Data as Code

The ThingsDB Book

First Edition

JEROEN VAN DER HEIJDEN

Data as Code - The ThingsDB Book

Author

Jeroen van der Heijden

Editors

Anja Bruls
Sasha Tychkovska

Pre Release
v7

Table of Contents

Preface

The primary objective of this book is to equip readers with a comprehensive understanding of how ThingsDB functions. It is not intended to be an all-inclusive reference guide and therefore does not cover every aspect of the platform. ThingsDB can be applied in a diverse range of applications, and once its potential is fully grasped, it has the power to inspire novel and innovative solutions. Our aim is to provide a balanced overview of ThingsDB's capabilities while also fostering creativity and encouraging readers to explore its multifaceted applications.

Consistent with a simplified approach, we present code blocks in this book without syntax highlighting. While sophisticated IDEs like vscode and dedicated tools like *things-prompt* and *ThingsGUI* provide syntax highlighting, coloring known functions, punctuation, etc., we've adopted a more straightforward approach for this book.

```
Dark blue for prompts
Orange for code that the user needs to enter
Green for comments
Light blue for responses
```

The first five chapters of this book can be read independently of each other and can be skipped if you have a strong grasp of the concepts covered. You can assess your understanding by completing the quizzes at the end of each chapter. However, from chapter 6 until chapter 13, the book will focus on developing a "to-do" application using the "todos" collection. The concepts introduced in these chapters build upon each other, making it essential to follow them in sequence to execute the code examples and fully comprehend the presented material. If you really want to jump directly into a specific chapter, you can download the necessary collection state from our website, which is explained in Appendix I - Chapter Data Import.

Installation

Like with learning any programming language, ThingsDB requires a setup to execute code examples. This setup includes at least one ThingsDB node, which acts as the interpreter responsible for executing the code. Throughout the first chapters of this book, we will primarily communicate with ThingsDB using the *"things-prompt"* application, a tool designed for running small code snippets. As we progress, we will delve into more practical examples and explore the realm of event-driven programming, showcasing the true power of ThingsDB when interacting with other programming languages. While this book includes some Python examples, you are free to choose a language that suits your preference, such as C#, Go, PHP, JavaScript/Node.js or another supported language. The provided examples are relatively straightforward to adapt to other programming languages.

Installation - Node

The process running ThingsDB is called a node. ThingsDB is built to run across multiple nodes for high availability and scalability but for most chapters in this book, running a single node will suffice.

Exploring Multiple Nodes:

If you are interested in experimenting with distributed setups, you can run multiple ThingsDB nodes on your machine. However, this involves manually setting up each node with unique data paths and ports (covered in detail in Chapter 15). Be aware that this process can be cumbersome.

Docker to the Rescue:

Docker offers a simpler and more efficient way to manage multiple ThingsDB nodes. Docker is a containerization platform that packages applications and their dependencies into self-contained units, ensuring consistent and portable environments.

Docker

This guide will walk you through installing and running ThingsDB using Docker Compose, a convenient tool for managing Docker configurations.

> 💡 **Prefer manual installation?** Skip this section and proceed to the guide for installing ThingsDB from source code on your specific operating system Linux, Mac, or Windows (WSL). Choose the instructions that match your system.

If you do not already have Docker installed, you can download it from here: https://docs.docker.com/get-docker/

Check your docker version: *(version 23 or higher is required)*

```
$ docker -v
Docker version 25.0.3, build 4debf41
```

Create a project directory and navigate to that directory:

```
$ mkdir thingsdb_book
$ cd thingsdb_book
~/thingsdb_book$
```

Paste the following content into a new file named `docker-compose.yml` in your project directory: (or download the file from here: https://docs.thingsdb.io/v1/book/docker-compose.yml)

```
x-ti-template: &ti
  image: ghcr.io/thingsdb/node
  restart: unless-stopped
  environment:
    - THINGSDB_BIND_CLIENT_ADDR = '::'
    - THINGSDB_BIND_NODE_ADDR = '::'
    - THINGSDB_HTTP_API_PORT = 9210
    - THINGSDB_HTTP_STATUS_PORT = 8080
    - THINGSDB_WS_PORT=9270
    - THINGSDB_MODULES_PATH = /modules/
    - THINGSDB_STORAGE_PATH = /data/
services:
  node0:
    << : *ti
    hostname: node0
    container_name: node0
    command: "--init"
    ports:
      - 9200:9200
      - 9210:9210
      - 9270:9270
      - 8080:8080
    volumes:
      - ./node0/data:/data/
      - ./node0/modules:/modules/
      - ./node0/dump:/dump/

## Uncomment the following sections to add more nodes (optional)
# node1:
#   << : *ti
#   hostname: node1
#   container_name: node1
#   command: "--secret pass"
#   ports:
#     - 8081:8080
#   volumes:
#     - ./node1/data:/data/
#     - ./node1/modules:/modules/
#     - ./node1/dump:/dump/
```

Make sure you are in the directory where your `docker-compose.yml` file is located and run the following command to pull the image:

```
~/thingsdb_book$ docker compose pull
…
```

Use this command to launch ThingsDB as a persistent background service:

```
~/thingsdb_book$ docker compose up -d
[+] Running 1/1
✔ Container node0  Started
```

13

> 💡 You'll see output indicating that the container is starting. Don't worry if only one node is initially active; you can learn about scaling to multiple nodes in Chapter 15.

You've successfully set up Docker. Let's move on to installing Python.

Linux

While building ThingsDB from source is possible on various Linux distributions, this documentation details the process for Ubuntu due to its popularity. We recognize other distributions might work, but they are outside the scope of this guide.

Step 1: Update and Install Packages:

Start with updating the packages list:

```
$ sudo apt update
…
```

Then, install the required packages:

```
$ sudo apt-get install -y \
    libuv1-dev \
    libpcre2-dev \
    libyajl-dev \
    libcurl4-nss-dev \
    libssl-dev \
    build-essential \
    cmake \
    git
…
```

Step 3: Clone and Build ThingsDB

Clone the ThingsDB repository:

```
$ git clone https://github.com/thingsdb/ThingsDB.git
…
```

Build ThingsDB:

```
$ ./ThingsDB/release-build.sh
…
```

Step 4: Create a Symlink (Optional)

To easily start ThingsDB from any directory, create a symlink:

```
$ sudo ln -sr ./ThingsDB/thingsdb /usr/bin/thingsdb
```

Step 5: Verify Installation

Check if ThingsDB is installed correctly:

```
$ thingsdb --version
   _____ _   _              ____  _____
  |_    _| |_|_|___ ___ ___|    \| __  |
    | | |   | |   | . |_ -|  |  | __ -|
    |_| |_|_|_|_|_|_  |___|____/|_____|   version: 1.6.0
                  |___|
ThingsDB Node 1.6.0
Maintainer: Jeroen van der Heijden <jeroen@cesbit.com>
Home-page: https://thingsdb.io
```

You've successfully built ThingsDB from source. Let's move on to
installing Python.

Mac

The section describes building ThingsDB on MacOS.

Prerequisites:

- **MacOS with Homebrew installed:** If you do not have Homebrew installed, follow the instructions on https://brew.sh
- **An active Internet connection**

Step 1: Update and Install Packages

Update the package lists:

```
% brew update
…
```

Then, install necessary packages:

```
% brew install cmake libuv pcre2 yajl curl
…
```

Step 2: Clone and Build ThingsDB

Clone the ThingsDB repository:

```
% git clone https://github.com/thingsdb/ThingsDB.git
…
```

Build ThingsDB:

```
% ./ThingsDB/release-build.sh
…
```

Step 3: Create a Symlink (Optional)

To easily start ThingsDB from any directory, create a symlink:

```
% ln -s ~/ThingsDB/thingsdb /opt/homebrew/bin/thingsdb
```

Step 4: Verify Installation

Check if ThingsDB is installed correctly:

```
% thingsdb --version

   _____ _   _                 ____  _____
  |_    _| |_|_|___ ___ ___|    \| __  |
    | | |    | |    | . |_ -|    |   | __ -|
    |_| |_|_|_|_|_|_  |___|____/|_____|    version: 1.6.0
                    |___|
ThingsDB Node 1.6.0
Maintainer: Jeroen van der Heijden <jeroen@cesbit.com>
Home-page: https://thingsdb.io
```

Next Steps:

Now you can start ThingsDB by typing `thingsdb` in your terminal. Let's move on to installing Python.

Windows (WSL)

While ThingsDB does not run natively on Windows, you can leverage WSL (Windows Subsystem for Linux) to create a Linux environment and enjoy all its functionalities! Here's how:

Prerequisites:

- **Windows 10 version 1903 or later (with WSL 2 enabled):** Check your version and enable WSL if needed: https://learn.microsoft.com/en-us/windows/wsl/
- **An active Internet connection**

Step 1: Install Ubuntu

Open a PowerShell window and run:

```
PS C:\Users\admin> wsl --install -d Ubuntu
Installing: Ubuntu
…
```

Follow the on-screen instructions to provide a username and password for your Ubuntu environment.

Step 2: Update and Install Packages

Once Ubuntu is installed, open a WSL terminal (start menu > search for "Ubuntu") and update the package lists:

```
ti@LAPTOP:~$ sudo apt update
…
```

Then, install necessary packages:

```
ti@LAPTOP:~$ sudo apt-get install -y \
    libuv1-dev \
    libpcre2-dev \
    libyajl-dev \
    libcurl4-nss-dev \
    libssl-dev \
    build-essential \
    cmake
…
```

Step 3: Clone and Build ThingsDB

Clone the ThingsDB repository:

19

```
ti@LAPTOP:~$ git clone https://github.com/thingsdb/ThingsDB.git
…
```

Build ThingsDB:

```
ti@LAPTOP:~$ ./ThingsDB/release-build.sh
…
```

Step 4: Create a Symlink (Optional)

To easily start ThingsDB from any directory, create a symlink:

```
ti@LAPTOP:~$ sudo ln -sr ./ThingsDB/thingsdb /usr/bin/thingsdb
```

Step 5: Verify Installation

Check if ThingsDB is installed correctly:

```
ti@LAPTOP:~$ thingsdb --version
    _____  _    _                   ____  _____
   |_    _| |_|_|___ ___ ___|     \|  __  |
     | | |    | |    | . |_ -|   |   | __ -|
     |_| |_|_|_|_|_|_  |___|____/|_____|    version: 1.6.0
                   |___|
ThingsDB Node 1.6.0
Maintainer: Jeroen van der Heijden <jeroen@cesbit.com>
Home-page: https://thingsdb.io
```

Next Steps:

Now you can start ThingsDB by typing `thingsdb` in your WSL terminal.
Remember to open a WSL terminal (e.g., Ubuntu) every time you want to use
ThingsDB.

Installation - Python

While the code examples in this book are compatible with Python versions as old as 3.7, it is recommended to use version 3.8 or higher to ensure optimal performance and stability. We won't cover the complete Python installation process in this guide, as there is plenty of documentation available online. You should also check if Python is already installed on your system as it is often pre-installed.

After installing Python, install PIP as well. PIP is the package manager for Python and is the simplest way to install the ThingsDB Python module. You can install the module using the following command:

> 💡 MacOS and some Linux distributions like Debian 12 have an older version of Python pre-installed. To ensure you are working with Python 3 and avoid compatibility issues, use `pip3` and `python3` for managing your packages on these systems.

```
$ pip install python-thingsdb
…
Successfully installed deprecation-2.1.0 msgpack-1.0.7 python-thingsdb-
1.0.7
```

To verify that the installation was successful, open Python and try to import the Client from the thingsdb module.

```
$ python
>>> from thingsdb.client import Client
>>>
```

If this command runs without errors, the installation was successful.

Installation - ThingsDB Prompt

Just like installing the ThingsDB Client module, you can install ThingsDB Prompt using PIP, Python's package manager.

```
$ pip install thingsprompt
…
Successfully installed install-1.3.5 setproctitle-1.3.3 thingsprompt-1.0.9
```

Once the installation is complete, ThingsDB Prompt will be installed in your system's PATH. You should now be able to run it directly from your terminal. To verify the installation, run the following command:

> 💡 For users encountering issues executing the `things-prompt` command due to the script directory not being included in their system's PATH environment variable, an alternative invocation method exists. Users can bypass the PATH requirement by directly invoking the module via `python -m thingsprompt` (note the lack of hyphen). This approach is particularly advantageous in scenarios where multiple Python environments host ThingsDB Prompt installations, and precise environment selection is crucial.

```
$ things-prompt --version
1.0.9
```

If the version is displayed, the installation was successful.

> 💡 There is also a ThingsGUI application for communicating with ThingsDB. This application is more user-friendly and visually appealing than the ThingsDB Prompt. However, we won't explicitly use or document the ThingsGUI in this book's examples, simply because we prefer to demonstrate code examples. Nonetheless, keep in mind that this tool is available if you prefer a graphical and easy to use interface.

Getting Started - Initial Setup

If you have not launched ThingsDB before, you will need to invoke the `--init` argument to initiate the creation of necessary files within a designated data directory, serving as the repository for all ThingsDB entities.

> 💡 If you are using Docker Compose as described in the Docker installation section, navigate to your `docker-compose.yml` directory and run: `docker compose up -d`
>
> This starts ThingsDB in the background, automatically initializing it with the `--init` argument, ensuring it's ready for use.

```
$ thingsdb --init

  _____  _    _               ____  _____
 |_    _| |_|_|___ ___ ___|    \| __  |
   | | |   | |    | . |_ -|  |  | __ -|
   |_| |_|_|_|_|_|_  |___|____/|_____|    version: 1.6.0
                 |___|
```

> 💡 This book is written based on ThingsDB version 1.6.0, and some examples may not work with older versions. It is always best to use the latest version of ThingsDB to ensure compatibility and avoid potential issues.

In a separate shell, we can establish a connection to ThingsDB using the *things-prompt* tool. We'll utilize this tool for some straightforward exercises in this book. The *things-prompt* is one of several methods for interacting with the interpreter.

> 💡 For more intricate (and real-world) examples, we'll employ Python code examples due to their ease of comprehension. If you wish to follow the code examples in this book, it is advisable to install Python and the ThingsDB client (python-thingsdb) library. However, you are free to utilize any programming language you prefer, as the focus will be on the ThingsDB code itself, not the specific Python code.

Okay, lets connect to the ThingsDB interpreter:

```
$ things-prompt -u admin -p pass -s //stuff
127.0.0.1:9200 (//stuff)>
```

To use the command we must specify a username with the `-u` argument. Optionally, we can also provide a password using the `-p` argument (if not given, the prompt will ask for the password). We can also provide a scope with the `-s` argument and choose the `//stuff` scope which represents the "stuff" collection. A *collection* acts like a container for your data, it is the place where things are stored. You can create as many collections as you need. When you start ThingsDB for the first time, a single collection "stuff" is created which is used in this first example.

The prompt shows the IP address (or hostname) of the ThingsDB *node* you are connected to. It also includes the port number and the scope where the queries will run in.

> In many cases, it is unnecessary to display connection information in the prompt. Therefore, ThingsDB Prompt provides an optional argument, `--hide-connection-info`, which suppresses the display of the connection address and port. In this book, we'll utilize this argument for cleaner code examples.

If you start using ThingsDB in a production environment, you should start with more than one node, preferably three or more. ThingsDB is designed from the start to run on multiple nodes and utilizes multiple nodes to perform, for example, garbage collection without blocking! Garbage collection is a problem for languages like Python, Go and JavaScript. Although it is usually fast, it might result in shortly blocking applications, especially when the application requires a lot of memory due to the initialization of many objects. ThingsDB has solved this problem by ensuring garbage collection will never run on more than one node at the same time. Queries which are received by the node who is busy with garbage collection will forward the query to another node so your client will not notice anything. The same technique is used for creating backups as well.

Congratulations! You're now equipped to embark on your ThingsDB journey. With this book as your guide, you'll gain a comprehensive understanding of this powerful data management solution and its capabilities. So, dive in, explore the concepts, and let ThingsDB unleash your creativity and innovation.

Chapter 1 - Introduction to ThingsDB

While ThingsDB is technically a programming language, it breaks the mold by not being primarily designed for writing traditional programs. Instead, it is geared towards providing developers with an intuitive and flexible approach to storing and retrieving data. This might seem more akin to SQL, another language primarily focused on data management. However, ThingsDB's syntax and structure closely resemble programming languages like JavaScript and Python, making it accessible to developers familiar with those paradigms.

ThingsDB shines as the connecting force within your application, integrating various components seamlessly. Unlike managing separate relational databases and message brokers, ThingsDB offers a unified solution. It provides relational data storage for structured information, a built-in pub/sub system for real-time event handling, and scheduled task automation. This simplifies your architecture and eliminates the need for multiple tools.

ThingsDB excels at scaling to ensure high availability, keeping your application running smoothly even under load. However, for specialized data like large logs or time-series data, dedicated databases might be more suitable. In such cases, ThingsDB's strength lies in its modular connectivity. You can connect to various external databases using pre-built or custom modules, giving you fine-grained control over data storage and retrieval, ensuring the right tool for the right job.

> As a compelling example of its versatility, ThingsDB is employed by InfraSonar (https://infrasonar.com), a comprehensive monitoring solution designed to handle complex and expansive infrastructures. In this context, ThingsDB functions as a robust back-end solution, where critical business logic resides alongside relational data. Additionally, it serves as a pivotal communication hub, orchestrating interactions between microservices and handling the timely dissemination of notifications and messages.

In contrast to languages like C/C++, Go, and Rust, which require compilation before execution, ThingsDB utilizes an interpreter much like, for example, Python. However, ThingsDB distinguishes itself from other interpreted languages by maintaining state persistence, ensuring uninterrupted data access and manipulation even after script termination.

To better grasp this concept, consider what transpires when Python is launched.

```
$ python
…
>>> value = 'Remember me'
```

Upon assigning the string `"Remember me"` to the variable `value`, Python retains this value as long as the interpreter remains open. To verify this, you can execute the following code snippet:

```
>>> print(value)
Remember me
```

However, if we terminate the interpreter and relaunch Python, our `value` variable will be permanently lost:

```
>>> exit()

$ python
…
>>> print(value)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name value is not defined
```

We encounter an error message indicating that the variable `value` is undefined. To retain this value, a programming language like Python must resort to external mechanisms such as writing the value to a file and reading from a file or utilizing a database connection.

This is where ThingsDB emerges as a solution. Similar to Python, ThingsDB is a programming language with a rich feature set encompassing task scheduling, type definitions, procedures, pub/sub system, and much more. Before delving into the technical details, let's examine how the aforementioned example would be implemented using ThingsDB.

The following code shows how to assign and store a variable:

```
(//stuff)> .value = 'Remember me';
"Remember me"
```

Notice the dot (`.`) in front of `.value`, this dot is important because it tells ThingsDB to create a property to our collection object. The property in this example is `value` and the actual value of the property is a string `"Remember me"`. It is important to understand that without the dot (`.`) in front of value, we would have created a variable, just like we did in Python.

We also finished our statement with a semicolon (`;`). Although ThingsDB can be forgiving when you forget the semicolon at the end (in fact, the code above

would yield the same result when omitting the semicolon) you should *always* end your statements with a semicolon to prevent unwanted behavior.

In the example we also used a single space before and after the equal sign (=). This white space is ignored by ThingsDB and could be left out, or multiple spaces or even tabs and newlines would not make any difference. The code below, for example, doesn't look very appealing, but is just as valid as the example above.

```
(//stuff)> .  value
    =
  'Remember me'
;
"Remember me"
```

> If you need to insert newlines within your command, you can use the CTRL+n keyboard shortcut.

Now that we have assigned the property value to the root of the collection, exit the prompt by pressing CTRL+d.

Stop ThingsDB *(Choose the method that matches your setup)*:

- **Without Docker:** Press CTRL+c in the terminal where ThingsDB is running.
- **With Docker Compose:** Navigate to the directory containing your `docker-compose.yml` file and run `docker compose stop`.

Now start ThingsDB again and also start the prompt again.

> You do not need the `--init` argument anymore although this argument will be ignored when ThingsDB is already initialized. If you really want to re-initialize ThingsDB then you could also add the `--force` argument. This would remove all ThingsDB data and start a clean node!

```
(//stuff)> .value;
"Remember me"
```

As you can see, ThingsDB still knows about the property `value`. You might also notice that however this example is very simple, the syntax is more comparable with programming languages than it is with something like SQL.

You might also wonder how the client is able to perform the query. The protocol for communication with ThingsDB is done with MessagePack. ThingsDB follows the same rules as MessagePack. Strings are assumed to

be UTF8. This, however, is not guaranteed by ThingsDB but depends on if the MessagePack protocol is correctly followed. Usually, when working with a trusted client, this is the case. If you are not sure about this, ThingsDB has a function `is_utf8()` to test if a string really is UTF8. It also can define this on a type but we shall explore types later. Besides MessagePack, ThingsDB has support for JSON but this is strictly used for the HTTP API.

In the code sample below we verify that `.value` contains a string with valid UTF-8 encoding:

```
(//stuff)> is_utf8(.value);
true
```

If we want to know which type `.value` was, we could use the `type()` function. This function always returns a string with type information.

```
(//stuff)> type(.value);
"str"
```

In ThingsDB it is allowed to use statements as arguments. The code below shows that the `is_utf8()` function returns with a boolean value:

```
(//stuff)> type(is_utf8(.value));
"bool"
```

Where is the property "value" stored?

1.1 Things and Why ThingsDB

As we mentioned earlier, the `value` has been assigned to the root object. We refer to these objects as *"things"*, which explains the name ThingsDB. Every collection has a root which is a regular thing. It is not possible to remove the root, unless you choose to remove the complete collection. Every stored thing also receives a unique ID from ThingsDB. To see which ID our collection has, you can use the `id()` method.

```
(//stuff)> .id();
1
```

The initial *stuff* collection most likely has ID 1, but it might be different on your computer depending on the version of ThingsDB used when the collection was created. We also start the method `id()` with a dot (`.`) in front of it. That is because `id()` is a method of a thing object.

1.2 Code Blocks

ThingsDB enables the placement of code within code blocks, which are enclosed by curly braces `{}` and must contain at least one statement. It is crucial to note that code blocks themselves are interpreted as statements, and, consequently, must be terminated with a semicolon.

```
(//stuff)>
{
   .value = 'Remember me';
};  // this semicolon ends the "block" statement
"Remember me"
```

Code blocks enable the grouping of statements, which can be particularly beneficial when employed in conjunction with conditional statements (`if`-statements), iteration constructs (`for`-loops), or closures, all of which are detailed in subsequent chapters.

1.3 Variables and Properties

Just as we did in Python, we can also assign a value to a variable.

```
(//stuff)> value = "I'm a variable";
"I'm a variable"
```

Without the dot, this is just a variable. Variables only exist within a single query. Therefore we cannot ask for `value` in the next query. If we did, ThingsDB would return with a LookupError.

```
(//stuff)> value;
LookupError: variable `value` is undefined
```

In this example we used value as the name of our variable. Note that this can be named as you wish, for example, `x`, `y`, `Foo`, `myValue` and `my_value` are all good examples of variable names. Variables, like almost everything in ThingsDB, are case sensitive. This means that `VALUE` is not the same as `value` and they both can exist as different variables. A variable must start with a charter of a-z (or capital A-Z) or an underscore (_) and may be followed by the same or a number. The variable is not allowed to start with a number.

Here are some examples:

```
dog = nil;          // valid
Dog = nil;          // valid
_dog = nil;         // valid
cat-or-dog = nil;   // invalid – no hyphen (-) allowed in a variable name
catOrDog = nil;     // valid
cat_or_dog = nil;   // valid
2dogs = nil;        // invalid – a variable name must not start with a number
```

The code above also includes comments, which are denoted by the `//` prefix. Everything behind the `//` on a line will be disregarded by ThingsDB. An alternative comment syntax using `/*` and `*/` allows for comments spanning multiple lines. However, in this book, we'll primarily utilize the `//` syntax, as it is less prone to errors and avoids the risk of forgetting to close the comment section.

Properties, on the other hand, have different naming rules. Almost anything is possible except for a few reserved single character names. For example, we can't use `#` as this is a reserved property name and is being used by ThingsDB to expose the ID. Keep in mind that it is possible to create a property which starts with a reserved character. For example, the property `"# 007"` is valid and can be used but if we wanted to assign this property, we can't just write it like before as this syntax is invalid.

```
(//stuff)> .# 007 = "James Bond";
SyntaxError: error at line 1, position 1, unexpected character `#`
```

Instead, we can use the method `set(key, value)` which is available on every object of type thing.

```
(//stuff)> .set("# 007", "James Bond");
"James Bond"
```

If we need to read the property, we require the `get(key)` method.

```
(//stuff)> .get("# 007");
"James Bond"
```

The `get()` method is also useful for when you are not sure a property exists. By default, the return value is `nil` for when a property is not found.

```
(//stuff)> .get("I do not exist");
null
```

Did you notice I mentioned `nil`? Well, the output shows `null`, so let's clarify the situation. The reason for seeing `null` is that our prompt is written in Python, where "nil" is called "None". However, when the output is converted to JSON

30

format for display in the console, JSON uses `null` to represent the `None` type. Be mindful of this when translating between different programming languages.

If we wanted an alternative value we could provide the `get()` method with an alternative value.

```
(//stuff)> .get("I do not exist", "but I do");
"but I do"
```

1.4 Lazy Arguments Evaluation

Arguments to build-in methods and functions in ThingsDB are lazy evaluated. Lazy arguments evaluation means that the code which is used as the argument is not executed in case the argument is not being used. In case of the `get()` method this means that we are allowed to raise an error as the second argument which will only be executed in case the property is not found.

```
(//stuff)> .get("foo", raise(lookup_err()));
LookupError: requested resource not found
```

The error is raised because the property `"foo"` does not exist. If we would use the same code with a property which does exist, no exception would be raised.

```
(//stuff)> .get("# 007", raise(lookup_err()));
"James Bond"
```

We don't go into detail for the code to raise an error. For now it is enough to see lazy evaluation of method and function arguments at work.

1.5 Query Response

Up to this point, the code snippets presented for each query involved single statements, and the response reflected the output of that statement. However, query code in ThingsDB can encompass as many statements as needed. The query response always encapsulates the outcome of the final executed statement. Typically, this refers to the last statement in the provided code, unless an exception is triggered or the `return` keyword is explicitly utilized.

Consider these examples:

31

```
(//stuff)>  // press CTRL+n for a new line
1 / 0;
"Hello!";
ZeroDivisionError: division or modulo by zero
```

In this instance, the division by zero raises an exception, and the subsequent statement, `"Hello!"`, is not executed. The response, therefore, reflects the error message generated by the exception.

> 💡 Technically, an error response in ThingsDB is not merely a message but a distinct protocol type. This distinction enables clients to discern error responses and handle them appropriately, separate from valid data responses.

On the other hand, if the `return` keyword is explicitly used to terminate the query, the response will be the value specified by the return statement:

```
(//stuff)>
return 123;
"Hi!";
123
```

Here, the `return` statement halts the execution of the query and delivers the value `123` as the response, effectively skipping the subsequent statement, `"Hi!"`.

In summary, the response of a ThingsDB query mirrors the outcome of the final executed statement, unless an exception arises or the `return` keyword is explicitly employed. This flexibility allows for more complex and versatile query operations.

1.6 Scopes

Before we dive deeper into the ThingsDB language, we must first explain more about scopes. The example above uses the "stuff" collection, which is automatically created on a new ThingsDB initialization. As mentioned before, a collection is the place where data is stored. If you like to compare with SQL then you can think of a collection like a database. In SQL, you may have multiple databases running on a single SQL instance and in ThingsDB you can have multiple collections.

ThingsDB creates a unique scope for each collection. The full scope name for the "stuff" collection is `/collection/stuff`. You can shorten the prefix "collection" or even omit it altogether, so `/c/stuff` and `//stuff` are considered equivalent scopes. Additionally, you can use the "@" symbol to represent a

scope. This syntax replaces the first "/" with "@" and the second with a semicolon, resulting in `@collection:stuff` or `@:stuff` for short.

Management of collections happens within the `/thingsdb` scope. ThingsDB allows you to abbreviate this scope name as much as you like, so you might commonly see the shortened version `/t` being used. Similarly to the collection scope example, you can also use the "@" symbol. So, `/thingsdb`, `@thingsdb`, `/t`, and `@t` all refer to the same scope.

> In the early days of ThingsDB, only full scope names using the `@` syntax were permitted. The `@` symbol was chosen because it is an unused character in the ThingsDB language, leaving open the possibility of implementing it as a native scope definition within the language. With the introduction of the HTTP API, however, the slash (`/`) syntax became more prevalent, as it aligns more closely with standard URL conventions. Recognizing the inconvenience of typing out lengthy scope names repeatedly, the developers decided to allow abbreviated scope names to enhance the user experience.

For switching within the prompt, we must use the @ syntax since otherwise the prompt does not understand that we want to switch from scope. This is because the "@" character is not part of the ThingsDB language and thus can be recognized by the prompt as a different action.

> In addition to collection scopes and the **/thingsdb** scope, ThingsDB provides a third type of scope specifically designed for retrieving node information. These node scopes can also be employed to execute certain node-specific tasks, such as creating backups or dynamically adjusting the logging level for a node.

Let's switch the prompt to the `@thingsdb` scope:

```
(//stuff)> @t
(@t)>
```

We see that we switched to another scope. The `@thingsdb` scope is not able to store regular data as it contains no root object (thing) to attach properties to.

```
(@t)> .cat = 'Leo';
LookupError: the `root` of the `@thingsdb` scope is inaccessible; you might
want to query a `@collection` scope?
```

Instead, we use the `@thingsdb` scope to manage users, access rights, collections, modules, nodes and more.

For now we do not go into details but we create a new empty collection using the `new_collection()` function which is available in the `@thingsdb` scope.

```
(@t)> new_collection('chapter2');
"chapter2"
```

The return value is the name for the newly created collection. While internally ThingsDB assigns a unique ID to the collection for communication purposes, it is the name that you will use for interactions with the collection. In the next chapter we shall use the collection we have just created.

1.7 Quiz - Challenge Your Understanding

1. How is ThingsDB different from other databases and programming languages?

2. What is the purpose of the `--init` argument when starting a ThingsDB node?

3. The provided code contains a mistake. Can you identify and correct it?

    ```
    {a = 1;} b = 2;
    ```

4. Suppose we have the following code:

    ```
    a = 1; .b = 2;
    ```

    Which sentence is correct?
    a. Both `a` and `b` are variable
    b. Both `a` and `b` are properties of the collection root
    c. `a` is a property of the collection root and `b` is a variable
    d. `a` is a variable and `b` is a property of the collection root

5. Which of the following are valid variable names?
    a. `4you`
    b. `_Color`
    c. `my-hobby`
    d. `P`
    e. `Mode_1`

6. What is the value of property `x` after executing the following code?

    ```
    .x = 1; .get("x", .x = 2);
    ```

7. What comprises the response generated by a query?

8. Is it possible to create a property `user` with the value `"Alice"` while using the `@thingsdb` scope?

9. Which scope can be used to execute the following code? *(multiple answers are possible)*

    ```
    new_collection("book");
    ```

    a. `/t`
    b. `@:t`
    c. `//thingsdb`
    d. `@things`
    e. `/node`

1.7.1 Quiz - Answers

1. ThingsDB is different in that it uses a distributed interpreter which preserves its state making it a programming language with database capabilities.

2. The `--init` argument is essential when initiating ThingsDB for the first time. It establishes the data path and creates a foundational collection. Subsequent invocations of ThingsDB with the `--init` argument will be disregarded unless accompanied by the `--force` argument to override existing configurations.

3. While the code block is not strictly necessary, it is missing a crucial semicolon to terminate the code block statement. As a result, running this code would generate an error message:
   `..unexpected character `b`, expecting: ; or end_of_statement`. To rectify the issue, either add a semicolon after the code block or remove the code block entirely.

4. Answer "d" is correct. In this code snippet, `a` is a variable, and `.b` is a property of the collection root. The equivalent code would be `root().b;`, which explicitly references the root collection and then accesses its `b` property.

5. The variable names "b", "d" and "e" are valid. A variable name cannot start with a number making `4you` invalid. A variable name may only consist of the letters a-z, A-Z, underscores (_) and digits 0-9 *(except for the first character which is not allowed to be a digit)*. Therefore, `my-hobby` is invalid as it contains the "-" character.

6. The value of property `x` is 1. The `get()` method effectively retrieves the value of property `x`, which is 1. It optionally takes a second argument to provide a fallback value in case the property does not exist. In this instance, since the property `x` is found, the second argument is never evaluated, demonstrating the concept of *"lazy evaluation"* in action.

7. The response of a query encapsulates the outcome of the final statement executed within the query. Typically, this refers to the last statement within the provided code, unless an exception arises or the return keyword is explicitly employed.

8. No. The `@thingsdb` scope is specifically designed for managing your ThingsDB cluster, not for storing data. It allows you to handle administrative tasks such as user access control, collection creation, node management, and more. While you can create procedures within the `@thingsdb` scope, it does not provide a root thing to store properties. Therefore, you must use a collection scope to store data.

9. Both "a" and "d" are correct. New collections can be created in the `@thingsdb` scope and both `/t` and `@thing` are valid abbreviations. Both "b" and "c" are collection scopes and "e" is an example of a node scope.

