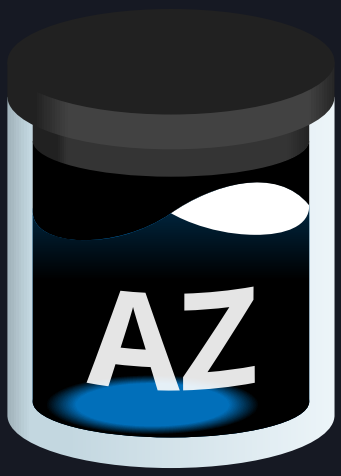
The Azle Book (Release Candidate)



Welcome to The Azle Book! This is a guide for building secure decentralized/replicated servers in TypeScript or JavaScript on ICP. The current replication factor is 13-40.

Please remember that Azle stable mode is continuously subjected to intense scrutiny and testing, however it has not yet undergone intense security review.

The Azle Book is subject to the following license and Azle's License Extension:

Candid RPC or HTTP Server

Azle applications (canisters) can be developed using two main methodologies: Candid RPC and HTTP Server.

Candid RPC embraces ICP's Candid language, exposing canister methods directly to Candid-speaking clients, and using Candid for serialization and deserialization purposes.

HTTP Server embraces traditional web server techniques, allowing you to write HTTP servers using popular libraries such as Express, and using JSON for simple serialization and deserialization purposes.

Candid RPC is now in the release candidate phase, and is heading towards 1.0 imminently.

HTTP Server will remain experimental for an unknown length of time.

Candid RPC

> Chapter 3 has been generated by AI based on our repository. The generated documentation is valuable, though it has not yet been meticulously reviewed by human beings. Please refer to the examples and the `JSDocs` of the imports from `azle` for the most up-to-date and accurate documentation.

This section documents the Candid RPC methodology for developing Azle applications. This methodology embraces ICP's Candid language, exposing canister methods directly to Candid-speaking clients, and using Candid for serialization and deserialization purposes.

Candid RPC is now in the release candidate phase, and is heading towards 1.0 imminently.

Quick Navigation

- Get Started - Installation and deployment guide
- Examples - Working examples and best practices
- Canister Class - How to structure your canister
- @dfinity/candid IDL - Type definitions and serialization
- Decorators - Method decorators for canister entry points
- IC API - Internet Computer platform APIs

Get Started

Azle helps you to build secure decentralized/replicated servers in TypeScript or JavaScript on ICP. The current replication factor is 13-40.

Please remember that Azle stable mode is continuously subjected to intense scrutiny and testing, however it has not yet undergone intense security review.

Azle runs in stable mode by default.

This mode is intended for production use after Azle's imminent 1.0 release. Its focus is on API and runtime stability, security, performance, TypeScript and JavaScript language support, the ICP APIs, and Candid remote procedure calls (RPC). There is minimal support for the Node.js standard library, npm ecosystem, and HTTP server functionality.

Installation

> Windows is only supported through a Linux virtual environment of some kind, such as WSL

You will need Node.js and dfx to develop ICP applications with Azle:

Node.js

It's recommended to use nvm to install the latest LTS version of Node.js:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.3/install.sh |
bash
```

Restart your terminal and then run:

```
nvm install --lts
```

Check that the installation went smoothly by looking for clean output from the following command:

```
node --version
```

dfx

Install the dfx command line tools for managing ICP applications:

```
DFX_VERSION=0.27.0 sh -ci "$(curl -fsSL https://internetcomputer.org/
install.sh)"
```

Check that the installation went smoothly by looking for clean output from the following command:

```
dfx --version
```

Deployment

To create and deploy a simple sample application called `hello_world`:

```
# create a new default project called hello_world
npx azle new hello_world
cd hello_world
```

```
# install all npm dependencies including azle
npm install
```

```
# start up a local ICP replica
dfx start --clean
```

In a separate terminal in the `hello_world` directory:

```
# deploy your canister
dfx deploy
```

Examples

Some of the best documentation for creating Candid RPC canisters is currently in the examples directory.

Basic Hello World

```
import { IDL, query } from 'azle';

export default class {
    @query([], IDL.Text)
    hello(): string {
        return 'Hello World!';
    }
}
```

Counter with State

```
import { IDL, query, update } from 'azle';

export default class {
    counter: number = 0;

    @query([], IDL.Nat32)
    get(): number {
        return this.counter;
    }

    @update([], IDL.Nat32)
    increment(): number {
        this.counter += 1;
        return this.counter;
    }

    @update([IDL.Nat32], IDL.Nat32)
    set(value: number): number {
        this.counter = value;
        return this.counter;
    }
}
```

User Management

```
import { IDL, msgCaller, Principal, query, update } from 'azle';

const User = IDL.Record({
    id: IDL.Principal,
    name: IDL.Text,
    age: IDL.Nat8
});
type User = {
    id: Principal;
    name: string;
    age: number;
};

export default class {
    users: Map<string, User> = new Map();

    @update([IDL.Text, IDL.Nat8], User)
    createUser(name: string, age: number): User {
        const caller = msgCaller();
        const user: User = {
            id: caller,
            name,
            age
        };

        this.users.set(caller.toText(), user);
        return user;
    }

    @query([], IDL.Vec(User))
    getUsers(): User[] {
        return Array.from(this.users.values());
    }

    @query([IDL.Principal], IDL.Opt(User))
    getUser(id: Principal): [User] | [] {
        const user = this.users.get(id.toText());
        return user !== undefined ? [user] : [];
    }
}
```

Canister Class

Your canister's functionality must be encapsulated in a class exported using the default export:

```
import { IDL, query } from 'azle';

export default class {
    @query([], IDL.Text)
    hello(): string {
        return 'world!';
    }
}
```

Required Structure

- Must use `export default class`
- Methods must use decorators to be exposed
- TypeScript types are optional but recommended

Multiple Canister Classes

For complex canisters, you can organize your functionality across multiple classes and export them as an array. This pattern is useful for:

- Organizing related methods into logical groups
- Better code separation and maintainability
- Modular canister design

Example Implementation

Create separate files for each class:

```
// user_service.ts
import { IDL, query, update } from 'azle';

export class UserService {
    users: Map<string, string> = new Map();

    @update([IDL.Text, IDL.Text], IDL.Bool)
    createUser(id: string, name: string): boolean {
        if (this.users.has(id)) {
            return false;
        }
        this.users.set(id, name);
        return true;
    }

    @query([IDL.Text], IDL.Opt(IDL.Text))
    getUser(id: string): string | undefined {
        return this.users.get(id);
    }
}
```

```
// notification_service.ts
import { IDL, query, update } from 'azle';

export class NotificationService {
    notifications: string[] = [];

    @update([IDL.Text], IDL.Nat)
    addNotification(message: string): number {
        this.notifications.push(message);
        return this.notifications.length;
    }

    @query([], IDL.Vec(IDL.Text))
    getNotifications(): string[] {
        return this.notifications;
    }
}
```

Then combine them in your main index file:

```
// index.ts
import { UserService } from './user_service';
import { NotificationService } from './notification_service';

export default [UserService, NotificationService];
```

Key Points for Multiple Classes:

1. **Array Export**: Use `export default [Class1, Class2,`

`...]` instead of a single class

2. **Separate Files**: Each class can be defined in its own file for better organization
3. **Method Merging**: All decorated methods from all classes become part of the canister's interface
4. **Independent State**: Each class maintains its own state within the same canister
5. **No Instantiation Needed**: Classes are automatically instantiated by Azle

All methods from all exported classes will be available in the final canister's Candid interface.

State Management

Class properties become canister state:

```
import { IDL, query, update } from 'azle';

export default class {
    // This becomes persistent canister state
    counter: number = 0;
    users: Map<string, string> = new Map();

    @query([], IDL.Nat)
    getCounter(): number {
        return this.counter;
    }

    @update([], IDL.Nat)
    increment(): number {
        this.counter += 1;
        return this.counter;
    }
}
```

Available Decorators

You must use these decorators to expose your canister's methods:

- `@query` - Read-only methods
- `@update` - Read-write methods
- `@init` - Initialization method
- `@postUpgrade` - Post-upgrade method
- `@preUpgrade` - Pre-upgrade method
- `@inspectMessage` - Message inspection method
- `@heartbeat` - Periodic execution method

- `@onLowWasmMemory` - Low memory handler method

System Method Decorators

@onLowWasmMemory

Marks a method to handle low Wasm memory conditions. This system method allows canisters to respond gracefully when running low on memory:

```
import { onLowWasmMemory, IDL } from 'azle';

export default class {
    @onLowWasmMemory
    handleLowMemory(): void {
        // Clean up unnecessary data
        this.cache.clear();

        // Log the event
        console.info('Low memory condition detected, cleaned up cache');

        // Perform garbage collection or other memory management
        this.performMemoryCleanup();
    }

    private performMemoryCleanup(): void {
        // Custom cleanup logic
    }
}
```

Key characteristics:

- Only one `@onLowWasmMemory` method allowed per canister
- Called automatically when the canister is running low on Wasm memory
- State: read-write access
- Replication: yes (replicated across all nodes)
- Async: supports async operations
- Instruction limit: 40,000,000,000 instructions

Method Visibility

Only decorated methods are exposed in the canister's Candid interface:

```typescript
import { IDL, query } from 'azle';

export default class {
    // This method is exposed
    @query([], IDL.Text)
    publicMethod(): string {
        return this.privateHelper();
    }

    // This method is private (not exposed)
    privateHelper(): string {
        return 'Hello from private method';
    }
}
```

@dfinity/candid IDL

For each of your canister's methods, deserialization of incoming arguments and serialization of return values is handled with a combination of decorators and the IDL object from the @dfinity/candid library.

`IDL` is re-exported by Azle, and has properties that correspond to Candid's supported types.

Basic Types

```
import { IDL } from 'azle';

// Text
IDL.Text;

// Numbers
IDL.Nat; // Unlimited precision unsigned integer
IDL.Nat64; // 64-bit unsigned integer
IDL.Nat32; // 32-bit unsigned integer
IDL.Nat16; // 16-bit unsigned integer
IDL.Nat8; // 8-bit unsigned integer

IDL.Int; // Unlimited precision signed integer
IDL.Int64; // 64-bit signed integer
IDL.Int32; // 32-bit signed integer
IDL.Int16; // 16-bit signed integer
IDL.Int8; // 8-bit signed integer

// Floating point
IDL.Float64; // 64-bit floating point
IDL.Float32; // 32-bit floating point

// Boolean and null
IDL.Bool;
IDL.Null;
```

Complex Types

```
import { IDL } from 'azle';

// Vector (array)
IDL.Vec(IDL.Text); // Array of text
IDL.Vec(IDL.Nat8); // Blob (array of bytes)

// Optional
IDL.Opt(IDL.Text); // Optional text

// Record (object)
IDL.Record({
    name: IDL.Text,
    age: IDL.Nat8,
    active: IDL.Bool
});

// Variant (union type)
IDL.Variant({
    Success: IDL.Text,
    Error: IDL.Text,
    Loading: IDL.Null
});
```

Advanced Types

```
import { IDL } from 'azle';

// Function reference
IDL.Func([IDL.Text], [IDL.Bool], ['query']);

// Service reference
IDL.Service({
    getName: IDL.Func([], [IDL.Text], ['query']),
    setName: IDL.Func([IDL.Text], [], ['update'])
});

// Principal
IDL.Principal;

// Reserved and Empty
IDL.Reserved;
IDL.Empty;
```

Usage Example

```
import { IDL, query, update } from 'azle';

const User = IDL.Record({
    name: IDL.Text,
    email: IDL.Text,
    age: IDL.Nat8
});
type User = {
    name: string;
    email: string;
    age: number;
};

export default class {
    @query([IDL.Text], IDL.Opt(User))
    getUser(id: string): [User] | [] {
        // Implementation here
        return [];
    }

    @update([User], IDL.Bool)
    createUser(user: User): boolean {
        // Implementation here
        return true;
    }
}
```

Decorators

Decorators expose your class methods as canister entry points. Each decorator specifies how the method should be called and what permissions it has.

Available Decorators

@query

Read-only methods that cannot modify state. Fast execution with optional cross-canister calls.

@update

Read-write methods that can modify canister state. Full async support with cross-canister calls.

@init

Initialization method called once during canister deployment. Sets up initial state.

@postUpgrade

Called after canister upgrade. Used for state migration and restoration.

@preUpgrade

Called before canister upgrade. Used for cleanup and state validation.

@inspectMessage

Called before every `@update` method. Provides access control and validation.

@heartbeat

Called periodically (~every second). Not recommended - use timers instead.

Quick Comparison

Decorator	State Access	Async	Replication	Instruction Limit
`@query`	read-only	yes*	possible	5B
`@update`	read-write	yes	yes	40B
`@init`	read-write	no	yes	300B
`@postUpgrade`	read-write	no	yes	300B
`@preUpgrade`	read-only	no	yes	300B
`@inspectMessage`	read-only	no	none	200M
`@heartbeat`	read-write	yes	yes	40B

*Only with `composite: true` option

Common Options

All decorators support these common options:

- `manual` : Manual argument/return handling
- `hidden` : Hide from Candid interface (except `@preUpgrade` , `@inspectMessage` , `@heartbeat`)

Basic Example

```
import { IDL, query, update, init } from 'azle';

export default class {
    counter: number = 0;
    owner: string = '';

    @init([IDL.Text])
    initialize(ownerName: string): void {
        this.owner = ownerName;
    }

    @query([], IDL.Nat)
    getCounter(): number {
        return this.counter;
    }

    @update([], IDL.Nat)
    increment(): number {
        this.counter += 1;
        return this.counter;
    }
}
```

@query

Read-only canister method. Cannot modify state.

- **State**: read-only
- **Replication**: possible
- **Async**: yes with `composite` set to true
- **Instruction limit**: 5_000_000_000

```
import { IDL, query } from 'azle';

export default class {
    counter: number = 0;

    @query([], IDL.Nat)
    getCounter(): number {
        return this.counter;
    }

    @query([IDL.Text], IDL.Text)
    echo(message: string): string {
        return `Echo: ${message}`;
    }
}
```

Composite Queries

```
import { IDL, query, call } from 'azle';

export default class {
    @query([], IDL.Text, { composite: true })
    async crossCanisterQuery(): Promise<string> {
        const result = await call('canister-id', 'method_name', {
            returnIdlType: IDL.Text
        });
        return result;
    }
}
```

@update

Read-write canister method. Can modify state.

- **State**: read-write
- **Replication**: yes
- **Async**: yes
- **Instruction limit**: 40_000_000_000

```
import { IDL, update } from 'azle';

export default class {
    counter: number = 0;

    @update([], IDL.Nat)
    increment(): number {
        this.counter += 1;
        return this.counter;
    }

    @update([IDL.Nat], IDL.Nat)
    setCounter(value: number): number {
        this.counter = value;
        return this.counter;
    }
}
```

@init

Canister initialization method. Called once during deployment.

- **State**: read-write
- **Replication**: yes
- **Async**: no
- **Instruction limit**: 300_000_000_000

```
import { IDL, init } from 'azle';

export default class {
    owner: string = '';
    initialized: boolean = false;

    @init([IDL.Text])
    initialize(ownerName: string): void {
        this.owner = ownerName;
        this.initialized = true;
    }
}
```

@postUpgrade

Called after canister upgrade. Used to restore state.

- **State**: read-write
- **Replication**: yes
- **Async**: no
- **Instruction limit**: 300_000_000_000 (shared with preUpgrade)

```
import { IDL, postUpgrade } from 'azle';

export default class {
    version: string = '1.0.0';

    @postUpgrade([IDL.Text])
    upgrade(newVersion: string): void {
        this.version = newVersion;
        console.log(`Upgraded to version ${newVersion}`);
    }
}
```

@preUpgrade

Called before canister upgrade. Used to save state.

- **State**: read-only
- **Replication**: yes

- **Async**: no
- **Instruction limit**: 300_000_000_000 (shared with postUpgrade)

```
import { IDL, preUpgrade } from 'azle';

export default class {
    counter: number = 0;

    @preUpgrade()
    saveState(): void {
        // Save critical state before upgrade
        console.log(`Current counter: ${this.counter}`);
    }
}
```

@inspectMessage

Called before every `@update` method. Can reject calls.

- **State**: read-only
- **Replication**: none
- **Async**: no
- **Instruction limit**: 200_000_000

```
import { IDL, inspectMessage, msgCaller } from 'azle';

export default class {
    owner: string = 'owner-principal-id';

    @inspectMessage()
    inspect(methodName: string): boolean {
        const caller = msgCaller();

        // Only allow owner to call sensitive methods
        if (methodName === 'sensitiveMethod') {
            return caller.toText() === this.owner;
        }

        return true; // Allow all other methods
    }

    @update([], IDL.Text)
    sensitiveMethod(): string {
        return 'Secret data';
    }
}
```

@heartbeat

Called periodically (~every second). Not recommended for most use cases.

- **State**: read-write
- **Replication**: yes
- **Async**: yes
- **Instruction limit**: 40_000_000_000

> **Note**: Use `setTimer` and `setTimerInterval` instead of `@heartbeat` for most periodic tasks.

