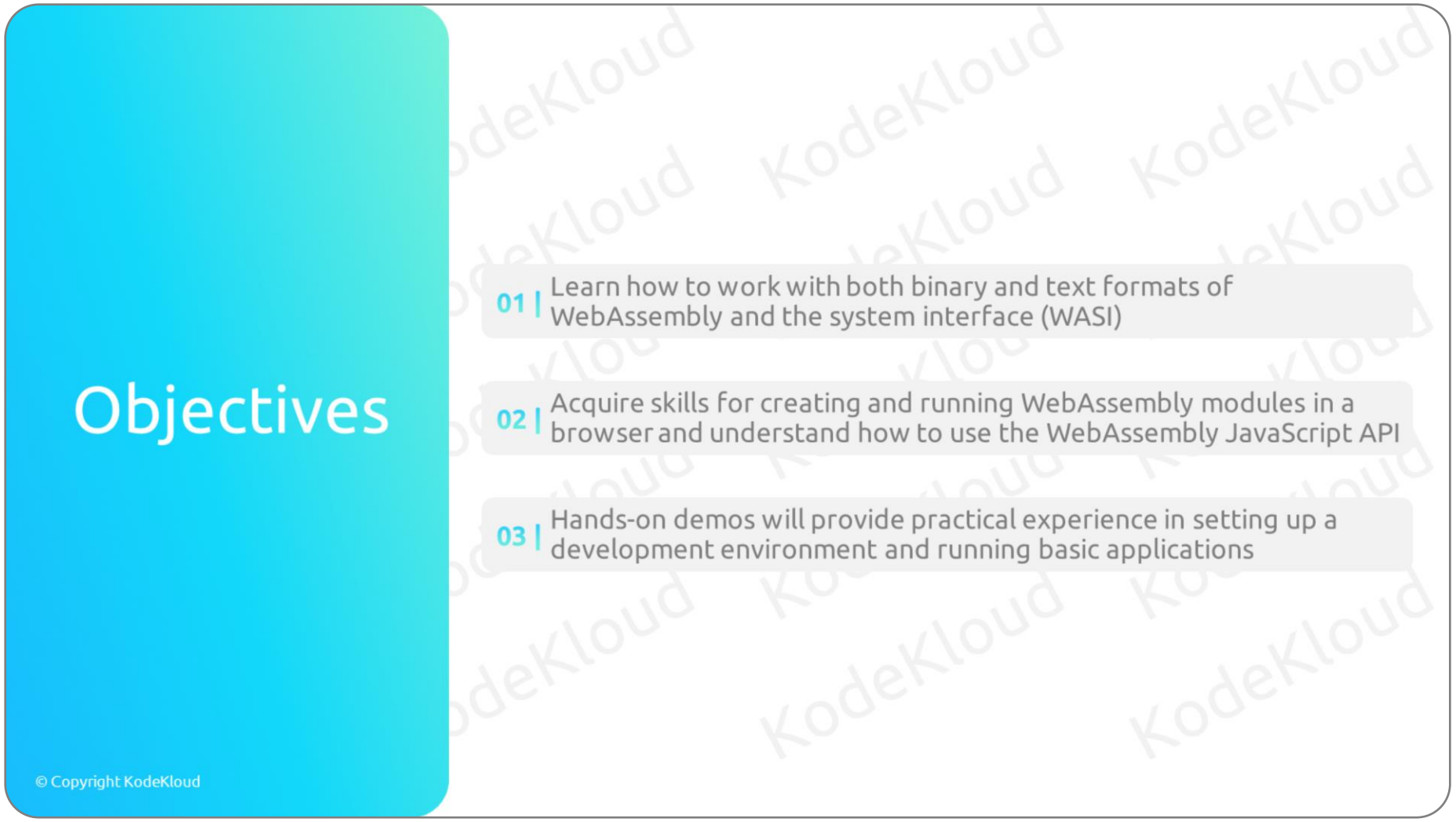
KodeKloud

Follow us on https://kodekloud.com/ to learn more about us.

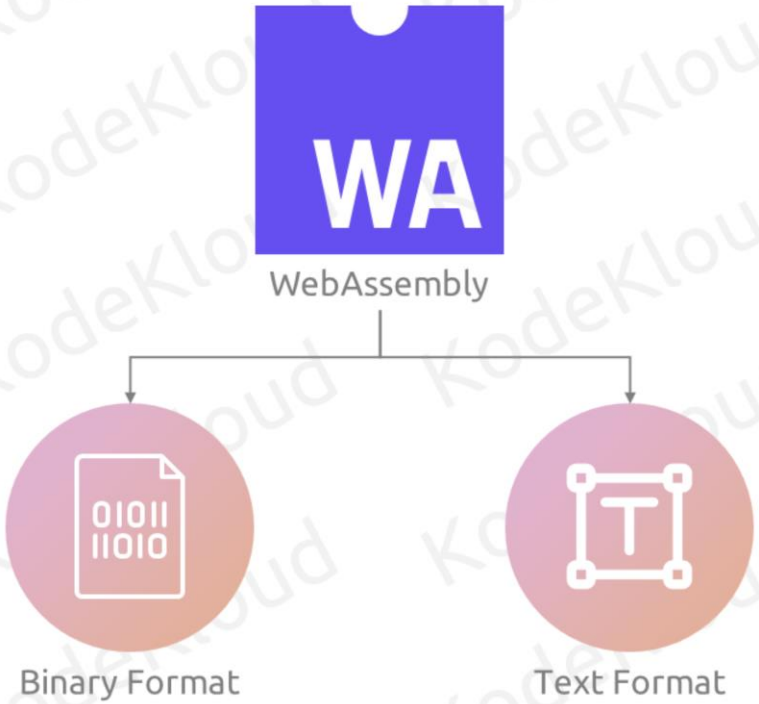
Getting Started with WebAssembly



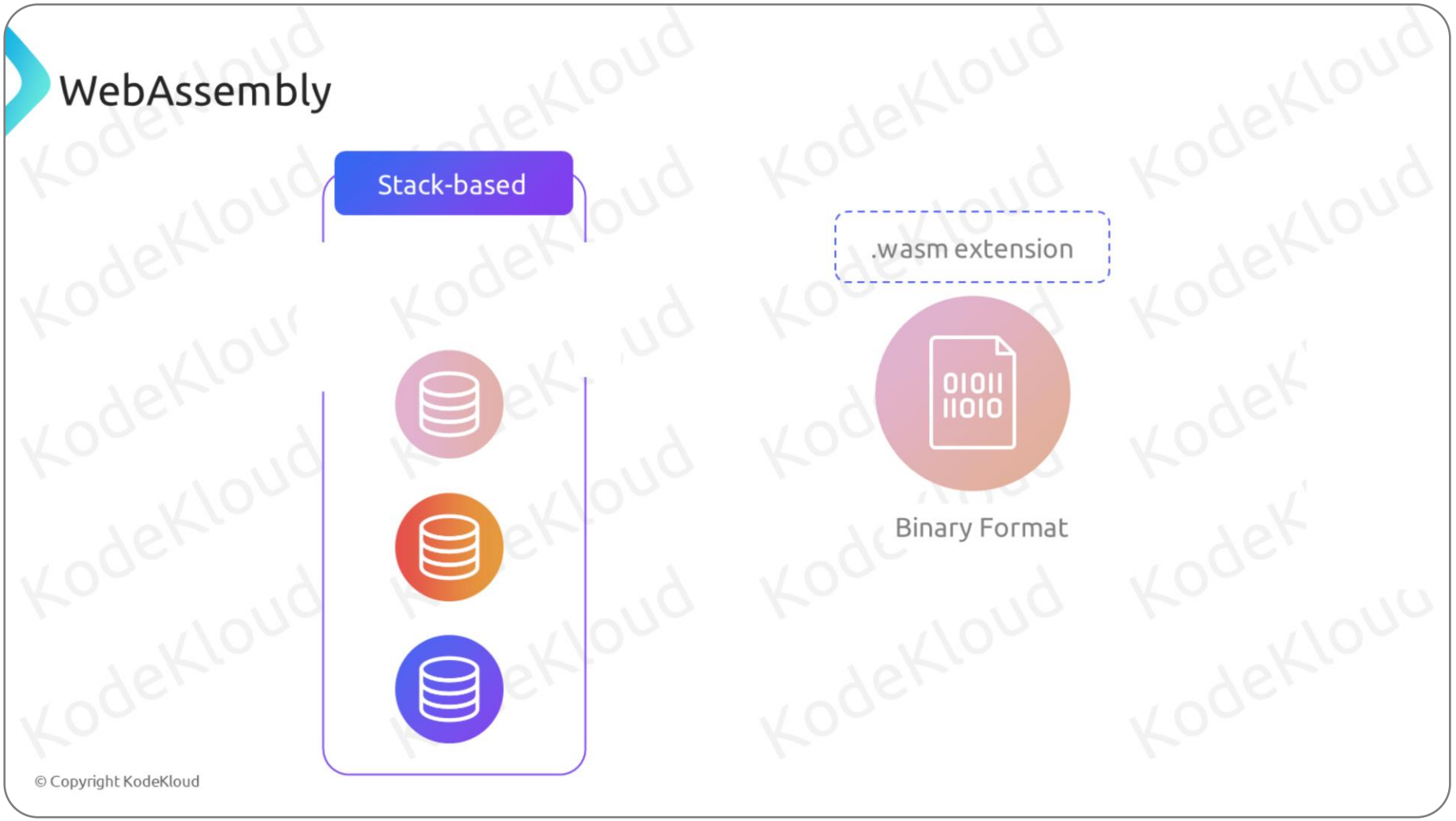
Participants will learn how to work with both binary and text formats of WebAssembly and the system interface (WASI). They will acquire skills for creating and running WebAssembly modules in a browser and understand how to use the WebAssembly JavaScript API. The hands-on demos will provide practical experience in setting up a development environment and running basic applications.

Understanding the WebAssembly Binary Format

WebAssembly



WebAssembly comes in two distinct formats: binary and text.



The binary version, represented by the .wasm extension, is a concise binary instruction set tailored for a stack-oriented virtual machine.

WebAssembly

It's crafted to serve as a universal compilation target for a number of high-level languages, such as to C, C++, Rust, C#, Go, and Python. This ensures that developers from various backgrounds can leverage WebAssembly's capabilities, enhancing web performance and bridging the gap between web and native applications.

Diving Into the WebAssembly Binary Format

In this lesson, we'll explore the WebAssembly (Wasm) binary format, which is essential for understanding how Wasm code is structured and executed. We'll break down each section of a Wasm binary file, provide code snippets, and detailed explanations for each section.

Before we delve into the WebAssembly binary format, let's first compile a simple C++ program to WebAssembly. This will give us a binary file to work with for our explanations.

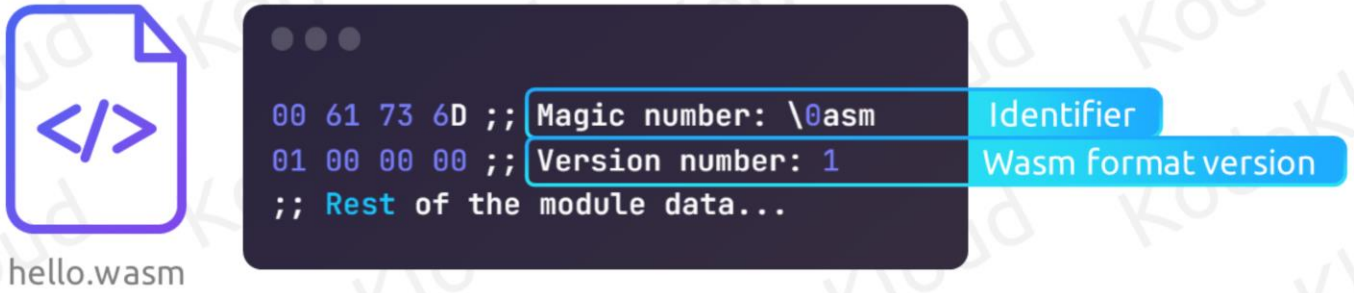
// code //

To compile this C++ program to WebAssembly, you'll need the Emscripten compiler (emcc hello.cpp -o hello.js). This will

generate both hello.wasm and hello.js files. We'll work with the hello.wasm binary file to explore its structure.

We'll dive deeper into the Emscripten compiler and the compilation process in upcoming lessons, so stay tuned for more details.

Module Structure



A Wasm binary file is organized into modules. Each module starts with a specific structure that includes a magic number and a version number. The magic number serves as an identifier for Wasm files, and the version number indicates the Wasm format version. Below is an example of this module structure:

// code //

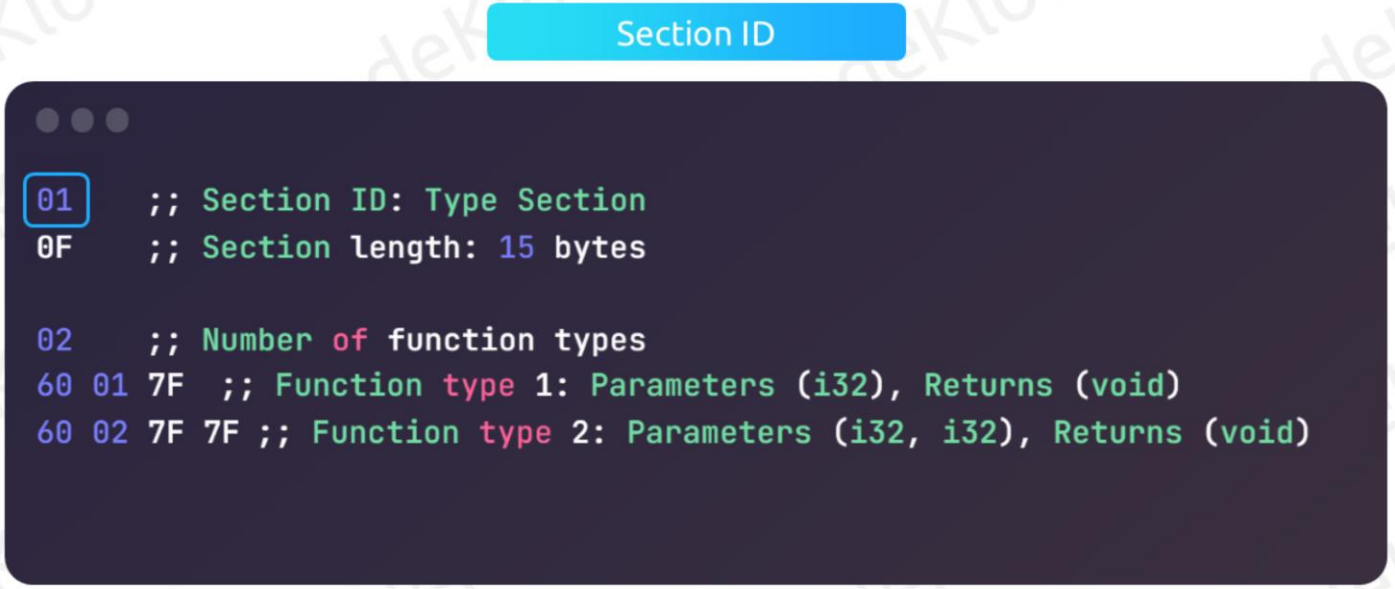
In this example, the magic number is \0asm, and the version number is 1.

Wasm modules consist of various sections, each serving a specific purpose. These sections are identified by unique section IDs and are arranged in ascending order of their IDs. Here is a generalized structure for a Wasm section:

// code //

Now, let's delve into some of the essential sections and their details.

Type Section (Section ID: 1)



The Type Section is used to define function types or signatures. It typically begins with 0x01 as the section ID and includes a list of function types. Each function type starts with 0x60, followed by parameter types and return value types. Here's an example:

// code //

In this example, we define two function types.

Type Section (Section ID: 1)

List of Function Types

```
01    ;; Section ID: Type Section
0F    ;; Section length: 15 bytes

02    ;; Number of function types
60 01 7F  ;; Function type 1: Parameters (i32), Returns (void)
60 02 7F 7F ;; Function type 2: Parameters (i32, i32), Returns (void)
```

Type Section (Section ID: 1)

Function Type Indicator

```
01    ;; Section ID: Type Section
0F    ;; Section length: 15 bytes

02    ;; Number of function types
60 01 7F  ;; Function type 1: Parameters (i32), Returns (void)
60 02 7F 7F ;; Function type 2: Parameters (i32, i32), Returns (void)
```

Type Section (Section ID: 1)

Parameter Types

```
01    ;; Section ID: Type Section
0F    ;; Section length: 15 bytes

02    ;; Number of function types
60 01 7F  ;; Function type 1: Parameters (i32), Returns (void)
60 02 7F 7F ;; Function type 2: Parameters (i32, i32), Returns (void)
```

Import Section (Section ID: 2)

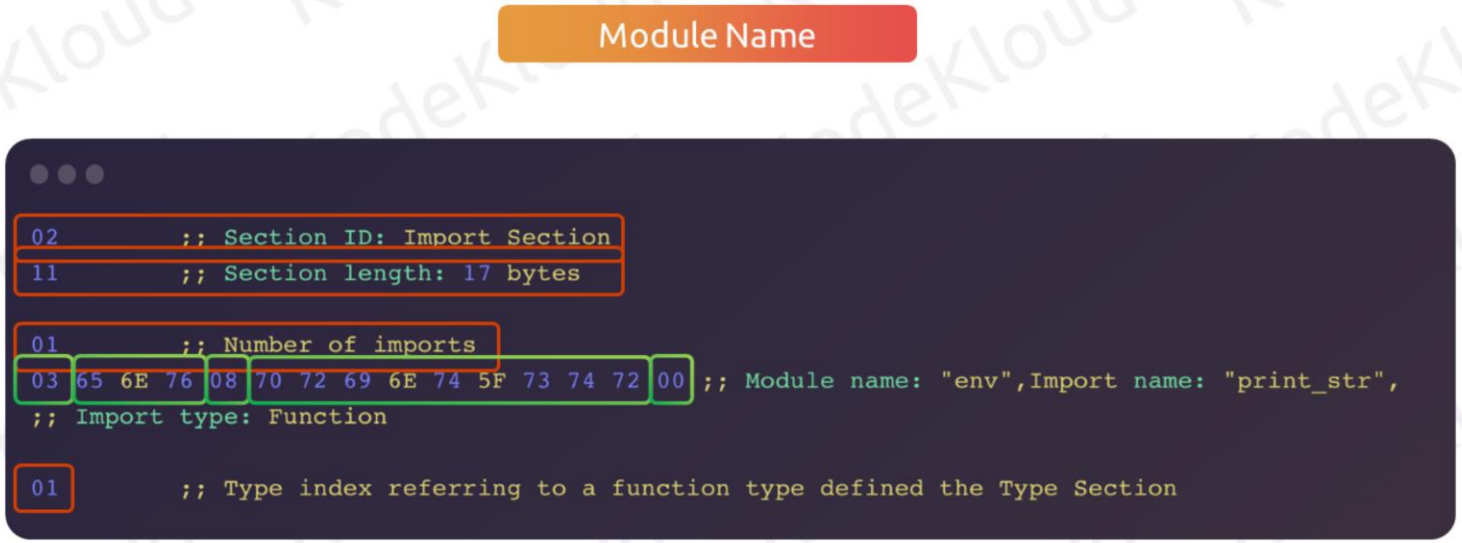
Import Section

- Functions
- Tables
- Memory
- Global Variables

Module

The Import Section is used to import various entities, such as functions, tables, memory, and global variables, into the module.

Import Section (Section ID: 2)



Each import has a module name, an import name, and an import description. Import descriptions start with a tag that specifies the type of import. Here's a simplified representation:

// code //

In this example, we import a function named "print_str" from the "env" module.

Function Section (Section ID: 3)

Four Functions

```
03    ;; Section ID: Function Section
05    ;; Section length: 5 bytes

04    ;; Number of functions
00 01 10 02 ;; Function indexes referring to function types in the Type Section
```


The Function Section associates functions with their corresponding types. It contains a list of function indexes that reference function types defined in the Type Section. Here's a simplified representation:

// code //

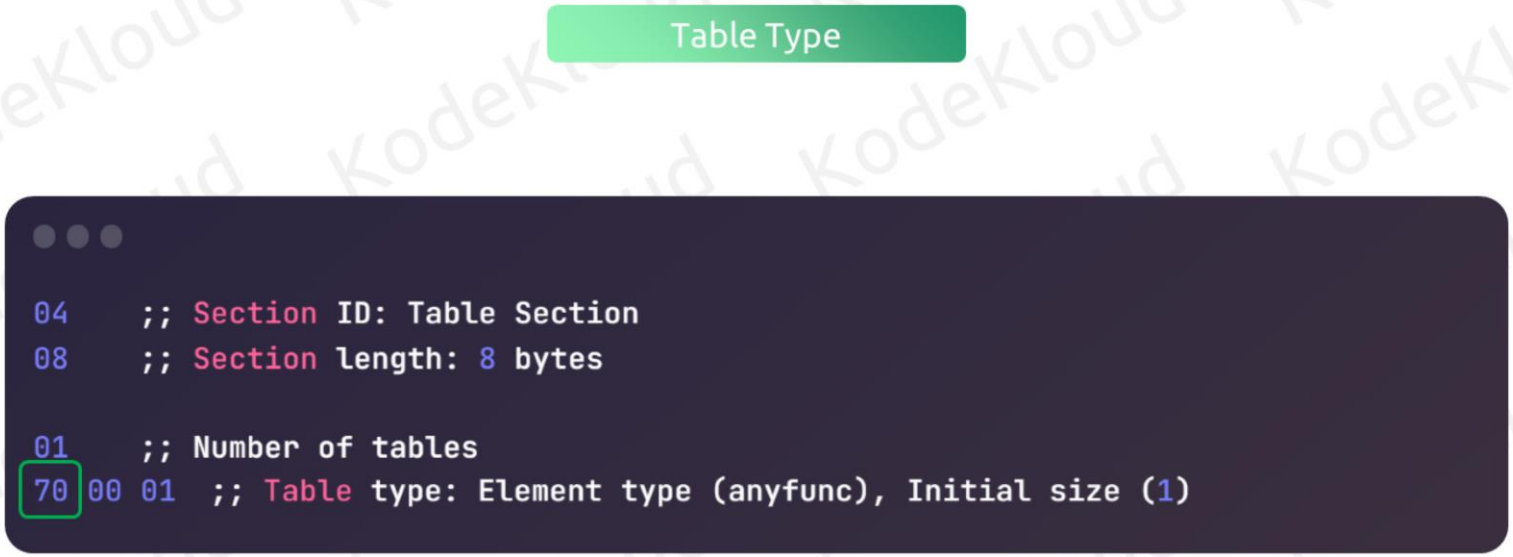
In this example, we associate four functions with their respective function types.