Chapter 2 - Integers, Floating Points, Booleans, Strings and Nil

For this chapter we switch to the newly created collection *(see the previous chapter on how to create the collection)*.

```
(@t)> @ //chapter2
(//chapter2)>
```

> 💡 The @ command tells the prompt you want to switch to another scope. In this example we used the forward-slash notation to switch to the "chapter2" collection.

2.1 Integers

In ThingsDB integers, strings, boolean and floating point values are all immutable. That is, you cannot change an immutable value once it is created. An integer is a number without a fractional component. Examples are 1, 2, 415 but also 0 and negative integers like -15. As we explained, integer values are immutable which means that we cannot change the value once assigned.

For example, if we assign the integer to the variable x, we cannot change the value.

```
(//chapter2)> x = 42;  // x will be immutable
42
```

We cannot change the value as x is immutable, but we can assign a *new* value to x.

```
(//chapter2)> // press CTRL+n for a new line
x = 2;
x = x + 1;  // overwrite x with a new integer value
3
```

The return value is indeed the last assignment.

Aside from the basic assignment operator (=), ThingsDB supports other assignment operations that allow you to modify values in more complex ways. For instance, you can use the addition assignment operator (+=) to add a value to another and assign the result to a variable. For example:

```
(//chapter2)> // press CTRL+n for a new line
x = 2;
x += 1;  // read as: x = x + 1
3
```

Table 2.1 - Assignment operators

Operator	Description
=	Assignment operator
*=	Multiplication assignment
/=	Float division assignment
%=	Modulo assignment
+=	Addition assignment
-=	Subtraction assignment
&=	Bitwise AND assignment
^=	Bitwise XOR assignment
|=	Bitwise OR assignment

The range of integers is limited to a minimum and maximum value. You can ask this minimum and maximum value with the keywords INT_MIN and INT_MAX.

```
(//chapter2)> INT_MIN;
-9223372036854775808
(//chapter2)> INT_MAX;
9223372036854775807
```

When you try to create an integer value outside this range, an Overflow error is raised as ThingsDB cannot store this value.

```
(//chapter2)> INT_MAX + 1;  // this will error
OverflowError: integer overflow
```

In addition to the familiar base-10 notation, integers can also be represented using hexadecimal (hex), octal (oct), and binary notation. Each of these notations has its own unique set of symbols and prefixes that distinguish it from the others.

Hexadecimal notation (hex) uses the digits 0-9 and the letters A-F to represent values from 0 to 15. Hex numbers are prefixed with the identifier "0x" to indicate their hexadecimal nature. For instance, the number `0xFF` represents the decimal value `255`.

Octal notation (oct) employs the digits 0-7 to represent values from 0 to 7. Octal numbers are prefixed with the identifier "0o" to distinguish them. For example, the number `0o77` stands for the decimal value `63`.

Binary notation (bin) utilizes only the digits 0 and 1 to represent values from 0 to 1. Binary numbers are prefixed with the identifier "0b". For example, the number `0b1010` equals the decimal value `10`.

Here are a few examples:

```
0xff;     // hex notation (255)
0x1F;     // hex notation (31) - capital letters are allowed
0X10;     // !! error !! - the "x" must be lowercase
123;      // decimal notation
0o77;     // octal notation (63) - the "o" must be lowercase
0b1010;   // binary notation (10) - the "b" must be lowercase
```

Assignments work from right to left. The example below first assigns `30` to `apples`, then `apples` is assigned to `grapes` and finally `grapes` is assigned to `lemons`.

```
(//chapter2)> lemons = grapes = apples = 30;
30
```

2.2 Floating Points

Floating-point numbers are used to represent numbers with decimal parts, such as `0.5` and `3.14159`. ThingsDB supports two notations for representing floating-point numbers: standard decimal point notation and scientific notation (E notation).

Standard decimal point notation is commonly used for representing everyday numbers, such as `1.0`, `56.0`, and `0.5`. The E notation is more compact and is typically used for representing very large or very small numbers. For example, the number `1.5e+3`, written in E notation, represents the same value as `1.5 * 1000`. Similarly, `1.2e-4`, written in E notation, represents the same value as `1.2 * 0.0001`.

ThingsDB adheres to strict rules for writing E notation. The "e" must always be lowercase, and the sign (+ or -) is required to indicate whether to multiply by 10 to the positive or negative power.

Due to the limited precision of 64-bit floating-point numbers in ThingsDB, some decimal values may not be represented accurately. For instance, the result of dividing `10.0` by `3.0` is displayed as `3.3333333333333335`, even though the actual decimal representation extends infinitely. This is because the 64-bit format can only store a finite number of decimal digits.

```
(//chapter2)> 10.0 / 3.0;  // the single / means division
3.3333333333333335
```

ThingsDB follows the standard floating-point behavior when performing arithmetic operations. If either operand is a float, the result will also be a float. However, if both operands are integers, the result will be an integer and will be rounded towards the nearest integer towards zero. For example, `10 / 3` will be `3`, and `10 / -3` will be `-3`. This is because integer division in ThingsDB truncates the fractional part, discarding any decimal places.

> The decision of whether to use integer or float values depends on the specific context and the desired precision. Integers have a slight advantage in terms of storage size, as they can be compressed using the MessagePack protocol to occupy only 1 byte up to max 9 bytes whereas floating point values always are serialized using 9 bytes.

Two special floating-point values exist in ThingsDB: `inf` (infinity) and `nan` (NaN, not a number). Infinity represents an infinitely large number, while NaN represents a value that cannot be represented accurately. In other programming languages, `nan` may be the result of operations like `0/0`, but in ThingsDB, such operations will throw a division by zero error.

In ThingsDB, you will encounter `nan` only when a value is explicitly set to `nan` or when incoming arguments contain `nan`. One important distinction to remember is that comparing `nan` values to other values will always yield false results. Instead, use the `is_nan()` function to explicitly check whether a value is `nan`.

```
(//chapter2)> nan == nan;  // this is false!!
false
(//chapter2)> is_nan(nan);  // use is_nan() to test for NaN
true
```

2.3 Numeric Tools

In addition to the arithmetic operators, ThingsDB provides a set of handy mathematical functions for performing calculations on numbers. These functions typically accept either a float or integer as input and return the calculated result. As an example, the `sqrt()` function calculates the square root of a given number. Here are a few examples:

41

```
(//chapter2)> sqrt(9);   // Square root of 9
3.0
(//chapter2)> pow(5, 3);   // 5 raised to the power of 3
125.0
(//chapter2)> round(loge(5.12), 3);   // Combine round() and loge()
1.633
(//chapter2)> ceil(6.1034);   // Ceil of a given number
7
```

For a comprehensive list of all available mathematical functions, refer to the official ThingsDB documentation:
https://docs.thingsdb.io/v1/collection-api/math/

2.4 Boolean

The bool type, named after George Boole, is used to represent truth values. ThingsDB automatically converts bool values to integers when needed. This conversion follows the standard convention of representing `true` as `1` and `false` as `0`. For instance, adding two `true` values would result in `2`. Similarly, subtracting two `false` values would result in `0`. Consider this example:

```
(//chapter2)> true + true;
2
```

In this example, the expression `true + true` is evaluated to `2`, even though both operands are bool values.

In ThingsDB, you can also convert other data types to the bool type. For example, an integer value of `0` converts to `false`, and any other integer value converts to `true`. This applies to other data types as well, such as floats and strings. An empty string converts to `false`, while a string with any content, even whitespace, converts to `true`.

Here is an example demonstrating this:

```
(//chapter2)> bool(0.0);   // float 0.0 (false)
false
(//chapter2)> bool("");   // empty string (false)
false
(//chapter2)> bool(-4.13);   // non-zero float (true)
true
(//chapter2)> bool("Hello");   // non-empty string (true)
true
```

The conversion rules for other data types are described later in this book, but the important thing to remember is that *any* type can be converted to the bool type.

Occasionally, you might encounter code that employs two exclamation marks (`!!`) instead of the `bool()` function. The single exclamation mark serves as the "not" operator, which first converts the value following it to a boolean and then flips that value. When a second exclamation mark is used, the inverted value is inverted once more. As you can see, this ultimately yields the same result as using the `bool()` function.

```
(//chapter2)> !!"Cool";  // non-empty string (true)
true
```

This usage serves as a concise alternative to explicitly calling the `bool()` function.

2.5 Strings

Strings are used to store textual data and are typically encoded using UTF-8. Whether the encoding is indeed UTF-8 depends on whether the MessagePack protocol is followed during communication with ThingsDB. Strings are sequences of characters and can be indexed, meaning they have a defined length.

```
(//chapter2)> .b = "bike";
"bike"
(//chapter2)> .b.len();  // len() is a method of type "str"
4
```

As demonstrated, strings are objects with associated methods. The `len()` method is an example that can be used to determine the length of a string.

```
(//chapter2)> .b[0];  // the first byte is at index 0
"b"
```

Indexing for strings begins at position 0. Negative indexing is also supported.

```
(//chapter2)> .b[-1];  // last byte, read as: .b[.b.len()-1]
"e"
(//chapter2)> .b[.b.len() - 1];  // last byte, the hard way
"e"
```

Notice that any expression can be used for indexing strings. ThingsDB supports this level of flexibility.

Besides indexing, strings *(and other sequences)* support slicing, which allows you to extract a specific segment from the sequence. Slicing involves specifying a starting position *(inclusive)*, an ending position *(exclusive)*, and an optional step size.

```
(//chapter2)> .b[1:3];  // slice from position 1 up to,
                        // but not including, position 3
"ik"
```

Slicing operations create a new string, leaving the original string unchanged. This is because strings in ThingsDB are immutable, meaning they cannot be directly modified.

The general format for a slice is `[start:end:step]`, where `start` defaults to `0`, `end` defaults to the length of the string, and `step` defaults to `1`.

```
(//chapter2)> .b[1:];  // equivalent to .b[1:.b.len()]
"ike"
(//chapter2)> .b[:-2]; // equivalent to .b[0:.b.len()-2]
"bi"
(//chapter2)> .b[:];   // equivalent to .b[0:.b.len()]
"bike"
```

The step parameter enables you to select specific elements of the sequence by skipping over certain values. Negative step values can also be used to begin from the end of the sequence.

```
(//chapter2)> .b[::2];  // slice with an even step size,
                        // extracting only even-indexed bytes
"bk"
(//chapter2)> .b[::-1]; // slice in reverse order
"ekib"
```

In prior discussions, we have often used the term "bytes" instead of "characters" when referring to strings. This is because the length of a string in ThingsDB is measured in bytes, not characters. This can be important to keep in mind, especially when dealing with UTF-8 encoded strings, which can use multiple bytes to represent a single character.

```
(//chapter2)> .u = "Hi! 😁";  // UTF-8 string with smiley
"Hi! \ud83d\ude01"
(//chapter2)> .u.len();  // Length of the string in bytes
8
(//chapter2)> is_utf8(.u);  // Check if UTF-8 encoded
true
```

In this example, the string `"Hi! 😁"` has a length of 8 bytes, even though it contains only 5 characters. This is because the Unicode character for the smiley face, 😁, requires 4 bytes to represent. When working with strings in ThingsDB, it is crucial to remember that strings are internally treated as sequences of bytes, not characters.

To further illustrate the distinction between bytes and characters, consider this example:

```
(//chapter2)> .u[-3:];  // don't try this
(!! your client will likely crash or at least display an unpacking error)
```

This code attempts to index the string `.u` starting from the third-last character. However, since UTF-8 encoded characters can span multiple bytes, attempting to index directly by character position can lead to unexpected behavior, as demonstrated by the potential client error.

On the other hand, ThingsDB internally handles strings as sequences of bytes. Therefore, while this specific indexing operation might cause issues on the client-side, ThingsDB itself can still process the string correctly. This is evident from the following command:

```
(//chapter2)> is_utf8(.u[-3:]);  // No problem for ThingsDB
false
```

This command confirms that ThingsDB can still determine that the extracted substring is not valid UTF-8 even though it is accessed using an invalid character index. This demonstrates that ThingsDB handles strings consistently as sequences of bytes, regardless of the specific indexing operations performed.

2.5.1 String Methods

We have already explored the `len()` method for determining the length of a string. However, ThingsDB offers a wealth of additional methods for string manipulation like, for example, the `upper()` method.

```
(//chapter2)> .b.upper();  // Produces a new string in uppercase
"BIKE"
```

The `upper()` method generates a new string with all characters converted to uppercase and exemplifies the various methods available for the string type. Other methods, such as `starts_with()`, accept arguments and can be used to determine if a given string begins with another string.

```
(//chapter2)> "this is a test".starts_with("this is");
true
(//chapter2)> "another test".starts_with("Ano");
false
```

To explore a comprehensive list of all the methods available for the string type, including `upper()` and `starts_with()`, refer to the official ThingsDB

documentation here: https://docs.thingsdb.io/v1/data-types/str/

2.5.2 Escaping and Multi-line Strings

Until now, we have primarily employed double quotes to delimit strings. However, what if we want to include double quotes within the string itself? In this situation, we have two options. The simplest approach is to enclose the string in single quotes, which ThingsDB also supports:

```
(//chapter2)> '"This is an example", she said';
"\"This is an example\", she said"
```

> 💡 Notice that the response is rendered in JSON format. In JSON format, we observe that double quotes are escaped with a backslash (\).

Alternatively, we can escape double quotes by placing another double quote directly before them:

```
(//chapter2)> """This is as well"", he said";
"\"This is as well\", he said"
```

This escaping technique also applies to single quotes within single-quoted strings.

In ThingsDB, all strings are inherently multiline. Simply utilize newline characters to introduce line breaks within a string.

```
(//chapter2)> "All
strings in ThingsDB
are multiline!!";
"All\nstrings in ThingsDB\nare multiline!!"
```

As before, the response is presented as JSON format, and therefore the newline characters are displayed as \n.

2.5.3 Concatenation and t-strings

String concatenation can be accomplished using the + operator.

```
(//chapter2)> "My " + .b + " is black";
"My bike is black"
```

Take note that this does not work with other data types unless we explicitly convert them to strings:

46

```
(//chapter2)> // CTRL-n for a new line
.n = 3;  // Assign integer 3 to property "n"
"I've " + .n + " apples";
TypeError: `+` not supported between `str` and `int`
```

> 💡 Remember that even though the code above resulted in an exception, the property "n" is still preserved..

To rectify the example above, we can utilize the `str()` method to convert the integer to a string:

```
(//chapter2)> "I've " + str(.n) + " apples";
"I've 3 apples"
```

In practice, it is advisable to avoid string concatenation. Instead, opting for t-strings is a superior choice. A t-string starts and ends with a backtick (`) and enables us to embed variables or even entire expressions within curly braces.

```
(//chapter2)> `I've {.n} apples`;
"I've 3 apples"
```

This approach does not only enhance readability but also eliminates the risk of forgetting to convert to a string, as the conversion is done automatically when using the t-string syntax.

If you need to include curly braces or backticks within a t-string, you can follow the same escaping method used for single and double-quoted strings. To escape a curly brace, use two consecutive curly braces, and for a backtick, use another backtick.

```
(//chapter2)> `Sentence with ``Backticks`` and {{Curlies}}.`;
"Sentence with `Backticks` and {Curlies}."
```

This approach ensures that the braces and backticks are interpreted as part of the string and not as special characters.

2.6 Nil

We've encountered `nil` before as the return value of functions, methods or statements that do not produce any meaningful output. This concept extends to queries where we don't expect a substantial response.

Consider the following:

```
(//chapter2)> .quote = "'So many books, so little time.' -- Frank Zappa";
"'So many books, so little time.' -- Frank Zappa"
```

This query assigns a quote but also sends the quote back to the client, consuming unnecessary network bandwidth. Since we have already stored the quote, we do not need it echoed back in the response. In such cases, it is considered a best practice to terminate the code with the `nil` type to prevent this unnecessary response:

```
(//chapter2)> .quote = "'So many books, so little time.' -- Frank Zappa";
nil;
null
```

By appending `nil`, we avoid sending the quote back to the client, minimizing network overhead. Explicitly ending a query with `nil` also enhances the code's readability and intent. It clearly indicates that the query does not expect a response, making the code more self-explanatory and easier to follow.

2.6.1 Avoiding Ambiguity with Nil as a Placeholder

The `nil` data type often serves as a placeholder for empty values. While this can be convenient, it is crucial to avoid relying solely on the fact that `nil` evaluates to `false`. This is because other data types, such as an integer value of `0` or an empty string, also evaluate to `false`. To accurately identify `nil`, use the dedicated `is_nil()` function. This ensures that you are explicitly checking for the absence of a value rather than relying on potentially misleading behavior.

2.7 Errors

Before delving into the quiz for this chapter, let's explore the *error* data type. Throughout this chapter, we have encountered a few errors in our queries. While errors themselves are normal types in ThingsDB, raising an error interrupts the query execution and triggers the dedicated error handling protocol.

Consider the `zero_div_err`, which represents the error raised when attempting to divide by zero. We can create an instance of this error type and access its properties:

```
(//chapter2)> zero_div_err().msg();
"division or modulo by zero"
(//chapter2)> zero_div_err().code();
-58
```

In this example, we only returned the error information without raising it. However, we can explicitly raise an error using the `raise()` function:

```
(//chapter2)> raise(zero_div_err());
ZeroDivisionError: division or modulo by zero
```

The client, in this case the ThingsDB Prompt, recognizes the error and returns a corresponding exception: `ZeroDivisionError`.

The default message can be overwritten. Here is an example of the `zero_div_err` error with a customized message:

```
(//chapter2)> zero_div_err("division by zero is not allowed");
"division by zero is not allowed"
```

As you can see, the default message is now replaced with your custom message. Importantly, when an error is returned (without being raised), it is sent to the client as a string containing the message (equal to explicitly calling the `msg()` method).

ThingsDB also allows you to define custom error codes. These codes must fall within a specific range: -127 to -50.

> 💡 To avoid conflicts with existing ThingsDB error codes (like `zero_div_err` which ranges from -99 to -50), it's recommended to keep your custom error codes within the range of -127 to -100. For a comprehensive list of error codes, refer to the official documentation: https://docs.thingsdb.io/v1/errors/

Here is an example of raising a custom error with a code:

```
(//chapter2)> raise(err(-100, "My Custom Error"));
CustomError: My Custom Error
```

It is worth noting that `-100` is the default code for custom errors. You can achieve the same result by raising the message directly:

```
(//chapter2)> raise("My Custom Error");
CustomError: My Custom Error
```

2.7.1 Capture Errors

Unlike traditional try-catch blocks found in many languages, ThingsDB utilizes a `try()` function for error handling. This function requires at least one argument, which is the code you want to execute. If an error arises within the

code, `try()` returns with the error, preventing disruption of the entire query execution.

Let's see this in action:

```
(//chapter2)> a = 7.0; b = 0.0;
x = try(a / b);  // try dividing a with b
`{a} / {b} = {is_err(x) ? 'NaN' : x}`;
"7 / 0 = NaN"
```

Dividing by zero triggers an error, which is captured by the `try()` function. Use the `is_err()` function to check if the returned value from `try()` contains an error.

While `try()` captures all errors by default, you can refine its behavior to handle only specific errors. To achieve this, provide the desired error types as additional arguments to the `try()` function.

```
(//chapter2)> try(1 / 0, zero_div_err()); "OK";
"OK"
```

This code only captures the `zero_div_err error`, any other errors would still be raised.

This chapter covered some fundamental types of ThingsDB, including the concept of error handling. Test your understanding with the quiz and join us in the next chapter where we delve into lists and tuples.

2.8 Quiz - Challenge Your Understanding

1. What data type will be the result of adding an integer to a float?

2. If you have a value of `0.0`, what will be the result of putting two exclamation marks in front of it?

3. Can you guess the result of evaluating the following expression?
   ```
   sqrt(25);
   ```

4. Is it possible to change the string `"alarm"` to `"ALARM"` by converting all characters to uppercase?

5. Choose the correct answer:
   a. Strings in ThingsDB are always encoded in UTF-8
   b. Strings in ThingsDB must be written on a single line
   c. Strings in ThingsDB are internally represented as sequences of bytes
   d. Strings in ThingsDB must at least contain one character

6. What string is created by the following code:
   ```
   "slicing"[2:5]
   ```

7. Can you guess which string is produced by the following code without executing it?
   ```
   "blue blue grass, blue blue sky".replace("blue", "green", 2);
   ```

   (Hint: consult the online documentation for the replace method)

8. When and why is it considered a good practice to end a query with `nil`?

9. What happens when executing the following statement in ThingsDB?
   ```
   try({2 / 0; .A = true;}); .B = true;
   ```
   a. Neither `.A` nor `.B` will be set because the division by zero raises an error
   b. Only `.A` will be set because the assignment happens within the `try()` block
   c. Only `.B` will be set because `try()` catches the error and execution continues
   d. Both `.A` and `.B` will be set to `true`

2.8.1 Quiz - Answers

1. The result will be a float. When one operand is a float, the other operand is internally promoted to a float and the result will also be a float.

2. The exclamation mark before the value `0.0` converts the value to a boolean (`false`) and then flips the value (`true`). The second exclamation mark inverts the result again, so the final result is `false`.

3. The result of evaluating the expression `sqrt(25);` is `5.0`. While you might initially think of the answer as simply `5`, remember that the `sqrt()` function always returns a float, even when the input is an integer.

4. No, strings in ThingsDB are immutable, meaning they cannot be changed after they are created. You can create a *new* string with uppercase characters by using the `upper()` method.

5. The correct answer is "c", strings are internally represented as sequences of bytes. A string should be encoded in UTF-8 but this is not guaranteed. Strings can be defined across multiple lines and an empty string evaluates to false when converted to bool, but is still a valid string.

6. The string `"ici"`. Indexing starts at zero, so index 2 corresponds to the third character. The slicing operation results in a substring consisting of the characters from index 2 *(inclusive)* to index 5 *(exclusive)*.

7. The resulting string is `"green green grass, blue blue sky"`. The `replace()` method can be employed to replace a specified sequence of characters within a string with another string. Optionally, you can specify the number of instances to be replaced. Had the optional argument not been set to `2`, the result would have been
   `"green green grass, green green sky"`.

   (The replace() method offers more capabilities than described here. Consult the documentation for a comprehensive overview of its power: https://docs.thingsdb.io/v1/data-types/str/replace/).

8. When you execute a query that does not produce a meaningful response. It prevents unnecessary transmission of data back to the client and improves clarity and readability as it clearly indicates that the query does not expect a response.

9. The answer is "c". Only `.B` is set to `true`. The division by zero in `try()` halts its execution, preventing the assignment to `.A`. Execution continues outside `try()`, allowing the assignment to `.B`.

Chapter 3 - Lists and Tuples

Before we proceed, let's create a new collection named "chapter3" and switch to its scope. If you need a refresher on creating new collections, refer back to section 1.6 Scopes.

This chapter delves into the realm of lists and tuples, exploring their nature as array-like data structures with a crucial distinction: lists are mutable, allowing their contents to be modified, while tuples are immutable, preserving their values once created. Additionally, we'll uncover the concept of closures, unnamed functions that capture their surrounding scope, and examine their role when used as arguments.

3.1 Lists

Lists are positionally ordered collections of arbitrarily typed objects, and they have no fixed size. They are also mutable. Unlike strings, lists can be modified in-place by assignment to offsets and by a variety of list method calls.

```
(//chapter3)> .L = [true, "Eggs", 0.1];
[
    true,
    "Eggs",
    0.1
]
```

Similar to strings, you can get the length of a list and use indexing and slicing:

```
(//chapter3)> .L.len();   // Length of the list
3
(//chapter3)> .L[1];      // Access the second item at index 1
"Eggs"
(//chapter3)> .L[-2:];    // Get the last 2 items
[
    "Eggs",
    0.1
]
```

Since lists are mutable, you can modify their contents directly:

```
(//chapter3)> .L[0] = false;           // Set the first item to false
false
(//chapter3)> .L[1:1] = ["Ham", "&"]; // Insert "Ham" and "&" at index 1
null
(//chapter3)> .L;
[
    false,
    "Ham",
    "&",
    "Eggs",
    0.1
]
```

More common is to use one of the list methods to update the list.

```
(//chapter3)> .L.shift(); // Remove the first item
false
(//chapter3)> .L.pop(); // Remove the last item
0.1
(//chapter3)> .L.unshift("The", "best"); // Insert "The" and "Best" at the
                                         // beginning
5
(//chapter3)> .L.push("recipes"); // Append "recipes" to the end
6
(//chapter3)> .L.extend(["big", "world"]); // Append another list
8
(//chapter3)> .L.splice(6, 1, "in", "the"); // Remove 1 item at index 6 and
                                            // replace it with "in" & "the"
[
    "big"
]
```

The return value of each method varies depending on its functionality. In general, update methods return the removed values or the new list length if they only add items. Among the methods discussed, `splice()` stands out for its versatility. It can remove multiple items, insert new ones, and even return the removed items in a separate list.

If you have followed the examples above in the correct order, your list should now contain only strings. You can concatenate these strings into a single string using the `join()` method:

```
(//chapter3)> .L.join(" ");  // Join using a space
"The best Ham & Eggs recipes in the world"
```

3.1.1 Bounds Checking

While lists in ThingsDB do not have a fixed size, accessing an index beyond the list's bounds will still trigger a LookupError.

```
(//chapter3)> .L[99];
LookupError: index out of range
(//chapter3)> .L[99] = 1;
LookupError: index out of range
```

Instead of silently expanding the list, ThingsDB raises a `lookup_err`. To extend a list, utilize methods like push or slice assignments, which were previously demonstrated.

3.1.2 Reference and Maintained Lists

While ThingsDB adheres to conventional list behavior in most cases, it employs a unique approach when lists are embedded within a thing.

When lists are assigned to variables, ThingsDB maintains a single reference to the underlying data structure. This means that modifying the list through one reference will also affect any other references to the same list.

```
(//chapter3)>
A = ["Alice", "Bob"];  // Assign a list to variable "A"
B = A;                 // Assign "A" to "B"
B.push("Charlie");     // Append "Charlie" to the list
A;                     // Return with the list using "A"
[
    "Alice",
    "Bob",
    "Charlie"
]
```

As we can observe, assigning A to B establishes a reference to the same underlying list data structure. This is evident from the fact that appending "Charlie" to B also modifies the list accessed through A. Both variables A and B effectively point to the same list entity.

However, when a list is assigned to a property, ThingsDB creates a distinct copy of the list. We refer to this type of list as a *maintained list* because it is stored and managed within a ThingsDB collection.

```
(//chapter3)>
a = ["Alice", "Bob"];  // Assign a list to variable "a"
.b = a;                // Assign variable "a" to property ".b"
.b.push("Charlie");    // Append "Charlie" to the list on "b"
a;                     // Return the list assigned to variable "a"
[
    "Alice",
    "Bob"
]
```

In summary, ThingsDB creates references when assigning lists to variables and copies them when lists are assigned to properties of a thing. We refer to the latter lists as *maintained lists* to differentiate them from the referenced lists used in variables.

3.2 Nesting and tuples

ThingsDB allows you to store and retrieve multidimensional data using arrays. However, when you nest arrays within an array, the nested arrays are converted to tuples. Tuples are immutable, meaning they cannot be changed after they are created. This is because ThingsDB cannot track and synchronize changes to nested lists.

```
(//chapter3)> .m = [
    [1, 2, 3],
    [4, 5, 6],
];
[[1, 2, 3], [4, 5, 6]]
```

This code creates a multidimensional array `.m` with two nested arrays. You can verify that `.m` is a list using the `type()` function:

```
(//chapter3)> type(.m);
"list"
```

You can also extend the list by appending another array to it:

```
(//chapter3)> .m.push([7, 8, 9]);
3
```

However, when you append a nested list to another list, the nested list is converted to a tuple. This means that you cannot modify the nested list after appending it to the outer list.

For example, the following code attempts to append an element to the last nested array:

```
(//chapter3)> .m.last().push(10);
LookupError: type `tuple` has no function `push`
```

Similarly, you cannot modify the elements of a nested tuple using index assignments:

```
(//chapter3)> .m.last()[0] = 10;
TypeError: type `tuple` does not support index assignments
```

Tuples have a subset of the methods available to lists. Only methods that do not manipulate the data structure are available for both tuples and lists.

If you assign a tuple to a variable, the variable holds a reference to the tuple. However, if you assign a tuple to a thing, it will be converted to a maintained list. This means that you can modify the elements of the new list after it has been assigned to a thing. For example, the following code assigns the first nested array from `.m` to a variable `a`:

```
(//chapter3)>
a = .m.first();  // Get the first item which is tuple [1, 2, 3]
type(a);         // Verify that this is still a tuple
"tuple"
```

However, if you assign the tuple to a property on a thing, for example to `.o`, it will be converted to a maintained list:

```
(//chapter3)> .o = .m.first();
[1, 2, 3]
```

The variable `.o` is now a list, and you can modify its elements:

```
(//chapter3)> .o[0] = 10;  // This works!!
10
```

Table 3.2 - Summary of when ThingsDB employs copying or reference assignment for a list

Assignment	Description
`a = b`	By reference (`b` is a list)
`a = .b`	By reference (`.b` is a maintained list)
`a = t`	By reference (`t` is a tuple)
`.a = b`	Copy to new maintained list (`b` is a list)
`.a = .b`	Copy to new maintained list (`.b` is a maintained list)
`.a = t`	Copy to new maintained list (`t` is a tuple)
`a.push([])`	Copy to new tuple (`a` is a list)

3.3 Looping Over a List or Tuple

Prior lessons have covered accessing a list item using its index. However, we have yet to explore iterating through all list elements. To illustrate this concept, let's create a list:

```
(//chapter3)> .numbers = [3, 8, 9, 0, 2];
[3, 8, 9, 0, 2]
```

Here is one approach to iterating through the list:

```
(//chapter3)> odd = [];
for (n in .numbers) {
  if (n % 2) {
    odd.push(n);  // If n is odd, append it to the list
  };
};    // Do not forget this semicolon!!
odd;  // Return the list
[3, 9]
```

> 💡 In this example, we utilize a trick to check the modulo 2 result, which returns `0` for even numbers and `1` for odd numbers. Since `0` converts to `false` and `1` to `true`, we can employ this to identify odd numbers.

The `for`-loop approach offers more flexibility. In addition to the value, they also provide the index of the value. A `for`-loop also supports two keywords: `continue`, which skips all code after `continue` and commences the next iteration of the loop, and `break`, which halts iteration.

```
(//chapter3)> even = [];
for (n, i in .numbers) {
  if (n % 2) {
    continue;  // Skip to the next iteration for odd numbers
  };
  even.push(`{n} at index {i}`);  // Append a t-string to the list
  if (i == 3) {
    break;    // If the index is 3, terminate the loop
  };
};
even;  // Return the list
[
    "8 at index 1",
    "0 at index 3"
]
```

While `for`-loops offer the most flexibility, for many use cases, methods like `filter()`, `map()`, `each()` and `reduce()` provide a more concise and elegant approach to iterating over and manipulating data structures. Let's examine the `filter()` method to replace the first `for`-loop:

```
(//chapter3)> .numbers.filter(|n| n % 2);
[3, 9]
```

The `filter()` method takes a closure as its first argument, which is essentially an unnamed function. The `filter()` method iterates over the list and executes the closure for each item. The arguments that the closure accepts are specified between the two pipe characters (`||`), and in the case of a list or tuple, the filter method provides both the value and index of the item to the callback function. You are not required to use both arguments, as demonstrated in our example.

As you can see, the `filter()` method with a closure provides a more concise and readable way to filter a list compared to a `for`-loop. It is a common pattern in functional programming, and it can significantly improve the readability and maintainability of your code.

The second `for`-loop, which utilized the `continue` and `break` keywords, seems more complex to replace by list methods. While the `map()` method can be helpful, it is not the ideal solution in this context. The `map()` method produces a new list of the same length as the original list, but each value in the list is generated by the callback function. In our example, we want to create a list of even numbers with an index smaller than 4. Combining the `filter()` and `map()` methods seems like a viable approach, but it introduces a new issue:

```
(//chapter3)>
.numbers
  .filter(|n, i| n % 2 == 0 && i < 4)
  .map(|n, i| `{n} at index {i}`);
[
    "8 at index 0",
    "0 at index 1"
]
```

As you can observe, the index values are not as expected. This is because the `filter()` method creates a new list, and the `map()` method operates on this new list instead of the original list. Consequently, the index values are no longer preserved.

To address this challenge, we can employ the `reduce()` method, which offers a functional approach to iterating over and manipulating data structures. The `reduce()` method takes a closure and an initial value as arguments. The closure is executed for each item in the list, and the initial value is the first argument in the callback function. For subsequent iterations, the argument passed to the callback function is the accumulated value from the previous iteration, and the result of the last iteration will be the result for the reduce method.

```
(//chapter3)> .numbers.reduce(|o, n, i| {
    if (n % 2 == 0 && i < 4) {
        o.push(`{n} at index {i}`);
    }; o;
}, []);
[
    "8 at index 1",
    "0 at index 3"
]
```

Admittedly, this solution is more complex than the `for`-loop. However, it demonstrates the power and versatility of functional programming techniques.