```
import { IDL, heartbeat } from 'azle';

export default class {
    heartbeatCount: number = 0;

    @heartbeat()
    periodicTask(): void {
        this.heartbeatCount += 1;
        console.log(`Heartbeat ${this.heartbeatCount}`);
    }
}
```

Manual Mode

All decorators support manual mode for advanced use cases:

```
import { IDL, query, msgArgData, msgReply, IDL as CandidIDL } from 'azle';

export default class {
    @query([], IDL.Text, { manual: true })
    manualQuery(): void {
        const args = msgArgData();
        const decodedArgs = CandidIDL.decode([IDL.Text], args);

        const result = `Processed: ${decodedArgs[0]}`;
        const encodedResult = CandidIDL.encode([IDL.Text], [result]);

        msgReply(encodedResult);
    }
}
```

@query

Read-only canister method. Cannot modify state.

- **State**: read-only
- **Replication**: possible
- **Async**: yes with `composite` set to true
- **Instruction limit**: 5_000_000_000

Basic Usage

```
import { IDL, query } from 'azle';

export default class {
    counter: number = 0;

    @query([], IDL.Nat)
    getCounter(): number {
        return this.counter;
    }

    @query([IDL.Text], IDL.Text)
    echo(message: string): string {
        return `Echo: ${message}`;
    }
}
```

Composite Queries

Enable cross-canister calls within query methods:

```
import { IDL, query, call } from 'azle';

export default class {
    @query([], IDL.Text, { composite: true })
    async crossCanisterQuery(): Promise<string> {
        const result = await call('canister-id', 'method_name', {
            returnIdlType: IDL.Text
        });
        return result;
    }
}
```

Options

- `composite`: Enable async cross-canister calls
- `manual`: Manual argument/return handling
- `hidden`: Hide from Candid interface

```
import { IDL, query, msgArgData, msgReply } from 'azle';

export default class {
    @query([], IDL.Text, { manual: true })
    manualQuery(): void {
        const args = msgArgData();
        // Process manually
        msgReply(new Uint8Array([1, 2, 3]));
    }

    @query([], IDL.Text, { hidden: true })
    hiddenQuery(): string {
        return 'This method is hidden from Candid interface';
    }
}
```

@update

Read-write canister method. Can modify state.

- **State**: read-write
- **Replication**: yes
- **Async**: yes
- **Instruction limit**: 40_000_000_000

Basic Usage

```
import { IDL, update } from 'azle';

export default class {
    counter: number = 0;

    @update([], IDL.Nat)
    increment(): number {
        this.counter += 1;
        return this.counter;
    }

    @update([IDL.Nat], IDL.Nat)
    setCounter(value: number): number {
        this.counter = value;
        return this.counter;
    }
}
```

Async Operations

```
import { IDL, update, call } from 'azle';

export default class {
    @update([IDL.Text], IDL.Text)
    async processData(data: string): Promise<string> {
        // Async processing
        await new Promise((resolve) => setTimeout(resolve, 1000));

        return `Processed: ${data}`;
    }

    @update([IDL.Principal, IDL.Text], IDL.Text)
    async callOtherCanister(
        canisterId: Principal,
        message: string
    ): Promise<string> {
        const result = await call(canisterId, 'process', {
            args: [message],
            paramIdlTypes: [IDL.Text],
            returnIdlType: IDL.Text
        });
        return result;
    }
}
```

Options

- `manual` : Manual argument/return handling
- `hidden` : Hide from Candid interface

```
import { IDL, update, msgArgData, msgReply, msgReject } from 'azle';

export default class {
    @update([], IDL.Text, { manual: true })
    manualUpdate(): void {
        try {
            const args = msgArgData();
            // Process manually
            const result = 'Success!';
            msgReply(new TextEncoder().encode(result));
        } catch (error) {
            msgReject(`Error: ${error}`);
        }
    }

    @update([IDL.Text], IDL.Text, { hidden: true })
    hiddenUpdate(secret: string): string {
        return `Hidden processing of: ${secret}`;
    }
}
```

@init

Canister initialization method. Called once during deployment.

- **State**: read-write
- **Replication**: yes
- **Async**: no
- **Instruction limit**: 300_000_000_000

Only one `@init` method is allowed per canister.

Basic Usage

```
import { IDL, init } from 'azle';

export default class {
    owner: string = '';
    initialized: boolean = false;

    @init([IDL.Text])
    initialize(ownerName: string): void {
        this.owner = ownerName;
        this.initialized = true;
        console.log(`Canister initialized with owner: ${ownerName}`);
    }
}
```

Complex Initialization

```
import { IDL, init, msgCaller } from 'azle';

type Config = {
    name: string;
    maxUsers: number;
    features: string[];
};

const ConfigRecord = IDL.Record({
    name: IDL.Text,
    maxUsers: IDL.Nat32,
    features: IDL.Vec(IDL.Text)
});

export default class {
    config: Config = { name: '', maxUsers: 0, features: [] };
    owner: Principal | null = null;
    users: Map<string, string> = new Map();

    @init([ConfigRecord])
    initialize(config: Config): void {
        this.config = config;
        this.owner = msgCaller();

        console.log(`Initialized canister "${config.name}"`);
        console.log(`Max users: ${config.maxUsers}`);
        console.log(`Features: ${config.features.join(', ')}`);
    }
}
```

No Arguments

```
import { IDL, init } from 'azle';

export default class {
    startTime: bigint = 0n;

    @init()
    initialize(): void {
        this.startTime = time();
        console.log('Canister initialized at:', this.startTime);
    }
}
```

Options

- `manual` : Manual argument handling

```
import { IDL, init, msgArgData, candidDecode } from 'azle';

export default class {
    @init([], { manual: true })
    initialize(): void {
        const args = msgArgData();
        const decodedArgs = candidDecode([IDL.Text], args);

        console.log('Manual init with args:', decodedArgs);
    }
}
```

@postUpgrade

Called after canister upgrade. Used to restore state.

- **State**: read-write
- **Replication**: yes
- **Async**: no
- **Instruction limit**: 300_000_000_000 (shared with preUpgrade)

Only one `@postUpgrade` method is allowed per canister.

Basic Usage

```
import { IDL, postUpgrade } from 'azle';

export default class {
    version: string = '1.0.0';

    @postUpgrade([IDL.Text])
    upgrade(newVersion: string): void {
        this.version = newVersion;
        console.log(`Upgraded to version ${newVersion}`);
    }
}
```

State Migration

```
import { IDL, postUpgrade } from 'azle';

type UserV1 = {
    name: string;
    age: number;
};

type UserV2 = {
    name: string;
    age: number;
    email: string; // New field
    active: boolean; // New field
};

export default class {
    users: Map<string, UserV2> = new Map();
    version: string = '2.0.0';

    @postUpgrade([IDL.Text])
    migrateToV2(previousVersion: string): void {
        console.log(`Migrating from ${previousVersion} to ${this.version}`);

        // Migrate existing users to new format
        for (const [id, user] of this.users.entries()) {
            const userV1 = user as any;
            if (!userV1.email) {
                const migratedUser: UserV2 = {
                    ...userV1,
                    email: `${userV1.name.toLowerCase()}@example.com`,
                    active: true
                };
                this.users.set(id, migratedUser);
            }
        }

        console.log(`Migration complete. ${this.users.size} users
migrated.`);
    }
}
```

No Arguments

```
import { IDL, postUpgrade, time } from 'azle';

export default class {
    lastUpgrade: bigint = 0n;
    upgradeCount: number = 0;

    @postUpgrade()
    handleUpgrade(): void {
        this.lastUpgrade = time();
        this.upgradeCount += 1;

        console.log(
            `Upgrade #${this.upgradeCount} completed at ${this.lastUpgrade}`
        );
    }
}
```

Options

- `manual` : Manual argument handling

```
import { IDL, postUpgrade, msgArgData, candidDecode } from 'azle';

export default class {
    @postUpgrade([], { manual: true })
    manualUpgrade(): void {
        const args = msgArgData();
        const decodedArgs = candidDecode([IDL.Text, IDL.Nat], args);

        console.log('Manual upgrade with args:', decodedArgs);
    }
}
```

@preUpgrade

Called before canister upgrade. Used to save state.

- **State**: read-only
- **Replication**: yes
- **Async**: no
- **Instruction limit**: 300_000_000_000 (shared with postUpgrade)

Only one `@preUpgrade` method is allowed per canister.

Basic Usage

```
import { IDL, preUpgrade } from 'azle';

export default class {
    counter: number = 0;

    @preUpgrade()
    saveState(): void {
        // Save critical state before upgrade
        console.log(`Current counter: ${this.counter}`);

        // State is automatically preserved
        // This is mainly for logging/cleanup
    }
}
```

State Validation

```
import { IDL, preUpgrade } from 'azle';

export default class {
    users: Map<string, any> = new Map();
    orders: Map<string, any> = new Map();

    @preUpgrade()
    validateState(): void {
        console.log(`Pre-upgrade validation:`);
        console.log(`- Users: ${this.users.size}`);
        console.log(`- Orders: ${this.orders.size}`);

        // Validate critical state
        if (this.users.size === 0) {
            console.warn('Warning: No users in system');
        }

        // Log important metrics
        const activeUsers = Array.from(this.users.values()).filter(
            (user) => user.active
        ).length;
        console.log(`- Active users: ${activeUsers}`);
    }
}
```

Cleanup Operations

```
import { IDL, preUpgrade, clearTimer } from 'azle';

export default class {
    activeTimers: Set<bigint> = new Set();

    @preUpgrade()
    cleanup(): void {
        console.log('Cleaning up before upgrade...');

        // Cancel all active timers
        for (const timerId of this.activeTimers) {
            clearTimer(timerId);
        }

        console.log(`Cleared ${this.activeTimers.size} timers`);

        // Other cleanup operations
        console.log('Cleanup complete');
    }
}
```

Backup State

```
import { IDL, preUpgrade } from 'azle';

export default class {
    criticalData: Map<string, string> = new Map();

    @preUpgrade()
    backupCriticalData(): void {
        const backup = {
            timestamp: Date.now(),
            dataCount: this.criticalData.size,
            keys: Array.from(this.criticalData.keys())
        };

        console.log('Backup info:', JSON.stringify(backup));

        // In a real scenario, you might want to store
        // backup data in stable storage
    }
}
```

No Manual Mode

Note: `@preUpgrade` does not support manual mode as it takes no arguments.

@inspectMessage

Called before every `@update` method. Can reject calls.

- **State**: read-only
- **Replication**: none
- **Async**: no
- **Instruction limit**: 200_000_000

Only one `@inspectMessage` method is allowed per canister.

Basic Usage

```
import { IDL, inspectMessage, msgCaller, update } from 'azle';

export default class {
    owner: string = 'owner-principal-id';

    @inspectMessage()
    inspect(methodName: string): boolean {
        const caller = msgCaller();

        // Only allow owner to call sensitive methods
        if (methodName === 'sensitiveMethod') {
            return caller.toText() === this.owner;
        }

        return true; // Allow all other methods
    }

    @update([], IDL.Text)
    sensitiveMethod(): string {
        return 'Secret data';
    }

    @update([IDL.Text], IDL.Text)
    publicMethod(message: string): string {
        return `Public: ${message}`;
    }
}
```

Role-Based Access Control

```
import { IDL, inspectMessage, msgCaller, update } from 'azle';

export default class {
    admins: Set<string> = new Set(['admin-principal-1', 'admin-
principal-2']);
    moderators: Set<string> = new Set(['mod-principal-1']);

    @inspectMessage()
    checkPermissions(methodName: string): boolean {
        const caller = msgCaller().toText();

        // Admin-only methods
        if (['deleteUser', 'systemReset'].includes(methodName)) {
            return this.admins.has(caller);
        }

        // Moderator or admin methods
        if (['banUser', 'deletePost'].includes(methodName)) {
            return this.admins.has(caller) || this.moderators.has(caller);
        }

        // Public methods - all users allowed
        return true;
    }

    @update([IDL.Text], IDL.Bool)
    deleteUser(userId: string): boolean {
        // Admin only - checked in inspectMessage
        return true;
    }

    @update([IDL.Text], IDL.Bool)
    banUser(userId: string): boolean {
        // Admin or moderator - checked in inspectMessage
        return true;
    }

    @update([IDL.Text], IDL.Text)
    createPost(content: string): string {
        // Public method - all users allowed
        return `Post created: ${content}`;
    }
}
```

Rate Limiting

```
import { IDL, inspectMessage, msgCaller, update, time } from 'azle';

export default class {
    lastCallTime: Map<string, bigint> = new Map();
    rateLimitSeconds: bigint = 60n * 1_000_000_000n; // 60 seconds in
nanoseconds

    @inspectMessage()
    rateLimit(methodName: string): boolean {
        const caller = msgCaller().toText();
        const now = time();

        // Only rate limit certain methods
        if (['expensiveOperation', 'sendEmail'].includes(methodName)) {
            const lastCall = this.lastCallTime.get(caller);

            if (lastCall && now - lastCall < this.rateLimitSeconds) {
                console.log(`Rate limit exceeded for ${caller}`);
                return false; // Reject the call
            }

            this.lastCallTime.set(caller, now);
        }

        return true;
    }

    @update([IDL.Text], IDL.Text)
    expensiveOperation(data: string): string {
        // Rate limited operation
        return `Processed: ${data}`;
    }

    @update([IDL.Text], IDL.Bool)
    sendEmail(recipient: string): boolean {
        // Rate limited operation
        return true;
    }
}
```

Method Arguments Access

```
import { IDL, inspectMessage, update } from 'azle';

export default class {
    @inspectMessage()
    validateArguments(methodName: string, ...args: unknown[]): boolean {
        console.log(`Method: ${methodName}, Args:`, args);

        // Validate specific method arguments
        if (methodName === 'transfer') {
            const [amount] = args as [number];
            if (amount <= 0 || amount > 1000000) {
                console.log('Invalid transfer amount');
                return false;
            }
        }

        if (methodName === 'setUsername') {
            const [username] = args as [string];
            if (username.length < 3 || username.length > 20) {
                console.log('Invalid username length');
                return false;
            }
        }

        return true;
    }

    @update([IDL.Nat], IDL.Bool)
    transfer(amount: number): boolean {
        return true;
    }

    @update([IDL.Text], IDL.Bool)
    setUsername(username: string): boolean {
        return true;
    }
}
```

Options

- `manual` : Manual argument handling

```
import { IDL, inspectMessage, msgArgData, candidDecode } from 'azle';

export default class {
    @inspectMessage([], { manual: true })
    manualInspect(): boolean {
        const args = msgArgData();
        const decoded = candidDecode([IDL.Text], args);

        console.log('Manual inspect with args:', decoded);
        return true;
    }
}
```

@heartbeat

Called periodically (~every second). Not recommended for most use cases.

- **State**: read-write
- **Replication**: yes
- **Async**: yes
- **Instruction limit**: 40_000_000_000

Only one `@heartbeat` method is allowed per canister.

> **Note**: Use `setTimer` and `setTimerInterval` instead of `@heartbeat` for most periodic tasks.

Basic Usage

```
import { IDL, heartbeat } from 'azle';

export default class {
    heartbeatCount: number = 0;

    @heartbeat()
    periodicTask(): void {
        this.heartbeatCount += 1;
        console.log(`Heartbeat ${this.heartbeatCount}`);
    }
}
```

Periodic Cleanup

```
import { IDL, heartbeat, time } from 'azle';

export default class {
    sessions: Map<string, { userId: string; lastActive: bigint }> = new
Map();
    lastCleanup: bigint = 0n;

    @heartbeat()
    cleanup(): void {
        const now = time();
        const oneHour = 60n * 60n * 1_000_000_000n; // 1 hour in nanoseconds

        // Only run cleanup every hour
        if (now - this.lastCleanup < oneHour) {
            return;
        }

