The Azle Book

This book is intended to be an in-depth guide to canister development in TypeScript on the Internet Computer (IC).

The first 19 chapters are an introductory guide into canister development with Azle. These chapters build on each other concept by concept, introducing the fundamentals required to create and deploy canisters to the IC.

Chapter 20 is an in-depth reference of the APIs available to Azle canisters.

Our intention is for new developers to use this book as a tutorial or course, starting at chapter 1 and working through chapter 19, using chapter 20 as a reference.

There will also be a companion video series on YouTube. Each chapter here will begin with the video companion as soon as it is available.

You should expect this book and its companion video series to continue to grow and change over time, as its authors and the IC grow and change.

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

Azle (Beta)

Azle is a TypeScript and JavaScript Canister Development Kit (CDK) for the Internet Computer (IC). In other words, it's a TypeScript/JavaScript runtime for building applications (canisters) on the IC.

- npm package
- GitHub repo
- Discord channel

Disclaimer

Azle may have unknown security vulnerabilities due to the following:

- Azle does not yet have extensive automated property tests
- Azle does not yet have multiple independent security reviews/audits
- Azle does not yet have many live, successful, continuously operating applications deployed to the IC

Roadmap

We hope to move Azle from beta to release candidates by the end of 2023, and to move from release candidates to 1.0 in early 2024.

Blockers for release candidates

- Good automated property test coverage
- Settling of API/syntax
- Good npm package support

Blockers for 1.0

- Extensive automated property test coverage
- Multiple independent security reviews/audits
- Broad npm package support

Demergent Labs

Azle is currently developed by Demergent Labs, a for-profit company with a grant from DFINITY.

Demergent Labs' vision is to accelerate the adoption of Web3, the Internet Computer, and sustainable open source.

Benefits and drawbacks

Azle and the IC provide unique benefits and drawbacks, and both are not currently suitable for all application use-cases.

The following information will help you to determine when Azle and the IC might be beneficial for your use-case.

Benefits

Azle intends to be a full TypeScript and JavaScript environment for the IC (a decentralized cloud platform), with support for all of the TypeScript and JavaScript language and as many relevant environment APIs as possible. These environment APIs will be similar to those available in the Node.js and web browser environments.

One of the core benefits of Azle is that it allows web developers to bring their TypeScript or JavaScript skills to the IC. For example, Azle allows the use of various npm packages and VS Code intellisense.

As for the IC, we believe its main benefits can be broken down into the following categories:

- Ownership
- Security
- Developer Experience

Most of these benefits stem from the decentralized nature of the IC, though the IC is best thought of as a progressively decentralizing cloud platform. As opposed to traditional cloud platforms, its goal is to be owned and controlled by many independent entities.

Ownership

- Full-stack group ownership
- Autonomous ownership

- Permanent APIs
- Credible neutrality
- Reduced platform risk

Full-stack group ownership

The IC allows you to build applications that are controlled directly and only (with some caveats) by a group of people. This is in opposition to most cloud applications written today, which must be under the control of a very limited number of people and often a single legal entity that answers directly to a cloud provider, which itself is a single legal entity.

In the blockchain world, group-owned applications are known as DAOs. As opposed to DAOs built on most blockchains, the IC allows full-stack applications to be controlled by groups. This means that the group fully controls the running instances of the frontend and the backend code.

Autonomous ownership

In addition to allowing applications to be owned by groups of people, the IC also allows applications to be owned by no one. This essentially creates autonomous applications or everlasting processes that execute indefinitely. The IC will essentially allow such an application to run indefinitely, unless it depletes its balance of cycles, or the NNS votes to shut it down, neither of which is inevitable.

Permanent APIs

Because most web APIs are owned and operated by individual entities, their fate is tied to that of their owners. If their owners go out of business, then those APIs may cease to exist. If their owners decide that they do not like or agree with certain users, they may restrict their access. In the end, they may decide to shut down or restrict access for arbitrary reasons.

Because the IC allows for group and autonomous ownership of cloud software, the IC is able to produce potentially permanent web APIs. A decentralized group of independent entities will find it difficult to censor API consumers or shut down an API. An autonomous API would take those difficulties to the extreme, as it would continue operating as long as consumers were willing to pay for it.

Credible neutrality

Group and autonomous ownership makes it possible to build neutral cloud software on the IC. This type of software would allow independent parties to coordinate with reduced trust in each other or a single third-party coordinator.

This removes the risk of the third-party coordinator acting in its own self-interest against the interests of the coordinating participants. The coordinating participants would also find it difficult to implement changes that would benefit themselves to the detriment of other participants.

Examples could include mobile app stores, ecommerce marketplaces, and podcast directories.

Reduced platform risk

Because the IC is not owned or controlled by any one entity or individual, the risk of being deplatformed is reduced. This is in opposition to most cloud platforms, where the cloud provider itself generally has the power to arbitrarily remove users from its platform. While deplatforming can still occur on the IC, the only endogenous means of forcefully taking down an application is through an NNS vote.

Security

- Built-in replication
- Built-in authentication
- Built-in firewall/port management
- Built-in sandboxing
- Threshold protocols
- Verifiable source code
- Blockchain integration

Built-in replication

Replication has many benefits that stem from reducing various central points of failure.

The IC is at its core a Byzantine Fault Tolerant replicated compute environment. Applications are deployed to subnets which are composed of nodes running replicas. Each replica is an independent replicated state machine that executes an application's state transitions (usually initiated with HTTP requests) and persists the results.

This replication provides a high level of security out-of-the-box. It is also the foundation of a number of protocols that provide threshold cryptographic operations to IC applications.

Built-in authentication

IC client tooling makes it easy to sign and send messages to the IC, and Internet Identity provides a novel approach to self-custody of private keys. The IC automatically authenticates messages with the public key of the signer, and provides a compact representation of that public key, called a principal, to the application. The principal can be used for authorization purposes. This removes many authentication concerns from the developer.

Built-in firewall/port management

The concept of ports and various other low-level network infrastructure on the IC is abstracted away from the developer. This can greatly reduce application complexity thus minimizing the chance of introducing vulnerabilities through incorrect configurations. Canisters expose endpoints through various methods, usually query or update methods. Because authentication is also built-in, much of the remaining vulnerability surface area is minimized to implementing correct authorization rules in the canister method endpoints.

Built-in sandboxing

Canisters have at least two layers of sandboxing to protect colocated canisters from each other. All canisters are at their core Wasm modules and thus inherit the built-in Wasm sandbox. In case there is any bug in the underlying implementation of the Wasm execution environment (or a vulnerability in the imported host functionality), there is also an OS-level sandbox. Developers need not do anything to take advantage of these sandboxes.

Threshold protocols

The IC provides a number of threshold protocols that allow groups of independent nodes to perform cryptographic operations. These protocols remove central points of failure while providing familiar and useful cryptographic operations to developers. Included are ECDSA, BLS, VRF-like, and in the future threshold key derivation.

Verifiable source code

IC applications (canisters) are compiled into Wasm and deployed to the IC as Wasm modules. The IC hashes each canister's Wasm binary and stores it for public retrieval. The Wasm binary hash can be retrieved and compared with the hash of an independently compiled Wasm binary derived from available source code. If the hashes match, then one can know with a high degree of certainty that the application is executing the Wasm binary that was compiled from that source code.

Blockchain integration

When compared with web APIs built for the same purpose, the IC provides a high degree of security when integrating with various other blockchains. It has a direct client integration with Bitcoin, allowing applications to query its state with BFT guarantees. A similar integration is coming for Ethereum.

In addition to these blockchain client integrations, a threshold ECDSA protocol (tECDSA) allows the IC to create keys and sign transactions on various ECDSA chains. These chains include Bitcoin and Ethereum, and in the future the protocol may be extended to allow interaction with various EdDSA chains. These direct integrations combined with tECDSA provide a much more secure way to provide blockchain functionality to end users than creating and storing their private keys on traditional cloud infrastructure.

Developer experience

- Built-in devops
- Orthogonal persistence

Built-in devops

The IC provides many devops benefits automatically. Though currently limited in its scalability, the protocol attempts to remove the need for developers to concern themselves with concepts such as autoscaling, load balancing, uptime, sandboxing, and firewalls/port management.

Correctly constructed canisters have a simple deploy process and automatically inherit these devops capabilities up unto the current scaling limits of the IC. DFINITY engineers are constantly working to remove scalability bottlenecks.

Orthogonal persistence

The IC automatically persists its heap. This creates an extremely convenient way for developers to store application state, by simply writing into global variables in their programming language of choice. This is a great way to get started.

If a canister upgrades its code, swapping out its Wasm binary, then the heap must be cleared. To overcome this limitation, there is a special area of memory called stable memory that persists across these canister upgrades. Special stable data structures provide a familiar API that allows writing into stable memory directly.

All of this together provides the foundation for a very simple persistence experience for the developer. The persistence tools now available and coming to the IC may be simpler than their equivalents on traditional cloud infrastructure.

Drawbacks

It's important to note that both Azle and the IC are early-stage projects. The IC officially launched in May of 2021, and Azle reached beta in April of 2022.

Azle

Some of Azle's main drawbacks can be summarized as follows:

- Beta
- Security risks
- Missing APIs

Beta

Azle reached beta in April of 2022. It's an immature project that may have unforeseen bugs and other issues. We're working constantly to improve it. We hope to get to a production-ready 1.0 in 2024. The following are the major blockers to 1.0:

- Extensive automated property test coverage
- Multiple independent security reviews/audits
- Broad npm package support

Security risks

As discussed earlier, these are some things to keep in mind:

- Azle does not yet have extensive automated property tests
- Azle does not yet have multiple independent security reviews/audits
- Azle does not yet have many live, successful, continuously operating applications deployed to the IC

Missing APIs

Azle is not Node.js nor is it V8 running in a web browser. It is using a JavaScript interpreter running in a very new and very different environment. APIs from the Node.js and web browser ecosystems may not be present in Azle. Our goal is to support as many of these APIs as possible over time.

IC

Some of the IC's main drawbacks can be summarized as follows:

- Early
- High latencies
- Limited and expensive compute resources
- Limited scalability
- Lack of privacy
- NNS risk

Early

The IC launched officially in May of 2021. As a relatively new project with an extremely ambitious vision, you can expect a small community, immature tooling, and an unproven track record. Much has been delivered, but many promises are yet to be fulfilled.

High latencies

Any requests that change state on the IC must go through consensus, thus you can expect latencies of a few seconds for these types of requests. When canisters need to communicate with each other across subnets or under heavy load, these latencies can be even longer. Under these circumstances, in the worst case latencies will build up linearly. For example, if canister A calls canister B calls canister C, and these canisters are all on different subnets or under heavy load, then you might need to multiply the latency by the total number of calls.

Limited and expensive compute resources

CPU usage, data storage, and network usage may be more expensive than the equivalent usage on traditional cloud platforms. Combining these costs with the high latencies explained above, it becomes readily apparent that the IC is currently not built for high-performance computing.

Limited scalability

The IC might not be able to scale to the needs of your application. It is constantly seeking to improve scalability bottlenecks, but it will probably not be able to onboard millions of users to your traditional web application.

Lack of privacy

You should assume that all of your application data (unless it is end-to-end encrypted) is accessible to multiple third-parties with no direct relationship and limited commitment to you. Currently all canister state sits unencrypted on node operator's machines. Application-layer access controls for data are possible, but motivated node operators will have an easy time getting access to your data.

NNS risk

The NNS has the ability to uninstall any canister and can generally change anything about the IC protocol. The NNS uses a simple liquid democracy based on coin/token voting and follower relationships. At the time of this writing most of the voting power on the NNS follows DFINITY for protocol changes, effectively giving DFINITY write control to the protocol while those follower relationships remain in place. The NNS must mature and decentralize to provide practical and realistic protections to canisters and their users.

Internet Computer Overview

The Internet Computer (IC) is a decentralized cloud platform. Actually, it is better thought of as a progressively decentralizing cloud platform. Its full vision is yet to be fulfilled.

It aims to be owned and operated by many independent entities in many geographies and legal jurisdictions throughout the world. This is in opposition to most traditional cloud platforms today, which are generally owned and operated by one overarching legal entity.

The IC is composed of computer hardware nodes running the IC protocol software. Each running IC protocol software process is known as a replica.

Nodes are assigned into groups known as subnets. Each subnet attempts to maximize its decentralization of nodes according to factors such as data center location and node operator independence.

The subnets vary in size. Generally speaking the larger the size of the subnet the more secure it will be. Subnets currently range in size from 13 to 40 nodes, with most subnets having 13 nodes.

IC applications, known as canisters, are deployed to specific subnets. They are then accessible through Internet Protocol requests such as HTTP. Each subnet replicates all canisters across all of its replicas. A consensus protocol is run by the replicas to ensure Byzantine Fault Tolerance.

View the IC Dashboard to explore all data centers, subnets, node operators, and many other aspects of the IC.

Canisters Overview

Canisters are Internet Computer (IC) applications. They are the encapsulation of your code and state, and are essentially Wasm modules.

State can be stored on the 4 GiB heap or in a larger 96 GiB location called stable memory. You can store state on the heap using your language's native global variables. You can store state in stable memory using low-level APIs or special stable data structures that behave similarly to native language data structures.

State changes must go through a process called consensus. The consensus process ensures that state changes are Byzantine Fault Tolerant. This process takes a few seconds to complete.

Operations on canister state are exposed to users through canister methods. These methods can be invoked through HTTP requests. Query methods allow state to be read and are low-latency. Update methods allow state to be changed and are higher-latency. Update methods take a few seconds to complete because of the consensus process.

Installation

Follow the instructions exactly as stated below to avoid issues.

Windows is only supported through a Linux virtual environment of some kind, such as WSL. Follow these instructions to install WSL.