Function Section (Section ID: 3)

Function Indexes

```
03    ;; Section ID: Function Section
05    ;; Section length: 5 bytes

04    ;; Number of functions
00 01 10 02 ;; Function indexes referring to function types in the Type Section
```

Table Section (Section ID: 4)



The Table Section defines tables used by the module. It specifies the table types, including the element type and limits (initial and maximum). Here's a simplified representation:

// code //

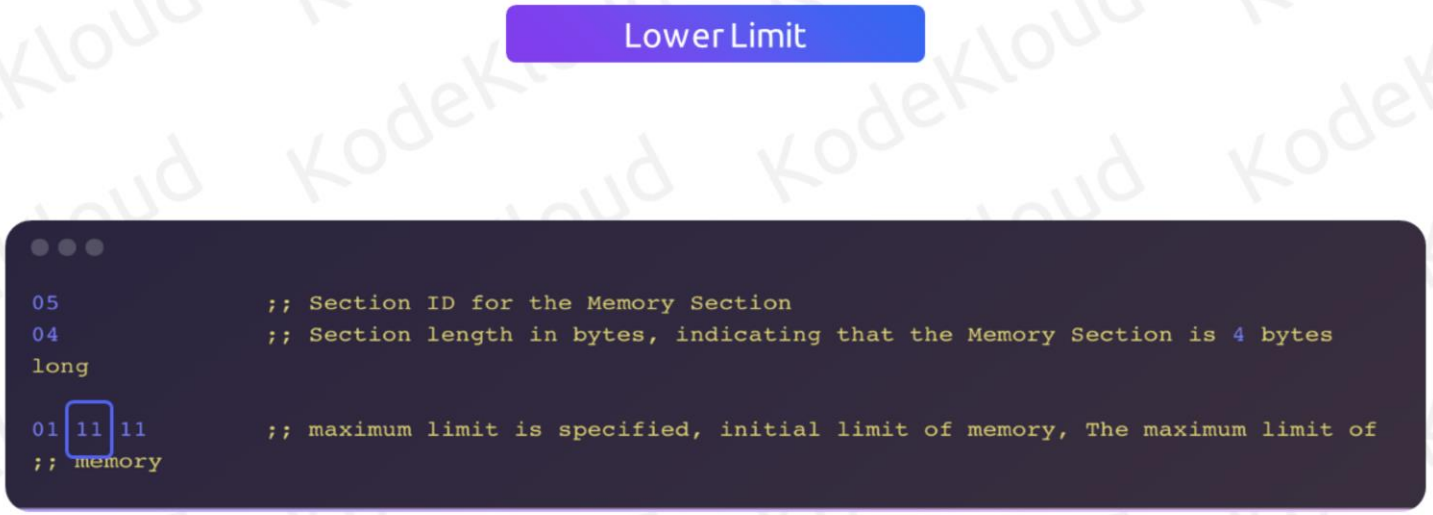
In this example, we define a table with an initial size of 1.

Table Section (Section ID: 4)

Limits

```
04    ;; Section ID: Table Section
08    ;; Section length: 8 bytes

01    ;; Number of tables
70 00 01  ;; Table type: Element type (anyfunc), Initial size (1)
```

Memory Section (Section ID: 5)



The Memory Section defines the memory used by the module. It specifies the lower and upper limits of memory pages required for the module's execution. Here's a simplified representation:

// code //

In this example, the memory has a lower limit of 0 pages and an upper limit of 17 pages.

Memory Section (Section ID: 5)

Upper Limit

```
05              ;; Section ID for the Memory Section
04              ;; Section length in bytes, indicating that the Memory Section is 4 bytes
long

01 11 11        ;; maximum limit is specified, initial limit of memory, The maximum limit of
;; memory
```

Global Section (Section ID: 6)

Global Variable Information

```
06              ;; Section ID for the Global Section
19              ;; Section length in bytes, 25 bytes long

03              ;; Number of Globale Variables
7F 01 41 0B     ;; Global Variable 1 : Type(i32), Mutability, Initialization(i32.const 11)
```

"Keep in mind that this sequence is not complete, as it only includes the first global variable's information."

The Global Section stores internal (non-imported) global variable information. It defines the type of the variable, whether it is mutable, and the bytecode used to initialize the global variable. Here's a simplified representation:

// code //

In this example, we define three global variables.

Global Section (Section ID: 6)

Variable Type

```
06              ;; Section ID for the Global Section
19              ;; Section length in bytes, 25 bytes long

03              ;; Number of Globale Variables
7F 01 41 0B     ;; Global Variable 1 : Type(i32), Mutability, Initialization(i32.const 11)
```

"Keep in mind that this sequence is not complete, as it only includes the first global variable's information."

Global Section (Section ID: 6)

Mutable or not

```
06              ;; Section ID for the Global Section
19              ;; Section length in bytes, 25 bytes long

03              ;; Number of Globale Variables
7F 01 41 0B     ;; Global Variable 1 : Type(i32), Mutability, Initialization(i32.const 11)
```

Global Section (Section ID: 6)

Bytecode Used to Initialize

```
06              ;; Section ID for the Global Section
19              ;; Section length in bytes, 25 bytes long

03              ;; Number of Globale Variables
7F 01 41 0B     ;; Global Variable 1 : Type(i32), Mutability, Initialization(i32.const 11)
```

Export Section (Section ID: 7)

The Export Section is used to export various entities from the module. Exported entities can be functions, tables, memory, or global variables. Each export specifies a name and the index of the exported entity. Here's a simplified representation:

// code //

In this example, we export a function named "main."

These are some of the fundamental sections in a Wasm binary file. Further sections, such as Start, Code, Data, and Custom sections, play important roles in defining a complete Wasm module. Understanding these sections is crucial for working effectively with WebAssembly.

Export Section (Section ID: 7)

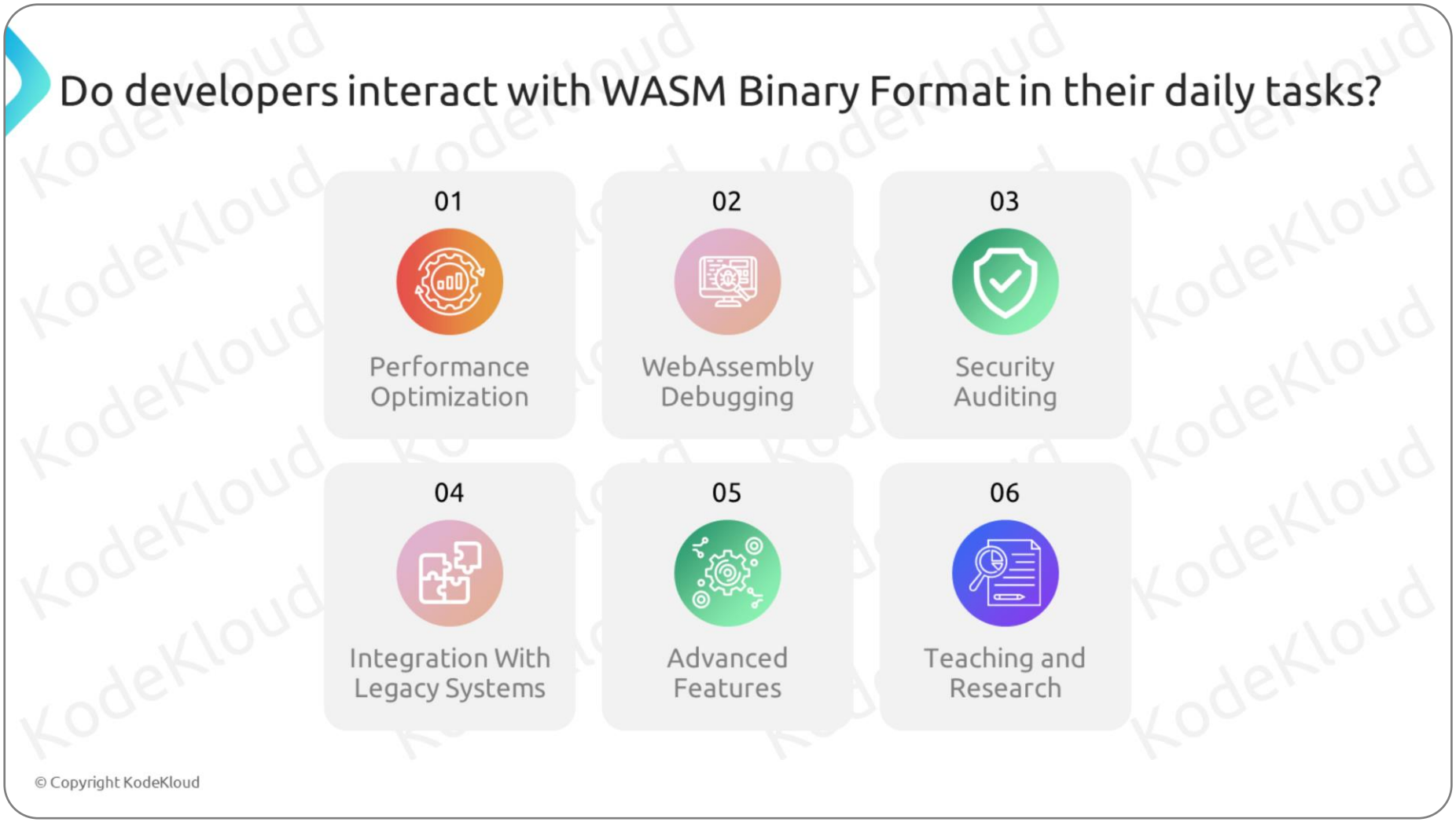
Export Type

```
07              ;; Section ID for the Export Section
2C              ;; Section length in bytes, 44 bytes long

04              ;;  length of the export name
04 6D 61 69 6E 00 03 ;; export name(main), export type(function), index(3)
```

Export Section (Section ID: 7)

Index

```
07              ;; Section ID for the Export Section
2C              ;; Section length in bytes, 44 bytes long

04              ;;  length of the export name
04 6D 61 69 6E 00 03 ;; export name(main), export type(function), index(3)
```



While WebAssembly (Wasm) has made significant strides in the web development landscape, a pertinent question arises: Do developers frequently interact with the Wasm binary format in their daily tasks? The answer largely hinges on the developer's role, the nature of the project, and specific application requirements. Let's explore some practical contexts where a grasp of the Wasm binary format might be advantageous:

1. Performance Optimization:

When: In performance-critical applications.
Why: Understanding the binary format allows you to optimize your code at a low level, resulting in faster execution and reduced resource consumption. This can be crucial in applications like games, simulations, or web apps with demanding computational tasks.
2. WebAssembly Debugging:
When: When you need to debug complex issues.
Why: Familiarity with the binary format can facilitate a more granular inspection of raw data, leverage the WebAssembly Text Format (Wat) for pinpoint debugging, and streamline problem resolution, especially in expansive or complex projects.
3. Security Auditing:
When: In security-critical applications.
Why: A clear understanding of the binary format's memory layouts and data access patterns is pivotal for spotting and addressing potential security weak points, especially in applications that handle confidential data or unfiltered user inputs.
4. Integration with Legacy Systems:
When: Merging WebAssembly modules into pre-existing projects.
Why: In projects that necessitate the seamless integration of WebAssembly with older systems, knowledge of the binary format can be instrumental in fostering smooth communication between Wasm modules and their host environments.
5. Advanced Features:
When: When utilizing advanced features.
Why: If you want to take full advantage of WebAssembly's capabilities, such as custom sections for embedding metadata, understanding the binary format is necessary. It enables you to harness advanced features effectively.
6. Teaching and Research:
When: In educational or research settings.
Why: If you're teaching or researching WebAssembly, a deep understanding of the binary format is essential. It allows you to explore and experiment with the technology comprehensively.

Example: A researcher might explore how different compilers translate high-level code into Wasm. By comparing the binary outputs, they can deduce the efficiency of each compiler.

While not every developer needs to delve into the binary format regularly, having the knowledge in your toolkit can be incredibly valuable when the need arises. In many cases, you may work at a higher level of abstraction using languages like Rust, C++, or JavaScript that compile to WebAssembly, but knowing what's happening under the hood can help you troubleshoot, optimize, and innovate effectively.

Do developers interact with WASM Binary Format in their daily tasks?

While not every developer needs to delve into the binary format regularly, having the knowledge in your toolkit can be incredibly valuable when the need arises.

Do developers interact with WASM Binary Format in their daily tasks?

In many cases, you may work at a higher level of abstraction using languages like Rust, C++, or JavaScript that compile to WebAssembly, but knowing what's happening under the hood can help you troubleshoot, optimize, and innovate effectively.

Understanding the WebAssembly Text Format

WebAssembly Text Format (WAT) – Introduction

WASM is a binary format. It's not directly readable or writable by humans.

WebAssembly Text Format (WAT) – Introduction

That's where the WebAssembly Text Format, or WAT, comes into play. WAT provides a human-readable representation of WASM, allowing developers to write, debug, and understand WebAssembly code with ease.

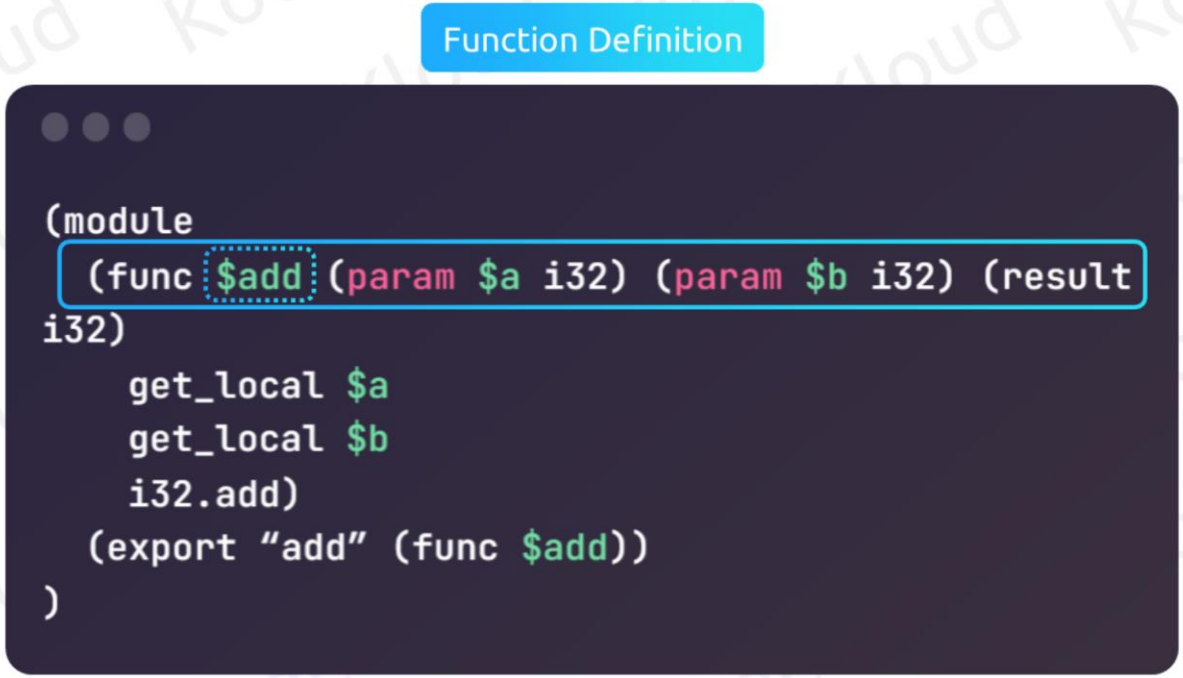
Have a look at a simple WAT example

Consider the following WAT code:

// code //

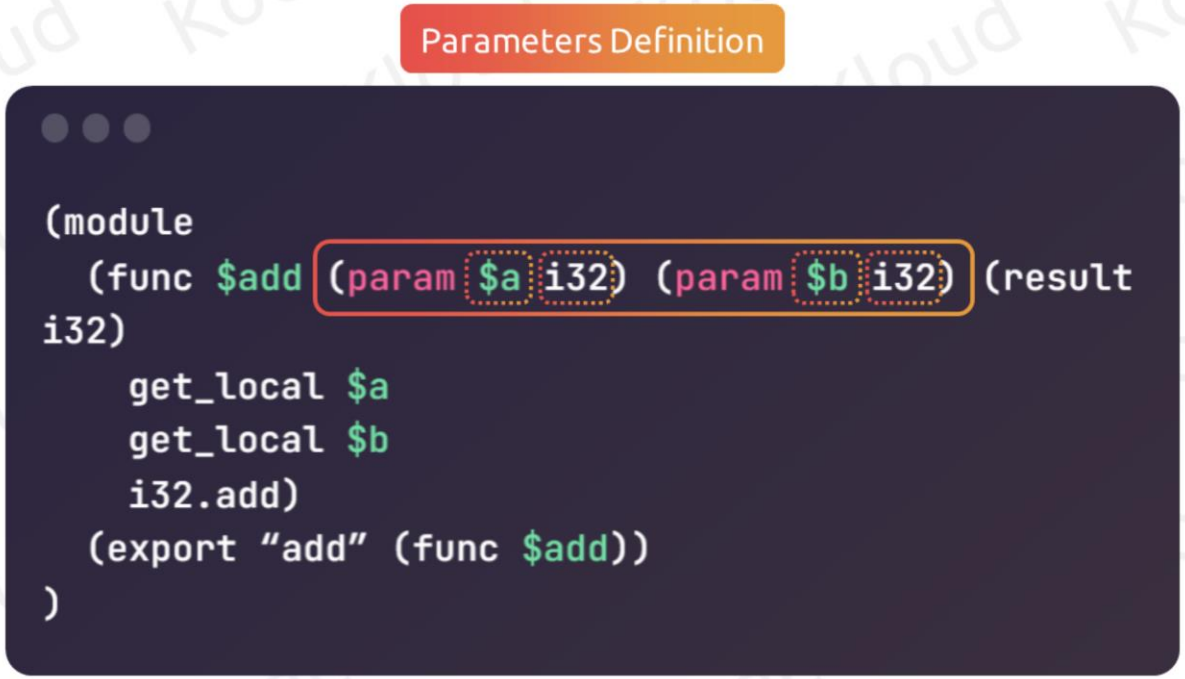
This module defines a simple function named $add that takes two 32-bit integers as parameters and returns their sum. The function is also exported with the name "add", allowing it to be called from outside the module.

Breaking Down the Example



Module Definition: The (module ...) construct defines a WebAssembly module. Everything inside this construct is part of the module.
Function Definition: The (func $add ...) construct defines a function named $add.

Breaking Down the Example



Parameters: (param $a i32) (param $b i32) declares two parameters, $a and $b, both of which are 32-bit integers.

Breaking Down the Example

Result Type: (result i32) indicates that the function will return a 32-bit integer.

Breaking Down the Example

Function Body: The body of the function contains the instructions to execute. In this case:
get_local $a pushes the value of $a onto the stack.
get_local $b pushes the value of $b onto the stack.
i32.add pops the two top values from the stack, adds them, and then pushes the result back onto the stack.

Breaking Down the Example

Export: The (export "add" (func $add)) construct makes the $add function accessible outside the module with the name "add".

WASM as a Language

Starting from this example, we can explore Web Assembly in more detail, so we'll look at its different parts, variations and instructions to understand how the web assembly text format is written. So let's start with the grammar and white space. In what? So like any language, So what? Or representably text format, like any language has a grammar, so a set of rules for structuring code.

Grammar and Whitespace in WAT

```
(module
  (func $add (param $a i32) (param $b i32) (result i32)
    get_local $a
    get_local $b
    i32.add)
  (export "add" (func $add))
)
```

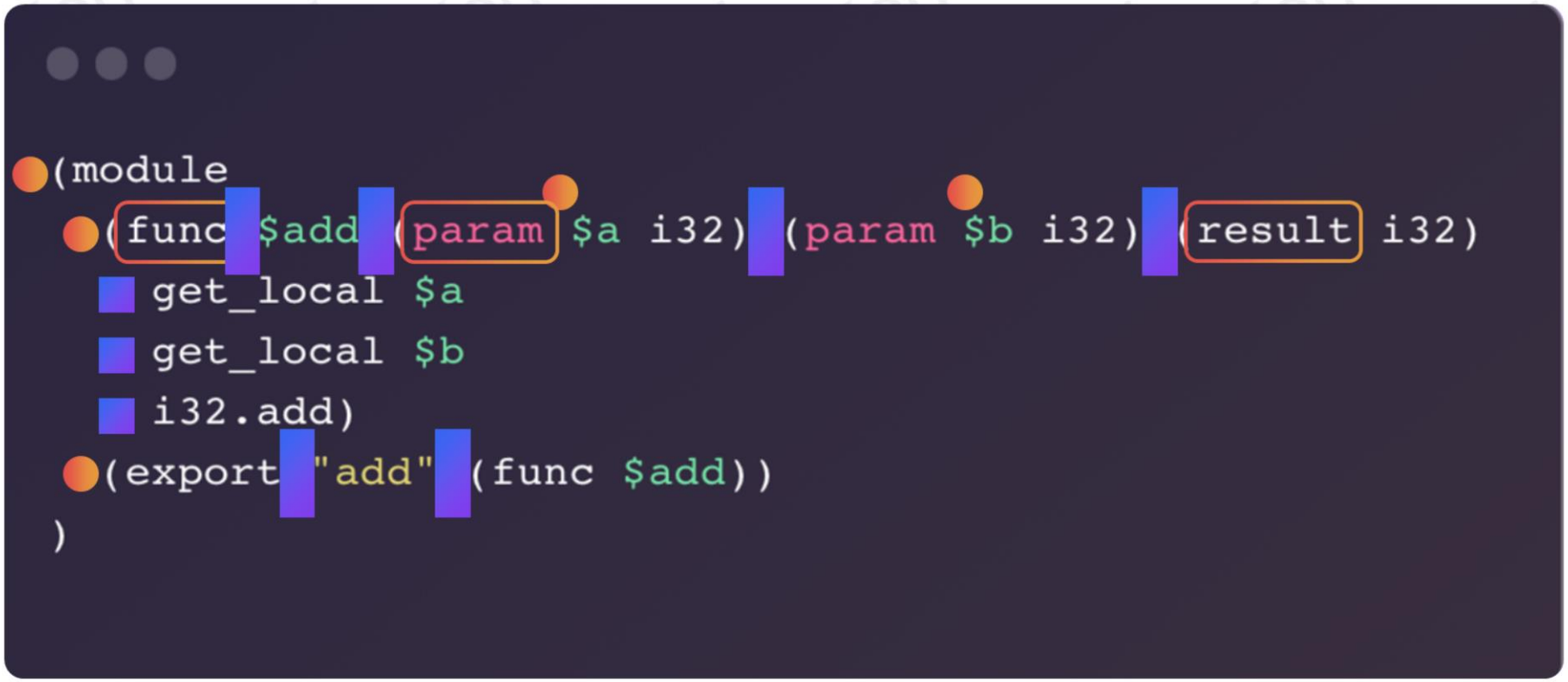

Let’s start with Grammar and whitespace in WAT.

WAT, like any language, has a grammar – a set of rules for structuring code.