3.4 Specialized Methods

Most challenges related to lists can be addressed by combining looping methods with methods for modifying lists. However, ThingsDB provides convenient shortcuts for handling common tasks. Here are a few examples:

```
(//chapter3)> .numbers.sum();   // Calculates the sum of all values
                                // in the list
22
(//chapter3)> .numbers.sort(); // Sorts the list in ascending order
                               // and returns a new list
[0, 2, 3, 8, 9]
(//chapter3)> .numbers.is_unique(); // Checks whether all values in
                                    // the list are unique
true
(//chapter3)> .numbers.index_of(9); // Finds the index of the first
                                    // occurrence of the value 9 in
                                    // the list
2
```

For a complete list of available methods, refer to the official documentation: https://docs.thingsdb.io/v1/data-types/list/

As ThingsDB evolves, new methods are added. Therefore, when encountering a problem, it is advisable to first consult the documentation to see if there is an existing method that can assist you. If you have a specific requirement, you can submit a pull request with your proposed method, and one of the ThingsDB developers may consider implementing it for future releases.

3.5 Lists for Multi-Value Returns

Within ThingsDB, *"lists"* offer a powerful way to retrieve multiple values from a single query.

Imagine you want to fetch both a status code and a corresponding message from your query. Using lists, you can achieve this elegantly by simply returning a list containing both desired values.

```
(//chapter3)> status = 0; message = "success";
[status, message];  // List containing both status code and message
[
    0,
    "success"
]
```

By embracing lists for multi-value returns, you unlock a flexible and efficient approach to retrieving data in ThingsDB.

3.6 Quiz - Challenge Your Understanding

1. If we execute the following code:
   ```
   a = [1, 2, 3];
   .b = a;
   ```

   What will happen?
   a. Property `.b` contains a copy of `a`
   b. Both variable `a` and property `.b` reference the same list
   c. This code will fail because you cannot assign a list to a property

2. What will happen if we add a list `A` as a value into another list `B`?

3. Consider the following code:
   ```
   a = [["foo", "bar"]];
   b = a[0];
   c = a;
   ```

   Which answer is correct?
   a. `a` is a tuple, `b` is a reference to a list and `c` is a maintained list
   b. `a`, `b` and `c` are all lists
   c. `a` is a list, `b` is a reference to a tuple and `c` is a maintained list
   d. `a` and `c` are the same list and `b` is a reference to a tuple

4. Can you tell which technique ThingsDB supports for looping over a list or tuple?
   a. `each()` method
   b. `map()` method
   c. `reduce()` method
   d. `filter()` method
   e. `for`-loop
   f. all of the above

5. What is the final result of the following code snippet?
   ```
   range(2, 5).reduce(|t, n, i| t += i * n, 10);
   ```

   (Try to answer without executing the code. For reference, the range() documentation is available at:
   https://docs.thingsdb.io/v1/collection-api/range/)

6. Suppose we have the following input:
   ```
   input = [[1, 2], [3, 4]];
   ```

   Can you write code which returns with `[1, 2, 3, 4]` from that given input?

3.6.1 Quiz - Answers

1. Answer "a" is correct. When assigning a list to a property, a copy is created, resulting in a maintained list. Changes to the original list will not reflect in the maintained list.

2. A copy of list `A` will be created as a tuple. Since the nested tuple is immutable, you cannot modify its contents after creation.

3. Answer "d" is correct. The code does not contain a maintained list as only variables are used and nothing is assigned to a thing. Nested lists are always converted to tuples and are accessed by reference when assigned to a variable.

4. The correct answer is "f". ThingsDB supports all the listed techniques for iterating over lists and tuples. Each method serves a distinct purpose, and the `for`-loop proves particularly valuable when you need to utilize the `break` keyword to prematurely terminate the iteration.

5. The result is 21. The function call `range(2, 5)` produces the list `[2, 3, 4]` and the reduce method uses a closure which adds the index multiplied by the number to a total with a start value of 10. The result is `10 + 0*2 + 1*3 + 2*4 = 10 + 0 + 3 + 8 = 21`.

6. While you could use a `for`-loop or the `reduce()` method to achieve this, ThingsDB provides a dedicated method called `flat()` that simplifies this process:
   ```
   input.flat();
   ```

   (see https://docs.thingsdb.io/v1/data-types/list/flat/)

Chapter 4 - Things

Let's start this chapter with a new collection named "chapter4" and switch to its scope. If you are unsure of the process, revisit the end of Chapter 1.6 for a refresher.

Why is this programming language named ThingsDB?

In ThingsDB, data is organized and stored within things, which are essentially objects with properties. Every collection starts with an empty root "thing". New "things" can be created by using the `thing()` function without arguments, or, more commonly, you can use curly braces `{}`:

```
(//chapter4)> t = {};  // Create an empty thing
t.x = 4;  // Assign property "x" with value 4 to "t"
t.y = 2;  // Assign property "y" with value 2 to "t"
t;        // Return "t"
{
    "x": 4,
    "y": 2
}
```

This can be simplified by initializing the thing with the properties in curly braces:

```
(//chapter4)> t = {x: 4, y: 2,};
{
    "x": 4,
    "y": 2
}
```

This initialization syntax only works for properties that adhere to the ThingsDB naming convention, which is the same as the convention for variable names *(see 1.3 Variables and Properties)*. The properties are defined as key-value pairs separated by commas. You can omit the comma after the last property, since ThingsDB will ignore it if no other key-value pairs follow.

> 💡 The same principle applies when initializing lists. The last comma in a list initialization is optional if no further items follow. For instance, both `[nil]` and `[nil,]` are valid representations of the same list.

ThingsDB supports a shorthand notation to initialize a thing when using a variable and using the same variable name as the property name:

```
(//chapter4)>
foo = 6;
bar = 7;
t = {foo:, bar:,};
{
    "bar": 7,
    "foo": 6
}
```

This syntax is equivalent to writing `t = {foo: foo, bar: bar};` but it eliminates the need to repeat variable names.

4.1 Things for Descriptive Multi-Value Returns

While lists, as shown in Chapter 3.5, provide a simple way to return multiple values, things provide a more descriptive alternative. They allow assigning meaningful names to each value, enhancing clarity and reducing reliance on positional interpretation. This structure improves readability and understanding, but comes at a slight network bandwidth cost compared to lists.

Example:

```
(//chapter4)> status = 0; message = "success";
{
    status:,
    message:,
};  // Thing containing both status code and message
{
    "message": "success",
    "status": 0
}
```

> For optimal efficiency of read only queries, construct the thing containing multiple values either at the end of your query or after a `return` statement. Setting individual properties one by one might trigger unnecessary updates. While this has no effect on queries modifying the collection, it can lead to less efficient execution for queries which are intended to be read only.

4.2 Thing IDs

Once a thing is stored, meaning it is connected to the collection root, it is assigned a unique identifier or ID. Back in Chapter 1.1 we already saw that

our root thing had an ID and that we could ask for this ID using the `id()` method.

```
(//chapter4)> .id();  // ID for the collection root
1
(//chapter4)> {}.id();  // Unattached thing lacks an ID
null
(//chapter4)> (.t = {}).id();  // Assign a new thing to property "t",
                               // guaranteeing a unique ID.

2
```

A thing possesses no ID as long as it remains unattached to the collection. In the last example, we attached a new thing to property `t` within the root and immediately queried its ID. The response displayed a unique identifier.

Instead of explicitly requesting the ID, it is also accessible as a reserved property named "#" within the thing's response.

```
(//chapter4)> .t;
{
    "#": 2
}
```

However, it is important to note that "#" is not a genuine property of the thing, as evidenced by the following code snippets:

```
(//chapter4)> .t["#"];  // Key '#' does not exist
LookupError: thing `#2` has no property `#`
(//chapter4)> .t.len();  // The length is 0
0
(//chapter4)> bool(.t);  // Empty thing evaluates as false and .t is empty
false
```

As ThingDB implicitly assigns IDs to things, it is crucial to grasp when this occurs. While the earlier example might seem straightforward, it is essential to recognize its nuances.

The code below employs the `assert()` function, which triggers an exception if the provided expression evaluates to `false`. This, along with the `is_nil()` and `is_int()` functions, facilitates straightforward testing for ID existence.

```
(//chapter4)>
arr = [];                   // Create an empty list
a = {name: "Alice"};        // Create a thing with a name property
assert(is_nil(a.id()));     // Confirm that the thing has no ID
arr.push(a);                // Add the thing to the list
assert(is_nil(a.id()));     // Verify that the thing still lacks an ID
.arr = arr;                 // Assign the list to a collection root property
assert(is_int(a.id()));     // Verify that the thing now has an ID
a;
{
    "#": 3,
    "name": "Alice"
}
```

In this example, we observe that the ID is generated when the thing is linked to the collection. Initially, we assigned the thing to a variable, followed by adding the variable to the list. Since the list remained unattached to the collection, the thing retained its lack of an ID. It is only when we incorporated the list as a property of the collection root that it became associated with the collection, triggering ID assignment.

Another noteworthy aspect is that ThingsDB always interprets things by reference. This observation is evident in the fact that we kept checking the original variable "a".

Once an ID is assigned to a thing, it remains permanently associated with that thing and cannot be altered or removed.

ThingsDB supports multiple references to the same ID, allowing you to link distinct properties or collections to the same underlying thing. This is demonstrated in the code snippet:

```
(//chapter4)> .t.p = .arr.first();
{
    "#": 3,
    "name": "Alice"
}
```

As evident, property p of thing t points to the same thing as the first element in the list.

If you know the ID of a specific thing, you can directly retrieve it using the thing() function. When provided with an integer as the first argument, ThingsDB searches the collection for the thing with that ID. This becomes particularly useful in practical scenarios.

```
(//chapter4)> thing(3).name = "Bob";
"Bob"
(//chapter4)> thing(99).name = "Charlie";
LookupError: collection `chapter4` has no `thing` with ID 99
```

The first example successfully modifies the `name` property of the thing with ID 3 to `"Bob"`. The second query fails because the collection does not contain a thing with ID 99.

Let's review the current state of the collection by querying the `root()` thing:

```
(//chapter4)> root();
{
    "#": 1,
    "arr": [{"#": 3}],
    "t": {"#": 2}
}
```

This output might raise an eyebrow. While the `root()` with properties `arr` and `t` is fully displayed, the `p` property of the thing with ID 2 and the `name` property of the thing with ID 3 are missing.

This is because ThingsDB, by default, returns things only *one level* deep. In this instance, the `root()` represents the first level, and the things within list `arr` and the thing within `t` are considered the second level. This safety measure prevents accidentally returning the entire collection. While this might be acceptable in our example due to its small size, it could pose issues with larger collections.

4.3 Control Response with return Statement

To address the issue of returning only *one level* deep, we can utilize the `return` keyword, which was introduced in Chapter 1.5. This statement accepts a second argument that specifies the deep level:

```
(//chapter4)> return root(), 3;
{
    "#": 1,
    "arr": [{"#": 3, "name": "Bob"}],
    "t": {"#": 2, "p": {"#": 3, "name": "Bob"}}
}
```

With this modified query, we now see the `root()` and its properties at *level 1*, the thing with ID 3 within `arr` at *level 2*, the thing with ID 2 at *level 2*, and the thing with ID 3 again, this time residing on property `p` at *level 3*.

Pre Release
v7

> 💡 While it is generally recommended to keep the default deep level unchanged, it is possible to modify it using the `set_default_deep()` function. The current deep setting can be checked using the `deep()` function within a collection or through the `collections_info()` function in the `@thingsdb` scope.

The `return` statement accepts an optional third argument, which allows you to specify flags that modify the returned data. Currently, only one flag is supported: `NO_IDS`. When enabled, this flag prevents ThingsDB from including ID values in the response.

```
(//chapter4)> return root(), 3, NO_IDS;
{
    "arr": [{"name": "Bob"}],
    "t": {"p": {"name": "Bob"}}
}
```

While you might initially consider using the `return` statement frequently to control the deep level or incorporate the `NO_IDS` flag, ThingsDB offers a more elegant and streamlined approach to retrieving the desired data. This will be introduced in Chapter 10, where we delve into typed things and wrapping.

4.4 Looping Over a Thing

We can use a `for`-loop to iterate through the key value pairs of a thing.

```
(//chapter4)> .gps = {lat: 51.36, long: 5.23};
for (key, value in .gps) {
    log(`{key}: {value}`);
};
WARNING:root:ThingsDB: lat: 51.36 (2)
WARNING:root:ThingsDB: long: 5.23 (2)
null
```

The `for`-loop is similar when used on a thing compared to when iterating a list. Only the arguments are different. While iterating over a list accepts a value and index, a thing accepts a key and value.

> 💡 In this code snippet, we introduced a new `log()` function *(distinct from the **loge()** function that calculates the natural logarithm (base e) of a number)*. This `log()` function serves debugging purposes and operates as follows: Upon execution, it generates a warning log message to the console where the ThingsDB Node is running and emits a warning event to the client initiating the query. This warning event precedes the transmission of query results. The ThingsDB Prompt conveniently handles this event by displaying the warning message within the prompt. While real-world implementations may vary, warnings are typically logged for future reference.

Similar to lists, a thing also offers methods for iterating over its contents. The provided example demonstrates how to achieve the same result as the `for`-loop using the `each()` method:

```
(//chapter4)> .gps.each(|key, value| log(`{key}: {value}`));
WARNING:root:ThingsDB: lat: 51.36 (2)
WARNING:root:ThingsDB: long: 5.23 (2)
null
```

Additional looping methods include `filter()`, `map()`, and `vmap()`.

4.5 Value Restriction

While keys in ThingsDB are always restricted to strings, values can have any type. However, if a specific type is required for all values, a value restriction can be applied to enforce this constraint.

Consider our existing "gps" example:

```
(//chapter4)> .gps.restriction();
null
```

The `restriction()` method returns `nil`, indicating that no value restriction is currently applied to the "gps" thing.

Attempting to apply an invalid restriction to a non-empty thing will result in an error. For instance, trying to restrict "gps" to integer values will fail because it already contains float values:

```
(//chapter4)> .gps.restrict("int");
ValueError: at least one of the existing values does not match the desired
restriction
```

70

To successfully apply a value restriction, all existing values must adhere to the specified type. In this case, we can successfully apply a float restriction:

```
(//chapter4)> .gps.restrict("float");
{
    "#": 5,
    "lat": 51.36,
    "long": 5.23
}
```

Once a value restriction is applied, attempting to assign a value of an incompatible type will trigger a type error:

```
(//chapter4)> .gps.lat = nil;
TypeError: restriction mismatch
```

This error prevents invalid value assignments and ensures data integrity.

4.6 Self-References

Lists and other data types in ThingsDB do not allow for self-references due to the creation of copies (tuple conversion) when lists are nested. However, self-references are common within the structure of things.

Consider a simple example of creating a book and its author:

```
(//chapter4)> // Create a book and author
author = {name: "Katja Hoyer"};
.book = {
    title: "Beyond the wall",
    author:,
};
{
    "#": 6,
    "author": {"#": 7},
    "title": "Beyond the wall"
}
```

Since we only retrieve the first deep level of detail, we do not see the entire author thing, but we can see that both the book and author have unique IDs (6 and 7, respectively).

To retrieve the author of the book, we can simply access the author property of the book thing:

```
(//chapter4)> thing(6).author.name;  // ID 6 is the book
"Katja Hoyer"
```

To retrieve the books written by the author, we can add a `books` property to the author thing and populate it with the book:

```
(//chapter4)> thing(7).books = [.book];  // ID 7 is the author
nil;  // Return nil as we do not need the assignment as response
null
```

Now, we can retrieve data in both directions, but we have created a nested self-reference for both the book and the author.

You might wonder what will happen if we use a large deep value and ask for either the book or the author? ThingsDB intelligently handles the situation by only returning the IDs of self-referential objects. This prevents infinite loops and ensures data integrity:

```
(//chapter4)> return .book, 99;
{
    "#": 6,
    "author": {
        "#": 7,
        "books": [{"#": 6}],
        "name": "Katja Hoyer"
    },
    "title": "Beyond the wall"
}
```

Establishing two-way relationships between things is a common practice in ThingsDB, but managing these connections manually can be cumbersome and error-prone. To address this challenge, ThingsDB introduces a sophisticated solution that simplifies and streamlines relationship management.

In Chapter 9, we'll delve into the details of this solution and explore its capabilities in detail. But before we embark on this journey, we encourage you to assess your comprehension by taking the quiz provided. This will help solidify your understanding of things and prepare you for the next chapter, where we'll explore the powerful concept of *sets* in depth.

4.7 Quiz - Challenge Your Understanding

1. Looking at the following code, will the thing "t" receive an unique ID and if so, at which line? *(a-e)*

   a. `t = {};`
   b. `p = [t];`
   c. `.p = p;`
   d. `.t = t;`
   e. `// "t" still lacks an ID`

2. What will be the response for a query using the following code?

   ```
   return {x: 1, y: 2}, 0;
   ```

3. What function is available to verify the *deep* value for a collection using a collection scope?

4. How can you see the ID of a thing in a query response? For example, in the response for this code?

   ```
   .x = {};
   ```

5. Which of the following code examples will work without returning an error?

   a. `{4: 2}.restrict("int");`
   b. `{}.restrict("int").x = 123;`
   c. `{m: 'Hello!'}.restrict("str");`
   d. `{}.restrict("int").x = 1.0;`
   e. `{pi: 3}.restrict("float").pi = MATH_PI;`

6. Can a *list* in ThingsDB be a part of itself?

7. Can a *thing* in ThingsDB be a part of itself?

8. Given the existing thing:

   ```
   .point = {x: 5, y: 7};
   ```

   Can you create a query to return the following desired response:

   ```
   {"x": 5.0, "y": 7.0}
   ```

   The returned object should contain float values instead of integer values for the x and y properties.

4.7.1 Quiz - Answers

1. At line "c". The thing is a member of the list `p` which on this line is assigned to the collection root via property `p`.

2. The *deep* value has been set to 0, so the query will return an empty object: `{}`

3. The `deep()` method can be used to retrieve the current deep value for a collection in the collection scope.

4. The ID of a thing is returned as the `#` property in the thing object unless explicitly the `NO_IDS` flag is used.

5. The code samples "b" and "c" will execute without errors. Code snippet "a" will fail because keys must be of type string, and snippet "d" will fail because the thing is restricted to integers and the value `1.0` is a float. Snipped "e" fails because the initial value of `pi` is an integer, and attempting to apply the restriction to type float will result in an error.

6. No, a *list* cannot be a part of itself. Lists are copied when nested, so they cannot be referenced by themselves.

7. Yes, a *thing* can be a part of itself. Things are always accessed by reference, so they can be self-referential.

8. The easiest way to accomplish this is by using the `vmap()` method which returns a new thing with equal keys but values computed as a result of a given closure callback.
    ```
    .point.vmap(|v| float(v));
    ```

    (see https://docs.thingsdb.io/v1/data-types/thing/vmap)

Chapter 5 - Sets

Just like in previous chapters, we shall begin by creating a new collection and switching the prompt to that collection.

In this chapter, we'll explore the concept of sets in ThingsDB. Let's start by creating an empty set.

```
(//chapter5)> .mammals = set();
[]
```

Notice that despite creating a set, the response displays an empty list. This is because MessagePack, the underlying data serialization format for ThingsDB, does not have explicit support for sets. As a result, sets are always converted to lists in responses.

To add elements to a set, we can utilize the `add()` method. Sets can only accommodate things. Attempting to add other data types will trigger a type error.

```
(//chapter5)> .mammals.add("dog");
TypeError: cannot add type `str` to a set
```

Let's add some mammals to the set. We can add multiple things simultaneously by separating them with commas.

```
(//chapter5)> .mammals.add(
    {animal: "dog"},
    {animal: "cat"},
    {animal: "elephant"},
    {animal: "lion"},
);
4
```

The number of things successfully added to the set is returned. In this instance, we added four new elements.

Let's examine the set we've created:

```
(//chapter5)> .mammals;
[
    {"#": 2, "animal": "dog"},
    {"#": 5, "animal": "lion"},
    {"#": 4, "animal": "elephant"},
    {"#": 3, "animal": "cat"}
]
```

As you can see, all things have been assigned unique IDs, but the order of elements does not match our expectation. This is because sets are inherently unordered.

Due to this lack of order, retrieving an element by index is not possible.

```
(//chapter5)> .mammals[0];
TypeError: type `set` is not indexable
```

Each thing can only exist once within a set. Remember that ThingsDB compares things by reference, so even if two things have identical properties, they are considered distinct entities.

```
(//chapter5)> .mammals.add({animal: "dog"});  // Add another dog
1
```

Despite having the same properties (`animal: "dog"`), this dog is considered a different entity due to reference comparison. It is not the same "dog" previously added, so it gets included in the set.

```
(//chapter5)> animal = .mammals.one();  // Take one element from the set
.mammals.add(animal);  // Try to add that animal
0
```

The final query yielded `0`. This is because the animal already exists within the set, and calling the `add()` method did not introduce any new elements.

5.1 Set Operations

Before moving forward, let's create two more sets, one for birds and another for reptiles. *(We apologize to any fish or other animal species that weren't included; we'll focus on these three classes for now ;-)*

```
(//chapter5)> .birds = set(
    {animal: "parrot"},
    {animal: "flamingo"},
);
.reptiles = set(
    {animal: "turtle"},
    {animal: "snake"},
);
nil;  // Return nil as we do not need a response
null
```

Now that we have our sets ready, let's create a zoo and populate it with some animals.

```
(//chapter5)>
animals = ["lion", "elephant", "flamingo", "turtle", "snake"];
all = .mammals | .birds | .reptiles;
.zoo = all.filter(|x| animals.has(x.animal));
.zoo.len();
5
```

Let's break down what happened here:

1. We defined a list (`animals`) containing the animals we want in the zoo.
2. Using the union operator (`|`), we created a new set (`all`) that combines the `mammals`, `birds`, and `reptiles` sets.
3. We employed the `filter()` method to iterate over the `all` set and select only those animals that are also present in the `animals` list.
4. Finally, we used the `len()` method to count the number of animals in the `zoo` set.

Sets in ThingsDB provide a range of useful operations that enable efficient manipulation of data. These operations, summarized in Table 5.1, facilitate tasks such as combining, filtering, and finding specific subsets of things within sets.

Table 5.1 - Set operations

Operator	Operation	Description
`|` (union)	`A	B`	The set of things that are in either `A` or `B`.
`&` (intersection)	`A & B`	The set of things that are in both `A` and `B`.
`-` (difference)	`A - B`	The set of things that are in `A` but not `B`.
`^` (symmetric difference)	`A ^ B`	The set of things that are in either `A` or `B`, but not in both.

We have already employed the union operation to populate our zoo with a diverse set of animals. Now, let's delve into additional examples showcasing how set operations can be utilized within the context of our zoo.

5.1.1 Identifying Birds Not in the Zoo

The `-` operator allows us to find birds that are not part of the zoo:

```
(//chapter5)> .birds - .zoo;
[
    {"#": 8, "animal": "parrot"}
]
```

This command retrieves the `birds` set and subtracts the `zoo` set, resulting in a new set containing only the birds that do not reside in the zoo.

5.1.2 Selecting Warm-Blooded Animals in the Zoo

We can combine the `|` and `&` operators to find warm-blooded animals that are also in the zoo:

```
(//chapter5)> (.mammals | .birds) & .zoo;
[
    {"#": 7, "animal": "flamingo"},
    {"#": 5, "animal": "lion"},
    {"#": 4, "animal": "elephant"}
]
```

This expression combines the `mammals` and `birds` sets using `|`, then intersects the resulting set with the `zoo` set, effectively selecting only the warm-blooded animals that are in the zoo.

These examples demonstrate how set operations can effectively manage and analyze data within sets, providing a powerful tool for building efficient and data-driven applications.

5.2 Determining Set Membership and Supersets/Subsets

In addition to the set operations we've covered, ThingsDB sets provide several additional useful operators and methods. These enable us to determine whether a set contains all things of another set, verify whether a set is a subset or superset of another, and check whether a particular thing belongs to a set.

5.2.1 Verifying Set Membership

The `has()` method can be employed to determine whether a specific thing exists within a set. This method is particularly efficient for sets compared to lists, as it does not require iterating through the entire collection to perform the check.

```
(//chapter5)> flamingo = .birds.find(|b| b.animal == "flamingo");
.zoo.has(flamingo);
true
```

5.2.2 Checking Subsets and Supersets

To determine whether one set is a subset of another, we can use the `<=` operator. This operator checks if all elements of the first set are also present in the second set.

```
(//chapter5)> .reptiles <= .zoo;   // Determines if "reptiles" is a
                                   // subset of "zoo"
true
```

Conversely, to verify whether a set is a superset of another, we employ the `>=` operator. This operator checks if the second set contains all the elements of the first set.

```
(//chapter5)> .zoo >= .reptiles;   // Determines if "zoo" is a
                                   // superset of "reptiles"
true
```

To distinguish between proper subsets and supersets, which explicitly exclude the possibility of equality, the `<` and `>` operators are employed. These operators function similarly to their respective `<=` and `>=` counterparts, but they return `false` when the sets are equal.

Table 5.2 - Subset and Superset operations

Operator	Operation	Description
`<=`	is subset	Determines if all things of the first set are contained within the second set, ignoring equality.
`<`	is proper subset	Determines if all things of the first set are contained within the second set, and the two sets are not equal.
`>=`	is superset	Determines if all things of the second set are contained within the first set, ignoring equality.
`>`	is proper superset	Determines if all things of the second set are contained within the first set, and the two sets are not equal.

5.3 Copy or Reference

Assignments with sets follow a similar pattern to lists. When assigning a set to a variable, it is treated as a reference. However, if you assign a set to a

property of a thing, a copy of the set will be created.

> 💡 Nested sets are not supported in ThingsDB. Sets can only contain things. However, you can add a set to a list. In this case, a copy of the set will be made, and it will be converted into an immutable tuple.

Table 5.3 - Summary of when ThingsDB employs copying or reference assignment for a set

Assignment	Description
`a = b`	By reference (`b` is a set)
`a = .b`	By reference (`.b` is a maintained set)
`.a = b`	Copy to new maintained set (`b` is a set)
`.a = .b`	Copy to new maintained set (`.b` is a maintained set)
`a.push(set())`	Copy to new tuple (`a` is a list)

5.4 Quiz - Challenge Your Understanding

1. What data types can be stored in a ThingsDB set?

2. Determine the resulting *set* after performing the following operation:
   `set(a, b) - set(b, c)`

3. Which of the following expressions evaluates to `true`?
   a. `set(a, c) <= set(a, b, d)`
   b. `set(a, c) >= set(a, b, d)`
   c. `set(a, c) > set(a, c)`
   d. `set(a, c) < set(a, b, c)`

4. How does a client receive a *set* when it is returned in a response from ThingsDB?

5. Can you explain what will happen when you append a *set* as an element to a *list*?

5.4.1 Quiz - Answers

1. Only unique "things" are allowed in a set.

2. The resulting set after performing the operation is `set(a)`. The `-` operator in ThingsDB is used to perform set difference, which removes elements from the first set that are also present in the second set.

3. The only expression which evaluates to `true` is the expression "d". The set with things `a` and `c` is a subset of the set with things `a`, `b` and `c`.

4. Because sets are not directly supported by the ThingsDB communication protocol, they must be converted to a more fundamental data type when sent to a client. Therefore, the *set* is converted into a *list*, which is a sequence of ordered elements. While this conversion enables clients to receive and process sets, it is essential to remember that sets are inherently unordered collections. As a result, you cannot rely on the order of things in the returned list as they may not reflect the order in which the things were added to the set.

5. When a *set* is appended to a *list*, ThingsDB will create a copy and convert the set into a *tuple*, which is an immutable collection of elements.

Chapter 6 - Procedures

In ThingsDB, we've encountered built-in functions like `is_nil()` and methods like `.len()`, which are functions called on instances of a type. We've also explored closures, which are unnamed functions commonly used as callback arguments to iterate over lists, sets or things.

ThingsDB introduces another variation of these concepts called *procedures*. Procedures bind a name to a closure and expose the procedure for direct execution. This means we can call procedures directly, eliminating the need for queries. Procedures are typically created within a collection, but ThingsDB also offers the `/thingsdb` scope for procedures that simplify management tasks like user creation which we will explore later in this chapter.

In this chapter, we'll begin crafting a to-do application, empowering you to manage Alice's tasks and keep her organized.

First, let's create a new collection named "todos" using the `/thingsdb` scope and switch to the `//todos` scope.

```
(/thingsdb)> new_collection("todos");
"todos"
(/thingsdb)> @ //todos
(//todos)>
```

To manage Alice's to-do items, we'll create a list to store them. This list will hold strings representing the to-do items. To control what gets added to the list, we'll use a procedure.

```
(//todos)> .todos = [];  // Create an empty list for to-do items
[]
```

Instead of directly adding to-do's to the list, we'll create a procedure to manage this process.

```
(//todos)> new_procedure("add_todo", |body| {
    "Adds a to-do to the list.";   // Docstring
    assert(is_str(body) && body);  // Check if the input is a
                                   // non-empty string
    .todos.push(body);  // Add the to-do to the list
    nil;
});
"add_todo"
```

This creates a procedure named `add_todo` that takes a single `todo` argument. The docstring specifies the procedure's purpose and is optional. The `assert()`

83

function ensures that the input is a valid, non-empty string.

To utilize the `add_todo` procedure, we can simply call it directly in a query, similar to a regular function.

```
(//todos)> add_todo("Read a book");
OperationError: closures with side effects require a change but none is
created; use `wse(...)` to enforce a change; see
https://docs.thingsdb.io/v1/collection-api/wse
```

ThingsDB throws an error indicating that the procedure contains a closure which may introduce changes to the collection and requires a change to synchronize the state. This is because ThingsDB must ensure data consistency across all nodes when changes occur.

6.1 Side Effects and Changes

To explicitly signal to ThingsDB that the query involves side effects and requires a change, we can use the `wse()` function. This function informs ThingsDB that the query will modify the collection and necessitates a change to maintain data integrity.

> In most cases, ThingsDB automatically detects and handles the need for changes when queries modify data. However, there may be situations where this automatic detection is not possible or desired.
>
> To explicitly instruct ThingsDB to initiate a change, the `wse()` *(with-side-effects)* function can be used. This function explicitly informs ThingsDB that the query involves modifications to the collection and requires a change operation to ensure data consistency across nodes.
>
> Conversely, there may be scenarios where ThingsDB incorrectly identifies a change as necessary, even though it is not the case. To prevent unnecessary changes, the `nse()` *(no-side-effects)* function can be employed. This function tells ThingsDB that the query does not involve any data modifications and therefore does not require a change operation.
>
> By utilizing these functions, developers can effectively manage change propagation in ThingsDB, ensuring data consistency and avoiding unnecessary change operations.

Here is the corrected code snippet with the wse() function:

```
(//todos)> wse();  // Tell ThingsDB to initiate a change
add_todo("Read a book");
null
```

By including the `wse()` function, ThingsDB recognizes the potential data modification and initiates the necessary change operations to maintain consistency across nodes. This approach ensures that any changes made by the procedure are applied in a coordinated manner, preserving the integrity of the data managed by ThingsDB.

In addition to calling procedures within queries, ThingsDB also allows direct procedure calls without the need for queries. While the ThingsDB Prompt does not support direct procedure calls, this opens up the opportunity to utilize Python code to interact with ThingsDB. As mentioned earlier, Python is not the only programming language supported by ThingsDB; there are also clients available for C#, Go, PHP, JavaScript/Node.js and maybe more by the time you read this book. Furthermore, procedures can be invoked through the ThingsDB HTTP API, a topic we'll explore in Chapter 16.

6.2 Python

To streamline the development process, we'll create a reusable template that serves as a foundation for our code examples. This template will encapsulate the common connection and authentication steps, allowing us to focus on the specific procedure call logic.

Create a file named `template.py` and save the following code:

```python
import asyncio
from thingsdb.client import Client

async def work(client: Client):
    pass

async def main():
    client = Client()
    client.set_default_scope('//todos')
    await client.connect('localhost')  # Replace with the actual ThingsDB
                                       # server address
    try:
        await client.authenticate('admin', 'pass')  # Replace with your
                                                     # ThingsDB credentials
        await work(client)
    finally:
        client.close()
        await client.wait_closed()

asyncio.run(main())
```

This template assumes ThingsDB is running on localhost. If this is not the case, modify the `connect()` function call to specify the correct address and, optionally, a port number. Similarly, update the `authenticate()` function call with your actual ThingsDB credentials.

To test the template, run the following command:

```
$ python template.py
$
```

The code should execute without errors, demonstrating the successful connection and authentication with ThingsDB.

6.2.1 Run Procedure

To execute the `add_todo` procedure directly, we'll create a copy of the template Python script and name it `add_todo.py`.

Within the work function, we'll modify the code to handle user input and invoke the procedure:

```
async def work(client: Client):
    todo_body = input("To-do body: ")  # Prompt user for to-do item
    await client.run('add_todo', body=todo_body)  # Call the add_todo
                                                   # procedure
```

> 💡 Since we previously called `client.set_default_scope('//todos')` in the template, we do not need to explicitly specify the scope in this instance.

Save the file and run the following command to execute the code:

```
$ python add_todo.py
To-do body: Brush your teeth
$
```

If successful, no errors should be shown.

6.2.2 Perform a Query

Next, create another file based on the template and name it `search_todos.py`:

```
async def work(client: Client):
    needle = input("Search for: ")  # Prompt user for search input
    res = await client.query("""//ti
        .todos.filter(|todo| todo.contains('""" + needle + """'));
    """)
    print(res)
```

> 💡 This code utilizes a multi-line string to define the query and prefixes it with `//ti` to improve syntax highlighting in some IDEs. However, this is not mandatory and does not affect the query execution.

Execute the following command to run the `search_todos.py` file:

```
$ python search_todos.py
Search for: book
['Read a book']
```

The output confirms that the to-do item `"Read a book"` has been added and retrieved successfully.

