Hyperledger Cactus

Release 1.0.0-rc.3

Hyperledger Cactus

Feb 16, 2022

TABLE OF CONTENTS:

Hyperledger Cactus aims to provide Decentralized, Secure and Adaptable Integration between Blockchain Networks. Hyperledger Cactus is currently undergoing a major refactoring effort to enable the desired to-be architecture which will enable plug-in based collaborative development to increase the breadth of use cases & Ledgers supported.

What is Cactus?

A pluggable, enterprise-grade framework to transact on multiple distributed ledgers without introducing yet another competing blockchain.

- Cactus allows developers to abstract the application layer from the DLT addressing protocol fragmentation, lowering coupling and reducing implementation risks
- Cactus allows different DLT networks to interact with each other, through atomic transactions and state commits, this eliminates information silos and increases network's value

Why use Cactus?

- Maximize flexibility and future-proofing through plug-in architecture.
- Avoid needing explicit action from users to have a secure Cactus deployment. Policies such as vaults are built into the SDK
- Keys and other credentials are not stored in source, configuration files, or environment variables
- Preserving Ledger Features Horizontal Scalability.

CHAPTER

ONE

HYPERLEDGER CACTUS

This project is an *Incubation* Hyperledger project. For more information on the history of this project see the Cactus wiki page. Information on what *Active* entails can be found in the Hyperledger Project Lifecycle document.

Hyperledger Cactus aims to provide Decentralized, Secure and Adaptable Integration between Blockchain Networks. Hyperledger Cactus is currently undergoing a major refactoring effort to enable the desired to-be architecture which will enable plug-in based collaborative development to increase the breadth of use cases & Ledgers supported.

1.1 Scope of Project

As blockchain technology proliferates, blockchain integration will become an increasingly important topic in the broader blockchain ecosystem. For instance, people might want to trade between multiple different blockchains that are run on different platforms. Hyperledger Cactus is a web application system designed to allow users to securely integrate different blockchains. It includes a set of libraries, data models, and SDK to accelerate development of an integrated services application. Our goal is to deliver a system that allows users of our code to securely conduct transactions between all of the most commonly used blockchains.

1.2 Run the Examples

1.2.1 Supply Chain Example

1. Ensure a working installation of Docker is present on your machine.
2. Run the following command to pull up the container that will run the example application and the test ledgers as well:

```
docker run \
  --rm \
  --privileged \
  -p 3000:3000 \
  -p 3100:3100 \
  -p 3200:3200 \
  -p 4000:4000 \
  -p 4100:4100 \
  -p 4200:4200 \
  ghcr.io/hyperledger/cactus-example-supply-chain-app:2021-09-08--docs-1312
```

3. Wait for the output to show the message `INFO (api-server): Cactus Cockpit reachable http://0.0.0.0:3100`
4. Visit http://localhost:3100 in a web browser with Javascript enabled
5. Use the graphical user interface to create data on both ledgers and observe that a consistent view of the data from different ledgers is provided.

Once the last command has finished executing, open link printed on the console with a web browser of your choice

1.2.2 Car Trade Example

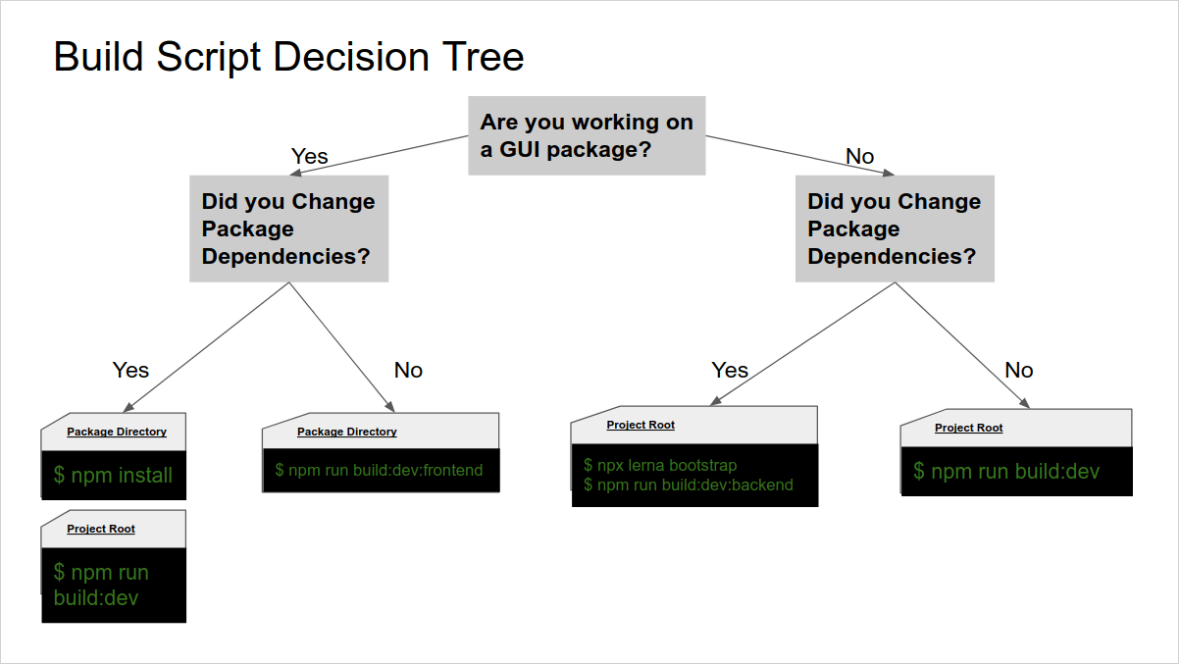
- The guidance is here.

1.2.3 Electricity Trade Example

- The guidance is here.

1.3 Documentation

- Project Wiki: Schedule and logs of the maintainer meetings
- Whitepaper: The official document on Cactus design specifications
- Contributing: How to get from an idea to an approved pull request
- Build: Instructions on how to set up the project for development



- FAQ: A collection of frequently asked questions

1.4 Roadmap

Can be found here: ROADMAP.md

1.5 Contact

- mailing list: cactus@lists.hyperledger.org
- rocketchat channel: https://chat.hyperledger.org/channel/cactus.

1.6 Build/Development Flow

To go from zero to hero with project setup and working on your contributions: BUILD.md

1.7 Contributing

We welcome contributions to Hyperledger Cactus in many forms, and there's always plenty to do!

Please review contributing guidelines to get started.

1.8 License

This distribution is published under the Apache License Version 2.0 found in the LICENSE file.

CHAPTER

TWO

HYPERLEDGER CACTUS BUILD INSTRUCTIONS

This is the place to start if you want to give Cactus a spin on your local machine or if you are planning on contributing.

> This is not a guide for `using` Cactus for your projects that have business logic but rather a guide for people who want to make changes to the code of Cactus. If you are just planning on using Cactus as an npm dependency for your project, then you might not need this guide at all.

The project uses Typescript for both back-end and front-end components.

2.1 Developers guide

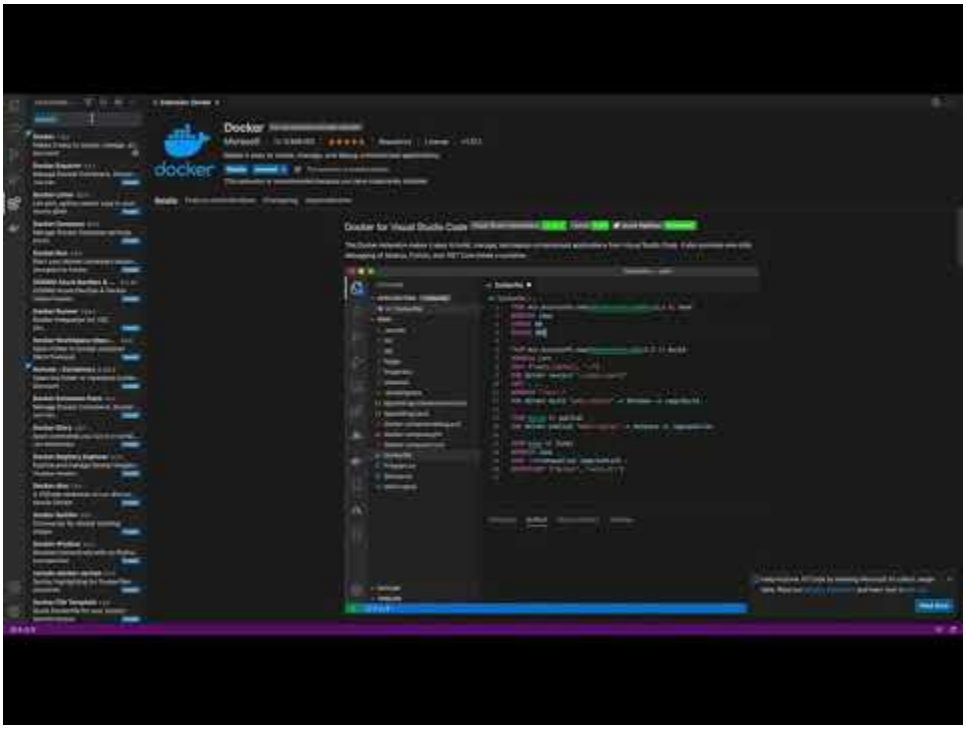
This is a video guide to setup Hyperledger Cactus on your local machine.