        // Clean up expired sessions
        const expiredSessions: string[] = [];
        for (const [sessionId, session] of this.sessions.entries()) {
            if (now - session.lastActive > oneHour * 24n) {
                // 24 hours
                expiredSessions.push(sessionId);
            }
        }

        for (const sessionId of expiredSessions) {
            this.sessions.delete(sessionId);
        }

        console.log(`Cleaned up ${expiredSessions.length} expired sessions`);
        this.lastCleanup = now;
    }
}
```

Async Operations

```
import { IDL, heartbeat, call } from 'azle';

export default class {
    lastHealthCheck: bigint = 0n;
    isHealthy: boolean = true;

    @heartbeat()
    async healthCheck(): Promise<void> {
        const now = time();
        const fiveMinutes = 5n * 60n * 1_000_000_000n;

        // Only check every 5 minutes
        if (now - this.lastHealthCheck < fiveMinutes) {
            return;
        }

        try {
            // Check external service
            const response = await call('external-service-canister', 'ping',
{
                returnIdlType: IDL.Bool
            });

            this.isHealthy = response;
            console.log(`Health check: ${this.isHealthy ? 'OK' : 'FAILED'}`);
        } catch (error) {
            this.isHealthy = false;
            console.log(`Health check failed: ${error}`);
        }

        this.lastCleanup = now;
    }
}
```

Why Use Timers Instead

Timers are more flexible and efficient:

```
import { IDL, init, setTimerInterval, clearTimer } from 'azle';

export default class {
    cleanupTimerId: bigint | null = null;

    @init()
    initialize(): void {
        // Set up periodic cleanup with timer instead of heartbeat
        this.cleanupTimerId = setTimerInterval(3600, () => {
            // Every hour
            this.performCleanup();
        });
    }

    performCleanup(): void {
        console.log('Performing scheduled cleanup...');
        // Cleanup logic here
    }

    stopCleanup(): void {
        if (this.cleanupTimerId) {
            clearTimer(this.cleanupTimerId);
            this.cleanupTimerId = null;
        }
    }
}
```

Limitations

- Cannot guarantee exact timing
- Runs on all replicas (waste of resources)
- May not execute during high load
- Cannot pass arguments
- Limited to ~1 second intervals

When to Use Heartbeat

Only use `@heartbeat` when you need:

- Guaranteed periodic execution across all replicas
- System-level maintenance tasks
- Monitoring that must run even when canister is idle

For most use cases, prefer `setTimer` and `setTimerInterval`.

@onLowWasmMemory

The `@onLowWasmMemory` decorator marks a method as the low Wasm memory handler for your canister. This system method is automatically called when your canister is running low on Wasm memory, allowing you to implement graceful memory management.

Usage

```
import { onLowWasmMemory } from 'azle';

export default class {
    cache: Map<string, any> = new Map();

    @onLowWasmMemory
    handleLowMemory(): void {
        // Clean up unnecessary data
        this.cache.clear();

        // Log the event
        console.info('Low memory condition detected, cleaned up cache');

        // Perform additional cleanup
        this.performMemoryCleanup();
    }

    private performMemoryCleanup(): void {
        // Custom cleanup logic
        // Remove old entries, compact data structures, etc.
    }
}
```

Characteristics

- **State Access**: Read-write
- **Replication**: Yes (replicated across all nodes)
- **Async Support**: Yes, can be async
- **Instruction Limit**: 40,000,000,000 instructions
- **Frequency**: Called automatically when memory is low
- **Limit**: Only one `@onLowWasmMemory` method per canister

Common Use Cases

Cache Management

```
import { onLowWasmMemory, query, update, IDL } from 'azle';

export default class {
    cache: Map<string, { data: any; timestamp: bigint }> = new Map();
    userData: Map<string, any> = new Map();

    @onLowWasmMemory
    handleLowMemory(): void {
        // Clear expired cache entries
        const currentTime = Date.now();
        const oneHourAgo = BigInt(currentTime - 3600000);

        for (const [key, value] of this.cache.entries()) {
            if (value.timestamp < oneHourAgo) {
                this.cache.delete(key);
            }
        }

        console.info(`Cleaned up cache, ${this.cache.size} entries
remaining`);
    }

    @update([IDL.Text, IDL.Text], IDL.Bool)
    cacheData(key: string, data: string): boolean {
        this.cache.set(key, {
            data,
            timestamp: BigInt(Date.now())
        });
        return true;
    }

    @query([IDL.Text], IDL.Opt(IDL.Text))
    getCachedData(key: string): string | undefined {
        return this.cache.get(key)?.data;
    }
}
```

Data Structure Optimization

```
import { onLowWasmMemory, StableBTreeMap } from 'azle';

export default class {
    tempData: any[] = [];
    stableStorage = new StableBTreeMap<string, string>(0);

    @onLowWasmMemory
    async handleLowMemory(): Promise<void> {
        // Move temporary data to stable storage
        for (let i = 0; i < this.tempData.length; i++) {
            const item = this.tempData[i];
            if (item.shouldPersist) {
                this.stableStorage.insert(item.id, JSON.stringify(item));
            }
        }

        // Clear temporary arrays
        this.tempData = [];

        console.info('Moved temporary data to stable storage');
    }
}
```

Memory Monitoring

```
import { onLowWasmMemory, canisterCycleBalance, time } from 'azle';

export default class {
    memoryEvents: { timestamp: bigint; action: string }[] = [];

    @onLowWasmMemory
    handleLowMemory(): void {
        const timestamp = time();
        const cycleBalance = canisterCycleBalance();

        // Log the memory event
        this.memoryEvents.push({
            timestamp,
            action: `Low memory detected. Cycle balance: ${cycleBalance}`
        });

        // Keep only recent events (last 100)
        if (this.memoryEvents.length > 100) {
            this.memoryEvents = this.memoryEvents.slice(-100);
        }

        // Perform cleanup based on available cycles
        if (cycleBalance < 1_000_000_000n) {
            // Aggressive cleanup if cycles are also low
            this.performAggressiveCleanup();
        } else {
            // Standard cleanup
            this.performStandardCleanup();
        }
    }

    private performAggressiveCleanup(): void {
        // More aggressive memory management
    }

    private performStandardCleanup(): void {
        // Standard memory management
    }
}
```

Best Practices

1. **Keep It Simple**: The low memory handler should be efficient and avoid complex operations
2. **Prioritize Cleanup**: Focus on freeing memory rather than performing business logic
3. **Log Events**: Track when low memory events occur for monitoring
4. **Consider Cycles**: Check cycle balance as low memory often correlates with resource constraints
5. **Test Thoroughly**: Simulate low memory conditions to ensure your handler works correctly

Important Notes

- This decorator is automatically triggered by the IC when memory is low
- Only one method per canister can have this decorator
- The method should complete quickly to avoid blocking the canister
- Consider the instruction limit when implementing complex cleanup logic
- This is a system-level method that doesn't appear in your Candid interface

IC API

The IC API is exposed as functions exported from `azle`. These functions provide access to Internet Computer platform capabilities.

Quick Example

```
import {
    msgCaller,
    time,
    canisterCycleBalance,
    call,
    IDL,
    query,
    update
} from 'azle';

export default class {
    @query([], IDL.Text)
    whoCalledWhen(): string {
        const caller = msgCaller().toText();
        const now = time();
        return `Called by ${caller} at ${now}`;
    }

    @query([], IDL.Nat)
    getBalance(): bigint {
        return canisterCycleBalance();
    }

    @update([IDL.Principal, IDL.Text], IDL.Text)
    async callOther(canisterId: Principal, message: string): Promise<string>
{
        const result = await call(canisterId, 'echo', {
            args: [message],
            paramIdlTypes: [IDL.Text],
            returnIdlType: IDL.Text
        });
        return result;
    }
}
```

IC API Functions

Message Information

- msgCaller - Get the caller's principal identity
- msgMethodName - Get the name of the currently executing method
- msgArgData - Get raw Candid-encoded arguments

Time

- time - Get the current Internet Computer time

Timers

- setTimer - Execute a callback after a delay
- setTimerInterval - Execute a callback repeatedly
- clearTimer - Cancel a scheduled timer

Cycles

- canisterCycleBalance - Get the canister's cycle balance
- msgCyclesAccept - Accept cycles sent with a call
- msgCyclesAvailable - Get cycles available in current call
- msgCyclesRefunded - Get cycles refunded from last call
- cyclesBurn - Permanently destroy cycles

Inter-Canister Calls

- call - Make calls to other canisters

Canister Information

- canisterSelf - Get the current canister's principal
- canisterVersion - Get the canister version number
- isController - Check if a principal is a controller

Error Handling

- trap - Terminate execution with an error
- msgRejectCode - Get rejection code from failed calls
- msgRejectMsg - Get rejection message from failed calls

Manual Response

- msgReply - Reply with raw Candid-encoded data
- msgReject - Reject with an error message

Random

- randSeed - Seed the random number generator

Advanced

- performanceCounter - Get performance metrics
- dataCertificate - Get data certificate for queries
- candidEncode - Encode values to Candid format
- candidDecode - Decode Candid format to values
- inReplicatedExecution - Check execution context
- chunk - Process data in chunks

API Reference Table

Function	Category	Use Case
`call`	Calls	Inter-canister communication
`candidDecode`	Advanced	Manual deserialization
`candidEncode`	Advanced	Manual serialization
`canisterCycleBalance`	Cycles	Resource monitoring
`canisterSelf`	Info	Self-reference
`canisterVersion`	Info	Version tracking
`chunk`	Advanced	Memory management
`clearTimer`	Timers	Cancel scheduled operations
`cyclesBurn`	Cycles	Deflationary mechanics
`dataCertificate`	Advanced	Query verification
`inReplicatedExecution`	Advanced	Environment detection
`isController`	Info	Admin access control
`msgArgData`	Message	Manual argument processing
`msgCaller`	Message	Authentication, access control
`msgCyclesAccept`	Cycles	Payment processing

Function	Category	Use Case
`msgCyclesAvailable`	Cycles	Payment validation
`msgCyclesRefunded`	Cycles	Cost tracking
`msgMethodName`	Message	Logging, analytics
`msgReject`	Manual	Custom error responses
`msgRejectCode`	Errors	Error classification
`msgRejectMsg`	Errors	Detailed error info
`msgReply`	Manual	Custom response handling
`performanceCounter`	Advanced	Performance monitoring
`randSeed`	Random	Secure randomness
`setTimer`	Timers	Delayed operations
`setTimerInterval`	Timers	Recurring tasks
`time`	Time	Timestamps, expiration
`trap`	Errors	Input validation

For detailed examples and usage patterns, click on any function name above to view its dedicated documentation page.

Message Information

msgCaller

Get the principal of the identity that initiated the current call:

```
import { msgCaller, IDL, query } from 'azle';

export default class {
    @query([], IDL.Text)
    whoAmI(): string {
        return msgCaller().toText();
    }

    @query([], IDL.Bool)
    isAnonymous(): boolean {
        return msgCaller().toText() === '2vxsx-fae';
    }
}
```

msgMethodName

Get the name of the currently executing method:

```
import { msgMethodName, IDL, update } from 'azle';

export default class {
    @update([], IDL.Text)
    currentMethod(): string {
        return msgMethodName(); // Returns "currentMethod"
    }
}
```

Time

time

Get the current ICP system time in nanoseconds:

```
import { time, IDL, query } from 'azle';

export default class {
    @query([], IDL.Nat64)
    getCurrentTime(): bigint {
        return time();
    }

    @query([], IDL.Text)
    getFormattedTime(): string {
        const nanos = time();
        const date = new Date(Number(nanos / 1_000_000n));
        return date.toISOString();
    }
}
```

Timers

setTimer

Execute a callback after a delay:

```
import { setTimer, IDL, update } from 'azle';

export default class {
    @update([IDL.Nat], IDL.Nat64)
    scheduleTask(delaySeconds: number): bigint {
        const timerId = setTimer(delaySeconds, () => {
            console.log('Timer executed!');
        });

        return timerId;
    }
}
```

setTimerInterval

Execute a callback repeatedly:

```
import { setTimerInterval, IDL, update } from 'azle';

export default class {
    counter: number = 0;

    @update([IDL.Nat], IDL.Nat64)
    startPeriodicTask(intervalSeconds: number): bigint {
        const timerId = setTimerInterval(intervalSeconds, () => {
            this.counter += 1;
            console.log(`Periodic task executed ${this.counter} times`);
        });

        return timerId;
    }
}
```

clearTimer

Cancel a scheduled timer:

```
import { setTimer, clearTimer, IDL, update } from 'azle';

export default class {
    activeTimers: Set<bigint> = new Set();

    @update([IDL.Nat], IDL.Nat64)
    scheduleTask(delaySeconds: number): bigint {
        const timerId = setTimer(delaySeconds, () => {
            console.log('Task executed!');
            this.activeTimers.delete(timerId);
        });

        this.activeTimers.add(timerId);
        return timerId;
    }

    @update([IDL.Nat64], IDL.Bool)
    cancelTask(timerId: bigint): boolean {
        if (this.activeTimers.has(timerId)) {
            clearTimer(timerId);
            this.activeTimers.delete(timerId);
            return true;
        }
        return false;
    }
}
```

Cycles

canisterCycleBalance

Get the canister's current cycle balance:

```
import { canisterCycleBalance, IDL, query } from 'azle';

export default class {
    @query([], IDL.Nat)
    getBalance(): bigint {
        return canisterCycleBalance();
    }

    @query([], IDL.Bool)
    hasEnoughCycles(): boolean {
        const balance = canisterCycleBalance();
        const minimumRequired = 1_000_000_000n; // 1 billion cycles
        return balance >= minimumRequired;
    }
}
```

msgCyclesAccept

Accept cycles sent with the current call:

```
import { msgCyclesAccept, msgCyclesAvailable, IDL, update } from 'azle';

export default class {
    @update([], IDL.Nat)
    acceptPayment(): bigint {
        const available = msgCyclesAvailable();
        const accepted = msgCyclesAccept(available);
        return accepted;
    }
}
```

Inter-Canister Calls

call

Make calls to other canisters:

```
import { call, IDL, update, Principal } from 'azle';

export default class {
    @update([IDL.Principal, IDL.Text], IDL.Text)
    async callOtherCanister(
        canisterId: Principal,
        message: string
    ): Promise<string> {
        const result = await call(canisterId, 'process_message', {
            args: [message],
            paramIdlTypes: [IDL.Text],
            returnIdlType: IDL.Text
        });

        return result;
    }

    @update([IDL.Principal], IDL.Nat)
    async transferCycles(recipient: Principal): Promise<bigint> {
        const cyclesToSend = 1_000_000n;

        await call(recipient, 'receive_cycles', {
            cycles: cyclesToSend
        });

        return cyclesToSend;
    }
}
```

Canister Information

canisterSelf

Get the current canister's principal:

```
import { canisterSelf, IDL, query } from 'azle';

export default class {
    @query([], IDL.Principal)
    myId(): Principal {
        return canisterSelf();
    }
}
```

canisterVersion

Get the current canister version:

```
import { canisterVersion, IDL, query } from 'azle';

export default class {
    @query([], IDL.Nat64)
    getVersion(): bigint {
        return canisterVersion();
    }
}
```

Error Handling

trap

Terminate execution with an error:

```
import { trap, IDL, update } from 'azle';

export default class {
    @update([IDL.Text], IDL.Text)
    processInput(input: string): string {
        if (input === '') {
            trap('Input cannot be empty');
        }

        return `Processed: ${input}`;
    }
}
```

Manual Response Handling

msgReply / msgReject

For manual response handling in decorators with `manual: true`:

```
import {
    msgReply,
    msgReject,
    msgArgData,
    IDL,
    update,
    candidDecode,
    candidEncode
} from 'azle';

export default class {
    @update([], IDL.Text, { manual: true })
    manualResponse(): void {
        try {
            const result = 'Success!';
            const encoded = candidEncode([IDL.Text], [result]);
            msgReply(encoded);
        } catch (error) {
            msgReject(`Error: ${error}`);
        }
    }
}
```

Random Numbers

randSeed

Get a random seed for pseudorandom number generation:

```
import { randSeed, IDL, query } from 'azle';

export default class {
    @query([], IDL.Nat)
    getRandomNumber(): number {
        const seed = randSeed();
        // Use seed for pseudorandom number generation
        return Math.abs(seed.reduce((a, b) => a + b, 0));
    }
}
```

Data Structures

StableBTreeMap

A persistent B-tree map backed by stable memory that automatically persists across canister upgrades:

```
import { StableBTreeMap, IDL, query, update } from 'azle';

export default class {
    // Create a stable map with memory ID 0
    userProfiles = new StableBTreeMap<string, { name: string; age: number }
>(0);

    // Create multiple maps with different memory IDs
    counters = new StableBTreeMap<string, bigint>(1);
    settings = new StableBTreeMap<string, boolean>(2);

    @update(
        [IDL.Text, IDL.Record({ name: IDL.Text, age: IDL.Nat })],
        IDL.Opt(IDL.Record({ name: IDL.Text, age: IDL.Nat }))
    )
    setUserProfile(
        userId: string,
        profile: { name: string; age: number }
    ): { name: string; age: number } | undefined {
        return this.userProfiles.insert(userId, profile);
    }

    @query([IDL.Text], IDL.Opt(IDL.Record({ name: IDL.Text, age: IDL.Nat })))
    getUserProfile(userId: string): { name: string; age: number } | undefined
{
        return this.userProfiles.get(userId);
    }

    @query(
        [],
        IDL.Vec(
            IDL.Tuple(IDL.Text, IDL.Record({ name: IDL.Text, age: IDL.Nat }))
        )
    )
    getAllProfiles(): [string, { name: string; age: number }][] {
        return this.userProfiles.items();
    }

    @update([IDL.Text], IDL.Opt(IDL.Record({ name: IDL.Text, age: IDL.Nat
}})))
    removeUser(userId: string): { name: string; age: number } | undefined {
        return this.userProfiles.remove(userId);
    }

    @query([], IDL.Nat32)
    getUserCount(): number {
        return this.userProfiles.len();
    }

    @query([IDL.Text], IDL.Bool)
    userExists(userId: string): boolean {
        return this.userProfiles.containsKey(userId);
    }
}
```

StableBTreeMap Constructor

```
new StableBTreeMap<Key, Value>(
    memoryId: number,
    keySerializable?: Serializable,
    valueSerializable?: Serializable
)
```

- `memoryId` - Unique identifier (0-253) for this map's stable memory
- `keySerializable` - Optional custom serialization for keys (defaults to ICP-enabled JSON)
- `valueSerializable` - Optional custom serialization for values (defaults to ICP-enabled JSON)

StableBTreeMap Methods

- `containsKey(key)` - Check if a key exists
- `get(key)` - Retrieve value by key
- `insert(key, value)` - Insert/update key-value pair
- `remove(key)` - Remove key and return its value
- `isEmpty()` - Check if map is empty
- `len()` - Get number of key-value pairs
- `keys(startIndex?, length?)` - Get keys in sorted order
- `values(startIndex?, length?)` - Get values in sorted order
- `items(startIndex?, length?)` - Get key-value pairs in sorted order

Custom Serialization

```
import { StableBTreeMap, Serializable } from 'azle';

// Custom serializer for numbers as little-endian bytes
const numberSerializer: Serializable = {
    toBytes: (num: number) => {
        const buffer = new ArrayBuffer(8);
        const view = new DataView(buffer);
        view.setFloat64(0, num, true); // little-endian
        return new Uint8Array(buffer);
    },
    fromBytes: (bytes: Uint8Array) => {
        const view = new DataView(bytes.buffer);
        return view.getFloat64(0, true); // little-endian
    }
};

export default class {
    // Map with custom number serialization for keys
    numericMap = new StableBTreeMap<number, string>(0, numberSerializer);
}
```

JSON Utilities

jsonStringify / jsonParse

ICP-enabled JSON utilities that handle special types like `Principal`, `BigInt`, and `Uint8Array`:

```
import { jsonStringify, jsonParse, Principal, IDL, query, update } from
'azle';

export default class {
    @update([IDL.Text], IDL.Text)
    processComplexData(input: string): string {
        // Parse ICP-enabled JSON
        const data = jsonParse(input);

        // Work with the data
        if (data.principal) {
            data.lastAccessed = time();
            data.accessCount = (data.accessCount || 0n) + 1n;
        }

        // Convert back to ICP-enabled JSON
        return jsonStringify(data);
    }

    @query([], IDL.Text)
    getExampleData(): string {
        const complexData = {
            principal: Principal.fromText('rdmx6-jaaaa-aaaah-qcaiq-cai'),
            balance: 123_456_789n, // BigInt
            buffer: new Uint8Array([1, 2, 3, 4]),
            metadata: {
                created: time(),
                isActive: true,
                tags: ['user', 'verified']
            }
        };

        return jsonStringify(complexData);
    }

    @update([IDL.Text], IDL.Principal)
    extractPrincipal(jsonData: string): Principal {
        const parsed = jsonParse(jsonData);
        return parsed.principal; // Automatically converted back to Principal
    }
}
```

Supported Special Types

The ICP-enabled JSON utilities automatically handle:

- `Principal` - Converted to/from text representation
- `BigInt` - Converted to/from string with special markers
- `Uint8Array` - Converted to/from array representation
- `undefined` - Properly preserved (standard JSON loses undefined values)

Custom JSON Processing

```
import { jsonStringify, jsonParse } from 'azle';

// Custom replacer function
function customReplacer(key: string, value: any): any {
    if (value instanceof Date) {
        return { __date__: value.toISOString() };
    }
    return value;
}

// Custom reviver function
function customReviver(key: string, value: any): any {
    if (value?.__date__) {
        return new Date(value.__date__);
    }
    return value;
}

export default class {
    @update([IDL.Text], IDL.Text)
    processWithCustomJSON(input: string): string {
        const data = jsonParse(input, customReviver);
        data.processedAt = new Date();
        return jsonStringify(data, customReplacer);
    }
}
```

call

Make calls to other canisters with full type safety.

```
import { call, IDL, update, Principal } from 'azle';

export default class {
    @update([IDL.Principal, IDL.Text], IDL.Text)
    async callOtherCanister(
        canisterId: Principal,
        message: string
    ): Promise<string> {
        const result = await call(canisterId, 'process_message', {
            args: [message],
            paramIdlTypes: [IDL.Text],
            returnIdlType: IDL.Text
        });

        return result;
    }

    @update([IDL.Principal], IDL.Nat)
    async transferCycles(recipient: Principal): Promise<bigint> {
        const cyclesToSend = 1_000_000n;

        await call(recipient, 'receive_cycles', {
            cycles: cyclesToSend
        });

        return cyclesToSend;
    }
}
```

The `call` function makes inter-canister calls with full type safety, automatic serialization/deserialization, and comprehensive error handling.

Parameters:

- `canisterId` : Target canister principal (`Principal`)
- `methodName` : Name of the method to call (`string`)
- `options` : Call configuration object

Options Object:

- `args?` : Array of arguments to pass
- `paramIdlTypes?` : IDL types for parameters
- `returnIdlType?` : IDL type for return value
- `cycles?` : Cycles to send with the call (`bigint`)

Returns: Promise resolving to the method's return value

Important Notes:

- Automatically handles Candid serialization/deserialization
- Supports cycle transfers
- Provides comprehensive error handling
- Works with both query and update methods

candidDecode

Decode Candid binary format to TypeScript/JavaScript values.

```
import { candidDecode, msgArgData, IDL, update } from 'azle';

export default class {
    @update([], IDL.Vec(IDL.Text))
    decodeArguments(): string[] {
        const rawArgs = msgArgData();

        // Decode assuming the call had [string, number, boolean] args
        const decoded = candidDecode([IDL.Text, IDL.Nat, IDL.Bool], rawArgs);

        return decoded.map((arg) => String(arg));
    }

    @update(
        [IDL.Vec(IDL.Nat8)],
        IDL.Record({
            name: IDL.Text,
            age: IDL.Nat,
            active: IDL.Bool
        })
    )
    decodeUserData(encodedData: Uint8Array): {
        name: string;
        age: number;
        active: boolean;
    } {
        const decoded = candidDecode(
            [
                IDL.Record({
                    name: IDL.Text,
                    age: IDL.Nat,
                    active: IDL.Bool
                })
            ],
            encodedData
        );

        return decoded[0] as {
            name: string;
            age: number;
            active: boolean;
        };
    }
}
```

The `candidDecode` function decodes Candid binary format back into TypeScript/JavaScript values for processing raw data.

Parameters:

- `idlTypes`: Array of IDL types describing the expected data structure
- `data`: Candid-encoded data as `Uint8Array`

Returns: Array of decoded values

Use Cases:

- Processing raw argument data with `msgArgData`
- Decoding data from external sources
- Manual handling of inter-canister call responses
- Working with stored Candid-encoded data

Important Notes:

- IDL types must match the encoded data structure exactly
- Returns an array even for single values
- Throws an error if decoding fails due to type mismatch

candidEncode

Encode TypeScript/JavaScript values to Candid format.

```
import { candidEncode, IDL, query } from 'azle';

export default class {
    @query([IDL.Text, IDL.Nat], IDL.Vec(IDL.Nat8))
    encodeValues(text: string, number: number): Uint8Array {
        return candidEncode([IDL.Text, IDL.Nat], [text, number]);
    }

    @query([], IDL.Vec(IDL.Nat8))
    encodeComplexData(): Uint8Array {
        const data = {
            name: 'Alice',
            age: 30,
            active: true
        };

        return candidEncode(
            [
                IDL.Record({
                    name: IDL.Text,
                    age: IDL.Nat,
                    active: IDL.Bool
                })
            ],
            [data]
        );
    }
}
```

The `candidEncode` function encodes TypeScript/JavaScript values into Candid binary format for low-level data manipulation or inter-canister communication.

Parameters:

- `idlTypes` : Array of IDL types describing the data structure
- `values` : Array of values to encode

Returns: Candid-encoded data as `Uint8Array`

Use Cases:

- Manual response handling with `msgReply`
- Custom serialization for storage
- Inter-canister communication with raw data
- Building protocol-level integrations

Important Notes:

- Values must match the provided IDL types exactly
- Resulting bytes can be decoded with `candidDecode`
- Used internally by Azle for automatic serialization

canisterCycleBalance

Get the canister's current cycle balance.

```
import { canisterCycleBalance, IDL, query } from 'azle';

export default class {
    @query([], IDL.Nat)
    getBalance(): bigint {
        return canisterCycleBalance();
    }

    @query([], IDL.Bool)
    hasEnoughCycles(): boolean {
        const balance = canisterCycleBalance();
        const minimumRequired = 1_000_000_000n; // 1 billion cycles
        return balance >= minimumRequired;
    }
}
```

The `canisterCycleBalance` function returns the current number of cycles in the canister's balance. This is essential for monitoring canister resources and making decisions about operations that consume cycles.

Returns: Current cycle balance as `bigint`

Use Cases:

- Monitor canister resource usage
- Implement cycle-based access control
- Track cycle consumption patterns
- Trigger low-balance alerts or actions

Important Notes:

- Cycle balance decreases with computation and storage usage
- Balance can increase through cycle transfers from other canisters
- Monitor balance regularly to prevent canister freezing

canisterSelf

Get the current canister's principal ID.

```
import { canisterSelf, IDL, query } from 'azle';

export default class {
    @query([], IDL.Principal)
    myId(): Principal {
        return canisterSelf();
    }

    @query([], IDL.Text)
    myIdText(): string {
        return canisterSelf().toText();
    }
}
```

The `canisterSelf` function returns the principal ID of the current canister. This is useful for self-reference in inter-canister calls and logging.

Returns: The current canister's principal (`Principal`)

Use Cases:

- Self-referencing in inter-canister calls
- Logging and debugging
- Building canister registries
- Identity verification

Important Notes:

- Always returns the same value for a given canister
- Available in all method types (@query, @update, etc.)
- Useful for building self-aware canister systems

canisterVersion

Get the current canister version number.

```
import { canisterVersion, IDL, query } from 'azle';

export default class {
    @query([], IDL.Nat64)
    getVersion(): bigint {
        return canisterVersion();
    }

    @query([], IDL.Text)
    getVersionInfo(): string {
        const version = canisterVersion();
        return `Canister version: ${version}`;
    }
}
```

The `canisterVersion` function returns the current version number of the canister. The version increments each time the canister is upgraded.

Returns: Current canister version (`bigint`)

Use Cases:

- Track canister upgrades
- Version-dependent logic
- Debugging and monitoring
- Migration management

Important Notes:

- Starts at 0 for newly installed canisters
- Increments by 1 with each upgrade
- Persists across canister upgrades

chunk

Process data in chunks for memory efficiency.

```
import { chunk, IDL, update } from 'azle';

export default class {
    @update([IDL.Vec(IDL.Nat8), IDL.Nat], IDL.Vec(IDL.Vec(IDL.Nat8)))
    processInChunks(data: Uint8Array, chunkSize: number): Uint8Array[] {
        return chunk(data, chunkSize);
    }

    @update([IDL.Vec(IDL.Text), IDL.Nat], IDL.Vec(IDL.Text))
    processTextChunks(texts: string[], chunkSize: number): string[] {
        const chunks = chunk(texts, chunkSize);

        return chunks.map((chunk) => chunk.join(' | '));
    }
}
```

The `chunk` function splits arrays or data into smaller chunks of a specified size, useful for memory management and batch processing.

Parameters:

- `data` : Array or Uint8Array to chunk
- `size` : Maximum size of each chunk (`number`)

Returns: Array of chunks

Use Cases:

- Process large datasets in smaller batches
- Memory-efficient data handling
- Pagination implementation
- Streaming data processing

Important Notes:

- Last chunk may be smaller than the specified size
- Works with both arrays and Uint8Array
- Useful for avoiding memory limits with large data
- Essential for processing large files or datasets

clearTimer

Cancel a scheduled timer.

```
import { setTimer, clearTimer, IDL, update } from 'azle';

export default class {
    activeTimers: Set<bigint> = new Set();

    @update([IDL.Nat], IDL.Nat64)
    scheduleTask(delaySeconds: number): bigint {
        const timerId = setTimer(delaySeconds, () => {
            console.log('Task executed!');
            this.activeTimers.delete(timerId);
        });

        this.activeTimers.add(timerId);
        return timerId;
    }

    @update([IDL.Nat64], IDL.Bool)
    cancelTask(timerId: bigint): boolean {
        if (this.activeTimers.has(timerId)) {
            clearTimer(timerId);
            this.activeTimers.delete(timerId);
            return true;
        }
        return false;
    }
}
```

The `clearTimer` function cancels a previously scheduled timer (created with either `setTimer` or `setTimerInterval`).

Parameters:

- `timerId`: The timer ID returned by `setTimer` or `setTimerInterval`

Returns: `void`

Important Notes:

- Safe to call with non-existent timer IDs (no error thrown)
- Works for both one-time timers (`setTimer`) and recurring timers (`setTimerInterval`)
- Once cleared, the timer ID cannot be reused

cyclesBurn

Permanently destroy cycles from the canister's balance.

```
import { cyclesBurn, canisterCycleBalance, IDL, update } from 'azle';

export default class {
    @update(
        [IDL.Nat],
        IDL.Record({
            burned: IDL.Nat,
            remaining: IDL.Nat
        })
    )
    burnCycles(amount: bigint): { burned: bigint; remaining: bigint } {
        const balanceBefore = canisterCycleBalance();

        if (amount > balanceBefore) {
            throw new Error('Cannot burn more cycles than available');
        }

        cyclesBurn(amount);

        return {
            burned: amount,
            remaining: canisterCycleBalance()
        };
    }
}
```

The `cyclesBurn` function permanently destroys the specified number of cycles from the canister's balance. The burned cycles are removed from circulation.

Parameters:

- `amount`: Number of cycles to burn (`bigint`)

Returns: `void`

Use Cases:

- Implement deflationary tokenomics
- Reduce cycle supply for economic reasons
- Clean up excess cycles
- Fee burning mechanisms

Important Notes:

- Cycles are permanently destroyed and cannot be recovered
- Cannot burn more cycles than the canister's current balance

- Use with caution as this reduces available resources

dataCertificate

Get the data certificate for certified queries.

