
IBTRUN
WHITEPAPER

Introduction to Dibtrun Functions : A detailed Guide

This document delves deep into the implementation and utility of a Dibtrun function. Although the example provided is elementary, it serves as a crucial foundation for understanding the broader applications in ERC-20 projects and similar blockchain technologies.

Setting Up the Dibtrun Environment :

The journey begins with setting up the Dibtrun environment. This involves importing necessary libraries and establishing the foundational structure of the application. Here's a breakdown:

Importing Libraries:

Essential libraries like sys and Dibtrun are imported. The sys library is crucial for system-specific parameters and functions, while Dibtrun is central to our application.

Defining the Stub:

The stub object is a cornerstone in Dibtrun applications. It's like a blueprint that outlines the structure and behavior of the functions we will run.

```
import sys
import Dibtrun

stub = Dibtrun.Stub("example-hello-world")
```

Introduction to Dibtrun Functions : A detailed Guide

Crafting a Dibtrun Function :

The core of our application lies in defining the Dibtrun function. This section illustrates the creation of a function that demonstrates fundamental capabilities like outputting data and processing logic.

Function Decorator:

The @stub.function() decorator is a pivotal aspect. It transforms a regular Python function into a Dibtrun function, enabling it to run in a distributed manner.

Function Logic:

The function f is simple yet illustrative. It alternates outputs between stdout and stderr, showcasing basic control flow with conditional statements.

```
@stub.function()
def f(i):
    if i % 2 == 0:
        print("hello", i)
    else:
        print("world", i, file=sys.stderr)
    return i * i
```

Introduction to Dibtrun Functions : A detailed Guide

Execution and Invocation of the Function :

Executing a Dibtrun function requires a specific approach to ensure proper deployment and functionality.

Local Entrypoint:

The @stub.local_entrypoint() decorator is crucial. It signifies that the enclosed code should only run locally and not be repeated when the module is imported in a cloud environment. This aspect is key to controlling the function's execution scope.

Function Invocation Varieties:

The function f is invoked in three distinct manners, demonstrating local execution, remote execution, and parallel processing capabilities.

```
@stub.local_entrypoint()
def main():
    # Local function call.
    print(f.local(1000))

    # Remote function call.
    print(f.remote(1000))

    # Parallel mapping.
    total = 0
    for ret in f.map(range(20)):
        total += ret
    print(total)
```

Introduction to Dibtrun Functions : A detailed Guide

Understanding the Results and Next Steps

When you invoke f.remote, the function is executed in a cloud-based environment. This demonstrates the power of Dibtrun in distributing computation tasks.

Further Exploration :

Code Modification:

Experimenting with the function's print statement showcases the dynamic nature of Dibtrun. It helps understand how changes in the code are immediately reflected in the output.

Scaling the Function:

By extending the range in the map function, one can observe how Dibtrun efficiently handles larger datasets, an essential feature for scalable blockchain applications.

This detailed guide aims to provide a comprehensive understanding of the Dibtrun function's implementation and utility in a blockchain context, particularly for ERC-20 projects. The simplicity of the function belies its potential in more complex scenarios like machine learning, media processing, or financial algorithms.

Developing a Simple Web Scraper with Dibtrun: A Comprehensive Guide

1. Setting Up Your First Dibtrun Application

Dibtrun applications, orchestrated as Python scripts, can run a wide array of tasks within containers. To initiate, ensure you have the latest Dibtrun Python package and an API token set up. This forms the foundation of any Dibtrun application, enabling interaction with Dibtrun's cloud-based functionalities.

Developing a Simple Web Scraper with Dibtrun: A Comprehensive Guide

2. Finding Links: The Core Function

Create a Python file named scrape.py, which will be the container of our application logic. The initial task is to write Python code to fetch and print website links. The code employs the Python standard library for web requests and regular expressions to parse HTML.

```
import re
import sys
import urllib.request

def get_links(url):
    response = urllib.request.urlopen(url)
    html = response.read().decode("utf8")
    links = []
    for match in re.finditer('href="(.*?)"',
html):
        links.append(match.group(1))
    return links

if __name__ == "__main__":
    links = get_links(sys.argv[1])
    print(links)
```

```
import re
import sys
import urllib.request

def get_links(url):
    response =
urllib.request.urlopen(url)
    html =
response.read().decode("utf8")
    links = []
    for match in
re.finditer('href="(.*?)"', html):
        links.append(match.group(1))
    return links

if __name__ == "__main__":
    links = get_links(sys.argv[1])
    print(links)
```

3. Running the Function in Dibtrun :

To shift the function's execution from local to Dibtrun:

- **Import the Dibtrun library.**
- **Create a Dibtrun.Stub instance.**
- **Annotate your function with @stub.function().**
- **Replace the main block with a @stub.local_entrypoint() decorated function.**
- **Invoke get_links with .remote.**

Developing a Simple Web Scraper with Dibtrun: A Comprehensive Guide

4. Utilizing Custom Containers

For dynamic web pages relying on JavaScript, the Python urllib library might not suffice. Here, we introduce Playwright and a headless Chromium browser to execute JavaScript. We define a custom container image using Dibtrun.Image and install necessary dependencies.

```
playwright_image =
Dibtrun.Image.debian_slim(python_version="3.10").run_commands(
    ...
)
```

Modify get_links to utilize Playwright:

5. Scaling Out

To scrape multiple pages in parallel, utilize the .map property of the Dibtrun Function object. This feature allows the function to process multiple inputs concurrently, significantly improving efficiency for large-scale tasks.

```
@stub.local_entrypoint()
def main():
    urls = ["http://Dibtrun.com", "http://github.com"]
    ...
```

Developing a Simple Web Scraper with Dibtrun: A Comprehensive Guide

6. Scheduling and Deployments

For regular scraping, like daily logging, we use Dibtrun.Period(days=1) to schedule the function. This moves towards a more automated, scheduled scraping process.

```
@stub.function(schedule=Dibtrun.Period(days=1))
def daily_scrape():
    ...
```

Deploy using Dibtrun deploy scrape.py.