WASM as a Language

Our current example already adheres to what's grammar, parenthesis defining scope and structure, while keywords like the funk and param and result specify the code's function. Now white space so that spaces and new lines is used for readability and doesn't impact the functionality.

Exploring WebAssembly (WAT)



For instance, parameters are clearly separated by spaces: (param $a i32) (param $b i32).

Comments in WAT

To make our code more understandable, let's add comments using ;;. Comments are crucial for explaining what certain parts of the code do, making it easier for others (or yourself at a later time) to understand the code.

Comments in WAT

Indicates a comment

```
(module
  ;; Define a function to add two integers
  (func $add (param $a i32) (param $b i32) (result i32)
    get_local $a
    get_local $b
    i32.add)
  (export "add" (func $add))
)
```

Lexical Format in WAT

If we talk about the Lexical Format in WAT, this involves the characters, tokens, and other basic syntax elements used. Identifiers like $add, $a, $b, and types like i32 are part of WAT's lexical structure. In our function, $add is an identifier for the function, while i32 specifies the 32-bit integer type.

Types and Values in WAT

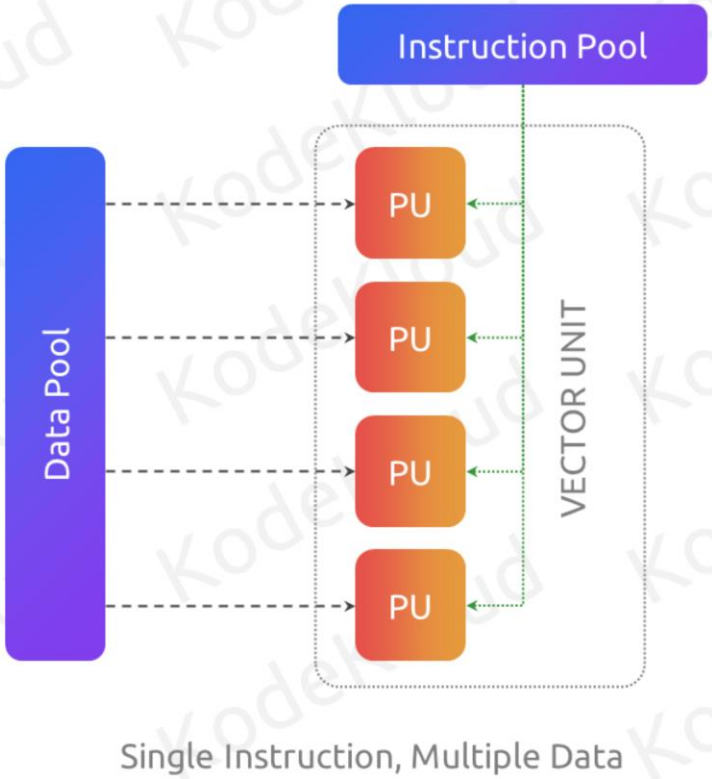
```
(module
  ;; Integer addition
  (func $add (param $a i32) (param $b i32) (result i32)
    get_local $a
    get_local $b
    i32.add)
  ;; Floating-point addition
  (func $add_float (param $a f32) (param $b f32) (result f32)
    get_local $a
    get_local $b
    f32.add)
  (export "add" (func $add))
  (export "add_float" (func $add_float))
)
```


Next the Types and Values in WAT.

Let's extend our function to also handle floating-point numbers. We introduce a new function $add_float that works with 32-bit floating points (f32). We'd change our parameter types and the add operation to accommodate f32. This would demonstrate the flexibility and variety of types in WAT.

Types and Values in WAT

WAT



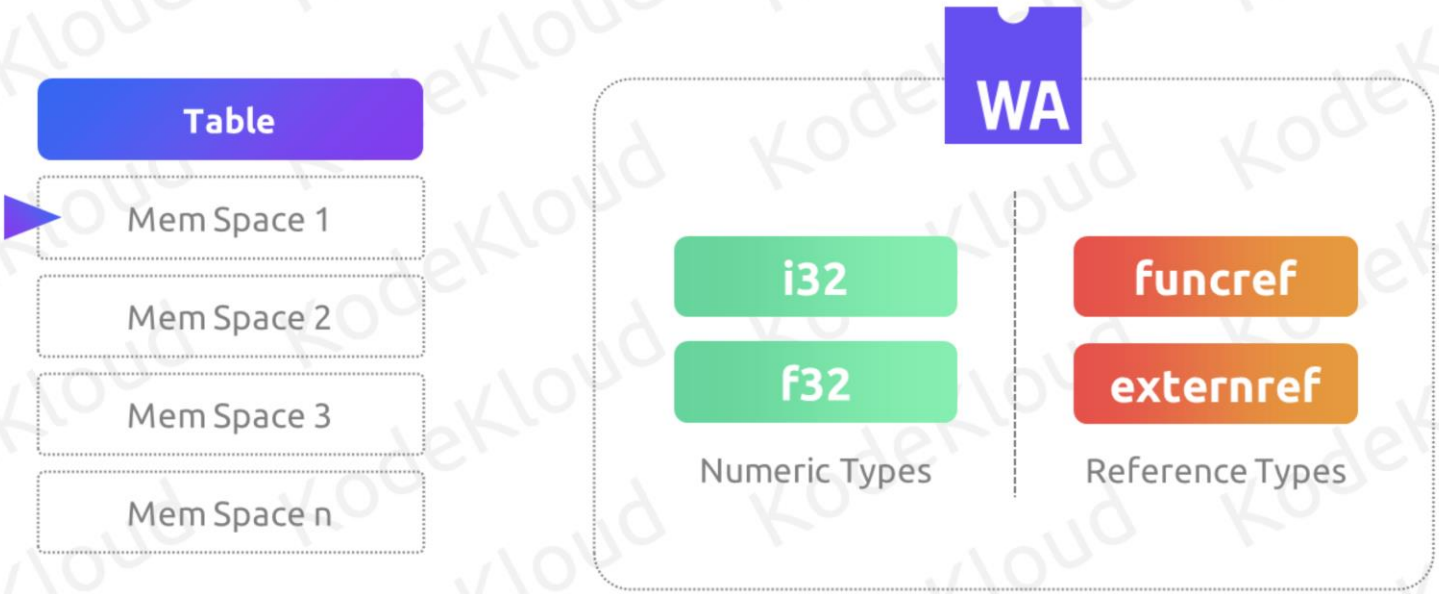
WAT supports vector types, which are useful for SIMD (Single Instruction, Multiple Data) operations. Let's assume we want to add a function that operates on vector types. We'll introduce v128, a vector type that represents 128 bits of data.

Types and Values in WAT

WAT supports vector types, which are useful for SIMD (Single Instruction, Multiple Data) operations. Let's assume we want to add a function that operates on vector types. We'll introduce v128, a vector type that represents 128 bits of data.

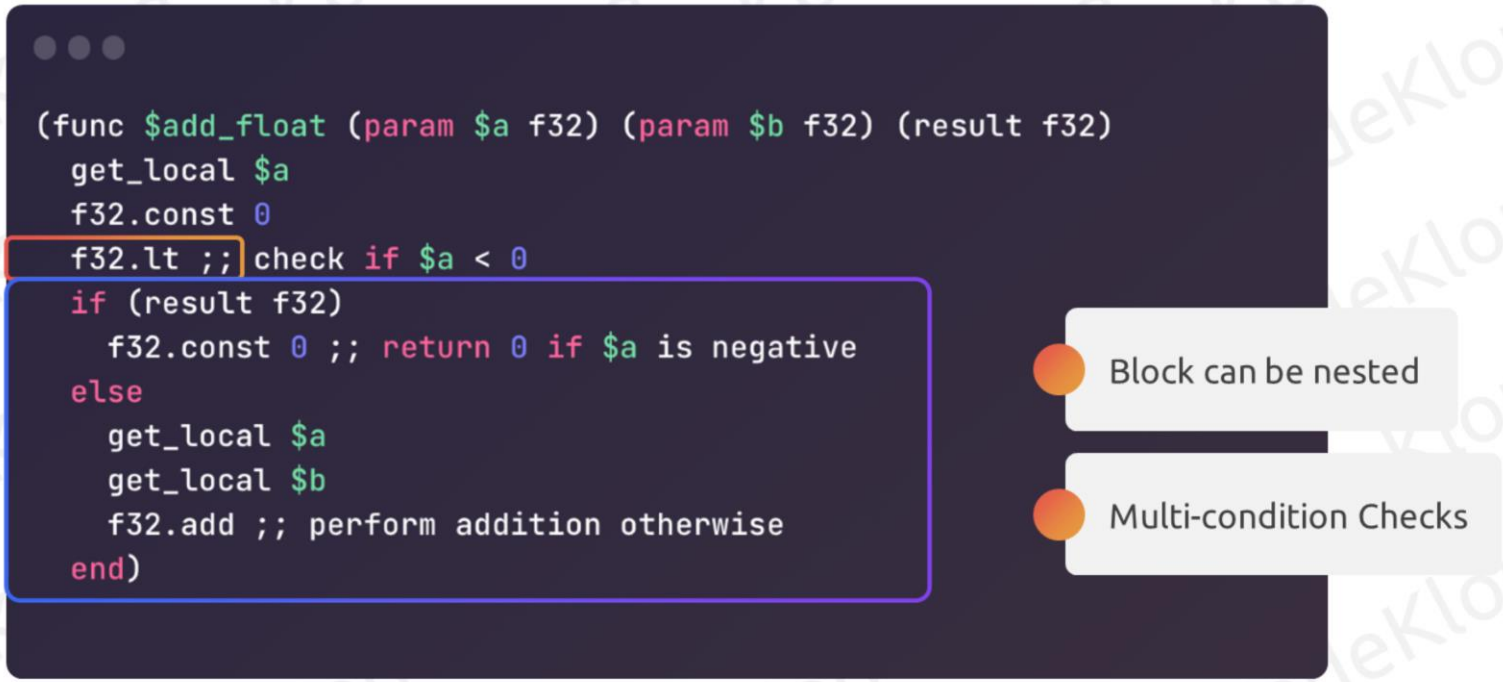
Here, v128.add performs an addition operation on two 128-bit vectors.

Types and Values in WAT



Next, the Reference types. They allow functions to work with references. Suppose we want to manipulate a reference to a table or memory. For simplicity, let's stick to numeric types in our example, but keep in mind that WAT also supports reference types like funcref or externref.

Instructions in WAT



Now, we are going to talk about the Instructions in WAT.

Suppose we want to add a condition in our function. We can introduce an if control structure. Let's add a condition to our float addition function to return 0 if the first parameter is negative:

// code //

Keep in mind that, the Blocks can be nested within each other for complex control flow. For instance, we might want to add nested if blocks in our functions for multi-condition checks. However, our current example doesn't naturally lend itself to nested blocks without complicating it unnecessarily. It's important to note that WAT allows for this complexity when needed.

Instructions in WAT

Block

```
(module
  ;; ... previous definitions ...

  (func $sum_array (param $ptr i32) (param $length i32) (result i32)
    (local $i i32) ;; Local variable for loop counter
    (local $sum i32) ;; Local variable for accumulating the sum
    (loop $loop
      (br_if $end (i32.eq (get_local $i) (get_local $length))) ;; Break if $i ==
$length
      (set_local $sum
        (i32.add (get_local $sum) (i32.load (get_local $ptr))))
      (set_local $ptr (i32.add (get_local $ptr) (i32.const 4))) ;; Move to the next
integer in memory
      (set_local $i (i32.add (get_local $i) (i32.const 1))) ;; Increment counter
      (br $loop)) ;; Repeat the loop
    $end
    (get_local $sum))

  ;; ... other functions ...

  (export "sum_array" (func $sum_array))
  ;; ... other exports ...
)
```

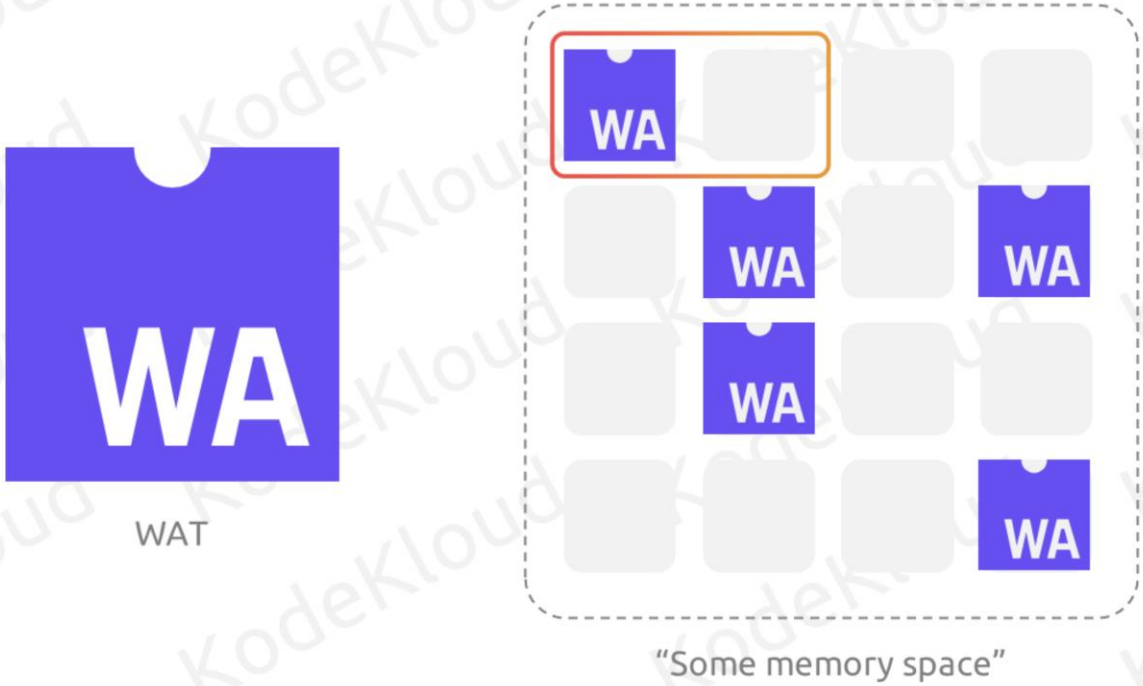

We can introduce loops to perform repetitive operations. Let's add a function that sums an array of integers using a loop.

Instructions in WAT

```
(module
  ;; ... previous definitions ...

  (func $sum_array (param $ptr i32) (param $length i32) (result i32)
    (local $i i32) ;; Local variable for loop counter
    (local $sum i32) ;; Local variable for accumulating the sum
    (loop $loop
      (br_if $end (i32.eq (get_local $i) (get_local $length))) ;; Break if $i ==
$length
      (set_local $sum
        (i32.add (get_local $sum) (i32.load (get_local $ptr))))
      (set_local $ptr (i32.add (get_local $ptr) (i32.const 4))) ;; Move to the next
integer in memory
      (set_local $i (i32.add (get_local $i) (i32.const 1))) ;; Increment counter
      (br $loop)) ;; Repeat the loop
    $end
    (get_local $sum))

  ;; ... other functions ...

  (export "sum_array" (func $sum_array))
  ;; ... other exports ...
)
```


In this function, we use a loop with a label $loop and a br_if instruction to break out of the loop when the condition is met.

Memory and Table instructions in WAT



Finally, we will go through the Memory and Table instructions in WAT which is advanced but let's just get an idea.

WebAssembly modules can declare their own linear memory space. Let's add a memory declaration to our module and modify our $add function to use this memory. Assume we store our integers in the first two positions of the memory:

Memory and Table instructions in WAT

```
(module
  (memory 1) ;; Declare a memory of 1 page (64KiB)

  (func $add (result i32)
    (i32.load offset=0) ;; Load the first integer from memory
    (i32.load offset=4) ;; Load the second integer from memory
    i32.add) ;; Add them

  ;; ... other functions ...

  (export "add" (func $add))
  ;; ... other exports ...
)
```


In this example, i32.load offset=0 and i32.load offset=4 are used to load integers from the first two 4-byte slots of the memory.

Memory and Table instructions in WAT

```
(module
  (table 2 anyfunc) ;; Declare a table with 2 slots for function references

  ;; ... memory and function definitions ...

  ;; Store function references in the table
  (elem (i32.const 0) $add $add_float)

  (func $call_add (param $index i32) (result i32)
    (i32.const 0) ;; Load the first argument for $add
    (i32.const 4) ;; Load the second argument for $add
    (call_indirect (type $add) (get_local $index)))

  ;; ... other functions ...

  (export "call_add" (func $call_add))
  ;; ... other exports ...
)
```


Tables in WebAssembly are used to store references, like function references. Let's say we want to create a table that can store references to our functions. We'll declare a table and use an indirect call to execute a function from the table.

// code //

In this example, (table 2 anyfunc) declares a table with two slots. The (elem ...) line initializes the table with the $add and

$add_float functions. The $call_add function demonstrates how to use call_indirect to call a function referenced in the table.

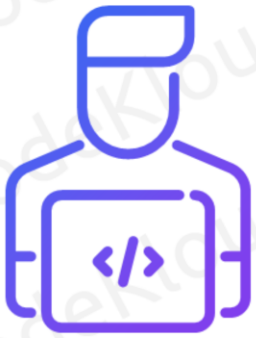
By integrating memory and table operations into our example, we've significantly broadened the scope of our WAT module, illustrating more complex and powerful features of WebAssembly.

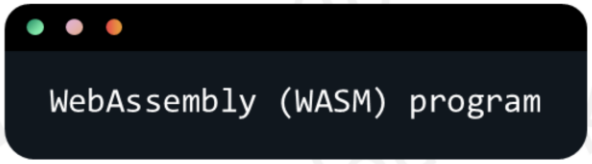
WebAssembly System Interface (WASI)

WebAssembly

Imagine you're a developer who has just written a WebAssembly (WASM) program designed to process large data files.

WebAssembly



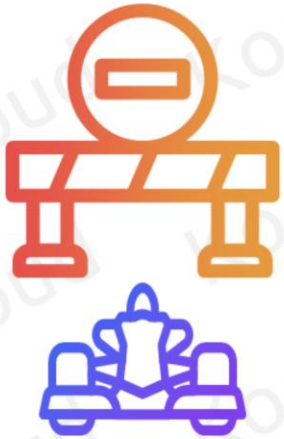


Your program is efficient and runs at near-native speed in browsers, thanks to the power of WebAssembly. Excitedly, you decide to test it on a local data file stored on your machine. However, you quickly realize there's a problem: your WASM program can't access the local file system. It's like having a powerful car with no roads to drive on.

This limitation isn't a flaw in your code but an inherent restriction of WebAssembly's design. By default, WASM operates in a sandboxed environment for security reasons, preventing it from directly accessing system functionalities like file reading.

While this sandboxing is great for security, it restricts the range of applications you can develop with just WebAssembly.

Enter WASI.



Your program is efficient and runs at near-native speed in browsers, thanks to the power of WebAssembly. Excitedly, you decide to test it on a local data file stored on your machine. However, you quickly realize there's a problem: your WASM program can't access the local file system. It's like having a powerful car with no roads to drive on.

This limitation isn't a flaw in your code but an inherent restriction of WebAssembly's design. By default, WASM operates in a sandboxed environment for security reasons, preventing it from directly accessing system functionalities like file reading. While this sandboxing is great for security, it restricts the range of applications you can develop with just WebAssembly.

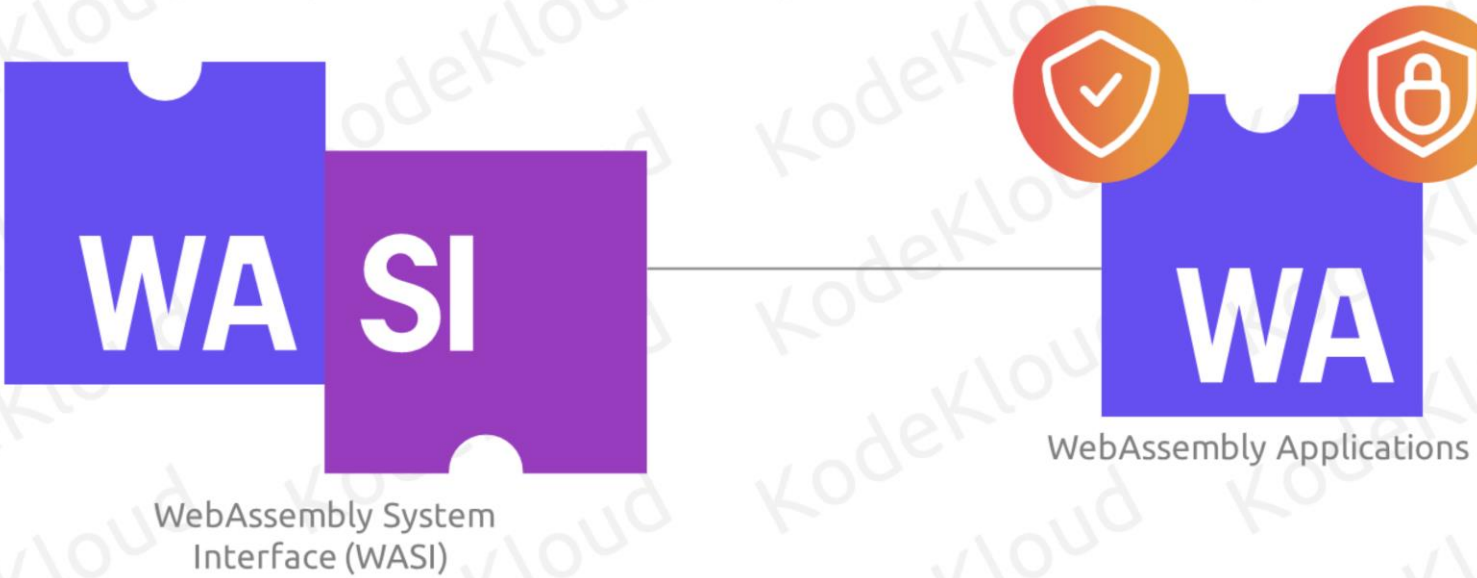
Enter WASI.

Why WASI?