2.1.1 Installing git

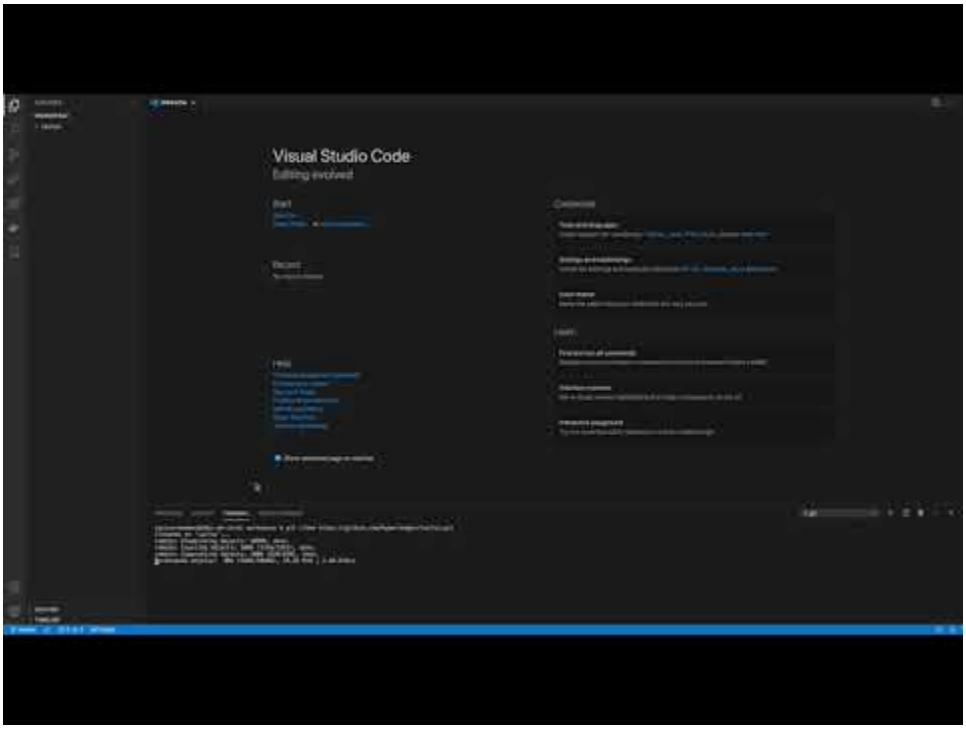
2.1.2 Installing and configuring docker

2.1.3 Installing npm and node

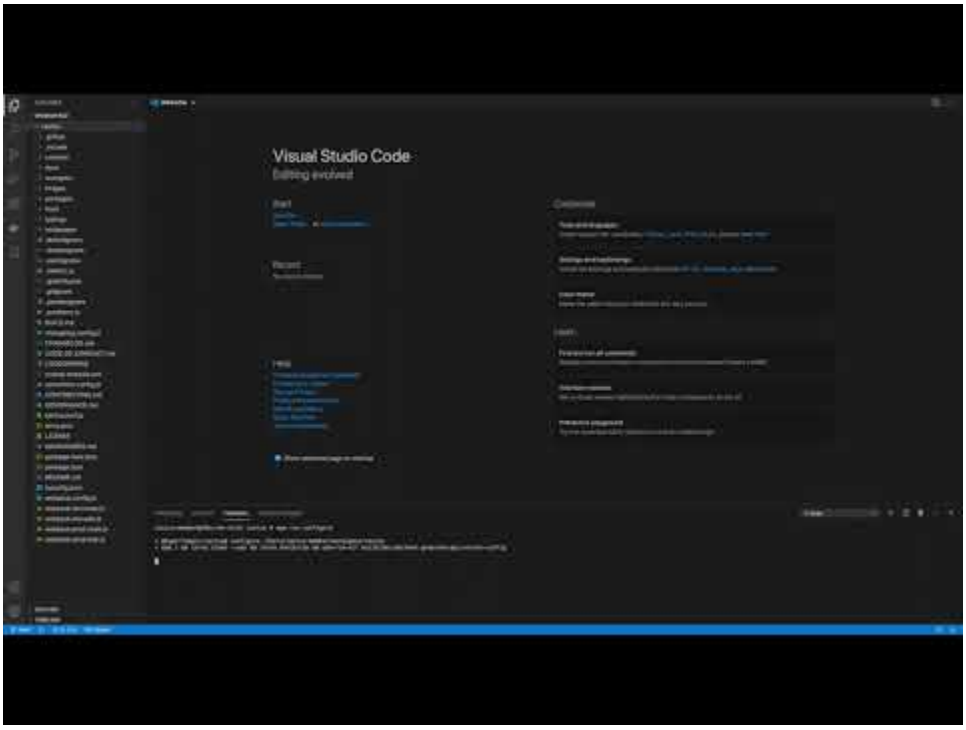
2.1.4 Installing VSCode and plugins



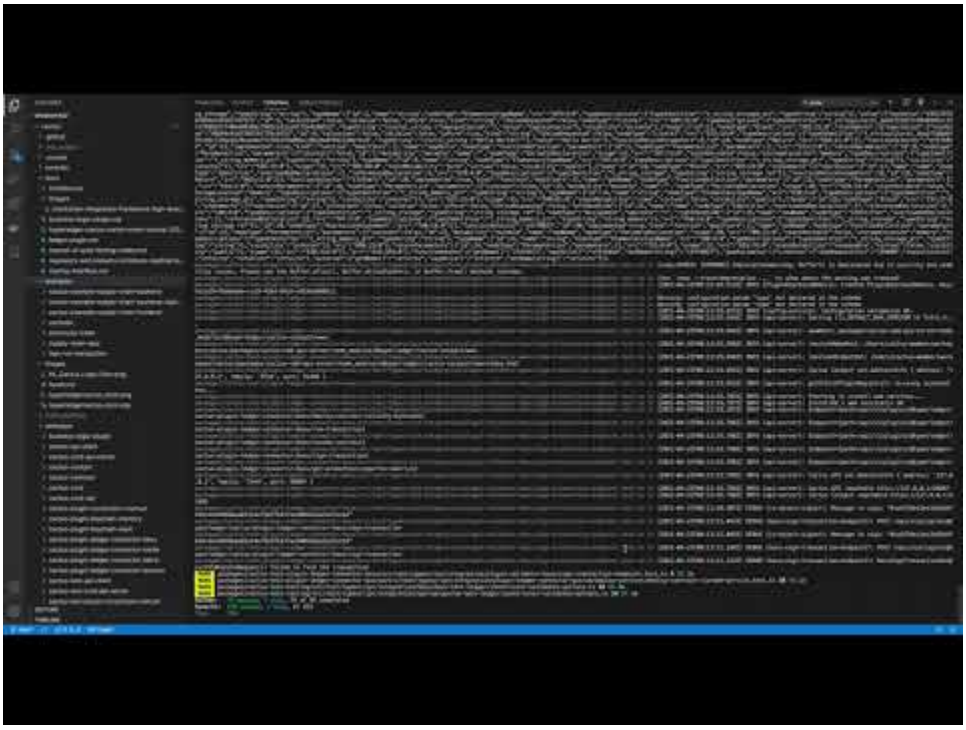
2.1.5 Clone the repository



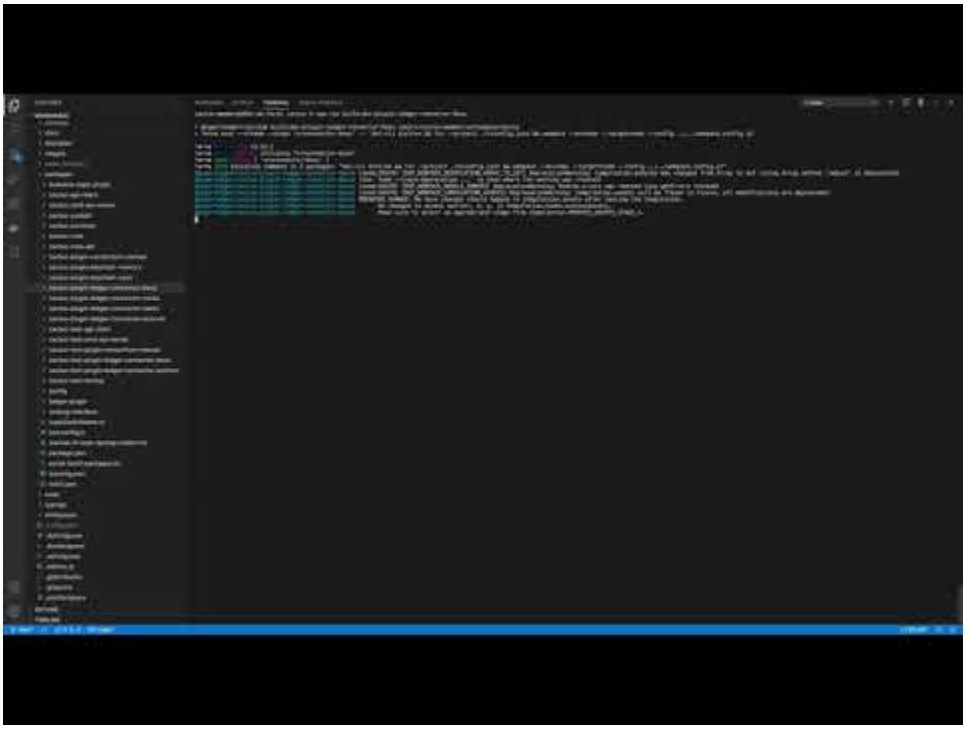
2.1.6 Compiling all packages



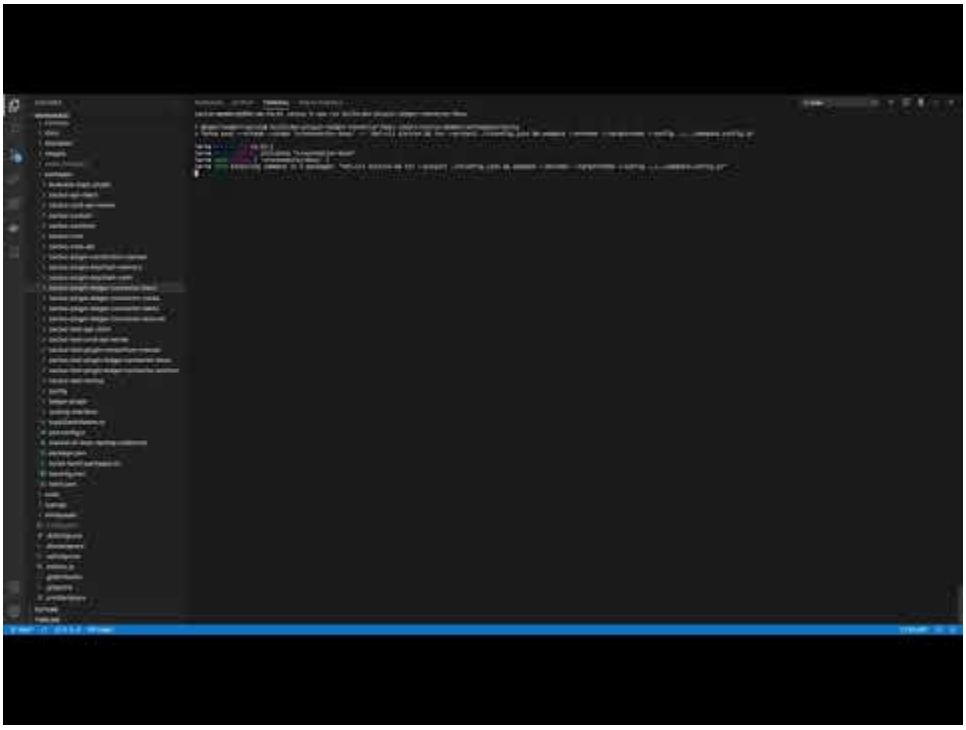
2.1.7 Testing all packages



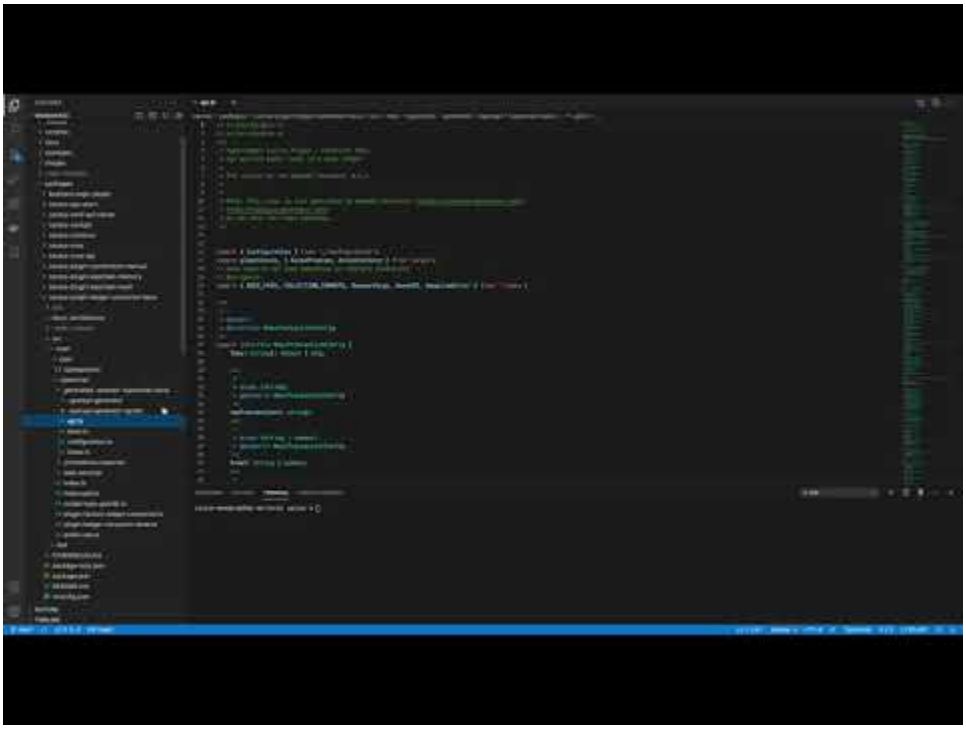
2.1.8 Compiling a specific packages



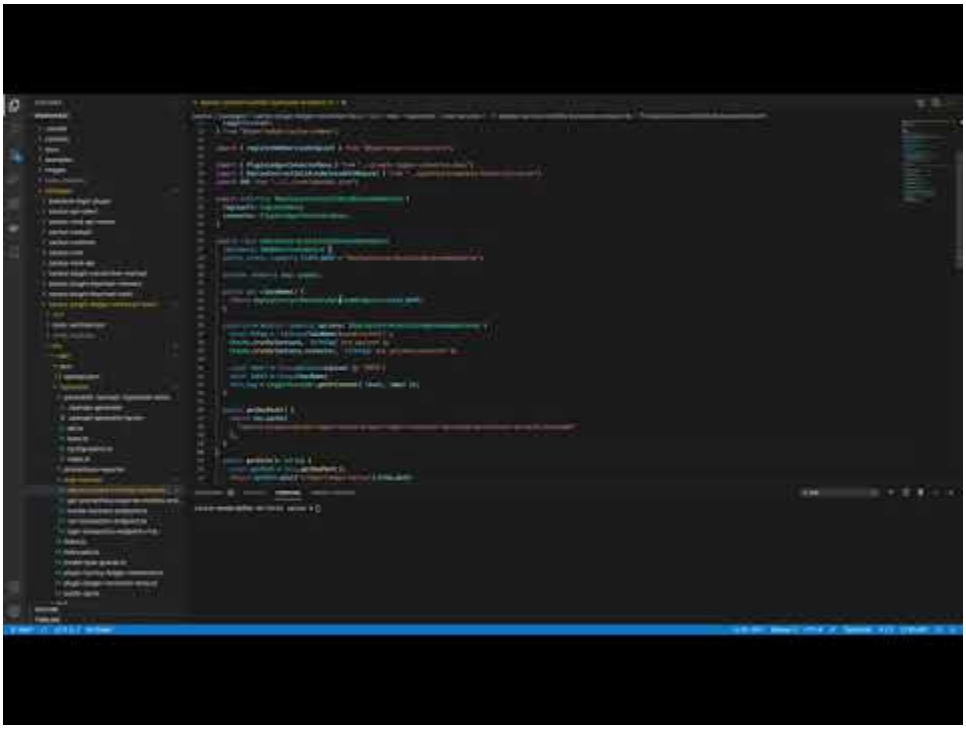
2.1.9 Testing a specific package



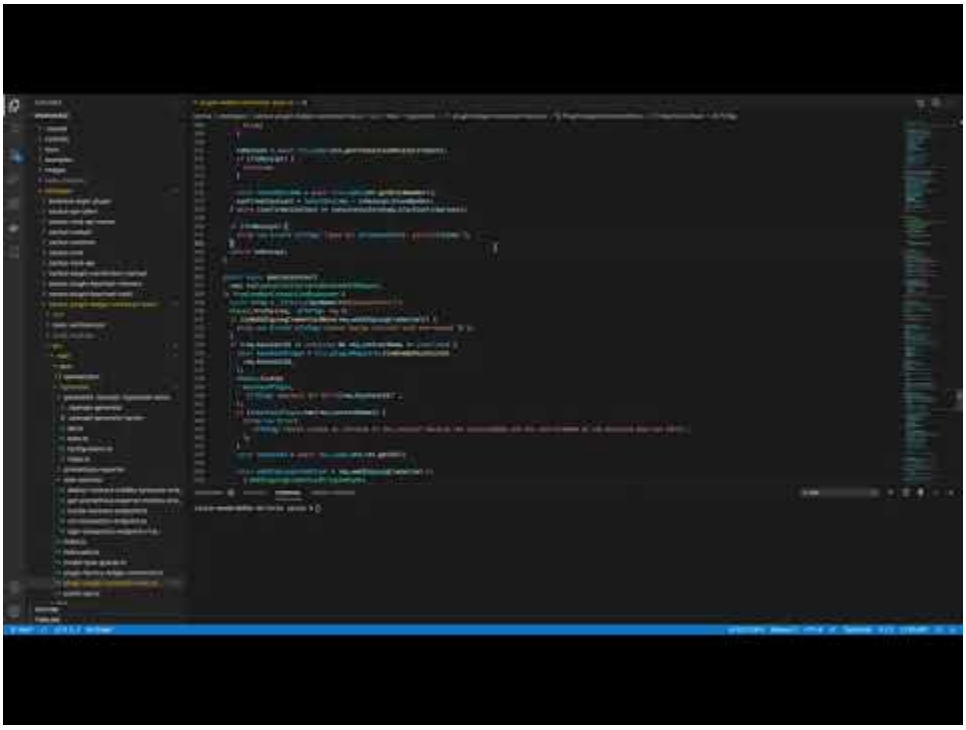
2.1.10 Package structure - OpenAPI



2.1.11 Package structure - Web Services



2.1.12 Package structure - Main and Factory Plugin class



2.1.13 Package structure - Test class

2.2 Fast Developer Flow / Code Iterations

We put a lot of thought and effort into making sure that fast developer iterations can be achieved (please file a bug if you feel otherwise) while working **on** the framework.

If you find yourself waiting too much for builds to finish, most of the time that can be helped by using the `npm run watch` script which can automatically recompile packages as you modify them (and only the packages that you have modified, not everything).

It also supports re-running the OpenAPI generator when you update any `openapi.json` spec files that we use to describe our endpoints.

The `npm run watch` script in action:

2.3 Getting Started

- Install OS level dependencies:
  - Windows Only
    - WSL2 or any virtual machine running Ubuntu 20.04 LTS
  - Git
  - NodeJS v16.13.1, npm v8.1.2 (we recommend using the Node Version Manager (nvm) if available for your OS)

```
nvm install 16.13.1
nvm use 16.13.1
```

  - Yarn
    - `npm run install-yarn` (from within the project directory)
  - Docker Engine
  - Docker Compose
  - OpenJDK (Corda support Java 8 JDK but do not currently support Java 9 or higher)
    - `sudo apt install openjdk-8-jdk-headless`
- Clone the repository

```
git clone https://github.com/hyperledger/cactus.git
```

Windows specific gotcha: `File paths too long` error when cloning. To fix: Open PowerShell with administrative rights and then run the following:

```
git config --system core.longpaths true
```

- Change directories to the project root

```
cd cactus
```

- Run the initial configuration script (can take a long time, 10+ minutes on a low-spec laptop)