You should be using a *nix environment (Linux, Mac OS, WSL if using Windows) with bash and have the following installed on your system:

- Build dependencies
- Node.js 18
- dfx 0.15.1

Build dependencies

It is best to install all of these dependencies based on your OS:

Ubuntu/WSL

```
sudo apt install clang
sudo apt install build-essential
sudo apt install libssl-dev
sudo apt install pkg-config
```

Mac

```
# Install the Xcode Command Line Tools
xcode-select --install
```

Node.js

We highly recommend using nvm to install Node.js (and npm, which is included with Node.js). Run the following commands to install Node.js and npm with nvm:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh |
bash
```

Now restart your terminal and run the following command:

```
nvm install 18
```

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

```
node --version
```

dfx 0.15.1

Run the following command to install dfx 0.15.1:

```
DFX_VERSION=0.15.1 sh -ci "$(curl -fsSL https://sdk.dfinity.org/install.sh)"
```

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

```
dfx --version
```

If after trying to run `dfx --version` you encounter an error such as `dfx: command not found`, you might need to add `$HOME/bin` to your path. Here's an example of doing this in your `.bashrc`:

```
echo 'export PATH="$PATH:$HOME/bin"' >> "$HOME/.bashrc"
```

Hello World

Let's build your first application (canister) with Azle!

Before embarking please ensure you've followed all of the installation instructions, especially noting the build dependencies.

We'll build a simple `Hello World` canister that shows the basics of importing Azle, exposing a query method, exposing an update method, and storing some state in a global variable. We'll then interact with it from the command line and from our web browser.

Quick Start

We are going to use the Azle `new` command which creates a simple example project.

First use the `new` command to create a new project called `azle_hello_world`:

```
npx azle new azle_hello_world
```

Now let's go inside of our project:

```
cd azle_hello_world
```

We should install Azle and all of its dependencies:

```
npm install
```

Start up your local replica:

```
dfx start
```

In another terminal, deploy your canister:

```
dfx deploy azle_hello_world
```

Call the `setMessage` method:

```
dfx canister call azle_hello_world setMessage '("Hello world!")'
```

Call the `getMessage` method:

```
dfx canister call azle_hello_world getMessage
```

If you run into an error during deployment, see the common deployment issues section.

See the official azle_hello_world example for more information.

Methodical start

The project directory and file structure

Assuming you're starting completely from scratch, run these commands to setup your project's directory and file structure:

```
mkdir azle_hello_world
cd azle_hello_world

mkdir src

touch src/index.ts
touch tsconfig.json
touch dfx.json
```

Now install Azle, which will create your `package.json` and `package-lock.json` files:

```
npm install azle
```

Open up `azle_hello_world` in your text editor (we recommend VS Code).

index.ts

Here's the main code of the project, which you should put in the `azle_hello_world/src`

`/index.ts` file of your canister:

```
import { Canister, query, text, update, Void } from 'azle';

// This is a global variable that is stored on the heap
let message = '';

export default Canister({
    // Query calls complete quickly because they do not go through consensus
    getMessage: query([], text, () => {
        return message;
    }),
    // Update calls take a few seconds to complete
    // This is because they persist state changes and go through consensus
    setMessage: update([text], Void, (newMessage) => {
        message = newMessage; // This change will be persisted
    })
});
```

Let's discuss each section of the code.

```
import { Canister, query, text, update, Void } from 'azle';
```

The code starts off by importing `Canister`, `query`, `text`, `update` and `Void` from `azle`. The `azle` module provides most of the Internet Computer (IC) APIs for your canister.

```
// This is a global variable that is stored on the heap
let message = '';
```

We have created a global variable to store the state of our application. This variable is in scope to all of the functions defined in this module. We have set it equal to an empty string.

```
export default Canister({
    ...
});
```

The `Canister` function allows us to export our canister's definition to the Azle IC environment.

```
// Query calls complete quickly because they do not go through consensus
getMessage: query([], text, () => {
    return message;
}),
```

We are exposing a canister query method here. This method simply returns our global `message` variable. We use a `CandidType` object called `text` to instruct Azle to encode the return value as a Candid `text` value. When query methods are called they execute quickly because they do not have to go through consensus.

```
// Update calls take a few seconds to complete
// This is because they persist state changes and go through consensus
setMessage: update([text], Void, (newMessage) => {
    message = newMessage; // This change will be persisted
});
```

We are exposing an update method here. This method accepts a `string` from the caller and will store it in our global `message` variable. We use a `CandidType` object called `text` to instruct Azle to decode the `newMessage` parameter from a Candid `text` value to a JavaScript string value. Azle will infer the TypeScript type for `newMessage`. We use a `CandidType` object called `Void` to instruct Azle to encode the return value as the absence of a Candid value.

When update methods are called they take a few seconds to complete. This is because they persist changes and go through consensus. A majority of nodes in a subnet must agree on all state changes introduced in calls to update methods.

That's it! We've created a very simple getter/setter `Hello World` application. But no `Hello World` project is complete without actually yelling `Hello world`!

To do that, we'll need to setup the rest of our project.

tsconfig.json

Create the following in `azle_hello_world/tsconfig.json`:

```
{
    "compilerOptions": {
        "strict": true,
        "target": "ES2020",
        "moduleResolution": "node",
        "allowJs": true,
        "outDir": "HACK_BECAUSE_OF_ALLOW_JS"
    }
}
```

dfx.json

Create the following in `azle_hello_world/dfx.json`:

```json
{
    "canisters": {
        "azle_hello_world": {
            "type": "custom",
            "main": "src/index.ts",
            "candid": "src/index.did",
            "build": "npx azle azle_hello_world",
            "wasm": ".azle/azle_hello_world/azle_hello_world.wasm",
            "gzip": true
        }
    }
}
```

Local deployment

Let's deploy to our local replica.

First startup the replica:

```
dfx start --background
```

Then deploy the canister:

```
dfx deploy
```

Common deployment issues

If you run into an error during deployment, see the common deployment issues section.

Interacting with your canister from the command line

Once we've deployed we can ask for our message:

```
dfx canister call azle_hello_world getMessage
```

We should see `("")` representing an empty message.

Now let's yell `Hello World!`:

```
dfx canister call azle_hello_world setMessage '("Hello World!")'
```

Retrieve the message:

```
dfx canister call azle_hello_world getMessage
```

We should see `("Hello World!")`.

Interacting with your canister from the web UI

After deploying your canister, you should see output similar to the following in your terminal:

```
Deployed canisters.
URLs:
  Backend canister via Candid interface:
    azle_hello_world: http://127.0.0.1:8000/?canisterId=ryjl3-tyaaa-aaaaa-
aaaba-cai&id=rrkah-fqaaa-aaaaa-aaaaq-cai
```

Open up http://127.0.0.1:8000/?canisterId=ryjl3-tyaaa-aaaaa-aaaba-cai&id=rrkah-fqaaa-aaaaa-aaaaq-cai or the equivalent URL from your terminal to access the web UI and interact with your canister.

Deployment

There are two main Internet Computer (IC) environments that you will generally interact with: the local replica and mainnet.

When developing on your local machine, our recommended flow is to start up a local replica in your project's root directoy and then deploy to it for local testing.

Starting the local replica

Open a terminal and navigate to your project's root directory:

```
dfx start
```

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

```
dfx start --background
```

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

```
dfx stop
```

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

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

Then try this:

```
sudo kill -9 627221
sudo kill -9 626923
sudo kill -9 627260
```

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

```
dfx start --clean
```

Deploying to the local replica

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

```
dfx deploy
```

To deploy an individual canister:

```
dfx deploy canister_name
```

Interacting with your canister

As a developer you can generally interact with your canister in three ways:

- dfx command line
- dfx web UI
- @dfinity/agent

dfx command line

You can see a more complete reference here.

The commands you are likely to use most frequently are:

```
# assume a canister named my_canister

# builds and deploys all canisters specified in dfx.json
dfx deploy

# builds all canisters specified in dfx.json
dfx build

# builds and deploys my_canister
dfx deploy my_canister

# builds my_canister
dfx build my_canister

# removes the Wasm binary and state of my_canister
dfx uninstall-code my_canister

# calls the methodName method on my_canister with a string argument
dfx canister call my_canister methodName '("This is a Candid string
argument")'
```

dfx web UI

After deploying your canister, you should see output similar to the following in your terminal:

```
Deployed canisters.
URLs:
  Backend canister via Candid interface:
    my_canister: http://127.0.0.1:8000/?canisterId=ryjl3-tyaaa-aaaaa-aaaba-
cai&id=rrkah-fqaaa-aaaaa-aaaaq-cai
```

Open up http://127.0.0.1:8000/?canisterId=ryjl3-tyaaa-aaaaa-aaaba-cai&id=rrkah-fqaaa-aaaaa-aaaaq-cai to access the web UI.

@dfinity/agent

@dfinity/agent is the TypeScript/JavaScript client library for interacting with canisters on the IC. If you are building a client web application, this is probably what you'll want to use.

There are other agents for other languages as well:

- Java
- Python
- Rust

Deploying to mainnet

Assuming you are setup with cycles, 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 canister_name
```

Common deployment issues

If you run into an error during deployment, try the following:

1. Ensure that you have followed the instructions correctly in the installation chapter, especially noting the build dependencies
2. Start the whole deployment process from scratch by running the following commands: `dfx stop` or simply terminate `dfx` in your terminal, `dfx start --clean`, `npx azle clean`, `dfx deploy`
3. Look for more error output by adding the `--verbose` flag to the `build` command in your `dfx.json` file like so: `"build": "npx azle build hello_world --verbose`
4. Look for errors in each of the files in `~/.config/azle/rust/[rust_version]/logs`
5. Reach out in the Discord channel

Examples

Azle has many example projects showing nearly all Azle APIs. They can be found in the examples directory of the Azle GitHub repository.

We'll highlight a few of them and some others here:

- Query
- Update
- Primitive Types
- Stable Structures
- Cycles
- Cross Canister Calls
- Management Canister
- Outgoing HTTP Requests
- Incoming HTTP Requests
- Pre and Post Upgrade
- Timers
- Multisig Vault
- ICRC-1
- IC Chainlink Data Feeds
- Bitcoin
- ckBTC

Query Methods

TLDR

- Created with the `query` function
- Read-only
- Executed on a single node
- No consensus
- Latency on the order of ~100 milliseconds
- 5 billion Wasm instruction limit
- 4 GiB heap limit
- ~32k queries per second per canister

The most basic way to expose your canister's functionality publicly is through a query method. Here's an example of a simple query method named `getString`:

```
import { Canister, query, text } from 'azle';

export default Canister({
    getString: query([], text, () => {
        return 'This is a query method!';
    })
});
```

Query methods are defined inside of a call to `Canister` using the `query` function.

The first parameter to `query` is an array of `CandidType` objects that will be used to decode the Candid bytes of the arguments sent from the client when calling your query method.

The second parameter to `query` is a `CandidType` object used to encode the return value of your function to Candid bytes to then be sent back to the client.

The third parameter to `query` is the function that receives the decoded arguments, performs some computation, and then returns a value to be encoded. The TypeScript signature of this function (parameter and return types) will be inferred from the `CandidType` arguments in the first and second parameters to `query`.

`getString` can be called from the outside world through the IC's HTTP API. You'll usually invoke this API from the `dfx command line`, `dfx web UI`, or an agent.

From the `dfx command line` you can call it like this:

```
dfx canister call my_canister getString
```

Query methods are read-only. They do not persist any state changes. Take a look at the following example:

```
import { Canister, query, text, Void } from 'azle';

let db: {
    [key: string]: string;
} = {};

export default Canister({
    set: query([text, text], Void, (key, value) => {
        db[key] = value;
    })
});
```

Calling `set` will perform the operation of setting the `key` property on the `db` object to `value`, but after the call finishes that change will be discarded.

This is because query methods are executed on a single node machine and do not go through consensus. This results in lower latencies, perhaps on the order of 100 milliseconds.

There is a limit to how much computation can be done in a single call to a query method. The current query call limit is 5 billion Wasm instructions. Here's an example of a query method that runs the risk of reaching the limit:

```
import { Canister, nat32, query, text } from 'azle';

export default Canister({
    pyramid: query([nat32], text, (levels) => {
        return new Array(levels).fill(0).reduce((acc, _, index) => {
            const asterisks = new Array(index + 1).fill('*').join('');
            return `${acc}${asterisks}\n`;
        }, '');
    })
});
```

From the `dfx command line` you can call `pyramid` like this:

```
dfx canister call my_canister pyramid '(1_000)'
```

With an argument of `1_000`, `pyramid` will fail with an error `...exceeded the instruction limit for single message execution`.

Keep in mind that each query method invocation has up to 4 GiB of heap available.

In terms of query scalability, an individual canister likely has an upper bound of ~36k queries per second.

Update Methods

TLDR

- Created with the `update` function
- Read-write
- Executed on many nodes
- Consensus
- Latency ~2-5 seconds
- 20 billion Wasm instruction limit
- 4 GiB heap limit
- 96 GiB stable memory limit
- ~900 updates per second per canister

Update methods are similar to query methods, but state changes can be persisted. Here's an example of a simple update method:

```
import { Canister, nat64, update } from 'azle';

let counter = 0n;

export default Canister({
    increment: update([], nat64, () => {
        return counter++;
    })
});
```

Calling `increment` will return the current value of `counter` and then increase its value by 1. Because `counter` is a global variable, the change will be persisted to the heap, and subsequent query and update calls will have access to the new `counter` value.

Because the Internet Computer (IC) persists changes with certain fault tolerance guarantees, update calls are executed on many nodes and go through consensus. This leads to latencies of ~2-5 seconds per update call.

Due to the latency and other expenses involved with update methods, it is best to use them only when necessary. Look at the following example:

```
import { Canister, query, text, update, Void } from 'azle';

let message = '';

export default Canister({
    getMessage: query([], text, () => {
        return message;
    }),
    setMessage: update([text], Void, (newMessage) => {
        message = newMessage;
    })
});
```

You'll notice that we use an update method, `setMessage`, only to perform the change to the global `message` variable. We use `getMessage`, a query method, to read the message.

Keep in mind that the heap is limited to 4 GiB, and thus there is an upper bound to global variable storage capacity. You can imagine how a simple database like the following would eventually run out of memory with too many entries:

```
import { Canister, None, Opt, query, Some, text, update, Void } from 'azle';

type Db = {
    [key: string]: string;
};

let db: Db = {};

export default Canister({
    get: query([text], Opt(text), (key) => {
        const value = db[key];
        return value !== undefined ? Some(value) : None;
    }),
    set: update([text, text], Void, (key, value) => {
        db[key] = value;
    })
});
```

If you need more than 4 GiB of storage, consider taking advantage of the 96 GiB of stable memory. Stable structures like `StableBTreeMap` give you a nice API for interacting with stable memory. These data structures will be covered in more detail later. Here's a simple example:

```
import { Canister, Opt, query, StableBTreeMap, text, update, Void } from
'azle';

let db = StableBTreeMap(text, text, 0);

export default Canister({
    get: query([text], Opt(text), (key) => {
        return db.get(key);
    }),
    set: update([text, text], Void, (key, value) => {
        db.insert(key, value);
    })
});
```

So far we have only seen how state changes can be persisted. State changes can also be discarded by implicit or explicit traps. A trap is an immediate stop to execution with the ability to provide a message to the execution environment.

Traps can be useful for ensuring that multiple operations are either all completed or all disregarded, or in other words atomic. Keep in mind that these guarantees do not hold once cross-canister calls are introduced, but that's a more advanced topic covered later.

Here's an example of how to trap and ensure atomic changes to your database:

```
import {
    Canister,
    ic,
    Opt,
    query,
    Record,
    StableBTreeMap,
    text,
    update,
    Vec,
    Void
} from 'azle';

const Entry = Record({
    key: text,
    value: text
});

let db = StableBTreeMap(text, text, 0);

export default Canister({
    get: query([text], Opt(text), (key) => {
        return db.get(key);
    }),
    set: update([text, text], Void, (key, value) => {
        db.insert(key, value);
    }),
    setMany: update([Vec(Entry)], Void, (entries) => {
        entries.forEach((entry) => {
            if (entry.key === 'trap') {
                ic.trap('explicit trap');
            }

            db.insert(entry.key, entry.value);
        });
    })
});
```

In addition to `ic.trap`, an explicit JavaScript `throw` or any unhandled exception will also trap.

