simple observation is that $\mathbb{T} = \mathbb{P} \times \mathbb{Y} \times \mathbb{M} \times \mathbb{U}$.

A Literal Name is a literal string containing only alphanumeric characters, starting with a lowercase character. The set of all Literal Names is shown by $\mathbb{L}$.

The set of all values possible to store in a variable is shown by $\mathbb{X}$.

A Dictionary is a partial function $D: \mathbb{L} \to \mathbb{X}$ and is usually expressed by all of its records. The set of all dictionaries is shown by $\mathbb{D}$.

For example, Let D be a dictionary with domain $\{\mathsf{name}, \mathsf{gender}\}$, $D(\mathsf{name}) = \mathsf{Kamyar}$, and $D(\mathsf{gender}) = \mathsf{male}$. We can express D as `{name: 'Kamyar', gender: 'male'}`.

The set of all stacks (PHP, Database, Dart, JS, Java, etc.) is shown by $\mathbb{K}$.

So we can formally say that a Super is a function $S: \mathbb{K} \times \mathbb{D} \to \mathbb{T}$.

But we technically define a super to be the 5-tuple $\mathfrak{S} =(\mathfrak{P}, \mathfrak{Y}, \mathfrak{M}, \mathfrak{U}, \mathfrak{V})$, where $\mathfrak{P}: \mathbb{D} \to \mathbb{P}$, $ \mathfrak{Y}: \mathbb{K} \times \mathbb{D} \to \mathbb{Y}$, $ \mathfrak{M}: \mathbb{K} \times \mathbb{D} \to \mathbb{M}$, $ \mathfrak{U}: \mathbb{K} \times \mathbb{D} \to \mathbb{U}$, and finally $ \mathfrak{V}: \mathbb{D} \to \mathbb{B}$ (We discuss the $ \mathfrak{V}$ later). The correspondence is the following. \begin{eqnarray*} \begin{cases} S(K, D) = (\mathfrak{P}(D), \mathfrak{Y}(K, D), \mathfrak{M}(K, D), \mathfrak{U}(K, D)) & \mathfrak{V}(D) \\ S(K, D) \textit{ raises error} & \neg \mathfrak{V}(D) \end{cases} \end{eqnarray*}

As one can see above, the Validation part of a Super $\mathfrak{V}$ is responsible for validating the given arguments.

Server Service Classes

(main chapter)

some data-types are more than an entity or a Super. Some classes like `DateTime` are neither in an equivalence relation with a database table nor possible to implement with Xua's Super, since the programmer may need methods such as `addDays`, `format`, or `toJalali` from this data-type.

Also, some procedures need to be implemented somewhere other than Methods, Supers, or Entities. For example, adding a watermark to an image or building a PDF.

Xua satisfies this need by letting programmers write their native PHP scripts. Although the codes must be written inside classes, the programmer is pretty much free to do everything inside the classes. The Xua engine will copy these native codes to the resulting project as they are.

Resource Files

In almost any project, there exist some files that are not native scripts; these files can be anything, assets, user-uploaded files, documents, etc. In Xua's terminology, we call any non-executable file a resource file. The resource files exist directly in the resulting project directory, and the programmer must mark the directories that contain resource files as resource directory in `config.json`. Read Configurations Chapter to see how to do this.

Public Resource Files

Resource files can be public, i.e., accessible by just going to the file's address. For example, when a user opens a web page in the browser, there are some assets, e.g., styles, fonts, images, etc., that browser fetches from the server. These files must be marked as public.

Private Resource Files

In many projects, some files are needed to be on the server but are not public files. For example, think of an online book store. There are many `.pdf` files on the server which the server sends them via email or shows them on the web only if the user is authorized and paid for them. These files must be marked as private.

Interfaces

(main chapter)

interfaces are the frontline of the project, immediately executed when a user requests on the interface's defined route. We usually avoid writing codes that include business logic; instead, we call a resource responsible for the route. Since we have a specific route called URPI that handles all the direct requests for resources, writing a new interface is only helpful in rendering a website, i.e., we define routes for pages the user visits, and ask a resource (a Method resource) to create an HTML page, and show the page created to the user.

Xua Server Generated Units

some units exist in the server project that are entirely generated by the Xua engine, and there is no need for a programmer to modify them, but it helps if the programmer knows how they work, and in some cases, the programmer may modify them in a way that is more suitable for the project.

Universal Resources Pool Interface (URPI)

Accessing resources on the same project is as easy as before; one unit can call a resource internally, and there is no complexity. Although accessing a resource from other machines needs a protocol, the client needs to marshal the request in a data-type that is available for transmitting on the network, then we need to unmarshal the request in a way that is meaningful for Universal Resource Pool, this is done by a predefined interface called URPI. The client tells URPI what resource is needed, URPI calls the resource, prepares the result for network transmitting, and sends the result back to the client, where the result can be unmarshalled and used.

Note. The URPI is an Interface existing in the interfaces directory of the template project. The URPI is a Xua source file, meaning it is not generated when building the project; it has been generated

only once when making a new project from Xua's default template. Therefore this unit is modifiable and even removable.

Route Detection Unit

The programmer can map URIs on the website to interfaces using a file called `routes.xrml` (Xua's routes minimal language). This way, when a user or client sends a request to the server, it is the job of RDU to detect which route matches the requested URI and method (such as POST, GET, OPTIONS, etc.). After the route is detected, Xua will call the corresponding interface, and then the script inside that interface returns the desired response.

There is also a predefined service called `RouteService`, which has a function that does the inverse of this mapping and helps find the URI of an interface.

Xua Exception Handler

After the interface returns the response, it is not prepared to show to the user yet. Immediately after interface code execution, the result goes to the XEH, any uncaught exception, fatal error, warning, etc. get caught here, and a default error page is shown to the user in order to avoid inside script or data leaks, as well as making the website more user friendly.

It is possible to modify how Xua catches exceptions by modifying a predefined service called `ExceptionService`. For example, one may want to make it possible for developers to see the exact error while users must see a simple error page. Alternatively, one may want to show an error message in JSON instead of showing an HTML page. In these cases or other similar cases, One can modify `ExceptionService` and check if the request is sent by a developer, for example, by checking a special header. Read ExceptionService for more practical information.

Marshal Library

The Marshal Library is a client library that can be built in several front-end languages. The library helps the front-end developers to access resources from the Universal Resources Pool. For example, if there is a method in the pool called `Method\Media\GetOne`, the front-end developer can call it using the following code in Dart.

```
result = Marshal.Method.Media.GetOne(mediaId);
```

The Marshal library sends a request to the server saying the method `Method\Post\Get` needs to be executed with given marshaled parameters. The URPI then unmarshals the given parameters, calls the method, receives the response, marshals it, and sends it back to the client. The Marshal library fetches it, unmarshals it, and returns the structured response, which then will be used in the front-end project.

Virtual Resource Pool

The Virtual Resource Pool is a set of methods and entities, twins of the ones in the server, containing only a server call and nothing more.

Entity Interfaces

For each entity on the server project, a twin entity exists inside VRP but with hollow methods. Each method only contains an HTTP request that tells the server to execute its twin; the result is fetched from the server and is returned.

Method Interfaces

For each method on the server project, a twin method exists inside VRP but with a hollow body. The body only contains an HTTP request that tells the server to execute its twin; the result is fetched from the server and is returned.

Local Service Classes

(main chapter)

Server Service Classes discussed above are classes that programmers may use to instantiate objects. These objects can be used in universal resources. For example, a method may return an instance of a service class. Xua marshals the result and sends it to the Marshal Library where it should be unmarshalled. But how are we going to represent the object there? Assume we have a method called `getNow`, and the method responses the variable `now`, an instance of `DateTimeService`. Then in the JS project, we want to get the Jalali representation of the `DateTimeService` using the method `formatJalali` on `DateTimeService`. This is not something we ask the server to do for us. We should be able to execute the method on the client side. Therefore, if we want to use some methods of some service classes on the client side, we should rewrite the class in the native language of the client side. The programmer can write native codes for the client, and the codes are copied to the Marshal Library when building.

Marshal Library Generated Units

Just as the server project, some units exist in the Marshal library that are entirely generated by the Xua engine, and there is no need for a programmer to modify them, but it helps if the programmer knows how they work, and in some cases, the programmer may modify them in a way that is more suitable for the project.

Marshal/Unmarshal Unit

This is the unit that the library derived its name from. This unit is responsible for marshaling request data and unmarshaling the response. The unit is affected by supers definitions but is not directly programmable.

Http Connection Unit

This is a rather technical unit that sends requests to the server and fetches responses. This unit can also modify requests and responses, for example, adding a user token, API key, etc., to the request. This unit is possible to modify. The programmer must modify a predefined service called `HttpConnectionService`. Read HttpConnectionService for more practical information.

Documentations

(main chapter)

Xua is capable of generating a set of documentations for the project. Currently, the documentations are available in HTML and LaTeX formats. Xua engine uses comments inside project source files to generate the doc. The current documentation you are reading is generated by Xua, and all these texts are just comments inside source files.

The documentations are file-to-file, meaning that for each source file, there exists a `.html` or `.tex` file and vice versa.

Service Classes Documentations

Service classes are the files labeled as services in `config.json`, and are only project source files without the `.xua` extension. For each of these files, the Xua engine generates a documentation file consisting of the comments.

Super Types Documentations

Other than including the comments in the documentation, Xua can automatically generate a table of super parameters, with their types, default values, and description. The description of a parameter is the comment that comes immediately after the parameter definition; if no comment is found, the description is set to `-`. Xua can also sectionize different parts of a super, like validation, marshal, unmarshal, etc., and shows comments of each one in the related section. There are also some other minor automated functions that Xua can perform.