6.2.3 Prevent Code Injections

While the code works, it also contains a potential security flaw. The direct embedding of user input into the query structure makes it vulnerable to code injections, allowing malicious code to manipulate the collection's properties.

Consider the scenario where a malicious user inputs the following:

```
$ python search_todos.py
Search for: ' + {.hacked = true; "book";} + '
['Read a book']
```

This input, when passed directly into the query, would effectively create an unauthorized property `.hacked` on the `//todos` collection root. This highlights the potential for code injections to compromise the integrity of ThingsDB data.

To address this vulnerability, it is essential to separate user input from the query structure by using variables. This technique allows for a more secure and controlled handling of user-provided data.

To enhance security and prevent code injections, we'll modify the `search_todos.py` code to use variables for user input:

```
async def work(client: Client):
    needle = input("Search for: ")  # Prompt user for search input
    res = await client.query(
        """//ti
            .todos.filter(|todo| todo.contains(needle));
        """,
        needle=needle)
    print(res)
```

The original code, which directly embedded the user input into the query, was vulnerable to code injections. By using variables, we can isolate the user input and prevent it from affecting the query structure. This safeguard protects the integrity of the ThingsDB database and ensures that only authorized operations are executed.

> Another benefit of separating input from the query structure is that it enables ThingsDB to cache the query, independent of the user input. This caching mechanism can significantly improve the performance of frequently executed queries, especially those that involve user-provided data.
>
> By isolating user input, ThingsDB can store the compiled query separately from the input values. This allows the database to pre-compile and optimize the query for later execution, regardless of the specific input received. As a result, subsequent executions of the same query can be performed more efficiently, reducing response times and improving overall application performance.
>
> In contrast, if the user input were directly embedded within the query, ThingsDB would need to recompile the query each time it was executed with different input. This would lead to increased processing overhead and potentially slower query performance.

6.2.4 Migrating from Query to Procedure

To enhance code organization, we'll migrate the *"search_todos"* query to a reusable procedure. This approach encapsulates the search logic and makes it easily accessible from multiple parts of the application.

Create the procedure in a tool like *things-prompt* or *ThingsGUI*. Having explored code-based query execution, you might even prefer to use Python or some other language for code execution. In this book we'll stick with the *things-prompt* for simple code snippets:

```
(//todos)> new_procedure("search_todos", |needle| {i
    "Search for todo's";
    .todos.filter(|todo| todo.contains(needle));
});
"search_todos"
```

Update the `work` method in `search_todos.py` to call the procedure:

```
async def work(client: Client):
    needle = input("Search for: ")  # Prompt user for search input
    res = await client.run("search_todos", needle=needle)
    print(res)
```

By using a procedure, we've encapsulated the search logic and made it reusable. This approach promotes code modularity, improves maintainability, and reduces development complexity.

6.3 Requesting Procedure Information

Procedures are stored within a scope, but they are not directly attached to the root of the collection. To retrieve information about procedures within a specific scope, the `procedures_info()` function is employed. This function returns a list of procedure information objects, each containing detailed metadata about the procedure.

Calling `procedures_info()` within the `//todos` scope produces the following output:

```
(//todos)> procedures_info();
[
    {
        "arguments": ["todo"],
        "created_at": 1706363614,
        "doc": "Adds a to-do to the list.",
        "name": "add_todo",
        "with_side_effects": true
    },
    {
        "arguments": ["needle"],
        "created_at": 1706540544,
        "doc": "Search for todo's",
        "name": "search_todos",
        "with_side_effects": false
    }
] //-- some information is left out to keep the sample comprehensible
```

The properties of each procedure in this list provide valuable insights. The `with_side_effects` property indicates whether invoking the procedure modifies the dataset or not. Additionally, each procedure's arguments are listed, and a brief description of its functionality is provided in the `doc` property *(this is the docstring we provided earlier in the closure body)*.

6.3.1 Extracting Properties from Information Objects

Now that we have a basic understanding of procedure information objects, let's focus on extracting just the procedure names.

```
(//todos)> type(procedures_info());
"list"
(//todos)> type(procedures_info().first());
"mpdata"
```

Observing the data types involved, we can see that the `procedures_info()` function returns a list of *mpdata* objects, not *things*. MPdata stands for "MessagePack-Data" and represents serialized MessagePack data. Information objects in ThingsDB are pre-serialized to minimize work and enhance performance.

To extract the procedure names, we can utilize the `load()` method provided by the mpdata type. This method allows us to deserialize mpdata objects into ThingsDB objects, enabling us to access their properties.

Applying `load()` to a procedure information object results in a thing object. This enables us to access the `name` property of the procedure, which provides the procedure's identifier.

With this approach, we can efficiently retrieve a list of procedure names using the following code:

```
(//todos)> procedures_info().map(|mpdata| mpdata.load().name);
[
    "add_todo",
    "search_todos"
]
```

This concise code demonstrates the ability to extract relevant information from pre-serialized MessagePack data.

6.3.2 Additional Procedure Functions

We've covered the `new_procedure()` and `procedures_info()` functions, but ThingsDB offers a few more functions that you can find in a dedicated documentation section.

For detailed information on these additional procedure functions, please refer to the ThingsDB documentation: https://docs.thingsdb.io/v1/procedures-api/

6.4 Thinking Ahead

While the current setup of the to-do application works well for Alice, it is important to consider scalability and maintainability from the start. While full admin access might be sufficient for now, granting such extensive permissions to the Python code might not be the best approach in the long run.

To address this, we can create a dedicated user for the Python code and grant them only the necessary permissions. This will provide a more secure and controlled environment for managing user access.

Using the `grant()` function, we can assign the appropriate permissions to the newly created user. The table below summarizes the various permission flags available in ThingsDB:

Table 6.4 - Summary of permission flags

Mask	Description
QUERY (1)	Grants access to execute queries
CHANGE (2)	Grants access to make changes
GRANT (4)	Grants administrative privileges, allowing users to grant and revoke permissions to other users
JOIN (8)	Grants join (and leave) privileges to a room
RUN (16)	Grants access to run procedures directly
USER (27)	Combination of QUERY, CHANGE, JOIN and RUN
FULL (31)	Combination of all privileges

Since we've migrated all the queries from the Python code to procedures, we no longer require QUERY access. Therefore, we can safely grant the user CHANGE and RUN permissions, which will allow them to modify data and execute procedures within the "todos" scope.

By adopting this approach, we can effectively differentiate between the administrative user (Alice) and the Python code, ensuring that the code has the appropriate level of access to perform its tasks while maintaining security and control over the system.

Instead of creating users manually, we can create a procedure to automate the process. This will make it easier to manage users and create new ones as needed.

Let's create a procedure called `new_todo_user` in the `/thingsdb` scope that takes a user name and access flags as input and generates a unique token for

authentication.

```
(//todos)> @ /t;  // Switch to the /thingsdb scope
(/t)> new_procedure("new_todo_user", |name, access| {
    "Creates a new user with access to the todos scope.";
    new_user(name);
    grant('//todos', name, access);
    new_token(name);
});
"new_todo_user"
```

This procedure first creates a new user with the specified name using the `new_user()` function. Then, it grants the desired user permission for the `//todos` scope using the `grant()` function. Finally, it generates a unique token for the user using the `new_token()` function.

> 💡 Instead of using a token, we could have opted for username and password authentication through the `set_password()` function. However, tokens offer several advantages: they are strong, easily revoked (using: `del_token()`), and allow for issuing multiple tokens per account for different purposes.

To create a user named `py` and obtain their token, run the following command:

```
(/t)> wse(); new_todo_user('py', RUN | CHANGE);
"9U0unWQglFlxl1DUBX5JsR"
```

This will create a user named `py` and provide you with their token, which you should copy and paste into the corresponding Python script.

Now, modify the authentication call in the `template.py` and `search_todos.py` scripts to replace the default token with the generated one:

```
    …
    try:
        await client.authenticate('9U0unWQglFlxl1DUBX5JsR')
        await work(client)
    …
```

Replace `"9U0unWQglFlxl1DUBX5JsR"` with the actual token you obtained from the previous command.

With this change, the Python scripts should now use the `py` user's token to authenticate with ThingsDB and access the `//todos` scope. This provides a more secure and maintainable approach to managing user access.

6.5 Quiz - Challenge Your Understanding

1. Under what circumstances is the `wse()` function necessary?

2. Will the following query succeed in executing the `set_x()` procedure and modifying the corresponding property in the collection?
   ```
   set_x(123);
   ```

3. How can you invoke a procedure in ThingsDB?
   a. Call the procedure directly from a query
   b. Use the ThingsDB protocol to run the procedure directly
   c. Use the HTTP API to run the procedure
   d. All of the above

4. In the following query, what is the data type of the variable `info`?
   ```
   info = procedure_info("set_x");
   type(info); // ???
   ```

5. Which authentication methods does ThingsDB support?

6. If you want to restrict your application to executing only predefined procedures that do not modify the state, which authentication flag(s) should you use?
   a. `QUERY`
   b. `READ`
   c. `CHANGE`
   d. `RUN`
   e. `USER`

7. Verify if the `.hacked` property is present in the todos collection. If so, can you remove the property to ensure the integrity of the data?

6.5.1 Quiz - Answers

1. ThingsDB employs intelligent algorithms to identify queries with side effects that modify system state. However, in certain scenarios, these algorithms may fail to detect side effects, necessitating the explicit use of the `wse()` function. One such instance arises when invoking procedures that themselves encompass side effects.

2. No, the provided query will not work without explicitly invoking the `wse()` function.

3. Answer "d". All methods can be used to execute procedures in ThingsDB.

   - You can simply include the procedure's name in your query, and ThingsDB will execute the procedure and return the result.
   - The ThingsDB protocol allows you to make a direct call to a procedure without invoking a query. Native clients for popular programming languages, including Python, C#, Go, PHP, JavaScript/Node.js and more, provide built-in support for this direct procedure invocation.
   - If you have enabled the HTTP API, you can also use it to call procedures. This can be a good option if you want to call procedures from a web application or other client that does not support the ThingsDB protocol. We'll explore the HTTP API in more detail in Chapter 16.

4. While the client response for `procedure_info()` might initially seem to imply that the return value is a *thing*, it is actually an instance of *mpdata*, a specialized data type that represents serialized MessagePack data. To convert mpdata, you can call the `load()` method on the mpdata object. This will effectively de-serialize the MessagePack data and return the corresponding thing instance.

5. ThingsDB offers two primary authentication methods: token-based and username/password. ThingsDB provides the `new_token()` function for generating secure tokens and the `set_password()` function for managing user credentials.

6. The correct answer is `RUN`.

   - The `QUERY` flag enables users to execute queries that retrieve and process data from ThingsDB. However, the goal is to restrict the application to executing only pre-defined procedures without modifying the system state.
   - The `READ` flag is not a valid authentication flag in ThingsDB.

- The CHANGE flag allows users to modify data. Since the goal is to prevent the application from modifying the state, the CHANGE flag is not suitable.
- The USER constant serves as a convenient shorthand for combining the commonly used QUERY, CHANGE, JOIN, and RUN permissions into a single name.

7. In , we explored code injection vulnerabilities. If you executed the code snippet provided in that section, a new property named .hacked may have been added to the collection. To remove this unintended property, you can use the following code:

```
.del("hacked");
```

95

Chapter 7 - Typed Things

Currently, Alice can add and search for to-do items, but she lacks a straightforward method to mark them as completed. While we could implement a procedure to remove completed to-do items, Alice prefers to retain them for future reference.

To achieve this, we'll transform the current *"todo"*, which is simply a string, into a thing with both a `body` property and a boolean property which we call `done`. We initially set `done` to `false` and update the property to `true` when the to-do is marked as completed.

However, using things without proper type constraints can lead to errors due to incorrect property names or unexpected values. To enhance control and prevent such issues, we'll explore the concept of types in this chapter. In practical applications, it is rare to store plain things; instead, we mainly choose to employ typed things.

> 💡 Typed things offer significant memory advantages over regular things with identical properties. This is because typed things in memory only store the data associated with that particular instance, while the property names are stored only once, centrally within the type definition. This centralized storage significantly reduces memory consumption, particularly for large collections of typed things.

7.1 Create Your First Type

To create a type, you'll need to use the `new_type()` function to define the type and the `set_type()` function to initialize it with property definitions. While it is technically possible to skip `new_type()` and call `set_type()` directly, this approach may lead to issues with self-referential type definitions. To avoid potential problems, it is recommended to follow the conventional practice of using `new_type()` first, followed by `set_type()`. Unlike procedures, types can only be created within a collection's scope.

```
(//todos)> new_type("Todo");
set_type("Todo", {
    body: "str",
    done: "bool",
});
null
```

Both `new_type()` and `set_type()` require a type name as their first argument. It is considered best practice to use CamelCase naming convention for types,

starting with a capital letter. The `set_type()` function takes a second argument, which is a *thing* object that defines the properties and their respective definitions. Property definitions are always expressed as strings.

Once the `Todo` type is created, you can initialize an instance of it by simply writing `Todo{}`. Every type can be initialized without explicitly specifying the properties. This implies that every property must have a default value at initialization.

For instance, here's how to initialize a `Todo` instance using its default values:

```
(//todos)> Todo{};  // Initialize a Todo using the default values
{
    "body": "",
    "done": false
}
```

This creates a new `Todo` instance with an empty `body` property and a `done` property set to `false`. As you can see, the default values for `body` and `done` are automatically assigned when the instance is created.

7.1.1 Enhancing the Todo Type using mod_type()

We defined the `Todo` type to accept empty strings for the `body` property. However, this is not ideal, as we want to ensure that to-do's always have a meaningful description. To address this, we can utilize property definitions to impose stricter constraints on the `body` property.

Instead of simply `"str"`, we can specify a range condition using `"str<1:>"`. This indicates that the body must be at least one character long. We could also define maximum length restrictions or additional default values if needed.

To modify the `Todo` type, we'll employ the `mod_type()` function. This function is specifically designed for type modification and is commonly used in migration scripts to streamline data migration processes. It is somewhat analogous to the `ALTER TABLE` syntax in SQL, often employed when introducing new features or enhancing existing applications.

The `mod_type()` function takes the type name as its first argument, followed by an *action* string specifying the desired operation. In this case, we'll use the `"mod"` action, which indicates a property definition modification.

Here's a table summarizing the available `mod_type()` actions:

Table 7.1.1 - Summary of mod_type() actions

Action	Description
`"add"`	Adds a new property or method to the type
`"all"`	Iterates over all instances of a given type
`"del"`	Deletes a property or method from the type
`"hid"`	Enables or disable *hide-id* for the type
`"mod"`	Modifies a property or method definition
`"rel"`	Creates a relation between types
`"ren"`	Renames a property or method
`"wpo"`	Enables or disable *wrap-only* mode for the type

The specific arguments required depend on the chosen action. For `"mod"`, we need to provide the property name, the new definition, and a migration callback if the new definition is incompatible with the old one. This is the case here, as the old definition allowed empty strings, while the new one does not. The migration callback will be invoked for each instance of the type and should return either `nil` to retain the original property value or a new value that complies with the updated definition.

Now, let's migrate the "todos" collection to utilize the updated `Todo` type:

```
(//todos)>
// Migrate existing string values to the Todo type
.todos = .todos.map(|body| Todo{body:});

// Modify the Todo type to disallow empty bodies
mod_type("Todo", "mod", "body", "str<1:>", |todo| !todo.body ? "-" : nil);

// Migrate procedures to ensure compatibility with the updated type
mod_procedure("add_todo", |body| {
    "Adds a to-do to the list.";
    .todos.push(Todo{body:});
    nil;
});
mod_procedure("search_todos", |needle| {
    "Search for to-do's";
    .todos.filter(|todo| todo.body.contains(needle));
});
null
```

By applying the migration in a single query, we eliminate the risk of using deprecated procedures or performing operations on incompatible data structures.

To verify that the migration to the `Todo` type has been successful, let's execute the `search_todos.py` script again:

```
$ python search_todos.py
Search for: book
[{'#': 2, 'body': 'Read a book', 'done': False}]
```

7.1.2 Customizing ID Representation in Responses

The previous example demonstrated how to retrieve a list of `Todo` objects using the `search_todos.py` script. The response included the properties of the `Todo` type along with the unique identifier for each object, represented by the `#` key.

While this approach works, it is possible to gain more control over how IDs are returned in responses by leveraging type definitions. Specifically, we can utilize the `"#"` definition to specify the desired key for the ID.

To achieve this, we'll employ the `mod_type()` function again. This time, we'll modify the `Todo` type to return the ID as `"id"` instead of `"#"`.

```
(//todos)> mod_type("Todo", "add", "id", "#");
null
```

This modification tells ThingsDB to return the unique identifier for each `Todo` object as the `"id"` key in responses.

Now, let's re-execute the `search_todos.py` script to observe the change:

```
$ python search_todos.py
Search for: book
[{'id': 2, 'body': 'Read a book', 'done': False}]
```

As expected, the response now includes the `"id"` key instead of `"#"`, providing a more consistent and user-friendly representation of the object identifier.

7.2 Collection Structure with Types

While we've established a well-defined structure for `Todo` objects, the `.todos` list still accepts items of other types, potentially introducing inconsistencies.

Additionally, the collection root lacks strict structure, allowing for arbitrary property assignments.

To address these concerns, we'll create a new type which we call `Root` to represent the collection root. The name `Root` is just a choice, another good name would be `Collection` or a more descriptive name like `TodoCollection`. This type will solely contain a property named `todos` with type definition `"[Todo]"`, ensuring that only `Todo` objects can be added to the list.

Let's create the `Root` type:

```
(//todos)> new_type("Root");
set_type("Root", {
    todos: "[Todo]",
});
null
```

To convert the collection root to the `Root` type, we'll use the `to_type()` method.

```
(//todos)> .to_type("Root");
null
```

> 💡 If you encounter an error indicating that the **Root** type is missing a property called hacked, it is likely due to the hacked property introduced in [Chapter 6.2.3](#). To resolve this, remove the hacked property using `.del("hacked");`

With the `Root` type applied to the collection root, we have achieved stronger type enforcement. Attempts to assign arbitrary properties or add non-`Todo` objects to the `.todos` list will trigger errors.

Here are some examples illustrating this enhanced type safety:

```
(//todos)> .x = 1;
LookupError: type `Root` has no property `x`
(//todos)> .todos.push(123);
TypeError: type `int` is not allowed in restricted array
(//todos)> type(root());
"Root"
(//todos)> .todos.restriction();
"Todo"
```

By leveraging typed roots, we've significantly enhanced the structure and type safety of our collection, promoting data integrity and consistency.

As your application grows and evolves, you may need to adapt your data structures to accommodate new requirements or enhance existing features.

To address such changes, the earlier discussed `mod_type()` function is available, enabling you to modify existing types by adding new properties, changing data types, or introducing restrictions.

If, for any reason, you need to revert a type back to its original *"thing"* form, the `to_thing()` method is your go-to solution. This method effectively destructures the typed thing, removing any imposed constraints and allowing for arbitrary property assignments.

7.3 Retrieving Typed Things by ID

In Chapter 4.2, we learned how to retrieve a specific thing using the `thing()` function by providing its ID. This method also applies to typed things, but for greater control, we can directly invoke the type name and pass the ID as an argument. This approach ensures that only instances of the specified type are returned.

Consider the following examples:

```
(//todos)> Todo(2);  // ID 2 is a Todo item
{
    "body": "Read a book",
    "done": false,
    "id": 2
}
(//todos)> Todo(1);  // ID 1 is the collection root
TypeError: `#1` is of type `Root`, not `Todo`
```

In the first example, the `Todo(2)` call successfully retrieves the `Todo` object with ID 2. However, when attempting to access the collection root using `Todo(1)`, an error is raised because the collection root is a `Root` object, not a `Todo` object.

7.4 Type Methods

Up to this point, the `Todo` type has a property named `done` to indicate whether a to-do item is completed. However, if Alice finishes reading a book, she needs a way to mark the corresponding to-do item as completed. While we could simply set the `done` property to `true`, it is better to adopt a more flexible approach that allows for potential future changes in data representation.

Type "methods" are self-contained functions within a type definition, allowing us to encapsulate specific behavior for that type. Instead of directly modifying the `done` property, we'll create a method named `mark_as_done` that handles the task completion logic.

101

In contrast to properties, which are defined using strings, methods are defined using closures. These closures encapsulate the method's logic and receive the `this` object as their first argument, which represents the instance of the type where the method is being called on.

We'll use the `mod_type()` function to add the `mark_as_done` method to the `Todo` type:

```
(//todos)> mod_type("Todo", "add", "mark_as_done", |this| this.done =
true);
null
```

Next, we'll create a procedure to interact with the `mark_as_done` method. This procedure will receive a `todo_id` as input and call the `mark_as_done` method on the corresponding `Todo` instance:

```
(//todos)> new_procedure("mark_as_done", |todo_id| {
    wse();
    Todo(todo_id).mark_as_done();
    nil;
});
"mark_as_done"
```

> 💡 Here, we again need to use the `wse()` function to inform ThingsDB that this procedure may modify the "todos" collection and potentially trigger side effects.

To test the new `mark_as_done` procedure, we'll create a Python file named `mark_as_done.py` based on the template and modify the work method to capture user input for the `todo_id` and call the procedure:

```
async def work(client: Client):
    todo_id = int(input("To-do ID: "))
    await client.run('mark_as_done', todo_id=todo_id)
```

Running this Python file will prompt you to enter a `todo_id`, which will then be used to invoke the `mark_as_done` procedure.

```
$ python mark_as_done.py
To-do ID: 2
```

If no errors occur, it indicates that the operation was successful.

To verify that the to-do item has been marked as completed, you can run the `search_todos.py` script again and observe the updated `done` property:

```
$ python search_todos.py
Search for: book
[{'id': 2, 'body': 'Read a book', 'done': True}]
```

7.5 Type Information

With the `types_info()` function, you can explore all the types defined within a collection. The function returns a list of information objects.

For detailed information about a single type, use the `type_info()` method. The `Todo` type information, for example, would look like this:

```
(//todos)> type_info("Todo");
{
    "created_at": 1706696007,
    "fields": [
        ["id", "#"],
        ["body", "str<1:>"],
        ["done", "bool"]
    ],
    "hide_id": false,
    "methods": {
        "mark_as_done": {
            "arguments": ["this"],
            "definition": "|this| this.done = true",
            "doc": "",
            "with_side_effects": true
        }
    },
    "modified_at": 1707128380,
    "name": "Todo",
    "relations": {},
    "type_id": 0,
    "wrap_only": false
}
```

7.6 Removing a Type

ThingsDB empowers you to remove existing types using the `del_type()` function. However, exercise caution, as this function operates even when instances of the targeted type still exist.

Consider the following example:

```
(//todos)> new_type("T");        // Create type T
set_type("T", {name: "str"});    // Define type T
t = T{name: "test"};             // Create an instance of T
assert(type(t) == "T");          // Verify the type
del_type("T");                   // Delete type T
assert(type(t) == "thing");      // Verify type is now "thing"
t;
{
    "name": "test"
}
```

As observed, deleting the type converts all existing instances to plain "things", discarding the specific type information. This transformation is not easily reversible unless you possess precise knowledge of the original type belonging to each instance.

7.6.1 Dependency Considerations

Furthermore, attempting to remove a type referenced by another type or enumerator will fail. To successfully delete such a type, you must first address the dependencies. Refer to Chapter 9 for in-depth exploration of type connections and Chapter 11 for detailed discussions on enumerators.

7.7 More Definitions

In this book we'll explore more definitions than discussed so far. We already used a definition to specify a string with a range, a boolean and a list restricted to a specific data type.

As always, use the documentation page for a full description of all the type property definitions which can be found here:
https://docs.thingsdb.io/v1/overview/type/

7.8 Quiz - Challenge Your Understanding

1. Choose which statement declares a new type?
   a. `del_type("A");`
   b. `new_type("A");`
   c. `mod_type("A");`
   d. `set_type("A");`

2. Can you identify which strings fit the property definition `"str<4:8>"`?
   a. `"ThingsDB"`
   b. `"Hi"`
   c. `nil`
   d. `"Hello"`
   e. `"Bicycle Race"`

3. How can you seamlessly integrate the ID of type `T` instances as a property named `"identifier"` in responses, given that the ID is currently exposed as `"#"`?

4. Suppose a type `T` exposes the ID for each instance as key `"identifier"` *(see previous question)*. How can you change the name for this key to `"id"`?

5. Complete the `set_type("P", ???);` statement to create type `P`, enabling the following code to function flawlessly:
   ```
   a = P{x: 3.0, y: 2.0};
   b = P{x: 7.0, y: 8.0};
   d = a.distance(b); // Should produce 7.211102550927978
   ```
   Hint: recall the distance formula: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

6. Can you produce a query that returns a list containing *only the type names* present in the collection?

7. When a type is removed using `del_type()`, what happens to existing instances of that type?

7.8.1 Quiz - Answers

1. The correct answer is "b". While `set_type()` can technically be used to create a new type, it is primarily used to set the properties and/or methods for a type. It requires an additional argument, even when not setting any properties or methods. `del_type()` is the opposite, it removes a type instead of creating one. `mod_type()` is used to modify a type and at least requires an "action" as second argument.

2. Both "a", `"ThingsDB"` and "d", `"Hello"` are valid. The defined range accepts strings from 4 up till 8 characters, both inclusive. The string `"ThingsDB"` contains exactly 8 characters so it is a valid string. Type `nil` is not allowed for this definition since only type string is allowed.

3. By executing the following statement:
   ```
   mod_type("T", "add", "identifier", "#");
   ```

4. A type can only have a single `"#"` definition. By executing the following statement the key will be renamed from `"identifier"` to `"id"`.
   ```
   mod_type("T", "ren", "identifier", "id");
   ```

5. To enable distance calculations between points, create type `P` with the following masterful definition:
   ```
   set_type("P", {
     // Properties for point coordinates
     x: "float",
     y: "float",

     // Method for calculating distance:
     distance: |this, other| {
       // Distance formula
       sqrt(pow(other.x - this.x, 2.0) + pow(other.y - this.y, 2.0));
     },
   });
   ```

6. Use `types_info().map(|info| info.load().name);` to get a list of type names. This iterates through type info objects, extracting and returning only the `name` property.

7. Instances of the removed type are downcast to regular things. This means that while their existing properties and values remain intact, the original type information becomes unavailable.

Chapter 8 - Date, Time and Tasks

Effectively managing to-do lists involves not only tracking tasks themselves, but also the timing of their creation and completion. In this chapter, we delve into two core methods for capturing time in ThingsDB: *timestamps*, the *datetime* type and the *task* type used for scheduling code execution.

8.1 Timestamps vs. Datetime

Timestamps represent time as a single numerical value, specifically the number of seconds elapsed since January 1, 1970, 00:00:00 UTC. This representation offers compact storage and is suitable for calculations involving relative time differences.

The `now()` function retrieves the current timestamp:

```
(//todos)> now();
1707145945.4736154
```

While timestamps offer efficient storage and ease of calculating time differences between events, their single numerical representation can pose challenges when dealing with absolute dates or time zones. These scenarios often require intricate calculations to extract the desired information.

ThingsDB simplifies time-related tasks with the datetime type, offering a human-readable and efficient approach to date and time manipulation.

Unless explicitly overridden with the `set_time_zone()` function, datetime objects in ThingsDB default to Coordinated Universal Time (UTC), the global standard for timekeeping.

Generate a datetime object reflecting the current date and time by simply calling the `datetime()` function without arguments:

```
(//todos)> datetime();
"2024-02-05T15:22:05Z"
```

The datetime object is returned in a response as a string using the ISO 8601 format.

> 💡 While powerful, datetime objects have limited precision, offering only up to the *second* level. If your needs require millisecond-level accuracy or finer granularity, alternative approaches like timestamps might be more suitable.

The `datetime()` function offers a variety of methods to create datetime objects, ensuring adaptability to diverse data sources and formatting preferences.

Construct datetime objects with year, month, and day:

```
(//todos)> datetime(2024, 2, 6);
"2024-02-06T00:00:00Z"
```

Add optional hour, minute, and seconds:

```
(//todos)> datetime(2024, 2, 6, 9, 57, 30);
"2024-02-06T09:57:30Z"
```

When the final argument passed to `datetime()` is a string, ThingsDB interprets it as a time zone identifier. This allows you to "ground" your datetime object in a specific geographical location, ensuring accurate representation across regions:

```
(//todos)> datetime(2024, 2, 6, 9, 58, "Europe/Kyiv");
"2024-02-06T09:58:00+0200"
```

In this example, the time zone `"Europe/Kyiv"` is specified, resulting in a datetime object reflecting the time in Kyiv, Ukraine. For a comprehensive overview of all available time zones supported by ThingsDB, use the `time_zones_info()` function.

Create datetime objects from ISO 8601 formatted strings:

```
(//todos)> datetime("2024-02-06T10:05:00Z");
"2024-02-06T10:05:00Z"
```

Accommodate diverse date and time representations using format specifiers:

```
(//todos)> datetime("2024-2-6", "%Y-%m-%d");
"2024-02-06T00:00:00Z"
```

Convert a timestamp to a datetime object and return using a custom format:

```
(//todos)> datetime(1707211569).format("%a, %d %b %Y %H:%M:%S %Z");
"Tue, 06 Feb 2024 09:26:09 UTC"
```

As illustrated in the provided examples, datetime object initialization in ThingsDB boasts remarkable flexibility. For a comprehensive overview of possible initialization options, delve into the detailed online documentation: https://docs.thingsdb.io/v1/collection-api/datetime/

Once initialized, datetime objects uphold immutability, safeguarding the integrity of their original data. Consequently, methods like `move()`, `to()`, and `replace()` always create new datetime objects, leaving the original unaltered.

Need to tweak specific components within a datetime object? The `replace()` method is your friend. It allows you to craft new instances with adjusted values, ensuring pinpoint accuracy:

```
(//todos)> datetime("2024-02-06T09:26:09Z").replace({
    minute: 0,
    second: 0,
});
"2024-02-06T09:00:00Z"
```

Accurate time representation across different regions is crucial. With the `to()` method, you can seamlessly apply different time zones to your datetime objects:

```
(//todos)> datetime("2024-02-06T09:26:09Z").to("Europe/Kyiv");
"2024-02-06T11:26:09+0200"
```

Move through time with ease using the `move()` method. It allows you to generate new datetime objects adjusted by specified intervals, empowering you to navigate temporal data efficiently:

```
(//todos)> datetime("2024-02-06T09:26:09Z").move("years", -1);
"2023-02-06T09:26:09Z"
```

By mastering these and other methods, you'll unlock a powerful toolkit for manipulating datetime objects within ThingsDB projects. This ensures precision and flexibility in managing your time-related data.

8.1.1 Bridging the Gap Between Datetime and Timestamp

Converting a datetime object to a timestamp can be achieved effortlessly by casting it as an integer.

```
(//todos)> int(datetime("2024-02-06T09:26:09Z"));
1707211569
```

ThingsDB also offers a data type called *timeval*. While functionally equivalent to *datetime*, it has a key difference: instead of returning an ISO 8601 formatted string, timeval returns a raw UTC timestamp integer in its responses. Think of it as a more compact and optimized version of datetime for timestamp-related tasks.

```
(//todos)> timeval("2024-02-06T09:26:09Z");
1707211569
```

8.2 Enhancing Todo with Datetime Properties

Now that you've grasped the power of datetime for representing and manipulating temporal data, let's apply this knowledge to enhance our understanding of the `Todo` type in ThingsDB. We'll focus on adding two useful properties: `creation` and `completion`.

We want to keep track of when each `Todo` item was created. To achieve this, we'll add a new property named `creation` with the datetime type:

```
(//todos)> mod_type("Todo", "add", "creation", "datetime");
null
```

This command seamlessly integrates the `creation` property into the `Todo` type. However, since we do not possess historical data about the exact creation dates of existing to-do's, the system automatically assigns the current date and time to them. This ensures consistency while acknowledging the limitations of our historical data.

Next, we want to record the moment when a `Todo` item is marked as completed. But here's a twist: not all to-do's might be completed yet. For this, we'll introduce a property named completion, but with the `"datetime?"` definition. The question mark (`?`) signifies that this property is optional, allowing it to hold either a datetime value (representing the completion time) or `nil` (indicating no completion yet).

Here's the command to add the `completion` property:

```
(//todos)>
// Adds the completion property
mod_type("Todo", "add", "completion", "datetime?", |this| {
    this.done ? datetime() : nil;
});
// Fix the mark_as_done method
mod_type("Todo", "mod", "mark_as_done", |this| {
    this.done = true;
    this.completion = datetime();
});
null
```

This code snippet does more than simply adding the property. It also defines a closure to initialize the `completion` value appropriately for existing to-do's and it also fixes the `mark_as_done` method to set the `completion` value after the to-do will be marked as done.

8.3 Scheduling Code Execution with Tasks

Tasks allow you to schedule code execution at specific dates and times.

Key features of the task type:

- **Guaranteed consistency:** ThingsDB ensures that each task runs on exactly one node, preventing data conflicts and maintaining data integrity across the system.
- **User context execution:** Tasks execute within the context of the user who created them, inheriting their permissions and ensuring appropriate access control. However, you can change the task owner using the `set_owner()` method, granting ownership and associated permissions to a different user.
- **Automatic node distribution:** No need to worry about task allocation! ThingsDB takes care of distributing tasks across available nodes.