```
import { dataCertificate, certifiedDataSet, IDL, query, update } from 'azle';

export default class {
    private certifiedValue: string = '';

    @update([IDL.Text])
    setCertifiedData(value: string): void {
        this.certifiedValue = value;

        // Set the certified data (up to 32 bytes)
        const encoder = new TextEncoder();
        const data = encoder.encode(value.slice(0, 32));
        certifiedDataSet(data);
    }

    @query(
        [],
        IDL.Record({
            data: IDL.Text,
            certificate: IDL.Opt(IDL.Vec(IDL.Nat8))
        })
    )
    getCertifiedData(): {
        data: string;
        certificate: [Uint8Array] | [];
    } {
        const certificate = dataCertificate();

        return {
            data: this.certifiedValue,
            certificate: certificate ? [certificate] : []
        };
    }
}
```

The `dataCertificate` function returns the data certificate that can be used to verify the authenticity of certified data in query calls.

Returns: Certificate as `Uint8Array` or `undefined` if not available

Use Cases:

- Verify authenticity of query responses
- Build trusted data verification systems
- Implement certified query responses
- Enable cryptographic proof of data integrity

Important Notes:

- Only available in query calls, not update calls
- Requires prior use of `certifiedDataSet`
- Certificate proves data authenticity to external verifiers
- Returns `undefined` if no certificate is available

inReplicatedExecution

Check if the code is running in replicated execution mode.

```
import { inReplicatedExecution, IDL, query, update } from 'azle';

export default class {
    @query([], IDL.Bool)
    isReplicated(): boolean {
        return inReplicatedExecution();
    }

    @update([IDL.Text], IDL.Text)
    conditionalOperation(data: string): string {
        if (inReplicatedExecution()) {
            // This code runs on all replicas
            console.log('Processing in replicated mode');
            return `Replicated: ${data}`;
        } else {
            // This code might run in local testing
            console.log('Processing in non-replicated mode');
            return `Local: ${data}`;
        }
    }
}
```

The `inReplicatedExecution` function determines whether the current execution is happening in the Internet Computer's replicated environment or in a local/testing context.

Returns: `true` if running in replicated execution, `false` otherwise

Use Cases:

- Conditional logic for production vs. testing
- Debug logging that only runs locally
- Feature flags based on execution context
- Performance optimizations for different environments

Important Notes:

- Returns `true` when running on the Internet Computer
- Returns `false` during local development/testing
- Useful for environment-specific behavior
- Helps distinguish between live and test environments

isController

Check if a given principal is a controller of the canister.

```
import { isController, msgCaller, IDL, query, update } from 'azle';

export default class {
    @query([IDL.Principal], IDL.Bool)
    checkController(principal: Principal): boolean {
        return isController(principal);
    }

    @update([IDL.Text], IDL.Text)
    adminOnlyFunction(data: string): string {
        const caller = msgCaller();

        if (!isController(caller)) {
            throw new Error('Access denied: caller is not a controller');
        }

        return `Admin processed: ${data}`;
    }

    @query([], IDL.Bool)
    amIController(): boolean {
        return isController(msgCaller());
    }
}
```

The `isController` function checks whether a given principal is a controller of the current canister. Controllers have administrative privileges and can upgrade the canister.

Parameters:

- `principal`: The principal to check (`Principal`)

Returns: `true` if the principal is a controller, `false` otherwise

Use Cases:

- Implement controller-only functions
- Access control for administrative operations
- Security checks before sensitive operations
- Role-based permissions

Important Notes:

- Controllers are set during canister creation or by other controllers
- Controllers can upgrade the canister code
- Use for high-privilege operations only

msgArgData

Get the raw Candid-encoded arguments of the current call.

```
import { msgArgData, IDL, update, candidDecode } from 'azle';

export default class {
    @update([], IDL.Text)
    inspectArgs(): string {
        const rawArgs = msgArgData();

        // Decode assuming the call was made with [string, number] args
        const decoded = candidDecode([IDL.Text, IDL.Nat], rawArgs);

        return `Raw args length: ${rawArgs.length}, Decoded:
${JSON.stringify(decoded)}`;
    }
}
```

The `msgArgData` function returns the raw bytes of the arguments passed to the current method call. This is typically used in advanced scenarios where you need to handle method arguments manually.

Use Cases:

- Custom argument validation
- Method argument introspection
- Debugging and logging raw call data
- Building generic proxy or forwarding mechanisms

Important Notes:

- Returns raw Candid-encoded bytes
- Useful for methods with `{ manual: true }` option
- Must be decoded using `candidDecode` to get typed values

msgCaller

Get the caller's principal identity.

```
import { msgCaller, IDL, query } from 'azle';

export default class {
    @query([], IDL.Principal)
    whoAmI(): Principal {
        return msgCaller();
    }
}
```

Access Control

```
import { msgCaller, IDL, update, Principal } from 'azle';

export default class {
    private owner: Principal = Principal.fromText(
        'rdmx6-jaaaa-aaaah-qcaiq-cai'
    );

    @update([IDL.Text], IDL.Text)
    adminFunction(data: string): string {
        const caller = msgCaller();

        if (!caller.compareTo(this.owner)) {
            throw new Error('Access denied: only owner can call this
function');
        }

        return `Admin processed: ${data}`;
    }

    @query([], IDL.Bool)
    isOwner(): boolean {
        return msgCaller().compareTo(this.owner);
    }
}
```

The `msgCaller` function returns the principal of the identity that invoked the current method. This is essential for authentication and access control in your canister.

Important Notes:

- Returns `Principal.anonymous()` for anonymous calls
- Available in all canister method types (@query, @update, etc.)
- Cannot be called from @heartbeat methods (returns anonymous principal)

msgCyclesAccept

Accept cycles sent with the current call.

```
import { msgCyclesAccept, msgCyclesAvailable, IDL, update } from 'azle';

export default class {
    @update([], IDL.Nat)
    acceptPayment(): bigint {
        const available = msgCyclesAvailable();
        const accepted = msgCyclesAccept(available);
        return accepted;
    }

    @update([], IDL.Nat)
    acceptPartialPayment(): bigint {
        const available = msgCyclesAvailable();
        const toAccept = available / 2n; // Accept half
        return msgCyclesAccept(toAccept);
    }
}
```

The `msgCyclesAccept` function accepts cycles that were sent along with the current method call. Any cycles not accepted are automatically refunded to the caller.

Parameters:

- `maxAmount`: Maximum number of cycles to accept (`bigint`)

Returns: Actual number of cycles accepted (`bigint`)

Important Notes:

- Must be called from an `@update` method (not `@query`)
- Cycles not accepted are refunded to the caller
- Cannot accept more cycles than were sent
- Accepted cycles are added to the canister's balance

msgCyclesAvailable

Get the number of cycles available in the current call.

```
import { msgCyclesAvailable, msgCyclesAccept, IDL, update } from 'azle';

export default class {
    @update(
        [],
        IDL.Record({
            available: IDL.Nat,
            accepted: IDL.Nat
        })
    )
    processPayment(): { available: bigint; accepted: bigint } {
        const available = msgCyclesAvailable();

        if (available < 1_000_000n) {
            throw new Error('Insufficient payment');
        }

        const accepted = msgCyclesAccept(available);

        return { available, accepted };
    }
}
```

The `msgCyclesAvailable` function returns the number of cycles that were sent along with the current method call and are available to be accepted.

Returns: Number of cycles available to accept (`bigint`)

Use Cases:

- Check payment amount before processing
- Implement minimum payment requirements
- Calculate partial acceptance amounts
- Log payment information

Important Notes:

- Available cycles decrease as `msgCyclesAccept` is called
- Unaccepted cycles are automatically refunded
- Returns 0 if no cycles were sent with the call

msgCyclesRefunded

Get the number of cycles refunded from the last inter-canister call.

```
import { call, msgCyclesRefunded, IDL, update, Principal } from 'azle';

export default class {
    @update(
        [IDL.Principal],
        IDL.Record({
            sent: IDL.Nat,
            refunded: IDL.Nat
        })
    )
    async transferCycles(
        recipient: Principal
    ): Promise<{ sent: bigint; refunded: bigint }> {
        const cyclesToSend = 1_000_000n;

        await call(recipient, 'receive_cycles', {
            cycles: cyclesToSend
        });

        const refunded = msgCyclesRefunded();

        return {
            sent: cyclesToSend,
            refunded
        };
    }
}
```

The `msgCyclesRefunded` function returns the number of cycles that were refunded from the most recent inter-canister call made by the current method.

Returns: Number of cycles refunded from the last call (`bigint`)

Use Cases:

- Track actual cycle costs of inter-canister calls
- Implement cycle accounting and monitoring
- Adjust future calls based on refund patterns
- Debug cycle transfer issues

Important Notes:

- Only reflects refunds from the most recent call
- Returns 0 if no cycles were refunded
- Must be called after an inter-canister call completes

msgMethodName

Get the name of the currently executing method.

```
import { msgMethodName, IDL, update } from 'azle';

export default class {
    private methodCallCount: Map<string, number> = new Map();

    @update([IDL.Text], IDL.Text)
    process(data: string): string {
        const method = msgMethodName();
        const count = (this.methodCallCount.get(method) || 0) + 1;
        this.methodCallCount.set(method, count);

        return `Method ${method} called ${count} times with data: ${data}`;
    }
}
```

The `msgMethodName` function returns the name of the method that was called to invoke the current execution. This is useful for logging, metrics, and debugging.

Use Cases:

- Method call tracking and analytics
- Dynamic routing based on method name
- Logging and debugging
- Method-specific processing logic

msgReject

Reject the current call with an error message.

```
import { msgReject, msgArgData, candidDecode, IDL, update } from 'azle';

export default class {
    @update([IDL.Text], IDL.Empty, { manual: true })
    validateAndProcess(input: string): void {
        if (input.length === 0) {
            msgReject('Input cannot be empty');
            return;
        }

        if (input.length > 100) {
            msgReject('Input too long: maximum 100 characters allowed');
            return;
        }

        // Process normally if validation passes
        const result = `Processed: ${input}`;
        const encoded = candidEncode([IDL.Text], [result]);
        msgReply(encoded);
    }
}
```

The `msgReject` function manually rejects the current method call with an error message. This is used in methods marked with `{ manual: true }`.

Parameters:

- `message`: Error message to include in the rejection (`string`)

Returns: `void`

Use Cases:

- Custom error handling in manual methods
- Input validation with specific error messages
- Conditional rejection based on complex logic
- Advanced error response formatting

Important Notes:

- Only use in methods with `{ manual: true }`
- Call this exactly once per method execution
- Cannot be combined with `msgReply`
- Equivalent to throwing an error in regular methods

msgRejectCode

Get the rejection code from a failed inter-canister call.

```
import { call, msgRejectCode, IDL, update, Principal } from 'azle';

export default class {
    @update(
        [IDL.Principal],
        IDL.Record({
            success: IDL.Bool,
            rejectCode: IDL.Opt(IDL.Nat8),
            message: IDL.Text
        })
    )
    async tryCall(canisterId: Principal): Promise<{
        success: boolean;
        rejectCode: [number] | [];
        message: string;
    }> {
        try {
            await call(canisterId, 'some_method', {});
            return {
                success: true,
                rejectCode: [],
                message: 'Call succeeded'
            };
        } catch (error) {
            const rejectCode = msgRejectCode();
            return {
                success: false,
                rejectCode: [rejectCode],
                message: this.getRejectMessage(rejectCode)
            };
        }
    }

    private getRejectMessage(rejectCode: number): string {
        switch (rejectCode) {
            case 1: // SysFatal
                return 'Error: System fatal error';
            case 2: // SysTransient
                return 'Error: System transient error';
            case 3: // DestinationInvalid
                return 'Error: Invalid destination canister';
            case 4: // CanisterReject
                return 'Error: Canister rejected the call';
            case 5: // CanisterError
                return 'Error: Canister error occurred';
            default:
                return `Error: Unknown rejection code ${rejectCode}`;
        }
    }
}
```

The `msgRejectCode` function returns the rejection code from the most recent failed inter-canister call. Use this in catch blocks to understand why a call failed.

Returns: Rejection code as `number`

Rejection Codes:

- `1` : SysFatal - Fatal system error
- `2` : SysTransient - Transient system error (may retry)
- `3` : DestinationInvalid - Invalid destination canister
- `4` : CanisterReject - Target canister rejected the call
- `5` : CanisterError - Error occurred in target canister

Use Cases:

- Error handling and recovery logic
- Retry mechanisms based on error type
- Logging and debugging failed calls
- User-friendly error messages

msgRejectMsg

Get the rejection message from a failed inter-canister call.

```
import {
    call,
    msgRejectCode,
    msgRejectMsg,
    IDL,
    update,
    Principal
} from 'azle';

export default class {
    @update(
        [IDL.Principal],
        IDL.Record({
            result: IDL.Opt(IDL.Text),
            error: IDL.Opt(
                IDL.Record({
                    code: IDL.Nat8,
                    message: IDL.Text
                })
            )
        })
    )
    async safeCall(canisterId: Principal): Promise<{
        result: [string] | [];
        error: [{ code: number; message: string }] | [];
    }> {
        try {
            const result = await call<any, string>(canisterId, 'get_data', {
                returnIdlType: IDL.Text
            });

            return {
                result: [result],
                error: []
            };
        } catch (error) {
            const code = msgRejectCode();
            const message = msgRejectMsg();

            return {
                result: [],
                error: [
                    {
                        code,
                        message: `Call failed (${code}): ${message}`
                    }
                ]
            };
        }
    }
}
```

The `msgRejectMsg` function returns the detailed rejection message from the most recent failed inter-canister call. This provides specific information about what went wrong.

Returns: Rejection message as `string`

Use Cases:

- Detailed error logging and debugging
- User-friendly error reporting
- Error analysis and monitoring
- Building robust error handling systems

Important Notes:

- Contains detailed error information from the target canister
- Combined with `msgRejectCode` for complete error context
- Only available in catch blocks after failed inter-canister calls
- Message content varies based on the type of failure

msgReply

Reply to the current call with raw Candid-encoded data.

```
import { msgReply, candidEncode, IDL, update } from 'azle';

export default class {
    @update([IDL.Text], IDL.Empty, { manual: true })
    manualEcho(input: string): void {
        const encoded = candidEncode([IDL.Text], [input]);
        msgReply(encoded);
    }

    @update([IDL.Nat], IDL.Empty, { manual: true })
    doubleNumber(n: number): void {
        const result = n * 2;
        const encoded = candidEncode([IDL.Nat], [result]);
        msgReply(encoded);
    }
}
```

The `msgReply` function manually sends a reply to the current method call using raw Candid-encoded data. This is used in methods marked with `{ manual: true }`.

Parameters:

- `data`: Raw Candid-encoded bytes to send as the reply (`Uint8Array`)

Returns: `void`

Use Cases:

- Custom response processing
- Streaming responses
- Advanced error handling
- Performance optimization for large responses

Important Notes:

- Only use in methods with `{ manual: true }`
- Data must be properly Candid-encoded
- Call this exactly once per method execution
- Cannot be combined with normal return statements

performanceCounter

Get performance metrics for the current execution.

```
import { performanceCounter, IDL, query } from 'azle';

export default class {
    @query([], IDL.Nat64)
    getInstructionCount(): bigint {
        return performanceCounter(0); // Instruction counter
    }

    @query(
        [],
        IDL.Record({
            instructions: IDL.Nat64,
            timestamp: IDL.Nat64
        })
    )
    getPerformanceMetrics(): {
        instructions: bigint;
        timestamp: bigint;
    } {
        return {
            instructions: performanceCounter(0),
            timestamp: performanceCounter(1) // Time in nanoseconds
        };
    }
}
```

The `performanceCounter` function provides access to various performance metrics of the current execution context.

Parameters:

- `counterType`: The type of counter to read (`number`)
  - `0`: Instruction counter
  - `1`: Current time in nanoseconds

Returns: Counter value (`bigint`)

Use Cases:

- Performance monitoring and optimization
- Execution cost analysis
- Benchmarking different implementations
- Resource usage tracking

Important Notes:

- Counter values are specific to the current call context
- Instruction counter includes all instructions executed so far
- Time counter provides high-precision timestamps

randSeed

Seed the pseudorandom number generator with cryptographically secure randomness.