Entities Documentations

Other than including the comments in the documentation, Xua can automatically generate a table of entity fields, with their types, default values, description, etc. The field description is the comment that comes immediately after the field definition; if no comment is found, the description is set to `-`. Xua can also sectionize different parts of an entity, and some other automated functions are also present.

Entities Graph

Almost any project has some related entities; for example, we may have an entity called `User` that has a field called `birthPlace` that comes from another Entity called `AdministrativeDivision`, and an entity called `Restaurant` that has a field called `costumers` which comes from `User`, and a field called `city` that comes from `City`. There would be many other entities with relations; these relations can be one-to-one, one-to-many, many-to-many, or many-to-one. Most of these complicated projects have a graph of entities as vertices, showing the relations by edges. Xua is capable of drawing this graph and modify it whenever an entity is modified.

Methods Documentations

Other than including the comments in the documentation, Xua can automatically generate a table of the request parameters, and a table of the response parameters, with their types, default values, description, etc. The description of a field is the comment that comes immediately after the

parameter definition; if no comment is found, the description is set to `-`. Xua can also sectionize different parts of a method, and some other automated functions are also present.

Book

Book is the name we call a `.xua` file, or a set of `.xua` files when the file(s) do(es) not affect the Server project or the Marshal library, just like the one you are reading currently.

Foreign Units

Some units are out of our control, they work as they are, but they are our concern since we want to communicate with them.

System Resources

System resources are usually procedures and scripts not written in PHP, such as cronjobs, shell commands, Redis queues, foreign language codes, etc. However, we may want to use these resources in our project. This use can be done through a service.

For example, assume we have a coding contest website and want to compile and run some tests on uploaded codes. We may have a foreign judge project written in Python that does this for us; we only need to call it. We can write a service called `JudgeService` and use it in a method. Or if we want to manage a queue, we can write a service called `QueuesService` and call it like the following.

```
QueuesService::addToQueue(QueuesService::QUEUE_WAITING_TO_JUDGE_CODES, $uploadedCode)
```

Libraries

There can be some PHP libraries that we do not wish to rewrite, for example, generating a QR Code, reading an excel file, etc. We can have these codes available and write a service to call these for us. It is essential to mark libraries as resource files in `config.json`, or they will be deleted when cleaning the project.

3rd Party Apps

Same as System Resources, and Libraries, we can write services and interfaces to communicate with 3rd Party Apps. For example assume we want to use `redis` which is a 3rd party tool, we can write a service that does the low-level communications and offers a set of high-level methods to the Xua project.

Database and Cache Servers

The main feature of any server project is communicating with a database server; Entities are configured to do so. However, it is also possible to configure entities to connect to a cache server instead. The cache server then communicates with the database if the data is not available.

Web Browser

The web browser is the interface that the user communicates with; if the project is not just a web-service, and contains a website, then the web browser is the application that communicates with

the server project, which may ask for different URLs on the server that routes and interfaces will handle.

Native Client Codes

Client Codes are the codes written by front-end developers that are the user interface of the project. These codes may not communicate with the server directly; instead, the Marshal library is invoked to communicate with the server.

INTERFACES & ROUTES;

THE PROJECT'S FRONTLINE

Introduction

Interfaces are the frontline of the project and are written in pure PHP language. When the Server project receives a request, the Route Detection Unit which is implemented as a service called `RouteService`, decides what Interface to execute based on a file called `routes.xrml`. The Interface then returns a string as the response.

Definition

Structure

An Interface block has the following structure.

```
# Path\To\Interface\InterfaceName

# Description of this Interface, probably in markdown formatting

Interface [extends Path\To\Another\Interface\InterfaceName] {
  INTERFACE_BODY
}
```

Note that there is no name for the Interface. An Interface name is its file path. Each file can contain at most one block, and the block inherits its name from the file. It is a good practice to have a comment in the first line of the file describing where the file is located.

Hierarchy

Interfaces can come in a hierarchy, just like PHP classes. It is possible to call the parent Interface body in the child Interface since the Interface content is written in PHP. Just note that the method, when compiled to PHP, is called `execute`. So, to execute the parent Interface body, one must write `parent::execute()`.

Xua's Route Minimal Language

The file `routes.xrml` is written in a specific language called XRML, with the following grammar.

An example of this file would be as following.

We use this file to explain all of the XRML features step-by-step.

Literal Routing & The Routes Tree

When the Server project receives a request, the main file asks the `RouteService` to find the desired Interface. The Route Service then explodes the requested URL into different pieces by the `/` character.

Assume the client sends a GET request to `localhost:8000/media/post`. The result of exploding this URL would be the list `['media', 'post']`. The Route Service then looks into the `routes.xrml` as a tree, finds the node matching the list's first item, and gets the subtree. Then looks for the list's second item in the subtree and continues the process until it finishes the list. Then looks into the Interfaces provided for that line and chooses the one matching the request method, `Post\GetAll`. If it can't find such an Interface, it throws an Exception called `RouteException`, which the `ExceptoinService` will handle.

Note. The `/` route is only valid on the root tree, and no subtree can contain such a node. Violating this rule will result in a `RouteDefinitionException`.

All Methods At Once

There are two ways to define Interfaces. One is to define an Interface for each HTTP method, in which case if the request method is not defined, the Route Service will throw a Route Exception. However, if one wants to set an Interface for all methods without specifying them, they can just set the Interface without any methods provided, so the Route Service will divert all requests on that URL to the defined Interface regardless of the HTTP method.

In the example above, Route Service will divert all requests to `localhost:8000/404` to the `Xua\NotFoundInterface`.

Variables & Priorities

Assume the client sends a GET request to `localhost:8000/media/post/12/delete`. The result of exploding this URL would be the list `['media', 'post', '12', 'delete']`. The Route Service gets the subtree with `post` as root, and in that tree, it can't find any route equal to the literal string `'12'`, but instead, there is the route `{id}` with opening and closing braces indicating a variable. This way, the Route Service knows it can match any string with that route, and it'll save the string `'12'` under the variable name, which is `id`. The Route Service will process further until it finds the Interface `Post\RemoveOne`. Now the variable is available in the body of that Interface as `RouteService::$routeArgs['id']`.

Note. If one of the siblings of a variable matches the desired route section, Route Service processes that node since literal route sections have a higher priority than the variable route sections; but if further processing the route would result in Route Exception, the Route Service would come back until the latest variable and processes the route in that direction, if it can't find the route in that direction either, it goes back and does the same procedure until possible.

Note. There can't be two variable route sections as siblings, as this will result in a `RouteDefinitionException`.

SUPERS;

A WHOLE LEVEL BEYOND TYPES

Introduction

Super types or simply supers are parametric types. Read Super Types section under Units chapter for theoretical explanations. In this chapter, we focus on practical details of defining and using supers.

Effects on the projects

Each super results in a PHP class in the Server project, extending the abstract class `Super` or another Super.

Also, the same happens in the Marshal library with respect to the language. However, the front-end clone of the super does not contain the predicate and is only accessible in the MUU (Marshal/Unmarshal Unit).

Terminology

Assume A is a set, the predicate P is called characteristic predicate of A if $P(x) \iff x \in A$. We can use set A and predicate P alternatively since they both carry the same concept.

A type is a set occupied with functions M, U, and T. Read Super Types section under Units chapter to see what these objects are. We usually show a type set by its characteristic predicate instead of the set itself.

Assume $T = (P, Y, M, U)$. We say type T explicitly accepts value x if $P(x)$ is true, and say type T implicitly accepts value x, if $P(x)$ is true or $P(U(x))$ is true. If type T explicitly (implicitly) accepts x, we say x explicitly (implicitly) fits in T. As a convention, when we say T accept x, we mean T implicitly accepts x, and the same is true for fitting.

A Super Type or simply Super is a function that receives some arguments and returns a type. This type is called an instance of the Super Type.

Definition

Structure

A super block definition has the following structure.

```
# Path\To\Super\SuperName

# Description of this Super, probably in markdown formatting

Super [extends Path\To\Another\Super\SuperName] {
  [arguments {
    [const] argName0 : Type0 [= DEFAULT_0]; # Description of argName0
```

```
    ...                                     ; # ...
    [const] argNameN : TypeN [= DEFAULT_N]; # Description of argNameN
  }]
  [ validation { VALIDATION_BODY } ]
  predicate { PREDICATE_BODY }
  [ type<TARGET_0> { TYPE_0_BODY } ]
  [ marshal<TARGET_0> { MARSHAL_0_BODY } ]
  [ unmarshal<TARGET_0> { UNMARSHAL_0_BODY } ]
  ...
  [ type<TARGET_N> { TYPE_N_BODY } ]
  [ marshal<TARGET_N> { MARSHAL_N_BODY } ]
  [ unmarshal<TARGET_N> { UNMARSHAL_N_BODY } ]
}
```

Note that there is no name for the Super. A Super name is its file path. Each file can contain at most one block, and the block inherits its name from the file. It is a good practice to have a comment in the first line of the file describing where the file is located.

Arguments

It is possible to define a set of arguments for a Super. The resulting type is based on those arguments. For each argument, it is possible to set a default value, and it is mandatory to set a type. However, the only way to mention a type is to call a super that returns a type, and we need types to define supers. So how to define the first Super? The answer is we do not NEED types to define a super since we do not need the super to have any argument. This fact leads us to the very first Super, called `Universal`.