```
npm run configure
```

At this point you should have all packages built for development.

You can start making your changes (use your own fork and a feature branch) or just run existing tests and debug them to see how things fit together.

For example you can *run a ledger single status endpoint test* via the REST API with this command:

```
npx tap --ts --timeout=600 packages/cactus-test-plugin-htlc-eth-besu/src/test/typescript/
↪integration/plugin-htlc-eth-besu/get-single-status-endpoint.test.ts
```

You can also start the API server and verify more complex scenarios with an arbitrary list of plugins loaded into Cactus. This is useful for when you intend to develop your plugin either as a Cactus maintained plugin or one on your own.

```
npm run generate-api-server-config
```

Notice how this task created a .config.json file in the project root with an example configuration that can be used a good starting point for you to make changes to it specific to your needs or wants.

The most interesting part of the `.config.json` file is the plugins array which takes a list of plugin package names and their options (which can be anything that you can fit into a generic JSON object).

Notice that to include a plugin, all you need is specify it's npm package name. This is important since it allows you to have your own plugins in their respective, independent Github repositories and npm packages where you do not have to seek explicit approval from the Cactus maintainers to create/maintain your plugin at all.

Once you are satisfied with the `.config.json` file's contents you can just:

```
npm run start:api-server
```

After starting the API server, you will see in the logs that plugins were loaded and that the API is reachable on the port you specified (4000 by default). The Web UI (Cockpit) is disabled by default but can be enabled by changing the property value 'cockpitEnabled' to true and it is reachable through port on the port your config specified (3000 by default).

> You may need to enable manually the CORS patterns in the configuration file. This may be slightly inconvenient, but something we are unable to compromise on despite valuing developer experience very much. We have decided that the software should be `secure by default` above all else and allow for customization/degradation of security as an opt-in feature rather than starting from that state.