```
import { randSeed, IDL, update } from 'azle';

export default class {
    @update([], IDL.Vec(IDL.Nat8))
    generateRandomBytes(): Uint8Array {
        // Seed with secure randomness from the IC
        randSeed();

        // Generate random bytes using standard Math.random()
        const bytes = new Uint8Array(32);
        for (let i = 0; i < bytes.length; i++) {
            bytes[i] = Math.floor(Math.random() * 256);
        }

        return bytes;
    }

    @update([], IDL.Nat)
    rollDice(): number {
        randSeed();
        return Math.floor(Math.random() * 6) + 1;
    }

    @update([IDL.Vec(IDL.Text)], IDL.Text)
    selectRandom(items: string[]): string {
        if (items.length === 0) {
            throw new Error('Cannot select from empty array');
        }

        randSeed();
        const index = Math.floor(Math.random() * items.length);
        return items[index];
    }
}
```

The `randSeed` function seeds JavaScript's `Math.random()` with cryptographically secure randomness from the Internet Computer. This ensures that random number generation is truly unpredictable.

Returns: `void`

Use Cases:

- Secure random number generation
- Lottery and gaming systems
- Random selection algorithms
- Cryptographic nonce generation

Important Notes:

- Provides cryptographically secure randomness
- Must be called before using `Math.random()` for security
- Randomness is consensus-based across all replicas
- Call once per method that needs randomness

setTimer

Execute a callback after a delay.

```
import { setTimer, IDL, update } from 'azle';

export default class {
    @update([IDL.Nat], IDL.Nat64)
    scheduleTask(delaySeconds: number): bigint {
        const timerId = setTimer(delaySeconds, () => {
            console.log('Timer executed!');
        });

        return timerId;
    }
}
```

Delayed Operations

```
import { setTimer, msgCaller, IDL, update } from 'azle';

export default class {
    private notifications: Map<string, string> = new Map();

    @update([IDL.Nat, IDL.Text], IDL.Text)
    scheduleNotification(delaySeconds: number, message: string): string {
        const caller = msgCaller().toText();

        setTimer(delaySeconds, () => {
            this.notifications.set(caller, message);
            console.log(`Notification for ${caller}: ${message}`);
        });

        return `Notification scheduled for ${delaySeconds} seconds from now`;
    }

    @query([], IDL.Opt(IDL.Text))
    getNotification(): [string] | [] {
        const caller = msgCaller().toText();
        const notification = this.notifications.get(caller);

        if (notification) {
            this.notifications.delete(caller);
            return [notification];
        }

        return [];
    }
}
```

The `setTimer` function schedules a callback to be executed after a specified delay. The timer executes exactly once and returns a timer ID that can be used with `clearTimer`.

Parameters:

- `delay`: Duration in seconds (as `number`)
- `callback`: Function to execute when timer fires

Returns: Timer ID (`bigint`) for use with `clearTimer`

Important Notes:

- Timers persist across canister upgrades
- Timer callbacks have access to canister state
- Failed timer callbacks are logged but don't crash the canister

setTimerInterval

Execute a callback repeatedly at specified intervals.

```
import { setTimerInterval, IDL, update } from 'azle';

export default class {
    counter: number = 0;

    @update([IDL.Nat], IDL.Nat64)
    startPeriodicTask(intervalSeconds: number): bigint {
        const timerId = setTimerInterval(intervalSeconds, () => {
            this.counter += 1;
            console.log(`Periodic task executed ${this.counter} times`);
        });

        return timerId;
    }
}
```

Health Monitoring

```
import { setTimerInterval, canisterCycleBalance, IDL, update } from 'azle';

export default class {
    private healthStatus: string = 'unknown';
    private lastCheckTime: bigint = 0n;

    @update([IDL.Nat], IDL.Nat64)
    startHealthMonitoring(intervalSeconds: number): bigint {
        return setTimerInterval(intervalSeconds, () => {
            const cycleBalance = canisterCycleBalance();
            const now = time();

            this.lastCheckTime = now;

            if (cycleBalance < 1_000_000_000n) {
                // Less than 1B cycles
                this.healthStatus = 'low_cycles';
                console.warn(`Low cycle balance: ${cycleBalance}`);
            } else {
                this.healthStatus = 'healthy';
                console.log(`Health check passed at ${now}`);
            }
        });
    }

    @query(
        [],
        IDL.Record({
            status: IDL.Text,
            lastCheck: IDL.Nat64,
            cycleBalance: IDL.Nat
        })
    )
    getHealthStatus(): {
        status: string;
        lastCheck: bigint;
        cycleBalance: bigint;
    } {
        return {
            status: this.healthStatus,
            lastCheck: this.lastCheckTime,
            cycleBalance: canisterCycleBalance()
        };
    }
}
```

The `setTimerInterval` function schedules a callback to execute repeatedly at specified intervals. Unlike `setTimer`, this continues executing until cancelled with `clearTimer`.

Parameters:

- `interval`: Duration between executions in seconds (as `number`)
- `callback`: Function to execute on each interval

Returns: Timer ID (`bigint`) for use with `clearTimer`

Important Notes:

- Continues executing until explicitly cancelled
- Each execution is independent - if one fails, others continue
- Use `clearTimer` to stop the interval

Time

API for getting the current Internet Computer system time.

time

Get the current ICP system time in nanoseconds since the epoch.

```
import { time, IDL, query } from 'azle';

export default class {
    @query([], IDL.Nat64)
    getCurrentTime(): bigint {
        return time();
    }

    @query([], IDL.Text)
    getFormattedTime(): string {
        const nanos = time();
        const date = new Date(Number(nanos / 1_000_000n));
        return date.toISOString();
    }
}
```

Time Tracking

```typescript
import { time, IDL, query, update } from 'azle';

export default class {
    createdAt: bigint = 0n;
    events: { timestamp: bigint; event: string }[] = [];

    @init()
    initialize(): void {
        this.createdAt = time();
    }

    @update([IDL.Text], IDL.Nat64)
    logEvent(event: string): bigint {
        const timestamp = time();
        this.events.push({ timestamp, event });
        return timestamp;
    }

    @query([], IDL.Nat64)
    getUptime(): bigint {
        return time() - this.createdAt;
    }

    @query([], IDL.Text)
    getUptimeFormatted(): string {
        const uptimeNanos = time() - this.createdAt;
        const uptimeSeconds = Number(uptimeNanos / 1_000_000_000n);

        const days = Math.floor(uptimeSeconds / 86400);
        const hours = Math.floor((uptimeSeconds % 86400) / 3600);
        const minutes = Math.floor((uptimeSeconds % 3600) / 60);

        return `${days}d ${hours}h ${minutes}m`;
    }
}
```

Time-Based Operations

```
import { time, IDL, query, update } from 'azle';

export default class {
    sessions: Map<string, { createdAt: bigint; lastActive: bigint }> =
        new Map();

    @update([IDL.Text], IDL.Bool)
    createSession(sessionId: string): boolean {
        const now = time();
        this.sessions.set(sessionId, {
            createdAt: now,
            lastActive: now
        });
        return true;
    }

    @update([IDL.Text], IDL.Bool)
    refreshSession(sessionId: string): boolean {
        const session = this.sessions.get(sessionId);
        if (session) {
            session.lastActive = time();
            return true;
        }
        return false;
    }

    @query([IDL.Text], IDL.Bool)
    isSessionValid(sessionId: string): boolean {
        const session = this.sessions.get(sessionId);
        if (!session) return false;

        const now = time();
        const oneHour = 60n * 60n * 1_000_000_000n; // 1 hour in nanoseconds

        return now - session.lastActive < oneHour;
    }

    @query([], IDL.Vec(IDL.Text))
    getExpiredSessions(): string[] {
        const now = time();
        const oneHour = 60n * 60n * 1_000_000_000n;

        return Array.from(this.sessions.entries())
            .filter(([_, session]) => now - session.lastActive >= oneHour)
            .map(([sessionId, _]) => sessionId);
    }
}
```

trap

Terminate execution with an error message.

```
import { trap, IDL, update } from 'azle';

export default class {
    @update([IDL.Text], IDL.Text)
    processInput(input: string): string {
        if (input === '') {
            trap('Input cannot be empty');
        }

        if (input.length > 1000) {
            trap('Input too long: maximum 1000 characters allowed');
        }

        return `Processed: ${input}`;
    }
}
```

The `trap` function immediately terminates the current execution with an error message. All state changes made during the current call are rolled back.

Parameters:

- `message`: Error message to include in the trap (`string`)

Returns: Never returns (execution stops)

Use Cases:

- Input validation with immediate failure
- Critical error conditions
- Guard clauses for invalid states
- Security-related assertions

Important Notes:

- Stops execution immediately
- Rolls back all state changes from the current call
- Cannot be caught or handled within the same call
- Use judiciously as it terminates the entire call

HTTP Server (Experimental)

This section documents the HTTP Server methodology for developing Azle applications. This methodology embraces traditional web server techniques, allowing you to write HTTP servers using popular libraries such as Express, and using JSON for simple serialization and deserialization purposes.

HTTP Server functionality will remain experimental for an unknown length of time.

Get Started

- Installation
- Deployment

Azle helps you to build secure decentralized/replicated servers in TypeScript or JavaScript on ICP. The current replication factor is 13-40.

Please remember that the HTTP Server functionality is only accessible in Azle's experimental mode.

Azle runs in experimental mode through explicitly enabling a flag in `dfx.json` or certain CLI commands.

This mode is intended for developers who are willing to accept the risk of using an alpha or beta project. Its focus is on quickly enabling new features and functionality without requiring the time and other resources necessary to advance them to the stable mode. The Node.js standard libary, npm ecosystem, and HTTP server functionality are also major areas of focus.

> NOTE: Keep clearly in mind that the experimental mode fundamentally changes the Azle Wasm binary. It is not guaranteed to be secure or stable in API changes or runtime behavior. If you enable the experimental mode, even if you only use APIs from the stable mode, you are accepting a higher risk of bugs, errors, crashes, security exploits, breaking API changes, etc.

Installation

> Windows is only supported through a Linux virtual environment of some kind, such as WSL

You will need Node.js and dfx to develop ICP applications with Azle:

Node.js

It's recommended to use nvm to install the latest LTS version of Node.js:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.3/install.sh |
bash
```

Restart your terminal and then run:

```
nvm install --lts
```

Check that the installation went smoothly by looking for clean output from the following command:

```
node --version
```

dfx

Install the dfx command line tools for managing ICP applications:

```
DFX_VERSION=0.27.0 sh -ci "$(curl -fsSL https://internetcomputer.org/
install.sh)"
```

Check that the installation went smoothly by looking for clean output from the following command:

```
dfx --version
```

Deployment

To create and deploy a simple sample application called `hello_world`:

```
# create a new default project called hello_world
npx azle new hello_world --http-server --experimental
cd hello_world
```

```
# install all npm dependencies including azle
npm install
```

```
# start up a local ICP replica
dfx start --clean
```

In a separate terminal in the `hello_world` directory:

```
# deploy your canister
dfx deploy
```

If you would like your canister to autoreload on file changes:

```
AZLE_AUTORELOAD=true dfx deploy
```

View your frontend in a web browser at `http://[canisterId].raw.localhost:8000`.

To obtain your application's [canisterId]:

```
dfx canister id backend
```

Communicate with your canister using any HTTP client library, for example using `curl`:

```
curl http://[canisterId].raw.localhost:8000/db
curl -X POST -H "Content-Type: application/json" -d "{ \"hello\": \"world\"
}" http://[canisterId].raw.localhost:8000/db/update
```

Examples

There are many Azle examples in the examples directory. We recommend starting with the following:

- apollo_server
- audio_and_video
- autoreload
- ethers
- ethers_base
- express
- fetch_ic
- file_protocol
- fs
- hello_world_http_server
- http_outcall_fetch
- hybrid_canister
- ic_evm_rpc
- internet_identity
- large_files
- sqlite
- tfjs
- web_assembly

Deployment

- Starting the local replica
- Deploying to the local replica
- Interacting with your canister
- Deploying to mainnet

There are two main ICP environments that you will generally interact with: the local replica and mainnet.

We recommend using the `dfx` command line tools to deploy to these environments. Please note that not all `dfx` commands are shown here. See the dfx CLI reference for more information.

Starting the local replica

We recommend running your local replica in its own terminal and on a port of your choosing:

```
dfx start --host 127.0.0.1:8000
```

Alternatively you can start the local replica as a background process:

```
dfx start --background --host 127.0.0.1:8000
```

If you want to stop a local replica running in the background:

```
dfx stop
```

If you ever see this kind of error after `dfx stop`:

```
Error: Failed to kill all processes.  Remaining: 627221 626923 627260
```

Then try this:

```
dfx killall
```

If your replica starts behaving strangely, we recommend starting the replica clean, which will clean the `dfx` state of your project:

```
dfx start --clean --host 127.0.0.1:8000
```

Deploying to the local replica

To deploy all canisters defined in your `dfx.json`:

```
dfx deploy
```

If you would like your canister to autoreload on file changes:

```
AZLE_AUTORELOAD=true dfx deploy
```

To deploy an individual canister:

```
dfx deploy [canisterName]
```

Interacting with your canister

You will generally interact with your canister through an HTTP client such as `curl`, `fetch`, or a web browser. The URL of your canister locally will look like this: `http://[canisterId].raw.localhost:[replicaPort]`. Azle will print your canister's URL in the terminal after a successful deploy.

```
# You can obtain the canisterId like this
dfx canister id [canisterName]

# You can obtain the replicaPort like this
dfx info webserver-port

# An example of performing a GET request to a canister
curl http://a3shf-5eaaa-aaaaa-qaafa-cai.raw.localhost:8000

# An example of performing a POST request to a canister
curl -X POST -H "Content-Type: application/json" -d "{ \"hello\": \"world\"
}" http://a3shf-5eaaa-aaaaa-qaafa-cai.raw.localhost:8000
```

Deploying to mainnet

Assuming you are setup with a cycles wallet, then you are ready to deploy to mainnet.

To deploy all canisters defined in your dfx.json:

```
dfx deploy --network ic
```

To deploy an individual canister:

```
dfx deploy --network ic [canisterName]
```

The URL of your canister on mainnet will look like this: `https://[canisterId].raw.icp0.io`.

Project Structure TL;DR

Your project is just a directory with a `dfx.json` file that points to your `.ts` or `.js` entrypoint.

Here's what your directory structure might look like:

```
hello_world/
│
├── dfx.json
│
└── src/
    └── api.ts
```

For an HTTP Server canister this would be the simplest corresponding `dfx.json` file:

```
{
    "canisters": {
        "api": {
            "type": "azle",
            "main": "src/api.ts",
            "custom": {
                "experimental": true,
                "candid_gen": "http"
            }
        }
    }
}
```

For a Candid RPC canister this would be the simplest corresponding `dfx.json` file:

```
{
    "canisters": {
        "api": {
            "type": "azle",
            "main": "src/api.ts"
        }
    }
}
```

Once you have created this directory structure you can deploy to mainnet or a locally running replica by running the `dfx deploy` command in the same directory as your `dfx.json` file.

dfx.json

The `dfx.json` file is the main ICP-specific configuration file for your canisters. The following are various examples of `dfx.json` files.

Automatic Candid File Generation

The command-line tools `dfx` require a Candid file to deploy your canister. Candid RPC canisters will automatically have their Candid files generated and stored in the `.azle` directory without any extra property in the `dfx.json` file. HTTP Server canisters must specify `"candid_gen": "http"` for their Candid files to be generated automatically in the `.azle` directory:

```
{
    "canisters": {
        "api": {
            "type": "azle",
            "main": "src/api.ts",
            "custom": {
                "experimental": true,
                "candid_gen": "http"
            }
        }
    }
}
```

Custom Candid File

If you would like to provide your own custom Candid file you can specify `"candid": "[path to your candid file]"` and `"candid_gen": "custom"`:

```
{
    "canisters": {
        "api": {
            "type": "azle",
            "main": "src/api.ts",
            "candid": "src/api.did",
            "custom": {
                "experimental": true,
                "candid_gen": "custom"
            }
        }
    }
}
```

Environment Variables

You can provide environment variables to Azle canisters by specifying their names in your `dfx.json` file and then accessing them through the `process.env` object in Azle.

You must provide the environment variables that you want included in the same process as your `dfx deploy` command.

Be aware that the environment variables that you specify in your `dfx.json` file will be included in plain text in your canister's Wasm binary.

```
{
    "canisters": {
        "api": {
            "type": "azle",
            "main": "src/api.ts",
            "custom": {
                "experimental": true,
                "candid_gen": "http",
                "env": ["MY_ENVIRONMENT_VARIABLE"]
            }
        }
    }
}
```

Assets

See the Assets chapter for more information:

```
{
    "canisters": {
        "api": {
            "type": "azle",
            "main": "src/api.ts",
            "custom": {
                "experimental": true,
                "candid_gen": "http",
                "assets": [
                    ["src/frontend/dist", "dist"],
                    ["src/backend/media/audio.ogg", "media/audio.ogg"],
                    ["src/backend/media/video.ogv", "media/video.ogv"]
                ]
            }
        }
    }
}
```

Build Assets

See the Assets chapter for more information:

```
{
    "canisters": {
        "api": {
            "type": "azle",
            "main": "src/api.ts",
            "custom": {
                "experimental": true,
                "candid_gen": "http",
                "assets": [
                    ["src/frontend/dist", "dist"],
                    ["src/backend/media/audio.ogg", "media/audio.ogg"],
                    ["src/backend/media/video.ogv", "media/video.ogv"]
                ],
                "build_assets": "npm run build"
            }
        }
    }
}
```

ESM Externals

This will instruct Azle's TypeScript/JavaScript build process to ignore bundling the provided named packages.