It is possible to define some arguments as constant, which means the programmer cannot pass that argument to the Super. This feature is useful when defining a child super. Read more in Hierarchy.

```
# \Xua\Supers\Universal
Super {
    Predicate {
        return true;
    }
}
```

After this, we may use the `Universal` Super to define other Supers. Although, we may define other Supers without using arguments, such as `Boolean` and `Trilean`.

Validation

The `validation` block is responsible for checking if arguments passed to the Super meet desired conditions. It is an optional Block, and if it is not provided, any argument that fits in the corresponding type will be accepted.

The body is written in pure PHP. All super arguments are available as PHP variables `$this->argName0` `$this->argName0`, ..., `$this->argNameN`, There is also another available variable `$exception`, which has a method called `setError`, and one can use it to add an error if some wanted conditions are not met.

Predicate

The `Predicate` is the main and only required block of a super. This block defines the characteristic predicate.

The body is written in pure PHP. All super arguments are available as PHP variables `$this->argName0` `$this->argName0`, ..., `$this->argNameN`, along with two extra variables `$input` and `$message`. The variable `$input` contains the value which the predicate should check. The value `true` must be returned if `$input` explicitly fits in the type, and `false` otherwise.

When returning `false`, it is possible to provide a reason of why `$input` failed to fit in the type. The `$message` variable is responsible for storing this reason.

Marshal & Unmarshal

Blocks `Marshal` and `Unmarshal` can be written for any supported language. Therefore these block names are valid: `marshal<php>`, `marshal<database>`, `marshal<dart>`, `marshal<php>`, `marshal<java>`, `marshal<javascript>`, `marshal<kotlin>`, `marshal<objectivec>`, and `marshal<swift>`. The body of `database` blocks, is written in PHP, but the methods are used for database store and restore procedures.

Any unwritten block is assumed to be the identity function.

The function of the `marshal` block is to cast a given value into a value that can be transmitted on the network or stored in the database (usually string, integer, or a stream of bytes) in an invertible way. The `unmarshal` does the inverse of the `marshal` function.

When the language is PHP (target is `php` or `database`), all Super arguments are available as PHP variables `$this->argName0`, ..., `$this->argNameN`, along with an extra variable `$input`, and the marshaled/unmarshaled data must be returned. When calling `Marshal`, it is guaranteed that `$input` explicitly fits in the type. But when calling `unmarshal`, it is possible that `$input` is not a valid input; in that case, the convention is to return the input value itself.

For other languages, the same concept holds with respect to the language syntax.

Native Type

There are some scenarios where we need to declare a type (the result of calling a Super) in another language. For example, if we use a type as the type of an Entity field, we must tell the database server to store the values. This declaration is the job of the `Type` block. All super arguments are available as PHP variables `$this->argName0`, ..., `$this->argNameN`. And the block must either return a string identifying the database type (for example, `VARCHAR(100)`) or `null` value, which means the type is not available in the specified target. If the block is not provided, the `null` return value is assumed.

Hierarchy

Supers can come in a hierarchy just like PHP classes, and each block is a class method. It is possible to call the parent method in the child method since the block's content is written in PHP. It is also possible to call parent methods in non-PHP blocks with respect to the language syntax.

To do so, it is mandatory to know the name of the class method generated by each block. Here is a list of blocks with corresponding methods.

```
# PHP Blocks
Validation            => protected _validation(SuperValidationException $exception)
Predicate             => protected _predicate($input, string &$message)
Marshal<database>     => protected _marshalDatabase($input)
Unmarshal<database>   => protected _unmarshalDatabase($input)
Marshal<php>          => protected _marshal($input)
Unmarshal<php>        => protected _unmarshal($input)
Type<database>          => protected _databaseType()
Type<php>          => protected _phpType()
# non-PHP Blocks
Marshal<FRONT_LANG>   => protected _marshal($input)
Unmarshal<FRONT_LANG> => protected _unmarshal($input)
Type<FRONT_LANG>          => protected _type()
```

So for example if one needs to call parent predicate in php, they may write `parent::predicate($input)`.

Also, the arguments of child super override the ones in the parent. This override includes type, default value, and being constant. It is also possible to add new arguments to the type. Read Examples for more details.

Visibility

The visibility of supers is controlled via the Marshal and Unmarshal blocks. Although the supers are never accessible in the front-end project, the Marshal library can use these methods to send and receive values of a type. However, if the methods are not available, this means that we do not want the Marshal library to be able to use them.

By the way, there is usually no point in controlling Super's visibility.

Usage

It is possible to use a defined super in both PHP and Xua languages, but not in Marshal library.

Inside PHP

Although it is not usually helpful to work with supers inside PHP codes, it is possible.

Make a type

The following code makes a type by giving arguments to a super.

```
$type = new Path\To\Super\SuperName([
    'argName0' => $value0,
    ...,
    'argNameN' => $valueN,
]);
```

Determine a type

Assume a type is given in a variable and we need to know what type is it. We can try to find the class of the object but what about arguments? The `parameters` method returns the array that was given to the Super at first place, and the `toString` method returns a string that describes the type.

```
var_dump(get_class($type));
var_dump($type->parameters());
var_dump($type->toString());
```

This code dumps the following.

```
string "Path\To\Super\SuperName"
[
    argName0 => value0,
    ...,
    argNameN => valueN
]
string "Path\To\Super\SuperName(argName0 = value0, ..., argNameN = valueN)"
```

Accepts, Implicit & Explicit

There are three `accepts` functions defined on a type.

explicitlyAccepts

This function will return true only if the value explicitly fits in the type. The second argument is optional, and if the return value is false, the function may fill it with a reason.

```
if ($type->explicitlyAccepts($value, $reason)) {
    var_dump($value);
    echo 'is of type ' . $type->toString();
} else {
    echo 'Rejected, because:' . $reason;
}
```

implicitlyAaccepts

This function will return true if the value explicitly fits in the type, or fits after unmarshaling. The unmarshaling methods must be determined by the caller. The function tries the value itself first, then tries to unmarshal and check if the value fits by given methods, one by one. If not passed to the function, the default value `[self::METHOD<em>IDENTITY, self::METHOD</em>UNMARSHAL, self::METHOD<em>UNMARSHAL</em>DATABASE]` is assumed.

```
if ($type->implicitlyAccepts($value, $reasons, [self::METHOD_IDENTITY, self::METHOD_UNMARSHAL
])) {
    var_dump($value);
```

```
    echo 'kinda fits in the type ' . $type->toString();
} else {
    echo 'fully rejected because of the following reasons';
    var_dump($reasons);
}
```

accepts

This function does the same job of `implicitlyAccepts`, but may alter the original value while trying to fit it in the type.

```
$originalValue = $value;
if ($type->accepts($value, $reasons, [self::METHOD_IDENTITY, self::METHOD_UNMARSHAL
])) {
    var_dump($originalValue);
    echo 'was changed to';
    var_dump($value);
    echo 'to fit in type ' . $type->toString();
} else {
    echo 'fully rejected because of the following reasons';
    var_dump($reasons);
}
```

Marshal & Unmarshal

There are two types of marshal and unmarshal functions available in the server project.

Network transmissions

Functions `marshal` and `unmarshal` are responsible to marshal and unmarshal values for purpose of network transmissions.

```
$marshaledValue = $type->marshal($value);
$originalValue = $type->unmarshal($marshaledValue);
if ($value !== $originalValue) {
    var_dump('something is wrong with marshal/unmarshal functions of ' . $type->toString());
}
```

Database Storing & Restoring

Functions `marshalDatabase` and `unmarshalDatabase` are responsible to marshal and unmarshal values for purpose of storing and restoring into/from database.

```
$marshaledValue = $type->marshalDatabase($value);
$originalValue = $type->unmarshalDatabase($marshaledValue);
if ($value !== $originalValue) {
```

```
    var_dump('something is wrong with marshalDatabase/unmarshalDatabase functions of ' . $type->toString());
}
```

Native Type

The functions `phpType` `databaseType` have no arguments and return a string that declares the type in PHP and database engine syntax. `phpType` is mostly used in PHPDocs to declare the classes properties type, while `databaseType` is mostly used to tell the database server how the type values must be stored. These functions are hardly helpful in programming.

Inside Xua

The main usage of Supers is to declare types for Xua Super arguments, Entity fields, and Method request and response signatures (and Method field signatures for VARQUE Methods). To mention a type, one must call a Super and give it arguments. The type then can be used for type declaration. The syntax is the following.

```
Path\To\Super\SuperName(
    argName0 = constant0,
    ...,
    argNameN = constantN,
)
```

Examples

Range

In this section, we want to work with a Super that accepts the range of integers in $[a, b)$.

Definition

First of all, we need to define the Super. We put it in the file `Supers/Integers/Range.xua`.

Extension

A value of this type needs to be an integer so we can reuse marshal, unmarshal, and native type methods of the `Integer` Super, and also the predicate of that Super would be useful to check if the value is an integer. So it seems like a good idea to extend the Super `Integer`.

```
Super extends Integer
```

Arguments

We need to have two arguments determining the start and end of the range. The arguments must be integer themselves.

```
# Supers\Integers\Range
Super extends Integer {
    arguments {
        start : Integer();
        end   : Integer();
    }
    validation {
        # TODO implement
    }
    predicate {
        # TODO implement
    }
}
```

Validation

For the validation, we must check that the second number is not less than the first one. Note that we do not need to check if the type of arguments is integer since we already declared their type so the type checking is automatically done.

```
validation {
    if ($this->end < $this->start) {
        $exception->setError('end', 'The argument `end` cannot be less than the argument `start`.');
    }
}
```

Predicate

We can just simply check if the `$input` is an Integer and is in the range.

```
Predicate {
    return parent::_predicate($input) and $this->start <= $input and $input < $this->end;
}
```

But we can make it a little more sophisticated by providing a reason of why the value may fail to fit.

```
Predicate {
    if (!parent::_validation($input, $message)) {
        // The message is already filled here by the parent _validation
        return false;
    }
    if ($input < $this->start) {
        $message = $input . ' is less than ' . $this->start;
        return false;
    }
    if ($input >= $this->end) {
```

```
        $message = $input . ' is not less than ' . $this->end;
        return false;
    }
    return true;
}
```

Note that it is OK to fill the $message when the return value is `true`. Xua automatically clears the `$message` in that case.