At this point, with the running API server, you can

- Test the REST API directly with tools like cURL or Postman
- Develop your own applications against it with the `Cactus API Client(s)`
- Create and test your own plugins

2.3.1 Random Windows specific issues not covered here

We recommend that you use WSL2 or any Linux VM (or bare metal). We test most frequently on Ubuntu 20.04 LTS

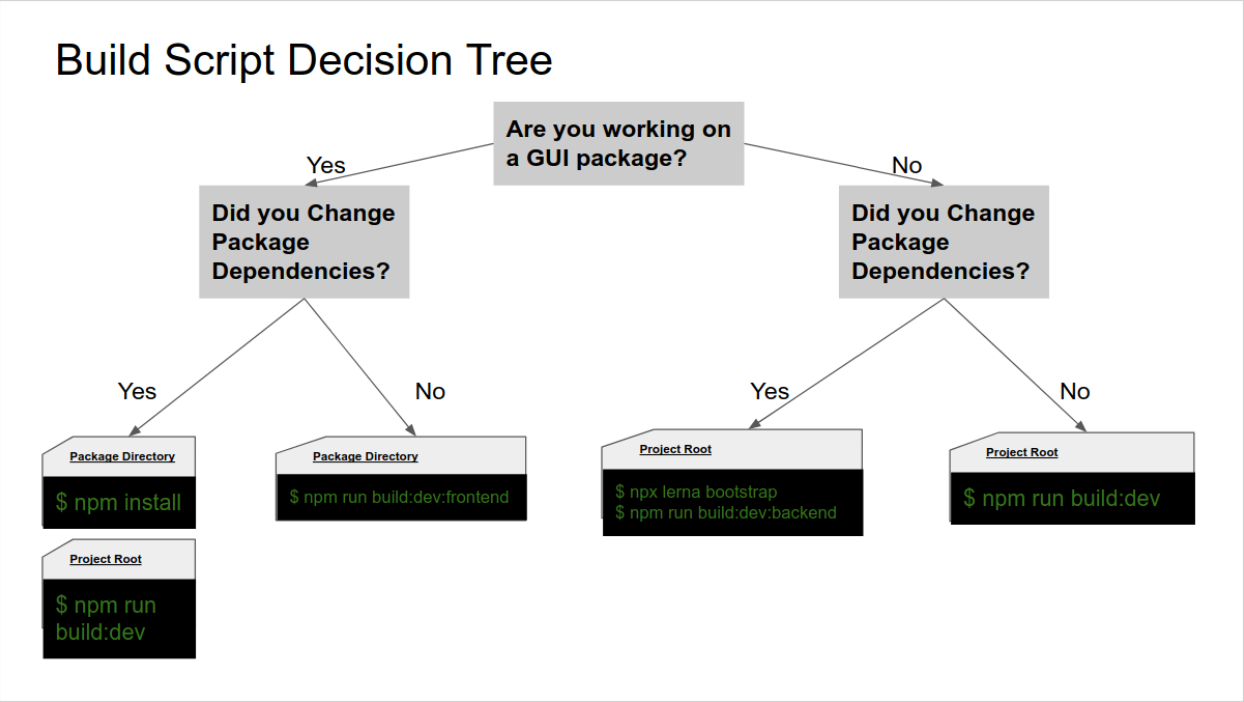
2.4 Build Script Decision Tree

The `npm run watch` script should cover 99% of the cases when it comes to working on Cactus code and having it recompile, but for that last 1% you'll need to get your hands dirty with the rest of the build scripts. Usually this is only needed when you are adding new dependencies (npm packages) as part of something that you are implementing.

There are a lot of different build scripts in Cactus in order to provide contributors fine(r) grained control over what parts of the framework they wish build.

> Q: Why the complexity of so many build scripts?
>
> A: We could just keep it simple with a single build script that builds everything always, but that would be a nightmare to wait for after having changed a single line of code for example.

To figure out which script could work for rebuilding Cactus, please follow the following decision tree (and keep in mind that we have `npm run watch` too)



2.5 Getting into the SSH connection

Upload your public key onto github if not done so already. A public key is necessary to join the ssh connection to use upterm. For a comprehensive guide, see the Generating a new SSH key and adding it to the ssh-agent.

Locate the `ci.yml` within `.github/workflows` and add to the `ci.yml` code listed below:

- name: Setup upterm session uses: lhotari/action-upterm@v1 with: repo-token: ${{ secrets.GITHUB_TOKEN }}

Keep in mind that the SSH upterm session should come after the checkout step (uses: actions/checkout@v2.3.4) to ensure that the CI doesn't hang without before the debugging step occurs. Editing the `ci.yml` will create a new upterm session within `.github/workflows` by adding a new build step. For more details, see the Debug your GitHub Actions by using ssh.

By creating a PR for the edited `ci.yml` file, this will the CI to run their tests. There are two ways to navigate to CIs.

1) Go to the PR and click the `checks` tab
2) Go to the `Actions` tab within the main Hyperledger Cactus Repository

Click on the `CI Cactus workflow`. There should be a new job you've created be listed underneath the `build (ubuntu-20.04)` jobs. Click on the the new job (what's you've named your build) and locate the SSH Session within the `Setup Upterm Session` dropdown. Copy the SSH command that start with `ssh` and ends in `.dev` (ex. ssh **********:**********@uptermd.upterm.dev). Open your OS and paste the SSH command script in order to begin an upterm session.

CHAPTER

THREE

EXAMPLES

This section shows the sample applications that are provisioned by the Hyperledger Cactus.

3.1 Hyperledger Cactus Example - Supply Chain App

3.1.1 Usage

1. Execute the following from:

```
docker run \
  --rm \
  --privileged \
  -p 3000:3000 \
  -p 3100:3100 \
  -p 3200:3200 \
  -p 4000:4000 \
  -p 4100:4100 \
  -p 4200:4200 \
  ghcr.io/hyperledger/cactus-example-supply-chain-app:2021-09-08--docs-1312
```

2. Observe the example application pulling up in the logs
    1. the test ledger containers,
    2. a test consortium with multiple members and their Cactus nodes
3. Wait for the output to show the message `INFO (api-server): Cactus Cockpit reachable http://0.0.0.0:3100`
4. Visit http://0.0.0.0:3100 in your web browser with Javascript enabled

3.1.2 Building and running the container locally

```
# Change directories to the project root

# Build the dockar image and tag it as "scaeb" for supply chain app example backend
DOCKER_BUILDKIT=1 docker build -f ./examples/supply-chain-app/Dockerfile . -t scaeb

# Run the built image with ports mapped to the host machine as you see fit
# The --privileged flag is required because we use Docker-in-Docker for pulling
```

(continues on next page)

(continued from previous page)

```
# up ledger containers from within the container in order to have the example
# be completely self-contained where you don't need to worry about running
# multiple different ledgers jus this one container.
docker run --rm -it --privileged -p 3000:3000 -p 3100:3100 -p 3200:3200 -p 4000:4000 -p␣
↪4100:4100 -p 4200:4200 scaeb
```

3.1.3 Running the Example Application Locally

Make sure you have all the dependencies set up as explained in `BUILD.md`

On the terminal, issue the following commands:

1. `npm run install-yarn`
2. `yarn configure`
3. `yarn start:example-supply-chain`

3.1.4 Debugging the Example Application Locally

On the terminal, issue the following commands (steps 1 to 6) and then perform the rest of the steps manually.

1. `npm run install-yarn`
2. `yarn configure`
3. `yarn build:dev`
4. `cd ./examples/supply-chain-app/`
5. `yarn --no-lockfile`
6. `cd ../../`
7. Locate the `.vscode/template.launch.json` file
8. Within that file locate the entry named `"Example: Supply Chain App"`
9. Copy the VSCode debug definition object from 2) to your `.vscode/launch.json` file
10. At this point the VSCode `Run and Debug` panel on the left should have an option also titled `"Example: Supply Chain App"` which

CHAPTER

FOUR

GOVERNANCE

Hyperledger Cactus is managed under an open governance model as described in the Hyperledger charter. Cactus is led by a set of maintainers, who can be found in the MAINTAINERS.md file.

Maintainers

Cactus is led by the project's maintainers. The maintainers are responsible for reviewing and merging all patches submitted for review, and they guide the overall technical direction of the project within the guidelines established by the Hyperledger Technical Steering Committee (TSC).

Becoming a Maintainer

The project's maintainers will, from time-to-time, consider adding or removing a maintainer. An existing maintainer can submit a change set to the MAINTAINERS.md file. A nominated contributor may become a maintainer by a three-quarters approval of the proposal by the existing maintainers. Once approved, the change set is then merged and the individual is added to (or alternatively, removed from) the maintainers group.

Maintainers may be removed by explicit resignation, for prolonged inactivity (3 or more months), or for some infraction of the code of conduct or by consistently demonstrating poor judgement. A maintainer removed for inactivity should be restored following a sustained resumption of contributions and reviews (a month or more) demonstrating a renewed commitment to the project. We require that maintainers that will be temporarily inactive do so "gracefully" and update other maintainers on their status and time availability rather than appearing to "fall off the face of the earth."

Releases

A majority of the maintainers may decide to create a release of Cactus. Any broader rules of Hyperledger pertaining to releases must be followed. Once the project is mature, there will be a stable LTS (long term support) release branch, as well as the main branch for upcoming new features.

Making Feature/Enhancement Proposals

Code changes that are either bug fixes, direct and small improvements, or things that are on the roadmap (see below) can be issued as PRs in a relatively quick time period, although we recommend creating a Github ticket to track even bugs and small improvements. For more substantial changes, however, a feature/enhancement proposal is required. These proceed through the approval process like typical PRs, and require the same "2 + 1" approval policy for acceptance.

In particular, all contributors to the project should have enough time to voice an opinion on feature/enhancement proposals before they are accepted. So the maintainers will determine some "comment period" between proposal submission and acceptance so that contributors have enough time to voice their opinions.

Significant changes can be marked as such via the predefined label with the same name. This is a tool that helps maintainers identify the most important issues/discussions to be had at any given time through the GitHub web interface.

To easily access the list of significant changes, navigate to the label: https://github.com/hyperledger/cactus/labels/Significant_Change

We also recommend reading our CONTRIBUTING.md file (https://github.com/hyperledger/cactus/blob/main/CONTRIBUTING.md) for more information about contributing.

Approving Pull Requests

Maintainers designated for review are required to review PRs in a timely manner (all circumstances considered, of course). Any pull request must be reviewed by at least two maintainers, and if a PR is submitted by a maintainer, these two reviewers must be different from the original submitter.

The technical requirements for submitting/approving/merging pull requests are further detailed in the CONTRIBUTING.md file where it is laid out in detail how to ensure git commit graph tidiness.

Reviewing Pull Requests

We are strongly committed to processing pull requests from everyone in a fair manner meaning that pull requests are to be reviewed in order of submission. Reviewing PRs in order of submission does not guarantee nor necessitate accepting/merging said PRs in order of submission since some PRs may require lengthy feedback loops while others may pass the muster without any change requests or feedback at all, depending on the nature of the change being proposed. Security related pull requests may be fast tracked even against the "in order of submission" principle if it appears that a vulnerability makes a pull request a time sensitive issue where the sooner we propagate a fix the better it is.

Maintainers Meeting

The maintainers hold regular maintainers meetings, which are open to everyone. The purpose of the maintainers meeting is to plan for and review the progress of releases, and to discuss the technical and operational direction of the project.