There is a limit to how much computation can be done in a single call to an update method. The current update call limit is 20 billion Wasm instructions. If we modify our database example, we can introduce an update method that runs the risk of reaching the limit:

```
import {
    Canister,
    nat64,
    Opt,
    query,
    StableBTreeMap,
    text,
    update,
    Void
} from 'azle';

let db = StableBTreeMap(text, text, 0);

export default Canister({
    get: query([text], Opt(text), (key) => {
        return db.get(key);
    }),
    set: update([text, text], Void, (key, value) => {
        db.insert(key, value);
    }),
    setMany: update([nat64], Void, (numEntries) => {
        for (let i = 0; i < numEntries; i++) {
            db.insert(i.toString(), i.toString());
        }
    })
});
```

From the `dfx command line` you can call `setMany` like this:

```
dfx canister call my_canister setMany '(10_000)'
```

With an argument of `10_000`, `setMany` will fail with an error `...exceeded the instruction limit for single message execution`.

In terms of update scalability, an individual canister likely has an upper bound of ~900 updates per second.

Candid

- text
- blob
- nat
- nat8
- nat16
- nat32
- nat64
- int
- int8
- int16
- int32
- int64
- float32
- float64
- bool
- null
- vec
- opt
- record
- variant
- func
- service
- principal
- reserved
- empty

Candid is an interface description language created by DFINITY. It can be used to define interfaces between services (canisters), allowing canisters and clients written in various languages to easily interact with each other. This interaction occurs through the serialization/encoding and deserialization/decoding of runtime values to and from Candid bytes.

Azle performs automatic encoding and decoding of JavaScript values to and from Candid bytes through the use of various `CandidType` objects. For example, `CandidType` objects are used when defining the parameter and return types of your query and update methods. They are also used to define the keys and values of a `StableBTreeMap`.

It's important to note that the `CandidType` objects decode Candid bytes into specific JavaScript runtime data structures that may differ in behavior from the description of the actual Candid type. For example, a `float32` Candid type is a JavaScript Number, a `nat64` is a JavaScript BigInt, and an `int` is also a JavaScript BigInt.

Keep this in mind as it may result in unexpected behavior. Each `CandidType` object and its equivalent JavaScript runtime value is explained in more detail in The Azle Book Candid reference.

A more canonical reference of all Candid types available on the Internet Computer (IC) can be found here.

The following is a simple example showing how to import and use many of the `CandidType` objects available in Azle:

```
import {
    blob,
    bool,
    Canister,
    float32,
    float64,
    Func,
    int,
    int16,
    int32,
    int64,
    int8,
    nat,
    nat16,
    nat32,
    nat64,
    nat8,
    None,
    Null,
    Opt,
    Principal,
    query,
    Record,
    Recursive,
    text,
    update,
    Variant,
    Vec
} from 'azle';

const MyCanister = Canister({
    query: query([], bool),
    update: update([], text)
});

const Candid = Record({
    text: text,
    blob: blob,
    nat: nat,
    nat64: nat64,
    nat32: nat32,
    nat16: nat16,
    nat8: nat8,
    int: int,
    int64: int64,
    int32: int32,
    int16: int16,
    int8: int8,
    float64: float64,
    float32: float32,
    bool: bool,
    null: Null,
    vec: Vec(text),
    opt: Opt(nat),
    record: Record({
        firstName: text,
        lastName: text,
        age: nat8
    }),
    variant: Variant({
        Tag1: Null,
        Tag2: Null,
        Tag3: int
    }),
    func: Recursive(() => Func([], Candid, 'query')),
    canister: Canister({
        query: query([], bool),
        update: update([], text)
    }),
    principal: Principal
});

export default Canister({
    candidTypes: query([], Candid, () => {
        return {
            text: 'text',
            blob: Uint8Array.from([]),
            nat: 340_282_366_920_938_463_463_374_607_431_768_211_455n,
            nat64: 18_446_744_073_709_551_615n,
            nat32: 4_294_967_295,
            nat16: 65_535,
            nat8: 255,
            int: 170_141_183_460_469_231_731_687_303_715_884_105_727n,
            int64: 9_223_372_036_854_775_807n,
            int32: 2_147_483_647,
            int16: 32_767,
            int8: 127,
            float64: Math.E,
            float32: Math.PI,
            bool: true,
            null: null,
            vec: ['has one element'],
            opt: None,
            record: {
                firstName: 'John',
                lastName: 'Doe',
                age: 35
            },
            variant: {
                Tag1: null
            },
            func: [
                Principal.fromText('rrkah-fqaaa-aaaaa-aaaaq-cai'),
                'candidTypes'
            ],
            canister: MyCanister(Principal.fromText('aaaaa-aa')),
            principal: Principal.fromText('ryjl3-tyaaa-aaaaa-aaaba-cai')
        };
    })
});
```

Calling `candidTypes` with `dfx` will return:

```
(
  record {
    func = func "rrkah-fqaaa-aaaaa-aaaaq-cai".candidTypes;
    text = "text";
    nat16 = 65_535 : nat16;
    nat32 = 4_294_967_295 : nat32;
    nat64 = 18_446_744_073_709_551_615 : nat64;
    record = record { age = 35 : nat8; lastName = "Doe"; firstName = "John"
};
    int = 170_141_183_460_469_231_731_687_303_715_884_105_727 : int;
    nat = 340_282_366_920_938_463_463_374_607_431_768_211_455 : nat;
    opt = null;
    vec = vec { "has one element" };
    variant = variant { Tag1 };
    nat8 = 255 : nat8;
    canister = service "aaaaa-aa";
    int16 = 32_767 : int16;
    int32 = 2_147_483_647 : int32;
    int64 = 9_223_372_036_854_775_807 : int64;
    null = null : null;
    blob = vec {};
    bool = true;
    principal = principal "ryjl3-tyaaa-aaaaa-aaaba-cai";
    int8 = 127 : int8;
    float32 = 3.1415927 : float32;
    float64 = 2.718281828459045 : float64;
  },
)
```

Stable Structures

TLDR

- 96 GiB of stable memory
- Persistent across upgrades
- Familiar API
- Must specify memory id
- No migrations per memory id

Stable structures are data structures with familiar APIs that allow access to stable memory. Stable memory is a separate memory location from the heap that currently allows up to 96 GiB of storage. Stable memory persists automatically across upgrades.

Persistence on the Internet Computer (IC) is very important to understand. When a canister is upgraded (its code is changed after being initially deployed) its heap is wiped. This includes all global variables.

On the other hand, anything stored in stable memory will be preserved. Writing and reading to and from stable memory can be done with a low-level API, but it is generally easier and preferable to use stable structures.

Azle currently provides one stable structure called `StableBTreeMap`. It's similar to a JavaScript Map and has most of the common operations you'd expect such as reading, inserting, and removing values.

Here's how to define a simple `StableBTreeMap`:

```
import { nat8, StableBTreeMap, text } from 'azle';

let map = StableBTreeMap(nat8, text, 0);
```

This is a `StableBTreeMap` with a key of type `nat8` and a value of type `text`. Key and value types can be any Candid type.

This `StableBTreeMap` also has a `memory id` of `0`. Each `StableBTreeMap` instance must have a unique `memory id` between `0` and `254`. Once a `memory id` is allocated, it cannot be used with a different `StableBTreeMap`. This means you can't create another `StableBTreeMap` using the same `memory id`, and you can't change the key or value types of an existing `StableBTreeMap`. This problem will be addressed to some extent.

Here's an example showing all of the basic `StableBTreeMap` operations:

```
import {
    bool,
    Canister,
    nat64,
    nat8,
    Opt,
    query,
    StableBTreeMap,
    text,
    Tuple,
    update,
    Vec
} from 'azle';

const Key = nat8;
const Value = text;

let map = StableBTreeMap(Key, Value, 0);

export default Canister({
    containsKey: query([Key], bool, (key) => {
        return map.containsKey(key);
    }),

    get: query([Key], Opt(Value), (key) => {
        return map.get(key);
    }),

    insert: update([Key, Value], Opt(Value), (key, value) => {
        return map.insert(key, value);
    }),

    isEmpty: query([], bool, () => {
        return map.isEmpty();
    }),

    items: query([], Vec(Tuple(Key, Value)), () => {
        return map.items();
    }),

    keys: query([], Vec(Key), () => {
        return map.keys();
    }),

    len: query([], nat64, () => {
        return map.len();
    }),

    remove: update([Key], Opt(Value), (key) => {
        return map.remove(key);
    }),

    values: query([], Vec(Value), () => {
        return map.values();
    })
});
```

With these basic operations you can build more complex CRUD database applications:

```
import {
    blob,
    Canister,
    ic,
    Err,
    nat64,
    Ok,
    Opt,
    Principal,
    query,
    Record,
    Result,
    StableBTreeMap,
    text,
    update,
    Variant,
    Vec
} from 'azle';

const User = Record({
    id: Principal,
    createdAt: nat64,
    recordingIds: Vec(Principal),
    username: text
});

const Recording = Record({
    id: Principal,
    audio: blob,
    createdAt: nat64,
    name: text,
    userId: Principal
});

const AudioRecorderError = Variant({
    RecordingDoesNotExist: Principal,
    UserDoesNotExist: Principal
});

let users = StableBTreeMap(Principal, User, 0);
let recordings = StableBTreeMap(Principal, Recording, 1);

export default Canister({
    createUser: update([text], User, (username) => {
        const id = generateId();
        const user: typeof User = {
            id,
            createdAt: ic.time(),
            recordingIds: [],
            username
        };

        users.insert(user.id, user);

        return user;
    }),
    readUsers: query([], Vec(User), () => {
        return users.values();
    }),
    readUserById: query([Principal], Opt(User), (id) => {
        return users.get(id);
    }),
    deleteUser: update([Principal], Result(User, AudioRecorderError), (id) =>
{
        const userOpt = users.get(id);

        if ('None' in userOpt) {
            return Err({
                UserDoesNotExist: id
            });
        }

        const user = userOpt.Some;

        user.recordingIds.forEach((recordingId) => {
            recordings.remove(recordingId);
        });

        users.remove(user.id);

        return Ok(user);
    }),
    createRecording: update(
        [blob, text, Principal],
        Result(Recording, AudioRecorderError),
        (audio, name, userId) => {
            const userOpt = users.get(userId);

            if ('None' in userOpt) {
                return Err({
                    UserDoesNotExist: userId
                });
            }

            const user = userOpt.Some;

            const id = generateId();
            const recording: typeof Recording = {
                id,
                audio,
                createdAt: ic.time(),
                name,
                userId
            };

            recordings.insert(recording.id, recording);

            const updatedUser: typeof User = {
                ...user,
                recordingIds: [...user.recordingIds, recording.id]
            };

            users.insert(updatedUser.id, updatedUser);
```

```
                return Ok(recording);
            }
        ),
        readRecordings: query([], Vec(Recording), () => {
            return recordings.values();
        }),
        readRecordingById: query([Principal], Opt(Recording), (id) => {
            return recordings.get(id);
        }),
        deleteRecording: update(
            [Principal],
            Result(Recording, AudioRecorderError),
            (id) => {
                const recordingOpt = recordings.get(id);

                if ('None' in recordingOpt) {
                    return Err({ RecordingDoesNotExist: id });
                }

                const recording = recordingOpt.Some;

                const userOpt = users.get(recording.userId);

                if ('None' in userOpt) {
                    return Err({
                        UserDoesNotExist: recording.userId
                    });
                }

                const user = userOpt.Some;

                const updatedUser: typeof User = {
                    ...user,
                    recordingIds: user.recordingIds.filter(
                        (recordingId) =>
                            recordingId.toText() !== recording.id.toText()
                    )
                };

                users.insert(updatedUser.id, updatedUser);

                recordings.remove(id);

                return Ok(recording);
            }
        )
    });

    function generateId(): Principal {
        const randomBytes = new Array(29)
            .fill(0)
            .map((_) => Math.floor(Math.random() * 256));

        return Principal.fromUint8Array(Uint8Array.from(randomBytes));
    }
```

The example above shows a very basic audio recording backend application. There are two types of entities that need to be stored, `User` and `Recording`. These are represented as `Candid` records.

Each entity gets its own `StableBTreeMap`:

```
import {
    blob,
    nat64,
    Principal,
    Record,
    StableBTreeMap,
    text,
    Vec
} from 'azle';

const User = Record({
    id: Principal,
    createdAt: nat64,
    recordingIds: Vec(Principal),
    username: text
});

const Recording = Record({
    id: Principal,
    audio: blob,
    createdAt: nat64,
    name: text,
    userId: Principal
});

let users = StableBTreeMap(Principal, User, 0);
let recordings = StableBTreeMap(Principal, Recording, 1);
```

Notice that each `StableBTreeMap` has a unique `memory id`.

Caveats

Performance

Azle's `StableBTreeMap` uses Candid encoding and decoding to store and retrieve all values. Azle's Candid encoding/decoding implementation is currently not well optimized, and Candid may not be the most optimal encoding format overall, so you may experience heavy instruction usage when performing many `StableBTreeMap` operations in succession. A rough idea of the overhead from our preliminary testing is probably 1-2 million instructions for a full Candid encoding and decoding of values per `StableBTreeMap` operation.

Migrations

Migrations must be performed manually by reading the values out of one `StableBTreeMap` and writing them into another. Once a `StableBTreeMap` is initialized to a specific `memory id`, that `memory id` cannot be changed unless the canister is completely wiped and initialized again.

Cross-canister

Examples:

- async_await
- bitcoin
- composite_queries
- cross_canister_calls
- cycles
- ethereum_json_rpc
- func_types
- heartbeat
- inline_types
- ledger_canister
- management_canister
- outgoing_http_requests
- threshold_ecdsa
- rejections
- timers
- tuple_types
- whoami

Canisters are generally able to call the query or update methods of other canisters in any subnet. We refer to these types of calls as cross-canister calls.

A cross-canister call begins with a definition of the canister to be called.

Imagine a simple canister called `token_canister`:

```
import {
    Canister,
    ic,
    nat64,
    Opt,
    Principal,
    StableBTreeMap,
    update
} from 'azle';

let accounts = StableBTreeMap(Principal, nat64, 0);

export default Canister({
    transfer: update([Principal, nat64], nat64, (to, amount) => {
        const from = ic.caller();

        const fromBalance = getBalance(accounts.get(from));
        const toBalance = getBalance(accounts.get(to));

        accounts.insert(from, fromBalance - amount);
        accounts.insert(to, toBalance + amount);

        return amount;
    })
});

function getBalance(accountOpt: Opt<nat64>): nat64 {
    if ('None' in accountOpt) {
        return 0n;
    } else {
        return accountOpt.Some;
    }
}
```

Now that you have the canister definition, you can import and instantiate it in another canister:

```
import { Canister, ic, nat64, Principal, update } from 'azle';
import TokenCanister from './token_canister';

const tokenCanister = TokenCanister(
    Principal.fromText('r7inp-6aaaa-aaaaa-aaabq-cai')
);

export default Canister({
    payout: update([Principal, nat64], nat64, async (to, amount) => {
        return await ic.call(tokenCanister.transfer, {
            args: [to, amount]
        });
    })
});
```

If you don't have the actual definition of the token canister with the canister method implementations, you can always create your own canister definition without method implementations:

```
import { Canister, ic, nat64, Principal, update } from 'azle';

const TokenCanister = Canister({
    transfer: update([Principal, nat64], nat64)
});

const tokenCanister = TokenCanister(
    Principal.fromText('r7inp-6aaaa-aaaaa-aaabq-cai')
);

export default Canister({
    payout: update([Principal, nat64], nat64, async (to, amount) => {
        return await ic.call(tokenCanister.transfer, {
            args: [to, amount]
        });
    })
});
```

The IC guarantees that cross-canister calls will return. This means that, generally speaking, you will always receive a response from `ic.call`. If there are errors during the call, `ic.call` will throw. Wrapping your cross-canister call in a `try...catch` allows you to handle these errors.