Marshal and Unmarshal

The defined Super is ready-to-go with no further modifications; since the `Integer` Super includes the desired Marshal and Unmarshal functions and `Range` inherits them. However, for the purpose of this documentation, we override them with a new network transmit marshaling system.

The silly idea is to shift starting number to zero to have smaller integers which are easier to transmit (practically useless). For example The range $[1000, 1100)$ can be shifted to $[0, 100)$.

```
Marshal<php> {
    return $input - $this->start;
}
Unmarshal<php> {
    return $input + $this->start;
}
Marshal<javascript> {
    return input - this.args.start;
}
Unmarshal<javascript> {
    return input + this.args.start;
}
```

Native Type

The PHP type is inherited and returns `int`, which is great. However, the database type is an interesting part of the definition since we can determine the length of MySQL `INT` by the range limits.

We know that `INT(n)` can store values in range $[-2^{n-1}, 2^{n-1}]$. So it is efficient to find the least n such that this range contains our range. First, we find the maximum absolute value that can fit in the type.

```
$min = $this->start;
$max = $this->end - 1;
$absMax = max(abs($min), abs($max));
```

Let us say this number is M. we must find n such that $M \leq 2^{n-1}$. $$\begin{eqnarray} M \leq 2^{n-1} & \iff & \log_2(M) \leq n - 1 \\ (\text{Since $n - 1$ is integer}) & \iff & \lceil \log_2(M) \rceil \leq n - 1 \\ & \iff & \lceil \log_2(M) \rceil + 1 \leq n \\ \end{eqnarray}$$

So the minimum value of n is $\lceil \log_2(M) \rceil + 1$.

```
$n = ceil(log($absMax, 2)) + 1;
```

This leads us to the following database type.

```
type<database> {
    $min = $this->start;
    $max = $this->end - 1;
    $absMax = max(abs($min), abs($max));
    $n = ceil(log($absMax, 2)) + 1;
    return "INT($n)";
}
```

Further Hierarchy

Validation

Assume that we want another Super with the same features, but only for positive values. We can extend again what we already have.

```
# Supers\Integers\PositiveRange
Super extends Range () {
    validation {
        parent::validation();
        if ($this->start<= 0) {
            $exception->setError('start', 'The range must be positive.');
        }
    }
}
```

Constant Arguments

Or assume we want a range super that can only start at zero.

```
# Supers\Integers\NaturalUpperLimit
Super extends Range {
    arguments {
        const start : Integer() = 0;
    }
}
```

Note the way we overrode the `start` with default value, but also set it constant so the caller cannot change it.

```
$type = new NaturalUpperLimit(['start' => 1, 'end' => 2]);
```

This code will result in an uncaught `SuperValidationException`.

Marshaling

But better than these, assume we use the network transmit marshaling procedures for the database, which actually impacts the table size. Furthermore, we can remove the silly network transmit marshaling procedure. Also, note that we must override the `type<database>` as well.

```
# Supers\Integers\EfficientRange
Super extends Range () {
    Marshal<php> {
        return $input;
    }
    Unmarshal<php> {
        return $input;
    }
    Marshal<javascript> {
        return $input;
    }
    Unmarshal<javascript> {
        return $input;
    }
    Marshal<database> {
        return $input - $this->start;
    }
    Unmarshal<database> {
        return $input + $this->start;
    }
    DatabaseType {
        $max = $this->end - $this->start - 1;
        $n = ceil(log($max, 2)) + 1;
        return "INT($n)";
    }
}
```

New Arguments

Another example is when we want to add arguments to a Super. For example, assume we want a `step` argument. By default, the step is one, but if we set the `step` to three, the type accepts `start` and every third number.

```
# Supers\Integers\StepRange
Super extends Range {
    arguments {
        step : Integer() = 1;
    }
    validation {
```

```
        parent::validation();
        if (step < 1) {
            $exception->setError('step', 'The step must be at least 1.');
        }
    }
    predicate {
        if (!parent::predicate($input, $message)) {
            return false;
        }
        if ($input - $this->argument['start'] % $this->argument['step'] != 0) {
            $message = "value {$input} minus starting point {$this->argument['start']} is not devisable by ste
            return false;
        }
        return true;
    }
}
```

Usage

After defining a super, we can use it to define types and use them.

Inside PHP

First, Let us define a type that accepts the range $[10, 30)$. We use the `EfficientRange`.

```
$type = new EfficientRange(['start' => 10, 'end' => 30]);
```

Let us see how Xua stringifies this type.

```
var_dump($type->toString());
```

This code dumps `Supers\Integers\EfficientRange(start = 10, end = 30)`.

The value `25` explicitly fits in the type, while the value `5` implicitly fits.

```
$value = 25;
var_dump($type->explicitlyAccepts($value)); # dumps true
$value = 5;
var_dump($type->implicitlyAccepts($value)); # dumps true
```

Of course the value `25` also fits in the type implicitly. This is because the function first checks if the value explicitly fits.

```
$value = 25;
var_dump($type->implicitlyAccepts($value)); # dumps true
```

We know that value `15` is ambiguous. It can be interpreted as `15` itself, a value in range, or the result of marshaling `25`. Let us call accept and check the result.

```
$value = 15;
var_dump($type->accepts($value)); # dumps true
var_dump($value); # dumps 15
```

Since the function first tries the explicit, if the value fits explicitly, function does not change the value. What about `5` that can only fit implicitly?

```
$value = 5;
var_dump($type->accepts($value)); # dumps true
var_dump($value); # dumps 15
```

This time the function tries to fit the value explicitly and fails, so it goes for unamrshaling, which leads to accept.

We know the value 5 does not fit explicitly, but we may wonder why. (It is super obvious but is a good way to see how to get the reason from the function.)

```
$value = 5;
var_dump($type->explicitlyAccepts($value, $reason)); # dumps false
var_dump($reason); # dumps '5 is less than 10'
```

What about value `30`?

```
$value = 30;
var_dump($type->explicitlyAccepts($value, $reason)); # dumps false
var_dump($reason); # dumps '30 is not less than 30'
```

But we know `30` does not even implicitly fit.

```
$value = 30;
var_dump($type->accepts($value, $reasons)); # dumps false
var_dump($reasons);
```

This function fills the `$reasons` with an array, reasoning about each failure.

```
[
    'identity' => '30 is not less than 30',
    'unmarshal' => '30 is not less than 30',
    'unmarshalDatabase' => '40 is not less than 30',
]
```

The `$reason` variable can work as a log; for example, we try the code above with `5` and get the following.

```
[
    'identity' => '5 is less than 10',
    'unmarshal' => '5 is less than 10',
    'unmarshalDatabase' => null,
]
```

We can see there is no reason for the `unmarshalDatabase`, and it makes sense since the value fits using this unmarshal method.

We know that the storing in the database is done by marshaling. The marshaled values are in $[0, 20)$.

```
var_dump($type->DatabaseType()); # Dumps 'INT(6)'
```

In this case, there is no efficiency since the original values would only need `6` bits too. Although we could make this better if we shifted the center of the range to zero instead of starting point. That way, we would have the range $[-10, 10)$ which needs only 5 bits to store. By the way, none of this is useful because MySQL occupies at least a byte which is `8` bits, and there is no difference between `5` and `6` in practice. Although this marshaling method may come in handy for big values, Xua's official `Range` does not use it because of the ambiguity.

Inside Xua

We may use a Super to define another Super.

```
# Supers\Gender
Super {
    arguments {
        possibilities : Range(start = 2, end = 4) = 2;
    }
    predicate {
        $choices = ['male', 'female'];
        if ($this->possibilities == 3) {
            $choices = ['male', 'female', 'non-binary'];
        }
        $message = '$input is not in ' . implode(", ", $choices);
        return in_array($input, $choices);
    }
}
```

Note how we set the `$message` without caring about the return value. Xua automatically clears the `$message` if the return value is `true`.

Good News

Although defining a simple Super seems easy, defining a complete Super with all features in many languages seems exhausting. The good news is, Xua provides a set of predefined supers that cover almost any need. We discuss them in the next chapter.

PREDEFINED SUPERS;

@TODO

Introduction

@TODO

ENTITIES;

THE BEATING HEART

Introduction

Xua can be configured to communicate with a database server (usually MySQL), but the programmer must specify the database structure. To do so, Xua offers Entity Blocks. Read Entities section under Units chapter for theoretical explanations. In this chapter, we focus on practical details of defining and using entities.

Effects on the projects

Xua will generate a PHP class extending a Xua abstract class called `Entity` for each Entity Block the programmer creates. These Entity classes are in a one-to-one correspondence with database tables, and each row of the table can be corresponded by an instance of the table's corresponding class.

Also, the same happens in the Marshal Library with respect to the language. However, the front-end clone of the entities only contains method signatures, and the bodies are just network calls.

Terminology

Server Project Side

A Literal Name is a literal string containing only alphanumeric characters, starting with a lowercase character. The set of all Literal Names is shown by $\mathbb{L}$.