WebAssembly System
Interface (WASI)

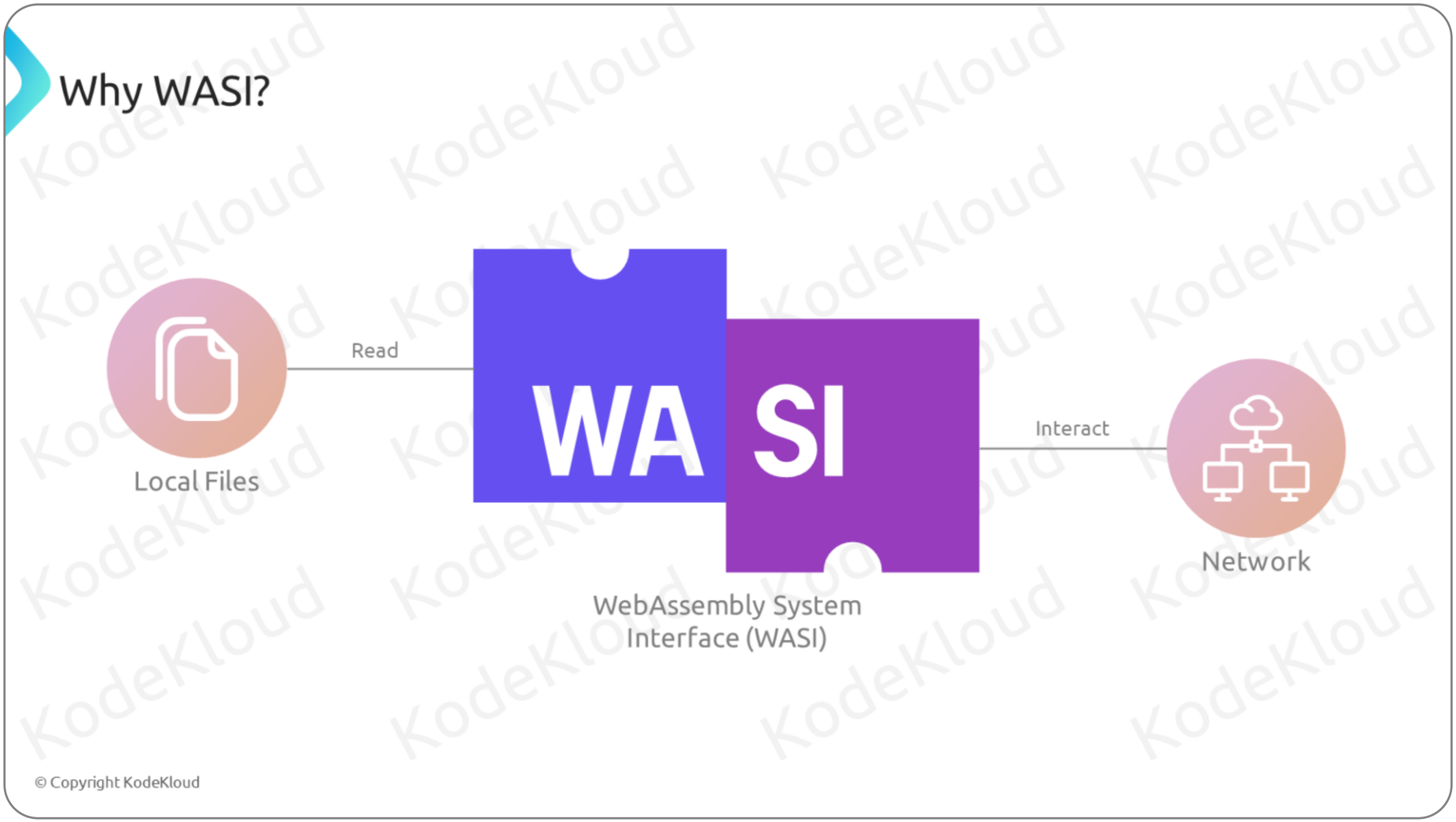
The WebAssembly System Interface (WASI) is the solution to this challenge. It acts as a bridge, allowing WebAssembly applications to safely and securely access system functionalities without compromising the inherent security benefits of WebAssembly.

Why WASI?



WebAssembly System Interface (WASI)

WebAssembly Applications



With WASI, your program can now read local files, interact with the network, and do so much more, all while maintaining the portability and speed that WebAssembly offers.

Need to Understand WASI

Imagine you've just crafted a WebAssembly application designed to process intricate graphics.

Need to Understand WASI

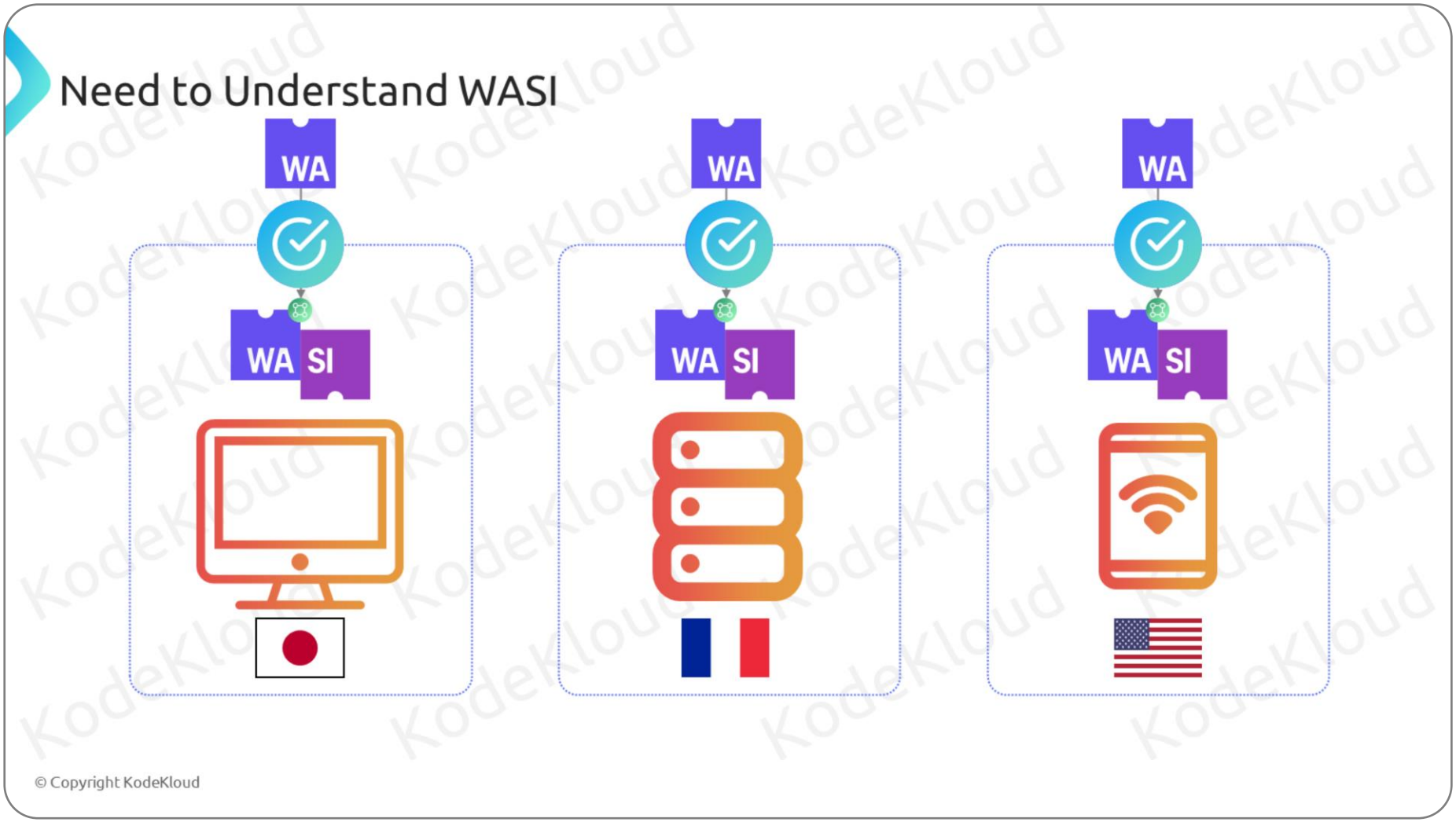
WebAssembly Applications

You proudly share it with colleagues across different continents, hoping they can immediately use it.

Need to Understand WASI

But here's the catch: without WASI, your application might stutter on different platforms or, worse, not run at all.

Need to Understand WASI



WASI ensures that whether your friend is using a computer in Tokyo, a server in Paris, or an IoT device in San Francisco, your application runs smoothly and consistently.

Now, think about a locked treasure chest. WebAssembly, by default, is like that chest, holding immense power but restricted in its interactions. WASI is the key, unlocking specific permissions to ensure the treasure inside (your application) is both safe and functional. It ensures that while your application can read a specific file, it won't accidentally delete

important system files.

Lastly, remember the early days of the internet when websites were static pages? WebAssembly initially had a similar phase, primarily designed for browsers. But with WASI, it's like giving those static web pages the dynamism of modern web apps. Your WebAssembly applications aren't just confined to browsers; they're now empowered to venture into the realms of cloud computing, IoT devices, and beyond.

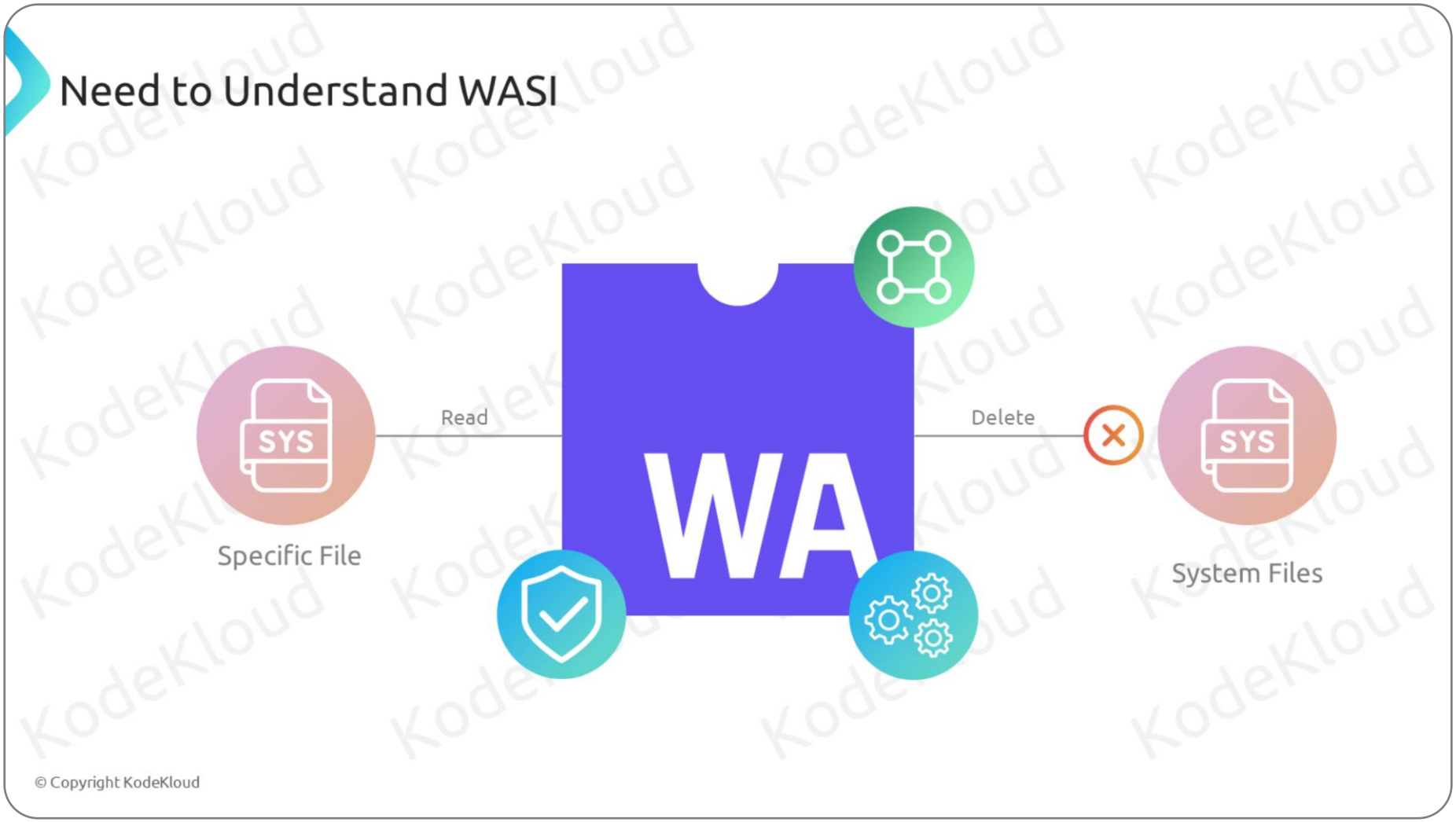
Need to Understand WASI

Now, think about a locked treasure chest. WebAssembly, by default, is like that chest, holding immense power but restricted in its interactions.

Need to Understand WASI

WASI is the key, unlocking specific permissions to ensure the treasure inside (your application) is both safe and functional.

Need to Understand WASI

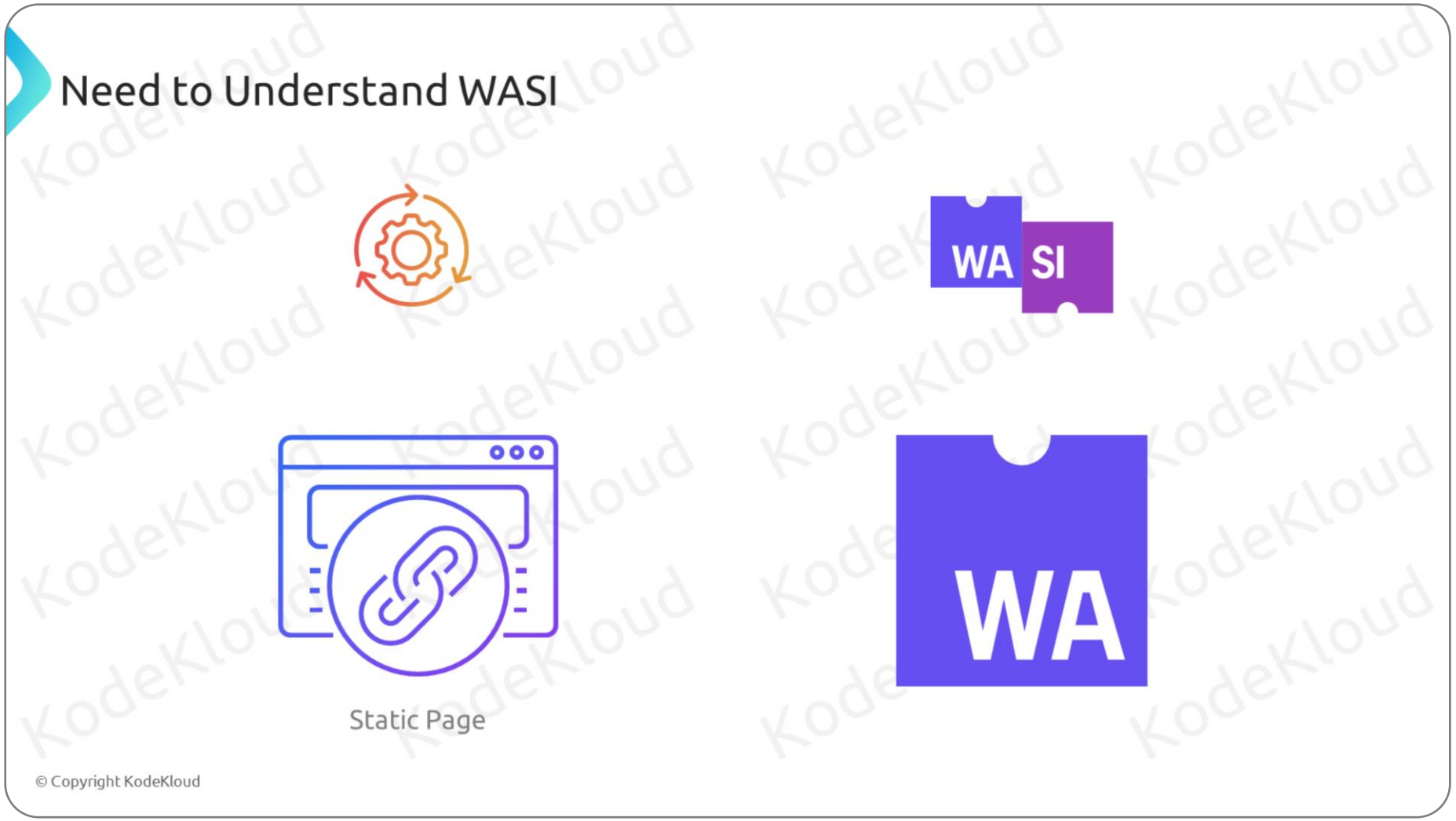


It ensures that while your application can read a specific file, it won't accidentally delete important system files.

Need to Understand WASI

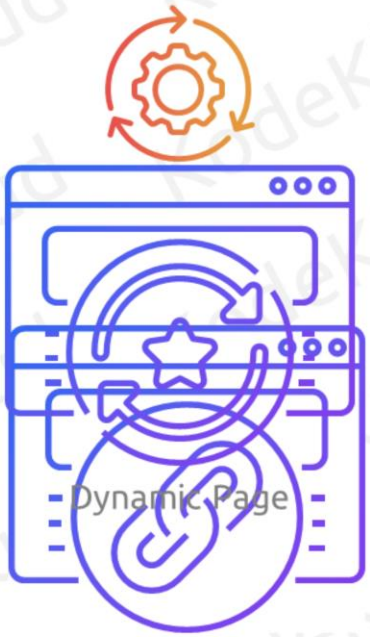
Static Page

Lastly, remember the early days of the internet when websites were static pages? WebAssembly initially had a similar phase, primarily designed for browsers.



But with WASI, it's like giving those static web pages the dynamism of modern web apps.

Need to Understand WASI



Your WebAssembly applications aren't just confined to browsers; they're now empowered to venture into the realms of cloud computing, IoT devices, and beyond.

What is WASI?

WASI, the WebAssembly System Interface, is a modular system interface for WebAssembly.

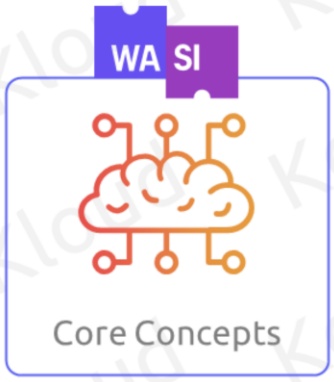
What is WASI?

Think of it as the bridge between WebAssembly modules and the host system.

What is WASI?

It provides a set of standardized APIs, allowing WebAssembly applications to perform operations like reading files, network activities, and more.

WASI – Core Concepts



Recall the WebAssembly application you built for processing intricate graphics. As it stands, the application is powerful, but with the integration of WASI's core concepts, its capabilities are enhanced manifold.

WASI – Core Concepts

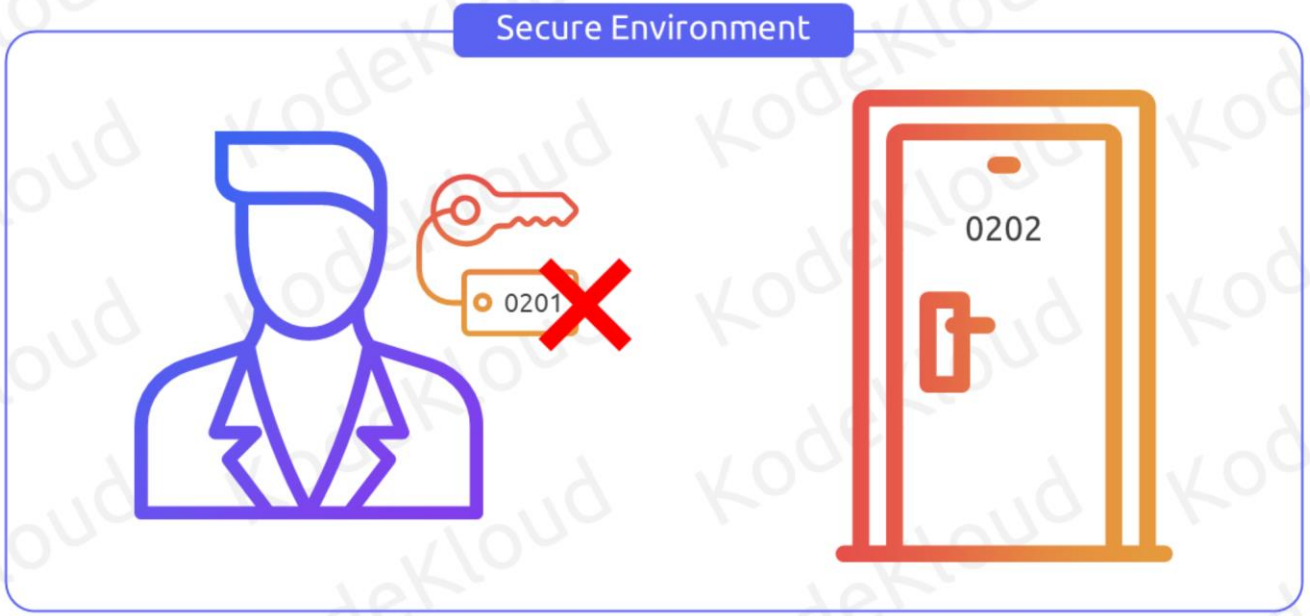
Capabilities-based Security Model:
Imagine your application needs to pull in various graphic assets from a user's system. Without WASI, this could be a security nightmare. But with WASI's capabilities-based security model, instead of giving your application a free pass to the entire system, you grant it permission only to access the photo directory. It's like a hotel giving a guest a room key; the guest can access only their room and not the entire hotel. This ensures the guest doesn't accidentally wander into someone else's room, and similarly, your application doesn't access or modify unintended parts of the system, guaranteeing a secure

environment.

WASI – Core Concepts

WebAssembly Applications

WASI – Core Concepts



WASI – Core Concepts

Modularity:
Your graphic processing application might require specific features like texture mapping or shading. WASI, with its modular design, allows you to incorporate just these specific functionalities without burdening your application with unnecessary features. Think of it like a graphic artist's toolkit: rather than carrying every possible color and brush, they only pack the specific shades and tools needed for a particular artwork, ensuring efficiency and lightness.

WASI – Core Concepts

WASI – Core Concepts

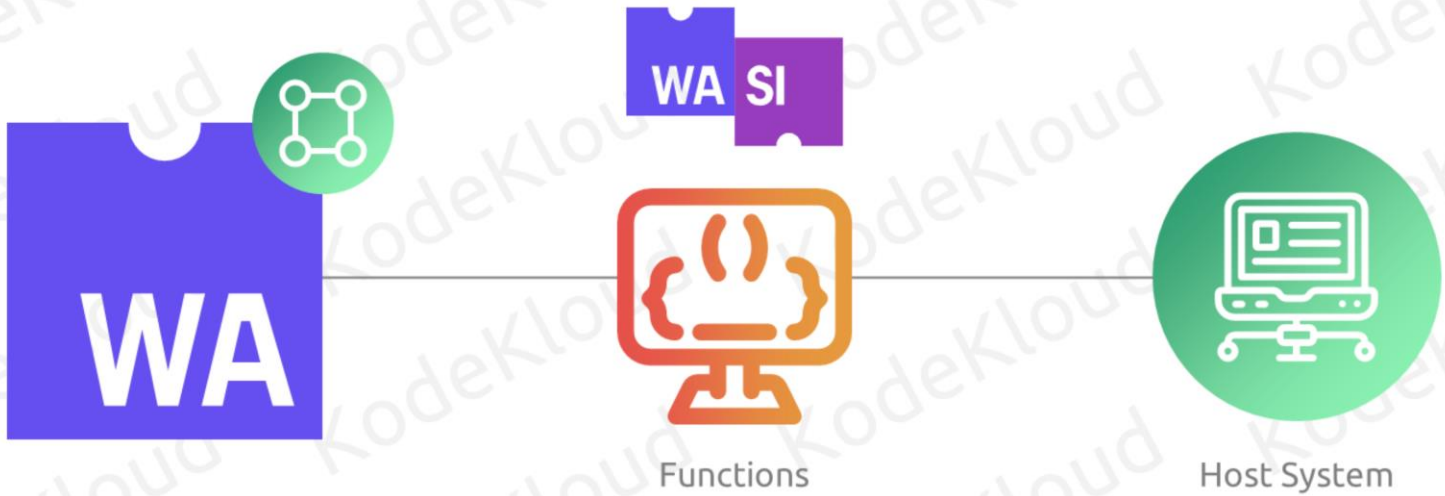
Standardized APIs:
APIs can be thought of as the language your application speaks. For your graphic processing application, this could be commands to adjust contrast, resize, or apply filters. With WASI, these commands are standardized.