Let's add to our example code and explore adding some practical error-handling to stop people from stealing tokens.

`token_canister`:

```
import {
    Canister,
    ic,
    nat64,
    Opt,
    Principal,
    StableBTreeMap,
    update
} from 'azle';

let accounts = StableBTreeMap(Principal, nat64, 0);

export default Canister({
    transfer: update([Principal, nat64], nat64, (to, amount) => {
        const from = ic.caller();

        const fromBalance = getBalance(accounts.get(from));

        if (amount > fromBalance) {
            throw new Error(`${from} has an insufficient balance`);
        }

        const toBalance = getBalance(accounts.get(to));

        accounts.insert(from, fromBalance - amount);
        accounts.insert(to, toBalance + amount);

        return amount;
    })
});

function getBalance(accountOpt: Opt<nat64>): nat64 {
    if ('None' in accountOpt) {
        return 0n;
    } else {
        return accountOpt.Some;
    }
}
```

payout_canister:

```
import { Canister, ic, nat64, Principal, update } from 'azle';
import TokenCanister from './index';

const tokenCanister = TokenCanister(
    Principal.fromText('bkyz2-fmaaa-aaaaa-qaaaq-cai')
);

export default Canister({
    payout: update([Principal, nat64], nat64, async (to, amount) => {
        try {
            return await ic.call(tokenCanister.transfer, {
                args: [to, amount]
            });
        } catch (error) {
            console.log(error);
        }

        return 0n;
    })
});
```

Throwing will allow you to express error conditions and halt execution, but you may find embracing the `Result` variant as a better solution for error handling because of its composability and predictability.

So far we have only shown a cross-canister call from an update method. Update methods can call other update methods or query methods (but not composite query methods as discussed below). If an update method calls a query method, that query method will be called in replicated mode. Replicated mode engages the consensus process, but for queries the state will still be discarded.

Cross-canister calls can also be initiated from query methods. These are known as composite queries, and in Azle they are simply `async` query methods. Composite queries can call other composite query methods and regular query methods. Composite queries cannot call update methods.

Here's an example of a composite query method:

```
import { bool, Canister, ic, Principal, query } from 'azle';

const SomeCanister = Canister({
    queryForBoolean: query([], bool)
});

const someCanister = SomeCanister(
    Principal.fromText('ryjl3-tyaaa-aaaaa-aaaba-cai')
);

export default Canister({
    querySomeCanister: query([], bool, async () => {
        return await ic.call(someCanister.queryForBoolean);
    })
});
```

You can expect cross-canister calls within the same subnet to take up to a few seconds to complete, and cross-canister calls across subnets take about double that time. Composite queries should be much faster, similar to query calls in latency.

If you don't need to wait for your cross-canister call to return, you can use `notify`:

```
import { Canister, ic, Principal, update, Void } from 'azle';

const SomeCanister = Canister({
    receiveNotification: update([], Void)
});

const someCanister = SomeCanister(
    Principal.fromText('ryjl3-tyaaa-aaaaa-aaaba-cai')
);

export default Canister({
    sendNotification: update([], Void, () => {
        return ic.notify(someCanister.receiveNotification);
    })
});
```

If you need to send cycles with your cross-canister call, you can add `cycles` to the `config` object of `ic.notify`:

```
import { Canister, ic, Principal, update, Void } from 'azle';

const SomeCanister = Canister({
    receiveNotification: update([], Void)
});

const someCanister = SomeCanister(
    Principal.fromText('ryjl3-tyaaa-aaaaa-aaaba-cai')
);

export default Canister({
    sendNotification: update([], Void, () => {
        return ic.notify(someCanister.receiveNotification, {
            cycles: 1_000_000n
        });
    })
});
```

HTTP

This chapter is a work in progress.

Incoming HTTP requests

Examples:

- http_counter

```
import {
    blob,
    bool,
    Canister,
    Func,
    nat16,
    None,
    Opt,
    query,
    Record,
    text,
    Tuple,
    Variant,
    Vec
} from 'azle';

const Token = Record({
    // add whatever fields you'd like
    arbitrary_data: text
});

const StreamingCallbackHttpResponse = Record({
    body: blob,
    token: Opt(Token)
});

export const Callback = Func([text], StreamingCallbackHttpResponse, 'query');

const CallbackStrategy = Record({
    callback: Callback,
    token: Token
});

const StreamingStrategy = Variant({
    Callback: CallbackStrategy
});

type HeaderField = [text, text];
const HeaderField = Tuple(text, text);

const HttpResponse = Record({
    status_code: nat16,
    headers: Vec(HeaderField),
    body: blob,
    streaming_strategy: Opt(StreamingStrategy),
    upgrade: Opt(bool)
});

const HttpRequest = Record({
    method: text,
    url: text,
    headers: Vec(HeaderField),
    body: blob,
    certificate_version: Opt(nat16)
});
```

```
export default Canister({
    http_request: query([HttpRequest], HttpResponse, (req) => {
        return {
            status_code: 200,
            headers: [],
            body: Buffer.from('hello'),
            streaming_strategy: None,
            upgrade: None
        };
    })
});
```

Outgoing HTTP requests

Examples:

- ethereum_json_rpc
- outgoing_http_requests

```
import {
    Canister,
    ic,
    init,
    nat32,
    Principal,
    query,
    Some,
    StableBTreeMap,
    text,
    update
} from 'azle';
import {
    HttpResponse,
    HttpTransformArgs,
    managementCanister
} from 'azle/canisters/management';

let stableStorage = StableBTreeMap(text, text, 0);

export default Canister({
    init: init([text], (ethereumUrl) => {
        stableStorage.insert('ethereumUrl', ethereumUrl);
    }),
    ethGetBalance: update([text], text, async (ethereumAddress) => {
        const urlOpt = stableStorage.get('ethereumUrl');

        if ('None' in urlOpt) {
            throw new Error('ethereumUrl is not defined');
        }

        const url = urlOpt.Some;

        const httpResponse = await ic.call(managementCanister.http_request, {
            args: [
                {
                    url,
                    max_response_bytes: Some(2_000n),
                    method: {
                        post: null
                    },
                    headers: [],
                    body: Some(
                        Buffer.from(
                            JSON.stringify({
                                jsonrpc: '2.0',
                                method: 'eth_getBalance',
                                params: [ethereumAddress, 'earliest'],
                                id: 1
                            }),
                            'utf-8'
                        )
                    ),
                    transform: Some({
                        function: [ic.id(), 'ethTransform'] as [
                            Principal,
                            string
                        ],
                        context: Uint8Array.from([])
                    })
                }
            ],
            cycles: 50_000_000n
        });

        return Buffer.from(httpResponse.body.buffer).toString('utf-8');
    }),
    ethGetBlockByNumber: update([nat32], text, async (number) => {
        const urlOpt = stableStorage.get('ethereumUrl');

        if ('None' in urlOpt) {
            throw new Error('ethereumUrl is not defined');
        }

        const url = urlOpt.Some;

        const httpResponse = await ic.call(managementCanister.http_request, {
            args: [
                {
                    url,
                    max_response_bytes: Some(2_000n),
                    method: {
                        post: null
                    },
                    headers: [],
                    body: Some(
                        Buffer.from(
                            JSON.stringify({
                                jsonrpc: '2.0',
                                method: 'eth_getBlockByNumber',
                                params: [`0x${number.toString(16)}`, false],
                                id: 1
                            }),
                            'utf-8'
                        )
                    ),
                    transform: Some({
                        function: [ic.id(), 'ethTransform'] as [
                            Principal,
                            string
                        ],
                        context: Uint8Array.from([])
                    })
                }
            ],
            cycles: 50_000_000n
        });

        return Buffer.from(httpResponse.body.buffer).toString('utf-8');
    }),
    ethTransform: query([HttpTransformArgs], HttpResponse, (args) => {
        return {
            ...args.response,
            headers: []
        };
    })
});
```

Management Canister

This chapter is a work in progress.

You can access the management canister like this:

```
import { blob, Canister, ic, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

export default Canister({
    randomBytes: update([], blob, async () => {
        return await ic.call(managementCanister.raw_rand);
    })
});
```

See the management canister reference section for more information.

Canister Lifecycle

This chapter is a work in progress.

```
import { Canister, init, postUpgrade, preUpgrade } from 'azle';

export default Canister({
    init: init([], () => {
        console.log('runs on first canister install');
    }),
    preUpgrade: preUpgrade(() => {
        console.log('runs before canister upgrade');
    }),
    postUpgrade: postUpgrade([], () => {
        console.log('runs after canister upgrade');
    })
});
```

Timers

This chapter is a work in progress.

```
import {
    blob,
    bool,
    Canister,
    Duration,
    ic,
    int8,
    query,
    Record,
    text,
    TimerId,
    update,
    Void
} from 'azle';
import { managementCanister } from 'azle/canisters/management';

const StatusReport = Record({
    single: bool,
    inline: int8,
    capture: text,
    repeat: int8,
    singleCrossCanister: blob,
    repeatCrossCanister: blob
});

const TimerIds = Record({
    single: TimerId,
    inline: TimerId,
    capture: TimerId,
    repeat: TimerId,
    singleCrossCanister: TimerId,
    repeatCrossCanister: TimerId
});

let statusReport: typeof StatusReport = {
    single: false,
    inline: 0,
    capture: '',
    repeat: 0,
    singleCrossCanister: Uint8Array.from([]),
    repeatCrossCanister: Uint8Array.from([])
};

export default Canister({
    clearTimer: update([TimerId], Void, (timerId) => {
        ic.clearTimer(timerId);
        console.log(`timer ${timerId} cancelled`);
    }),
    setTimers: update([Duration, Duration], TimerIds, (delay, interval) => {
        const capturedValue = '🚩';

        const singleId = ic.setTimer(delay, oneTimeTimerCallback);

        const inlineId = ic.setTimer(delay, () => {
            statusReport.inline = 1;
            console.log('Inline timer called');
        });

        const captureId = ic.setTimer(delay, () => {
            statusReport.capture = capturedValue;
            console.log(`Timer captured value ${capturedValue}`);
        });

        const repeatId = ic.setTimerInterval(interval, () => {
            statusReport.repeat++;
            console.log(`Repeating timer. Call ${statusReport.repeat}`);
        });

        const singleCrossCanisterId = ic.setTimer(
            delay,
            singleCrossCanisterTimerCallback
        );

        const repeatCrossCanisterId = ic.setTimerInterval(
            interval,
            repeatCrossCanisterTimerCallback
        );

        return {
            single: singleId,
            inline: inlineId,
            capture: captureId,
            repeat: repeatId,
            singleCrossCanister: singleCrossCanisterId,
            repeatCrossCanister: repeatCrossCanisterId
        };
    }),
    statusReport: query([], StatusReport, () => {
        return statusReport;
    })
});

function oneTimeTimerCallback() {
    statusReport.single = true;
    console.log('oneTimeTimerCallback called');
}

async function singleCrossCanisterTimerCallback() {
    console.log('singleCrossCanisterTimerCallback');

    statusReport.singleCrossCanister = await ic.call(
        managementCanister.raw_rand
    );
}

async function repeatCrossCanisterTimerCallback() {
    console.log('repeatCrossCanisterTimerCallback');

    statusReport.repeatCrossCanister = Uint8Array.from([
        ...statusReport.repeatCrossCanister,
        ...(await ic.call(managementCanister.raw_rand))
    ]);
}
```

Cycles

This chapter is a work in progress.

Cycles are essentially units of computational resources such as bandwidth, memory, and CPU instructions. Costs are generally metered on the Internet Computer (IC) by cycles. You can see a breakdown of all cycle costs here.

Currently queries do not have any cycle costs.

Most important to you will probably be update costs.

TODO break down some cycle scenarios maybe? Perhaps we should show some of our analyses for different types of applications. Maybe show how to send and receive cycles, exactly how to do it.

Show all of the APIs for sending or receiving cycles?

Perhaps we don't need to do that here, since each API will show this information.

Maybe here we just show the basic concept of cycles, link to the main cycles cost page, and show a few examples of how to break down these costs or estimate these costs.

Caveats

Unknown security vulnerabilities

Azle is a beta project. See the disclaimer for more information.

npm packages

Some npm packages will work and some will not work. It is our long-term goal to support as many npm packages as possible. There are various reasons why an npm package may not currently work, including the small Wasm binary limit of the IC and unimplemented web or Node.js APIs. Feel free to open issues if your npm package does not work in Azle.

JavaScript environment APIs

You may encounter various missing JavaScript environment APIs, such as those you would expect in the web or Node.js environments.

High Candid encoding/decoding costs

Candid encoding/decoding is currently very unoptimized. This will most likely lead to a ~1-2 million instruction extra fixed cost for all calls, plus more if you use `StableBTreeMap` or any other API or data structure that engages in Candid encoding/decoding.

Promises

Though promises are implemented, the underlying queue that handles asynchronous operations is very simple. This queue will not behave exactly as queues from the major JS engines.

Reference

Bitcoin

The Internet Computer (IC) interacts with the Bitcoin blockchain through the use of `tECDSA`, the `Bitcoin integration`, and a ledger canister called `ckBTC`.

tECDSA

`tECDSA` on the IC allows canisters to request access to threshold ECDSA keypairs on the `tECDSA` subnet. This functionality is exposed through two management canister methods:

- ecdsa_public_key
- sign_with_ecdsa

The following are examples using `tECDSA`:

- basic_bitcoin
- threshold_ecdsa

Bitcoin integration

The `Bitcoin integration` allows canisters on the IC to interact directly with the Bitcoin network. This functionality is exposed through the following management canister methods:

- bitcoin_get_balance
- bitcoin_get_current_fee_percentiles
- bitcoin_get_utxos
- bitcoin_send_transaction

The following are examples using the `Bitcoin integration`:

- basic_bitcoin
- bitcoin

ckBTC

`ckBTC` is a ledger canister deployed to the IC. It follows the `ICRC` standard, and can be accessed easily from an Azle canister using `azle/canisters/ICRC` if you only need the

`ICRC` methods. For access to the full ledger methods you will need to create your own Service for now.

The following are examples using `ckBTC`:

- ckBTC

Call APIs

- accept message
- arg data raw
- arg data raw size
- call
- call raw
- call raw 128
- call with payment
- call with payment 128
- caller
- method name
- msg cycles accept
- msg cycles accept 128
- msg cycles available
- msg cycles available 128
- msg cycles refunded
- msg cycles refunded 128
- notify
- notify raw
- notify with payment 128
- reject
- reject code
- reject message
- reply
- reply raw

accept message

This section is a work in progress.