Let's explore what happens when we create tasks in ThingsDB:

```
(//todos)> task(datetime(), || nil);
"<task:4 owner:admin run_at:2024-02-07T14:49:36Z status:nil>"
```

This task is scheduled for immediate execution as `datetime()` specifies the current date and time. It doesn't perform any actions (`|| nil`) but serves as an example of task creation.

```
(//todos)> task(datetime(), || 1/0);
"<task:5 owner:admin run_at:2024-02-07T14:49:50Z status:nil>"
```

111

This task attempts to perform a division by zero, which will result in an error when executed. It is useful for demonstrating how ThingsDB handles task errors.

```
(//todos)> task(datetime().move("days", 1), || nil);
"<task:6 owner:admin run_at:2024-02-08T14:50:01Z status:nil>"
```

This task is scheduled to run a day later using `datetime().move("days", 1)`, showcasing how to schedule actions for specific future moments.

Use the `tasks()` function to view all tasks within the current scope:

```
(//todos)> tasks();
[
    "<task:5 owner:admin run_at:nil status:err>",
    "<task:6 owner:admin run_at:2024-02-08T14:50:01Z status:nil>"
]
```

Note that successfully completed tasks are removed from the list, while those with errors and pending tasks remain.

Task with ID 5 has encountered an error, preventing its successful completion. To uncover the cause of the problem, we'll utilize the `err()` method:

```
(//todos)> task(5).err();
"division or modulo by zero"
```

8.3.1 Cancel or Delete a Task

There are two ways to stop tasks from running. Using the `cancel()` method preserves the task information for later use and marks it with a `cancelled_err` while the `del()` method permanently removes it.

The code below illustrates the use of both methods:

```
(//todos)> task(6).cancel();  // Cancel task with ID 6
null

(//todos)> task(6).err();  // Observe cancelled error
"operation is cancelled before completion"

(//todos)> task(6).del();  // Delete task
null

(//todos)> task(6);  // Attempt to access task with ID 6 (now deleted)
LookupError: task with id 6 not found
```

112

8.3.2 Repeating Tasks

Incorporating `again_in()` within your task's code allows you to reschedule its execution based on desired time units and values. It acts like a built-in calendar reminder, prompting the task to reactivate itself after a defined period.

Let's illustrate this with an example: Alice needs a daily reminder to send reports to Bob. She can achieve this with a single task coded as follows:

```
(//todos)> task(datetime(), |task, body| {
    task.again_in("days", 1);  // Reschedule the task in 1 day
    add_todo(body);  // Add the "Send report to Bob" to-do
}, ["Send report to Bob"]);
"<task:7 owner:admin run_at:2024-02-07T14:52:25Z status:nil>"
```

In this code, the task immediately executes using `datetime()`. Within the closure, `task.again_in("days", 1)` reschedules it to run again in one day. The task then adds a to-do with the message `"Send report to Bob"`.

But what if you need to modify the report content later? No worries! The task type offers the `set_args()` method, enabling you to dynamically change task arguments without recreating the entire task.

While `again_in()` offers convenient rescheduling relative to the original task time, the task type also provides `again_at()` for pinpoint accuracy in setting future execution.

Choosing the right method:

- `again_in()`:
  - Reschedules a task relative to its original execution time.
  - Is ideal for creating repeating tasks at specific intervals (e.g., daily, hourly).
  - Example: `task.again_in("days", 1)` reschedules the task to run in 1 day.

- `again_at()`:
  - Reschedules a task to a specific new date and time.
  - Is useful for one-time adjustments or scheduling tasks for future deadlines.
  - Example: `task.again_at(datetime(2024, 02, 09, 10, 00))` reschedules the task for February 9th, 2024, at 10:00 AM.

8.3.3 Status for Repeating Task

113

As we explored, completed tasks vanished from the ThingsDB list, leaving no trace of their successful execution. But what about repeating tasks? How do we track their progress and ensure they are functioning smoothly?

A successfully completed run of a repeating task changes its status from `"nil"` to `"ok"` indicating a successful execution. You can check this status using the `task()` function, providing the task ID of your repeating task:

```
(//todos)> task(7);
"<task:7 owner:admin run_at:2024-02-08T14:52:25Z status:ok>"
```

This example shows that the task with ID 9, scheduled to repeat on February 8th, 2024, at 2:52 PM UTC, has successfully completed its previous run.

If you need to check the task status in code, use the `err()` method. This reveals any errors encountered during the last execution.

```
(//todos)> task(7).err();
null
```

If everything went well, the output will be `nil`, indicating no errors.

8.3.4 Deleting All Tasks with a Single Statement

Before starting the quiz, let's clean up the tasks we created during this chapter. Can you guess how to delete all of them at once?

Indeed! The following code snippet achieves this efficiently:

```
(//todos)> tasks().each(|task| task.del());
null
```

This expression iterates through every task using `tasks()`, and for each `task`, it calls the `del()` method to permanently remove it.

To verify, you can check the list again:

```
(//todos)> tasks();
[]
```

As you can see, the list is now empty, indicating all tasks have been successfully deleted.

8.4 Farewell, done Property

Remember the `done` property for marking completed to-do's? We can actually remove it! The `completion` property serves this purpose perfectly.

Here's how we clean up the data model:

```
(//todos)>
// Remove the done property
mod_type("Todo", "del", "done");

// Update the mark_as_done method
mod_type("Todo", "mod", "mark_as_done", |this| {
    this.completion = datetime();
});
null
```

This updated method only sets the `completion` property to the current date and time, indicating the to-do is complete.

8.5 Quiz - Challenge Your Understanding

1. What data type is returned by the `now()` function, and what does it represent?

2. How would you calculate the timestamp for January 25, 2025, at 13:05 Kyiv time, considering time zone differences and potential daylight saving time adjustments?

3. Remember the range definition from the previous chapter? Consider the property definition `"int<-1:1>?"`. What range of values would be allowed for this property?

4. You have a task that you no longer want to execute, but you still want to keep it for reference. How can you achieve this without permanently removing the task from the task list?

5. Which of the following will create a daily repeating task?
   a. `task(datetime(), |task| nil).repeat("daily");`
   b. `task(datetime(), |task| task.again_in("days", 1));`
   c. `task(datetime(), |task| task.again_at(datetime(2025, 1, 1)));`

6. Write a single line of code that evaluates all tasks and returns `true` only if all tasks are in a non-error state. If at least one task is in an error state, the code should return `false`.

7. Which of the following statements is true?
   a. The *datetime* type has millisecond precision.
   b. The *datetime* type is mutable, allowing changes like time zone adjustment.
   c. The *datetime* and *timeval* types represent the same information but differ in response format.
   d. The *timeval* type represents a timestamp, while the *datetime* type can be zone-aware.

8.5.1 Quiz - Answers

1. The `now()` function returns a floating-point number representing the current time as the number of seconds since the Unix epoch (January 1, 1970, 00:00:00 UTC).

2. The code `timeval(2025, 1, 25, 13, 5, "Europe/Kyiv");` constructs a datetime object representing the specified date and time in the Europe/Kyiv time zone and directly returns a UNIX timestamp in the response. This differs from the `datetime()` function, which returns an ISO 8601 formatted string representation of the datetime object. To obtain a UNIX timestamp from a datetime object you can convert the datetime object to an integer using the `int()` function.

3. The property definition `"int<-1:1>?"` allows integer values from -1 to 1, inclusive. The question mark (`?`) denotes that the property can also be left `nil`. So, valid options are `-1`, `0`, `1` and `nil`.

4. The `cancel()` method prevents a task from executing while keeping it in the list. The task's error state will be set to `cancelled_err`. This is different from using the `del()` method, which completely removes the task from the list, including its data.

5. Option "b" is the correct approach for creating a daily repeating task. The `again_in()` method with a `"days"` argument allows you to define a relative delay from the current time. In this case, a recurring delay of 1 day, effectively scheduling the task to run every day until it is explicitly canceled. While `again_at()` can be used, it schedules the task for a specific time on that date only. Using `again_at(datetime(2025, 1, 1))` would run the task once on January 1, 2025, not daily. In fact, if the current date is past January 1, 2025, it would indeed result in a `value_err` *("start time out-of-range")* error.

6. The following line of code would accomplish this task:

   ```
   tasks().every(|task| is_nil(task.err()));
   ```

   The function `tasks()` returns a list with all tasks and the `every()` method on the list iterates through each task and only returns `true` if the provided closure evaluates to `true` for every task. Within the closure, `is_nil(task.err())` checks if the `err()` method of each task returns with `nil`.

7. Statement "c" is correct: *datetime* and *timeval* represent the same information, but differ in response format. The *datetime* and *timeval* types offer only second precision. Both types are immutable, so

117

attempting to change their time zone creates a new object reflecting the change.

Chapter 9 - Two-Way Links

Alice's trusty to-do app has caught Bob's eye, and now it's time to take it multi-user! Let's dive into how we can transform the app to accommodate different users and their to-do's.

Let's start with creating a new type called `User` with properties for their `name` and `todos`:

```
(//todos)> new_type("User");
set_type("User", {
    name: "str",
    todos: "{Todo}",
});
null
```

> 💡 We're switching from a *list* to a *set* for `todos`. This prepares us for introducing relations later in the chapter, as relations only work between sets and plain properties, not lists.

Now comes the migration magic! Though slightly longer than previous code, keep in mind this single query transforms the entire collection from single-user to multi-user. It seamlessly adds users for Alice and Bob, migrates Alice's existing to-do's, and updates all procedures to function with the new setup.

```
(//todos)>
// Add user lookup
mod_type("Root", "add", "users", "thing<User>");

// Add procedure for adding a user
new_procedure("add_user", |name| {
    "Add a new todo user.";
    uid = name.lower();
    assert(!.users.has(uid));
    user = .users[uid] = User{name:,};
    user.id();
});

// Add users and migrate Alice's to-do's
user_id = add_user("Alice");
User(user_id).todos |= set(.todos);
add_user("Bob");

// Remove todos from Root
mod_type("Root", "del", "todos");

// Update search_todos
mod_procedure("search_todos", |needle| {
    "Search for to-do's";
    .users.values().reduce(|total, user| {
        total |= user.todos.filter(|todo| todo.body.contains(needle));
        total;
    }, set());
});

// Update add_todo
mod_procedure("add_todo", |name, body| {
    "Adds a to-do to a user's todo list.";
    todo = Todo{body:,};
    .users[name.lower()].todos.add(todo);
    todo.id();
});
null
```

So far, everything seems familiar. We can easily find a user's to-do's because they are directly stored under the `todos` property. But what if we have a specific to-do? How do we know which user it belongs to without searching through all users?

This becomes evident when using our existing `search_todos.py` script:

```
$ python search_todos.py
Search for: report
[{'id': 8, 'body': 'Send report to Bob', 'creation': '2024-02-
07T14:52:26Z', 'completion': None}]
```

As you can see, the search does not reveal which user this to-do belongs to.

120

9.1 Introducing Relations

To address the issue described in the previous paragraph, we can add a `user` property to the `Todo` type. However, manually maintaining its accuracy could be cumbersome. ThingsDB offers a more elegant solution: *relations*.

Let's add a relation between the `Todo` and `User` types using the following code:

```
(//todos)>
// 1. Add the optional user property to Todo
mod_type("Todo", "add", "user", "User?");

// 2. Create the bidirectional relation
mod_type("Todo", "rel", "user", "todos");
null
```

Let's break down the key steps:

1. We add a new property named user to the `Todo` type. This property holds a reference to a `User` object, but here is the crucial part: it is nillable (indicated by the question mark `?`). Why is this important? Imagine we remove a `Todo` from a user's set. With a nillable property, ThingsDB can effortlessly set the `user` property to `nil`, reflecting the broken relationship. Without nillable properties, such changes would lead to errors or inconsistencies.
2. Now comes the magic: creating a bidirectional relationship between `Todo` and `User`. We do this by specifying the `user` property on the `Todo` side and the `todos` property on the `User` side. This tells ThingsDB how they're connected. As a result, you can easily access a user's to-do's through their `todos` property and identify the owner of a specific `Todo` using its `user` property.

> 💡 This relationship can be initiated from either the **Todo** or **User** type, both would yield the same result.

Now that we've built the bridge between `Todo` and `User`, let's see if it holds. Let's use our trusty `search_todos.py` script:

```
$ python search_todos.py
Search for: book
[{
    'id': 2,
    'body': 'Read a book',
    'creation': '2024-02-07T14:47:49Z',
    'completion': '2024-02-07T14:48:06Z',
    'user': {'#': 10}
}]
```

Look! ThingsDB automatically populated the `user` property for us. But the real magic happens when we modify this relationship. Imagine changing the `"Read a book"` to-do's owner from Alice to Bob. Thanks to the *relation*, this change would seamlessly ripple through the system. The to-do automatically gets removed from Alice's `todos` set and added to Bob's! This dynamic behavior ensures your data stays consistent and reflects real-world ownership of to-do's.

9.2 Exploring More Relations

Our to-do example showcased a *one-to-many* relationship between a single property `user` and a set todos. But ThingsDB's relational power doesn't stop there! Let's dive into the other types which are briefly described in Table 9.2 below.

Table 9.2 - Relationship Types

Relation	Definition	Description	Example
One-on-One	`T?` <-> `T?`	Type to Type	Passport and owner
One-to-Many	`T?` <-> `{T}`	Type to Set	User and their to-do's
Many-to-Many	`{T}` <-> `{T}`	Set to Set	Users and their friends

Let's take a quick break from our current topic and set up a dedicated space for exploring the different types of relationships in ThingsDB. Using the `/thingsdb` scope, we'll create a new collection called `"relations"` to use for upcoming examples.

Here's how we'll do it:

```
(//todos)> @t
(/t)> new_collection("relations");
"relations"
(/t)> @ //relations
(//relations)>
```

Now, with the `"relations"` collection ready and waiting, we can now examine the different types of relationships available in ThingsDB.

9.2.1 One-on-One Relationship

Imagine a `Passport` type linked to its owner through a `user` property. Each owner can have only one passport, and each passport belongs to exactly one

owner. This scenario represents a *one-on-one* relationship, denoted as `T?` <-> `T?` in [Table 9.2](#).

One-on-One relationships in ThingsDB offer flexibility beyond just connecting different types. You can create a relationship where one instance connects to exactly one other instance of the same type. This even allows for reflexive relationships, where an instance connects to itself!

Defining the type `Pair` to demonstrate this type of relationship:

```
(//relations)> new_type("Pair");
set_type("Pair", {other: "Pair?"});

// Create a relationship to the same property
mod_type("Pair", "rel", "other", "other");
null
```

Creating a `Pair` in action:

```
(//relations)> .p = Pair{};
.p.other= Pair{};
.p.other.other == .p;  // Returns true, confirming the link
true
```

Building on the idea of self-referencing relationships, let's dive into chain relationships. These are perfect for modeling parent-child scenarios where each parent has only one child, and vice versa. Imagine a chain of connected elements, where each element points to the next one in line.

Creating the `Chain` Type:

```
(//relations)> new_type("Chain");
set_type("Chain", {prev: "Chain?", next: "Chain?"});

// Create a relationship between "prev" and "next"
mod_type("Chain", "rel", "prev", "next");
null
```

Now, let's create some `Chain` instances and connect them:

```
(//relations)> .chain = Chain{};
.chain.next = Chain{};
.chain.next.next = Chain{};
.chain.next.next.prev == .chain.next;  // Returns true, confirming the link
true
```

See how each element's `next` property points to the next one in the chain, and the last element's `prev` points back to the second element!

9.2.2 Many-to-Many Relationship

Now, consider a social network scenario where users have friends. Adding Bob as a friend for Alice should automatically make Alice a friend of Bob. This *many-to-many* relationship connects two sets.

Let's create a `Person` type in the `//relations` scope to model this scenario:

```
(//relations)> new_type("Person");
set_type("Person", {friends: "{Person}"});

// Define the many-to-many relationship
mod_type("Person", "rel", "friends", "friends");
null
```

Now, let's see how Thelma and Louise become friends:

```
(//relations)> .Thelma = Person{}; .Louise = Person{};
.Thelma.friends.add(.Louise);  // Thelma adds Louise as a friend
.Louise.friends.has(.Thelma);  // Returns true, confirming their friendship
true
```

By adding Louise to Thelma's `friends` set, the relationship automatically applies to Louise's `friends` set as well, thanks to the defined relationship.

Like other relationship types, *many-to-many* relationships aren't limited to connecting instances of the same type. As long as both participating entities have sets defined with each other's type, you can create a bidirectional connection. In Table 9.2, we used the notation `{T}` <-> `{T}` to represent this flexibility.

We've explored different relationships in ThingsDB. Now, let's head back to the `//todos` collection to continue building our to-do app!

9.3 Creating Relations: A Deeper Look

Adding a relation in ThingsDB might seem like a simple task, but there's more to it than meets the eye.

> In ThingsDB, sets are deliberately chosen for the *"many"* side of relationships due to their unique strengths: they naturally avoid order-related issues when removing things (unlike lists), guarantee uniqueness (aligned with expected behavior) and offer optimized performance for managing relations.

Before officially establishing a relation, ThingsDB performs a thorough check to ensure everything is in order. Think of it like a safety net to prevent potential data conflicts.

These checks are crucial for maintaining data integrity and consistency. Imagine accidentally creating a relation that would lead to duplicate data or inconsistent information. By proactively identifying such issues, ThingsDB safeguards your data and prevents headaches down the line.

Now that we understand the importance of pre-checks, let's remember a key requirement for relations: *they only work with stored data*. This means that at least one of the entities involved in creating a relation (e.g. a `Todo` and its `User`) needs to have an ID assigned by ThingsDB.

Consider this code snippet that tries to create a relation without this requirement:

```
(//todos)>
t = Todo{};  // Todo without ID
u = User{};  // User without ID
u.todos.add(t);
TypeError: relations must be created using a property on a stored thing (a
thing with an Id)
```

In this example, we attempt to link a `Todo` and a `User` by adding the `Todo` object to the user's `todos` property. However, the `User` object lacks an ID, which is essential for establishing the relation. Remember from [Chapter 4.2](#) that things only get IDs when they are assigned to a collection.

To correctly create the relation, first assign the `User` to the collection before adding the `Todo`. This will grant them IDs, enabling ThingsDB to establish the desired connection.

> 💡 Storing a thing before creating a relation is a fundamental principle in ThingsDB. While it might seem like an extra step, it often aligns with the natural flow of data creation. Take the **add_todo** procedure, for example. While we did not explicitly mention the rule there, adding the **Todo** object to the **todos** set on the stored user is the correct way to establish the relation with a **User**.

9.4 Code Cleanup

Previously, adding to-do's was limited to a specific user. Now, we'll prompt the user for a name during the process. This information will be passed to the `add_todo` procedure, ensuring the to-do gets linked to the correct user.

Here's the updated `add_todo.py` script:

```python
async def work(client: Client):
    name = input("Name: ")
    todo_body = input("To-do body: ")
    todo_id = await client.run('add_todo', name=name, body=todo_body)
    print(todo_id);
```

With the `add_todo.py` back on track, it's time to test your knowledge with the quiz! Then, in the next chapter, we'll master controlling ThingsDB responses.

9.5 Quiz - Challenge Your Understanding

1. Which of the following are valid relation types which can be handled by ThingsDB?

    a. `T? <-> T?`
    b. `T? <-> [T]`
    c. `{T} <-> {T}`
    d. `T? <-> {T}`
    e. `T <-> {T}`

2. Can you identify which statement(s) establish a relation between type `X` on property `one` with definition `"Y?"` and type `Y` on a property called `many` using definition `"{X}"`?

    a. `mod_type("X", "rel", "one", "many");`
    b. `mod_type("X", "rel", "many", "one");`
    c. `mod_type("Y", "rel", "one", "many");`
    d. `mod_type("Y", "rel", "many", "one");`

3. Imagine we're building a to-do app with users and their tasks. We have a `Todo` type and a `User` type connected by a *one-to-many* relation on the `user` and `todos` properties. Now, consider the following code:

    ```
    Todo{user: User(10)};
    ```

    This code tries to create a new `Todo` object and assign a `User` object with ID 10 to its `user` property. The `User` with ID 10 exists, so why would this code fail? Can you identify the issue and explain why it would not work in ThingsDB?

4. Remember our to-do app with the `Todo` and `User` types linked by a *one-to-many* relation? Now, consider the following code snippet:

    ```
    todo = Todo(2);
    user = User(10);
    assert(todo.user == user);
    user.todos -= set(todo);
    todo.user; // ??? what will it be?
    ```

    Given that this code runs without errors, can you predict what value will be assigned to `todo.user` after the final line?

9.5.1 Quiz - Answers

1. Valid relation types are "a", "c" and "d".
   a. `T? <-> T?` *(One-on-One relation)*
   b. `T? <-> [T]` *(!!! Invalid - A relation with a "list" is not possible)*
   c. `{T} <-> {T}` *(Many-to-Many relation)*
   d. `T? <-> {T}` *(One-to-Many relation)*
   e. `T <-> {T}` *(!!! Invalid - The property must be marked nillable)*

2. The correct statements are "a" and "d". When establishing a relationship, ThingsDB relies on the third argument to locate the other type based on the provided property. Subsequently, the fourth argument serves as a pointer, directing ThingsDB to the target property on that type where the relationship will be formed.

3. In ThingsDB, establishing a relation between two types relies on stored data and their associated IDs. In your code, the `User` object with ID 10 exists, but you are trying to assign it to a new `Todo` without an ID. The solution is to use the stored user:

```
User(10).todos.add(Todo{});
```

   When working with relations in ThingsDB, always keep in mind the importance of stored data and IDs. Utilize existing objects with IDs to establish connections.

4. The final value of `todo.user` in the given code snippet will be `nil`. The key lies in the line `user.todos -= set(todo)`. Here, we're using set subtraction to remove the `Todo` object from the `user.todos` set. ThingsDB then automatically updates the `user` property of the `Todo` object to `nil`, reflecting the "broken" relationship. This behavior ensures consistency and eliminates the possibility of orphaned references.

Chapter 10 - Control Responses

Remember how `search_todos.py` returned a `Todo` instance? While it is functional, imagine customizing the response to something like this:

```
{
    "id": 2,
    "body": "Read a book",
    "done": true,
    "user": {
        "name": "Alice",
    }
}
```

This example showcases the desired structure:

- Essential keys like `"id"` and `"body"` remain unchanged.
- The `completion` property is replaced with a `"done"` flag.
- User information is simplified, including only the `"name"`.

10.1 The Power of Wrap-Only Types

While code manipulation can achieve this, ThingsDB offers a more efficient approach: Wrap-Only types. These types act as templates defining how to present existing data. They can't be directly created, but they can "wrap" other types, transforming their output.

Let's dive into defining our wrap-only type for to-do's.

```
(//todos)> new_type("_Todo", true);
"_Todo"
```

The name we choose for our wrap-only type plays a role. While not strictly required, many developers adopt the convention of starting it with an underscore (`_Todo` in our case). This helps visually distinguish wrap-only types from regular ones at a glance.

The second boolean argument (`true`) passed to `new_type()` serves as a flag, enabling the "wrap-only" mode for the defined type (`_Todo`). This signifies that direct instantiation of instances is prohibited. Instead, its functionality lies in dynamically transforming existing data structures based on the properties and computed values specified within its definition.

Having declared our `_Todo` wrap-only type, we proceed to define its structure and behavior using `set_type()`:

```
(//todos)> set_type("_Todo", {
    id: "#",
    body: "any",
    done: |this| is_datetime(this.completion),
});
null
```

Let's dissect the key aspects of our wrap-only type configuration:

- `id: "#"`

  This familiar selection directly includes the unique identifier (id) of the wrapped `Todo` object in the response.

- `body: "any"`

  This indicates we want to incorporate the entire text content of the `body` property, regardless of its original data type. To include a property in the wrapped data, its definition in the wrap-only type must be *equal or less restrictive* compared to the original. In this case, `"any"` covers any possible data type for the `body`.

- `done: |this| is_datetime(this.completion)`

  This is where things get interesting! We're utilizing a *computed property*. This method dynamically examines the `completion` property of the wrapped `Todo`. If it finds a datetime value present, it sets the `done` flag to `true`, otherwise it remains `false`. This demonstrates the power of computed properties to not only select specific data but also transform it based on defined logic, adding new insights to the wrapped representation.

Fundamental to the design of wrap-only types is the restriction on direct instance creation. Let's confirm this behavior:

```
(//todos)> _Todo{};
TypeError: type `_Todo` has wrap-only mode enabled
```

As expected, attempting to directly create an instance of the `_Todo` type throws an error.

To leverage the power of `_Todo`, we employ the `wrap()` method, available on thing objects. Let's see how it works:

```
(//todos)> Todo(2).wrap("_Todo");
{
    "body": "Read a book",
    "done": true,
    "id": 2
}
```

We've achieved a significant transformation. The wrapped `Todo` now exhibits the desired structure, including the calculated `done` property. However, one crucial element remains missing - the `user` information.

To incorporate `user` information, let's create another wrap-only type specifically for this purpose:

```
(//todos)> new_type("_TodoUser", true, true);
"_TodoUser"
```

The first `true` argument activates the familiar wrap-only mode functionality. However, the inclusion of the second `true` argument introduces a privacy feature. By specifying this additional flag, we instruct ThingsDB to exclude the ID property within the response.

We'll keep the `_TodoUser` type simple, focusing on the essential `name` property:

```
(//todos)> set_type("_TodoUser", {name: "str"});
null
```

Now, let's merge `_TodoUser` into our `_Todo` type:

```
(//todos)> mod_type("_Todo", "add", "user", "&_TodoUser?");
null
```

Let's delve into the meaning of the `"&_TodoUser?"` definition assigned to the `user` property.

The `?` symbol is crucial as it preserves the original nillability `"User?"` of the `user` property. If we omitted the `?`, it would remove the nillability flag, making the definition stricter and preventing the `user` property to be included in a response.

The `&` symbol at the beginning of the `"&_TodoUser?"` definition is a prefix flag in ThingsDB. Several such flags exist, each influencing how properties are returned in a query response. Remember the optional `deep` value in the `return` statement? It is actually rarely needed to define this `deep` value. The `&` flag acts similarly, instructing ThingsDB to directly include the specified property in the response without manual deep level specification. It effectively maintains the same deep level as the parent property.

Table 10.1 - Summarizing prefix flags

Flag	Description
`"&"`	Use parent's deep level
`"+"`	Force the maximum deep level of 126 (127 in parent)
`"-"`	Force deep level of 0 (1 in parent)
`"^"`	Apply `NO_IDS` return flag
`"*"`	Return enumerator names instead of values

> While the ^ flag effectively hides IDs, you might wonder if it is equivalent to setting the hide-ID flag for the `_TodoUser` type like we did. Both achieve the same result here, but with a key difference. The ^ flag applies the `NO_IDS` restriction recursively, anonymizing IDs throughout nested user data. In contrast, hide-ID only affects the `_TodoUser` type itself, potentially exposing IDs in nested objects.

Now that we've incorporated the `user` property, let's witness the magic!

```
(//todos)> Todo(2).wrap("_Todo");
{
    "id": 2,
    "body": "Read a book",
    "done": true,
    "user": {
        "name": "Alice"
    }
}
```

This output aligns perfectly with our desired format!

No need for a `return` statement or manual deep level specification! Our wrap-only types seamlessly weave together to-do information and the user's name, presenting a cohesive response in the desired format.

10.2 Update Search Todos

Now that we have a robust wrap-only type for to-do's, let's adapt the `search_todos` procedure to leverage its functionality.

Initially, we could employ the familiar `map()` method to wrap each filtered `Todo` instance:

132

```
.users.values().reduce(|total, user| {
    total |= user.todos.filter(|todo| todo.body.contains(needle));
    total;
}, set()).map(|todo| todo.wrap("_Todo"));
```

However, ThingsDB offers a dedicated and more efficient option specifically designed for this common wrapping scenario: the `map_wrap()` method.

Update the the `search_todos` procedure by incorporating the `map_wrap()` method:

```
(//todos)> mod_procedure("search_todos", |needle| {
    "Search for to-do's";
    .users.values().reduce(|total, user| {
        total |= user.todos.filter(|todo| todo.body.contains(needle));
        total;
    }, set()).map_wrap("_Todo");  // map_wrap() for efficient conversion
});
null
```

To confirm the successful integration of the `Todo` type, let's re-run the `search_todos.py` script:

```
$ python search_todos.py
Search for: book
[{'id': 2, 'body': 'Read a book', 'user': {'name': 'Alice'}, 'done': True}]
```

As expected, the output now reflects the desired structure, incorporating the to-do details and user information within the expected format.

10.3 Wrap Every-Thing

In this chapter, we crafted the `_Todo` wrap-only type, but did you notice a key detail? We never explicitly specified the target type for wrapping. That's because wrap-only types possess remarkable versatility! Any thing, any instance, can be wrapped using `_Todo`, regardless of its original definition.

Let's observe what happens when we wrap an empty thing:

```
(//todos)> {}.wrap("_Todo");
{
    "done": "thing `#0` has no property `completion`"
}
```

While not ideal, the result is understandable. The computed `done` property returns an error message as the wrapped thing lacks the required `completion` property.

133

Now, let's try wrapping an object with some properties:

```
(//todos)> {completion: nil, body: 123, smile: true}.wrap("_Todo");
{
    "body": 123,
    "done": false
}
```

As expected, the `done` property is calculated correctly based on the provided `completion` property. Remember, the `body` property in `_Todo` is defined as `"any"`, allowing it to accept various data types like integers here. Since `smile` is not defined in `_Todo`, it is simply ignored.

> 💡 While ThingsDB can wrap even plain things, it is worth noting that wrapping typed things offers a significant performance advantage. This is due to an internal optimization mechanism: The first time a typed thing is wrapped, ThingsDB creates a mapping table. This table efficiently stores how each property in the typed object maps to the corresponding property in the wrap-only type. For subsequent wraps of the same type, ThingsDB can leverage this pre-calculated table, drastically reducing computation overhead. This optimization excludes computed properties, which inherently require dynamic evaluation.

10.4 Modifying Wrap-Only and Hide-ID Flags

In this chapter, we've explored creating wrap-only types and defining hide-ID behavior. While we covered the standard approach, you might be wondering what happens if you forget to set these flags initially?

The `mod_type()` function provides two key actions for modifying these configurations after creation: `"wpo"` for managing wrap-only mode and `"hid"` for controlling hide-ID behavior.

The following code demonstrates how to apply these actions in practice:

```
(//todos)> mod_type("_Todo", "wpo", true);
null
(//todos)> mod_type("_TodoUser", "hid", true);
null
```

Enabling wrap-only mode for an existing type is restricted to when no instances of that type currently exist. Attempting to activate it with existing instances will result in an error.

We trust you've gained a solid grasp of the transformative potential of wrap-only types. By leveraging them effectively, you can streamline your code,

eliminating custom deep levels within `return` statements and achieving your desired data structures effortlessly.

We wish you the best of luck with the quiz! In the next chapter, we'll delve into the exciting world of implementing flags and enumerators, further expanding your data manipulation capabilities within ThingsDB.

10.5 Quiz - Challenge Your Understanding

1. Both type `A` and `B` have an `email` property. On type `A` the property is defined as `"email?"`. In which scenario will the `email` property appear in the response when an instance of type `A` is wrapped with type `B`?

   Choose the correct definition(s) for the `email` property in type `B`:

   a. `"any"`
   b. `"str"`
   c. `"str?`
   d. `"email"`

2. Consider the following code snippet:
   ```
   new_type("T", true, true);
   ```

   What do the two boolean arguments in the provided `new_type()` function call signify?

   a. The first `true` activates wrap-only mode, the second activates the hide-ID flag.
   b. The first `true` activates wrap-only mode, the second activates the show-ID flag.
   c. The first `true` activates the hide-ID flag, the second activates wrap-only mode.
   d. The first `true` activates the show-ID flag, the second activates wrap-only mode.

3. Wrap-only types empower you to define properties or methods. What purpose do methods serve in this context?

4. Imagine you have a variable `people` holding a list of things of type `Person`. Your task is to return this list with each `Person` instance transformed and represented using the `_Person` wrap-only type. Write the most concise and efficient statement possible to achieve this transformation.

5. Imagine you're creating a wrap-only type to transform existing instances of the `Person` type. Which of the following names are valid for your new wrap-only type?

   a. `Beer`
   b. `_Person`
   c. `person`
   d. `_PersonCompact`
   e. *all of the above*

6. What happens when a *computed property* encounters an error during its calculation? Choose the correct answer:

a. It throws an exception that needs to be handled elsewhere.
b. It silently returns a default value like `nil`.
c. It adds the property with a string value containing the error message.
d. It continues with the calculation, potentially leading to unexpected results.

10.5.1 Quiz - Answers

1. Both `"str?"` and `"any"` can inherit the `email` property due to their less restrictive nature compared to the original `"email?"` definition. This follows the principle of compatibility where definitions should be either the same or less restrictive for properties to appear. While `"str"` enforces string type, it loses the nillability aspect from the original. On the other hand, `"any"` allows any type, including `nil`, effectively inheriting the nillability due to its broadness.

2. Answer "a". The first `true` enables wrap-only mode, and the second `true` activates the hide-ID flag. Both arguments default to `false` if omitted.

3. Methods in wrap-only types define *computed properties*: properties whose values are dynamically calculated on response creation.

4. The most efficient way to achieve this transformation is using the built-in `map_wrap()` method. Simply calling `people.map_wrap("_Person");` will wrap each `Person` object in the list with the `_Person` type. While other methods like `map()` can also be used, `map_wrap()` offers the most concise and efficient solution in this case.