WASI – Core Concepts

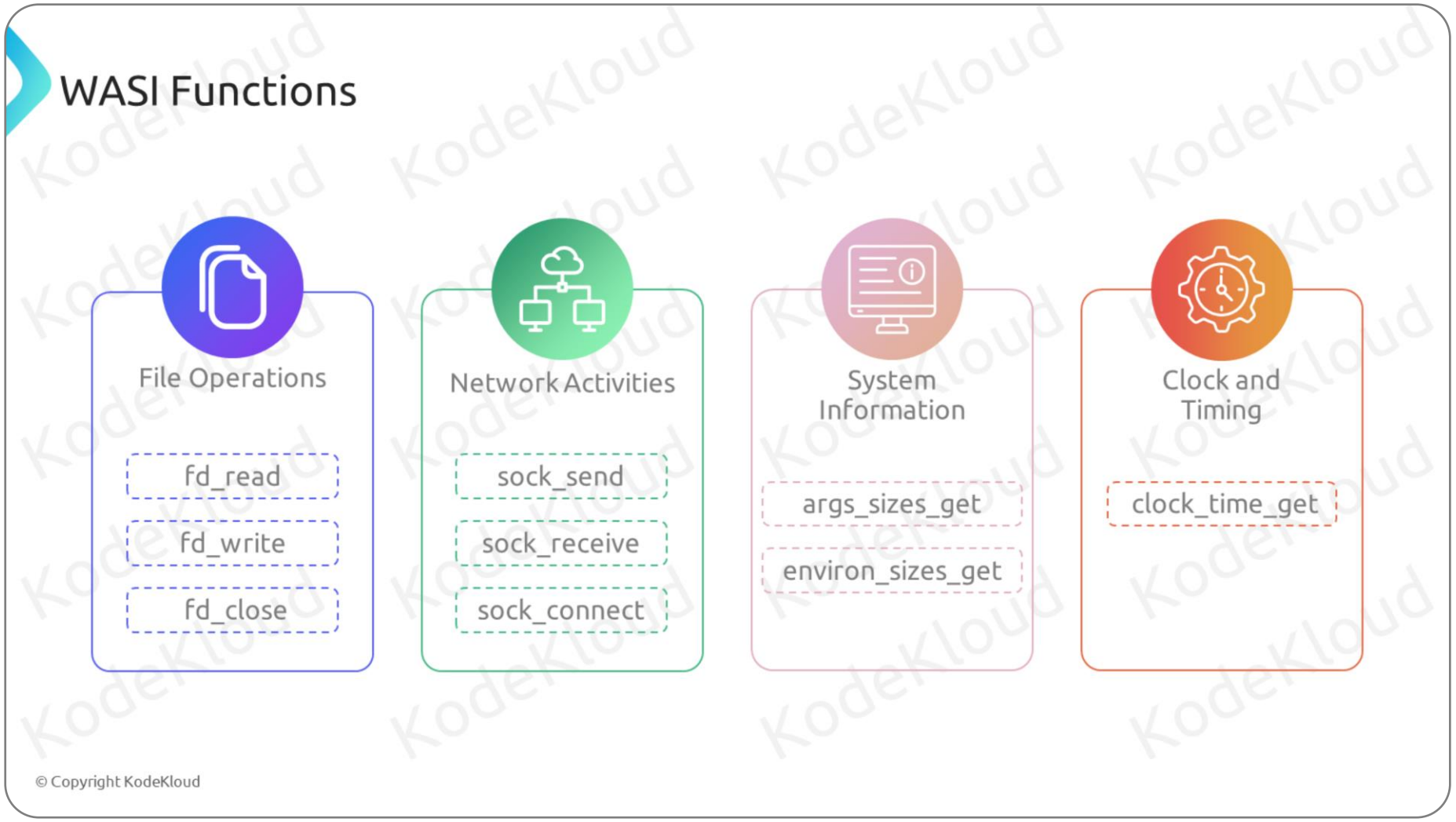
WASI – Core Concepts

Whether the application is run on a computer in Tokyo or an IoT device in San Francisco, the command to "increase brightness" would universally be understood and executed in the same way.

WASI Functions



WASI offers a suite of functions that serve as bridges between WebAssembly applications and the host system.



These functions encompass a broad spectrum of operations, such as:

File Operations: Functions like fd_read, fd_write, and fd_close allow applications to interact with the file system, enabling reading, writing, opening, and closing of files.
Network Activities: Functions such as sock_send, sock_receive, and sock_connect give WebAssembly the power to send, receive, and establish network connections.

System Information: Using functions like args_sizes_get or environ_sizes_get, applications can retrieve information about the environment they're running in, like command-line arguments or environment variables.
Clock & Timing: With functions like clock_time_get, applications can fetch the current time, facilitating time-based operations.

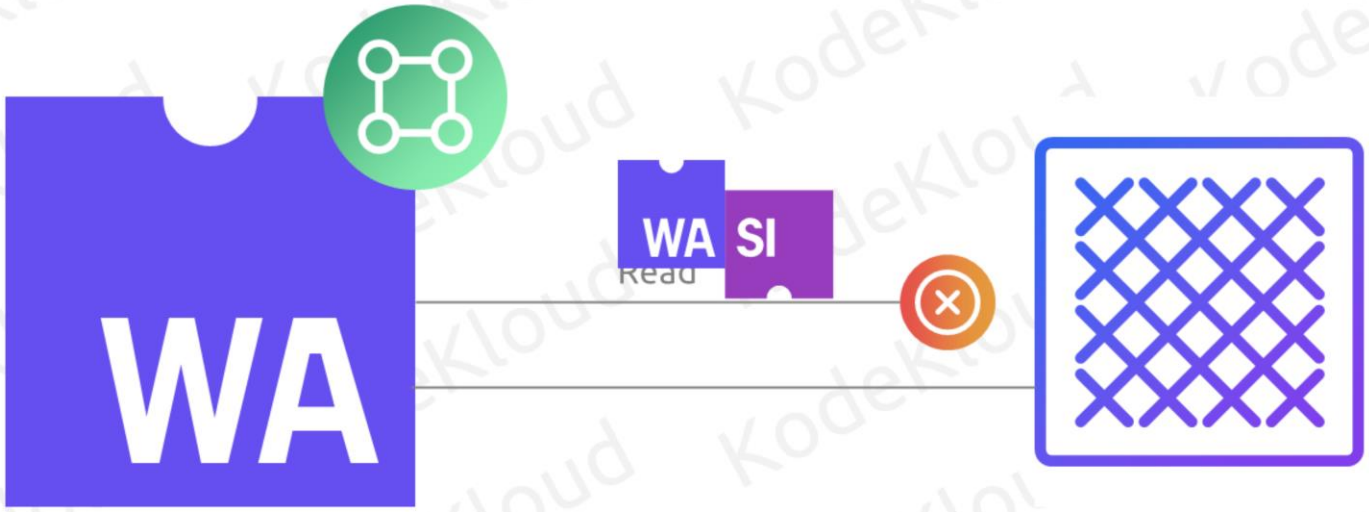
Armed with this diverse set of functions, WASI empowers WebAssembly applications to interact deeply and meaningfully with their host systems.

For more about WASI Functions

https://github.com/WebAssembly/WASI/blob/main/legacy/preview1/docs.md#sock_send

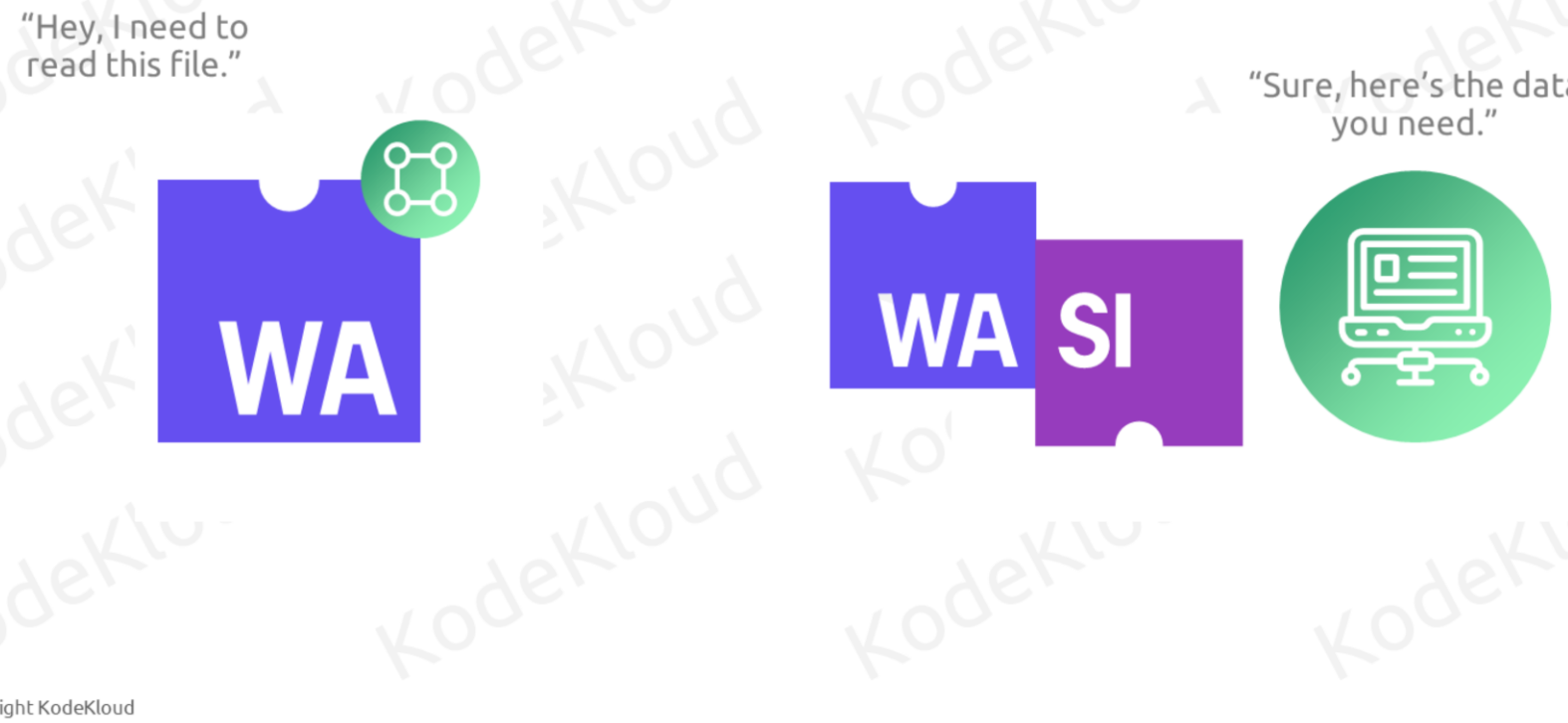
For more information on these WASI functions, please refer to the official WASI documentation here.
https://github.com/WebAssembly/WASI/blob/main/legacy/preview1/docs.md#sock_send

WASI in Action – A Simple Example



Consider our WebAssembly application designed for processing intricate graphics. Let's say it needs to read a specific texture file to apply to an image. Without a system bridge like WASI, this task would be near impossible.

WASI in Action – A Simple Example



However, with WASI, the application can easily import the fd_read function, specifically designed for reading operations. This function acts as a mediator, allowing the application to fetch the required file from the file descriptor. In essence, the WebAssembly application sends a request saying, "Hey, I need to read this file," and WASI responds with, "Sure, here's the data you need."

This example showcases the seamless interaction between WebAssembly and the system, facilitated by the tools provided by WASI.

WASI in Action – A Simple Example

```
(module
    ;; Import the WASI fd_read function
    (import "wasi_snapshot_preview1" "fd_read"
        (func $fd_read (param i32 i32 i32 i32) (result i32)))

    ;; Memory declarations and other necessary code...

    ;; A function to read data from a file
    (func $read_file
        ;; Assuming file descriptor 0 (standard input)
        (i32.const 0)                ;; file descriptor
        (i32.const data_offset)      ;; pointer to memory where data should be
stored
        (i32.const data_length)      ;; length of data to read
        (i32.const result_offset)    ;; pointer to memory where the result should
be stored
        (call $fd_read)

        ;; Handle the read data as necessary...
    )

    ;; Export our read_file function
    (export "read_file" (func $read_file))
)
```

WASI

In this example, the application imports the fd_read function from WASI to read from a file descriptor. This showcases how WASI provides WebAssembly applications with the tools they need to interact with the system.

A Future With WASI

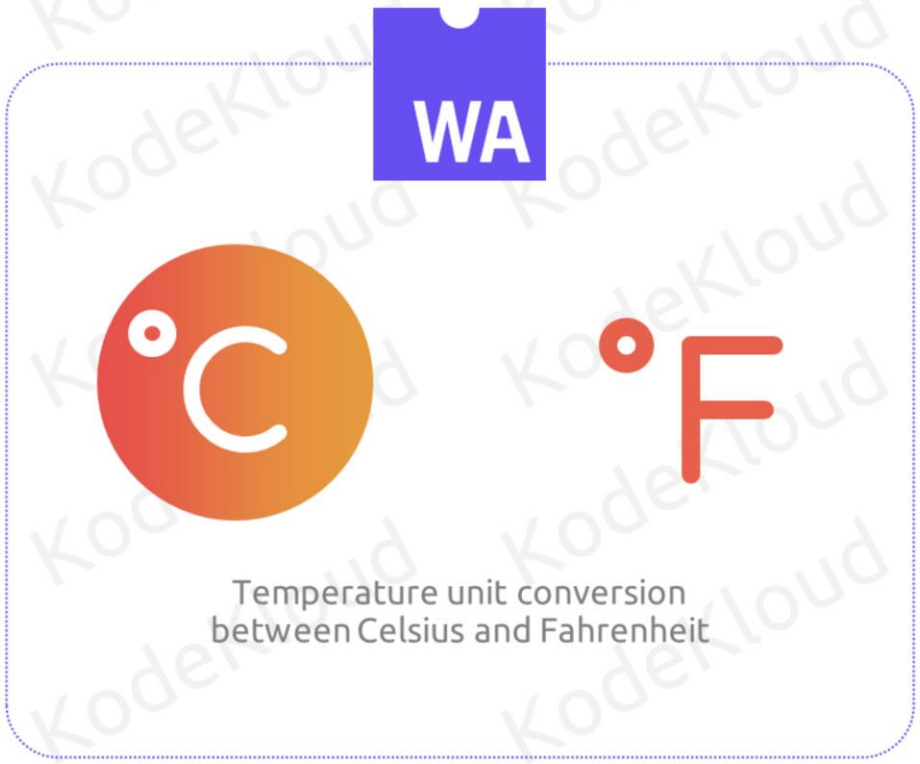
Imagine a future where the same piece of code works on your watch, phone, and computer without any extra effort.

A Future With WASI

That's what WASI aims to do. It helps programs talk to different devices easily. With WASI, moving apps between devices becomes a breeze, all while keeping things safe and secure. As more gadgets talk to each other, WASI's job becomes even more important.

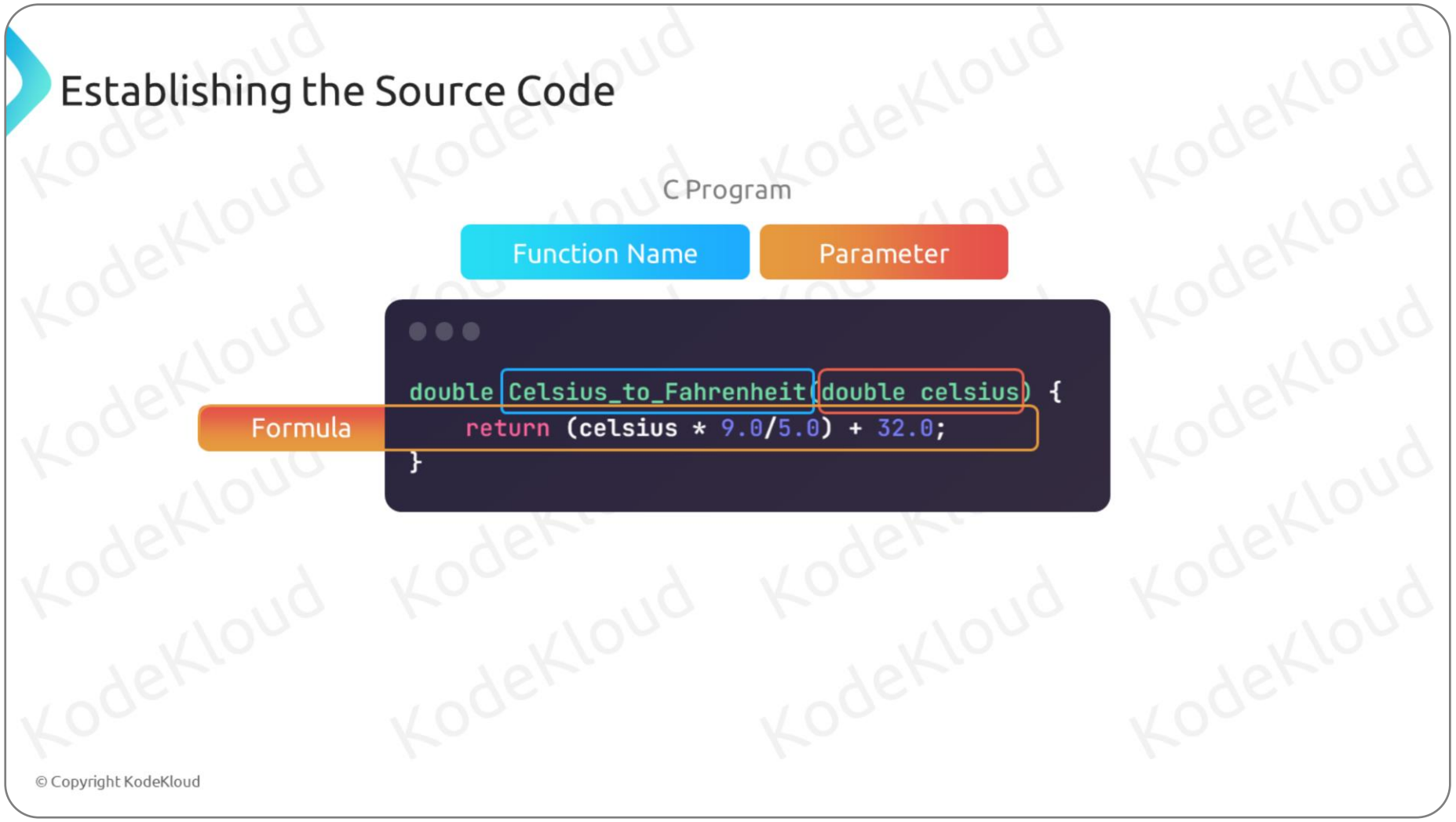
Creating a Simple WebAssembly Module

WebAssembly Module – Example



Let's dive into how we create a WebAssembly module. We'll use a simple temperature converter as our example, walking through each step from starting with code to getting a working module. Let's get started!

We're building a tool that converts temperature unit Celsius to Fahrenheit and vice versa. This function is essential for various applications, from weather websites to scientific calculators.



1. Establishing the Source Code:
Starting off, we need our basic function. Here's the C code that performs the temperature conversion:

Let's break it down:

The function is named celsius_to_fahrenheit and it takes one parameter, celsius, which is the temperature value we want to

convert.

Inside the function, we perform the conversion by multiplying the Celsius value by 9.0/5.0, and then adding 32.0 to the result. This formula is the standard way to convert Celsius to Fahrenheit.

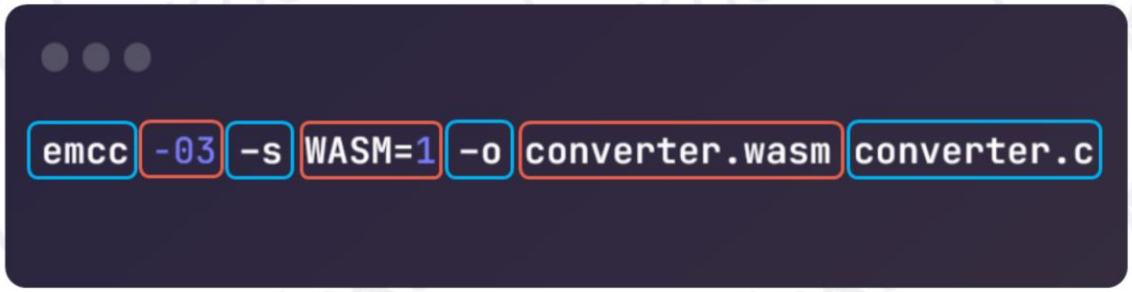
The function then returns the converted value.

Transition to WebAssembly

```c
double Celsius_to_Fahrenheit(double celsius) {
    return (celsius * 9.0/5.0) + 32.0;
}
```

2. Transition to WebAssembly:
Now, let's convert this C program into a WebAssembly module by compiling. We will be diving deeper into the process of compiling into WebAssembly and exploring the associated tools in upcoming lessons. But for this lesson, we'll introduce a command and provide a bit of background to make things clearer.

Emscripten and emcc – A Brief Introduction



Emscripten is a standout tool in the WebAssembly landscape, allowing us to turn C and C++ code into WebAssembly or even JavaScript. At its heart is the emcc command, which acts as the compiler. It's our key to making traditional code ready for the web.

For our temperature converter, we use:

// code //

This command, as previously explained, optimizes our code, ensures the output is in the WebAssembly format, and names the output file accordingly.

While this gives you a glimpse of the WebAssembly transformation process, there's a vast world of tools and techniques in WebAssembly compilation that we'll uncover in future lessons.

Let's understand this command step-by-step:

emcc: This is the Emscripten compiler. Emscripten is a toolchain that allows us to compile C and C++ code into WebAssembly. By using emcc, we're telling our system to utilize this compiler for the upcoming task.