Examples:

- inspect_message
- run_time_errors

```
import { Canister, ic, inspectMessage } from 'azle';

export default Canister({
    inspectMessage: inspectMessage(() => {
        ic.acceptMessage();
    })
});
```

arg data raw

This section is a work in progress.

Examples:

- ic_api

```
import { blob, bool, Canister, ic, int8, query, text } from 'azle';

export default Canister({
    // returns the argument data as bytes.
    argDataRaw: query(
        [blob, int8, bool, text],
        blob,
        (arg1, arg2, arg3, arg4) => {
            return ic.argDataRaw();
        }
    )
});
```

arg data raw size

This section is a work in progress.

Examples:

- ic_api

```
import { blob, bool, Canister, ic, int8, nat, query, text } from 'azle';

export default Canister({
    // returns the length of the argument data in bytes
    argDataRawSize: query(
        [blob, int8, bool, text],
        nat,
        (arg1, arg2, arg3, arg4) => {
            return ic.argDataRawSize();
        }
    )
});
```

call

This section is a work in progress.

Examples:

- async_await
- bitcoin
- composite_queries
- cross_canister_calls
- cycles
- ethereum_json_rpc
- func_types
- heartbeat
- inline_types
- ledger_canister
- management_canister
- outgoing_http_requests
- threshold_ecdsa
- rejections
- timers
- tuple_types
- whoami

```
import { Canister, ic, init, nat64, Principal, update } from 'azle';

const TokenCanister = Canister({
    transfer: update([Principal, nat64], nat64)
});

let tokenCanister: typeof TokenCanister;

export default Canister({
    init: init([], setup),
    postDeploy: init([], setup),
    payout: update([Principal, nat64], nat64, async (to, amount) => {
        return await ic.call(tokenCanister.transfer, {
            args: [to, amount]
        });
    })
});

function setup() {
    tokenCanister = TokenCanister(
        Principal.fromText('r7inp-6aaaa-aaaaa-aaabq-cai')
    );
}
```

call raw

This section is a work in progress.

Examples:

- call_raw
- outgoing_http_requests

```
import { Canister, ic, nat64, Principal, text, update } from 'azle';

export default Canister({
    executeCallRaw: update(
        [Principal, text, text, nat64],
        text,
        async (canisterId, method, candidArgs, payment) => {
            const candidBytes = await ic.callRaw(
                canisterId,
                method,
                ic.candidEncode(candidArgs),
                payment
            );

            return ic.candidDecode(candidBytes);
        }
    )
});
```

call raw 128

This section is a work in progress.

Examples:

- call_raw

```
import { Canister, ic, nat, Principal, text, update } from 'azle';

export default Canister({
    executeCallRaw128: update(
        [Principal, text, text, nat],
        text,
        async (canisterId, method, candidArgs, payment) => {
            const candidBytes = await ic.callRaw128(
                canisterId,
                method,
                ic.candidEncode(candidArgs),
                payment
            );

            return ic.candidDecode(candidBytes);
        }
    )
});
```

call with payment

This section is a work in progress.

Examples:

- bitcoin
- cycles
- ethereum_json_rpc
- management_canister
- outgoing_http_requests
- threshold_ecdsa

```
import { blob, Canister, ic, Principal, update, Void } from 'azle';
import { managementCanister } from 'azle/canisters/management';

export default Canister({
    executeInstallCode: update(
        [Principal, blob],
        Void,
        async (canisterId, wasmModule) => {
            return await ic.call(managementCanister.install_code, {
                args: [
                    {
                        mode: { install: null },
                        canister_id: canisterId,
                        wasm_module: wasmModule,
                        arg: Uint8Array.from([])
                    }
                ],
                cycles: 100_000_000_000n
            });
        }
    )
});
```

call with payment 128

This section is a work in progress.

Examples:

- cycles

```
import { blob, Canister, ic, Principal, update, Void } from 'azle';
import { managementCanister } from 'azle/canisters/management';

export default Canister({
    executeInstallCode: update(
        [Principal, blob],
        Void,
        async (canisterId, wasmModule) => {
            return await ic.call128(managementCanister.install_code, {
                args: [
                    {
                        mode: { install: null },
                        canister_id: canisterId,
                        wasm_module: wasmModule,
                        arg: Uint8Array.from([])
                    }
                ],
                cycles: 100_000_000_000n
            });
        }
    )
});
```

caller

This section is a work in progress.

Examples:

- ic_api
- threshold_ecdsa
- whoami

```
import { Canister, ic, Principal, update } from 'azle';

export default Canister({
    // returns the principal of the identity that called this function
    caller: update([], Principal, () => {
        return ic.caller();
    })
});
```

method name

This section is a work in progress.

Examples:

- inspect_message
- run_time_errors

```
import { bool, Canister, ic, inspectMessage, update } from 'azle';

export default Canister({
    inspectMessage: inspectMessage(() => {
        console.log('inspectMessage called');

        if (ic.methodName() === 'accessible') {
            ic.acceptMessage();
            return;
        }

        if (ic.methodName() === 'inaccessible') {
            return;
        }

        throw `Method "${ic.methodName()}" not allowed`;
    }),
    accessible: update([], bool, () => {
        return true;
    }),
    inaccessible: update([], bool, () => {
        return false;
    }),
    alsoInaccessible: update([], bool, () => {
        return false;
    })
});
```

msg cycles accept

This section is a work in progress.

Examples:

- cycles

```
import { Canister, ic, nat64, update } from 'azle';

export default Canister({
    // Moves all transferred cycles to the canister
    receiveCycles: update([], nat64, () => {
        return ic.msgCyclesAccept(ic.msgCyclesAvailable() / 2n);
    })
});
```

msg cycles accept 128

This section is a work in progress.

Examples:

- cycles

```
import { Canister, ic, nat64, update } from 'azle';

export default Canister({
    // Moves all transferred cycles to the canister
    receiveCycles128: update([], nat64, () => {
        return ic.msgCyclesAccept128(ic.msgCyclesAvailable128() / 2n);
    })
});
```

msg cycles available

This section is a work in progress.

Examples:

- cycles

```
import { Canister, ic, nat64, update } from 'azle';

export default Canister({
    // Moves all transferred cycles to the canister
    receiveCycles: update([], nat64, () => {
        return ic.msgCyclesAccept(ic.msgCyclesAvailable() / 2n);
    })
});
```

msg cycles available 128

This section is a work in progress.

Examples:

- cycles

```
import { Canister, ic, nat64, update } from 'azle';

export default Canister({
    // Moves all transferred cycles to the canister
    receiveCycles128: update([], nat64, () => {
        return ic.msgCyclesAccept128(ic.msgCyclesAvailable128() / 2n);
    })
});
```

msg cycles refunded

This section is a work in progress.

Examples:

- cycles

```
import { Canister, ic, nat64, update } from 'azle';
import { otherCanister } from './other_canister';

export default Canister({
    // Reports the number of cycles returned from the Cycles canister
    sendCycles: update([], nat64, async () => {
        await ic.call(otherCanister.receiveCycles, {
            cycles: 1_000_000n
        });

        return ic.msgCyclesRefunded();
    })
});
```

msg cycles refunded 128

This section is a work in progress.

Examples:

- cycles

```
import { Canister, ic, nat64, update } from 'azle';
import { otherCanister } from './other_canister';

export default Canister({
    // Reports the number of cycles returned from the Cycles canister
    sendCycles128: update([], nat64, async () => {
        await ic.call128(otherCanister.receiveCycles128, {
            cycles: 1_000_000n
        });

        return ic.msgCyclesRefunded128();
    })
});
```

notify

This section is a work in progress.

Examples:

- cross_canister_calls
- cycles

```
import { Canister, ic, update, Void } from 'azle';
import { otherCanister } from './otherCanister';

export default Canister({
    sendNotification: update([], Void, () => {
        return ic.notify(otherCanister.receiveNotification, {
            args: ['This is the notification']
        });
    })
});
```

notify raw

This section is a work in progress.

Examples:

- notify_raw

```
import { Canister, ic, Principal, update, Void } from 'azle';

export default Canister({
    sendNotification: update([], Void, () => {
        return ic.notifyRaw(
            Principal.fromText('ryjl3-tyaaa-aaaaa-aaaba-cai'),
            'receiveNotification',
            Uint8Array.from(ic.candidEncode('()')),
            0n
        );
    })
});
```

notify with payment 128

This section is a work in progress.

Examples:

- cycles

```
import { Canister, ic, update, Void } from 'azle';
import { otherCanister } from './otherCanister';

export default Canister({
    sendCycles128Notify: update([], Void, () => {
        return ic.notify(otherCanister.receiveCycles128, {
            cycles: 1_000_000n
        });
    })
});
```

reject

This section is a work in progress.

Examples:

- ic_api
- manual_reply
- rejections

```
import { Canister, empty, ic, Manual, query, text } from 'azle';

export default Canister({
    reject: query(
        [text],
        Manual(empty),
        (message) => {
            ic.reject(message);
        },
        { manual: true }
    )
});
```

reject code

This section is a work in progress.

Examples:

- rejections

```
import { Canister, ic, RejectionCode, update } from 'azle';
import { otherCanister } from './other_canister';

export default Canister({
    getRejectionCodeDestinationInvalid: update([], RejectionCode, async () =>
{
        await ic.call(otherCanister.method);
        return ic.rejectCode();
    })
});
```

reject message

This section is a work in progress.

Examples:

- rejections

```
import { Canister, ic, text, update } from 'azle';
import { otherCanister } from './other_canister';

export default Canister({
    getRejectionMessage: update([], text, async () => {
        await ic.call(otherCanister.method);
        return ic.rejectMessage();
    })
});
```

reply

This section is a work in progress.

Examples:

- composite_queries
- manual_reply

```
import { blob, Canister, ic, Manual, update } from 'azle';

export default Canister({
    updateBlob: update(
        [],
        Manual(blob),
        () => {
            ic.reply(
                new Uint8Array([83, 117, 114, 112, 114, 105, 115, 101, 33]),
                blob
            );
        },
        { manual: true }
    )
});
```

reply raw

This section is a work in progress.

Examples:

- manual_reply
- outgoing_http_requests

```
import {
    blob,
    bool,
    Canister,
    ic,
    int,
    Manual,
    Null,
    Record,
    text,
    update,
    Variant
} from 'azle';

const Options = Variant({
    High: Null,
    Medium: Null,
    Low: Null
});

export default Canister({
    replyRaw: update(
        [],
        Manual(
            Record({
                int: int,
                text: text,
                bool: bool,
                blob: blob,
                variant: Options
            })
        ),
        () => {
            ic.replyRaw(
                ic.candidEncode(
                    '(record { "int" = 42; "text" = "text"; "bool" = true;
"blob" = blob "Surprise!"; "variant" = variant { Medium } })'
                )
            );
        },
        { manual: true }
    )
});
```

Candid

- blob
- bool
- empty
- float32
- float64
- func
- int
- int8
- int16
- int32
- int64
- nat
- nat8
- nat16
- nat32
- nat64
- null
- opt
- principal
- record
- reserved
- service
- text
- variant
- vec

blob

The `CandidType` object `blob` corresponds to the Candid type blob, is inferred to be a TypeScript `Uint8Array` and will be decoded into a JavaScript Uint8Array at runtime.

TypeScript or JavaScript:

```
import { blob, Canister, query } from 'azle';

export default Canister({
    getBlob: query([], blob, () => {
        return Uint8Array.from([68, 73, 68, 76, 0, 0]);
    }),
    printBlob: query([blob], blob, (blob) => {
        console.log(typeof blob);
        return blob;
    })
});
```

Candid:

```
service : () -> {
    getBlob : () -> (vec nat8) query;
    printBlob : (vec nat8) -> (vec nat8) query;
}
```

dfx:

```
dfx canister call candid_canister printBlob '(vec { 68; 73; 68; 76; 0; 0; })'
(blob "DIDL\00\00")

dfx canister call candid_canister printBlob '(blob "DIDL\00\00")'
(blob "DIDL\00\00")
```

bool

The `CandidType` object `bool` corresponds to the Candid type bool, is inferred to be a TypeScript `boolean`, and will be decoded into a JavaScript Boolean at runtime.

TypeScript or JavaScript:

```
import { bool, Canister, query } from 'azle';

export default Canister({
    getBool: query([], bool, () => {
        return true;
    }),
    printBool: query([bool], bool, (bool) => {
        console.log(typeof bool);
        return bool;
    })
});
```

Candid:

```
service : () -> {
    getBool : () -> (bool) query;
    printBool : (bool) -> (bool) query;
}
```

dfx:

```
dfx canister call candid_canister printBool '(true)'
(true)
```

empty

The `CandidType` object `empty` corresponds to the Candid type empty, is inferred to be a TypeScript `never`, and has no JavaScript value at runtime.

TypeScript or JavaScript:

```
import { Canister, empty, query } from 'azle';

export default Canister({
    getEmpty: query([], empty, () => {
        throw 'Anything you want';
    }),
    // Note: It is impossible to call this function because it requires an
argument
    // but there is no way to pass an "empty" value as an argument.
    printEmpty: query([empty], empty, (empty) => {
        console.log(typeof empty);
        throw 'Anything you want';
    })
});
```

Candid:

```
service : () -> {
    getEmpty : () -> (empty) query;
    printEmpty : (empty) -> (empty) query;
}
```

dfx:

```
dfx canister call candid_canister printEmpty '("You can put anything here")'
Error: Failed to create argument blob.
Caused by: Failed to create argument blob.
  Invalid data: Unable to serialize Candid values: type mismatch: "You can
put anything here" cannot be of type empty
```

float32

The `CandidType` object `float32` corresponds to the Candid type float32, is inferred to be a TypeScript `number`, and will be decoded into a JavaScript Number at runtime.

TypeScript or JavaScript:

```
import { Canister, float32, query } from 'azle';

export default Canister({
    getFloat32: query([], float32, () => {
        return Math.PI;
    }),
    printFloat32: query([float32], float32, (float32) => {
        console.log(typeof float32);
        return float32;
    })
});
```

Candid:

```
service : () -> {
    getFloat32 : () -> (float32) query;
    printFloat32 : (float32) -> (float32) query;
}
```

dfx:

```
dfx canister call candid_canister printFloat32 '(3.1415927 : float32)'
(3.1415927 : float32)
```

float64

The `CandidType` object `float64` corresponds to the Candid type float64, is inferred to be a TypeScript `number`, and will be decoded into a JavaScript Number at runtime.

TypeScript or JavaScript:

```
import { Canister, float64, query } from 'azle';

export default Canister({
    getFloat64: query([], float64, () => {
        return Math.E;
    }),
    printFloat64: query([float64], float64, (float64) => {
        console.log(typeof float64);
        return float64;
    })
});
```

Candid:

```
service : () -> {
    getFloat64 : () -> (float64) query;
    printFloat64 : (float64) -> (float64) query;
}
```

dfx:

```
dfx canister call candid_canister printFloat64 '(2.718281828459045 :
float64)'
(2.718281828459045 : float64)
```

func

Values created by the `CandidType` function `Func` correspond to the Candid type func, are inferred to be TypeScript `[Principal, string]` tuples, and will be decoded into JavaScript array with two elements at runtime.