Please see the wiki for maintainer meeting details.

One point to mention about meetings is that new feature/enhancement proposals as described above should be presented to a maintainers meeting for consideration, feedback, and acceptance.

Roadmap

The Cactus maintainers are required to maintain a roadmap. There is a technical roadmap, with all of the issues as they directly relate to code, and a more public-friendly roadmap that anyone can digest. The required features to be implemented will be maintained as issues at the official github repository of Cactus with tag string 'for current release' or 'for future release'. The task which is not volunteered to work, will be dispatched to specific contributors following consensus among the majority of maintainers.

The technical roadmap is implicitly derived from the Github "milestones" feature. To access the list of milestones for Cactus use this link: https://github.com/hyperledger/cactus/milestones

Communications

We use the Cactus email list for long-form communications and RocketChat for short, informal announcements and other communications. We encourage all communication, whenever possible, to be public and in the clear (i.e. rather than sending an email directly to a person or two, send it out to the whole list if it pertains to the project).

Future Changes

The governance of Cactus may change as the project evolves. In particular, if the project becomes large, we will incorporate tiered maintainership, with top-level maintainers, subprojects, subproject maintainers, release managers, and so forth. We emphasize that this document is intended to be "living" and will be updated periodically.

We require that changes to this document require a three-quarters approval of the existing maintainers. Note that this may also be changed in the future if deemed necessary.

Attribution

This document is based on the Hyperledger Fabric governance document, with some substantial changes.

CHAPTER

FIVE

CODE OF CONDUCT GUIDELINES

Please review the Hyperledger Code of Conduct before participating and abide by these community standards.

CHAPTER

SIX

CONTRIBUTING

- Git Know How / Reading List
- PR Checklist - Contributor/Developer
- PR Checklist - Maintainer/Reviewer
- Create local branch
  - Directory structure
- Test Automation
  - Summary
  - Test Case Core Principles
- Working with the Code
  - Running/Debugging the tests
    - Running a single test case
    - Running all test cases (unit+integration)
    - Running unit tests only
    - Running integration tests only
    - What is npx used for?
    - What's the equivalent of npx for Yarn?
    - Debugging a test case
  - All-In-One Docker Images for Ledger Connector Plugins
    - Test Automation of Ledger Plugins
  - Building the API Client(S)
  - Adding new dependencies:
  - Reload VSCode Window After Adding Dependencies
  - On Reproducible Builds

Thank you for your interest to contribute to Hyperledger Cactus! :tada:

First things first, please review the Hyperledger Code of Conduct before participating.

There are many ways to contribute to Hyperledger Cactus, both as a user and as a developer.

As a user, this can include:

- Making Feature/Enhancement Proposals
- Reporting bugs

As a developer:

- if you only have a little time, consider picking up a "help-wanted" or "good-first-issue" task
- If you can commit to full-time development, then please contact us on our Rocketchat channel to work through logistics!

6.1 Git Know How / Reading List

This section is for you if you do not know your way around advanced git concepts such as

- rebasing (interactive or otherwise)
- splitting commits/PRs
- when to use and not to use force push

A word on the controversial topic of force pushes: In many git guides you will read that force push is basically forbidden. This is true 99% of the time, BUT if you are the only person working on a branch (which is most of time true for a feature/fix branch of yours that you are planning to submit as a PR) then force pushing is not just allowed but necessary to avoid messy git commit logs. The question you need to ask yourself before force pushing is this: Am I going to destroy someone else's work on the remote branch? If nobody else is working on the branch then the answer is of course no and force push can be used safely. If others are working with you on the branch on the other hand, it is considered polite to ask and warn them in advance prior to force pushing so that they can take the necessary precautions on their side as well.

A handy tool to avoid destroying other's work accidentally is the new(ish) git feature called `--force-with-lease`: Using `git push --force-with-lease` instead of vanilla `--force` is highly recommended: https://softwareengineering.stackexchange.com/a/312710

The rustlang documentation has an excellent write-up and additional links on pretty much everything you need to know. The only difference between their PR requirements and Cactus' is that we do encourage people referencing github issues in commit messages. Quoting the most relevant parts below (and thanks to the Rust maintainers for this).

> Pull requests are the primary mechanism we use to change Rust. GitHub itself has some great documentation on using the Pull Request feature. We use the "fork and pull" model described here, where contributors push changes to their personal fork and create pull requests to bring those changes into the source repository.
>
> Please make pull requests against the main branch.
>
> Rust follows a no merge policy, meaning, when you encounter merge conflicts you are expected to always rebase instead of merge. E.g. always use rebase when bringing the latest changes from the main branch to your feature branch. Also, please make sure that fixup commits are squashed into other related commits with meaningful commit messages.
>
> GitHub allows closing issues using keywords. This feature should be used to keep the issue tracker tidy.

Source: https://github.com/rust-lang/rust/blob/53702a67e2ae8a404169a0329f6a38d73bf7494d/CONTRIBUTING.md#pull-requests

Further reading:

- https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-collaborative-development-models
- https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests

6.2 PR Checklist - Contributor/Developer

To avoid issues in the future, do not install dependencies globally. Ensure all dependencies are kept self-contained.

1. Fork hyperledger/cactus via Github UI
   - If you are using the Git client on the Windows operating system, you will need to enable long paths for git which you can do in PowerShell by executing the command below. To clarify, this may also apply if you are using any Git GUI application on Windows such as `Github Desktop` or others.

     ```
     git config --global core.longpaths true
     ```

2. Clone the fork to your local machine
3. (Optional) Create local branch for minimizing code conflicts when you want to contribute multiple changes regarding different issues in parallel.
4. Complete the desired changes and where possible test locally
   1. You can run the full CI suite on Mac/Linux/WSL by running the script at `./tools/ci.sh`
   2. If you do not have your environment set up for running bash scripts, do not worry, all pull requests will automatically have the same script executed for it when opened. The only downside is the slower feedback loop.
5. Make sure you have set up your git signatures
   1. Note: Always sign your commits using the `git commit -S`
   2. For more information see here
6. Think about/decide on what your commit message will be.
   1. The commit message syntax might be hard to remember at first so you we invite you to use the `npm run commit` command which upon execution presents you with a series of prompts that you can fill out and have your input validated in realtime, making it impossible (or at least much harder) to produce an invalid commit message that the commit lint bot on Github will flag with an error.
      1. The use of this tool described above is entirely optional in case you need a crutch.
      2. Note that running the `npm run commit` command will also attempt to perform the actual commit at the end unless you kill the process with `Ctrl + C` or whatever is your terminal's shortcut for the same action.
      3. The `npm run commit` command will also attempt to sign the produced commit so make sure that it is set up properly prior to using it.
7. Commit your changes
   1. Make sure your commit message follows the formatting requirements (details above) and here: Conventional Commits syntax; this aids in release notes generation which we intend to automate
   2. Be aware that we are using git commit hooks for the automation of certain mundane tasks such as applying the required code style and formatting so your code will be wrapped at 80 characters each line automatically. If you wish to see how your changes will be altered by the formatter you can run the `npm run prettier` command from a terminal or install an IDE extension for the `Prettier` tool that can do the same (VSCode has one that is known to work).
8. Ensure your branch is rebased onto the `upstream` main branch where `upstream` is fancy git talk for the main Cactus repo on Github (the one you created your fork from).

1. **Do not** duplicate your pull request after it has been reviewed. Duplication here means closing the existing PR and then opening a brand new one which does not contain the review history anymore. If you encounter issues with version control that you do not know how to solve the maintainers will be happy to assist to ensure that you do not need to open a new pull request from scratch.
    1. The only exception from the rule above is if you mistakenly named your branch to contain special characters and somehow ended up in a state where it has become impossible to push changes to the remote due to this (which has happened before with branch names like `refactor(core-api): x` that had to be renamed to `refactor-core-api-x` and then a new PR had to be created in that case because GitHub does not let you rename the remote branch that your pull request is tied to)
2. If you are having trouble, there are many great resources out there (so we will not write another here).
    1. If you are having trouble locating a suitable guide specifically on the mechanics of rebasing, we can recommend this one. Thanks to Rafael for the link!
    2. If you went through that tutorial and still not quite sure what's up, give this one a shot as well: https://about.gitlab.com/blog/2020/11/23/keep-git-history-clean-with-interactive-rebase/
3. If merge conflicts arise, you must fix these at rebase time since omitting this step does not magically make the conflicts go away, just pushes it over the fence to the maintainer who will attempt to merge your pull request at a later point in time.
4. If the above happens, at that point said maintainer will most likely ask you (if not already) to perform the rebase anyway since as the author of a change you are best positioned to resolve any conflicts on the code level. Occassionally maintainers may do the merge/conflict resolution themselves, but do not count on this nor try to make a habit out of relying on the potential kindness.
5. After successful rebasing, take another look at your commit(s). Ideally there should be just one in each pull request, but also on the other hand each commit should be as small, simple and self contained as possible, so there can be cases where it makes sense to submit a PR with multiple commits if for example you also had to change something in the test tooling while implementing a feature (in which case there could be a commit for the feature itself and another for the necessary changes to the test tooling package). What we respectfully ask though is that you try to avoid these situations and submit most of your PRs with a single, self contained commit that does not touch multiple things. This significantly reduces the cognitive load required to review the changes which in turn makes everyone happier: the maintainers will have an easier job reviewing, which means they'll be doing it faster which will (probably) cause you joy in turn.