-O3: This flag is an optimization instruction. In simpler terms, it tells the compiler to make our code as fast and efficient as possible. The 3 in -O3 is a level of optimization, with 3 being one of the highest levels, ensuring our code is both speedy and compact.

In our upcoming lesson on the Emscripten compiler, we'll delve deeper into optimization flags. These flags typically start at -O0. It's important to note that if no specific -O level is set in the command, then most optimizations are entirely disabled. This default behavior ensures that without explicit optimization instructions, the compiler focuses on reducing compilation time.

-s WASM=1: This flag is an instruction to output WebAssembly. The -s stands for setting, and WASM=1 specifies that we want a WebAssembly output. Without this, Emscripten might default to producing asm.js, an older format.

-o converter.wasm: The -o flag indicates the output file's name. Here, we're specifying that our output file should be named converter.wasm.

converter.c: This is simply the name of our input C file that contains the temperature conversion function.

Emscripten and emcc – A Brief Introduction

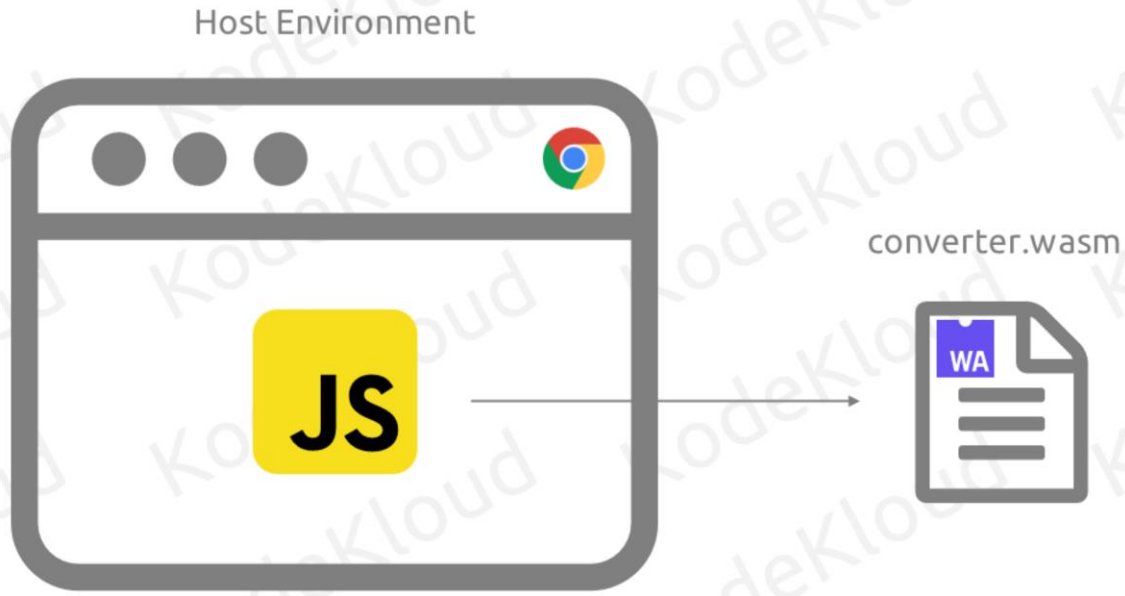
When we run this command, Emscripten takes our C file, processes it with the specified optimizations, and outputs a .wasm binary file or we call it as a WASM module. This file is a compact, optimized version of our original function, ready to be used in web environments.

Ever wondered how all this WebAssembly stuff works behind the scenes? It's pretty cool! We'll dig into the nitty-gritty of it in our upcoming lesson: 'Introduction to Compiling to WebAssembly'

Glimpse of the Next Lesson – Running a WASM Module

We've walked through the process of creating a WebAssembly module, understanding its foundation and how it's structured. But our journey with this temperature converter module isn't over yet. In our next lesson, we'll breathe life into this module by interacting with it.

Glimpse of the Next Lesson – Running a WASM Module



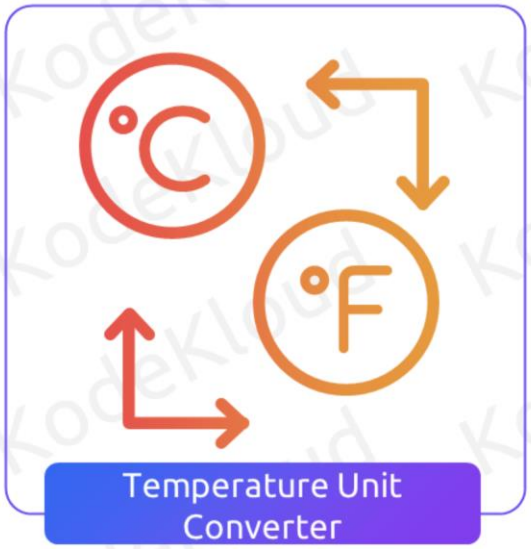
While our converter.wasm is a powerful tool, on its own, it's like an engine without a car. It needs a host environment to run, and in the context of the web, JavaScript is our driver.

Glimpse of the Next Lesson – Running a WASM Module

We'll explore how to fetch our .wasm module using JavaScript, initiate it, and then call our celsius_to_fahrenheit function. Along the way, we'll delve into WebAssembly's memory management, understanding how data flows between JavaScript and our WebAssembly module.

Running WebAssembly in the Browser

Introduction



Building on what we learned in our previous lesson, we've successfully created a WebAssembly module for our temperature unit converter. But how do we bring this module to life? The answer lies in our web browser.

Introduction

WebAssembly was initially designed to run in browsers, enhancing web performance. In this context, we'll explore how to run our WebAssembly module, the temperature unit converter, in a browser environment. But it's worth noting that the versatility of WebAssembly extends beyond browsers. And we'll delve into how it operates outside the browser in subsequent lessons.

JavaScript Bridges the Gap for WebAssembly in the Browser

To bring our WebAssembly module to life in the browser, we lean on JavaScript. It acts as the intermediary, bridging the gap between the WebAssembly module and the browser.

JavaScript Bridges the Gap for WebAssembly in the Browser

```
fetch('converter.wasm')
```

```
WebAssembly.instantiateStreaming(fetch('converter.wasm')) .then(obj => {
    // The module is now ready for action
});
```


JavaScript is responsible for fetching the WebAssembly module. It does this using the fetch API:

// code //

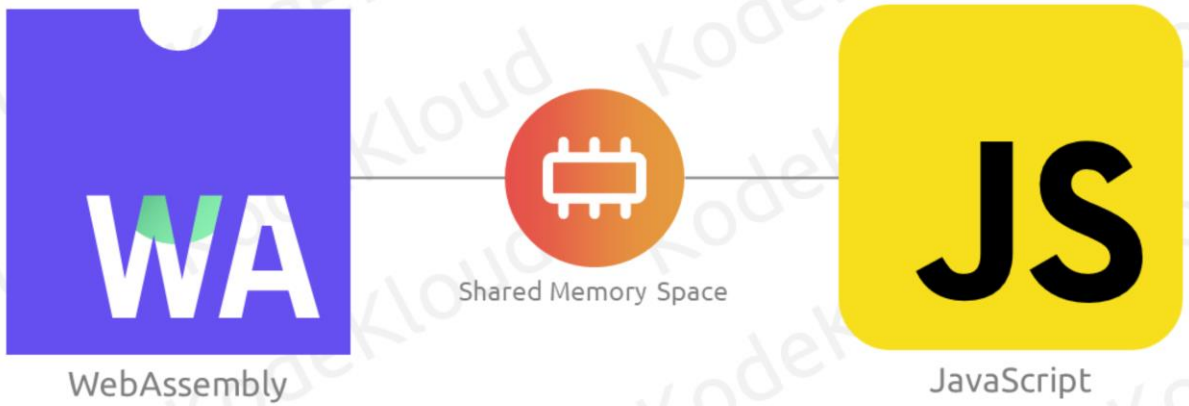
Once the module is fetched, it needs to be initialized. This is typically done using the WebAssembly.instantiateStreaming() function. However, it's part of the WebAssembly JavaScript API, which involves some coding intricacies and advanced

concepts. We won't delve deep into the WASM JS API in this course, but it's a topic to go through in the next lesson.

// code //

JavaScript Bridges the Gap for WebAssembly in the Browser

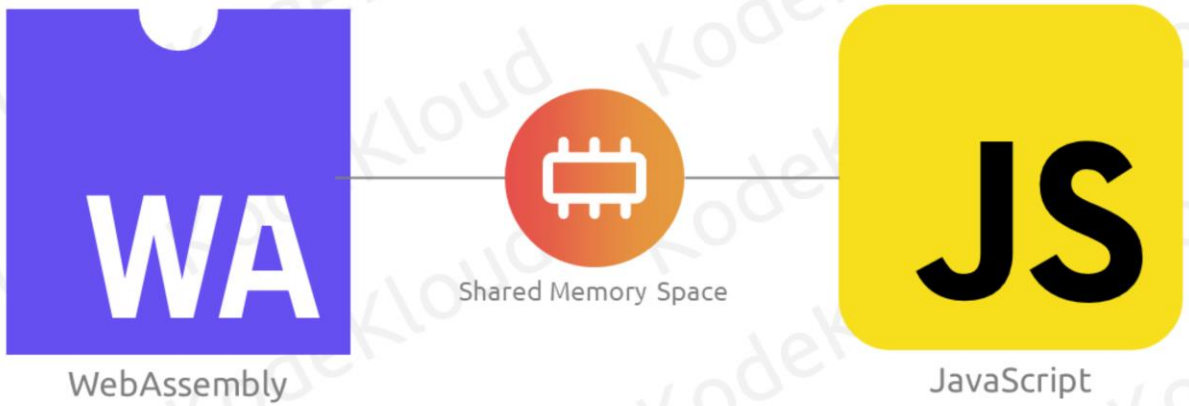
```
let result = obj.instance.exports.Celsius_to_Fahrenheit(25);
console.log('25°C is ${result}°F');
```



For our current purpose, just know that this function allows the WebAssembly module to be prepared and ready for action. Once initialized, we can call its functions. For instance, with our temperature unit converter, we can invoke the celsius_to_fahrenheit function:

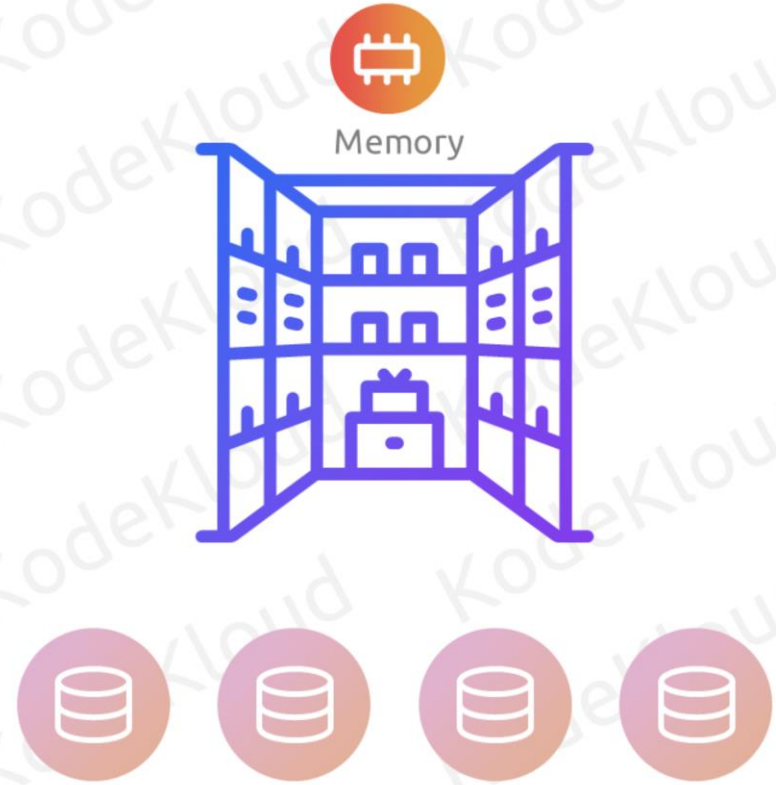
JavaScript Bridges the Gap for WebAssembly in the Browser

```
let result = obj.instance.exports.Celsius_to_Fahrenheit(25);
console.log('25°C is ${result}°F');
```



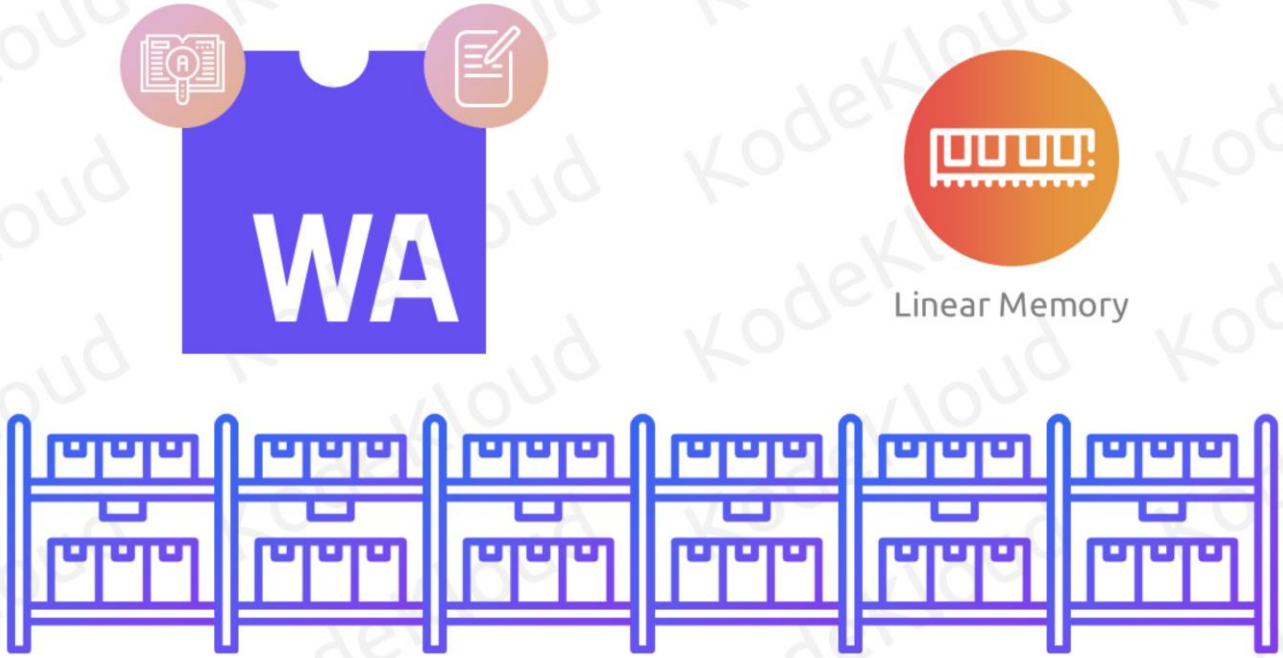
The interaction between JavaScript and WebAssembly occurs through a shared memory space. When we input a temperature value, it's placed into this shared space. After the conversion, the result is also stored here, making it easily accessible by JavaScript.

Understanding WebAssembly's Linear Memory

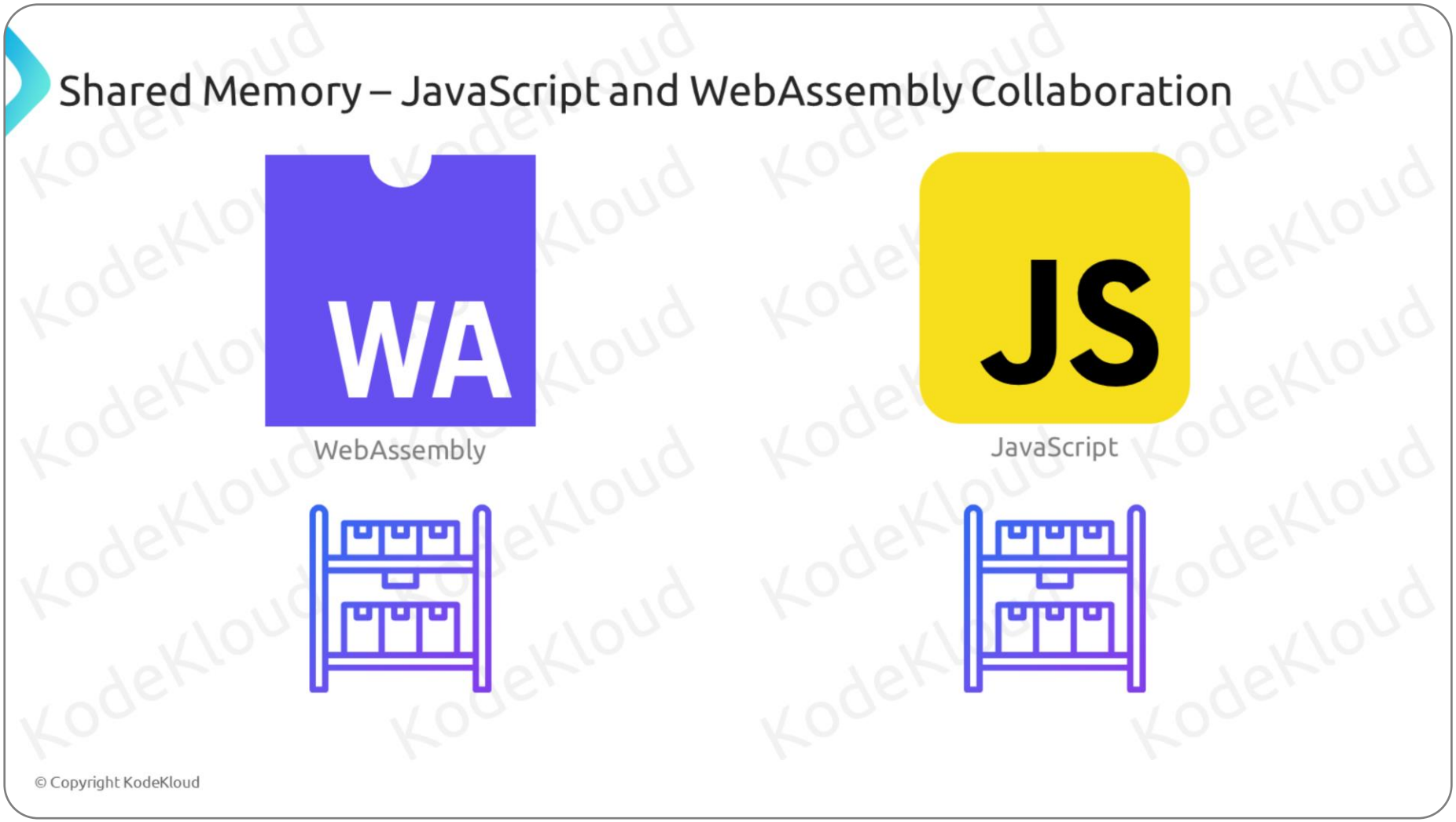


In the realm of computing, memory is like a vast storage room where data is kept. When we talk about WebAssembly and JavaScript sharing memory, imagine this storage room being divided into sections, with each section having its own shelves to store data.

Understanding WebAssembly's Linear Memory

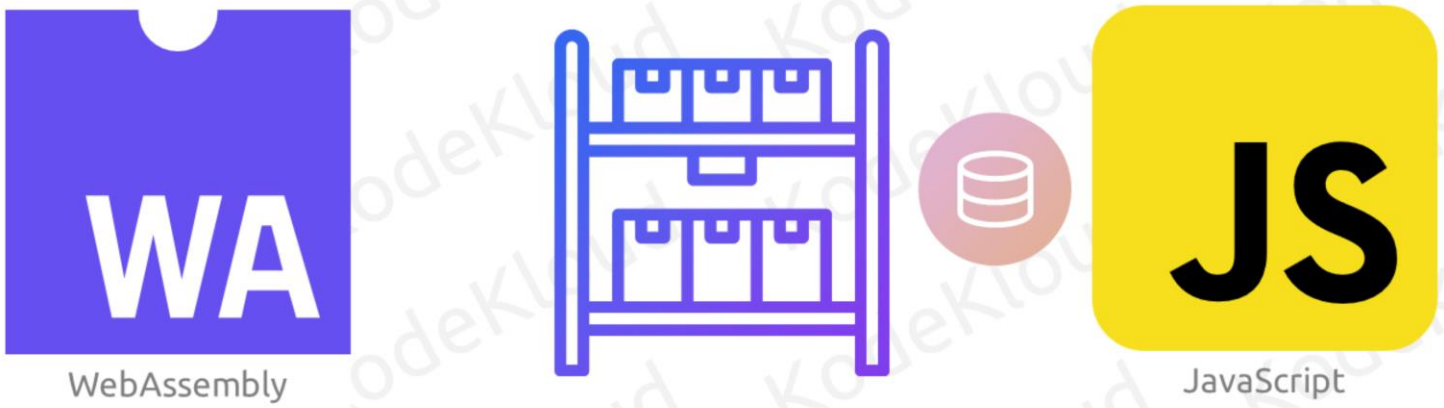


WebAssembly has its own dedicated section called "linear memory." It's a continuous block of memory, much like a long shelf, where data is stored sequentially. This linear memory is the primary way WebAssembly reads and writes data.