The first element is an @dfinity/principal and the second is a JavaScript string. The `@dfinity/principal` represents the `principal` of the canister/service where the function exists, and the `string` represents the function's name.

A `func` acts as a callback, allowing the `func` receiver to know which canister instance and method must be used to call back.

TypeScript or JavaScript:

```
import { Canister, Func, Principal, query, text } from 'azle';

const BasicFunc = Func([text], text, 'query');

export default Canister({
    getBasicFunc: query([], BasicFunc, () => {
        return [
            Principal.fromText('rrkah-fqaaa-aaaaa-aaaaq-cai'),
            'getBasicFunc'
        ];
    }),
    printBasicFunc: query([BasicFunc], BasicFunc, (basicFunc) => {
        console.log(typeof basicFunc);
        return basicFunc;
    })
});
```

Candid:

```
service : () -> {
    getBasicFunc : () -> (func (text) -> (text) query) query;
    printBasicFunc : (func (text) -> (text) query) -> (
        func (text) -> (text) query,
      ) query;
}
```

dfx:

```
dfx canister call candid_canister printBasicFunc '(func "r7inp-6aaaa-aaaaa-
aaabq-cai".getBasicFunc)'
(func "r7inp-6aaaa-aaaaa-aaabq-cai".getBasicFunc)
```

int

The `CandidType` object `int` corresponds to the Candid type int, is inferred to be a TypeScript `bigint`, and will be decoded into a JavaScript BigInt at runtime.

TypeScript or JavaScript:

```
import { Canister, int, query } from 'azle';

export default Canister({
    getInt: query([], int, () => {
        return 170_141_183_460_469_231_731_687_303_715_884_105_727n;
    }),
    printInt: query([int], int, (int) => {
        console.log(typeof int);
        return int;
    })
});
```

Candid:

```
service : () -> {
    getInt : () -> (int) query;
    printInt : (int) -> (int) query;
}
```

dfx:

```
dfx canister call candid_canister printInt
'(170_141_183_460_469_231_731_687_303_715_884_105_727 : int)'
(170_141_183_460_469_231_731_687_303_715_884_105_727 : int)
```

int8

The `CandidType` object `int8` corresponds to the Candid type int8, is inferred to be a TypeScript `number`, and will be decoded into a JavaScript Number at runtime.

TypeScript or JavaScript:

```
import { Canister, int8, query } from 'azle';

export default Canister({
    getInt8: query([], int8, () => {
        return 127;
    }),
    printInt8: query([int8], int8, (int8) => {
        console.log(typeof int8);
        return int8;
    })
});
```

Candid:

```
service : () -> {
    getInt8 : () -> (int8) query;
    printInt8 : (int8) -> (int8) query;
}
```

dfx:

```
dfx canister call candid_canister printInt8 '(127 : int8)'
(127 : int8)
```

int16

The `CandidType` object `int16` corresponds to the Candid type int16, is inferred to be a TypeScript `number`, and will be decoded into a JavaScript Number at runtime.

TypeScript or JavaScript:

```
import { Canister, int16, query } from 'azle';

export default Canister({
    getInt16: query([], int16, () => {
        return 32_767;
    }),
    printInt16: query([int16], int16, (int16) => {
        console.log(typeof int16);
        return int16;
    })
});
```

Candid:

```
service : () -> {
    getInt16 : () -> (int16) query;
    printInt16 : (int16) -> (int16) query;
}
```

dfx:

```
dfx canister call candid_canister printInt16 '(32_767 : int16)'
(32_767 : int16)
```

int32

The `CandidType` object `int32` corresponds to the Candid type int32, is inferred to be a TypeScript `number`, and will be decoded into a JavaScript Number at runtime.

TypeScript or JavaScript:

```
import { Canister, int32, query } from 'azle';

export default Canister({
    getInt32: query([], int32, () => {
        return 2_147_483_647;
    }),
    printInt32: query([int32], int32, (int32) => {
        console.log(typeof int32);
        return int32;
    })
});
```

Candid:

```
service : () -> {
    getInt32 : () -> (int32) query;
    printInt32 : (int32) -> (int32) query;
}
```

dfx:

```
dfx canister call candid_canister printInt32 '(2_147_483_647 : int32)'
(2_147_483_647 : int32)
```

int64

The `CandidType` object `int64` corresponds to the Candid type int64, is inferred to be a TypeScript `bigint`, and will be decoded into a JavaScript BigInt at runtime.

TypeScript or JavaScript:

```
import { Canister, int64, query } from 'azle';

export default Canister({
    getInt64: query([], int64, () => {
        return 9_223_372_036_854_775_807n;
    }),
    printInt64: query([int64], int64, (int64) => {
        console.log(typeof int64);
        return int64;
    })
});
```

Candid:

```
service : () -> {
    getInt64 : () -> (int64) query;
    printInt64 : (int64) -> (int64) query;
}
```

dfx:

```
dfx canister call candid_canister printInt64 '(9_223_372_036_854_775_807 :
int64)'
(9_223_372_036_854_775_807 : int64)
```

nat

The `CandidType` object `nat` corresponds to the Candid type nat, is inferred to be a TypeScript `bigint`, and will be decoded into a JavaScript BigInt at runtime.

TypeScript or JavaScript:

```
import { Canister, nat, query } from 'azle';

export default Canister({
    getNat: query([], nat, () => {
        return 340_282_366_920_938_463_463_374_607_431_768_211_455n;
    }),
    printNat: query([nat], nat, (nat) => {
        console.log(typeof nat);
        return nat;
    })
});
```

Candid:

```
service : () -> {
    getNat : () -> (nat) query;
    printNat : (nat) -> (nat) query;
}
```

dfx:

```
dfx canister call candid_canister printNat
'(340_282_366_920_938_463_463_374_607_431_768_211_455 : nat)'
(340_282_366_920_938_463_463_374_607_431_768_211_455 : nat)
```

nat8

The `CandidType` object `nat8` corresponds to the Candid type nat8, is inferred to be a TypeScript `number`, and will be decoded into a JavaScript Number at runtime.

TypeScript or JavaScript:

```
import { Canister, nat8, query } from 'azle';

export default Canister({
    getNat8: query([], nat8, () => {
        return 255;
    }),
    printNat8: query([nat8], nat8, (nat8) => {
        console.log(typeof nat8);
        return nat8;
    })
});
```

Candid:

```
service : () -> {
    getNat8 : () -> (nat8) query;
    printNat8 : (nat8) -> (nat8) query;
}
```

dfx:

```
dfx canister call candid_canister printNat8 '(255 : nat8)'
(255 : nat8)
```

nat16

The `CandidType` object `nat16` corresponds to the Candid type nat16, is inferred to be a TypeScript `number`, and will be decoded into a JavaScript Number at runtime.

TypeScript or JavaScript:

```
import { Canister, nat16, query } from 'azle';

export default Canister({
    getNat16: query([], nat16, () => {
        return 65_535;
    }),
    printNat16: query([nat16], nat16, (nat16) => {
        console.log(typeof nat16);
        return nat16;
    })
});
```

Candid:

```
service : () -> {
    getNat16 : () -> (nat16) query;
    printNat16 : (nat16) -> (nat16) query;
}
```

dfx:

```
dfx canister call candid_canister printNat16 '(65_535 : nat16)'
(65_535 : nat16)
```

nat32

The `CandidType` object `nat32` corresponds to the Candid type nat32, is inferred to be a TypeScript `number`, and will be decoded into a JavaScript Number at runtime.

TypeScript or JavaScript:

```
import { Canister, nat32, query } from 'azle';

export default Canister({
    getNat32: query([], nat32, () => {
        return 4_294_967_295;
    }),
    printNat32: query([nat32], nat32, (nat32) => {
        console.log(typeof nat32);
        return nat32;
    })
});
```

Candid:

```
service : () -> {
    getNat32 : () -> (nat32) query;
    printNat32 : (nat32) -> (nat32) query;
}
```

dfx:

```
dfx canister call candid_canister printNat32 '(4_294_967_295 : nat32)'
(4_294_967_295 : nat32)
```

nat64

The `CandidType` object `nat64` corresponds to the Candid type nat64, is inferred to be a TypeScript `bigint`, and will be decoded into a JavaScript BigInt at runtime.

TypeScript or JavaScript:

```
import { Canister, nat64, query } from 'azle';

export default Canister({
    getNat64: query([], nat64, () => {
        return 18_446_744_073_709_551_615n;
    }),
    printNat64: query([nat64], nat64, (nat64) => {
        console.log(typeof nat64);
        return nat64;
    })
});
```

Candid:

```
service : () -> {
    getNat64 : () -> (nat64) query;
    printNat64 : (nat64) -> (nat64) query;
}
```

dfx:

```
dfx canister call candid_canister printNat64 '(18_446_744_073_709_551_615 :
nat64)'
(18_446_744_073_709_551_615 : nat64)
```

null

The `CandidType` object `null` corresponds to the Candid type null, is inferred to be a TypeScript `null`, and will be decoded into a JavaScript null at runtime.

TypeScript or JavaScript:

```
import { Canister, Null, query } from 'azle';

export default Canister({
    getNull: query([], Null, () => {
        return null;
    }),
    printNull: query([Null], Null, (null_) => {
        console.log(typeof null_);
        return null_;
    })
});
```

Candid:

```
service : () -> {
    getNull : () -> (null) query;
    printNull : (null) -> (null) query;
}
```

dfx:

```
dfx canister call candid_canister printNull '(null)'
(null : null)
```

opt

The `CandidType` object `Opt` corresponds to the Candid type opt, is inferred to be a TypeScript `Opt<T>`, and will be decoded into a JavaScript Object at runtime.

It is a variant with `Some` and `None` cases. At runtime if the value of the variant is `Some`, the `Some` property of the variant object will have a value of the enclosed `Opt` type at runtime.

TypeScript or JavaScript:

```
import { bool, Canister, None, Opt, query, Some } from 'azle';

export default Canister({
    getOptSome: query([], Opt(bool), () => {
        return Some(true); // equivalent to { Some: true }
    }),
    getOptNone: query([], Opt(bool), () => {
        return None; //equivalent to { None: null}
    })
});
```

Candid:

```
service : () -> {
    getOptNone : () -> (opt bool) query;
    getOptSome : () -> (opt bool) query;
}
```

dfx:

```
dfx canister call candid_canister getOptSome
(opt true)

dfx canister call candid_canister getOptNone
(null)
```

principal

The `CandidType` object `Principal` corresponds to the Candid type principal, is inferred to be a TypeScript @dfinity/principal `Principal`, and will be decoded into an @dfinity/principal Principal at runtime.

TypeScript or JavaScript:

```
import { Canister, Principal, query } from 'azle';

export default Canister({
    getPrincipal: query([], Principal, () => {
        return Principal.fromText('rrkah-fqaaa-aaaaa-aaaaq-cai');
    }),
    printPrincipal: query([Principal], Principal, (principal) => {
        console.log(typeof principal);
        return principal;
    })
});
```

Candid:

```
service : () -> {
    getPrincipal : () -> (principal) query;
    printPrincipal : (principal) -> (principal) query;
}
```

dfx:

```
dfx canister call candid_canister printPrincipal '(principal "rrkah-fqaaa-
aaaaa-aaaaq-cai")'
(principal "rrkah-fqaaa-aaaaa-aaaaq-cai")
```

record

Objects created by the `CandidType` function `Record` correspond to the Candid record type, are inferred to be TypeScript `Object`s, and will be decoded into JavaScript Objects at runtime.

The shape of the object will match the object literal passed to the `Record` function.

TypeScript or JavaScript:

```
import { Canister, Principal, query, Record, text } from 'azle';

const User = Record({
    id: Principal,
    username: text
});

export default Canister({
    getUser: query([], User, () => {
        return {
            id: Principal.fromUint8Array(Uint8Array.from([0])),
            username: 'lastmjs'
        };
    }),
    printUser: query([User], User, (user) => {
        console.log(typeof user);
        return user;
    })
});
```

Candid:

```
type User = record { id : principal; username : text };
service : () -> {
    getUser : () -> (User) query;
    printUser : (User) -> (User) query;
}
```

dfx:

```
dfx canister call candid_canister printUser '(record { id = principal "2ibo7-
dia"; username = "lastmjs" })'
(record { id = principal "2ibo7-dia"; username = "lastmjs" })
```

reserved

The `CandidType` object `reserved` corresponds to the Candid type reserved, is inferred to be a TypeScript `any`, and will be decoded into a JavaScript null at runtime.

TypeScript or JavaScript:

```
import { Canister, query, reserved } from 'azle';

export default Canister({
    getReserved: query([], reserved, () => {
        return 'anything';
    }),
    printReserved: query([reserved], reserved, (reserved) => {
        console.log(typeof reserved);
        return reserved;
    })
});
```

Candid:

```
service : () -> {
    getReserved : () -> (reserved) query;
    printReserved : (reserved) -> (reserved) query;
}
```

dfx:

```
dfx canister call candid_canister printReserved '(null)'
(null : reserved)
```

service

Values created by the `CandidType` function `Canister` correspond to the Candid service type, are inferred to be TypeScript `Object` s, and will be decoded into JavaScript Objects at runtime.

The properties of this object that match the keys of the service's `query` and `update` methods can be passed into `ic.call` and `ic.notify` to perform cross-canister calls.

TypeScript or JavaScript:

```
import { bool, Canister, ic, Principal, query, text, update } from 'azle';

const SomeCanister = Canister({
    query1: query([], bool),
    update1: update([], text)
});

export default Canister({
    getService: query([], SomeCanister, () => {
        return SomeCanister(Principal.fromText('aaaaa-aa'));
    }),
    callService: update([SomeCanister], text, (service) => {
        return ic.call(service.update1);
    })
});
```

Candid:

```
type ManualReply = variant { Ok : text; Err : text };
service : () -> {
  callService : (
      service { query1 : () -> (bool) query; update1 : () -> (text) },
    ) -> (ManualReply);
  getService : () -> (
      service { query1 : () -> (bool) query; update1 : () -> (text) },
    ) query;
}
```

dfx:

```
dfx canister call candid_canister getService
(service "aaaaa-aa")
```

text

The `CandidType` object `text` corresponds to the Candid type text, is inferred to be a TypeScript `string`, and will be decoded into a JavaScript String at runtime.

TypeScript or JavaScript:

```
import { Canister, query, text } from 'azle';

export default Canister({
    getString: query([], text, () => {
        return 'Hello world!';
    }),
    printString: query([text], text, (string) => {
        console.log(typeof string);
        return string;
    })
});
```

Candid:

```
service : () -> {
    getString : () -> (text) query;
    printString : (text) -> (text) query;
}
```

dfx:

```
dfx canister call candid_canister printString '("Hello world!")'
("Hello world!")
```

variant

Objects created by the `CandidType` function `Variant` correspond to the Candid variant type, are inferred to be TypeScript `Object` s, and will be decoded into JavaScript Objects at runtime.