9. Push your changes to your main (or whatever you named your feature branch, that is entirely up to you at the end of the day)
10. Initiate a pull request from your fork to the base repository
11. Remember: Opening a pull request is like saying "Hey maintainers, I have this change finalized and ready for you to spend time on reviewing it." The word `finalized` here is understood to imply that you are not planning on doing any more changes on the branch apart from when being asked to by the reviewers.
12. It is perfectly acceptable to open a pull request and mark it as `draft` (a GitHub feature) which then signals to the maintainers that if they have time, they are welcome to look at the change, but it may or may not be in its final form yet so you are not responsible for potential loss of time on their end if the review has to be performed multiple times on account of changes. Once you promote your draft PR to a real one, the comments from the point above apply however.
13. If your pull request contains a significant change, we recommend that you apply the similarly named github label on in it as well. It is okay if you do not do this, if we detect that the change is indeed significant, we will apply the label. If you do it in advance however, it will probably speed up the proceedings by removing one communication roundtrip from the review process of your pull request.
14. Await CI, DCO & linting quality checks, as well as any feedback from reviewers

15. If you need to update your pull request either because you discovered an issue or because you were asked to do so we ask that you:
16. try to add the change in a way that does not produce additional commits on the PR but instead do an `git commit --amend --signoff` on your local branch and then a force push to the remote branch of yours (the PR essentially). Again, if the change you are doing does not fit within any one of the existing commits of your PR, then it is justified to add a new commit and this is up to your discretion (maintainers may respectfully ask you to squash if they see otherwise)
17. The rule of thumb for any and all things in git/Cactus is to maintain a clean, tidy commit log/history that enables everyone to easily look up changes and find accurate answers to the basic questions of `Who? / What? / When / Why?`. If you have ever been in a situation when you tried to figure out the original point a bug was introduced (and tried to figure out why the offending change was made in the first place) and the git blame just lead you to a 10 megabyte large patch with the message 'merge xyz', then you know exactly what it is we are trying to avoid here. :-)

6.3 PR Checklist - Maintainer/Reviewer

Ensure all the following conditions are met (on top of you agreeing with the change itself)

1. All automated checks that are not explicitly called out here are also passing/green.
2. Branch is rebased onto main and there are no dangling/duplicate commits.
3. Commits appear simple and self contained. Simple is always relative to the mangitude of the change itself of course. A 1k line change can still be simple if all it does is rename some commonly used variable in each place its being used.
4. If the contributors are having trouble with git basic functionality such as rebasing / force pushing, DCO, do your best to help them out, when in doubt feel free to reach out to Peter (who is the one insisting an all these git rules so he deserves to be the primary contact for all git related issues).
    1. Remember that we want to foster a welcoming community so if someone is new to git try to be extra patient with them on this front.
5. Ensure the commit messages are according to the standard format.
    1. Remember that if you select 'squash' on the Github UI when accepting the pull request, Github will (by default) offer up the title of the pull request as the new commit message for your squash commit. This is not good unless the title happens to be a valid commit message, but in the likely event of it not being as such, you must take special care to type in a valid commit message right there and then on the Github UI.
    2. To avoid the hassle/potential issues with the above, it is recommended that you always use 'rebase' when accepting a pull request even if there are multiple commits that you'd otherwise like to see squashed.
    3. If you are adamant that you do not want to merge a PR with multiple commits, that is completely understandable and fair game. The recommended approach there is to ask the contributor to break the pull request up to multiple pull requests by doing an interactive rebase on their branch and cherry picking/re-ordering things accordingly. This is a fairly advanced git use case so you might want to help them out with it (or ask Peter who is the one constantly nagging everyone about these git rules…)

To protect the Hyperledger Cactus source code, GitHub pull requests are accepted from forked repositories only. There are also quality standards identified and documented here that will be enhanced over time.

6.4 Create local branch

Whenever you begin work on a new feature or bugfix, it's important that you create a new branch.

1. Clone your fork to your local machine
2. Setup your local fork to keep up-to-date (optional)

```
# Add 'upstream' repo to list of remotes
git remote add upstream https://github.com/hyperledger/cactus.git

# Verify the new remote named 'upstream'
git remote -v

# Checkout your main branch and rebase to upstream.
# Run those commands whenever you want to synchronize with main branch
git fetch upstream
git checkout main
git rebase upstream/main
```

3. Create your branch.

```
# Checkout the main branch - you want your new branch to come from main
git checkout main

# Create a new branch named `<newfeature>` (give simple informative name)
git branch <newfeature>
```

4. Checkout your branch and add/modify files.

```
git checkout <newfeature>
git rebase main
# Happy coding !
```

5. Commit changes to your branch.

```
# Commit and push your changes to your fork
git add -A
git commit -s -m "<type>[optional scope]: <description>"
git push origin <newfeature>
```

6. Once you've committed and pushed all of your changes to GitHub, go to the page for your fork on GitHub, select your development branch, and click the pull request button.
7. Repeat step 3 to 6 when you need to prepare posting new pull request.

NOTE: Once you submitted pull request to Cactus repository, step 6 is not necessary when you made further changes with `git commit --amend` since your amends will be sent automatically.

NOTE: You can refer original tutorial 'GitHub Standard Fork & Pull Request Workflow'

6.4.1 Directory structure

Whenever you begin to use your codes on Hyperledger Cactus, you should follow the directory strecture on Hyperledger Cactus. The current directory structure is described as the following:

- contrib/ : Contributions from each participants, which are not directly dependent on Cactus code.
  - Fujitsu-ConnectionChain/
  - Accenture-BIF/
- docs/
  - API/
    - business-logic-plugin.md
    - ledger-plugin.md
    - routing-interface.md
- examples/
  - example01-car-trade/
    - src/
- plugins/
  - business-logic-plugin/
    - lib/ : libraries for building Business Logic Plugin
  - ledger-plugin/ : Codes of Ledger Plugin
    - (ledger-name)/ : Including the ledger name (e.g. Ethereum, Fabric, . . .)
      - verifier/
      - src/ : Source codes of Verifier on Ledger Plugin
      - unit-test/ : Unit test codes of Verifier on Ledger Plugin (single driver / driver and docker env / . . .)
      - validator/
      - src/ : Source codes of Validator on Ledger Plugin
      - unit-test/ : Unit test codes of Validator on Ledger Plugin (single driver / driver and docker env / . . .)
  - routing-interface/
- whitepaper/
- test/
  - docker-env/
  - kubernetes-env/

6.5 Test Automation

Mantra: Testable code is maintainable code

6.5.1 Summary

We are all about automating the developer flow wherever possible and a big part of this is automated testing of course.

Whenever contributing a change it is important to have test coverage for the specific change that you are making. This is especially important for bugs and absolutely essential for security related changes/fixes.

Writing testable code is very important to us as not doing so can (and will) snowball into an avalanche of technical debt that will eventually destroy code quality and drive people away who would otherwise be happy to contribute and use the software. So, we want to make sure that does not happen with all this.

This also means that occassionally, when making a change that looks simple on the surface you may find that the reviewer of your pull request asks you to do additional, seemingly unrelated changes that have nothing to do with the actual feature/bug that you just implemented/fixed, but instead are designed to ensure that tests can be written for it or for related code.

This can feel like a chore (because it is) but we respectfully ask everyone to try their best in accomodating this because it really helps steering the ship on the long run.

One of the simplest examples for the above is when you have a class that does something, anything, and it depends on some shared resource to achieve it. The shared resource can be the file system or a network port that is open for TCP connections for example. You can implement your class hardcoding the port number and functionally it will be correct (if you did that part right) *BUT* if your class does not allow for the customization of said port through the constructor or a setter method, then one of our more obsessive maintainers (like Peter) will immediately be onto you asking for a change so that the port can be customized at runtime, allowing test cases to pass in port 0 that makes the test executable in parallel with other tests without being flaky.

If you oppose this idea, said maintainers will happily refer you to this writing or conjure up an entirely new essay right there on the pull request.

6.5.2 Test Case Core Principles

There are other principles specific per unit and integration tests, but the list below applies to all tests regardless of their nature.

`All test cases must be...`

- Self contained programs that can be executed on their own if necessary
  - This ensures that if you are iterating on a single test case while trying to make it pass, you will always have the freedom to run just that one test instead of running the full suite of which the execution time will grow rapidly as we add test coverage, so, better nip that in the bud with this principle.
- Excluded from the public API surface of the package they are in by ensuring that the test classes/types/interfaces are NOT exported through the `public-api.ts` file of that particular package.
  - The only exception from this is if a package is itself designed for tests for which a delightful example is the `test-tooling` package which as the name suggests is entirely designated for providing utilities for writing tests and therefore in the case of this package it is allowed and even expected that it will expose test related classes/types in it's public API surface. Do note however that indirectly the principle still applies, meaning that any package must not depend on the `test-tooling` package as an npm `dependency` but rather it must declare it as a `devDependency` in the relevant section of the `package.json` file.
- Compatible with the TestAnythingProtocol

  - The NodeJS implementation of said protocol is in the `node-tap` npm package:
  - Assertions API of Node TAP
  - Simplest possible test case:

```
const test, { Test } = require("tape");
import * as publicApi from "../../../main/typescript/public-api";

test("Module can be loaded", (t: Test) => {
   t.ok(publicApi);
   t.end(); // yaay, test coverage
});
```

  - An end to end test case showcasing everything in action that is being preached in this document about test automation

- Focus/verify a single bug-fix/feature/etc.
- Clearly separated from non-test (aka `main`) source code. This means in practice that we ask that your test cases are either in the
  1. `./src/test/...` tree of the package that you are testing OR
  2. your test cases are in the `./src/test/...` tree (yes same name) BUT in an entirely separate package if the dependencies necessitate so. An example to when you would need a separate testing package is if you are developing a ledger plugin that has REST API endpoints shipping with it and you wish to verify in a test that the plugin can be loaded to the `ApiServer` and then called via the web service/SDK. In this case, you cannot place your test case in the ledger plugin's package because you want to avoid having to pull in the API server as a dependency of your ledger plugin package (to ensure that there will be no circular dependencies).
- Executable with unlimited parallelism (so if I have a 128 test cases and run all of them in parallel on my new 128 CPU core computer, then every single test case runs at the same time)
  - This is important because it weeds out flakyness and hardcoded references to shared resources (Remember the rant about network ports in the previous section?)
  - *BUT* it is also very important because we (as in humanity) spent the last decade making the average CPUs ship with more and more cores as increasing frequency failed in the late 2000s as a performance increasing strategy.

    What this means is that to utilize the average consumer laptop that most people will have for development, you will need your test cases to run in parallel which will save time for everyone working on the code and faster turnaround times make for a better developer experience which makes for a happier community around our open source project. It's all connected. ;-)
- Test cases don't depend on code outside of the `./src/*` directory trees of the packages.
  - Do not depend on any of the example code in your test cases.
  - If you need to import code that is not JS/JSON/TS you can still do so via the Typescript compiler's relevant feature that allows importing arbitrary files.

6.6 Working with the Code

There are additional details about this in the BUILD.md file in the project root as well.

We use Lerna for managing the monorepo that is Cactus.

> We heavily rely on Docker for testing the ledger plugins.

6.6.1 Running/Debugging the tests

Make sure to have the build succeed prior to attempting to run the tests. If you just checked out the project, it is best to just to just run the CI script which will do a full build and run all tests for you. If it fails you can open a bug in the issue tracker.

Assuming you have built the sources, below are the different methods to run the tests:

Running a single test case

You execute unit and integration tests in the same way, but here are examples for both them separately anyway:

- An integration test:

```
npx tap --ts --timeout=600 packages/cactus-test-plugin-consortium-manual/src/test/
↪typescript/integration/plugin-consortium-manual/get-consortium-jws-endpoint.test.
↪ts
```

- A unit test:

```
npx jest packages/cactus-common/src/test/typescript/unit/objects/get-all-method-
↪names.test.ts
```

Running all test cases (unit+integration)

```
npm run test:all
```

Running unit tests only

```
npm run test:unit
```

Running integration tests only

```
npm run test:integration
```

What is npx used for?