5. All the proposed names technically work. However, best practices favor naming conventions for clarity and consistency. Options like `_Person` or the more descriptive name `_PersonCompact` using the underscore prefix instantly reveal the wrap-only purpose, enhancing code understanding and avoiding confusion.

6. The correct answer is "c". ThingsDB handles errors in computed properties gracefully by incorporating the error message itself as the property's string value within the result. This approach ensures visibility of any issues that may arise during calculation, promoting transparency and facilitating troubleshooting.

Chapter 11 - Flags, Enumerators and Regex

This chapter equips you with more powerful tools to manage data efficiently in ThingsDB: bitwise flags, regular expressions and enumerators. Let's jump right in with enumerators!

Consider a `severity` property for your to-do items. An enumerator allows you to establish a finite list of valid options, such as "Low", "Medium", and "High". This enforces data integrity by ensuring only permitted values are assigned, preventing inconsistencies or invalid entries

11.1 Crafting Your First Enumerator

Unlike conventional data types, enumerators in ThingsDB are created directly using the `set_enum()` function. Here's the code to define the `Severity` enumerator for our to-do application:

```
(//todos)> set_enum("Severity", {
    Low: 0,
    Medium: 1,
    High: 2,

    // Optional method for customized string representation
    str: |this| `{this.name()} ({this.value()})`,
});
null
```

Key Technical Properties:

- **Supported Data Types:** Enumerators can accommodate integers, floats, strings, bytes, and things. Other types like sets, lists, and tasks are not supported.
- **Unique Value Requirement:** Each value within an enumerator must be distinct.
- **Mutable Object Handling:** Things, being mutable, can have identical content within an enumerator as long as they reference different objects.
- **Method Inclusion:** Enumerators can also house methods, like the `str()` method in our example, used for custom string formatting.
- **Member Rules:** At least one member is mandatory, and all members must share the same data type (e.g., all integers). Mixing types like integers and floats is not allowed.

11.1.1 Setting the Default Member of an Enumerator

Like other ThingsDB elements, enumerators possess default members. To avoid unexpected behavior, it is highly recommended to explicitly define these defaults using the `mod_enum()` function. Here's how to set `Medium` as the default member for the `Severity` enumerator:

```
(//todos)> mod_enum("Severity", "def", "Medium");
null
```

To verify that the default member is correctly configured, simply call the `Severity` enumerator without providing a specific value to get the default member:

```
(//todos)> Severity();
1
```

As demonstrated, when a member is returned in a response, its underlying value is displayed by default. In this case, `1` (representing `Medium`) is returned.

11.1.2 Working with Enumerator Methods

Each enumerator member in ThingsDB offers two built-in methods: `name()` and `value()`. These methods provide programmatic access to the enumerator's name and value, respectively. Additionally, any custom methods defined within the enumerator become available to its members.

Let's delve into practical examples using the `Severity` enumerator we defined earlier:

```
(//todos)> Severity().name();
"Medium"
(//todos)> Severity().value();
1
```

As expected, `name()` returns the string `"Medium"`, while `value()` retrieves the corresponding value, `1`.

Remember the optional `str()` method we defined for formatting? We can access it through a member:

```
(//todos)> Severity().str();
"Medium (1)"
```

This demonstrates how custom methods enhance the functionality of enumerator members.

11.1.3 Retrieving Enumerator Members

Need a member with a specific value? Simply pass the value directly to the enumerator constructor:

```
(//todos)> Severity(2).name();  // Value 2 for High
"High"
```

This straightforward approach is ideal when the value is readily available.

Know the enumerator name beforehand? Use static access with curly braces:

```
(//todos)> Severity{High}.str();
"High (2)"
```

This method is useful when the name is known and fixed.

Working with dynamic names stored in variables? Employ a closure within curly braces:

```
(//todos)> name = "High"; Severity{||name};
2
```

While the example might seem unusual, the name variable is simply for demonstration. In practice, you might receive it as an input argument or retrieve it dynamically. This method ensures security against code injection vulnerabilities, as described in Chapter 6.2.3.

11.1.4 Enumerator Validation

When you attempt to access an enumerator using an invalid value or name, it raises an error instead of silently continuing. This prevents unexpected behavior and ensures data accuracy.

Here's what happens when you try to use non-existent values or names:

```
(//todos)> Severity(9);
LookupError: enum `Severity` has no member with value 9
(//todos)> Severity{Insane};
LookupError: enum `Severity` has no member `Insane`
(//todos)> Severity{||"NotAtAll"};
LookupError: enum `Severity` has no member `NotAtAll`
```

As you can see, ThingsDB throws a `lookup_err` in all cases, clearly indicating the invalid input.

11.2 Enumerator Information

ThingsDB offers `enums_info()` to explore all defined enumerators within a collection, returning a list of information objects. Each object encapsulates details like its name, members, and methods.

For a specific enumerator, use `enum_info()`, which retrieves a single information object. Remember that information methods like `enum_info()` return "mpdata" requiring further processing. Utilize the `load()` method to convert it into a usable structure where you can extract individual properties.

For instance, verifying the default member of the `Severity` enumerator involves:

```
(//todos)> enum_info("Severity").load().default;
"Medium"
```

This code retrieves the `Severity` information object, loads it, and extracts the `default` property, revealing the default member: `"Medium"`.

11.2.1 Enumerator Members

Accessing an enumerator's member can be valuable for integrating with other platforms or systems. Ideally, we want these members readily available as a single "thing" for simpler handling.

While `enum_info()` exposes members, processing them requires code to convert the returned list of tuples into a desired format. Here's an example using `reduce()`:

```
(//todos)> enum_info("Severity").load().members.reduce(
    |t, m| {t.set(m[0], m[1]); t;},
    {}
);
{"Low": 0, "Medium": 1, "High": 2}
```

As you can see, this approach involves several steps and can be cumbersome.

Fortunately, ThingsDB provides a built-in solution: `enum_map()`. This function directly returns the enumerator members as a convenient "thing":

```
(//todos)> enum_map("Severity");
{"Low": 0, "Medium": 1, "High": 2}
```

11.3 Modifying Enumerators

Need to modify an existing enumerator after its creation? Don't worry, the `mod_enum()` function comes to the rescue! We previously saw it in action for setting the default member. But its power extends beyond that. It offers a versatile suite of actions to tailor your enumerators to your evolving needs.

Here's a quick reference table summarizing the available actions:

Table 11.3 - Summary of mod_enum() actions

Action	Description
`"add"`	Adds a new member or method to the enumerator
`"def"`	Sets the default member for the enumerator
`"del"`	Deletes a member or method from the enumerator
`"mod"`	Modifies a member value or method closure
`"ren"`	Renames a member or method

11.4 Implementation and Other Enumerator Solutions

While we've created a custom `Severity` enumerator, ThingsDB offers other built-in solutions for common purposes. Let's briefly compare these options before proceeding.

11.4.1 Revisiting Range Definitions

Recall that ThingsDB empowers you with precise control over integer values using range definitions. Let's revisit how to implement this for the `severity` property within the `Todo` type:

```
(//todos)> mod_type("Todo", "add", "severity", "int<0:2:1>");
null
```

This code snippet concisely bolsters data integrity within the `Todo` type by introducing the `severity` property. This new property restricts values to integers within the strict `0-2` range, effectively preventing invalid input. Furthermore, a default value of `1` is established, ensuring a meaningful

143

starting point for both existing and new instances. Notably, this constraint is retroactively applied to all existing `Todo` instances, guaranteeing data integrity throughout the entire dataset.

Let's demonstrate the default value and error handling of the `severity` property:

Creating a `Todo` instance without specifying `severity` assigns the default value:

```
(//todos)> Todo{}.severity;
1
```

Attempting to create a `Todo` with an invalid severity triggers an error:

```
(//todos)> Todo{severity: 99};
ValueError: mismatch in type `Todo`; property `severity` requires an
integer value between 0 and 2 (both inclusive)
```

11.4.2 Regular Expressions using Regex

While enumerators offer a structured and type-safe approach to defining value sets, you can also achieve similar results using regular expressions. ThingsDB supports regular expressions through the "regex" data type.

Here's how to write a regular expression in ThingsDB:

```
/pattern/flags
```

Regular expressions in ThingsDB go beyond just validating strings. They act as powerful tools for various string manipulation tasks. By using string methods like `replace()` and `split()`, you can transform and extract portions of strings based on defined patterns.

A key method for pattern matching is the regex method `test()`, which returns `true` if a given string matches the pattern, and `false` otherwise.

Let's see the `test()` method in action:

```
(//todos)> /^(Low|Medium|High)$/.test("Low");
true
(//todos)> /^(Low|Medium|High)$/.test("MEDIUM");
false
(//todos)> /^(Low|Medium|High)$/i.test("LOW"); // Case-insensitive matching
true
```

While crafting regular expressions is beyond this book's scope, let's briefly break down the pattern used in the example:

- `^`: Matches the beginning of the string.
- `(Low|Medium|High)`: Matches one of the listed options (`"Low"`, `"Medium"`, or `"High"`).
- `$`: Matches the end of the string.
- `i` flag: Enables case-insensitive matching.

Leveraging the understanding of regular expressions, let's define the `severity` property using a pattern-based approach. Remember, definitions in ThingsDB are always strings. We encapsulate the regular expression within quotes and specify a default value. Crucially, this default value must strictly adhere to the defined pattern for successful definition acceptance.

```
(//todos)> mod_type(
    "Todo",
    "mod",
    "severity",
    "/^(Low|Medium|High)$/<Medium>",
    |this| ["Low", "Medium", "High"][this.severity],
);
null
```

Key Points:

1. **Regex Definition:** The code snippet utilizes a regex string (`/^(Low|Medium|High)$/`) to constrain valid severity values to `"Low"`, `"Medium"`, or `"High"`.
2. **Default Value:** The `<Medium>` suffix sets the default value to `"Medium"`. Remember, default values must adhere to the defined pattern for acceptance.
3. **Migration Closure:** This scenario necessitates a migration closure due to the incompatibility of the new regex with the previous integer range.
4. **Closure Functionality:** The provided closure maps existing integer severity values (`0`, `1`, `2`) to their corresponding string equivalents (`"Low"`, `"Medium"`, `"High"`).

Let's validate the default value and constraint for the string-based `severity` property:

Creating a new `Todo` without specifying `severity` assigns the default value:

```
(//todos)> Todo{}.severity;
"Medium"
```

Attempting to create a `Todo` with an invalid `severity` triggers an error:

145

```
(//todos)> Todo{severity: "Unknown"};
ValueError: mismatch in type `Todo`; property `severity` has a requirement
to match pattern /^(Low|Medium|High)$/
```

11.4.3 Transitioning to the Enumerator

Having explored both integer range and string-based solutions, let's migrate the `severity` property to leverage our custom `Severity` enumerator:

```
(//todos)> mod_type(
    "Todo",
    "mod",
    "severity",
    "Severity",
    |this| Severity{||this.severity},
);
"Medium"
```

As before, a migration closure is necessary due to the change in data representation. This closure translates existing string values (`"Low"`, `"Medium"`, `"High"`) to their corresponding enumerator members (`Severity{Low}`, `Severity{Medium}`, `Severity{High}`).

To confirm successful migration, let's create a new `Todo` with default severity:

```
(//todos)> Todo{}.severity.str();
"Medium (1)"
```

Excellent! We can now leverage the `str()` method we defined, indicating that the `severity` property is now strictly bound to the enumerator's members.

11.4.4 Wrap-Only Types with Enumerators

We can further elevate the `_Todo` wrap-only type by incorporating the `severity` property. Additionally, ThingsDB offers the optional `*` prefix flag to specify returning the enumerator member's *name* instead of its *value*. Let's demonstrate this in action:

```
(//todos)> mod_type("_Todo", "add", "severity", "*Severity");
null
```

Now, let's test this modification on an empty `Todo`:

```
(//todos)> Todo{}.wrap("_Todo");
{
    "body": "-",
    "done": false,
    "severity": "Medium",
    "user": null
}
```

As you can see, the `severity` property now displays the member name `"Medium"` instead of its numerical value (`1`).

11.5 Understanding Flags

Before we proceed with the quiz, let's explore a use case for flags in ThingsDB.

> 💡 Flags are a data structure used internally by ThingsDB and many other software to efficiently store access rights and other bitwise information. Recall how we granted the RUN and CHANGE access flags to our `"py"` user. These flags represent individual bits, specifically `16` (`10000`) for RUN and `2` (`00010`) for CHANGE. By granting both, we effectively stored the value `18` (`10010`) within a single integer property. Leveraging 64-bit integers, ThingsDB empowers you to store up to 64 flags within a single property, promoting data density and efficiency.

11.5.1 Using Enumerators to Store Flags

Before utilizing flags, we need to establish them within your ThingsDB project. Enumerators provide a convenient and safe approach for this task.

While it may seem unrelated to our to-do app, let's imagine Alice and Bob, with no more tasks remaining, decide to play cards. We'll create a `Suits` enumerator to represent the four card suits:

```
(//todos)> set_enum("Suits", {
    Spades:   1<<0,  // 0b0001 (1)
    Clubs:    1<<1,  // 0b0010 (2)
    Diamonds: 1<<2,  // 0b0100 (4)
    Hearts:   1<<3,  // 0b1000 (8)
});
null
```

We've used the bitwise left shift operator (`<<`) to assign values to each suit. While not mandatory, this approach helps prevent errors during flag definition, as it leverages the inherent binary representation of integers. Using simple

numbers (`1`, `2`, `4`, `8`) would also work, but bit shifting offers additional safety and clarity.

11.5.2 Working with Flags

Now that our `Suits` enumerator is defined, let's delve into how to use these flags effectively in your code.

> While this section delves into using flags in ThingsDB, the underlying bitwise operations are similar to those found in most programming languages. If you are comfortable with bitwise operators like `<<` and `&`, you may find this section reinforces general concepts rather than ThingsDB-specific details.

Bitwise OR (`|`) Operator: Combines multiple flags into a single variable:

```
(//todos)> // Sets both Spades and Hearts flags
my_suits = Suits{Spades}.value() | Suits{Hearts}.value();
9
```

Bitwise OR-Equals (`|=`) Operator: Adds a flag to an existing value:

```
(//todos)>
my_suits = Suits{Spades}.value();  // Initially set Spades flag
my_suits |= Suits{Hearts}.value(); // Adds Hearts flag
9
```

Bitwise AND (`&`) Operator: Checks if a specific flag is present:

```
(//todos)> my_suits = Suits{Spades}.value() | Suits{Hearts}.value();
// Test for Hearts flag
assert(bool(my_suits & Suits{Hearts}.value()) == true);

// Test for Diamonds flag
assert(bool(my_suits & Suits{Diamonds}.value()) == false);
null
```

Bitwise NOT (`~`) Operator: Inverts all bits in a value, effectively removing a specific flag when combined with bitwise AND:

```
(//todos)> my_suits = Suits{Spades}.value() | Suits{Hearts}.value();
my_suits &= ~Suits{Hearts}.value();  // Unsets the Hearts flag
1
```

Bitwise XOR (`^`) Operator: Flips the bits of a specified flag, switching its state on or off:

148

```
(//todos)> my_suits = Suits{Spades}.value() | Suits{Hearts}.value();
my_suits ^= Suits{Hearts}.value();   // Toggles the Hearts flag
my_suits ^= Suits{Diamonds}.value(); // Toggles the Diamonds flag
5
```

Flags offer a powerful and compact way to store multiple boolean values within a single integer, enhancing data efficiency and representation. This can be particularly advantageous for managing access control, permissions, and various configuration settings.

> When integrating ThingsDB with other languages, remember the `enum_map()` function. This function helps effortlessly translate between ThingsDB enumerators and their corresponding values in different languages, streamlining data exchange and interoperability.

Best of luck with the quiz! In the next chapter, we embark on the exciting journey of events and rooms in ThingsDB, where you'll explore powerful mechanisms for real-time communication and data interactions within your applications.

11.6 Quiz - Challenge Your Understanding

1. Select all statements that are true about enumerators:
   a. Members of an enumerator must be of the same type.
   b. An enumerator is created using the `new_enum()` function.
   c. Members must have unique names, and their values must also be unique.
   d. Enumerators are immutable, meaning you cannot modify members or methods after creation.

2. While creating an enumerator, why is it recommended to use `mod_enum()` at least once?

3. Imagine you have a variable `x` containing the name of a member within the `Colors` enumerator. How would you retrieve the corresponding member using that variable?

4. What built-in methods are available on every member of an enumerator?

5. What does the `*` prefix flag signify when used in a type property definition for an enumerator? Explain its purpose and impact on data representation.

6. Write a statement using a regular expression that checks if the string stored in the variable `animal` matches either `"Dog"` or `"Cat"`, regardless of case sensitivity.

7. Which code snippet accurately initializes the `Colors` enumerator with `R`, `G`, and `B` members so they can be independently used as distinct flags for bitwise operations?
   a. `set_enum("Colors", {R: 0, G: 1, : 2});`
   b. `set_enum("Colors", {R: "#FF0000", G: "#00FF00", B: "#0000FF"});`
   c. `set_enum("Colors", {R: 1, G: 2, : 3});`
   d. `set_enum("Colors", {R: 1<<0, G: 1<<1, : 1<<2});`

11.6.1 Quiz - Answers

1. The correct choices are "a" and "c". Enumerators enforce strict data type consistency for their members and require unique names and values. While `new_enum()` might seem fitting, remember that ThingsDB uses `set_enum()` to create enumerators. Enumerators are mutable, allowing you to add, rename, delete, and change members and methods using the `mod_enum()` function.

2. Setting a default value using `mod_enum()` is generally recommended for enumerators, particularly those used within a type to influence its behavior. While less crucial for enumerators solely used for storing values like flags, defining a default value through `mod_enum()` provides additional control and clarity in various use cases.

3. To dynamically retrieve the Colors member based on the name stored in x, you can use a closure expression enclosed in curly braces:
    ```
    Colors{||x};
    ```

4. The two basic methods accessible to every enumerator member are:

    - `name()`: This method returns the string name of the member as defined in the enumerator declaration.
    - `value()`: This method returns the value associated with the member.

    Remember that enumerators themselves can have additional methods defined beyond these two, depending on how they were created using `set_enum()` or `mod_enum()`.

5. The `*` prefix flag, when used in a type property definition for an enumerator, influences the value returned when the instance is wrapped for the response. It instructs the system to return the *name* of the corresponding enumerator member instead of its *value*.

6. The following regular expression, combined with the `test()` method, can be used to check if the variable `animal` matches either `"Dog"` or `"Cat"` (case-insensitive):
    ```
    /^(dog|cat)$/i.test(animal);
    ```

7. The correct answer is "d":
    ```
    set_enum("Colors", {R: 1<<0, G: 1<<1, : 1<<2});
    ```

    This option uses the bitwise left shift operator (`<<`) to assign distinct binary values to each member. Answer "a" and "c" are using consecutive integer values (0, 1, 2) and (1, 2, 3) which do not provide distinct binary

representations for bitwise operations. Answer "b" is using string values for their members, which cannot be used for bitwise operations.

Chapter 12 - Real-Time Data Updates with Events and Rooms

Imagine Alice and Bob want to build a live dashboard displaying the number of pending to-do's for all users in their to-do application. However, simply polling the collection every few seconds is inefficient and creates unnecessary network traffic.

Enter the world of *Events* and *Rooms* in ThingsDB!

> Remember from [Chapter 4.4](#) how we used `log()` to emit warning events to the client? Well, events are a much broader concept in ThingsDB, acting as messengers carrying information triggered by various actions. Besides warnings, ThingsDB, for example, also generates events for node status changes.

12.1 Rooms

This chapter focuses on a specific type of event: those emitted to *rooms*.

Think of a room as a lightweight, dedicated communication channel within your collection. Each room receives a unique ID when it is associated with the collection, similar to individual things.

Creating a room is simple:

```
(//todos)> room();
"room:nil"
```

You'll see `"room:nil"` because no ID has been assigned yet. Assigning the room to the collection grants it an ID, just like how it works with things.

Since we converted the root of the collection to type `Root` back in [Chapter 7.2](#), modifying its structure requires the `mod_type()` function. We'll add a property named `dashboard` of type room to the `Root` type.

```
(//todos)> mod_type("Root", "add", "dashboard", "room");
null
```

> 💡 While assigning a separate room to each `Todo` might seem intuitive, it would not be efficient for a comprehensive dashboard that requires an overall view.

Let's verify if the room was created by accessing the `dashboard` property:

```
(//todos)> .dashboard;
"room:12"
```

You should see a unique room ID like `"room:12"`, indicating a successfully created room associated with the collection.

We can retrieve the plain ID of the dashboard room using the `id()` method. This method comes in handy when we later want to join the room for event listening.

```
(//todos)> .dashboard.id();
12
```

12.1.1 Sending Your First Message via Events

With the "dashboard" room in place, we're ready to emit our first event!

The process is simple: use the `emit()` method on your room object. The first argument is the *event name*, a string between 1 and 255 characters. You can optionally include additional arguments that will be sent along with the event.

Here's an example of sending a "chat-message" event:

```
(//todos)> .dashboard.emit("chat-message", "Who's listening???");
null
```

Remember, no one has joined the room yet. So, while the event was successfully emitted, no one received the "chat-message".

> 💡 While we emphasized the event name as the first argument for `emit()`, it can also accept an alternative *deep* value. Optionally, a second argument allows setting flags like `NO_IDS`. The actual event name and accompanying data always follow, regardless of these advanced options which are hardly ever required.
>
> For example: `room.emit(2, NO_IDS, "<EVENT_NAME>", …)`

12.2 Listen for Events

Just like before, we'll demonstrate using Python to create a listener for the "dashboard" room. Remember, the concepts are transferable to other languages like C#, Go, PHP or JavaScript/Node.js with minor adjustments.

Before proceeding, let's create a dedicated user for the dashboard. We can leverage the same `new_todo_user()` procedure we established in [Chapter 6.4](#):

```
(//todos)> @t
(@t)> wse(); new_todo_user("dashboard", JOIN | RUN | QUERY);
"14VdoACr4WydUi/PEi1MV6"
```

This grants the "dashboard" user:

- `JOIN`: To subscribe to the room and receive events.
- `QUERY`: To retrieve the room ID (needed for joining).
- `RUN`: To execute a future procedure for grabbing the initial state.

12.2.1 Creating Your Dashboard Listener

Save the following code as `dashboard.py`:

```
import asyncio
from thingsdb.client import Client
from thingsdb.room import Room, event

class Dashboard(Room):

    @event("chat-message")
    def on_chat_message(self, message):
        print(message)

async def main():
    client = Client()
    client.set_default_scope('//todos')

    room = Dashboard("""//ti
        .dashboard.id();  // This must return the room ID
    """)  # Fetch the dashboard room ID dynamically

    await client.connect('localhost')  # Replace with your address/port
    try:
        # Authenticate with the dashboards user's token
        await client.authenticate('14VdoACr4WydUi/PEi1MV6')
        await room.join(client)  # Join the dashboard room
        await asyncio.Future()  # Run forever
    finally:
        client.close()
        await client.wait_closed()

asyncio.run(main())
```

Open a terminal and run the script:

```
$ python dashboard.py
…
```

If successful, you'll see no immediate output. The listener is now actively waiting for events in the background.

With the listener running, let's send a test message from the `//todos` scope:

```
(//todos)> .dashboard.emit("chat-message", "Hello dear listeners");
null
```

If everything is set up correctly, you should see the following output in the terminal running your `dashboard.py` script:

```
$ python dashboard.py
Hello dear listeners
…
```

156

This confirms that your listener successfully received and printed the message!

12.3 Retrieving Initial Dashboard State

While sending chat messages is fun, our ultimate goal is to create a functional dashboard. Like many real-world applications, the dashboard needs to fetch an initial state, in this case, the current number of uncompleted tasks for each user.

First, we need a way to represent this information. Let's create a new wrap-only type named `_DashboardUser`:

```
(//todos)> new_type("_DashboardUser", true);
"_DashboardUser"
```

Next, we define the structure of `_DashboardUser`:

```
(//todos)> set_type("_DashboardUser", {
    id: "#",
    name: "str",
    count: |this| list(this.todos).count(|todo| is_nil(todo.completion)),
});
null
```

The `_DashboardUser` type defines three properties: `id` to identify a user, `name` for retrieving user names and `count` for calculating uncompleted to-do's using a computed property.

Finally, we create a procedure named `get_dashboard_state` that retrieves and formats the initial data:

```
(//todos)> new_procedure("get_dashboard_state", || {
    .users.values().map_wrap("_DashboardUser");
});
"get_dashboard_state"
```

Let's check if it works:

```
(//todos)> get_dashboard_state();
[
    {"id": 10, "count": 2, "name": "Alice"},
    {"id": 11, "count": 0, "name": "Bob"}
]
```

This confirms that the procedure successfully retrieves the desired initial state.

12.3.1 Integration of the Initial State

Now that we have the `get_dashboard_state` procedure, let's integrate it into the dashboard application:

```python
class Dashboard(Room):
    def on_init(self):
        # Called only once
        self.state = []

    async def on_join(self):
        # Called when joining the room
        self.state = await self.client.run("get_dashboard_state")
        self.print_state()

    def print_state(self):
        print('-' * 20)
        for user in self.state:
            name, count = user["name"], user["count"]
            print(f"{name:<15}{count:>5}")
```

- **Initial state in `on_init()`:**

    The `on_init()` method, inherited from `Room`, is called once during room initialization. We use it to initialize an empty state.

- **Updating state on join in `on_join()`:**

    The asynchronous `on_join()` method, called when joining the room, retrieves the current state using `get_dashboard_state` and updates the state. It then calls `print_state` to display the information.

    > 💡 **Important Note:** Unlike `on_init()`, `on_join()` can be called multiple times due to reconnections. This ensures the dashboard retrieves the latest state after any disruptions.

- **Custom `print_state()` method:**

    This custom method formats and prints the user names and uncompleted to-do counts from the state.

Running `dashboard.py` again will now print the state:

```
$ python dashboard.py
--------------------
Alice              2
Bob                0
```

158

12.3.2 Implement Event Handlers

Now we need to handle events that indicate changes to the data, ensuring our dashboard reflects the latest information.

Identifying state-changing procedures and methods:

- `add_todo`: Adds a new task to a user's list.
- `add_user`: Creates a new user.
- `mark_as_done`: Marks a task as completed.

Here's the code to update the procedures and methods:

```
(//todos)>
mod_procedure("add_todo", |name, body| {
    "Adds a to-do to the list.";
    todo = Todo{body:,};
    .users[name.lower()].todos.add(todo);
    .dashboard.emit("add-todo", todo.user.id());  // New line
    todo.id();
});
mod_procedure("add_user", |name| {
    "Add a new to-do user.";
    uid = name.lower();
    assert(!.users.has(uid));
    user = .users[uid] = User{name:,};
    .dashboard.emit("add-user", user.wrap("_DashboardUser"));  // New line
    user.id();
});
mod_type("Todo", "mod", "mark_as_done", |this| {
    .dashboard.emit("mark-as-done", this.user.id());  // New line
    this.completion = datetime();
});
null
```

Implementing the event handlers in `dashboard.py`:

```python
class Dashboard(Room):
    @event("add-todo")
    def on_add_todo(self, user_id):
        # Update user's to-do count and display updated state
        self.get_user(user_id)["count"] += 1
        self.print_state()

    @event("add-user")
    def on_add_user(self, user):
        # Add user to state and display updated state
        self.state.append(user)
        self.print_state()

    @event("mark-as-done")
    def on_mark_as_done(self, user_id):
        # Update user's to-do count and display updated state
        self.get_user(user_id)["count"] -= 1
        self.print_state()

    def get_user(self, user_id):
        # Find user object based on ID
        for user in self.state:
            if user["id"] == user_id:
                return user

    # ... other methods
```

12.3.3 Test Your Dashboard in Action!

Restart your `dashboard.py` script and have some fun:

1. **Add new tasks:** Create tasks for various users ("Alice", "Bob", or even a new user like "Charlie"). Observe how the dashboard automatically updates their to-do counts.
2. **Mark tasks done:** Mark some tasks as completed and witness the immediate reflection in the dashboard's user counts.
3. **Add new users:** Create additional users and see how they're automatically added to the dashboard.

Example output:

```
$ python dashboard.py
-------------------
Alice              2
Bob                0
-------------------
Alice              2
Bob                1
-------------------
Alice              1
Bob                1
-------------------
Alice              1
Bob                1
Charlie            0
…
```

As you interact, the dashboard dynamically updates the list, displaying the latest to-do counts for each user in real-time. This demonstrates the power of event-driven communication between your application and ThingsDB!

12.4 Room for More

We hope this chapter provided a solid foundation for using events and rooms in ThingsDB. While it does not cover everything, it equips you to start building interactive applications.

Expand your knowledge! For deeper insights into rooms and events, explore the client website specific to your chosen language. This website likely contains detailed information and examples beyond what is covered here. Do not miss the Connectors section of this book for quick access to these valuable resources.

12.5 Quiz - Challenge Your Understanding

1. Who will receive the message `"Hello World"` in the following statement?

    ```
    room().emit("new-message", "Hello World");
    ```

    Remember: Rooms must have an ID before they can be joined.

2. What access flag is required for a program to join a room in ThingsDB?
    a. QUERY
    b. CHANGE
    c. RUN
    d. JOIN

3. What method is available to get a room ID?

4. What is the most appropriate point to synchronize the state of a *Room* with the current state of the collection?
    a. `on_init(..)` *(During room initialization)*
    b. `on_join(..)` *(When joining the room)*
    c. `on_emit(..)` *(When emitting an event)*

5. Which of the following are valid event names?
    a. `"123"`
    b. `"new-building"`
    c. `"stop the program!"`
    d. `""`
    e. `"ADD_USER"`

6. Consider the following Python code snippet belonging to a `Room` class that listens to the `.math` room:

    ```python
    @event("add-two-numbers")
    def on_add_two_numbers(self, a, b):
        print(f"{a} + {b} = {a + b}")
    ```

    The observed output is:

    ```
    8 + 5 = 13
    ```

    Can you write the ThingsDB code that could have triggered this event, resulting in the displayed response?

12.5.1 Quiz - Answers

1. No one will receive the message `"Hello World"` because the statement lacks a room ID. Rooms require an ID for joining, and without one, a program cannot join the intended room.

2. The absolute requirement for joining a room in ThingsDB is the `JOIN` access flag. While the `QUERY` flag is not strictly mandatory, it offers valuable utility. It allows you to retrieve the room ID dynamically using a query.

3. The `id()` method is directly available on the room object and returns the ID as integer. While technically possible, extracting the ID from the string representation of the room object is less efficient and generally discouraged. The room object itself returns a string in the format `room:ID`.

4. Answer "b". The most appropriate point to synchronize the state with the current collection in a `Room` implementation is `on_join(..)`. While the `on_init(..)` could be used for initial state retrieval, it would not account for potential changes after a connection loss. The `on_join(..)` handler is called every time a room is joined, making it ideal for ensuring the latest state is retrieved from the collection.

5. All event names are valid *except for empty strings* (answer "d"). Remember, event names need at least one character and cannot exceed 255 characters.

6. The following ThingsDB code could have triggered the `"add-two-numbers"` event, leading to the observed output:

   ```
   .math.emit("add-two-numbers", 8, 5);
   ```

Chapter 13 - Futures and Modules

Ready to push your ThingsDB projects further? This chapter dives into the world of *modules*, unlocking a vast array of functionalities to extend your applications beyond the core features.

Imagine sending automated emails, interacting with external APIs, storing files in the cloud, or even connecting to other databases – modules make these and more possible!

But before we explore the exciting world of modules, let's take a quick detour and understand a key concept: *futures* in ThingsDB.

By understanding futures, you'll be well-equipped to harness the true power of modules and unlock the full potential of your ThingsDB projects.

13.1 Demystifying Futures: Waiting with Purpose

Think of futures as placeholders for tasks running in the background, allowing your code to continue without getting stuck waiting.

We'll start by creating an "empty future", which doesn't have a specific task but still waits for a query to complete before executing asynchronously:

```
(//todos)> future(nil);
[
    null
]
```

This code creates a future without any specific task. It simply waits for the query to finish and then executes asynchronously. The response you see (`[null]`) is the result of the future task (which, in this case, is nothing).

13.1.1 Using the Future's Task Result

We have seen how futures hold the place for background tasks. But what if we want to do something with the result of that task once it is finished? That is where the `then()` method comes to the rescue!

Imagine you delegate a task to a friend. When they finish, they share the result. The `then()` method acts like your friend, waiting for the future to finish and then running a closure you provide with the result.

Here is an example:

```
(//todos)> future(nil).then(|result| is_nil(result));
true
```

As you can see, the query response now shows the outcome of the closure (`true` in this case).

The closure provided to the `then()` method operates in a separate context than your main code. It cannot directly access variables defined outside, like in this example:

```
(//todos)>
x = 10;
future(nil).then(|| x + 10);
LookupError: variable `x` is undefined
```

So, how do we use existing variables within a future's closure? We simply pass them along! Wrap them up and send them together with the future using the `future()` function:

```
(//todos)>
x = 10;
future(nil, x).then(|_, x| x + 10);
20
```

Specifying `nil` as an argument and then ignoring it could get a bit tedious. Well, ThingsDB offers a handy shortcut!