The shape of the object will match the object literal passed to the `Variant` function, however it will contain only one of the enumerated properties.

TypeScript or JavaScript:

```
import { Canister, Null, query, Variant } from 'azle';

const Emotion = Variant({
    Happy: Null,
    Indifferent: Null,
    Sad: Null
});

const Reaction = Variant({
    Fire: Null,
    ThumbsUp: Null,
    Emotion: Emotion
});

export default Canister({
    getReaction: query([], Reaction, () => {
        return {
            Fire: null
        };
    }),
    printReaction: query([Reaction], Reaction, (reaction) => {
        console.log(typeof reaction);
        return reaction;
    })
});
```

Candid:

```
type Emotion = variant { Sad; Indifferent; Happy };
type Reaction = variant { Emotion : Emotion; Fire; ThumbsUp };
service : () -> {
    getReaction : () -> (Reaction) query;
    printReaction : (Reaction) -> (Reaction) query;
}
```

dfx:

```
dfx canister call candid_canister printReaction '(variant { Fire })'
(variant { Fire })
```

vec

The `CandidType` object `Vec` corresponds to the Candid type vec, is inferred to be a TypeScript `T[]`, and will be decoded into a JavaScript array of the specified type at runtime (except for `Vec<nat8>` which will become a `Uint8Array`, thus it is recommended to use the `blob` type instead of `Vec<nat8>`).

TypeScript or JavaScript:

```
import { Canister, int32, Vec, query } from 'azle';

export default Canister({
    getNumbers: query([], Vec(int32), () => {
        return [0, 1, 2, 3];
    }),
    printNumbers: query([Vec(int32)], Vec(int32), (numbers) => {
        console.log(typeof numbers);
        return numbers;
    })
});
```

Candid:

```
service : () -> {
    getNumbers : () -> (vec int32) query;
    printNumbers : (vec int32) -> (vec int32) query;
}
```

dfx:

```
dfx canister call candid_canister printNumbers '(vec { 0 : int32; 1 : int32;
2 : int32; 3 : int32 })'
(vec { 0 : int32; 1 : int32; 2 : int32; 3 : int32 })
```

Canister APIs

- candid decode
- candid encode
- canister balance
- canister balance 128
- canister version
- canister id
- data certificate
- instruction counter
- is controller
- performance counter
- print
- set certified data
- time
- trap

candid decode

This section is a work in progress.

Examples:

- call_raw
- candid_encoding

```
import { blob, Canister, ic, query, text } from 'azle';

export default Canister({
    // decodes Candid bytes to a Candid string
    candidDecode: query([blob], text, (candidEncoded) => {
        return ic.candidDecode(candidEncoded);
    })
});
```

candid encode

This section is a work in progress.

Examples:

- call_raw
- candid_encoding
- manual_reply
- notify_raw
- outgoing_http_requests

```
import { blob, Canister, ic, query, text } from 'azle';

export default Canister({
    // encodes a Candid string to Candid bytes
    candidEncode: query([text], blob, (candidString) => {
        return ic.candidEncode(candidString);
    })
});
```

canister balance

This section is a work in progress.

Examples:

- cycles
- ic_api

```
import { Canister, ic, nat64, query } from 'azle';

export default Canister({
    // returns the amount of cycles available in the canister
    canisterBalance: query([], nat64, () => {
        return ic.canisterBalance();
    })
});
```

canister balance 128

This section is a work in progress.

Examples:

- cycles
- ic_api

```
import { Canister, ic, nat, query } from 'azle';

export default Canister({
    // returns the amount of cycles available in the canister
    canisterBalance128: query([], nat, () => {
        return ic.canisterBalance128();
    })
});
```

canister version

This section is a work in progress.

Examples:

- ic_api

```
import { Canister, ic, nat64, query } from 'azle';

export default Canister({
    // returns the canister's version number
    canisterVersion: query([], nat64, () => {
        return ic.canisterVersion();
    })
});
```

canister id

This section is a work in progress.

Examples:

- ethereum_json_rpc
- ic_api
- http_counter
- outgoing_http_requests
- whoami

```
import { Canister, ic, Principal, query } from 'azle';

export default Canister({
    // returns this canister's id
    id: query([], Principal, () => {
        return ic.id();
    })
});
```

data certificate

This section is a work in progress.

Examples:

- ic_api

```
import { blob, Canister, ic, Opt, query } from 'azle';

export default Canister({
    // When called from a query call, returns the data certificate
    // authenticating certified_data set by this canister. Returns None if
not
    // called from a query call.
    dataCertificate: query([], Opt(blob), () => {
        return ic.dataCertificate();
    })
});
```

instruction counter

This section is a work in progress.

Examples:

- ic_api

```
import { Canister, ic, nat64, query } from 'azle';

export default Canister({
    // Returns the number of instructions that the canister executed since
the
    // last entry point.
    instructionCounter: query([], nat64, () => {
        return ic.instructionCounter();
    })
});
```

is controller

This section is a work in progress.

Examples:

- ic_api

```
import { bool, Canister, ic, Principal, query } from 'azle';

export default Canister({
    // determines whether the given principal is a controller of the canister
    isController: query([Principal], bool, (principal) => {
        return ic.isController(principal);
    })
});
```

performance counter

This section is a work in progress.

Examples:

- ic_api

```
import { Canister, ic, nat64, query } from 'azle';

export default Canister({
    performanceCounter: query([], nat64, () => {
        return ic.performanceCounter(0);
    })
});
```

print

This section is a work in progress.

Examples:

- ic_api
- null_example

```
import { bool, Canister, ic, query, text } from 'azle';

export default Canister({
    // prints a message through the local replica's output
    print: query([text], bool, (message) => {
        ic.print(message);

        return true;
    })
});
```

set certified data

This section is a work in progress.

Examples:

- ic_api

```
import { blob, Canister, ic, update, Void } from 'azle';

export default Canister({
    // sets up to 32 bytes of certified data
    setCertifiedData: update([blob], Void, (data) => {
        ic.setCertifiedData(data);
    })
});
```

time

This section is a work in progress.

Examples:

- audio_recorder
- ic_api

```
import { Canister, ic, nat64, query } from 'azle';

export default Canister({
    // returns the current timestamp
    time: query([], nat64, () => {
        return ic.time();
    })
});
```

trap

This section is a work in progress.

Examples:

- cross_canister_calls
- ethereum_json_rpc
- http_counter
- ic_api
- outgoing_http_requests
- threshold_ecdsa

```
import { bool, Canister, ic, query, text } from 'azle';

export default Canister({
    // traps with a message, stopping execution and discarding all state
within the call
    trap: query([text], bool, (message) => {
        ic.trap(message);

        return true;
    })
});
```

Canister Methods

- heartbeat
- http_request
- http_request_update
- init
- inspect message
- post upgrade
- pre upgrade
- query
- update

heartbeat

This section is a work in progress.

Examples:

- heartbeat
- run_time_errors

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

export default Canister({
    heartbeat: heartbeat(() => {
        console.log('this runs ~1 time per second');
    })
});
```

http_request

This section is a work in progress.

Examples:

- http_counter

```
import {
    blob,
    bool,
    Canister,
    Func,
    nat16,
    None,
    Opt,
    query,
    Record,
    text,
    Tuple,
    Variant,
    Vec
} from 'azle';

const Token = Record({
    // add whatever fields you'd like
    arbitrary_data: text
});

const StreamingCallbackHttpResponse = Record({
    body: blob,
    token: Opt(Token)
});

export const Callback = Func([text], StreamingCallbackHttpResponse, 'query');

const CallbackStrategy = Record({
    callback: Callback,
    token: Token
});

const StreamingStrategy = Variant({
    Callback: CallbackStrategy
});

type HeaderField = [text, text];
const HeaderField = Tuple(text, text);

const HttpResponse = Record({
    status_code: nat16,
    headers: Vec(HeaderField),
    body: blob,
    streaming_strategy: Opt(StreamingStrategy),
    upgrade: Opt(bool)
});

const HttpRequest = Record({
    method: text,
    url: text,
    headers: Vec(HeaderField),
    body: blob,
    certificate_version: Opt(nat16)
});
```

```
export default Canister({
    http_request: query([HttpRequest], HttpResponse, (req) => {
        return {
            status_code: 200,
            headers: [],
            body: Buffer.from('hello'),
            streaming_strategy: None,
            upgrade: None
        };
    })
});
```

http_request

This section is a work in progress.

Examples:

- http_counter

```
import {
    blob,
    bool,
    Canister,
    Func,
    nat16,
    None,
    Opt,
    Record,
    text,
    Tuple,
    update,
    Variant,
    Vec
} from 'azle';

const Token = Record({
    // add whatever fields you'd like
    arbitrary_data: text
});

const StreamingCallbackHttpResponse = Record({
    body: blob,
    token: Opt(Token)
});

export const Callback = Func([text], StreamingCallbackHttpResponse, 'query');

const CallbackStrategy = Record({
    callback: Callback,
    token: Token
});

const StreamingStrategy = Variant({
    Callback: CallbackStrategy
});

type HeaderField = [text, text];
const HeaderField = Tuple(text, text);

const HttpResponse = Record({
    status_code: nat16,
    headers: Vec(HeaderField),
    body: blob,
    streaming_strategy: Opt(StreamingStrategy),
    upgrade: Opt(bool)
});

const HttpRequest = Record({
    method: text,
    url: text,
    headers: Vec(HeaderField),
    body: blob,
    certificate_version: Opt(nat16)
});
```

```
export default Canister({
    http_request_update: update([HttpRequest], HttpResponse, (req) => {
        return {
            status_code: 200,
            headers: [],
            body: Buffer.from('hello'),
            streaming_strategy: None,
            upgrade: None
        };
    })
});
```

init

This section is a work in progress.

Examples:

- ethereum_json_rpc
- func_types
- init
- persistent-storage
- pre_and_post_upgrade
- whoami

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

export default Canister({
    init: init([], () => {
        console.log('This runs once when the canister is first initialized');
    })
});
```

inspect message

This section is a work in progress.

Examples:

- inspect_message
- run_time_errors

```
import { bool, Canister, ic, inspectMessage, update } from 'azle';

export default Canister({
    inspectMessage: inspectMessage(() => {
        console.log('inspectMessage called');

        if (ic.methodName() === 'accessible') {
            ic.acceptMessage();
            return;
        }

        if (ic.methodName() === 'inaccessible') {
            return;
        }

        throw `Method "${ic.methodName()}" not allowed`;
    }),
    accessible: update([], bool, () => {
        return true;
    }),
    inaccessible: update([], bool, () => {
        return false;
    }),
    alsoInaccessible: update([], bool, () => {
        return false;
    })
});
```

post upgrade

This section is a work in progress.

Examples:

- pre_and_post_upgrade
- whoami

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

export default Canister({
    postUpgrade: postUpgrade([], () => {
        console.log('This runs after every canister upgrade');
    })
});
```

pre upgrade

This section is a work in progress.

Examples:

- pre_and_post_upgrade

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

export default Canister({
    preUpgrade: preUpgrade(() => {
        console.log('This runs before every canister upgrade');
    })
});
```

query

This section is a work in progress.

```
import { Canister, query, text } from 'azle';

export default Canister({
    simpleQuery: query([], text, () => {
        return 'This is a query method';
    })
});
```

update

This section is a work in progress.

```
import { Canister, query, text, update, Void } from 'azle';

let message = '';

export default Canister({
    getMessage: query([], text, () => {
        return message;
    }),
    setMessage: update([text], Void, (newMessage) => {
        message = newMessage;
    })
});
```

Environment Variables

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

dfx.json

Modify your `dfx.json` file with the `env` property to specify which environment variables you would like included in your Azle canister's binary. In this case, `CANISTER1_PRINCIPAL` and `CANISTER2_PRINCIPAL` will be included:

```json
{
    "canisters": {
        "canister1": {
            "type": "custom",
            "main": "src/canister1/index.ts",
            "build": "npx azle canister1",
            "candid": "src/canister1/index.did",
            "wasm": ".azle/canister1/canister1.wasm",
            "gzip": true,
            "declarations": {
                "output": "test/dfx_generated/canister1",
                "node_compatibility": true
            },
            "env": ["CANISTER1_PRINCIPAL", "CANISTER2_PRINCIPAL"]
        }
    }
}
```

process.env

You can access the specified environment variables in Azle like so:

```
import { Canister, query, text } from 'azle';

export default Canister({
    canister1PrincipalEnvVar: query([], text, () => {
        return (
            process.env.CANISTER1_PRINCIPAL ??
            'process.env.CANISTER1_PRINCIPAL is undefined'
        );
    }),
    canister2PrincipalEnvVar: query([], text, () => {
        return (
            process.env.CANISTER2_PRINCIPAL ??
            'process.env.CANISTER2_PRINCIPAL is undefined'
        );
    })
});
```

Management Canister

bitcoin_get_balance

This section is a work in progress.

Examples:

- bitcoin

```
import { Canister, ic, None, text, update } from 'azle';
import { managementCanister, Satoshi } from 'azle/canisters/management';

const BITCOIN_API_CYCLE_COST = 100_000_000n;

export default Canister({
    getBalance: update([text], Satoshi, async (address) => {
        return await ic.call(managementCanister.bitcoin_get_balance, {
            args: [
                {
                    address,
                    min_confirmations: None,
                    network: { Regtest: null }
                }
            ],
            cycles: BITCOIN_API_CYCLE_COST
        });
    })
});
```

bitcoin_get_current_fee_percentiles

This section is a work in progress.

Examples:

- bitcoin

```
import { Canister, ic, update, Vec } from 'azle';
import {
    managementCanister,
    MillisatoshiPerByte
} from 'azle/canisters/management';

const BITCOIN_API_CYCLE_COST = 100_000_000n;

export default Canister({
    getCurrentFeePercentiles: update([], Vec(MillisatoshiPerByte), async ()
=> {
        return await ic.call(
            managementCanister.bitcoin_get_current_fee_percentiles,
            {
                args: [
                    {
                        network: { Regtest: null }
                    }
                ],
                cycles: BITCOIN_API_CYCLE_COST
            }
        );
    })
});
```

bitcoin_get_utxos

This section is a work in progress.

Examples:

- bitcoin

```
import { Canister, ic, None, text, update } from 'azle';
import { GetUtxosResult, managementCanister } from 'azle/canisters
/management';

const BITCOIN_API_CYCLE_COST = 100_000_000n;

export default Canister({
    getUtxos: update([text], GetUtxosResult, async (address) => {
        return await ic.call(managementCanister.bitcoin_get_utxos, {
            args: [
                {
                    address,
                    filter: None,
                    network: { Regtest: null }
                }
            ],
            cycles: BITCOIN_API_CYCLE_COST
        });
    })
});
```

bitcoin_send_transaction

This section is a work in progress.