The set of all values possible to store in a variable is shown by $\mathbb{X}$.

A Dictionary is a partial function $D: \mathbb{L} \to \mathbb{X}$ and is usually expressed by all of its records. The set of all dictionaries is shown by $\mathbb{D}$.

A Field (T, D) is a tuple, where $T \in \mathbb{T}$ is a type (described in Supers chapter) called Field Type and $D \in \mathbb{X}$ is a value called Field Default Value (providing D is optional). Note that if D is provided, it is mandatory that $D: T$.

An Entity Signature is a dictionary F such that $F(s) = (T_s, D_s)$ or $F(s)$ is undefined, where s is a Field Name. Domain of the F is called the Set of Field Names of the Entity Signature.

The set $\{ \mathsf{ASC}, \mathsf{DESC} \}$ is called the set of all Order Indicators and is shown by $\mathbb{O}$.

An Index is a k-tuple of tuples (f, o) with an extra boolean determining if the index is unique, where f is a Field Name, and $o \in \mathbb{O}$ is an Order Indicator. Indexes are used in the MySQL engine for faster `select` queries. We try to have a simple explanation here. Let $i_0 = \Big(\big((f_0, o_0), \dots, (f_n, o_n) \big), u \Big)$ be an Index. Intuitively speaking, The MySQL engine will create a list of pointers to data rows, sorted by the mentioned field, which makes it faster to search on those fields. Also if the Index is marked as unique, i.e., $u = \mathsf{1}$, the

combination of fields must be unique in data, i.e., we cannot have two rows with same value of f_0, same value of f_1, ..., and same value of f_n at the same time.

An Entity Indexes is a set of indexes I.

An Entity Validation is a function with no output that inputs a data row and checks if the data is valid. In case of invalid data, an exception is thrown. For example, assume we store data about some events, each event has two fields, `start_time` and `end_time`, and the start must be sooner than the end. The Entity Validation checks if this condition holds, and if the end is sooner than the start, it will throw an `EntityFieldException`.

An Entity Class is a triplet (F, I, V) of fields, indexes, and the validation function.

An Entity Instance is an instance of an entity class, which contains actual values for entity fields.

MySQL Side

A database is a set of tables; each table has a structure consisting of columns, and a set of rows as data.

`insert` is the action of adding new rows to a table.

`select` is the action of retrieving table rows on some specific conditions.

`update` is the action of modifying some table rows on some specific conditions.

`delete` is the action of removing some table rows on some specific conditions.

Correspondence

Each table is in one-to-one correspondence with an Entity Class. Methods of these classes can communicate with the database server to select, insert, update, and delete data. Fields of the entity class represent the columns of the table, and instances of the class represent the rows of the table.

Definition

Structure

An entity block has the following structure.

```
# Path\To\Entity\EntityName

# Description of what this Entity is all about, probably in markdown formatting.

Entity [extends Path\To\Another\Entity\EntityName] {
    fields {
        field0 : type0 [= DEFAULT_0]; # Description of field0
        ...                         ; # ...
        fieldN : typeN [= DEFAULT_0]; # Description of fieldN
    }
    [ indexes : {
```

```
        ([-]filedName00, ..., [-]fieldName0K)[*]; # Description of index number 0
        ...                                     ; # ...
        ([-]filedNameM0, ..., [-]fieldNameML)[*]; # Description of index number M
    } ]
    [ validation : { VALIDATION_BODY } ]
    [ override<METHOD_NAME_0> { METHOD_0_BODY } ]
    ...
    [ override<METHOD_NAME_P> { METHOD_P_BODY } ]
}
```

Note that there is no name for the Entity. An Entity name is its file path. Each file can contain at most one block, and the block inherits its name from the file. It is a good practice to have a comment in the first line of the file describing where the file is located.

Note. All entities have a read-only field called `id`, of its own type `Identifier()`, defined implicitly. The field `id` is used in some underlying Xua codes and cannot be removed.

Fields

Each Entity represents a class and is responsible for storing properties of its instances as data in the database. The object (instance) can have different properties, each with its type and default value. These properties are called fields and should be defined with their type and default value in the `fields` part.

Indexes

An index is a list of fields along with a boolean determining if the index is unique. Indexes are used in the database engine for faster `select` queries. All fields are assumed to be ascending by default unless the programmer specifies one as descending by a `-` sign before it, which tells the database engine to sort that field in descending order. A `*` sign at the end of an index definition makes it a unique index. If the index is marked as unique, the combination of fields must be unique in data. Note that the implicit field `id` is a unique field by default.

Validation

The body of the `validation` block is written in pure PHP.

The `validation` block is responsible for checking if an instance of the Entity is valid, and in case of invalid data, an `EntityFieldException` must be thrown. For example, assume we have a table of restaurants in an entity called `Restaurant`. This Entity has two fields, `title` and `active`. The `title` field is unique, but it is impossible to mark it as a unique index in `indexes` because we may have many inactive records sharing the same title, and the title is unique among the active restaurants. (There is a workaround here to solve this problem only using `indexes`, but for the purpose of this documentation, we assume there is not.) We can check this in the `validation` block and throw an `EntityFieldException` if the title is duplicate.

Overriding Methods

Xua generates a PHP class from each entity, extending `Entity`. This class have many methods which are possible to override. Here is a list of these methods, with the PHP method that is actually being overridden.

```
Override<initialize>   => protected static _initialize();
Override<getOne>       => protected static _getOne(Condition $condition, Order $order, string $caller);
Override<store>        => protected _store(string $caller);
Override<storeQueries> => protected _storeQueries(string $caller);
Override<delete>       => protected _delete(string $caller);
Override<getMany>      => protected static _getMany(Condition $condition, Order $order, Pager $pager, string $
Override<countMany>    => protected static _countMany(Condition $condition, Order $order, Pager $pager, string
Override<deleteMany>   => protected static _deleteMany(Condition $condition, Order $order, Pager $pager, strin
Override<setMany>      => protected static _setMany(array $changes, Condition $condition, Order $order, Pager
```

Xua provides final methods that include the actual logic, which one can use when overriding a method. Here is a list of these method names.

Name in `.xua` file	**Original Method**
`<initialize>`	_x_initialize
`<getOne>`	_x_getOne
`<store>`	_x_store
`<storeQueries>`	_x_storeQueries
`<delete>`	_x_delete
`<getMany>`	_x_getMany
`<countMany>`	_x_countMany
`<deleteMany>`	_x_deleteMany
`<setMany>`	_x_setMany

So, as an example, one can override the `storeQueries` procedure like this.

```
Override<storeQueries> {
    if (isset(static::fieldSignatures()['updatedAt'])) {
        $this->updatedAt = DateTimeService::now();
    }
    try {
        $return = static::_x_storeQueries(); // original Xua's store Queries logic
        LogService::logDatabaseChange($this);
    } catch (Exception as $e) {
        LogService::logDatabaseException($e);
    }
```

```
    return $return;
}
```

Usually, the original methods suffice, and there is no need to override them, but in case of necessity, be careful not to corrupt the functionality.

Hierarchy

Entities can come in a hierarchy just like PHP classes, and the `validation` block is a class method.

```
protected function _validation();
```

Also, the fields and indexes of a child entity override the ones in the parent. This override includes the type and default value. Also, it is possible to add new fields or indexes to the type, but it is not possible to remove existing fields. Read Examples for more details.

Visibility

The visibility of entities is controlled by overriding methods. There is an additional variable accessible in all methods, including validation, called `$caller`. This variable contains a string telling what party called this method. The values are accessible as class constants in the class `\Xua\Tools\Visibility`. These values are `Visibility::CALLER_PHP`, `Visibility::CALLER_DART`, etc.

So it is possible to block foreign callers like the following.

```
if ($caller != Visibility::CALLER_PHP) {
    throw AccessForbiddenException();
}
```

But there is more than this. It is possible to customize procedures according to the caller. For example

```
if ($caller != Visibility::CALLER_PHP) {
    if ($this->id) {
        if (!UserService::hasAccess(AccessService::MODIFY_SOME_ENTITY)) {
            throw AccessForbiddenException();
        }
    } else
        if (!UserService::hasAccess(AccessService::CREATE_SOME_ENTITY)) {
            throw AccessForbiddenException();
        }
    }
}
$this->updatedByCaller = $caller;
```

Note. Accessing The entities through `URPI` is disabled by default, and therefore the `$caller` is always PHP. One can enable this feature, but they must be super careful since it may result in severe vulnerabilities.

Special Field Types

In addition to defined supers that can be called to generate type for field types, Xua offers two categories of unusual types that make the development of a project significantly easier and faster and the resulting project more efficient and more secure. Here we try to cover these two outstanding features of Xua.

Virtual Field Supers

In some cases, one needs some fields for an entity that does not contain new data, so if defined as regular fields, this will result in duplicate/not-synced data. These supers will help mix up other fields and generate a new field that is calculated each time called but not stored. There are two types of virtual supers. One is calculated by the PHP engine and the other by the database engine. The Database Virtual Field is used when the programmer wants to use the result in a query, e.g., using in condition or order, while the PHP Virtual Field is used for more complicated mixtures of fields.