`npx` is a standard top level binary placed on the path by NodeJS at installation time. We use it to avoid having to place every node module (project dependencies) on the OS path or to install them globally (`npm install some-pkg -g`)

Read more about npx here: https://blog.npmjs.org/post/162869356040/introducing-npx-an-npm-package-runner

What's the equivalent of npx for Yarn?

Yarn itself. E.g. `npx lerna clean` becomes `yarn lerna clean`.

Debugging a test case

Open the `.vscode/template.launch.json` file and either copy it with a name of `launch.json` (if you don't already have such a file) or just cherry pick the example Visual Studio Code debug tasks that you'd like to use. For debugging a single test case, you need the debug task from the template launch.json file that is called `TAP: Current TS Test File`. Prior to running that debug task you must have your VSCode editor opened to the test file that you wish to run. Breakpoints will work as long as you are debugging code in the same package.

> Source map support is partial at this point but actively being worked on.

6.6.2 All-In-One Docker Images for Ledger Connector Plugins

If you are working on a new ledger connector you'll need an `all-in-one` docker image as well, which will allow the expected level of test automation. If your chosen ledger's maintainers provide an adequate docker image, then you might not need to develop this yourself, but this is rarely the case so YMMV.

To see an existing set of examples for `besu` and `quorum` images take a peek at the `tools/docker/besu-all-in-one` and `tools/docker/quorum-all-in-one` folders. These produce the `ghcr.io/hyperledger/cactus-besu-all-in-one` and `ghcr.io/hyperledger/cactus-quorum-all-in-one` images respectively. Both of these are used in the test cases that are written for the specific ledger connector plugins at:

- `packages/cactus-test-plugin-ledger-connector-quorum/src/test/typescript/integration/plugin-ledger-connector-quorum/deploy-contract/deploy-contract-via-web-service.test.ts`
- `packages/cactus-plugin-ledger-connector-besu/src/test/typescript/integration/plugin-ledger-connector-besu/deploy-contract/deploy-contract-from-json.test.ts`

The specific classes that utilize the `all-in-one` images can be found in the `test-tooling` package under these paths:

- `packages/cactus-test-tooling/src/main/typescript/besu/besu-test-ledger.ts`
- `packages/cactus-test-tooling/src/main/typescript/quorum/quorum-test-ledger.ts`

Test Automation of Ledger Plugins

Ledger plugin tests are written the same way as any other test (which is difficult to achieve, but we thrive to get it done).

The only difference between a ledger connector plugin test case and any unit test is that the ledger connector plugin's test case will pull up a docker container from one of the `all-in-one` images that we maintain as part of Cactus and then use that `all-in-one-*` container to verify things such as the ability of the ledger connector plugin to deploy a contract to said ledger.

As a generic best practice, the test cases should never re-use any `all-in-one` ledger container for the execution of multiple test cases because that will almost surely lead to flaky/unstable test cases over the long run and needless

complexity, ordering dependencies and so on. It is recommended that if you have two test cases for a ledger connector plugin, they both pull up a newly created container from scratch, execute the test scenario and then tear down and delete the container completely.

An example for a ledger connector plugin and it's test automation implemented the way it is explained above: `packages/cactus-test-plugin-ledger-connector-quorum/src/test/typescript/integration/plugin-ledger-connector-quorum/deploy-contract/deploy-contract-via-web-service.test.ts`

> This test case is also an example of how to run an ApiServer independently with a single ledger plugin which is how the test case is set up to begin with.

Another option if you want to perform some tests manually is to run the API server with a configuration of your choice:

```
# Starting from the project root directory

chmod +x ./packages/cactus-cmd-api-server/dist/lib/main/typescript/cmd/cactus-api.js

./packages/cactus-cmd-api-server/dist/lib/main/typescript/cmd/cactus-api.js --config-
↪file=.config.json
```

You can run this test case the same way you would run any other test case (which is also a requirement in itself for each test case):

```
npx tap --ts --timeout=600 packages/cactus-test-plugin-ledger-connector-quorum/src/test/
↪typescript/integration/plugin-ledger-connector-quorum/deploy-contract/deploy-contract-
↪via-web-service.test.ts
```

You can specify an arbitrary set of test cases to run in a single execution via glob patterns. Examples of these glob patterns can be observed in the root directory's `package.json` file which has npm scripts for executing all tests with a single command (the CI script uses these):

```
"test:all": "tap --ts --jobs=1 --timeout=600 \"packages/cactus-*/src/test/typescript/
↪{unit,integration}/\"",
"test:unit": "tap --ts --timeout=600 \"packages/cactus-*/src/test/typescript/unit/\"",
"test:integration": "tap --ts --jobs=1 --timeout=600 \"packages/cactus-*/src/test/
↪typescript/integration/\""
```

Following a similar pattern if you have a specific folder where your test cases are, you can run everything in that folder by specifying the appropriate glob patterns (asterisks and double asterisks as necessary depending on the folder being a flat structure or with sub-directories and tests nested deep within them).

For example this can work as well:

```
# Starting from the project root
cd packages/cactus-test-plugin-ledger-connector-quorum/src/test/typescript/integration/
↪plugin-ledger-connector-quorum
npx tap --ts --jobs=1 --timeout=600 \"./\"
```

> Be aware that glob patterns need quoting in some operating system's shell environments and not necessarily on others. In the npm scripts Cactus uses we quote all of them to ensure a wider shell compatibility.

6.6.3 Building the API Client(S)

You do not need to do anything special to have the API Client sources generated and compiled. It is all part of the `npm run build:dev:backend` task which you can run yourself or as part of the CI script (`./tools/ci.sh`).

The API client code is automatically generated from the respective `openapi.json` file of each package that exposes ay web serices (REST/SocketIO/gRPC/etc.) and can be dependend on by other packages where applicable. There's a dedicated `@hyperledger/cactus-api-client` package that is meant to contain common functionality among the rest of API clients. The concept here is similar to abstract classes and their sub-class implementations.

Each `openapi.json` produces its own API client via the code generator that also contains relevant model definitions, such as interfaces describing the request/response bodies of all possible operations and validation constraints as well.

The API clients are designed to be a universal components, meaning that it runs just fine in browser and also NodeJS environments. This is very important as we do not wish to maintain two (or more) separate API client codebases for the various platforms and we also want as much of it being generated automatically as possible (currently this is close to 100%).

6.6.4 Adding new dependencies:

Example:

```
# Adds "got" as a dependency to the cactus common package
# Note that you must specify the fully qualified package name as present in
# the package.json file
yarn workspace @hyperledger/cactus-common add got --save-exact
```

You need to know which package of the monorepo will be using the package and then run the `yarn workspace` command with an additional parameters specifying the package name and the dependency name. See Yarn Workspaces Documentation for the official Yarn documentation for further details and examples.

After adding new dependencies, you might need to Reload VSCode Window After Adding Dependencies

Always specify the `--save-exact` when installing new dependencies to ensure reproducible builds

6.6.5 Reload VSCode Window After Adding Dependencies

If you added a new dependency and VSCode is showing an error when you try to import it, then sometimes the issue is just a matter of nudging VSCode to reload the Typescript definitions from scratch so that it "notices" the new dependency you just added.

The recommended way of doing this is by hitting the `F1` key (or whatever you have bound the command menu to) and then searching and selecting `Developer: Reload Window` As a simpler alternative you can also just quit and relaunch the VSCode application of course.

6.6.6 On Reproducible Builds

As a best practice, any given revision (commit hash) stored in version control should produce the exact same build artifacts regardless of when or where the build was performed. This can only be achieved if npm dependency versions are locked down instead of being automatically upgraded by npm (which makes the build time and machine dependent).

Bottom line: Do not use the the ^, ~ and * syntax elements while declaring your npm dependencies.

Further details:

- https://reproducible-builds.org/
- https://spin.atomicobject.com/2016/12/16/reproducible-builds-npm-yarn/

CHAPTER

SEVEN

MAINTAINERS

Active Maintainers

Name	GitHub	Chat
Jonathan Hamilton	jonathan-m-hamilton	JHamilton
Izuru Sato	izuru0	izuru
Peter Somogyvari	petermetz	peter_somogyvari
Takuma TAKEUCHI	takeutak	takeutak
Jagpreet Singh Sasan	jagpreetsinghsasan	jagpreetsinghsasan

CHAPTER

EIGHT

WHITEPAPER

8.1 Hyperledger CactusWhitepaper

8.1.1 Version 0.1 (Early Draft)

8.2 Contributors

Contributors/Reviewers	Email
Hart Montgomery	hmontgomery@us.fujitsu.com
Hugo Borne-Pons	hugo.borne-pons@accenture.com
Jonathan Hamilton	jonathan.m.hamilton@accenture.com
Mic Bowman	mic.bowman@intel.com
Peter Somogyvari	peter.somogyvari@accenture.com
Shingo Fujimoto	shingo_fujimoto@fujitsu.com
Takuma Takeuchi	takeuchi.takuma@fujitsu.com
Tracy Kuhrt	tracy.a.kuhrt@accenture.com
Rafael Belchior	rafael.belchior@tecnico.ulisboa.pt

8.3 Document Revisions

Date of Revision	Description of Changes Made
February 2020	Initial draft

- 5.1 Deployment Scenarios
  - 5.1.1 Production Deployment Example
  - 5.1.2 Low Resource Deployment Example
- 5.2 System architecture and basic flow
  - 5.2.1 Definition of key components in system architecture
  - 5.2.2 Bootstrapping Cactus application
  - 5.2.3 Processing Service API call
- 5.3 APIs and communication protocols between Cactus components
  - 5.3.1 Cactus Service API
    - Open Endpoints
    - Restricted Endpoints
  - 5.3.2 Ledger plugin API
  - 5.3.3 Exection of "business logic" at "Business Logic Plugin"
- 5.4 Technical Architecture
  - 5.4.1 Monorepo Packages
    - 5.4.1.1 cmd-api-server
    - 5.4.1.1.1 Runtime Configuration Parsing and Validation
    - 5.4.1.1.2 Configuration Schema - API Server
    - 5.4.1.1.3 Plugin Loading/Validation
    - 5.4.1.2 core-api
    - 5.4.1.3 API Client
    - 5.4.1.4 keychain
    - 5.4.1.5 tracing
    - 5.4.1.6 audit
    - 5.4.1.7 document-storage
    - 5.4.1.8 relational-storage
    - 5.4.1.9 immutable-storage
  - 5.4.2 Deployment Diagram
  - 5.4.3 Component Diagram
  - 5.4.4 Class Diagram
  - 5.4.5 Sequence Diagram - Transactions
- 5.5 Transaction Protocol Specification
  - 5.5.1 Handshake Mechanism
  - 5.5.2 Transaction Protocol Negotiation
- 5.6 Plugin Architecture
  - 5.6.1 Ledger Connector Plugins

8.4 1. Abstract

Blockchain technologies are growing in usage, but fragmentation is a big problem that may hinder reaching critical levels of adoption in the future.