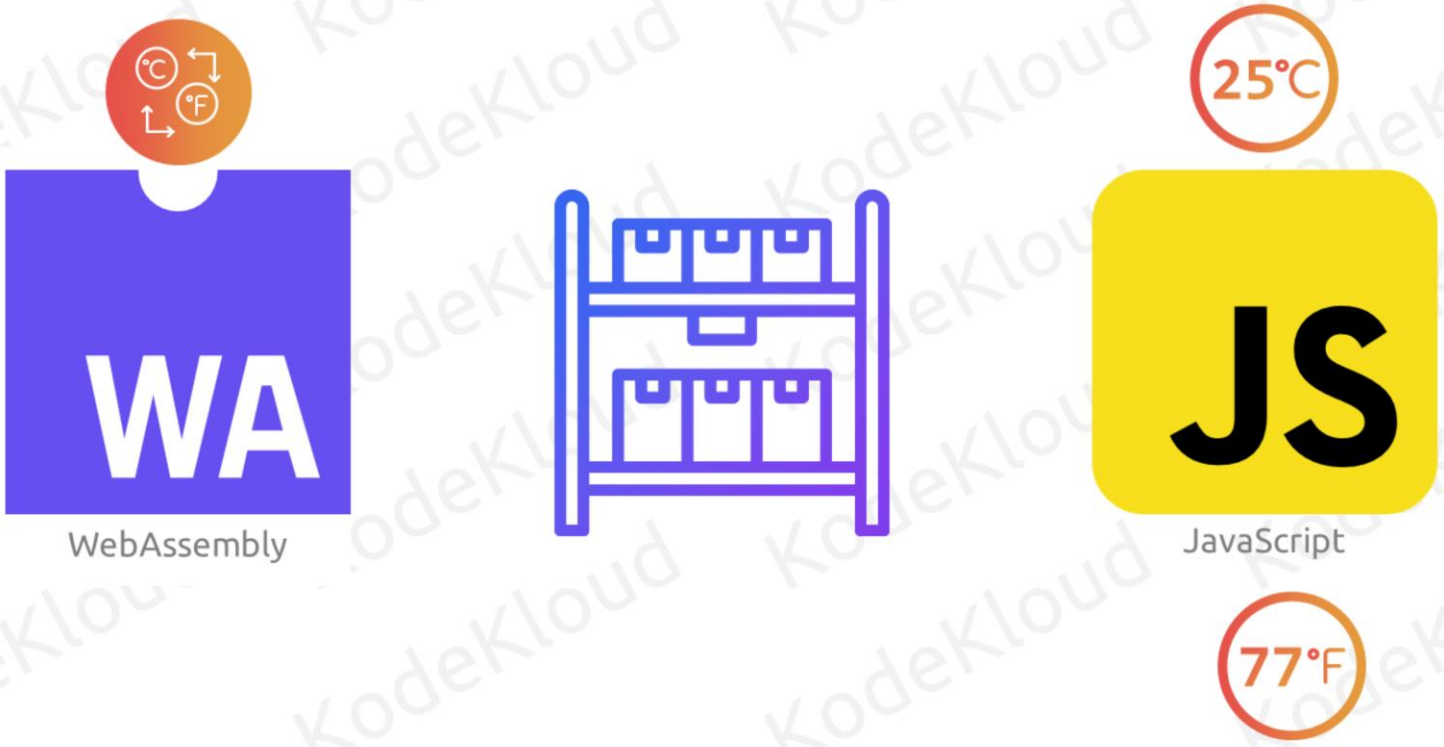
Now, here's where the collaboration between JavaScript and WebAssembly becomes interesting. Instead of keeping their data in completely separate rooms, they decide to share a section of this storage room. This shared section is the aforementioned linear memory.

Shared Memory – JavaScript and WebAssembly Collaboration



When JavaScript wants to send data to WebAssembly, it places the data on this shared shelf. Similarly, when WebAssembly wants to send results back to JavaScript, it puts the results on the same shelf. This way, both can easily access and exchange data without having to move between different rooms.

Shared Memory – JavaScript and WebAssembly Collaboration



For instance, with our temperature unit converter:

JavaScript wants to convert 25°C to Fahrenheit. It places the number 25 on the shared shelf (linear memory).
WebAssembly picks up the number, processes the conversion, and then places the result (77°F) back on the shared shelf.
JavaScript then retrieves the result from the shared shelf and displays it.

This shared memory system ensures a smooth and efficient data exchange. It eliminates the need for complex data transfers and allows both JavaScript and WebAssembly to work in harmony, making operations like our temperature unit conversion swift and seamless.

Creating a Simple Web Page in WebAssembly

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Temperature Converter</title>
</head>
<body>
  Enter Celsius: <input id="celsiusInput" type="number">
  <button onclick="convert()">Convert</button>
  <div id="output"></div>
  <script src="script.js"></script>
</body>
</html>
```


To see our WebAssembly module in action, we'll create a simple web page. Here's how:

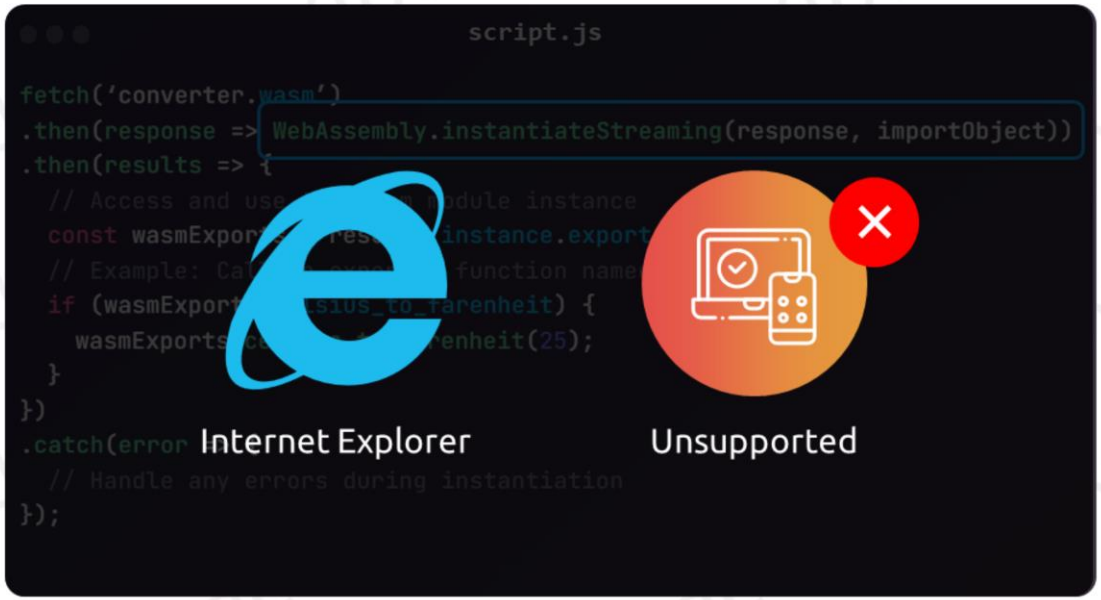
Let’s create an HTML page. This is our display area. We'll have a spot to enter a temperature in Celsius and a button to get the Fahrenheit equivalent. From here, we are showing how students can run this wasm module with simple HTML file along with some JS.

// code //

JavaScript Code for WebAssembly Instantiation

Next, we need some Javascript code to make things work. In a file named script.js, we'll add the code to instantiate wasm module as mentioned before.

JavaScript Code for WebAssembly Instantiation



This code uses instantiateStreaming to fetch and instantiate the WebAssembly module in one step. It's a cleaner and potentially faster approach, but always ensure that the target browsers or environments support it before using it in production.

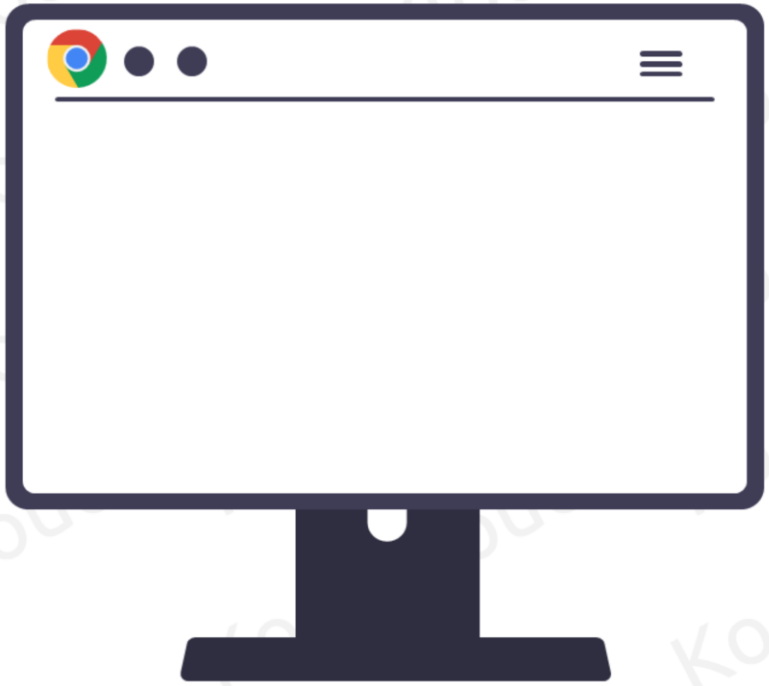
Ensuring Compatibility With WebAssembly Instantiate Options

```
let wasmModule;

fetch('converter.wasm')
  .then(response => response.arrayBuffer())
  .then(bytes => WebAssembly.instantiate(bytes))
  .then(results => {
    wasmModule = results.instance;
  });

function convert() {
  const celsius =
parseFloat(document.getElementById('celsiusInput').value);
  const fahrenheit = wasmModule.exports.celsius_to_fahrenheit(celsius);
  document.getElementById('output').innerText = `That's
${fahrenheit.toFixed(2)}°F!`;
}
```


Some older browsers or environments might not support instantiateStreaming. In that case, the WebAssembly.instantiate() approach would be the best option as shown here.

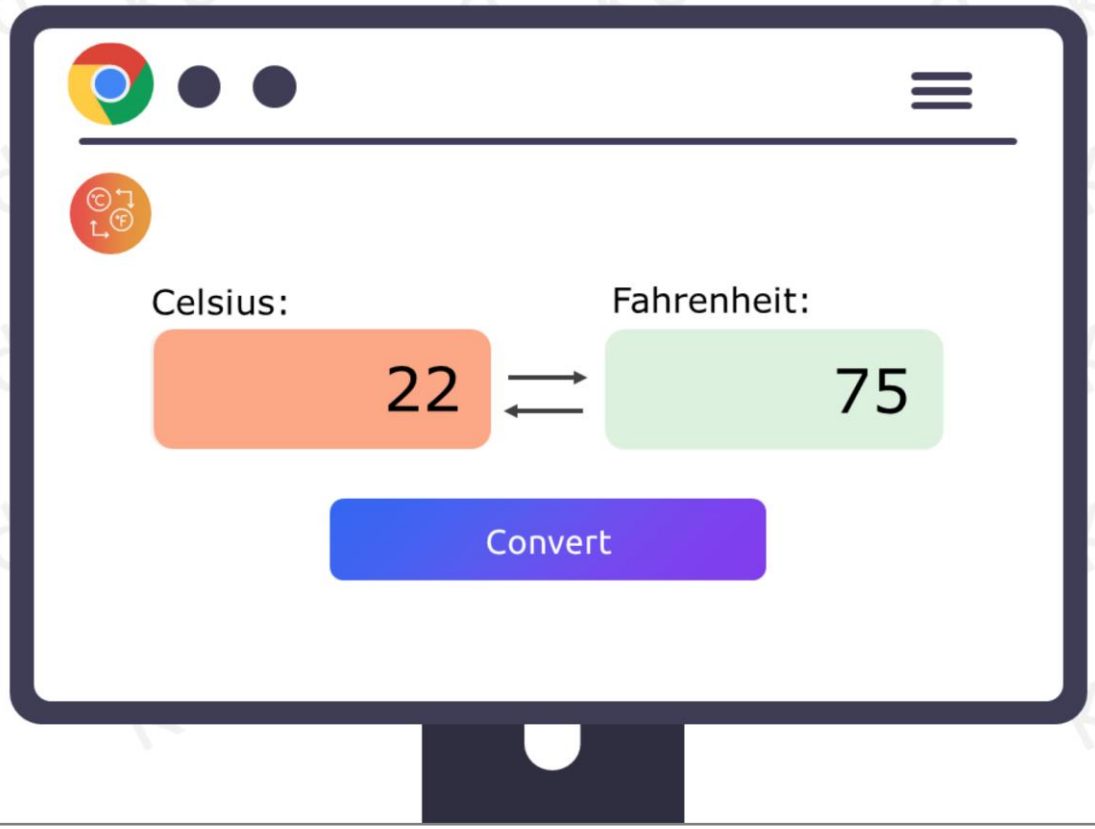
// code //

Ensuring Compatibility With WebAssembly Instantiate Options



Now, Open the HTML file in your browser. Type a number into the input box, click "Convert", and you'll see the result in Fahrenheit.

Ensuring Compatibility With WebAssembly Instantiate Options

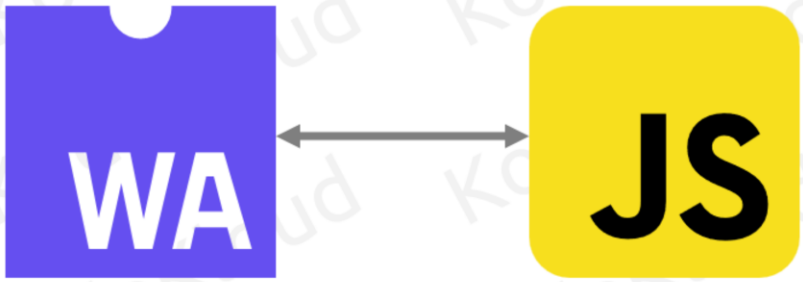


That's it! We've got a simple web page where you can convert Celsius to Fahrenheit using WebAssembly. This is just a basic example, but it shows how WebAssembly and JavaScript can work together in the browser.

Now that you've seen how to create and run a WebAssembly module in a browser environment, you might be wondering how it all fits together. After all, JavaScript has its own engine to run, so how does WebAssembly fit into the picture?

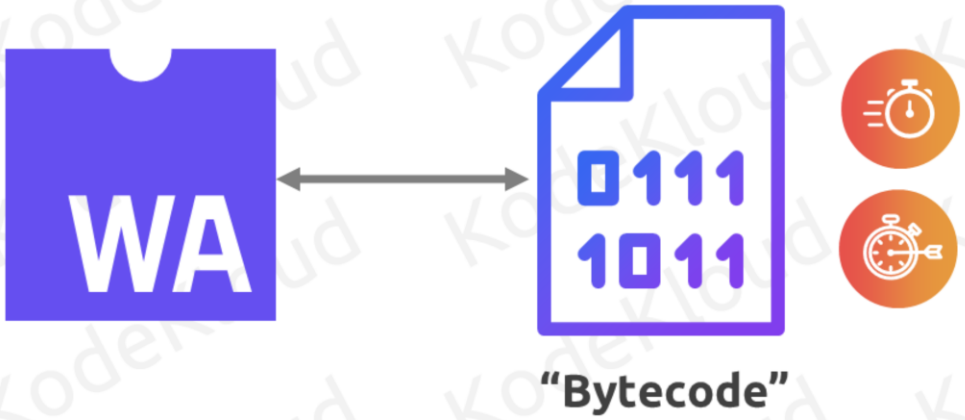
To understand WebAssembly's place, we first need to look at JavaScript. Every modern browser comes equipped with a JavaScript engine, like V8 in Chrome or SpiderMonkey in Firefox. This engine reads the JavaScript code, compiles it into machine code, and then runs it. It's a well-oiled machine, optimized over years to make JavaScript run efficiently.

WASM Browser Runtime



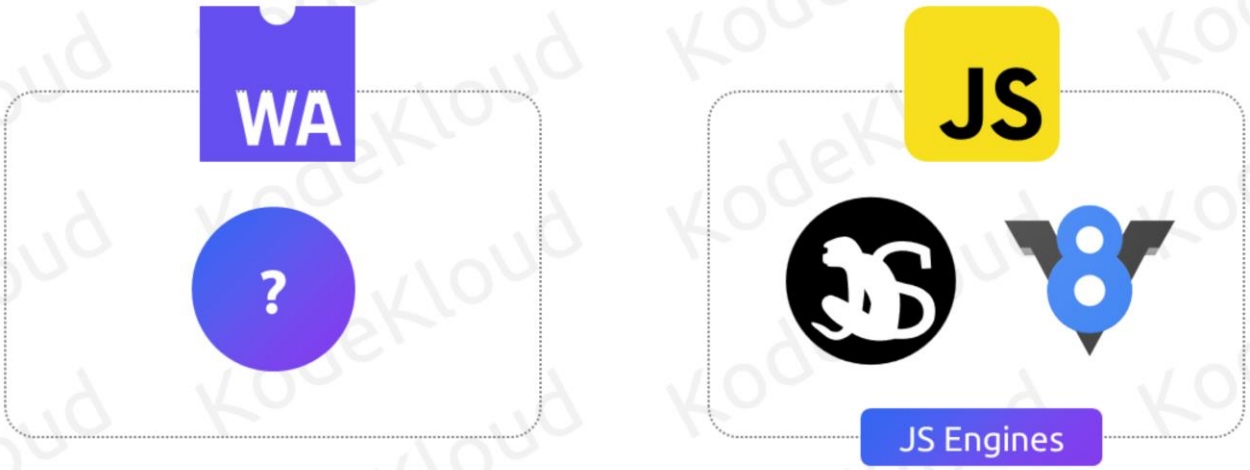
WebAssembly, or WASM, is a different technology.

WASM Browser Runtime



While it's designed to complement JavaScript, it's not just another scripting language.

WASM Browser Runtime



Instead, it's a compact binary format, a bytecode, that's designed for quick decoding and execution. But if JavaScript has its engine, how does WebAssembly run?

WASM Browser Runtime

The beauty of this integration is that WebAssembly doesn't need an entirely separate engine. Instead, the existing JavaScript engines have been extended to handle WebAssembly. So, when a browser encounters a WebAssembly module, it's the JavaScript engine that takes charge, but with a different set of tools tailored for WebAssembly.

WASM Browser Runtime

Google Chrome

Mozilla Firefox

Apple Safari

Microsoft Edge

Most modern browsers support WebAssembly, including:
Google Chrome
Mozilla Firefox
Apple Safari
Microsoft Edge

By understanding the WebAssembly browser runtime, we can see how it seamlessly integrates with the existing JavaScript ecosystem, providing a powerful tool for developers to enhance web performance.

WebAssembly JavaScript API

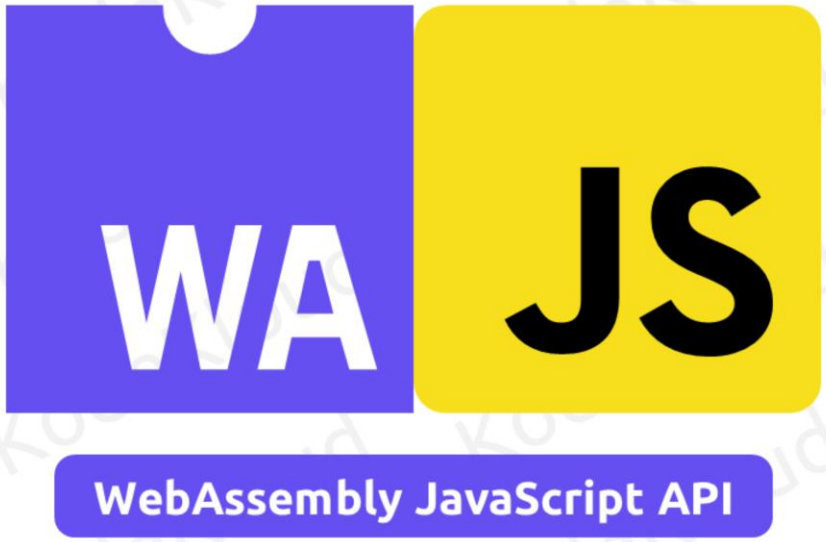
Introduction

WebAssembly

JavaScript

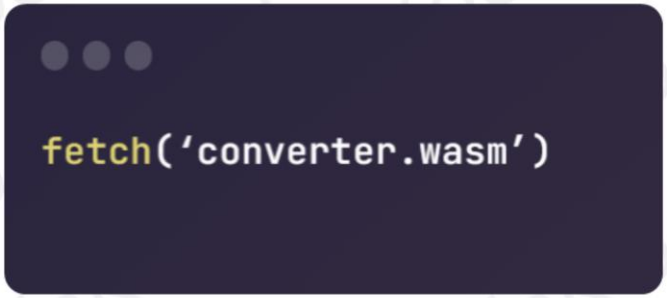
In our last lesson, we delved into the basics of running a WebAssembly (Wasm) module in the browser. We explored how WebAssembly seamlessly integrates with traditional web technologies like JavaScript and HTML.

Introduction



A pivotal part of this integration is the "glue" that binds Wasm with JavaScript, allowing them to work in harmony. This "glue" is facilitated by the WebAssembly JavaScript API.

Introduction



In our previous lesson, we utilized the following code to download the Wasm module.

Introduction

```
fetch('converter.wasm')
```

This code loads the binary file "converter.wasm" from the internet. You can also fetch the Wasm network resource using the Javascript XMLHttpRequest method. The fetch() API is a standard JS method which we reuse to load wasm binary files.

WebAssembly Modules using Two Pivotal Methods

WebAssembly.instantiate()

```
$:let wasmModule;

fetch('converter.wasm')
  .then(response => response.arrayBuffer())
  .then(bytes => WebAssembly.instantiate(byte))
  .then(results => {
    wasmModule = results.instance;
  });

function convert() {
  const celsius = parseFloat(document.getElementById('celsiusInput').value);
  const fahrenheit = wasmModule.exports.celsius_to_fahrenheit(celsius);
  document.getElementById('output').innerText = `That's ${fahrenheit.toFixed(2)}°F!`;
}
```


In previous lesson's hands-on examples, we instantiated our WebAssembly modules using two pivotal methods: WebAssembly.instantiate() and WebAssembly.instantiateStreaming(). These methods provided us with a practical understanding of how WebAssembly modules come to life in the browser.

WebAssembly Modules using Two Pivotal Methods

WebAssembly.instantiateStreaming()

```
fetch('path_to_module.wasm')
.then(response => WebAssembly.instantiateStreaming(response, importObject))
.then(results => {
  // Access and use the Wasm module instance
  const wasmExports = results.instance.exports;
  // Example: Call an exported function named 'calculate'
  if (wasmExports.celsius_to_farenheit) {
    wasmExports.celsius_to_farenheit(25);
  }
})
.catch(error => {
  // Handle any errors during instantiation
});
```


Both WebAssembly.instantiate() and WebAssembly.instantiateStreaming() are derived from what we refer to as the WebAssembly JavaScript API. This API is the bridge that facilitates the interaction between JavaScript and WebAssembly.

The WebAssembly.instantiate() and WebAssembly.instantiateStreaming() being among the most fundamental. Their role is to compile and instantiate WebAssembly modules, making them ready for execution in the browser environment.

WebAssembly Modules using Two Pivotal Methods

WebAssembly JavaScript API

WebAssembly.instantiate()

```
let wasmModule;

fetch('converter.wasm')
  .then(response => response.arrayBuffer())
  .then(bytes => WebAssembly.instantiate(bytes))
  .then(results => {
    wasmModule = results.instance;
  });
function convert() {
  const celsius =
parseFloat(document.getElementById('celsiusInput').value);
  const fahrenheit = wasmModule.exports.celsius_to_fahrenheit(celsius);
  document.getElementById('output').innerText = `That's
${fahrenheit.toFixed(2)}°F!`;
}
```

WebAssembly.instantiateStreaming()

```
fetch('path_to_module.wasm')
.then(response => WebAssembly.instantiateStreaming(response, importObject))
.then(results => {
  // Access and use the Wasm module instance
  const wasmExports = results.instance.exports;
  // Example: Call an exported function named 'calculate'
  if (wasmExports.celsius_to_farenheit) {
    wasmExports.celsius_to_farenheit(25);
  }
})
.catch(error => {
  // Handle any errors during instantiation
});
```

Simple Breakdown of Fundamental Methods