Sometimes the build process is overly eager to include packages that won't actually be used at runtime. This can be a problem if those packages wouldn't even work at runtime due to limitations in ICP or Azle. It is thus useful to be able to exclude them:

```
{
    "canisters": {
        "api": {
            "type": "azle",
            "main": "src/api.ts",
            "custom": {
                "experimental": true,
                "candid_gen": "http",
                "esm_externals": ["@nestjs/microservices", "@nestjs/
websockets"]
            }
        }
    }
}
```

ESM Aliases

This will instruct Azle's TypeScript/JavaScript build process to alias a package name to another pacakge name.

This can be useful if you need to polyfill certain packages that might not exist in Azle:

```
{
    "canisters": {
        "api": {
            "type": "azle",
            "main": "src/api.ts",
            "custom": {
                "experimental": true,
                "candid_gen": "http",
                "esm_aliases": {
                    "crypto": "crypto-browserify"
                }
            }
        }
    }
}
```

Servers TL;DR

Just write Node.js servers like this:

```
import { createServer } from 'http';

const server = createServer((req, res) => {
    res.write('Hello World!');
    res.end();
});

server.listen();
```

or write Express servers like this:

```
import express, { Request } from 'express';

let db = {
    hello: ''
};

const app = express();

app.use(express.json());

app.get('/db', (req, res) => {
    res.json(db);
});

app.post('/db/update', (req: Request<any, any, typeof db>, res) => {
    db = req.body;

    res.json(db);
});

app.use(express.static('/dist'));

app.listen();
```

or NestJS servers like this:

```
import { NestFactory } from '@nestjs/core';
import { NestExpressApplication } from '@nestjs/platform-express';

import { AppModule } from './app.module';

async function bootstrap() {
    const app = await NestFactory.create<NestExpressApplication>(AppModule);
    await app.listen(3000);
}

bootstrap();
```

Servers

- Node.js http.server
- Express
- Server
- Limitations

Azle supports building HTTP servers on ICP using the Node.js http.Server class as the foundation. These servers can serve static files or act as API backends, or both.

Azle currently has good but not comprehensive support for Node.js http.Server and Express. Support for other libraries like Nest are works-in-progress.

Once deployed you can access your server at a URL like this locally `http://bkyz2-fmaaa-aaaaa-qaaaq-cai.raw.localhost:8000` or like this on mainnet `https://bkyz2-fmaaa-aaaaa-qaaaq-cai.raw.icp0.io`.

You can use any HTTP client to interact with your server, such as `curl`, `fetch`, or a web browser. See the Interacting with your canister section of the deployment chapter for help in constructing your canister URL.

Node.js http.server

Azle supports instances of Node.js http.Server. `listen()` must be called on the server instance for Azle to use it to handle HTTP requests. Azle does not respect a port being passed into `listen()`. The port is set by the ICP replica (e.g. `dfx start --host 127.0.0.1:8000`), not by Azle.

Here's an example of a very simple Node.js http.Server:

```
import { createServer } from 'http';

const server = createServer((req, res) => {
    res.write('Hello World!');
    res.end();
});

server.listen();
```

Express

Express is one of the most popular backend JavaScript web frameworks, and it's the recommended way to get started building servers in Azle. Here's the main code from the hello_world_http_server example:

```
import express, { Request } from 'express';

let db = {
    hello: ''
};

const app = express();

app.use(express.json());

app.get('/db', (req, res) => {
    res.json(db);
});

app.post('/db/update', (req: Request<any, any, typeof db>, res) => {
    db = req.body;

    res.json(db);
});

app.use(express.static('/dist'));

app.listen();
```

jsonStringify

When working with `res.json` you may run into errors because of attempting to send back JavaScript objects that are not strictly `JSON`. This can happen when trying to send back an object with a `BigInt` for example.

Azle has created a special function called `jsonStringify` that will serialize many ICP-specific data structures to `JSON` for you:

```
import { jsonStringify } from 'azle/experimental';
import express, { Request } from 'express';

let db = {
    bigInt: 0n
};

const app = express();

app.use(express.json());

app.get('/db', (req, res) => {
    res.send(jsonStringify(db));
});

app.post('/db/update', (req: Request<any, any, typeof db>, res) => {
    db = req.body;

    res.send(jsonStringify(db));
});

app.use(express.static('/dist'));

app.listen();
```

Server

If you need to add canister methods to your HTTP server, the `Server` function imported from `azle` allows you to do so.

Here's an example of a very simple HTTP server:

```
import { Server } from 'azle/experimental';
import express from 'express';

export default Server(() => {
    const app = express();

    app.get('/http-query', (_req, res) => {
        res.send('http-query-server');
    });

    app.post('/http-update', (_req, res) => {
        res.send('http-update-server');
    });

    return app.listen();
});
```

You can add canister methods like this:

```
import { query, Server, text, update } from 'azle/experimental';
import express from 'express';

export default Server(
    () => {
        const app = express();

        app.get('/http-query', (_req, res) => {
            res.send('http-query-server');
        });

        app.post('/http-update', (_req, res) => {
            res.send('http-update-server');
        });

        return app.listen();
    },
    {
        candidQuery: query([], text, () => {
            return 'candidQueryServer';
        }),
        candidUpdate: update([], text, () => {
            return 'candidUpdateServer';
        })
    }
);
```

The `default` export of your `main` module must be the result of calling `Server`, and the callback argument to `Server` must return a Node.js http.Server. The `main` module is specified by the `main` property of your project's dfx.json file. The `dfx.json` file must be at the root directory of your project.

The callback argument to `Server` can be asynchronous:

```
import { Server } from 'azle/experimental';
import { createServer } from 'http';

export default Server(async () => {
    const message = await asynchronousHelloWorld();

    return createServer((req, res) => {
        res.write(message);
        res.end();
    });
});

async function asynchronousHelloWorld() {
    // do some asynchronous task
    return 'Hello World Asynchronous!';
}
```

Limitations

For a deeper understanding of possible limitations you may want to refer to The HTTP Gateway Protocol Specification.

- The top-level route `/api` is currently reserved by the replica locally
- The `Transfer-Encoding` header is not supported
- `gzip` responses most likely do not work
- HTTP requests are generally limited to ~2 MiB
- HTTP responses are generally limited to ~3 MiB
- You cannot set HTTP status codes in the 1xx range

Assets TL;DR

You can automatically copy static assets (essentially files and folders) into your canister's filesystem during deploy by using the `assets` and `build_assets` properties of the canister object in your project's `dfx.json` file.

Here's an example that copies the `src/frontend/dist` directory on the deploying machine into the `dist` directory of the canister, using the `assets` and `build_assets` properties:

```
{
    "canisters": {
        "backend": {
            "type": "azle",
            "main": "src/backend/index.ts",
            "custom": {
                "experimental": true,
                "assets": [["src/frontend/dist", "dist"]],
                "build_assets": "npm run build"
            }
        }
    }
}
```

The `assets` property is an array of tuples, where the first element of the tuple is the source directory on the deploying machine, and the second element of the tuple is the destination directory in the canister. Use `assets` for total assets up to ~2 GiB in size. We are working on increasing this limit further.

The `build_assets` property allows you to specify custom terminal commands that will run before Azle copies the assets into the canister. You can use `build_assets` to build your frontend code for example. In this case we are running `npm run build`, which refers to an npm script that we have specified in our `package.json` file.

Once you have loaded assets into your canister, they are accessible from that canister's filesystem. Here's an example of using the Express static middleware to serve a frontend from the canister's filesystem:

```
import express from 'express';

const app = express();

app.use(express.static('/dist'));

app.listen();
```

Assuming the `/dist` directory in the canister has an appropriate `index.html` file, this

canister would serve a frontend at its URL when loaded in a web browser.

Authentication TL;DR

Azle canisters can import `caller` from `azle` and use it to get the principal (public-key linked identifier) of the initiator of an HTTP request. HTTP requests are anonymous (principal `2vxsx-fae`) by default, but authentication with web browsers (and maybe Node.js) can be done using a JWT-like API from `azle/experimental/http_client`.

First you import `toJwt` from `azle/experimental/http_client`:

```
import { toJwt } from 'azle/experimental/http_client';
```

Then you use `fetch` and construct an `Authorization` header using an @dfinity/agent `Identity`:

```
const response = await fetch(
    `http://bkyz2-fmaaa-aaaaa-qaaaq-cai.raw.localhost:8000/whoami`,
    {
        method: 'GET',
        headers: [['Authorization', toJwt(this.identity)]]
    }
);
```

Here's an example of the frontend of a simple web application using `azle/experimental/http_client` and Internet Identity:

```
import { Identity } from '@dfinity/agent';
import { AuthClient } from '@dfinity/auth-client';
import { toJwt } from 'azle/experimental/http_client';
import { html, LitElement } from 'lit';
import { customElement, property } from 'lit/decorators.js';

@customElement('azle-app')
export class AzleApp extends LitElement {
    @property()
    identity: Identity | null = null;

    @property()
    whoami: string = '';

    connectedCallback() {
        super.connectedCallback();
        this.authenticate();
    }

    async authenticate() {
        const authClient = await AuthClient.create();
        const isAuthenticated = await authClient.isAuthenticated();

        if (isAuthenticated === true) {
            this.handleIsAuthenticated(authClient);
        } else {
            await this.handleIsNotAuthenticated(authClient);
        }
    }

    handleIsAuthenticated(authClient: AuthClient) {
        this.identity = authClient.getIdentity();
    }

    async handleIsNotAuthenticated(authClient: AuthClient) {
        await new Promise((resolve, reject) => {
            authClient.login({
                identityProvider: import.meta.env.VITE_IDENTITY_PROVIDER,
                onSuccess: resolve as () => void,
                onError: reject,
                windowOpenerFeatures: `width=500,height=500`
            });
        });

        this.identity = authClient.getIdentity();
    }

    async whoamiUnauthenticated() {
        const response = await fetch(
            `${import.meta.env.VITE_CANISTER_ORIGIN}/whoami`
        );
        const responseText = await response.text();

        this.whoami = responseText;
    }
```

```
    async whoamiAuthenticated() {
        const response = await fetch(
            `${import.meta.env.VITE_CANISTER_ORIGIN}/whoami`,
            {
                method: 'GET',
                headers: [['Authorization', toJwt(this.identity)]]
            }
        );
        const responseText = await response.text();

        this.whoami = responseText;
    }

    render() {
        return html`
            <h1>Internet Identity</h1>

            <h2>
                Whoami principal:
                <span id="whoamiPrincipal">${this.whoami}</span>
            </h2>

            <button
                id="whoamiUnauthenticated"
                @click=${this.whoamiUnauthenticated}
            >
                Whoami Unauthenticated
            </button>
            <button
                id="whoamiAuthenticated"
                @click=${this.whoamiAuthenticated}
                .disabled=${this.identity === null}
            >
                Whoami Authenticated
            </button>
        `;
    }
}
```

Here's an example of the backend of that same simple web application:

```
import { caller } from 'azle';
import express from 'express';

const app = express();

app.get('/whoami', (req, res) => {
    res.send(caller().toString());
});

app.use(express.static('/dist'));

app.listen();
```

Authentication

Examples:

- fetch_ic
- internet_identity

Under-the-hood

Authentication of ICP calls is done through signatures on messages. @dfinity/agent provides very nice abstractions for creating all of the required signatures in the correct formats when calling into canisters on ICP. Unfortunately this requires you to abandon traditional HTTP requests, as you must use the agent's APIs.

Azle attempts to enable you to perform traditional HTTP requests with traditional libraries. Currently Azle focuses on `fetch`. When importing `toJwt`, `azle/experimental/http_client` will overwrite the global `fetch` function and will intercept `fetch` requests that have `Authorization` headers with an `Identity` as a value.

Once intercepted, these requests are turned into `@dfinity/agent` requests that call the http_request and http_request_update canister methods directly, thus performing all of the required client-side authentication work.

We are working to push for ICP to more natively understand JWTs for authentication, without the need to intercept `fetch` requests and convert them into agent requests.

fetch TL;DR

Azle canisters use a custom `fetch` implementation to perform cross-canister calls and to perform HTTPS outcalls.

Here's an example of performing a cross-canister call:

```
import { serialize } from 'azle/experimental';
import express from 'express';

const app = express();

app.use(express.json());

app.post('/cross-canister-call', async (req, res) => {
    const to: string = req.body.to;
    const amount: number = req.body.amount;

    const response = await fetch(`icp://dfdal-2uaaa-aaaaa-qaama-cai/
transfer`, {
        body: serialize({
            candidPath: '/token.did',
            args: [to, amount]
        })
    });
    const responseJson = await response.json();

    res.json(responseJson);
});

app.listen();
```

Keep these important points in mind when performing a cross-canister call:

- Use the `icp://` protocol in the URL
- The `canister id` of the canister that you are calling immediately follows `icp://` in the URL
- The `canister method` that you are calling immediately follows the `canister id` in the URL
- The `candidPath` property of the `body` is the path to the Candid file defining the method signatures of the canister that you are calling. You must obtain this file and copy it into your canister. See the Assets chapter for info on copying files into your canister
- The `args` property of the `body` is an array of the arguments that will be passed to the `canister method` that you are calling

Here's an example of performing an HTTPS outcall:

```
import express from 'express';

const app = express();

app.use(express.json());

app.post('/https-outcall', async (_req, res) => {
    const response = await fetch(`https://httpbin.org/headers`, {
        headers: {
            'X-Azle-Request-Key-0': 'X-Azle-Request-Value-0',
            'X-Azle-Request-Key-1': 'X-Azle-Request-Value-1',
            'X-Azle-Request-Key-2': 'X-Azle-Request-Value-2'
        }
    });
    const responseJson = await response.json();

    res.json(responseJson);
});

app.listen();
```

fetch

Azle has custom `fetch` implementations for clients and canisters.

The client `fetch` is used for authentication, and you can learn more about it in the Authentication chapter.

Canister `fetch` is used to perform cross-canister calls and HTTPS outcalls. There are three main types of calls made with canister `fetch`:

1. Cross-canister calls to a candid canister
2. Cross-canister calls to an HTTP canister
3. HTTPS outcalls

Cross-canister calls to a candid canister

Examples:

- async_await
- bitcoin
- canister
- ckbtc
- composite_queries
- cross_canister_calls
- cycles

- func_types
- heartbeat
- ic_evm_rpc
- icrc
- ledger_canister
- management_canister
- threshold_ecdsa
- whoami
- recursion
- rejections
- timers

Cross-canister calls to an HTTP canister

We are working on better abstractions for these types of calls. For now you would just make a cross-canister call using `icp://` to the `http_request` and `http_request_update` methods of the canister that you are calling.

HTTPS outcalls

Examples:

- ethereum_json_rpc
- http_outcall_fetch
- outgoing_http_requests

npm TL;DR

If you want to know if an npm package will work with Azle, just try out the package.

It's extremely difficult to know generally if a package will work unless it has been tried out and tested already. This is due to the complexity of understanding and implementing all required JavaScript, web, Node.js, and OS-level APIs required for an `npm` package to execute correctly.

To get an idea for which `npm` packages are currently supported, the Azle examples are full of example code with tests.

You can also look at the wasmedge-quickjs documentation here and here, as `wasmedge-quickjs` is our implementation for much of the Node.js stdlib.

npm

Azle's goal is to support as many npm packages as possible.

The current reality is that not all `npm` packages work well with Azle. It is also very difficult to determine which `npm` packages might work well.

For example, when asked about a specific package, we usually cannot say whether or not a given package "works". To truly know if a package will work for your situation, the easiest thing to do is to install it, import it, and try it out.

If you do want to reason about whether or not a package is likely to work, consider the following:

1. Which web or Node.js APIs does the package use?
2. Does the package depend on functionality that ICP supports?
3. Will the package stay within these limitations?

For example, any kind of networking outside of HTTP is unlikely to work (without modification), because ICP has very limited support for non-ICP networking.