Instead of `then()`, simply provide the closure as the first argument to `future()`. This magical closure inherits variables from your main context, eliminating the need to pass them explicitly:

```
(//todos)>
x = 5;
future(|x| x + 10);   // No more `nil`s!
15
```

As you noticed, the task result is not explicitly mentioned, but since it is an empty future, we know it is `nil` anyway.

Do you want to pass the arguments explicitly to the closure? No problem! Wrap them in a list as the second argument to `future()`:

```
(//todos)>
future(|x| x + 10, [8]);   // `x` is provided by the first item in the list
18
```

165

By leveraging this simplified syntax, you can create and work with empty futures in a more concise and efficient way, making your asynchronous operations in ThingsDB cleaner and easier to manage.

13.1.2 Empty Futures: Isolating Side Effects for Efficiency

You might wonder, "Why bother with empty futures?" The key lies in their ability to create separate contexts for closures, unlocking hidden potential.

Imagine this scenario in our to-do application: we need a procedure to retrieve or create a user and then grab their ID (or other information, but let's focus on ID for now).

One way to tackle this challenge is the following implementation:

```
(//todos)> new_procedure("get_or_create_user_id", |name| {
    user = .users.get(name.lower());
    if (is_nil(user)) {
        wse();  // Enforced Side effect for the add_user() call
        return add_user(name);
    };
    user.id();
});
"get_or_create_user_id"
```

While this works for both existing and new users, it has a flaw: it triggers side effects even when a user already exists. This can be inefficient, especially if the procedure is called frequently.

Here's how we can use an empty future to isolate side effects and improve efficiency:

```
(//todos)> mod_procedure("get_or_create_user_id", |name| {
    user = .users.get(name.lower());
    if (is_nil(user)) {
        return future(|name| {
            wse();  // Side effect now contained within the future
            add_user(name);
        });
    };
    user.id();
});
null
```

Now, the main branch retrieves the existing user ID without causing a change. If a new user needs to be created, the future's closure handles it asynchronously, ensuring updates only happen when necessary.

166

Remember from [Chapter 6.3](#) that we can verify if a procedure triggers side effects by checking the `with_side_effects` property. Let's confirm that our modified `get_or_create_user_id` procedure is truly free of them:

```
(//todos)>
procedure_info("get_or_create_user_id").load().with_side_effects;
false
```

The value `false` indicates no side effects are detected within the main logic of the procedure.

13.1.3 Caution! Futures Don't Always Behave Like Values

Before you store a future in your ThingsDB project, it is crucial to understand a key quirk: **futures store their current state, not the eventual result**. Just assigning a future to a variable works as expected, but attempting to store it in collections or object properties can lead to unexpected results.

Let's explore the scenario:

Imagine you create a future that calculates `2 + 2` and store it in a list:

```
(//todos)> [future(|| 2 + 2)];
[
    null
]
```

You might expect the list to contain `[4]`. However, the actual result is `[null]`. Why? Because you are not storing the future itself, but rather its *initial* state, which is `nil` before the calculation finishes.

13.2 Enhance Your ThingsDB with Modules

Now that you are familiar with futures, let's explore modules, which can significantly expand ThingsDB's capabilities. Modules can unlock various features like sending emails, connecting to diverse databases, or making HTTP(S) requests.

Module Types:

- **Python Modules:** Python modules offer a familiar language option for future potential customization.
- **Binary Modules:** ThingsDB utilizes pre-written binary modules, providing various functionalities. Writing your own modules in a language like Go is possible but beyond this book's scope.

For this book, we'll focus on pre-built binary modules, readily available and requiring no additional coding on your part.

> To install official modules, you'll need Internet access as they are downloaded directly from GitHub repositories.

13.2.1 Install The Demo Module

Module installation in ThingsDB happens smoothly within the `/thingsdb` scope using the `new_module()` function. Just give your module a friendly alias (like `"demo"`), then provide the corresponding GitHub repository URL where it resides.

> Want a specific version? Simply add `@v0.1.0` (or similar) alongside the URL, and you're good to go!

Switch to the `/thingsdb` scope and install the `demo` module:

```
(//todos)> @t
(@t)> new_module('demo', 'github.com/thingsdb/module-go-demo');
null
```

Once you have installed the module, use the `module_info()` function to confirm its success.

```
(@t)> module_info("demo");
{
    "conf": null,
    "created_at": 1707931759,
    "doc": "https://github.com/thingsdb/module-go-demo#readme",
    "exposes": {
        "echo": {
            "argmap": ["message"],
            "defaults": {"load": true},
            "doc": "Echo message"
        }
    },
    "file": "/usr/lib/thingsdb-modules/demo/bin/demo_linux_amd64.bin",
    "github_owner": "thingsdb",
    "github_ref": "default",
    "github_repo": "module-go-demo",
    "github_with_token": false,
    "name": "demo",
    "scope": null,
    "status": "running",
    "version": "0.1.0"
}
```

> 💡 If the status is still `"installing module..."`, wait a few seconds longer for the installation to complete.

Now that the `demo` module is installed, let's explore how to use its capabilities by understanding the methods it is exposing.

```
"echo": {
    "argmap": ["message"],
    "defaults": {"load": true},
    "doc": "Echo message"
}
```

Every module exposes specific methods. For `demo`, it is the `echo()` method. This method takes a single argument, a `message`, and echoes it back. The `load` attribute being set to `true` means the echoed message will be loaded directly into ThingsDB, making it readily accessible.

> 💡 By setting `load` to `false` (the default), you can unlock a special response format called "mpdata". This is ideal for modules that fetch data from external sources where you do not need to store it in ThingsDB. Instead, the data gets delivered directly to the client, avoiding unnecessary unpacking and repackaging steps. This significantly reduces processing overhead, making your application more efficient.

Let's test the `echo()` method:

```
(@t)> demo.echo("Hello Module!");
[
    "Hello Module!"
]
```

As you see, the result is a list containing the echoed message. This is because calling a module method ultimately returns a future, similar to our previous future examples. However, in this case, the future holds the actual result (the echoed message) instead of being empty.

We can use the familiar `then()` method to capture and manipulate the echoed message:

```
(@t)> demo.echo("Hello Module!").then(|result| result.upper());
"HELLO MODULE!"
```

Here, `then()` grabs the echoed message (`"Hello Module!"`), converts it to uppercase, and returns the result.

> 💡 All exposed methods accept a *fixed amount* of arguments, specified in the `argmap` property. Any arguments exceeding this limit will not be passed to the module's task itself. Instead, they will be parsed to the subsequent `then()` or `else()` callback methods you might use for handling results or errors.

13.2.3 Sending NTFYs with the HTTP(S) Request Module

This section demonstrates another powerful module: the HTTP(S) request module. It allows you to seamlessly interact with other web services and APIs.

> 💡 **Install the "ntfy" app:** Download the app on your phone (https://ntfy.sh/) and subscribe to the `"ThingsDB_Book"` topic (or create your own).

Install the `requests` module using this code:

```
(@t)> new_module("requests", "github.com/thingsdb/module-go-requests");
null
```

Verify the installation:

```
(@t)> module_info("requests").load().status;
"running"
```

The `requests` module boasts a powerful `post_json()` method. Watch how it sends a message to your NTFY app:

```
(@t)> requests.post("https://ntfy.sh/", json_dump({
    topic: "ThingsDB_Book",
    message: "This book is fantastic!"
})).then(|| "OK").else(|err| err.msg());
"OK"
```

Check your NTFY app – you should see the `"This book is fantastic!"` message.

Just like `then()`, `else()` is available for all futures and plays a crucial role in error handling. It defines what happens when a request or task encounters an error.

While `else()` is not needed for simple futures that always resolve successfully, it becomes essential when dealing with external sources like NTFY. These

170

external systems can potentially fail due to network issues, server errors, or other unforeseen circumstances. If both `else()` and `then()` are defined, only one will be called based on the outcome (success or error).

13.2.4 Talking to Yourself: Connecting ThingsDB Scopes

While the modules we have seen so far work out-of-the-box, others require configuration for specific tasks. Let's explore the `thingsdb` module, which allows communication across different ThingsDB scopes.

You might think, *"We're already in ThingsDB, why connect to it again?"*. This module unlocks the ability to communicate between different scopes within the same ThingsDB node or even connect to separate ThingsDB clusters. Imagine you have data stored in a different scope that you need in your current scope, or you want to trigger actions across multiple clusters. This module makes it possible!

As always, installation comes first:

```
(@t)> new_module("thingsdb", "github.com/thingsdb/module-go-thingsdb");
null
```

ThingsDB requires proper authentication. To grant the module access, let's create a dedicated user. We can leverage the same `new_todo_user()` procedure we established in [Chapter 6.4](#):

```
(@t)> wse(); new_todo_user("module", RUN);
"1w1WtiUZqSpCEX+B3p/Dtq"
```

With the token in hand, we'll configure the module using `set_module_conf()`.

Here's how:

```
(@t)> set_module_conf("thingsdb", {
    token: "1w1WtiUZqSpCEX+B3p/Dtq",   // Replace with your token
    host: "localhost",
});
null
```

The function `set_module_conf()` operates asynchronously, meaning it initiates the configuration process in the background and returns `nil` immediately.

To confirm successful configuration, always use `module_info()`:

```
(@t)> module_info("thingsdb");
{
    "conf": {
        "host": "localhost",
        "token": "1w1WtiUZqSpCEX+B3p/Dtq"
    },
    "status": "running",
    …  // — other information
}
```

Look for a status of `"running"` and correct configuration in the `conf` section.

Troubleshooting:

- **Error Messages:** If the `status` shows an error, carefully examine the message for guidance.
- **Detailed Logging:** Set the log level to `DEBUG` for more information in the node console *(see Chapter 15.4 on how to enable debug logging)*.
- **Node specific:** For deeper troubleshooting, utilize `module_info()` within a node scope. This unlocks the number of active `tasks` and `restarts`. Active tasks reveal current workload on that node, potentially indicating bottlenecks. Restarts provide insight into module stability, highlighting frequent restarts as potential red flags.

Now that the ThingsDB module is configured, let's put its cross-scope communication power to the test!

We'll use the `search_todos()` procedure to find to-do's, even though we're currently in the `/thingsdb` scope:

```
(@t)> thingsdb.run("//todos", "search_todos", ["book"]).then(|res| res);
[
    {
        "body": "Read a book",
        "done": true,
        "id": 2,
        "severity": "Medium",
        "user": {"name": "Alice"}
    }
]
```

As you can see, we successfully retrieved a to-do containing `"book"` from the `//todos` scope! This demonstrates the magic of cross-scope communication enabled by the ThingsDB module.

13.3 Managing Your Modules: Access, Updates, and More

13.3.1 Controlling Module Access

By default, all installed modules are accessible by any scope. This is indicated by a `nil` value in the `scope` property of the module information.

To limit a module's accessibility to a specific scope, use the `set_module_scope()` function. Remember, each module can only have one assigned scope at a time, making it either globally accessible or scoped to a single entity.

13.3.2 Multiple Configurations, Multiple Installations

If you need the same module with different configurations, or with access restrictions to different scopes, consider installing it multiple times with different names. This allows you to tailor each instance to its specific needs.

For example: Imagine connecting to different ThingsDB clusters. Install the `thingsdb` module twice with unique names, each configured to access a different cluster.

13.3.3 Keeping Your Modules Up-to-Date

Modules are constantly evolving, so you might need to update them to benefit from new features and bug fixes. The `refresh_module()` function helps you out here. It stops the module, checks for available updates, performs the update if necessary, and restarts the module.

For granular control over module source updates, leverage the `deploy_module()` function. While sharing similarities with `refresh_module()`, it offers the crucial capability to specify a new source for the module. This empowers you to, for instance, "pin" the module to a specific version, ensuring a stable and predictable environment. This is particularly valuable when working with dependencies or critical modules where consistent behavior is paramount.

13.4 Quiz - Challenge Your Understanding

1. Can you simplify the following code while preserving its functionality and using a future?

   ```
   future(nil, 6, 7).then(|_, a, b| a * b);
   ```

2. Can you answer this tricky question: Is it possible to directly return *two futures* in a single query result in ThingsDB?

3. You encounter an exposed method within a module, and its `load` property has a default value of `true`. What does this imply?

   a. The module automatically loads into memory upon usage.
   b. The method's response will be deserialized.
   c. The property indicates the method is ready for use.
   d. The property has no effect on the method's behavior.

4. You are working with a ThingsDB module named `foo`. Its exposed method `bar()` has an intriguing `argmap` property set to `["*"]`. Consider the following code snippet:

   ```
   foo.bar(nil, 42).then(|A, B| nil).else(|C, D| nil);
   ```

   Choose the most accurate statement about this call to the `bar()` method:

   a. Both `A` and `D` will be nil regardless of success or failure.
   b. On success, `A` will be `nil` and `B` will contain the method's response.
   c. On success, `B` will be `42` and on failure, `C` will be `42`.
   d. On success, `A` will contain the response and on failure, `C` will hold the error.

   Important: The `"*"` in an `argmap` signifies a single, unnamed argument that can hold either `nil` or a thing

5. Uh oh! A module in your ThingsDB application isn't behaving as expected. What steps can you take to diagnose the issue and get it back on track?

   Choose the most effective troubleshooting approach from the following:

   a. Restart the node and hope for the best.
   b. Reinstall the module without checking anything else.
   c. Use the `module_info()` function to gather information and check the node console logs.
   d. Ignore the issue and hope it fixes itself eventually.

6. Scenario: You have a module pinned to version `v0.2.1` using the `@` notation. Now, a newer version (`v0.3.0`) with exciting features is

available. How can you seamlessly update the module to tap into these enhancements?

13.4.1 Quiz - Answers

1. ThingsDB offers a clever feature to create futures without an actual task. This allows you to directly combine argument unpacking with future creation, eliminating the need for an empty placeholder. Here is the code:

    ```
    future(|a, b| a * b, [6, 7]);
    ```

    This code accomplishes the identical multiplication as the original, but leverages a more concise and efficient syntax.

2. While futures offer flexibility, directly returning two futures in a single response is not possible. Remember, a future can only be assigned to a variable. Assigning it to a list, for example, would simply store `nil` instead of the actual future.

    But wait! There is a workaround! While returning two direct futures is not possible, you can achieve similar behavior by chaining futures using the `then()` method. The first future's result can be passed to the second future, effectively creating a chained execution. One more workaround is to parse them as arguments to an empty future:

    ```
    future(nil, future(|| 2+2), future(|| 3*3)); // response: [nil, 4, 9]
    ```

3. Answer "b" is correct. When set to `true`, it instructs ThingsDB to automatically deserialize the response from the method call, making it readily accessible for further processing or manipulation. However, if you set the `load` property to `false` (the default), the method returns the response as "mpdata". This option is particularly useful when you do not need to use the data within your ThingsDB application or when you plan to send it directly to a client. In these cases, deserializing and then serializing it again would be unnecessary and potentially slow down your application.

4. Answer "d" is the correct answer. The `argmap` property set to `["*"]` signifies that the `bar()` method can accept a single argument (`nil` or a thing). In this case, you are providing nil as the argument. Additional arguments (like the `42` in the code snippet) won't be processed by the module itself. Instead, they'll be passed directly to the `then()` or `else()` callbacks making B or D equal to `42` depending on success or failure. This allows you to parse data to your callback logic.

5. Option "c" offers the most effective path to diagnosis. Use `module_info()` to get its `config` and `status`, plus check the node console logs for clues. These steps unlock insights into the issue, paving the way for a fix. Remember, understanding the root cause is key!

6. When seeking precise control over module updates, leverage the `deploy_module()` function. This tool empowers you to define the new source for the update, replacing any pinned version tags with the desired reference (e.g., `@v0.3.0` instead of `@v0.2.1`). Remember to prioritize testing updates in a non-production environment before deploying them to ensure compatibility and smooth operation. While `refresh_module()` can also update modules, it respects pinned versions.

Chapter 14 - Safeguarding Your Data: Backup, Restore, Export and Import

Now that you have explored the power of ThingsDB, it is crucial to consider data protection. While ThingsDB is designed for minimal downtime by scaling across multiple nodes, accidents or unforeseen events can still occur, potentially leading to data loss. Here's where backups come in!

The good news is, ThingsDB offers scheduled backups without impacting ongoing operations. Unlike some solutions that lock access during backup creation, ThingsDB seamlessly handles this on *multi-node* setups.

Later in this chapter we will also look at single collection exports and how to import them when required. This is also useful for development as it allows you to quickly load another collection on your local development machine without disrupting other collections you might be working on.

14.1 Node Scopes for Backups

So far, we have used the `/thingsdb` scope for managing user accounts and collections, and collection scopes for interacting with data. Now, for backups, we need a new scope: node scopes.

Unlike the previous scopes, backups are node-specific. While each node eventually stores the same data, you tell ThingsDB which node should create a backup.

> 💡 **Production-Ready Backup Strategy**
>
> For reliable data protection, consider using at least two different nodes for backups. This ensures backups continue even if one node fails. Creating backups on all nodes is generally not required.

Before scheduling a backup, we need to identify the node responsible for creating it.

Access the node scope by typing `/node` (`@n` or `/n` for short) in the prompt:

```
(@t)> @ /node
(/node)>
```

When you do not specify a node ID in the scope, ThingsDB automatically uses the node you are currently connected to. To check which node this is,

use the `node_info()` function and extract the `node_id` property using the following command:

```
(/node)> node_info().load().node_id;
0
```

The output `0` signifies you are currently connected to the node with ID 0.

The `nodes_info()` function displays all nodes within your ThingsDB setup, providing valuable information like their IDs, names, and statuses:

```
(/node)> nodes_info();
[
    {
        "committed_change_id": 1,
        "node_id": 0,
        "node_name": "node0",
        "port": 9220,
        "status": "READY",
        "stored_change_id": 1,
        "stream": null,
        "syntax_version": "v1",
        "zone": 0
    }
]
```

In this set-up, we only have one node with ID 0, explaining the current connection.

> Wonder why `node_info()` offers more detailed information compared to `nodes_info()`? Both commands are executed on a specific node, meaning you receive information based on that node's knowledge about itself and other nodes.

However, if you have a multi-node setup, specifying the node ID is crucial for targeted operations. To force a query on a specific node, simply include its ID within the scope like this:

```
(/node)> @ /node/0
(/node/0)>
```

Now that you have selected the right scope, let's create your first ThingsDB backup!

14.2 Your First Backup

179

To create a backup, use the `new_backup()` function. It requires at least one argument: the target filename. Backup filenames must end with `".tar.gz"`. This ensures consistency and helps you understand the format of the saved data.

ThingsDB offers handy template variables you can embed in your filename. These variables get automatically filled when the backup is created, making it easier to organize and identify your backups.

Table 14.2 - Summarizing backup template variable

Variable	Description	Example
{DATE}	Current date (YYYYMMDD format)	20240221
{TIME}	Current time (HHMMSS format)	092713
{CHANGE_ID}	Last committed change ID	123456

Here's a code snippet that creates a backup named `"/tmp/ti-backup-{DATE}-{TIME}.tar.gz"`.

Note: If you are using Docker Compose as described in this book, remember to adjust `/tmp` to `/dump` (which is volume-mounted to your local disk).

```
(/node/0)> new_backup("/tmp/ti-backup-{DATE}-{TIME}.tar.gz");
0
```

The return value (`0` in this case) is a unique backup ID assigned within that specific node.

Now that you have the backup ID, let's check its status using the `backup_info()` function:

```
(/node/0)> backup_info(0);  // Use the backup ID (0 in our example)
{
    "created_at": 1708507632,
    "file_template": "/tmp/ti-backup-{DATE}-{TIME}.tar.gz",
    "files": [
        "/tmp/ti-backup-20240221-092713.tar.gz"
    ],
    "id": 0,
    "result_code": 0,
    "result_message": "success - 2024-02-21 09:27:13Z"
}
```

In this example, the `result_code` is `0`, indicating a successful backup. Additionally, the `result_message` confirms this with

180

`"success - 2024-02-21 09:27:13Z"`. In case of issues, the result message offers further information or troubleshooting guidance.

> ### 💡 Production-Ready Backup Strategy
>
> During backup creation, the node enters a temporary mode called `AWAY_MODE`. This mode ensures data consistency by temporarily suspending regular queries while the backup process takes place. However, queries specifically directed to that node using the node scope are still accepted and handled normally.
>
> For multi-node setups, this temporary unavailability goes unnoticed by clients as queries are automatically forwarded to other available nodes. ThingsDB ensures only one node enters `AWAY_MODE` at a time, guaranteeing uninterrupted service while maintaining data integrity.

14.3 Restoring Your Data

Having created a backup, let's discuss restoring it. For full backups like the one we made earlier, a comprehensive restore is the only option. To ensure a smooth restoration process, ThingsDB performs some essential pre-checks:

> ### 💡 Production-Ready Backup Strategy
>
> Instead of directly restoring onto your primary ThingsDB, consider a safe testing approach using separate storage paths. Set temporary environment variables for `THINGSDB_STORAGE_PATH` and `THINGSDB_MODULES_PATH` pointing to dedicated directories. This allows you to run ThingsDB using this alternate location leaving your primary data intact.

User Permissions:

- The user initiating the restore must possess `FULL` privileges on the `/thingsdb` scope.

Data Presence:

- No existing collections: Verify this using `collections_info()` and, if necessary, remove them with `del_collection()`. *(Note: empty collections may exist and are ignored)*
- No existing modules: Use `modules_info()` to check and `del_module()` to remove any modules if needed.
- No tasks: Check for tasks in the `/thingsdb` scope using the `tasks()` function and remove them if necessary.

Node Status:

- All nodes must be online and ready: If not, either remove the problematic node or wait for it to be ready. Use `nodes_info()` to monitor node status.
- Multi-node setups: All committed changes must be stored on all nodes. `nodes_info()` displays both `committed_change_id` and `stored_change_id` for each node. This check is not required for single-node setups.

Initiate the restore process using the `restore()` function within the `/thingsdb` scope. This function requires the filename of the backup you want to restore. Additionally, you can provide an optional thing argument containing extra restore options for further customization.

Here's an example using restore options:

```
(/t)> restore("/tmp/ti-backup-20240221-092713.tar.gz", {
    take_access: true,
    restore_tasks: true,
});
0
```

In the provided example, the two key restore options are:

- `take_access: true`
  This grants full access to all restored scopes only to the user performing the restore. If you keep this option as the default `false`, access will be restored based on the original user permissions associated with each scope.

- `restore_tasks: true`
  This includes scheduled tasks in the restoration process. However, keep in mind that tasks might automatically trigger based on their schedules once restored. This could be undesirable if you need time to adjust the environment after the restore. The default `false` setting prevents task execution.

After completing the restore, even with `take_access: true`, re-authenticate by signing out and signing in again due to potential user ID changes.

> While most modules function after a ThingsDB restore, some may need a nudge. Check their status with `modules_info()`. If any are not working properly, use `refresh_module()` to reload their configuration and state. Alternatively, restarting the node(s) will also refresh all modules.

14.3.1 Restoring Data from a Multi-Node Setup

When restoring data from a backup created in a multi-node ThingsDB setup, you might encounter unintended behavior on single-node environments like your development machine. During the restore process, ThingsDB attempts to contact other nodes based on the information stored in the backup, which can be disruptive in this context.

To prevent this unnecessary search for non-existent nodes, simply restart your ThingsDB instance after the restore is complete using the `--forget-nodes` command-line argument. This instructs ThingsDB to disregard any stored node information and operate as a single-node setup, aligning with your development environment.

14.4 Automating Your Backups

ThingsDB empowers you to schedule automatic backups, ensuring regular data protection without manual intervention. Let's explore how to set up scheduled backups:

The `new_backup()` function offers three additional arguments for scheduling:

- `start_ts`
  Define the **start date and time** for the initial backup. If omitted, the backup starts immediately.

- `repeat`
  Set the **repetition interval** in seconds. For daily backups, use `24 * 3600`. Leaving this blank creates a one-time backup.

- `max_files`
  Determine the **maximum number of backup files** to retain. When this limit is reached, the oldest backup is automatically removed. The default is `7`.

Let's schedule a daily backup at 22 PM Kyiv time, keeping up to 14 backup files:

```
(/t)> @ /node/0
(/node/0)> new_backup(
    "/tmp/daily-{DATE}-{TIME}.tar.gz",
    datetime().to("Europe/Kyiv").replace({hour: 22, minute: 0, second: 0}),
    24*3600,  // Repeat every 24 hours
    14,  // Keep 14 backup files
);
1
```

This starts a backup immediately as we start with a time in the past, the next one 24 hours later relative to the given start time.

Use `backup_info()` and the backup ID (`1`) to confirm the schedule:

```
(/node/0)> backup_info(1);
{
    "created_at": 1708624740,
    "file_template": "/tmp/daily-{DATE}-{TIME}.tar.gz",
    "files": [],
    "id": 1,
    "max_files": 14,
    "next_run": "2024-02-22 20:00:00Z",
    "repeat": 86400
}
```

This shows details of your backup schedule, including the next run at 22:00 PM Kyiv time (`2024-02-22 20:00:00Z`), repeat interval, and maximum file count.

14.4.1 Ensuring Backup Health

ThingsDB offers a convenient function called `backups_ok()` to quickly check the overall health of your backup system. It returns a simple `true` or `false` value, making it easy to monitor if all backups are running as planned.

- `true`: All scheduled backups are successful or have not yet started.
- `false`: At least one backup has failed.

```
(/node/0)> backups_ok();
true
```

This response indicates that all your scheduled backups are currently successful or have not started yet.

14.4.2 Google Cloud Storage

Looking to run ThingsDB on Google Cloud Platform (GCP)? We've got you covered! ThingsDB offers specialized Docker images for GCP, enabling you to write backups directly to a storage bucket.

While we do not delve into the complete workflow here, we want you to be aware of this powerful option. More cloud platforms might be supported in future releases, so stay tuned to the documentation for updates.

For GCP users, the ThingsDB GitHub repository provides a detailed guide for setting up backups on the GKE platform:

- GitHub Link: https://github.com/thingsdb/ThingsDB/tree/main/gke#readme

What you will find:

- Step-by-step instructions for configuring ThingsDB on GKE with Google Cloud Storage integration.
- Guidance on utilizing the `new_backup()` function to write backups to your storage bucket.
- Essential considerations for managing and monitoring your backups.

14.5 Exporting and Importing Collections

While backups are crucial for disaster recovery, sometimes you need to work with specific collections or data subsets. ThingsDB's `export()` function provides flexibility for these scenarios.

Collection Schemas

Run `export()` without arguments to retrieve the collection schema definition in a readable format. This includes enumerators, types, and procedures, but not the actual data.

Exporting Everything

Call `export()` with a "thing" argument containing a `dump` property set to `true`. This exports the entire collection, including data and tasks (if any), in MessagePack format. You can then use the `import()` function to restore this data.

ThingsDB Prompt simplifies the export/import process by letting you directly invoke `export()` and `import()` functions from the command line, eliminating the need for manual file handling.

Use `export` as an argument to write an export to a file. The default for `things-prompt` is to export everything:

```
$ things-prompt -u admin -p pass -s //todos export /tmp/dump.mp
```

Use `import` as an argument to import the exported data into a new collection:

```
$ things-prompt -u admin -p pass -s //dev import /tmp/dump.mp
```

This creates a new collection *"dev"* with data identical to the original *"todos"* collection.

> 💡 **Production-Ready Backup Strategy**
>
> Tasks are not automatically imported during collection imports to prevent unintended immediate execution. If your exported collection contains tasks and you want them imported, use the `--tasks` flag with the `import` command.

14.5.1 Exporting Collection Schemas

You can also export just the collection's definition (enumerators, types, procedures) for future reference or sharing. This exported schema is in a readable ThingsDB code format.

Include the `--structure-only` flag with the `export` command:

```
$ things-prompt \
    -u admin -p pass -s //todos export /tmp/dump.ti --structure-only
```

This creates a pain text file `/tmp/dump.ti` containing the collection's code, including enum definitions like:

```
// Enums
set_enum('Severity', {
        Medium: 1,
        Low: 0,
        High: 2,
        str: |this| `{this.name()} ({this.value()})`,
});
// … more lines
```

As before, use `import` to import the exported schema into a (new) collection.

```
$ things-prompt -u admin -p pass -s //schema import /tmp/dump.ti
```

This creates a new collection "schema" with the same enumerators, types and procedures as the original, but without any data.

> 💡 Importing full collections with data can only be done into new or empty collections. This prevents accidental data overwrites.
>
> However, plain text imports (like created by a `--structure-only` export) treat the imported code as a regular ThingsDB query and can apply it to any existing collection. This means it directly executes the provided code within that collection, potentially modifying data or creating new entities.

14.6 Choosing Your Tool: Backups vs. Exports

In conclusion, exports and backups are both valuable tools, but serve distinct purposes:

- **Exports:** Ideal for development environments. They provide an easy way to extract specific collections or their structures for testing, sharing data subsets, or migrating between instances.
- **Backups:** Essential for disaster recovery and protecting production data. They create complete copies at specific points in time, ensuring you can recover from unexpected events without impacting your ThingsDB operations. Backups are non-intrusive and run seamlessly in the background, safeguarding your data without disrupting your workflow.

Good luck with the quiz, and see you in the next chapter on adding nodes to your ThingsDB set-up!

14.7 Quiz - Challenge Your Understanding

1. Which approach is more suitable for safeguarding your ThingsDB data?: backups or exports?

   a. Backups
   b. Exports
   c. Both, depending on the situation

2. For disaster recovery, what is the recommended scope for creating a backup using the `new_backup()` function?

   a. A collection scope, like: `//stuff` or `//todos`
   b. The client connection node scope: `/n`
   c. ThingsDB scope: `/thingsdb`
   d. A specific node scope like: `/n/0` or `/n/1`

3. Disaster strikes! You accidentally forget your password and lose access to your ThingsDB instance. Thankfully, you have a recent backup. What is the best way to regain access to your data?

   a. Restore the backup without any additional options and hope for the best.
   b. Restore the backup using `restore()` and manually grant yourself access to each individual scope.
   c. Restore the backup using `restore()` with the `take_access: true` property, regaining access to all restored scopes automatically.
   d. There is no way to recover access without knowing the password.

4. How can you verify that your ThingsDB backups are functioning as intended?

5. Consider the behavior of tasks during imports and restores: By default, they are not included. Why do you think this happens, and what are the potential implications?

   a. To prevent unintended execution and potential data loss.
   b. To reduce the file size of the import or restore.
   c. To allow users to selectively restore or import specific tasks.
   d. Because tasks are not considered part of the core data structure.

14.7.1 Quiz - Answers

1. Answer "a" is correct. Backups are your data's safety net. They create complete copies at specific points in time, ensuring you can recover from disasters like hardware failures or accidental deletion. These backups run silently in the background on designated nodes and can be scheduled for automatic execution, safeguarding your data without disrupting your ThingsDB operations.

   Exports on the other hand allow you to extract specific collections or structures for development purposes. This makes them ideal for creating testing environments, sharing data subsets, or migrating data between instances.

2. The correct answer is "d". While both `/n` and `/n/0` are technically valid scopes for creating a backup, using an explicit node scope like `/n/0` is recommended as it explicitly specifies the node on which the backup will be scheduled, eliminating ambiguity and potential confusion.

3. The correct answer is "c". Restore the backup using `restore()` with the `take_access: true` property. This efficiently restores your data and automatically grants you access to all restored scopes, ensuring a smooth recovery process.

   *(Once the restore is completed, always **re-authenticate** due to possible user ID changes!!)*

4. The `backups_ok()` function provides a convenient boolean answer on backup success, but it does not give you the "why" behind failures. For that, you need the `backups_info()` function, which goes beyond a simple `true` / `false` by providing a result message, often pinpointing the exact issue or offering clues for troubleshooting.

5. Including tasks in imports or restores can lead to unexpected behavior and potential data manipulation if not carefully considered. If, for example, a task is scheduled for a specific date in the past, restoring it might trigger immediate execution, potentially causing issues with outdated data or unintended consequences. Therefore, answer "a" is correct.

Chapter 15 - Multiple Nodes and Debugging

Ready to unleash the full potential of your ThingsDB setup? This chapter dives into two essential topics:

First, we'll guide you through the process of integrating additional nodes. Next, we'll equip you with valuable troubleshooting tools, from identifying bottlenecks to analyzing data flow, ensuring your ThingsDB runs smoothly and efficiently.

Not interested in exploring multiple nodes for development? Feel free to skip the next sections and continue at .

15.1 Scaling Up: Adding Nodes

In this section, we shall guide you through two options for adding nodes:

- ThingsDB from Source: We'll provide step-by-step instructions for starting a new node in this scenario.
- Using Docker Compose: If you followed the Docker installation instructions, we'll explain how to add another node using Docker Compose.

> 💡 **Real-world deployments:** While this section focuses on testing with multiple nodes on a single machine for development purposes, remember that in production environments, you would typically run each node on a separate machine or leverage a container orchestration platform like Kubernetes.

15.1.1 ThingsDB from Source

Ready to test multiple ThingsDB nodes on a single machine? Let's explore how to achieve this when you have built ThingsDB from source.

Each node requires unique TCP ports and data storage paths. You can configure them through environment variables or a configuration file.