```
const wasmCode = new Uint8Array([...]); // some binary code
const wasmModule = new WebAssembly.Module(wasmCode);
```


Well. Let's provide a simple breakdown of some other fundamental methods from the WebAssembly JavaScript API, apart from the two we've already discussed.

When diving into WebAssembly, the first thing you might want to do is create a WebAssembly module from some given data. For this, we can use WebAssembly.Module(). Imagine you have a set of building instructions; this method turns those instructions into a blueprint.

// code //

Simple Breakdown of Fundamental Methods

```
const wasmCode = new Uint8Array([...]); // some binary code
const wasmModule = new WebAssembly.Module(wasmCode);
const wasmInstance = new WebAssembly.Instance(wasmModule);
```


Once you have your blueprint (or module) ready, the next step is to build something from it. This is where WebAssembly.Instance() comes into play. It takes the blueprint and constructs a usable instance, much like building a house from a set of plans.

// code //

Simple Breakdown of Fundamental Methods

```
const wasmCode = new Uint8Array([...]); // some binary code
const wasmModule = new WebAssembly.Module(wasmCode);
const wasmInstance = new WebAssembly.Instance(wasmModule);
const memory = new WebAssembly.Memory({ initial: 10, maximum: 100});
```


Now, every house needs storage rooms to keep things. In the WebAssembly world, this storage is managed by WebAssembly.Memory(). It sets up memory storage for our WebAssembly module, ensuring there's space to store and retrieve data.

// code //

Simple Breakdown of Fundamental Methods

In the context of WebAssembly and its JavaScript API, the WebAssembly.Memory() constructor is used to create and manage a memory resource for WebAssembly modules. It sets up memory storage for our WebAssembly module, ensuring there's space to store and retrieve data.

// code //

As you could see in this example,

The WebAssembly.Memory is a built-in object in the WebAssembly JavaScript API. It's used to create and manage linear memory for WebAssembly modules.

The initial parameter specifies the initial size of the memory, in terms of WebAssembly memory pages. One WebAssembly memory page is 64KiB (Kibibytes, where 1 KiB = 1024 bytes). So, initial: 10 means the memory starts with 10 pages, which is
10×64 KiB=640 KiB10×64KiB=640KiB.

The maximum: This parameter sets the maximum size to which the memory can grow, also in WebAssembly memory pages. maximum: 100 means the memory can be expanded up to 100 pages if needed, equating to 100×64 KiB=6400 KiB100×64KiB=6400KiB, or 6.4 MiB (Mebibytes).

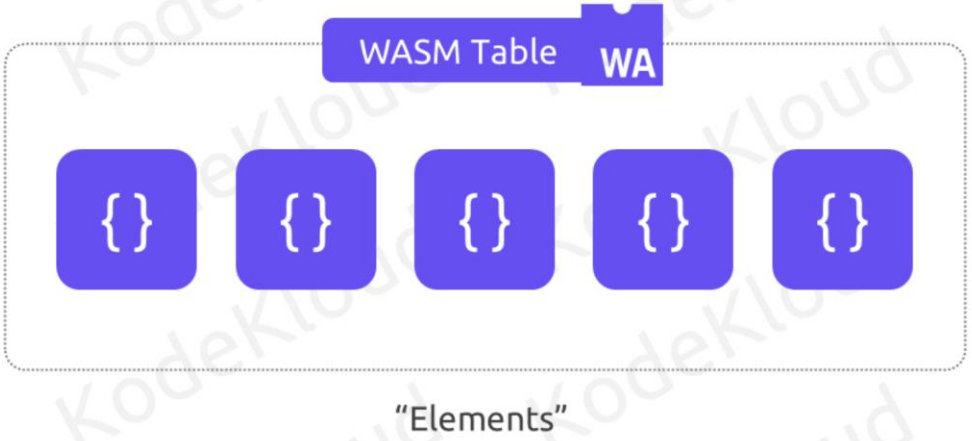
Simple Breakdown of Fundamental Methods

```
const memory = new WebAssembly.Memory({ initial: 10, maximum: 100 });
```

You should keep in mind that the 10, 100 numbers represent the number of memory pages, not the size in MB or GB. Each memory page in WebAssembly is a fixed size of 64KiB. So, 10 pages amount to 640KiB, and 100 pages amount to 6400KiB (or approximately 6.4MiB).

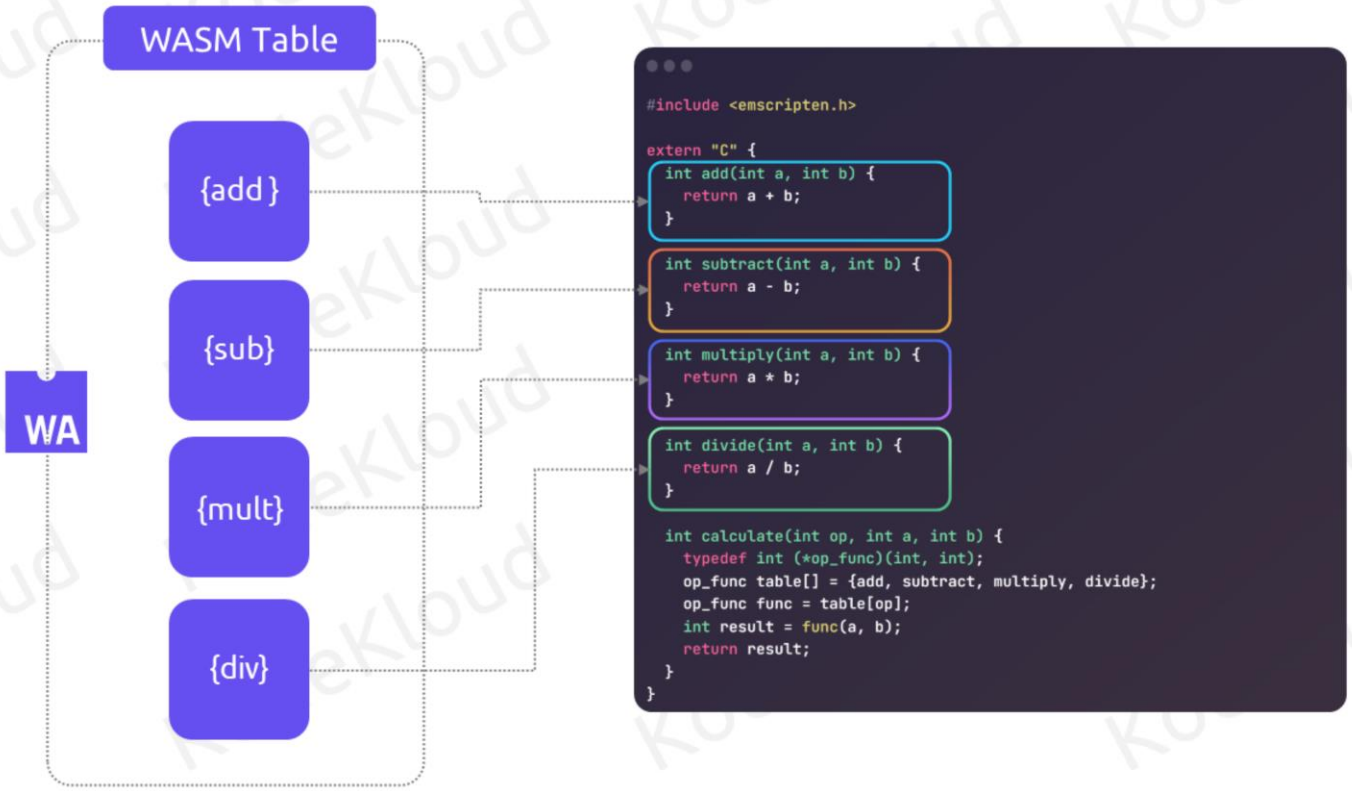
```
const wasmCode = new Uint8Array([...]); // some binary code
const wasmModule = new WebAssembly.Module(wasmCode);
const wasmInstance = new WebAssembly.Instance(wasmModule);
const memory = new WebAssembly.Memory({ initial: 10, maximum: 100});
const table = new WebAssembly.Table({ initial: 10, element: 'anyfunc'});
```

Simple Breakdown of Fundamental Methods



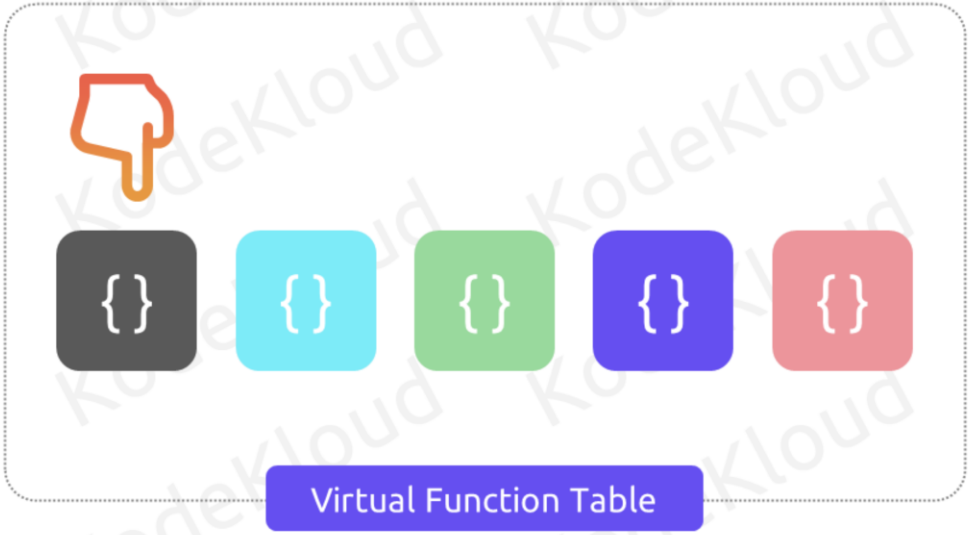
In addition to memory, sometimes we need a way to organize and call upon different functions in our WebAssembly module. A WebAssembly Table is a structure that holds references, typically to functions. These references are called "elements" in the table.

Simple Breakdown of Fundamental Methods



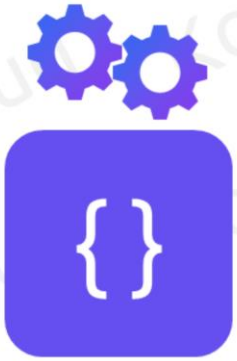
The table provides an indirect way to call these functions, which is particularly useful in certain programming scenarios like implementing virtual function tables or dynamic linking.

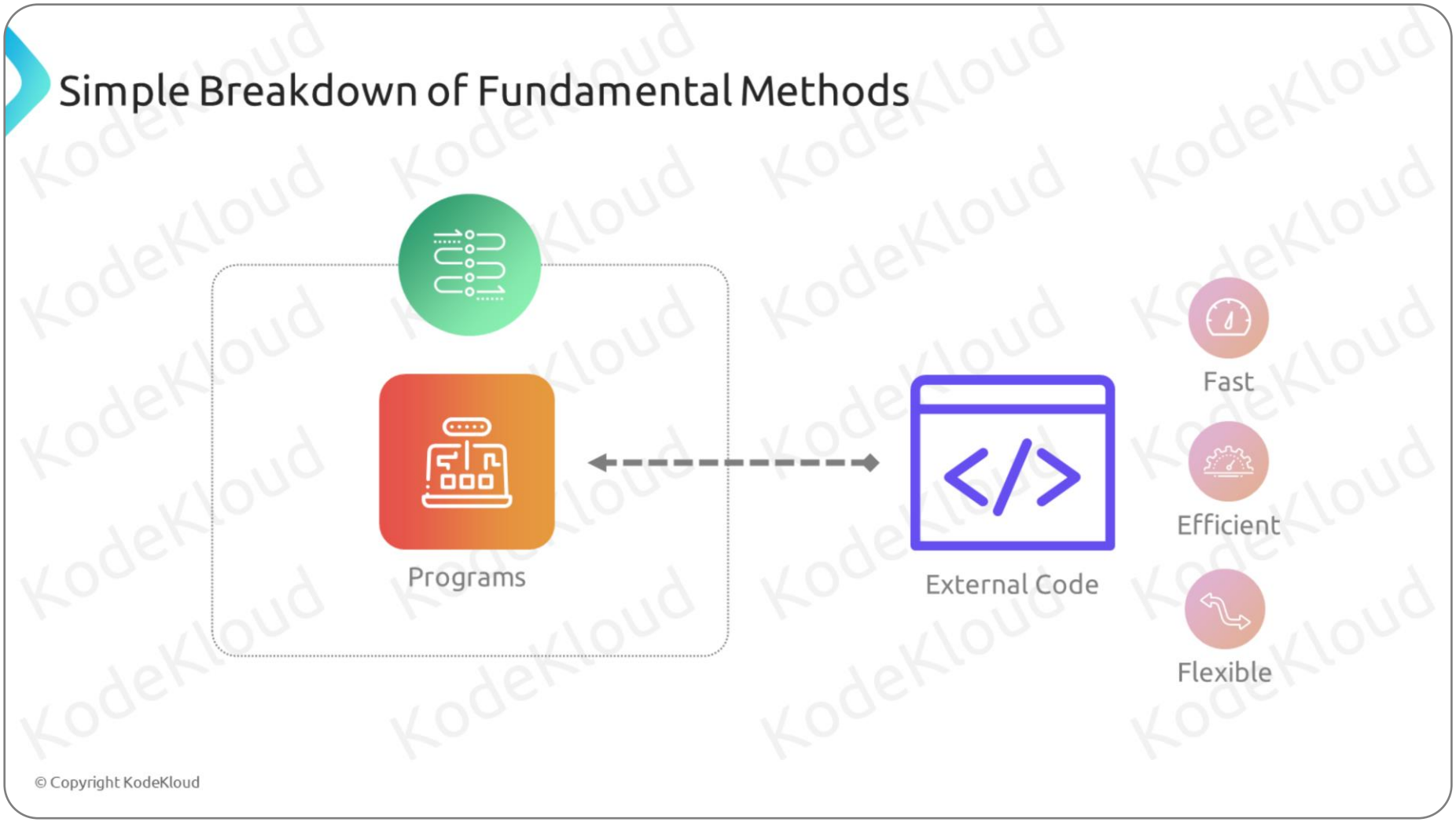
Simple Breakdown of Fundamental Methods



Virtual Function Table is like a control panel that helps decide which specific action (function) an object should perform, useful in scenarios where objects have different versions of the same function.

Simple Breakdown of Fundamental Methods





Dynamic Linking is like a machine using tools from a library as needed, allowing programs to use external code on the fly, making them more flexible and efficient.

Simple Breakdown of Fundamental Methods

```
const table = new WebAssembly.Table({ initial: 10, element: 'anyfunc' });
```


When you create a Table object, you specify its initial size and the type of elements it will hold.

Simple Breakdown of Fundamental Methods

```
const table = new WebAssembly.Table({ initial: 10, element: 'anyfunc' });
```


In this example,
initial: 10 specifies that the table should be initialized with space for 10 elements. It doesn't fill the table with functions; it just allocates space for them.
element: 'anyfunc' indicates that each element in the table will be a reference to a function. The 'anyfunc' type means that the table can hold references to any kind of function, regardless of its signature.

Let’s take a look at how it works internally,

Once you have a WebAssembly Table, you can store references to functions in it. These functions could be imported from JavaScript or defined in the WebAssembly module.

To call a function stored in a Table, you would typically use an indirect call mechanism. This means you specify the index of the function in the table rather than calling the function directly. This allows for a more dynamic approach to function invocation, which can be useful in complex applications.

The contents of a Table can be dynamically updated, meaning you can modify which functions are referenced at different indices as your program runs.

Simple Breakdown of Fundamental Methods

"WebAssembly Table stores references to functions"

WASM Module

JavaScript Module

"Once you have a WebAssembly Table, you can store references to functions in it. These functions could be imported from JavaScript or defined in the WebAssembly module.

To call a function stored in a Table, you would typically use an indirect call mechanism. This means you specify the index of the function in the table rather than calling the function directly. This allows for a more dynamic approach to function

invocation, which can be useful in complex applications.

The contents of a Table can be dynamically updated, meaning you can modify which functions are referenced at different indices as your program runs.
"

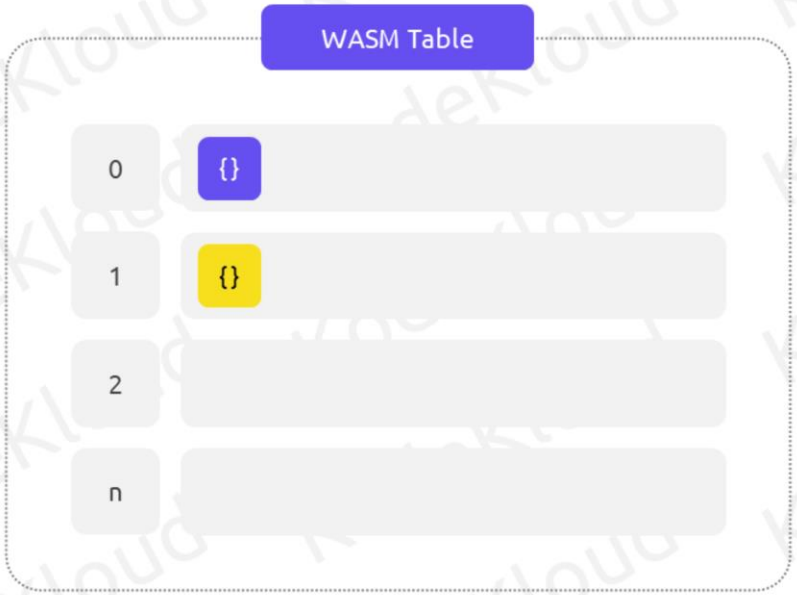
Simple Breakdown of Fundamental Methods

"WebAssembly Table stores references to functions"

WASM Module

JavaScript Module

Simple Breakdown of Fundamental Methods



"Importing and Defining Functions"

WASM Module

JavaScript Module

Simple Breakdown of Fundamental Methods

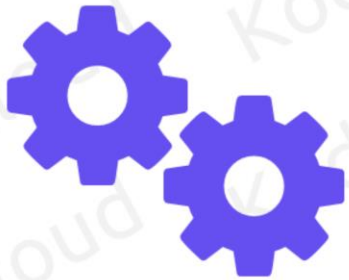
"Importing and Defining Functions"

WASM Module

JavaScript Module

Simple Breakdown of Fundamental Methods

Simple Breakdown of Fundamental Methods



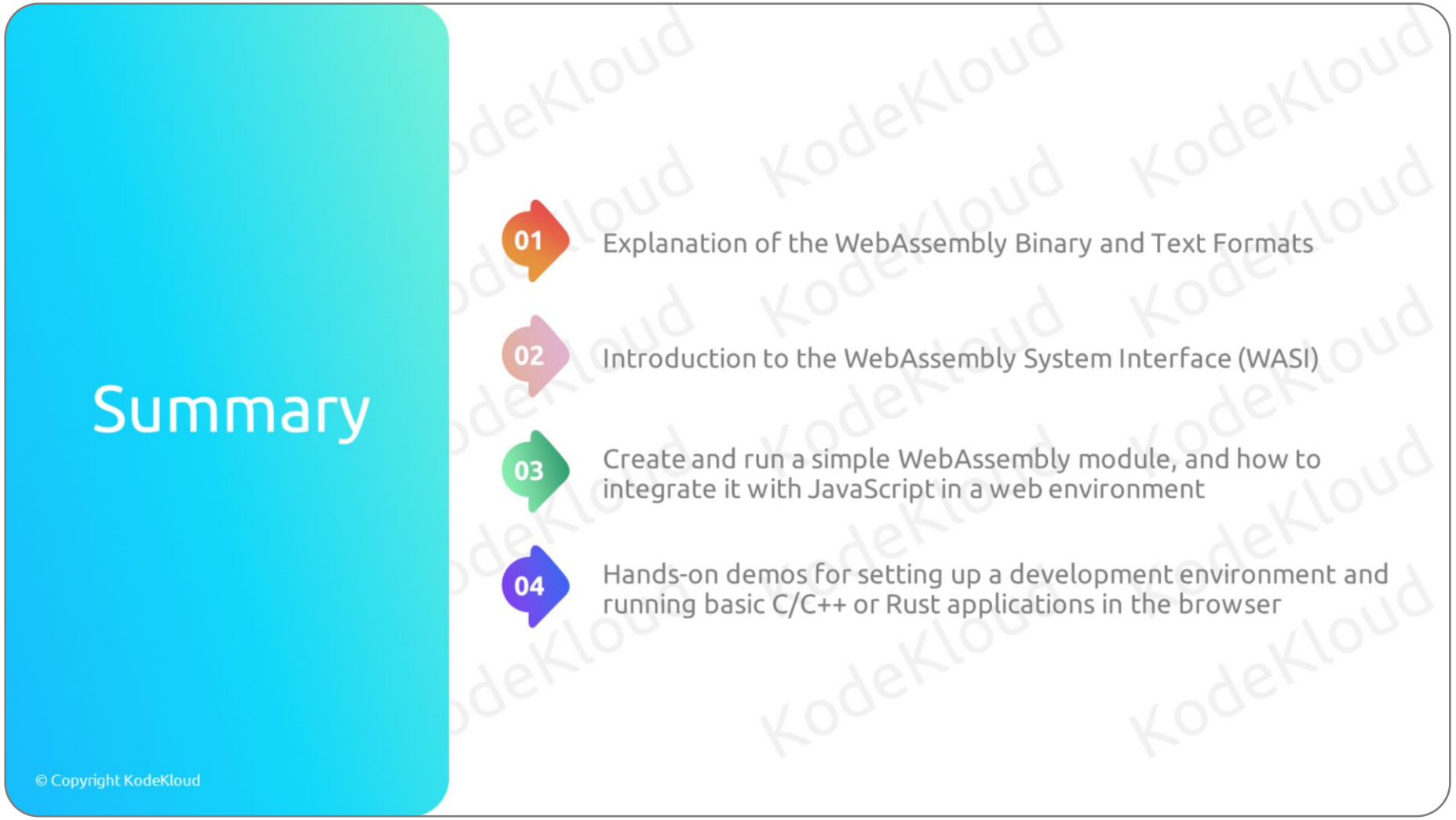
"Updates and changes during runtime"

Simple Breakdown of Fundamental Methods

WebAssembly provides specific error types to help us understand and fix issues. For instance, if there's a problem with the blueprint or the construction process, WebAssembly will let us know through errors like WebAssembly.CompileError, WebAssembly.LinkError, or WebAssembly.RuntimeError.

// code //

In essence, the WebAssembly JavaScript API provides us with the tools to seamlessly integrate WebAssembly into our browser, from creating modules to handling errors.



This practical section guides learners through the initial steps of working with WebAssembly. It begins with an explanation of the WebAssembly Binary and Text Formats, followed by an introduction to the WebAssembly System Interface (WASI). The section then moves on to practical applications, demonstrating how to create and run a simple WebAssembly module, and how to integrate it with JavaScript in a web environment. It includes hands-on demos for setting up a development environment and running basic C/C++ or Rust applications in the browser.

KodeKloud

Follow us on https://kodekloud.com/ to learn more about us.