Also any kind of heavy computation is unlikely to work (without modification), because ICP has very limited instruction limits per call.

We use wasmedge-quickjs as our implementation for much of the Node.js stdlib. To get a feel for which Node.js standard libraries Azle supports, see here and here.

Tokens TL;DR

Canisters can either:

1. Interact with tokens that already exist
2. Implement, extend, or proxy tokens

Canisters can use cross-canister calls to interact with tokens implemented using ICRC or other standards. They can also interact with non-ICP tokens through threshold ECDSA.

Canisters can implement tokens from scratch, or extend or proxy implementations already written.

Demergent Labs does not keep any token implementations up-to-date. Here are some old implementations for inspiration and learning:

- ICRC-1
- extendable-token-azle

Tokens

Examples:

- basic_bitcoin
- bitcoin
- bitcoinjs-lib
- bitcore-lib
- ckbtc
- ethereum_json_rpc
- ethers
- ethers_base
- extendable-token-azle
- ic_evm_rpc
- icrc
- ICRC-1
- ledger_canister

Bitcoin

Examples:

- basic_bitcoin
- bitcoin
- bitcoin_psbt
- bitcoinjs_lib
- bitcore_lib
- ckbtc

There are two main ways to interact with Bitcoin on ICP: through the management canister and through the ckBTC canister.

management canister

To sign Bitcoin transactions using threshold ECDSA and interact with the Bitcoin blockchain directly from ICP, make cross-canister calls to the following methods on the management canister: `ecdsa_public_key`, `sign_with_ecdsa`, `bitcoin_get_balance`, `bitcoin_get_balance_query`, `bitcoin_get_utxos`, `bitcoin_get_utxos_query`, `bitcoin_send_transaction`, `bitcoin_get_current_fee_percentiles`.

To construct your cross-canister calls to these methods, use `canister id aaaaa-aa` and the management canister's Candid type information to construct the arguments to send in the `body` of your `fetch` call.

Here's an example of doing a test cross-canister call to the `bitcoin_get_balance` method:

```
import { serialize } from 'azle/experimental';

// ...

const response = await fetch(`icp://aaaaa-aa/bitcoin_get_balance`, {
    body: serialize({
        args: [
            {
                'bc1q34aq5drpuwy3wgl9lhup9892qp6svr8ldzyy7c',
                min_confirmations: [],
                network: { regtest: null }
            }
        ],
        cycles: 100_000_000n
    })
});
const responseJson = await response.json();

// ...
```

ckBTC

ckBTC is an ICRC canister that wraps underlying bitcoin controlled with threshold ECDSA.

ICRCs are a set of standards for ICP canisters that define the method signatures and corresponding types for those canisters.

You interact with the `ckBTC` canister by calling its methods. You can do this from the frontend with @dfinity/agent, or from an Azle canister through cross-canister calls.

Here's an example of doing a test cross-canister call to the `ckBTC` `icrc1_balance_of` method:

```
import { ic, serialize } from 'azle/experimental';

// ...

const response = await fetch(
    `icp://mc6ru-gyaaa-aaaar-qaaaq-cai/icrc1_balance_of`,
    {
        body: serialize({
            candidPath: `/candid/icp/icrc.did`,
            args: [
                {
                    owner: ic.id(),
                    subaccount: [
                        padPrincipalWithZeros(ic.caller().toUint8Array())
                    ]
                }
            ]
        })
    }
);
const responseJson = await response.json();

// ...

function padPrincipalWithZeros(principalBlob: Uint8Array): Uint8Array {
    let newUin8Array = new Uint8Array(32);
    newUin8Array.set(principalBlob);
    return newUin8Array;
}
```

Ethereum

Examples:

- ethereum_json_rpc
- ethers
- ethers_base
- ic_evm_rpc

Databases

The eventual goal for Azle is to support as many database solutions as possible. This is difficult for a number of reasons related to ICP's decentralized computing paradigm and Wasm environment.

SQLite is the current recommended approach to databases with Azle. We plan to provide Postgres support through pglite next.

Azle has good support for SQLite through sql.js. It also has good support for ORMs like Drizzle and TypeORM using `sql.js`.

The following examples should be very useful as you get started using SQLite in Azle:

Examples:

- sqlite
- sqlite_drizzle
- sqlite_typeorm

sql.js

SQLite in Azle works using an asm.js build of SQLite from `sql.js` without modifications to the library. The database is stored entirely in memory on the heap, giving you ~2 GiB of space. Serialization across upgrades is possible using stable memory like this:

```ts
// src/index.its

import {
    init,
    postUpgrade,
    preUpgrade,
    Server,
    StableBTreeMap,
    stableJson
} from 'azle/experimental';
import { Database } from 'sql.js/dist/sql-asm.js';

import { initDb } from './db';
import { initServer } from './server';

export let db: Database;

let stableDbMap = StableBTreeMap<'DATABASE', Uint8Array>(0, stableJson, {
    toBytes: (data: Uint8Array) => data,
    fromBytes: (bytes: Uint8Array) => bytes
});

export default Server(initServer, {
    init: init([], async () => {
        db = await initDb();
    }),
    preUpgrade: preUpgrade(() => {
        stableDbMap.insert('DATABASE', db.export());
    }),
    postUpgrade: postUpgrade([], async () => {
        db = await initDb(stableDbMap.get('DATABASE').Some);
    })
});
```

```ts
// src/db/index.ts

import initSqlJs, {
    Database,
    QueryExecResult,
    SqlValue
} from 'sql.js/dist/sql-asm.js';

import { migrations } from './migrations';

export async function initDb(
    bytes: Uint8Array = Uint8Array.from([])
): Promise<Database> {
    const SQL = await initSqlJs({});

    let db = new SQL.Database(bytes);

    if (bytes.length === 0) {
        for (const migration of migrations) {
            db.run(migration);
        }
    }

    return db;
}
```

Debugging TL;DR

If your terminal logs ever say `did not produce a response` or `response failed classification=Status code: 502 Bad Gateway`, it most likely means that your canister has thrown an error and halted execution for that call. Use `console.log` and `try/catch` liberally to track down problems and reveal error information. If your error logs do not have useful messages, use `try/catch` with a `console.log` of the catch error argument to reveal the underlying error message.

Debugging

- console.log and try/catch
- Canister did not produce a response
- No error message
- Final Compiled and Bundled JavaScript

Azle currently has less-than-elegant error reporting. We hope to improve this significantly in the future.

In the meantime, consider the following tips when trying to debug your application.

console.log and try/catch

At the highest level, the most important tip is this: use `console.log` and `try/catch` liberally to track down problems and reveal error information.

Canister did not produce a response

If you ever see an error that looks like this:

```
Replica Error: reject code CanisterError, reject message IC0506: Canister
bkyz2-fmaaa-aaaaa-qaaaq-cai did not produce a response, error code
Some("IC0506")
```

or this:

```
2024-04-17T15:01:39.194377Z  WARN icx_proxy_dev::proxy::agent: Replica Error
2024-04-17T15:01:39.194565Z ERROR tower_http::trace::on_failure: response
failed classification=Status code: 502 Bad Gateway latency=61 ms
```

it most likely means that your canister has thrown an error and halted execution for that call. First check the replica's logs for any errors messages. If there are no useful error messages, use console.log and try/catch liberally to track down the source of the error and to reveal more information about the error.

Don't be surprised if you need to `console.log` after each of your program's statements (including dependencies found in `node_modules`) to find out where the error is coming from. And don't be surprised if you need to use `try/catch` with a `console.log` of the catch error argument to reveal useful error messaging.

No error message

You might find yourself in a situation where an error is reported without a useful message like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>    at &lt;anonymous&gt; (.azle/main.js:110643)<br>  
 at handle (.azle/main.js:73283)<br>    at next (.azle/
main.js:73452)<br>    at dispatch (.azle/main.js:73432)<br>  
 at handle (.azle/main.js:73283)<br>    at &lt;anonymous&gt;
(.azle/main.js:73655)<br>    at process_params (.azle/
main.js:73692)<br>    at next (.azle/main.js:73660)<br>  
 at expressInit (.azle/main.js:73910)<br>    at handle (.azle/
main.js:73283)<br>    at trim_prefix (.azle/main.js:73684)<br>
   at &lt;anonymous&gt; (.azle/main.js:73657)<br>    at
process_params (.azle/main.js:73692)<br>    at next (.azle/
main.js:73660)<br>    at query3 (.azle/main.js:73938)<br>  
 at handle (.azle/main.js:73283)<br>    at trim_prefix (.azle/
main.js:73684)<br>    at &lt;anonymous&gt; (.azle/
main.js:73657)<br>    at process_params (.azle/main.js:73692)<br>
   at next (.azle/main.js:73660)<br>    at handle (.azle/
main.js:73587)<br>    at handle (.azle/main.js:76233)<br>  
 at app2 (.azle/main.js:78091)<br>    at call (native)<br>
   at emitTwo (.azle/main.js:9782)<br>    at emit2
(.azle/main.js:10023)<br>    at httpHandler (.azle/
main.js:87618)<br></pre>
</body>
</html>
```

or like this:

```
2024-04-17 14:35:30.433501980 UTC: [Canister bkyz2-fmaaa-aaaaa-qaaaq-cai] "
at <anonymous> (.azle/main.js:110643)\n    at handle (.azle/main.js:73283)\n
at next (.azle/main.js:73452)\n    at dispatch (.azle/main.js:73432)\n    at
handle (.azle/main.js:73283)\n    at <anonymous> (.azle/main.js:73655)\n
at process_params (.azle/main.js:73692)\n    at next (.azle/main.js:73660)\n
at expressInit (.azle/main.js:73910)\n    at handle (.azle/main.js:73283)\n
at trim_prefix (.azle/main.js:73684)\n    at <anonymous> (.azle/
main.js:73657)\n    at process_params (.azle/main.js:73692)\n    at next
(.azle/main.js:73660)\n    at query3 (.azle/main.js:73938)\n    at handle
(.azle/main.js:73283)\n    at trim_prefix (.azle/main.js:73684)\n    at
<anonymous> (.azle/main.js:73657)\n    at process_params (.azle/
main.js:73692)\n    at next (.azle/main.js:73660)\n    at handle (.azle/
main.js:73587)\n    at handle (.azle/main.js:76233)\n    at app2 (.azle/
main.js:78091)\n    at call (native)\n    at emitTwo (.azle/main.js:9782)\n
at emit2 (.azle/main.js:10023)\n    at httpHandler (.azle/main.js:87618)\n"
2024-04-17T14:35:31.983590Z ERROR tower_http::trace::on_failure: response
failed classification=Status code: 500 Internal Server Error latency=101 ms
2024-04-17 14:36:34.652587412 UTC: [Canister bkyz2-fmaaa-aaaaa-qaaaq-cai] "
at <anonymous> (.azle/main.js:110643)\n    at handle (.azle/main.js:73283)\n
at next (.azle/main.js:73452)\n    at dispatch (.azle/main.js:73432)\n    at
handle (.azle/main.js:73283)\n    at <anonymous> (.azle/main.js:73655)\n
at process_params (.azle/main.js:73692)\n    at next (.azle/main.js:73660)\n
at expressInit (.azle/main.js:73910)\n    at handle (.azle/main.js:73283)\n
at trim_prefix (.azle/main.js:73684)\n    at <anonymous> (.azle/
main.js:73657)\n    at process_params (.azle/main.js:73692)\n    at next
(.azle/main.js:73660)\n    at query3 (.azle/main.js:73938)\n    at handle
(.azle/main.js:73283)\n    at trim_prefix (.azle/main.js:73684)\n    at
<anonymous> (.azle/main.js:73657)\n    at process_params (.azle/
main.js:73692)\n    at next (.azle/main.js:73660)\n    at handle (.azle/
main.js:73587)\n    at handle (.azle/main.js:76233)\n    at app2 (.azle/
main.js:78091)\n    at call (native)\n    at emitTwo (.azle/main.js:9782)\n
at emit2 (.azle/main.js:10023)\n    at httpHandler (.azle/main.js:87618)\n"
```

In these situations you might be able to use `try/catch` with a `console.log` of the catch error argument to reveal the underlying error message.

For example, this code without a `try/catch` will log errors without the message `This is the error text`:

```
import express from 'express';

const app = express();

app.get('/hello-world', (_req, res) => {
    throw new Error('This is the error text');
    res.send('Hello World!');
});

app.listen();
```

You can get the message to print in the replica terminal like this:

```
import express from 'express';

const app = express();

app.get('/hello-world', (_req, res) => {
    try {
        throw new Error('This is the error text');
        res.send('Hello World!');
    } catch (error) {
        console.log(error);
    }
});

app.listen();
```

Final Compiled and Bundled JavaScript

Azle compiles and bundles your TypeScript/JavaScript into a final JavaScript file to be included and executed inside of your canister. Inspecting this final JavaScript code may help you to debug your application.

When you see something like `(.azle/main.js:110643)` in your error stack traces, it is a reference to the final compiled and bundled JavaScript file that is actually deployed with and executed by the canister. The right-hand side of `.azle/main.js` e.g. `:110643` is the line number in that file.

You can find the file at `[project_name]/.azle/[canister_name]/canister/src/main.js`. If you have the `AZLE_AUTORELOAD` environment variable set to `true` then you should instead look at `[project_name]/.azle/[canister_name]/canister/src/main_reloaded.js`

Limitations TL;DR

There are a number of limitations that you are likely to run into while you develop with Azle on ICP. These are generally the most limiting:

- 5 billion instruction limit for query calls (HTTP GET requests) (~1 second of computation)
- 40 billion instruction limit for update calls (HTTP POST/etc requests) (~10 seconds of computation)
- 2 MiB request size limit
- 3 MiB response size limit
- 4 GiB heap limit
- High request latency relative to traditional web applications (think seconds not milliseconds)
- High costs relative to traditional web applications (think ~10x traditional web costs)
- StableBTreeMap memory id `254` is reserved for the stable memory file system

Read more here for in-depth information on current ICP limitations.

Reference

- Autoreload
- Environment Variables

Autoreload

You can turn on automatic reloading of your canister's final compiled JavaScript by using the `AZLE_AUTORELOAD` environment variable during deploy:

```
AZLE_AUTORELOAD=true dfx deploy
```

The autoreload feature watches all `.ts` and `.js` files recursively in the directory with your `dfx.json` file (the root directory of your project), excluding files found in `.azle`, `.dfx`, and `node_modules`.

Autoreload only works properly if you do not change the methods of your canister. HTTP-based canisters will generally work well with autoreload as the query and update methods `http_request` and `http_request_update` will not need to change often. Candid-based canisters with explicit `query` and `update` methods may require manual deploys more often.

Autoreload will not reload assets uploaded through the `assets` property of your `dfx.json`.

Setting `AZLE_AUTORELOAD=true` will create a new `dfx` identity and set it as a controller of your canister. By default it will be called `_azle_file_uploader_identity`. This name can be changed with the `AZLE_UPLOADER_IDENTITY_NAME` environment variable.

Environment Variables

- AZLE_AUTORELOAD
- AZLE_IDENTITY_STORAGE_MODE
- AZLE_INSTRUCTION_COUNT
- AZLE_PROPTEST_NUM_RUNS
- AZLE_PROPTEST_PATH
- AZLE_PROPTEST_QUIET
- AZLE_PROPTEST_SEED
- AZLE_PROPTEST_VERBOSE
- AZLE_TEST_FETCH
- AZLE_UPLOADER_IDENTITY_NAME
- AZLE_VERBOSE

AZLE_AUTORELOAD

Set this to `true` to enable autoreloading of your TypeScript/JavaScript code when making any changes to `.ts` or `.js` files in your project.

AZLE_IDENTITY_STORAGE_MODE

Used for automated testing.

AZLE_INSTRUCTION_COUNT

Set this to `true` to see rough instruction counts just before JavaScript execution completes for calls.

AZLE_PROPTEST_NUM_RUNS

Used for automated testing.

AZLE_PROPTEST_PATH

Used for automated testing.

AZLE_PROPTEST_QUIET

Used for automated testing.

AZLE_PROPTEST_SEED

Used for automated testing.

AZLE_PROPTEST_VERBOSE

Used for automated testing.

AZLE_TEST_FETCH

Used for automated testing.

AZLE_UPLOADER_IDENTITY_NAME

Change the name of the `dfx` identity added as a controller for uploading large assets and autoreload.

AZLE_VERBOSE

Set this to `true` to enable more logging output during `dfx deploy`.