Table 15.1.1 summarizes the essential variables for setting up multiple nodes. For a full list of configuration options, refer to the documentation: https://docs.thingsdb.io/v1/getting-started/configuration/

Table 15.1.1 - Relevant environment variables for multiple nodes

Variable	Description
THINGSDB_LISTEN_CLIENT_PORT	Listen to this TCP port for client socket connections.
THINGSDB_LISTEN_NODE_PORT	Listen to this TCP port for node connections.
THINGSDB_HTTP_API_PORT	(Optional) Listen to this TCP port for HTTP API requests.
THINGSDB_HTTP_STATUS_PORT	(Optional) Listen to this TCP port for HTTP status requests.
THINGSDB_WS_PORT	(Optional) Listen to this TCP port for WebSocket requests.
THINGSDB_MODULES_PATH	Path where ThingsDB modules are stored.
THINGSDB_STORAGE_PATH	Location to store ThingsDB data.

Create a separate directory: (we use "node1" and consider the first one as "node0")

```
$ mkdir ~/node1
```

Start the node with unique port numbers and data paths:

```
$ THINGSDB_LISTEN_CLIENT_PORT=9201 \
  THINGSDB_LISTEN_NODE_PORT=9221 \
  THINGSDB_HTTP_API_PORT=9211 \
  THINGSDB_HTTP_STATUS_PORT=8081 \
  THINGSDB_WS_PORT=9271 \
  THINGSDB_MODULES_PATH=~/node1/modules \
  THINGSDB_STORAGE_PATH=~/node1/data \
  thingsdb --secret pass
…
Waiting for an invite from a node to join ThingsDB...

You can use the following query to add this node:

    new_node('pass', 'mylaptop', 9221);
```

With the node waiting for an invite, skip the next section and jump to 15.2 - Invite the New Node.

191

15.1.2 Using Docker Compose

In your `docker-compose.yml` file, find the "services" section and uncomment (or add) the following configuration for a second node:

```
services:
  node1:
  << : *ti
  hostname: node1
  container_name: node1
  command: "--secret pass"
  ports:
    - 8081:8080
  volumes:
    - ./node1/data:/data/
    - ./node1/modules:/modules/
    - ./node1/dump:/dump/
```

Navigate to the directory containing your `docker-compose.yml` file and run:

```
$ docker compose up -d
[+] Running 2/2
 ✔ Container node1  Started
 ✔ Container node0  Running
```

Verify the new node is running by checking the container logs:

```
$ docker logs node1
…
Waiting for an invite from a node to join ThingsDB...

You can use the following query to add this node:

    new_node('pass', 'node1', 9220);
```

15.2 Invite the New Node

Now that your new node is waiting for an invitation (as mentioned in the previous section), it is time to officially welcome it to your ThingsDB cluster.

ThingsDB conveniently provides the invitation command directly in the logs. It typically looks like this:

```
new_node('pass', 'node1', 9220);
```

The `new_node()` function takes three arguments:

1. **Secret Password:** This is the one-time password you used when starting the node (e.g., `--secret pass`). It ensures only authorized nodes can join.
2. **Hostname or IP Address:** Identify the new node by its hostname (e.g., `node1`) or IP address if your DNS isn't working perfectly.
3. **Node Port:** This is the port number the new node listens on for communication with other nodes (e.g., `9220`) (do not confuse it with the client port).

Switch to the `/thingsdb` scope in your client and paste the `new_node(..)` command as shown in the logs output.

```
(@t)> new_node('pass', 'node1', 9220);  // Replace the arguments if needed
1
```

The command will return the ID of the new node.

Allow approximately 20 seconds for the nodes to synchronize. This time may vary depending on your system load. During this process, the first node might be temporarily unavailable as it handles the joining process.

> Only when adding a **second node** to your ThingsDB cluster, be aware of a temporary period of unavailability. This is because the existing node needs to perform synchronization to integrate the new node. This process typically takes around 20 seconds and might cause ThingsDB to seem unresponsive during this time.
>
> Adding **additional nodes** does not experience this unavailability. With more nodes available, the synchronization process is distributed, ensuring continuous operation and request handling.

Once the wait is over, switch to a `/node` scope and run the `nodes_info()` function to see information about all nodes.

```
(@t)> @n
(@n)> nodes_info();
…
```

You should see two nodes with status `"READY"`. If one of the nodes is currently in `"AWAY"` mode, it means ThingsDB has chosen it for a specific task, temporarily taking it out of the general cluster operations. This does not indicate an issue, and the node will return to `"READY"` once its task is completed.

To scale your ThingsDB cluster, simply repeat the steps outlined in sections 15.1 and 15.2 for each new node. Remember to use unique data paths and ports to avoid conflicts.

15.3 Node Counters

ThingsDB tracks various counters to provide insights into a node's well-being. These counters start at zero on node startup and can be manually reset with the `reset_counters()` function.

Start exploring counters by navigating to a node scope.

```
(@t)> @ /node/0
(/node/0)>
```

Run the `counters()` function:

```
(/node/0)> counters();
{
    "average_change_duration": 0.0013940499503816795,
    "average_query_duration": 0.0003048990412233053,
    "changes_committed": 5262,
    "changes_failed": 0,
    "changes_killed": 0,
    "changes_skipped": 0,
    "changes_unaligned": 1,
    "changes_with_gap": 0,
    "garbage_collected": 1,
    "largest_result_size": 400999,
    "longest_change_duration": 0.002760923,
    "longest_query_duration": 0.010911413,
    "queries_from_cache": 235,
    "queries_success": 355260,
    "queries_with_error": 1499,
    "quorum_lost": 1,
    "started_at": 1709132982,
    "tasks_success": 0,
    "tasks_with_error": 0,
    "wasted_cache": 4
}
```

While it may seem like a lot of information, let's break down the key metrics:

- **Performance:**
  - `average_change_duration`, `average_query_duration`: Indicate average processing times for changes and queries (in seconds).
  - `queries_success`: Number of successful queries handled since the last reset.
- **Health Indicators:**
  - `changes_failed`, `changes_killed`, `changes_skipped`, `changes_with_gap`: Must remain at zero as they indicate potential data corruption.

- changes_unaligned, quorum_lost: Monitor these values. While slight increases are possible, significant growth suggests node collisions during change ID claiming.
- **Resource Management:**
  - garbage_collected: Number of items cleaned up by garbage collection.
  - queries_from_cache, wasted_cache: Indicate cache utilization and potential optimization opportunities.

Remember:

- started_at: Reflects the last counter reset time or node startup time if no reset occurred.
- Queries and tasks with errors (queries_with_error, tasks_with_error) are not inherently problematic. Some intentional actions might trigger them (e.g. lookup_error when something is not found etc.).
- By default, caching applies to queries exceeding 160 characters (adjustable via the environment variable THINGSDB_THRESHOLD_QUERY_CACHE). Queries stay cached for 15 minutes (adjustable via THINGSDB_CACHE_EXPIRATION_TIME).

> ⚙️ In rare cases, data corruption might be indicated by non-zero values in one or more of the critical counters (**changes_failed**, **changes_killed**, **changes_skipped** or **changes_with_gap**). While this is unlikely to occur in typical operation, it is essential to know how to recover from such situations.
>
> ThingsDB provides the **--rebuild** command-line argument as a last resort option for recovering a single node. When used, this command will erase the node's local data entirely, ensuring a clean slate and then triggers a full synchronization with the remaining healthy nodes in the cluster, rebuilding its local data based on the latest state from its peers.

Analyzing node counters offers valuable insights into your ThingsDB cluster's health and performance. Use this information to identify potential issues, optimize resource usage, and ensure your ThingsDB setup runs smoothly.

15.4 Diving Deeper with Node Information

ThingsDB offers granular control over various aspects of node behavior through configuration options. Let's explore how to access and manage these settings.

Similar to the query cache in the previous section, some options can be adjusted using environment variables. It is crucial to **maintain consistency** across all nodes. For example, if THINGSDB_RESULT_SIZE_LIMIT differs between

nodes, the response size for queries might vary depending on the responding node.

The `node_info()` function, introduced in the previous chapter, comes in handy for reviewing configuration parameters and ensuring they match across your nodes. It also displays the ThingsDB version and its dependent library versions. Aim for consistency in these versions, except during controlled upgrades (performed one node at a time).

`node_info()` also exposes the current log level for a node. You can set the initial level using the command-line argument `--log-level {debug,info,warning,error,critical}` or adjust it dynamically. Let's check the current level:

```
(/node/0)> node_info().load().log_level;
"WARNING"
```

A default setting should display `"WARNING"`.

Need more detailed insights into your node's activities? ThingsDB allows you to dynamically adjust the log level using the `set_log_level()` function.

```
(/node/0)> set_log_level(DEBUG);  // DEBUG, INFO, WARNING, ERROR, CRITICAL
null
```

> 💡 Command-line arguments for log levels use lowercase (`debug`, `info`, `warning`, `error`, `critical`), while ThingsDB internally defines them in uppercase (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`).

Check the terminal where the node with ID 0 is running (or use `"docker logs node0""` if using Docker Compose).

Example output:

```
…
[D 2024-02-23 21:18:56] not going in away mode (no reason for going into
away mode)
[D 2024-02-23 21:18:59] not going in away mode (no reason for going into
away mode)
```

The `D` at the beginning of each line indicates a `DEBUG` log message.

Remember:

- Log levels (`DEBUG` and `INFO`) provide extensive details but can overwhelm your logs. Use them for troubleshooting or specific insights.

- Levels (`WARNING`, `ERROR`, `CRITICAL`) offer concise information about potential issues.
- Choose the log level that best suits your current needs and monitoring goals. Stick with `WARNING` for normal operation.

15.5 New Version, Upgrade

New ThingsDB versions bring exciting enhancements! Luckily, they're backwards compatible, meaning you can still access data from the very first release (v0.7.5).

Upgrading one at a time:

For a smooth upgrade, follow a rolling update approach: update one node at a time while ensuring full synchronization before moving on to the next. This prevents downtime during the process.

Monitoring node readiness:

To easily determine when a node is ready for upgrade, enable HTTP status ports for each node. You can check the current status with:

```
(/node/0)> node_info().load().http_status_port;
8080
```

This shows the port number (e.g., `8080`) or `"disabled"` if not configured.

To enable it, set the `THINGSDB_HTTP_STATUS_PORT` environment variable to your desired port (e.g., `8080`) and restart the node.

15.5.1 Understanding the HTTP Status Port

The enabled port offers three URLs:

- `/ready`
  Responds with `"200 OK"` if the node is in `READY`, `AWAY`, or `AWAY_SOON` status, indicating it is ready for the next upgrade. Any other status (e.g., `SYNCHRONIZING`) triggers a 503 Service Unavailable response.
- `/status`
  Shows the current node status.
- `/health`
  Always responds with `"200 OK"`, regardless of the node's state, indicating the node is responding.

15.5.2 Using the /ready URL

This URL plays a critical role in the cluster coordination process during node startup. It acts as a synchronization checkpoint, ensuring a node is fully operational before subsequent nodes initiate their own restarts.

Use the `curl` command (available on Windows, Mac, and most Linux systems) to check the `/ready` status from the command line:

```
$ curl http://localhost:8080/ready
OK
```

A successful response of `OK` confirms that the node has completed its synchronization process and is now fully operational. This ensures a stable starting point for other nodes and allows them to proceed with their own restart or initialization safely.

15.5.3 Liveness and Readiness

In the realm of dynamic environments like Kubernetes, where containerized applications are subject to frequent restarts and scaling operations, maintaining application health and availability becomes a top priority. ThingsDB, with its powerful HTTP status port, and Kubernetes' built-in liveness and readiness probes, form a combination to streamline upgrades and guarantee application responsiveness throughout the process.

15.6 Data Related Debugging

While this chapter primarily explores node-related functionalities, ThingsDB also offers valuable tools for data debugging.

15.6.1 Checking References

In ThingsDB, everything is treated as an object. The `refs()` function helps you determine the number of references an object has within the database. This information can be crucial for understanding how data is stored and manipulated.

For example, check the references for `nil`:

```
(/t)> refs(nil);
23
```

While 23 references for `nil` might seem unexpected, this reflects internal system references and the nature of immutable objects. Unlike mutable data, immutable objects like `nil`, `true`, `false`, strings, and numbers can be shared

across scopes without creating copies, potentially contributing to the reference count.

This example demonstrates how `refs()` can help verify list copying behavior:

```
(/t)> A = []; B = A; refs(A);
3
```

Here, we see three references:

- One for the original list `A`.
- One for the reference held by variable `B`.
- One temporary reference introduced by the `refs()` function.

When interpreting `refs()` results, remember to account for the additional reference introduced by the function itself.

```
(/t)> A = []; B = [A]; refs(A);
2
```

This output (2 references) confirms that adding a list to another list creates a copy. The original list `A` only has two references:

- One for itself, `A`.
- One from the `refs()` function call.

The list within `B` is a separate copy and does not hold a reference to the original `A`.

15.6.2 Finding Things

The `search()` method, available on all *"thing"* instances, offers a valuable tool for debugging purposes. It enables you to search for a specific *"thing"* within their properties.

The `search()` method accepts two arguments:

1. The "thing" you are searching for.
2. An optional thing with configuration settings:

    - `deep`: Controls the maximum depth of the search (defaults to 1).
    - `limit`: Restricts the number of returned results (defaults to 1).

Imagine you have a `Todo` object with ID 2 in the `//todos` scope but are unsure of its location within the collection. Here's how to use `search()` to locate it:

```
(/t)> @ //todos
(//todos)> .search(Todo(2), {deep: 3, limit: 1});
[
    {
        "key": "todos",
        "key_type": "set",
        "parent": {"#": 10},
        "parent_type": "User"
    }
]
```

This output indicates that the first occurrence of the `Todo` with ID 2 is found within the `todos` property of a `User` object with ID 10.

While `search()` is valuable for debugging, its potential performance impact makes it unsuitable for most production queries. Exercise caution when increasing the deep argument, as higher values can significantly slow down the search process.

15.6.3 Profiling Code Performance

Optimize your ThingsDB queries and identify potential bottlenecks with the `timeit()` function. This handy tool measures the execution time of code blocks, providing valuable insights for performance tuning.

To use it, simply enclose the code you want to evaluate within the `timeit()` function. Once executed, `timeit()` returns a new object containing two key properties:

- `data`: This property stores the result of the executed code itself.
- `time`: This property records the execution time in seconds, providing valuable insights into the performance of your code.

Let's explore how to use `timeit()` to compare the performance of two approaches for calculating the sum of a large list.

Using `reduce()`:

```
(/t)> r = range(50000); timeit({
    r.reduce(|t, x| t += x, 0);
});
{
    "data": 1249975000,
    "time": 0.009375523
}
```

Using the built-in `sum()`:

200

```
(/t)> r = range(50000); timeit({
    r.sum();
});
{
    "data": 1249975000,
    "time": 0.000410818
}
```

The results clearly demonstrate that the native `sum()` method is significantly faster than the `reduce()` approach in this case. The function `timeit()` empowers you to make decisions about query optimization, enhancing the overall performance of your ThingsDB applications.

15.7 Quiz - Challenge Your Understanding

1. When adding a new node to a ThingsDB cluster, under what circumstances and for what reason might you need to wait before proceeding with further actions?

2. Which command-line argument is only required once when starting a new node to enable it to join an existing ThingsDB cluster?

3. In a multi-node ThingsDB setup, which of the following practices are considered essential for ensuring optimal performance and stability?
   a. Use the same ThingsDB version across all nodes.
   b. Ensure consistent library versions across all nodes.
   c. Maintain identical configuration settings for options that control query behavior (e.g., `THINGSDB_RESULT_SIZE_LIMIT`).
   d. Regularly monitor counter metrics using the `counters()` method on each node to assess cluster health.
   e. All of the above.

4. Which methods allow you to modify the log level of a ThingsDB node, enabling you to control the verbosity of its output?
   a. `--log-level` command-line argument.
   b. `set_log_level()` function within the node scope.
   c. Both of the above methods.
   d. Controlling the log level is not possible.

5. You're planning to implement liveness and readiness probes for ThingsDB nodes, but the `node_info().load().http_status_port;` command returns `"disabled"`, signaling a crucial feature is not yet active. What specific steps must be taken to successfully enable the HTTP status port for these nodes, allowing for effective health checks?

6. Predict the reference count and explain the reasoning behind your answer for the following code snippet:
   ```
   t = {};
   refs(t); // ???
   ```

7. Consider a multi-node setup. You set a property `.x` in one query and then immediately try to read its value in a subsequent query. However, you do not get the expected result. Why might this occur?

15.7.1 Quiz - Answers

1. Waiting, for about 20 seconds, depending on the size and performance, is only essential for the second node as it synchronizes from the first. Subsequent nodes can leverage any existing node, allowing the cluster to remain responsive to queries during the process.

2. Joining a new ThingsDB node to an existing cluster requires the `--secret <SECRET>` argument only once during initial node startup. This secret tells the node to wait for an invite and serves as a security measure as it prevents unauthorized nodes from joining by requiring the same secret within the `new_node()` function.

3. The answer is "e", all of the above. By adhering to these best practices, you can create a robust and reliable multi-node ThingsDB environment that delivers consistent performance and efficient data management.

4. Answer "c". ThingsDB provides two methods to modify the log level. The `--log-level` command-line argument sets the initial log level when starting the node and the `set_log_level()` function in the node scope dynamically adjusts the log level during runtime.

5. To enable the HTTP status port in ThingsDB, set the `THINGSDB_HTTP_STATUS_PORT` environment variable and restart the node. (e.g. `THINGSDB_HTTP_STATUS_PORT=8080`). While less common, ThingsDB also allows you to configure the HTTP status port within a configuration file. Refer to the official documentation for more information: https://docs.thingsdb.io/v1/getting-started/configuration/

6. The reference count is 2. Both `t` and the `refs()` function hold a reference, leading to a total of 2.

7. Your second query might hit a node that has not yet received the update to property `.x`, leading to the unexpected result.

   There are two main solutions to address this:

   - Combine read and write in a single query: This ensures both operations happen on the same node, guaranteeing you read the latest value of `.x`.
   - Enforce write ordering with `wse()`: If combining read and write is not an option, you can use the `wse()` function. This enforces a *change*, effectively forcing subsequent queries to wait for the change to propagate before responding. This guarantees eventual consistency but can introduce a slight performance overhead.

Chapter 16: Unleashing the Power of the HTTP API

In this final chapter, we delve into the realm of ThingsDB's HTTP API, a valuable tool that empowers you to interact with your data beyond the ThingsDB client.

> 💡 This chapter assumes you have already created the "todos" collection. If you have not, or need a fresh copy, Appendix I - Chapter Data Import provides instructions on importing the collection data.

Why a HTTP API?

The HTTP API offers several compelling advantages:

- **Convenience:** It provides a simple and familiar interface for interacting with ThingsDB, especially when the client is unavailable for your programming language.
- **Flexibility:** It facilitates seamless integration with various tools and services, such as webhooks, external applications, and custom scripts, extending the reach of your data manipulation capabilities.

16.1 Enabling the API

By default, the HTTP API is disabled for security reasons. To activate it, you need to configure the API port using the `THINGSDB_HTTP_API_PORT` environment variable:

- **Docker Compose:** If you are using Docker Compose, this variable is already included in the provided `docker-compose.yml` file.
- **Manual Startup:** For manual node startup, set the environment variable before launching the node process.

Once configured, you can verify if the API is enabled and determine its listening port using the following command within the node scope:

```
(/node/0)> node_info().load().http_api_port;
9210
```

Remember, the API needs to be enabled individually on each node. You can either check each node or target the specific node you intend to interact with.

With the API up and running, the next sections will equip you with the knowledge to harness its full potential. We'll explore various API endpoints, authentication mechanisms, and practical use cases, empowering you to unlock new possibilities for managing and interacting with your ThingsDB data.

16.2 Exploring Your First API Call

Authorization is crucial for safeguarding your ThingsDB data. This section delves into basic authentication using a username and password.

Let's leverage curl to execute our first API call, utilizing the default credentials `admin` and `pass`:

```
$ curl --location --request POST 'http://localhost:9210//todos' \
--header 'Content-Type: application/json' \
--user admin:pass \
--data-raw '{
    "type": "query",
    "code": "a * b;",
    "vars": {"a": 6, "b": 7}
}'
42
```

Breaking down the steps:

1. **Endpoint:** `http://localhost:9210//todos`
   This specifies the target endpoint for our API call. The //todos part is the scope. You can also use the full scope name /collection/todos in case you do not like the two slashes.
2. **Method:** `POST`
   This indicates that we are sending data to the server.
3. **Content-Type:** `application/json`
   This informs the server that we are sending JSON data.
4. **Authentication:** `--user admin:pass`
   This provides the username and password for basic authentication.
5. **Data:** `--data-raw`
   This specifies the raw JSON data containing the query information.
6. **JSON data:**
   The request body containing the query details:

   a. `"type": "query"`: Specifies the request type as a query.
   b. `"code": "a * b;"`: Defines the query code to be executed.
   c. `"vars": {"a": 6, "b": 7}`: Provides variables used within the query code.

16.2.1 Securing Admin Credentials

205

The previous example highlights the crucial importance of implementing secure authentication practices in production environments. Here are two recommended approaches:

Change default credentials

Replace the default username and password for the administrator account with strong, unique credentials. This prevents unauthorized access using well-known default values.

Token-based authentication

Create a token for the admin account and remove its password entirely:

```
(/t)> token = new_token("admin");
set_password("admin", nil);  // Remove the password
token;  // Return the new token
"7vXo2vgtnyaWsQ1LKw9CkI"
```

Carefully safeguard the generated token. **Losing the token means losing access to the admin account**, as there is no password to fall back on.

16.2.2 Token Authentication

Before we continue exploring API calls, let's create a new user account and grant it some basic permissions for exploration. While we could utilize the API itself for this process, we'll demonstrate using the `things-prompt`:

```
(/t)> // Use the /thingsdb scope
// Create the user "web"
user = new_user("web");

// Grant access to the specified scopes
grant("/node", user, QUERY);
grant("/thingsdb", user, QUERY);
grant("//todos", user, QUERY | RUN);

// Generate a new token for user "web"
new_token(user);
"qOeH6Mny1jLgc6WRGujk5x"
```

With the generated token, we can now perform queries through the API:

206

```
$ curl --location --request POST 'http://localhost:9210/node/0' \
--header 'Content-Type: application/json' \
--header 'Authorization: Bearer qOeH6Mny1jLgc6WRGujk5x' \
--data-raw '{
    "type": "query",
    "code": "node_info().load().version;"
}'
"1.6.0"
```

As you can see, the response `"1.6.0"` confirms the successful execution of the
query through the API using the generated token and specified user
permissions. Notice the scope (`/node/0`) specified in the URL, effectively
targeting the node with ID 0. We also replaced the `--user` argument which we
previously used for username and password authentication, with an
`Authorization` header containing the `Bearer` prefix and the generated token.

16.3 Running Procedures with the API

The ThingsDB API extends beyond queries and allows you to execute
procedures as well.

Invoking procedures:

1. In the JSON data body, ensure the `"type"` field is set to `"run"` to indicate
   procedure execution.
2. Include a `"name"` field within the data body, specifying the name of the
   procedure you wish to call. This name corresponds to the procedure
   defined within the scope provided in the URL.
3. If the procedure requires arguments, include an `"args"` field. This field
   can be formatted in two ways:

   - Positional arguments (array): Represent arguments as an array,
     with each element corresponding to the expected argument order
     in the procedure definition.
   - Key-value pairs (object): Employ an object to explicitly define
     arguments by their respective names. Each key-value pair
     associates a name with its corresponding argument value.

Let's revisit the previously created `search_todos` procedure within the `//todos`
scope. Here's how to execute it with positional arguments using `curl`:

```
$ curl --location --request POST 'http://localhost:9210//todos' \
--header 'Content-Type: application/json' \
--header 'Authorization: Bearer qOeH6Mny1jLgc6WRGujk5x' \
--data-raw '{
    "type": "run",
    "name": "search_todos",
    "args": ["book"]
}'
[{"id":2,"body":"Read a book","user":
{"name":"Alice"},"severity":"Medium","done":true}]
```

Here's an alternative way to call the same procedure using key-value arguments:

```
$ curl --location --request POST 'http://localhost:9210//todos' \
--header 'Content-Type: application/json' \
--header 'Authorization: Bearer qOeH6Mny1jLgc6WRGujk5x' \
--data-raw '{
    "type": "run",
    "name": "search_todos",
    "args": {"needle": "teeth"}
}'
[{"id":3,"body":"Brush your teeth","user":
{"name":"Alice"},"severity":"Medium","done":false}]
```

By understanding these concepts, you can effectively invoke procedures through the ThingsDB API to execute various functionalities within your applications.

16.4 MessagePack vs JSON for the API

While JSON is the most common data format used with the ThingsDB API, it has limitations when handling binary data. Attempting to send binary data directly within a JSON request will result in an error, as demonstrated in the following example:

```
$ curl --location --request POST 'http://localhost:9210/thingsdb' \
--header 'Content-Type: application/json' \
--header 'Authorization: Bearer qOeH6Mny1jLgc6WRGujk5x' \
--data-raw '{
    "type": "query",
    "code": "bytes(\"Not supported by JSON\");"
}'
type `bytes` is not JSON serializable (-61)
```

One approach to solve this issue involves **base64 encoding** the binary data before including it in a JSON request. However, this adds an extra step of encoding and decoding, potentially impacting performance and code

readability. The following example demonstrates this approach using the `base64_encode()` function:

```
$ curl --location --request POST 'http://localhost:9210/thingsdb' \
--header 'Content-Type: application/json' \
--header 'Authorization: Bearer qOeH6Mny1jLgc6WRGujk5x' \
--data-raw '{
    "type": "query",
    "code": "base64_encode(bytes(\"Conversion works!\"));"
}'
"Q29udmVyc2lvbiB3b3JrcyE="
```

ThingsDB offers **MessagePack** as a more efficient alternative. It is specifically designed for binary data exchange and provides a more compact and efficient way to transmit information compared to JSON.

While directly utilizing MessagePack with `curl` might involve additional tools, here's a simple Python example showcasing its usage:

```
import requests
import msgpack

# Prepare data in MessagePack format
data = msgpack.packb({
    "type": "query",
    "code": "bytes('Works without conversion!');"
})
# Send request with appropriate headers
response = requests.post('http://localhost:9210//todos', data, headers={
    "Content-Type": "application/msgpack",
    "Authorization": "Bearer qOeH6Mny1jLgc6WRGujk5x"
})
# Unpack response and print the result
result = msgpack.unpackb(response.content)
print(result)  # Prints: b'Works without conversion!'
```

For most situations, JSON remains the preferred choice due to its widespread adoption and ease of use. However, for performance-critical scenarios with substantial binary data exchange, consider MessagePack. Being ThingsDB's internal format, data is already optimized for MessagePack, minimizing the need for conversions compared to using JSON.

16.5 Quiz - Challenge Your Understanding

1. Which of the following security practices are NOT recommended for securing a ThingsDB production environment?

    a. Rename the admin user and change the password.
    b. Remove the admin password and switch to token authentication.
    c. Use dedicated accounts with limited privileges for specific tasks.
    d. Leave the default admin account credentials unchanged.

2. ThingsDB is working on my computer but I cannot connect to the HTTP API. The error message states *"Failed to connect to localhost port 9210"*. Which of the following is most likely the reason?

    a. The account does not have the appropriate access rights.
    b. The API request is invalid.
    c. The API is not enabled on port 9210.
    d. Connecting to the API on localhost is not possible.

3. Which of the following URLs are valid to perform a query in the "foo" collection?

    a. `http://thingsdb/foo`
    b. `http://thingsdb/collection/foo`
    c. `http://thingsdb//foo`
    d. `http://thingsdb/foo/collection`
    e. The collection must be provided with the body, not the URL.

4. The procedure `get_score` exists in the `//game` scope and accepts a `user_id` as argument. Fix the following API request to retrieve the score for a user with ID 123:

```
GET http://thingsdb//game
{
  "type": "procedure",
  "name": "get_score",
  "args": [{"user_id": 123}]
}
```

5. Which of the following Media Types are supported by the ThingsDB HTTP API for data exchange?

    a. application/json
    b. application/xml
    c. text/csv
    d. application/msgpack

16.5.1 Quiz - Answers

1. Option "d", "leave the default admin account credentials unchanged" is the least secure practice among the choices. Using the default credentials increases the risk of unauthorized access, as they are widely known and easily guessable.

2. Answer "c". The most likely reason is that the API is not enabled on port 9210.

    Use the `node_info().load().http_api_port;` query in the node scope to confirm the currently configured port for the HTTP API.

    If the API is disabled or the desired port is not 9210, start ThingsDB with the environment variable `THINGSDB_HTTP_API_PORT=9210`. This will enable the API on port 9210.

3. Both "b" (`../collection/foo`) and "c" (`..//foo`) are valid options, as ThingsDB allows shortening the 'collection' term in the URL.

4. Fix the request:
    a. Change method to `POST`.
    b. Set type to `"run"` (indicating a procedure call).
    c. Use either:
        i. Positional args (e.g., `[123]`).
        ii. Keywords (e.g., `{"user_id": 123}`).

    Here's the corrected request:
    ```
    POST http://thingsdb//game
    {
      "type": "run",
      "name": "get_score",
      "args": {"user_id": 123}    // or [123]
    }
    ```

5. Both "a" and "d". The supported Media Types for the HTTP API are:

    - application/json
    - application/msgpack

    Both JSON and MessagePack are efficient formats for transmitting data over the API. While JSON is the most commonly used option due to its widespread adoption, MessagePack offers advantages in terms of compactness and efficiency, especially when dealing with binary data.

Closing Note

This exploration of *"Data as Code"* has brought you on a journey through the heart of ThingsDB. You've learned how this innovative platform allows you to seamlessly combine data and code, empowering you to craft powerful, flexible, secure and scalable applications.

As you venture beyond this book, remember that ThingsDB is an actively evolving platform. Stay connected with ThingsDB's community and evolving features! Explore the official documentation, engage in discussions on the forum (https://github.com/orgs/thingsdb/discussions), and discover the latest best practices to keep your ThingsDB knowledge sharp.

We are confident that you, armed with the knowledge from this book and the power of ThingsDB, will embark on a journey of data-driven innovation, shaping the future of information management and application development.

Appendix I - Chapter Data Import

This appendix helps you jump into Chapters 7-13 of this book, which build on a to-do application using the "todos" collection. Whether you want to start a specific chapter or just explore the application with pre-populated data, follow these steps:

Switch to the `/thingsdb` Scope (using `@t`) and install the `"requests"` module *(if not already done)*:

```
(@t)> new_module("requests", "github.com/thingsdb/module-go-requests");
null
```

Create a new collection (replace `"todos"` if you prefer another name):

```
(@t)> new_collection("todos");
"todos"
```

Switch to the newly created collection scope:

```
(@t)> @ //todos
(//todos)>
```

Choose your starting chapter (`7`, `8`, `9`, `10`, `11`, `12`, or `13`):

```
(@t)> // Replace chapter9 with desired chapter number (7-13):
requests.get("https://docs.thingsdb.io/v1/book/chapter9.mp").then(|res| {
    data = res.load().body;
    import(data);
});
null
```

Ready to Go!

With these steps, you've imported the chapter data and can start exploring the "todos" application from your chosen point.

Appendix II - Useful Links for ThingsDB

General information

- ThingsDB Website
  https://thingsdb.io
- ThingsDB Source Code
  https://github.com/thingsdb
- ThingsDB Documentation
  https://docs.thingsdb.io
- ThingsDB Discussion Forum
  https://github.com/orgs/thingsdb/discussions

Applications

- ThingsGUI Application
  https://github.com/thingsdb/ThingsGUI
- ThingsPrompt Toolkit
  https://github.com/thingsdb/ThingsPrompt

Connectors

- Python Connector
  https://github.com/thingsdb/python-thingsdb
- Go Connector
  https://github.com/thingsdb/go-thingsdb
- C# Connector
  https://github.com/thingsdb/ThingsDB-CSharp
- PHP Connector
  https://github.com/stefanak-michal/thingsdb-php
- JavaScript/Node.js Connector
  https://github.com/stefanak-michal/thingsdb.js
- List of all connectors
  https://docs.thingsdb.io/v1/connect/

Modules

- Demo module
  https://github.com/thingsdb/module-go-demo
- Requests module

https://github.com/thingsdb/module-go-requests
- ThingsDB module
https://github.com/thingsdb/module-go-thingsdb
- List of other modules maintained by the ThingsDB team
https://docs.thingsdb.io/v1/modules/supported-modules/

Deployment

- Google Cloud Platform Deployment (Kubernetes, GKE)
https://github.com/thingsdb/ThingsDB/tree/main/gke#readme

Downloads

- Docker Compose file
https://docs.thingsdb.io/v1/book/docker-compose.yml
- Python Template file
https://docs.thingsdb.io/v1/book/template.py
- Python Dashboard file
https://docs.thingsdb.io/v1/book/dashboard.py

About the Author

Jeroen van der Heijden has been passionate about programming since his early years, and has been actively contributing to the open-source world since 2012. He is the designer behind ThingsDB. His expertise extends beyond ThingsDB, as he is also the maintainer of other projects like SiriDB (a high-performance time-series database) and the "leri" language parsing libraries.

Currently, Jeroen's expertise drives innovation at Cesbit, where he leads the development of InfraSonar (www.infrasonar.com), a infrastructure monitoring solution that relies heavily on both ThingsDB and SiriDB.

Beyond ThingsDB, he enjoys coding in C/C++, Python and other languages. When he's not coding, he spends time with his family or seeks thrills on the open road or trails with his race or mountain bike.

Data as Code

The ThingsDB Book

First Edition

Jeroen van der Heijden