PHP Virtual Field

The PHP Virtual Field has the following signature.

```
PHPVirtualField{
    arguments {
        getter: Callback(
            nullable = false,
            parameters = [
                {
                    name:            null,
                    type:            @php(Entity::class),
                    allowSubtype:    true,
                    required:        true,
                    checkDefault:    false,
                    default:         null,
                    passByReference: false,
                },
            ]
        );
        setter: Callback(
            nullable = true,
            parameters = [
                {
                    name:            null,
                    type:            @php(Entity::class),
                    allowSubtype:    true,
                    required:        true,
```

```
                    checkDefault:    false,
                    default:         null,
                    passByReference: true,
                },
                {
                    name:            null,
                    type:            null,
                    allowSubtype:    true,
                    required:        true,
                    checkDefault:    false,
                    default:         null,
                    passByReference: false,
                },
            ]
        ) = null;
    }
    ...
}
```

PHP Virtual Field Example

Let us say we have fields `gender`, `firstName`, and `lastName` in entity `User`, and want to create a field called `title` based on these fields. We can define it like this.

```
title : PHPVirtualField (
    getter = (User $user) => {
        if ($user->gender == User::GENDER_MALE) {
            $honorific = "Mr ";
        } elseif ($user->gender == self::GENDER_FEMALE) {
            $honorific = "Miss ";
        } else {
            $honorific = "";
        }
        return $honorific . $user->firstName . " " . $user->lastName;
    }
);
```

Database Virtual Field

The Database Virtual Field has the following signature.

```
DatabaseVirtualField{
    arguments {
        getter: Callback(
            nullable = false,
            parameters = [
                {
                    name:            null,
                    type:            @php(Entity::class),
```

```
                    allowSubtype:    true,
                    required:        true,
                    checkDefault:    false,
                    default:         null,
                    passByReference: false,
                },
                {
                    name:            'params',
                    type:            'array',
                    allowSubtype:    true,
                    required:        true,
                    checkDefault:    false,
                    default:         null,
                    passByReference: false,
                },
            ]
        );
    }
    ...
}
```

Note that one cannot set a database virtual field, and therefore there is no setter method available on this field.

Note that another method argument is available called `params`, which is used to pass some extra parameters into the getter method. We discuss it in more detail in the Example section.

Database Virtual Field Example

Let us say that we have an entity called `City`, and we need a field that tells if the city is a town or a big city. At the moment, we consider cities that have a population of less than a million to be town, but this might change; either the population may change, or we may think of the area as an item, or we can have a more complicated way that involves both population and area of a city. So we cannot calculate the field `isTown` each time we need it somewhere (this may result in duplicate code). Instead, we need a field that does this so we can change it later and the change affect all usages. We can do this by defining the field `isTown` this way.

```
isTown : DatabaseVirtualField (
    getter (City $city, array $params) => {
        return "{Entity::F(self::_POPULATION)->name} < 1000000";
    }
)
```

For a more complex example, consider this scenario. Let us say that we have an entity called `Restaurant`, and we want to sort the restaurants by distance in ascending order, so we need a field called distance. A PHP Virtual Field can do this, but in that case, we need to fetch all restaurants from the database server and then sort them, which takes a significant amount of time and space. Instead, we can define it as a Database Virtual Field that allows us to use it while creating an order

expression and tell the database server to sort the restaurants by itself and give us the first page. We define this field using the following code.

```
distance : DatabaseVirtualField (
    getter (City $city, array $params) => {
        here = $params['here']; # Here, one can understand the application of the params argument.
        $lat0 = "(PI() * {$here->lat} / 180)";
        $long0 = "(PI() * {$here->long} / 180)";
        $lat1 = "(PI() * {Entity::F(self::_GEO_LAT)->name} / 180)";
        $long1 = "(PI() * {Entity::F(self::_GEO_LONG)->name} / 180)";
        $a = "(
            POWER(SIN(($lat0 - $lat1) / 2), 2) +
            COS($lat0) *
            COS($lat1) *
            POWER(SIN(($long0 - $long1) / 2), 2)
        )";
        $c = "(2 * ATAN2(POWER($a, .5), POWER(1 - $a, .5)))";
        $d = "6371000 * $c";
        return $d;
    }
)
```

Entity Relation

In almost any back-end project, some Entities are in relation with each other. For example, in a simple food delivery app, restaurants are handling orders, orders have items, items are being liked/ commented by users, users are ordering orders, restaurants are being liked/ commented by users, etc.

There is a unique and special Super called `EntityRelation` responsible for handling such relations.

But before we discuss this Super, we need to discuss different relations classes based on how we implement them.

A Little Formalism on Relation Classes

Assume $\mathcal{A}$ and $\mathcal{B}$ are two Entities, and A and B are sets of their instances, respectively. $R \subseteq A \times B$ is called a relation between $\mathcal{A}$ and $\mathcal{B}$. We define nine different classes of relations based on how we implement them. Any possible relation fits in one of these classes; actually, all of them fit in NN. However, choosing the best class when defining the database structure is a matter of restriction and efficiency.

1. $\mathsf{O11O}$ (Optional one-to-one Optional) is the class of all relations with the following conditions. \begin{eqnarray*} & i. & \forall a \in A, |\{ b \in B : aRb \}| \leq 1 \\ & ii. & \forall b \in B, |\{ a \in A : aRb \}| \leq 1 \end{eqnarray*}

2. $\mathsf{O11R}$ (Optional one-to-one Required) is the class of all relations with the following conditions. \begin{eqnarray*} & i. & \forall a \in A, |\{ b \in B : aRb \}| \leq 1 \\ & ii. & \forall b \in B, |\{ a \in A : aRb \}| = 1 \end{eqnarray*}

3. $\mathsf{R11O}$ (Required one-to-one Optional) is the class of all relations with the following conditions. \begin{eqnarray*} & i. & \forall a \in A, |\{ b \in B : aRb \}| = 1 \\ & ii. & \forall b \in B, |\{ a \in A : aRb \}| \leq 1 \end{eqnarray*}

4. $\mathsf{R11R}$ (Required one-to-one Required) is the class of all relations with the following conditions. \begin{eqnarray*} & i. & \forall a \in A, |\{ b \in B : aRb \}| = 1 \\ & ii. & \forall b \in B, |\{ a \in A : aRb \}| = 1 \end{eqnarray*}

5. $\mathsf{ON1}$ (Optional many-to-one) is the class of all relations with the following condition. \begin{eqnarray*} \forall a \in A, |\{ b \in B : aRb \}| \leq 1 \end{eqnarray*}

6. $\mathsf{RN1}$ (Required many-to-one) is the class of all relations with the following condition. \begin{eqnarray*} \forall a \in A, |\{ b \in B : aRb \}| = 1 \end{eqnarray*}

7. $\mathsf{1NO}$ (one-to-many Optional) is the class of all relations with the following condition. \begin{eqnarray*} \forall b \in B, |\{ a \in A : aRb \}| \leq 1 \end{eqnarray*}

8. $\mathsf{1NR}$ (one-to-many Required) is the class of all relations with the following condition. \begin{eqnarray*} \forall b \in B, |\{ a \in A : aRb \}| = 1 \end{eqnarray*}

9. NN (many-to-many) is the class of all relations.

The Signature

The `EntityRelation` has the following signature.

```
EntityRelation{
    arguments {
        # Standard Arguments
              relatedEntity : Universal(                          )        ;
              relation      : Enum     (values   = self::REL_     )        ;
              invName       : Symbol   (nullable = true           ) = null ;
        # Constant Arguments
        const fromMany      : Boolean  (                          ) = false;
        const fromOne       : Boolean  (                          ) = false;
        const toMany        : Boolean  (                          ) = false;
        const toOne         : Boolean  (                          ) = false;
        const is11          : Boolean  (                          ) = false;
        const isN1          : Boolean  (                          ) = false;
        const is1N          : Boolean  (                          ) = false;
        const isNN          : Boolean  (                          ) = false;
        const optional      : Boolean  (                          ) = false;
        const nullable      : Boolean  (                          ) = false;
        const required      : Boolean  (                          ) = false;
        const invOptional   : Boolean  (                          ) = false;
        const invRequired   : Boolean  (                          ) = false;
        const hasJunction   : Boolean  (                          ) = false;
        # Definition Side Arguments
              definedOn     : Enum     (values   = self::DEFINED_ON_)        ;
        const definedHere   : Boolean  (                          ) = false;
        const definedThere  : Boolean  (                          ) = false;
        const columnHere    : Boolean  (                          ) = false;
```

```
        const columnThere   : Boolean  (                          ) = false;
    }
    ...
}
```

Related Entity

To create a relation R between two Entities $\mathcal{L}$ and $\mathcal{R}$, one has to define a field that represents R with a type generated from the `EntityRelation` Super. The field must be defined on $\mathcal{L}$ (called the Left Entity), and the `relatedEntity` must be set to $\mathcal{R}$ (called the Right Entity).

Relation Class

The `relation` argument determines the class of the relation and has one of the following values.

```
EntityRelation::REL_011O
EntityRelation::REL_011R
EntityRelation::REL_R11O
EntityRelation::REL_R11R
EntityRelation::REL_ON1
EntityRelation::REL_RN1
EntityRelation::REL_1NO
EntityRelation::REL_1NR
EntityRelation::REL_NN
```

Name & Inverse Name

One can use this field to get all the related rows of a row in the database. The `invName` argument is used to do the inverse job. We try to make it clear by an example.

Assume one defines a field called `rel` on the `LeftEntity` like this.

```
LeftEntity {
    fields {
        ...
        rel : EntityRelation(
            relatedEntity = @php(RightEntity::class),
            relation = EntityRelation::REL_NN,
            invName = 'invRel',
        )
        ...
    }
    ...
}
```

The Xua engine automatically generates an implicit field like this.

```
RightEntity {
    fields {
        ...
        invRel : EntityRelation(
            relatedEntity = @php(LeftEntity::class),
            relation = EntityRelation::REL_NN,
            invName = 'rel',
        )
        ...
    }
    ...
}
```

Now, one can access the related instances using these fields.

```
$l = new LeftEntity();
// $l->rel is the set of all instances of RightEntity $r s.t. $l is in relation with $r.
$r = new RightEntity();
// $r->invRel is the set of all instances of LeftEntity $l s.t. $l is in relation with $r.
```

Note. In the X-to-one cases, the result of retrieving a field is not a set but instead a value that can be an empty Entity in optional cases.