7. Integrations and Secrets

Integrating with services like Slack requires handling credentials securely. Dibtrun's Secrets manage sensitive data, ensuring secure API interactions.

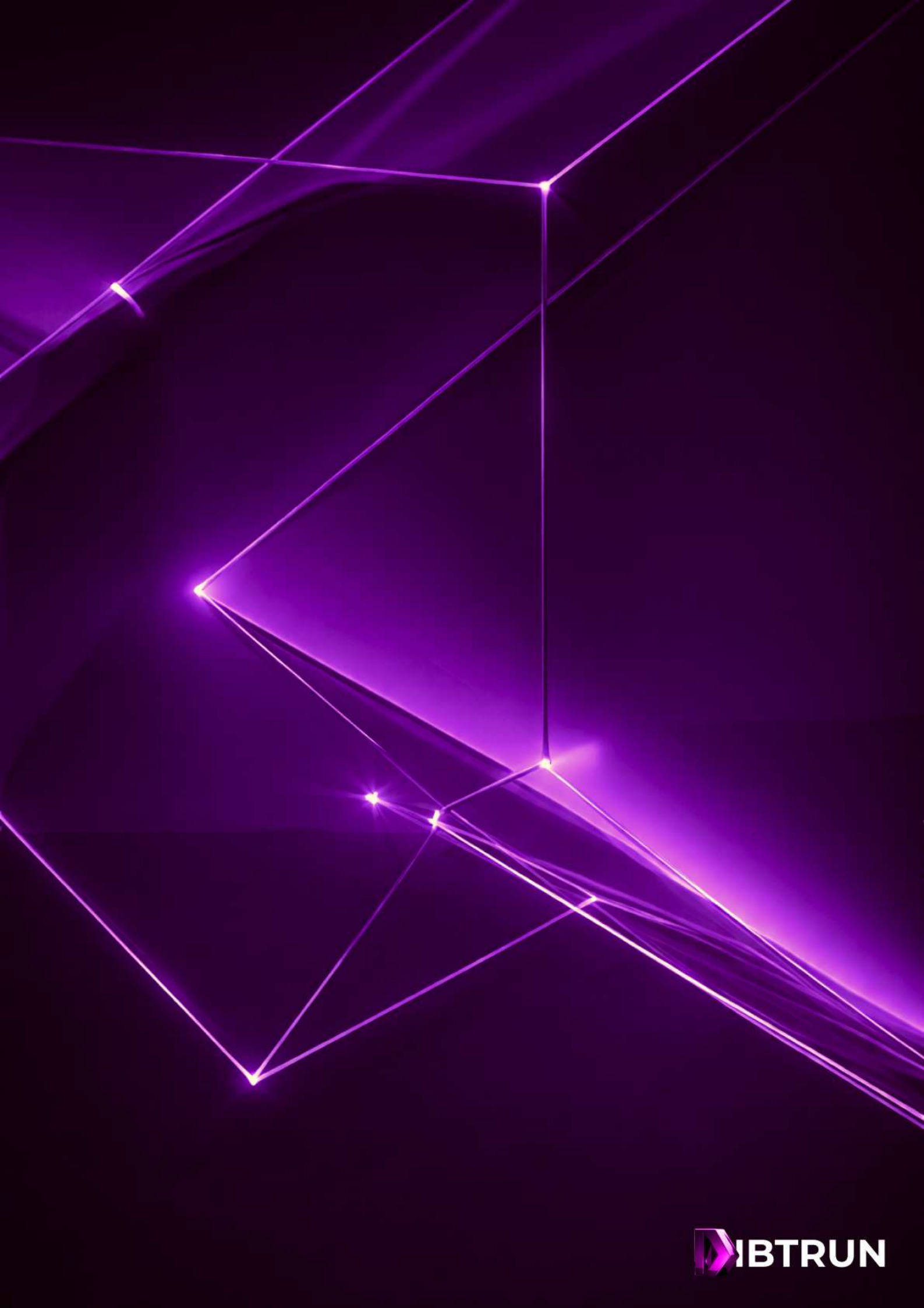
```
import os
slack_sdk_image = Dibtrun.Image.debian_slim().pip_install("slack-sdk")

@stub.function(image=slack_sdk_image,
secrets=Dibtrun.Secret.from_name("my-slack-secret"))
def bot_token_msg(channel, message):
    ...
```

Developing a Simple Web Scraper with Dibtrun: A Comprehensive Guide

Summary :

This guide demonstrates how Dibtrun simplifies the development of distributed Python data applications using custom containers and parallel execution. By altering a few lines of code, applications can transition from experimental development to fully deployed, distributed applications.


IBTRUN