Examples:

```
import { blob, bool, Canister, ic, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

const BITCOIN_BASE_TRANSACTION_COST = 5_000_000_000n;
const BITCOIN_CYCLE_COST_PER_TRANSACTION_BYTE = 20_000_000n;

export default Canister({
    sendTransaction: update([blob], bool, async (transaction) => {
        const transactionFee =
            BITCOIN_BASE_TRANSACTION_COST +
            BigInt(transaction.length) *
                BITCOIN_CYCLE_COST_PER_TRANSACTION_BYTE;

        await ic.call(managementCanister.bitcoin_send_transaction, {
            args: [
                {
                    transaction,
                    network: { Regtest: null }
                }
            ],
            cycles: transactionFee
        });

        return true;
    })
});
```

canister_status

This section is a work in progress.

Examples:

- management_canister

```
import { Canister, ic, update } from 'azle';
import {
    CanisterStatusArgs,
    CanisterStatusResult,
    managementCanister
} from 'azle/canisters/management';

export default Canister({
    getCanisterStatus: update(
        [CanisterStatusArgs],
        CanisterStatusResult,
        async (args) => {
            return await ic.call(managementCanister.canister_status, {
                args: [args]
            });
        }
    )
});
```

create_canister

This section is a work in progress.

Examples:

- management_canister

```
import { Canister, ic, None, update } from 'azle';
import {
    CreateCanisterResult,
    managementCanister
} from 'azle/canisters/management';

export default Canister({
    executeCreateCanister: update([], CreateCanisterResult, async () => {
        return await ic.call(managementCanister.create_canister, {
            args: [{ settings: None }],
            cycles: 50_000_000_000_000n
        });
    })
});
```

delete_canister

This section is a work in progress.

Examples:

- management_canister

```
import { bool, Canister, ic, Principal, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

export default Canister({
    executeDeleteCanister: update([Principal], bool, async (canisterId) => {
        await ic.call(managementCanister.delete_canister, {
            args: [
                {
                    canister_id: canisterId
                }
            ]
        });

        return true;
    })
});
```

deposit_cycles

This section is a work in progress.

Examples:

- management_canister

```
import { bool, Canister, ic, Principal, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

export default Canister({
    executeDepositCycles: update([Principal], bool, async (canisterId) => {
        await ic.call(managementCanister.deposit_cycles, {
            args: [
                {
                    canister_id: canisterId
                }
            ],
            cycles: 10_000_000n
        });

        return true;
    })
});
```

ecdsa_public_key

This section is a work in progress.

Examples:

- threshold_ecdsa

```
import { blob, Canister, ic, None, Record, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

const PublicKey = Record({
    publicKey: blob
});

export default Canister({
    publicKey: update([], PublicKey, async () => {
        const caller = ic.caller().toUint8Array();

        const publicKeyResult = await ic.call(
            managementCanister.ecdsa_public_key,
            {
                args: [
                    {
                        canister_id: None,
                        derivation_path: [caller],
                        key_id: {
                            curve: { secp256k1: null },
                            name: 'dfx_test_key'
                        }
                    }
                ]
            }
        );

        return {
            publicKey: publicKeyResult.public_key
        };
    })
});
```

http_request

This section is a work in progress.

Examples:

- ethereum_json_rpc
- outgoing_http_requests

```
import { Canister, ic, None, Principal, query, Some, update } from 'azle';
import {
    HttpResponse,
    HttpTransformArgs,
    managementCanister
} from 'azle/canisters/management';

export default Canister({
    xkcd: update([], HttpResponse, async () => {
        return await ic.call(managementCanister.http_request, {
            args: [
                {
                    url: `https://xkcd.com/642/info.0.json`,
                    max_response_bytes: Some(2_000n),
                    method: {
                        get: null
                    },
                    headers: [],
                    body: None,
                    transform: Some({
                        function: [ic.id(), 'xkcdTransform'] as [
                            Principal,
                            string
                        ],
                        context: Uint8Array.from([])
                    })
                }
            ],
            cycles: 50_000_000n
        });
    }),
    xkcdTransform: query([HttpTransformArgs], HttpResponse, (args) => {
        return {
            ...args.response,
            headers: []
        };
    })
});
```

install_code

This section is a work in progress.

Examples:

- management_canister

```
import { blob, bool, Canister, ic, Principal, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

export default Canister({
    executeInstallCode: update(
        [Principal, blob],
        bool,
        async (canisterId, wasmModule) => {
            await ic.call(managementCanister.install_code, {
                args: [
                    {
                        mode: {
                            install: null
                        },
                        canister_id: canisterId,
                        wasm_module: wasmModule,
                        arg: Uint8Array.from([])
                    }
                ],
                cycles: 100_000_000_000n
            });

            return true;
        }
    )
});
```

provisional_create_canister_with_cycles

This section is a work in progress.

Examples:

- management_canister

```
import { Canister, ic, None, update } from 'azle';
import {
    managementCanister,
    ProvisionalCreateCanisterWithCyclesResult
} from 'azle/canisters/management';

export default Canister({
    provisionalCreateCanisterWithCycles: update(
        [],
        ProvisionalCreateCanisterWithCyclesResult,
        async () => {
            return await ic.call(
                managementCanister.provisional_create_canister_with_cycles,
                {
                    args: [
                        {
                            amount: None,
                            settings: None
                        }
                    ]
                }
            );
        }
    )
});
```

provisional_top_up_canister

This section is a work in progress.

Examples:

- management_canister

```
import { bool, Canister, ic, nat, Principal, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

export default Canister({
    provisionalTopUpCanister: update(
        [Principal, nat],
        bool,
        async (canisterId, amount) => {
            await ic.call(managementCanister.provisional_top_up_canister, {
                args: [
                    {
                        canister_id: canisterId,
                        amount
                    }
                ]
            });

            return true;
        }
    )
});
```

raw_rand

This section is a work in progress.

Examples:

- async/await
- heartbeat
- management_canister
- timers

```
import { blob, Canister, ic, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

export default Canister({
    getRawRand: update([], blob, async () => {
        return await ic.call(managementCanister.raw_rand);
    })
});
```

sign_with_ecdsa

This section is a work in progress.

Examples:

- threshold_ecdsa

```
import { blob, Canister, ic, Record, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

const Signature = Record({
    signature: blob
});

export default Canister({
    sign: update([blob], Signature, async (messageHash) => {
        if (messageHash.length !== 32) {
            ic.trap('messageHash must be 32 bytes');
        }

        const caller = ic.caller().toUint8Array();

        const signatureResult = await ic.call(
            managementCanister.sign_with_ecdsa,
            {
                args: [
                    {
                        message_hash: messageHash,
                        derivation_path: [caller],
                        key_id: {
                            curve: { secp256k1: null },
                            name: 'dfx_test_key'
                        }
                    }
                ],
                cycles: 10_000_000_000n
            }
        );

        return {
            signature: signatureResult.signature
        };
    })
});
```

start_canister

This section is a work in progress.

Examples:

- management_canister

```
import { bool, Canister, ic, Principal, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

export default Canister({
    executeStartCanister: update([Principal], bool, async (canisterId) => {
        await ic.call(managementCanister.start_canister, {
            args: [
                {
                    canister_id: canisterId
                }
            ]
        });

        return true;
    })
});
```

stop_canister

This section is a work in progress.

Examples:

- management_canister

```
import { bool, Canister, ic, Principal, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

export default Canister({
    executeStopCanister: update([Principal], bool, async (canisterId) => {
        await ic.call(managementCanister.stop_canister, {
            args: [
                {
                    canister_id: canisterId
                }
            ]
        });

        return true;
    })
});
```

uninstall_code

This section is a work in progress.

Examples:

- management_canister

```
import { bool, Canister, ic, Principal, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

export default Canister({
    executeUninstallCode: update([Principal], bool, async (canisterId) => {
        await ic.call(managementCanister.uninstall_code, {
            args: [
                {
                    canister_id: canisterId
                }
            ]
        });

        return true;
    })
});
```

update_settings

This section is a work in progress.

Examples:

- management_canister

```
import { bool, Canister, ic, None, Principal, Some, update } from 'azle';
import { managementCanister } from 'azle/canisters/management';

export default Canister({
    executeUpdateSettings: update([Principal], bool, async (canisterId) => {
        await ic.call(managementCanister.update_settings, {
            args: [
                {
                    canister_id: canisterId,
                    settings: {
                        controllers: None,
                        compute_allocation: Some(1n),
                        memory_allocation: Some(3_000_000n),
                        freezing_threshold: Some(2_000_000n)
                    }
                }
            ]
        });

        return true;
    })
});
```

Plugins

Azle plugins allow developers to wrap Rust code in TypeScript/JavaScript APIs that can then be exposed to Azle canisters, providing a clean and simple developer experience with the underlying Rust code.

Plugins are in a very early alpha state. You can create and use them now, but be aware that the API will be changing significantly in the near future.

You can use the following example plugins as you create your own plugins:

Local plugin

If you just want to create a plugin in the same repo as your project, see the plugins example.

npm plugin

If you want to create a plugin that can be published and/or used with npm, see the ic-sqlite-plugin example.

Stable Memory

- stable structures
- stable bytes
- stable grow
- stable read
- stable size
- stable write
- stable64 grow
- stable64 read
- stable64 size
- stable64 write

stable structures

This section is a work in progress.

Examples:

- audio_recorder
- ethereum_json_rpc
- func_types
- http_counter
- inline_types
- persistent-storage
- pre_and_post_upgrade
- stable_structures

```
import {
    bool,
    Canister,
    nat64,
    nat8,
    Opt,
    query,
    StableBTreeMap,
    text,
    Tuple,
    update,
    Vec
} from 'azle';

const Key = nat8;
const Value = text;

let map = StableBTreeMap(Key, Value, 0);

export default Canister({
    containsKey: query([Key], bool, (key) => {
        return map.containsKey(key);
    }),
    get: query([Key], Opt(Value), (key) => {
        return map.get(key);
    }),
    mapInsert: update([Key, Value], Opt(Value), (key, value) => {
        return map.insert(key, value);
    }),
    isEmpty: query([], bool, () => {
        return map.isEmpty();
    }),
    items: query([], Vec(Tuple(Key, Value)), () => {
        return map.items();
    }),
    keys: query([], Vec(Key), () => {
        return map.keys();
    }),
    len: query([], nat64, () => {
        return map.len();
    }),
    mapRemove: update([Key], Opt(Value), (key) => {
        return map.remove(key);
    }),
    values: query([], Vec(Value), () => {
        return map.values();
    })
});
```

stable bytes

This section is a work in progress.

Examples:

- stable_memory

```
import { blob, Canister, ic, query } from 'azle';

export default Canister({
    stableBytes: query([], blob, () => {
        return ic.stableBytes();
    })
});
```

stable grow

This section is a work in progress.

Examples:

- stable_memory

```
import { Canister, ic, nat32, update } from 'azle';

export default Canister({
    stableGrow: update([nat32], nat32, (newPages) => {
        return ic.stableGrow(newPages);
    })
});
```

stable read

This section is a work in progress.

Examples:

- stable_memory

```
import { blob, Canister, ic, nat32, query } from 'azle';

export default Canister({
    stableRead: query([nat32, nat32], blob, (offset, length) => {
        return ic.stableRead(offset, length);
    })
});
```

stable size

This section is a work in progress.

Examples:

- stable_memory

```
import { Canister, ic, nat32, query } from 'azle';

export default Canister({
    stableSize: query([], nat32, () => {
        return ic.stableSize();
    })
});
```

stable write

This section is a work in progress.

Examples:

- stable_memory

```
import { blob, Canister, ic, nat32, update, Void } from 'azle';

export default Canister({
    stableWrite: update([nat32, blob], Void, (offset, buf) => {
        ic.stableWrite(offset, buf);
    })
});
```

stable64 grow

This section is a work in progress.

Examples:

- stable_memory

```
import { Canister, ic, nat64, update } from 'azle';

export default Canister({
    stable64Grow: update([nat64], nat64, (newPages) => {
        return ic.stable64Grow(newPages);
    })
});
```

stable64 read

This section is a work in progress.

Examples:

- stable_memory

```
import { blob, Canister, ic, nat64, query } from 'azle';

export default Canister({
    stable64Read: query([nat64, nat64], blob, (offset, length) => {
        return ic.stable64Read(offset, length);
    })
});
```

stable64 size

This section is a work in progress.

Examples:

- stable_memory

```
import { Canister, ic, nat64, query } from 'azle';

export default Canister({
    stable64Size: query([], nat64, () => {
        return ic.stable64Size();
    })
});
```

stable64 write

This section is a work in progress.

Examples:

- stable_memory

```
import { blob, Canister, ic, nat64, update, Void } from 'azle';

export default Canister({
    stable64Write: update([nat64, blob], Void, (offset, buf) => {
        ic.stable64Write(offset, buf);
    })
});
```

Timers

- clear timer
- set timer
- set timer interval

clear timer

This section is a work in progress.

Examples:

- timers

```
import { Canister, ic, TimerId, update, Void } from 'azle';

export default Canister({
    clearTimer: update([TimerId], Void, (timerId) => {
        ic.clearTimer(timerId);
    })
});
```

set timer

This section is a work in progress.

Examples:

- timers

```
import { Canister, Duration, ic, TimerId, Tuple, update } from 'azle';

export default Canister({
    setTimers: update([Duration], Tuple(TimerId, TimerId), (delay) => {
        const functionTimerId = ic.setTimer(delay, callback);

        const capturedValue = '🚩';

        const closureTimerId = ic.setTimer(delay, () => {
            console.log(`closure called and captured value
${capturedValue}`);
        });

        return [functionTimerId, closureTimerId];
    })
});

function callback() {
    console.log('callback called');
}
```

set timer interval

This section is a work in progress.

Examples:

- timers

```
import { Canister, Duration, ic, TimerId, Tuple, update } from 'azle';

export default Canister({
    setTimerIntervals: update(
        [Duration],
        Tuple(TimerId, TimerId),
        (interval) => {
            const functionTimerId = ic.setTimerInterval(interval, callback);

            const capturedValue = '🚩';

            const closureTimerId = ic.setTimerInterval(interval, () => {
                console.log(
                    `closure called and captured value ${capturedValue}`
                );
            });

            return [functionTimerId, closureTimerId];
        }
    )
});

function callback() {
    console.log('callback called');
}
```

Wasm Binary Optimization

The IC currently limits Wasm binaries to a relatively small size of ~2MiB (with some caveats). You are likely to hit this limit as your Azle canisters grow in size. Azle provides some automatic optimizations to help you deal with this limit. It is hoped that the IC-imposed limit will be greatly increased sometime in 2023.

To optimize the Wasm binary of an Azle canister, you can add the `opt_level` property to your `dfx.json` with the following options: `"0"`, `"1"`, `"2"`, `"3"`, or `"4"`. `"0"` is the default option if `opt_level` is not specified.

Each option is intended to reduce the size of your Wasm binary as the value increases. Each option is likely to take longer to compile than the previous option. It is recommended to start at `"1"` and increase only as necessary.

Here's an example using `opt_level` `"1"`:

```
{
    "canisters": {
        "hello_world": {
            "type": "custom",
            "main": "src/index.ts",
            "build": "npx azle hello_world",
            "candid": "src/index.did",
            "wasm": ".azle/hello_world/hello_world.wasm.gz",
            "opt_level": "1"
        }
    }
}
```