Note. The `invName` argument is optional, and if it is not provided, the Xua engine does not generate the implicit inverse field.

Constant Arguments

`EntityRelation` offers a set of constant arguments calculated based on the relation class, that help with recognizing a relation better. The names are pretty explanatory by themselves, but here we provide the way we calculate each.

```
$this->fromMany     = in_array($this->relation, [self::REL_NN, self::REL_ON1, self::REL_RN1]);
$this->fromOne      = !$this->fromMany;
$this->toMany       = in_array($this->relation, [self::REL_NN, self::REL_1NO, self::REL_1NR]);
$this->toOne        = !$this->toMany;
$this->is11         = ($this->fromOne and $this->toOne);
$this->isN1         = ($this->fromMany and $this->toOne);
$this->is1N         = ($this->fromOne and $this->toMany);
$this->isNN         = ($this->fromMany and $this->toMany);
$this->optional     = in_array($this->relation, [self::REL_011O, self::REL_011R, self::REL_ON1]);
$this->nullable     = $this->optional;
$this->required     = !$this->optional;
$this->invOptional  = in_array($this->relation, [self::REL_011O, self::REL_R11O, self::REL_1NO]);
$this->invRequired  = !$this->invOptional;
$this->hasJunction  = $this->isNN;
```

Definition Side Arguments

There is a particular argument called `definedOn` which can either be `here` or `there`. This argument is not to be filled by the Xua programmer. If a relational field is defined explicitly on an Entity, the Xua engine sets this argument to `here`, and for the implicit inverse field, the value of this argument is `there`. There are also some constant fields in this regard; all of these arguments are used by the Xua core to decide how to store and process data.

```
$this->definedHere  = ($this->definedOn == self::DEFINED_ON_HERE);
$this->definedThere = ($this->definedOn == self::DEFINED_ON_THERE);
$this->columnHere   = (($this->is11 and $this->definedHere) or $this->isN1);
$this->columnThere  = (($this->is11 and $this->definedThere) or $this->is1N);
```

Entity Relation Example

For example, assume we have two Entities called `User` and `City`. Further, assume we want a field in the `User` entity called `currentCity`, and we want this field to refer to a row of the `City` table. We have to add the following field to the `User` entity.

```
User {
    fields {
        ...
        currentCity : EntityRelation(
            relatedEntity = @php(City::class),
            relation = EntityRelation::REL_RNI,
            invName = 'citizens',
        )
        ...
    }
    ...
}
```

Relation. The `EntityRelation::REL_RNI` stands for a many-to-one relation required on the left side, which means that no user can be in more than one city simultaneously but needs to be in a city, although one city can have many citizens at once.

Inverse Name. Here, we created a field called `currentCity` that shows us a relation between users and cities, and we can get related cities of a user by `$user->currentCity`, but how can we get related users of a city? The Xua engine generates an implicit field based on the `invName`. So `$city->citizens` gives us the list of all users that their `currentCity` is `$city`.

Helper Classes

Field Class

Each field defined under a name is accessible using the static method `Entity::F({fieldName})` of the entity class. This property is an instance of a class called `EntityFieldSignature`. This class has the following structure.

```
public string   $entity,
public string   $name,
public Super    $type,
public mixed    $default = null,
public function p(?array $param = null): array|EntityFieldSignature;
```

Field Class Example

Assume the Database Virtual Field Example we have provided above. Here we work a little around the field `$restaurant->distance`.

Notations `Restaurant::fieldSignatures()[Restaurant::DISTANCE]` and `Entity::F(Restaurant::_DISTANCE)` both refer to the same value, an instance of class `EntityFieldSignature` describing this field. So we know the following expressions are true.

```
Entity::F(Restaurant::_DISTANCE)->entity  == Restaurant::class;
Entity::F(Restaurant::_DISTANCE)->name    == 'distance';
Entity::F(Restaurant::_DISTANCE)->type    == DatabaseVirtualField (...);
Entity::F(Restaurant::_DISTANCE)->default == null;
// The parameters of a field is null by default.
Entity::F(Restaurant::_DISTANCE)->p()     == null;
// We can modify the parameters of a field by calling the method `p` on its signature.
Entity::F(Restaurant::_DISTANCE)->p(['here' => (object)['lat' => 42, 'long' => 59]]); // https://en.wikipedia.
// Now it is set.
Entity::F(Restaurant::_DISTANCE)->p()     == ['here' => (object)['lat' => 42, 'long' => 59]];
```

Conditional Field Class

The `ConditionField` class is similar to the regular field class but has some features used in defining conditions and orders, which we discuss later. The Condition Field of each field is accessible as the static method `Entity::C({fieldName})` of the entity class. This class has the following structure.

```
public function __construct(public EntityFieldSignature $signature);
public function rel(ConditionField $conditionField): static;
public function name() : string;
public function joins(): array;
```

So the `Entity::C({fieldName})` is actually `new ConditionField(Entity::F({fieldName}))`, an instance of `ConditionField` based on the field signature.

The critical feature of this class is the `rel` method, which makes it possible to access fields on the related tables. We cover this in the Condition and Order sections.

The methods `name` and `joins` are not usually helpful for the Xua programmer and are used in the Xua core. So we do not cover them here.

Condition

Condition is a Xua built-in class that we use to create `WHERE` expressions with.

Theoretically speaking, a condition node is a deciding machine that inputs a row of a specific table and returns a boolean that indicates whether the given row is accepted in the condition or not.

Each instance of Condition is a node in a semi-binary tree. There are two ways two create an instance of this class. One is to create a new node (a leaf), and the other is to operate on existing nodes.

Leaf Condition

There are three common types of leaf conditions: relational leaf, true leaf, and false leaf. However, one can create a custom leaf as a raw leaf.

Relational Leaf

The method for creating a relational leaf is the following.

```
Condition::leaf(ConditionField $field, string $relation, mixed $value = null);
```

This method will create a condition asserting that the field `$filedName` must be in `$relation` relation with the value `$value`.

Field

As described above, for each entity, conditional fields are available as instances of `ConditionField` using the static methods of the form `Entity::C({fieldName})`.

Relation

XUA also provides relation constants as class constants of the `Condition` class. These constants are listed below.

Name	SQL equivalent	Description
`Condition::GRATER`	`>`	Greater than
`Condition::NGRATER`	`<=`	Negation of `GRATER`
`Condition::GRATEREQ`	`>=`	Greater than or equal
`Condition::NGRATEREQ`	`<`	Negation of `GRATEREQ`

Name	SQL equivalent	Description
`Condition::LESS`	`<`	Less than
`Condition::NLESS`	`>=`	Negation of `LESS`
`Condition::LESSEQ`	`<=`	Less than or equal
`Condition::NLESSEQ`	`>`	Negation of `LESSEQ`
`Condition::EQ`	`=`	Equal
`Condition::NEQ`	`!=`	Negation of `EQ`
`Condition::NULLSAFEEQ`	`<=>`	NULL-safe equal
`Condition::NNULLSAFEEQ`	`!(... <=> ...)`	Negation of `NNULLSAFEEQ`
`Condition::BETWEEN`	`BETWEEN ... AND ...`	Whether a value is within a range of values (This relation requires the argument `$value` to be an array consisting of two values)
`Condition::NBETWEEN`	`NOT BETWEEN ... AND ...`	Negation of `BETWEEN`
`Condition::IN`	`IN`	Whether a value is within a set of values (This relation requires the argument `$value` to be an array of values)
`Condition::NIN`	`NOT IN`	Negation of `IN`
`Condition::IS`	`IS`	Test a value against a boolean
`Condition::NIS`	`IS NOT`	Negation of `IS`
`Condition::ISNULL`	`IS NULL`	NULL value test (This relation does not depend on value of argument `$value`)
`Condition::NISNULL`	`IS NOT NULL`	Negation of `ISNULL` (This relation does not depend on value of argument `$value`)
`Condition::LIKE`	`LIKE`	Simple pattern matching

Name	SQL equivalent	Description
`Condition::NOT_LIKE`	`NOT LIKE`	Negation of `LIKE`
`Condition::REGEXP`	`REGEXP`	Simple regular expression pattern matching
`Condition::NOT_REGEXP`	`NOT REGEXP`	Negation of `REGEXP`

Value

The value argument must usually fit in the field type. However, in some cases (some relations), one value does not suffice (e. g. `Condition::BETWEEN`), so we need to provide two values of the field type as an array.

Condition on Related Entities

To create a condition node that presents a condition on one of the related entities, one can use the `rel` method on condition field instances.

Example

For example, a condition node that asserts that a person is born in the '90s would look like the following.

```
Condition::leaf(
    Entity::C(User::_BIRTH_DATE),
    Condition::BETWEEN,
    [
        DateTimeInstance::fromGregorianYmd('1990-1-1'),
        DateTimeInstance::fromGregorianYmd('2000-1-1')
    ]
)
```

As an other example, assume we want to have a condition that refers to all posts written by people living in Palo Alto. We may use the following node.

```
Condition::leaf(
    Entity::C(Post::AUTHOR)
        ->rel(Entity::C(User::_LIVING_IN))
        ->rel(Entity::C(AdministrativeDivision::_TITLE)),
    Condition::EQ,
    'Palo Alto'
)
```

True Leaf & False Leaf

The special leaf methods `Condition::trueLeaf()` and `Condition::falseLeaf()` are always true and false respectively.

Raw Leaf

The raw leaf method is described as follows.

```
Condition::rawLeaf(string $template, array $parameters = [], array $joins = [])
```

This method is used to inject pure and raw SQL script to the WHERE expression in case other leaf methods do not satisfy the programmer's needs. The use of this method is highly discouraged as the other methods are powerful enough to satisfy almost all of the programmer's needs. Still, a somehow funny usage of this method would be the following.

```
Condition::rawLeaf("RAND() > ?", [0.5])
```

We will discuss parameter binding used in the above code in more detail later.

Operations on Conditions

In addition to the above methods for creating new leaf nodes, Xua provides some methods to create new nodes using existing nodes.

Logical Operators

four logical operators are used to create a new node using existing nodes. These operators are binary-operators AND, OR, XOR, and unary-operator NOT.

Classic Operators

The primitive way to create a new node is to provide operands to an operator and receive a new node as the return value. The four methods for this are the followings.

```
public static function _and_(Condition $leftCondition, Condition $rightCondition): Condition;
public static function _or_(Condition $leftCondition, Condition $rightCondition): Condition;
public static function _xor_(Condition $leftCondition, Condition $rightCondition): Condition;
public static function _not_(Condition $condition): Condition;
```

High-Level Operators

Although the described methods can theoretically create any condition, Xua provides some methods to improve code readability.

C Operators

The first group is called the C (Condition) group. These methods are used on a node to connect it to another node.

```
public function andC(Condition $condition): Condition;
public function orC(Condition $condition): Condition
public function xorC(Condition $condition): Condition;
public function not(): Condition;
```

For example, take a look at the following code.

```
$l = Condition::trueLeaf();
// $l is now rendered as 'TRUE'
$l->andC(Condition::falseLeaf());
// $l is now rendered as '(TRUE) AND (FALSE)'
$l->not();
// $l is now rendered as 'NOT ((TRUE) AND (FALSE))'
```

Common Operators

These methods are used on a node to connect it to a relational node, but the creation of the second node is embedded.

```
public function and(ConditionField $field, string $relation, mixed $value = null): Condition
{
    return $this->andC(Condition::leaf($field, $relation, $value));
}
public function or(ConditionField $field, string $relation, mixed $value = null): Condition
{
    return $this->orC(Condition::leaf($field, $relation, $value));
}
public function xor(ConditionField $field, string $relation, mixed $value = null): Condition
{
    return $this->xorC(Condition::leaf($field, $relation, $value));
}
```

R Operators

This group is called the R (Raw) group. These methods are used on a node to connect it to a raw node, but the creation of the second node is embedded.

```
public function andR(string $template, array $parameters = [], array $joins = []): Condition
{
    return $this->andC(Condition::rawLeaf($template, $parameters, $joins));
}
public function orR(string $template, array $parameters = [], array $joins = []): Condition
{
    return $this->orC(Condition::rawLeaf($template, $parameters, $joins));
}
public function xorR(string $template, array $parameters = [], array $joins = []): Condition
{
```

```
    return $this->xorC(Condition::rawLeaf($template, $parameters, $joins));
}
```

Order

Order is a Xua built-in class that is usually used to create `ORDER BY` expressions.

To create an instance of this class, one needs to start with an empty instance and append order expressions. Each new order appended has a lesser priority than the last one. The bootstrap instance is created using the following method.

```
Order::noOrder(): Order;
```

To append a raw order expression in SQL syntax, one may use the following method, but it is highly discouraged.

```
public function addRaw(string $order): Order;
```

To append an order based on an entity field, one may use the following method, which is preferred.

```
public function add(ConditionField $field, string $direction): Order
{
    return $this->addRaw($field->name() . ' ' . $direction);
}
```

Note that the direction can be one of the values `Order::ASC` and `Order::DESC`.

There are some other come in handy methods to use in order to avoid using the `addRaw` method, such as the following.

```
public function addRandom(): Order
{
    return $this->addRaw('RAND()');
}
```

Pager

Pager is a Xua built-in class that is usually used to generate `LIMIT ... OFFSET ...` expressions.

Each class instance contains these arguments (`limit` and `offset`) as properties, but it is a good practice not to modify them manually. The class constructor takes two arguments, `$pageSize` and `$pageIndex`, and calculates the limit and offset values according to them. It is possible to modify these arguments as well. There is also one specific instance accessible using `Pager::unlimited()`, which is used to fetch all rows. The following properties and methods are present in the pager class.

Properties	Description
`private int $pageSize;`	Page Size
`private int $pageNumber;`	Page Number

Methods	Description
`<em></em>construct(int $pageSize, int $pageNumber);`	Creates a new instance with the given data.
`next();`	Goes to the next page.
`previous();`	Goes to the previous page.
`getPageSize();`	Returns page size.
`setPageSize(int $pageSize);`	Sets page size.
`getPageNumber();`	Returns page number.
`setPageNumber(int $pageNumber);`	Sets page number.
static `pages(Condition $condition, int $pageSize);`	Returns the number of pages if we store rows of `$entityName` under $condition in pages of size `$pageSize`.

Note. The pages start from one.

Practicalities

Deploy & Alters

When a new Entity is created, or modified, these changes should be synced with the MySQL server. There is a particular service called `EntityAlterService`, which is responsible for doing this. The static method `alters(): string` on this class adds newly created entities as tables to the database by itself and returns `ALTER` queries for modified entities. This method does not run the alters by itself, instead, it prints out the alters; which are possible to run by the programmer or another person in charge.

The convention is to use this method in the deploy procedure, check for the alters, and generate a warning if the database structure is not synced with the program. It is also a convention to give developers the ability to complete the deploy process with the undone alters. This process is called

force deploy, and the alters which are possible to ignore while deploying are called force-friendly alters.

Usage

It is possible to use a defined entity in both server project (PHP language) and front-end native codes via the Marshal Library, but not in the Xua source codes. The usage in the Marshal Library is controlled via the visibility of entity methods and the modification of URPI, and it is not that different from the PHP usage.

In this section, we try to cover all methods, properties, and constants available on an Entity class.

Properties & Constants

Xua generates a class property and a class constant for each field defined in an entity. The property is defined on each instance and refers to the corresponding value in the database, while the constant (named `<em>{FIELD</em>NAME<em>IN</em>SCREAMING<em>SNAKE</em>CASE}`) contains the field name. For example for field `birthDate` on `User`, Xua generate `$user->birthDate` and `User::<em>BIRTH</em>DATE = 'User.birthDate'`. This constant can be used to get Field and Conditional Field instances using `Entity::F(User::<em>BIRTH</em>DATE)` and `Entity::C(User::<em>BIRTH</em>DATE)` respectively.

Helpers

@TODO

Execute

@TODO

Field Signatures

@TODO

Indexes

@TODO

Fields

@TODO

Table

@TODO

VARQUE Actions

In order to discuss other entity methods, we need to discuss the VARQUE actions first.

There are different actions that one can take on a database. We divide these functionalities into three categories: Select Data Actions, Create/Modify Data Actions, and Delete Data Actions. We

may also divide the functions based on whether they act on a single row or a group of rows, which takes us to two categories Singular Actions and Conditional Actions. This gives us six types of actions, which we name each one as follows.

	Retrieve	**Create/Modify**	**Delete**
Singular	View	Adjust	Remove
Conditional	Query	Update	Eliminate

These actions are called the **VARQUE** (**V**iew, **A**djust, **R**emove, **Q**uery, **U**pdate, **E**liminate) actions.

View

Two methods are used to retrieve a single row of a table. The first one has the following signature.

```
static function getOne(
    ?Condition $condition = null,
    ?Order $order = null,
    string $caller = Visibility::CALLER_PHP
): static;
```

This function takes an instance of the Condition class and an instance of the Order class (both are described before) and returns the first row (ordered by `$order`) that matches the `$condition` criteria. If no such row is found, it returns an empty instance of the entity. To check if an instance of an entity class is empty, one can check if the `id` property is null or not (`$entity->id === null` or not).

The second one is the class constructor.

```
function __construct(?int $id = null);
```

This function is just an alias for the following call, designed for simplicity.

```
EntityName::getOne(
    Condition::leaf(
        Entity::C(EntityName::_ID),
        Condition::EQ,
        $id
    )
);
```

Query

@TODO `getMany` and `count`

Adjust

@TODO `store`

Update

@TODO `storeMany` and `EntityBuffer`

Remove

@TODO `delete`

Eliminate

@TODO `deleteMany`

METHODS;

TINY BLOCKS OF COMPUTATION

Introduction

@TODO

VARQUE METHODS;

@TODO

Introduction

@TODO

SERVICES;

@TODO

Introduction

@TODO

PREDEFINED SERVICES;

@TODO

Introduction

@TODO

```
Back: RouteService
Back: Exception Service
Front: Http Connection Service
```

DOCUMENTATIONS;

@TODO

Introduction

@TODO

any base folder is part, all files inside are chapters (pages), and inside files there are sections(h1),

```
subsections(h2), subsubsections(h3), etc.
```

CONGRATULATIONS;

FINIS CORONAT OPUS

It's Over, It's Done

Nice to finally meet you here. Congratulations. We are thrilled that you took the time to read this e-book about the Xua Language and wish you to use this tool to build some great back-end applications.

Please keep in touch with the development team using kmirzavaziri@gmail.com and let us know your opinions and comments.

We also appreciate it if you want to contribute to this project since this is an open-source project and cannot grow without your contributions.