We propose a protocol and it's implementation to connect as many of them as possible in an attempt to solve the fragmentation problem by creating a heterogeneous system architecture 1.

8.5 2. Introduction to Blockchain Interoperability

There are two inherent problemsa that have to be solved when connecting different blockchains:

- How to provide a proof of the networkwideb ledger state of a connected blockchain from the outside?
- How can other entities verify a given proof of the state of a connected blockchain from the outside?

The `Cactus` consortium operates for each connected blockchain a group of validator nodes, which as a group provides the proofs of the state of the connected ledger. The group of validator nodes runs a consensus algorithm to agree on the state of the underlying blockchain. Since a proof of the state of the blockchain is produced and signed by several validator nodesc with respect to the rules of the consensus algorithm, the state of the underlying blockchain is evaluated networkwide.

The validator nodes are ledger-specific plug-ins, hence a smart contract on the connected blockchain can enable the ledger-specific functionalities necessary for a validator node to observe the ledger state to finalize a proof. The validator nodes are easier discovered from the outside than the blockchain nodes. Hence, the benefit of operating the `Cactus` network to enable blockchain interoperability relies on the fact that for any cross-blockchain interaction the same type of validator node signatures can be used. That means, the cross-blockchain interaction can be done canonically with the validator node signatures in `Cactus` rather than having to deal with many different ledger-specific types of blockchain node signatures.

Outside entities (verifier nodes) can request and register the public keys of the validator nodes of a blockchain network that they want to connect to. Therefore, they can verify the signed proofs of the state of the blockchain since they have the public keys of the validator nodes. This implies that the verifier nodes trust the validator nodes as such they trust the `Cactus` consortium operating the validator nodes.

Figure description: V1, V2, and V3 are validator nodes which provide proofs of the underlying blockchain network through ledger-specific plug-ins. V1, V2, and V3 run a consensus algorithm which is independent of the consensus algorithm run by the blockchain network nodes N1, N2, N3, N4, and N5.

8.5.1 2.1 Terminology of Blockchain Interoperability

This section acts as a building block to describe the different flavors of blockchain interoperability. Business use cases will be built on these simple foundation blocks leveraging a mix of them simultaneously and even expanding to several blockchains interacting concurrently.

2.1.1 Ledger Object Types

To describe typical interoperability patterns between different blockchains, the types of objects stored on a ledger have to be distinguished. The following three types of objects stored on a ledger are differentiated as follows:

- FA: Fungible asset (value token/coin) – cannot be duplicated on different ledgers
- NFA: Non-fungible asset – cannot be duplicated on different ledgersd
- D: Data – can be duplicated on different ledgers

Difference between a fungible (FA) and a non-fungible asset (NFA)

A fungible asset is an asset that can be used interchangeably with another asset of the same type, like a currency. For example, a 1 USD bill can be swapped for any other 1 USD bill. Cryptocurrencies, such as ETH (Ether) and BTC (Bitcoin), are FAs. A non-fungible asset is an asset that cannot be swapped as it is unique and has specific properties. For example, a car is a non-fungible asset as it has unique properties, such as color and price. CryptoKitties are NFAs as well. There are two standards for fungible and non-fungible assets on the Ethereum network (ERC-20 Fungible Token Standard and ERC-721 Non-Fungible Token Standard).

Difference between an asset (FA or NFA) and data (D)

Unicity applies to FAs and NFAs meaning it guarantees that only one valid representation of a given asset exists in the system. It prevents double-spending of the same token/coin in different blockchains. The same data package can have several representations on different ledgers while an asset (FA or NFA) can have only one representation active at any time, i.e., an asset exists only on one blockchain while it is locked/burned on all other blockchains. If fundamental disagreement persists in the community about the purpose or operational upgrades of a blockchain, a hard fork can split a blockchain creating two representations of the same asset to coexist. For example, Bitcoin split into Bitcoin and Bitcoin Cash in 2017. Forks are not addressing blockchain interoperability so the definition of unicity applies in a blockchain interoperability context. A data package that was once created as a copy of another data package might divert from its original one over time because different blockchains might execute different state changes on their data packages.

2.1.2 Blockchain Interoperability Types

Blockchain interoperability implementations can be classified into the following types:

- Ledger transfer:

An asset gets locked/burned on one blockchain and then a representation of the same asset gets released in the other blockchaine. There are never two representations of the same asset alive at any time. Data is an exception since the same data can be transferred to several blockchains. There are one-way or two-way ledger transfers depending on whether the assets can be transferred only in one direction from a source blockchain to a destination blockchain or assets can be transferred in and out of both blockchains with no designated source blockchain and destination blockchain.

- Atomic swapf:

A write transaction is performed on Blockchain A concurrently with another write transaction on blockchain B. There is no asset/data/coin leaving any blockchain environment. The two blockchain environments are isolated but, due to the blockchain interoperability implementation, both transactions are committed atomically. That means either both transactions are committed successfully or none of the transactions are committed successfully.

- Ledger interactionf:

An actiong happening on Blockchain A is causing an action on Blockchain B. The state of Blockchain A causes state changes on Blockchain B. There are one-way or two-way ledger interactions depending on whether only the state of one of the blockchains can affect the state on the other blockchain or both blockchain states can affect state changes on the corresponding other blockchain.

- Ledger entry point coordination:

This blockchain interoperability type concerns end-user wallet authentication/ authorization enabling read and write operations to independent ledgers from one single entry point. Any read or write transaction submitted by the client is forwarded to the corresponding blockchain and then committed/executed as if the blockchain would be operate on its own.

The ledger transfer has a high degree of interference between the blockchains since the livelihood of a blockchain can be reduced in case too many assets are locked/burned in a connected blockchain. The ledger interaction has a high degree of interference between the blockchains as well since the state of one blockchain can affect the state of another blockchain. Atomic swaps have less degree of interference between the blockchains since all assets/data stay in their respective blockchain environment. The ledger entry point coordination has no degree of interference between the blockchains since all transactions are forwarded and executed in the corresponding blockchain as if the blockchains would be operated in isolation.

Figure description: One-way ledger transfer

Figure description: Two-way ledger transfer

Figure description: Atomic swap

Figure description: Ledger interaction

Figure description: Ledger entry point coordination

Legend:

Object 1

State of object *O1* at the beginning of transaction

State of object *O2* at the beginning of transaction

State of object *O1* at the end of transaction

State of object *O2* at the end of transaction

O1end = Representation of object *O1* at the end of transaction in another blockchain

O2end = Representation of object } *O2* at the end of transaction in another blockchain

Transaction 1

Transaction 2

T1 = Representation of transaction 1 (T1) in another blockchain

T1 = Representation of transaction 2 (T2) in another blockchain

Event 1

Event 2 depends on *O1* or *T1* or *E1*

Transaction 2 depends on *O1* or *T1* or *E1*

2.1.3 Burning or Locking of Assets

To guarantee unicity, an asset (NFA or FA) has to be burned or locked before being transferred into another blockchain. Locked assets can be unlocked in case the asset is retransferred back to its original blockchain, whereas the burning of assets is an irreversible process. It is worth noting that locking/burning of assets is happening during a ledger transfer but can be avoided in use cases where both parties have wallets/accounts on both ledgers by using atomic swaps instead. Hence, most cryptocurrency exchange platforms rely on atomic swaps and do not burn FAs. For example, ordinary coins, such as Bitcoin or Ethereum, can only be generated by mining a block. Therefore, Bitcoin or Ethereum exchanges have to rely on atomic swaps rather than two-way ledger transfers because it is not possible to create BTC or ETH on the fly. In contrast, if the minting process of an FA token can be leveraged on during a ledger transfer, burning/locking of an asset becomes a possible implementation option, such as in the ETH token ledger transfer from the old PoW chain (Ethereum 1.0) to the PoS chain (aka Beacon Chain in Ethereum 2.0). Burning of assets usually applies more to tokens/coins (FAs) and can be seen as a donation to the community since the overall value of the cryptocurrency increases.

Burning of assets can be implemented as follows:

- Assets are sent to the address of the coinbase/generation transactionh in the genesis block. A coinbase/generation transaction is in every block of blockchains that rely on mining. It is the address where the reward for mining the block is sent to. Hence, this will burn the tokens/coins in the address of the miner that mined the genesis block. In many blockchain platforms, it is proven that nobody has the private key to this special address.
- Tokens/Coins are subtracted from the user account as well as optionally from the total token/coin supply value.

8.5.2 2.2 Footnotes (Introduction)

a: There is an alternative approach for an outside entity A to verify the state of a connected blockchain if this connected blockchain uses Merkle Trees to store its blockchain state. An outside entity A can store the Merkle Tree roots from the headers of committed blocks of a connected blockchain locally to verify any state claims about the connected blockchain. Any untrusted entity can then provide a state of the connected blockchain, such as a specific account balance on the connected blockchain, because the outside entity A can act as a lightweight client and use concepts like simple payment verification (SPV) to verify that the state claim provided by the untrusted entity is valid. SPV can be done without checking the entire blockchain history. Polkadot uses this approach in its Relay Chain and the BTCRelay on the Ethereum blockchain uses this approach as well. Private blockchains do not always keep track of their state through Merkel trees and signatures produced by nodes participating in such private blockchains are rarely understood by outside parties not participating in the network. For that reason, the design principle of `Cactus` is to rely on the canonical validator node signatures for verifying proofs of blockchain states. Since `Cactus` should be able to incorporate any type of blockchain in the future, `Cactus` can not use the approach based on Merkle Trees.

b: A networkwide ledger view means that all network nodes have to be considered to derive the state of the blockchain which means that it is not the state of just one single blockchain node.

c: The validator nodes in `Cactus` have similarities with trusted third-party intermediaries. The terminology trusted third-party intermediaries, federation schemes, and notary schemes are used when a blockchain can retrieve the state of another blockchain through these intermediaries. In contrast, the terminology relay is used when a chain can retrieve the state of another blockchain through reading, writing, or event listening operations directly rather than relying on intermediaries. This terminology is used in the central Relay Chain in Polkadot and the BTCRelay on the Ethereum network.

d: There might be use cases where it is desired to duplicate an NFA on different ledgers. Nonetheless, we stick to the terminology that an NFA cannot be duplicated on a different ledger because an NFA can be represented as data packages on different ledgers in such cases. Data is a superset of NFAs.

e: In the case of data, the data can be copied from Blockchain A to Blockchain B. It is optional to lock/burn/delete the data object on Blockchain A after copying.

f: The process in Blockchain A and the process in Blockchain B can be seen to happen concurrently, and consecutively, in atomic swaps, and ledger interactions, respectively.

g: An action can be either a read transaction or a write transaction performed on Blockchain A or an event that is emitted by Blockchain A. Some examples of that type of ledger interoperability are as follows:

- Cross-chain oracles which are smart contracts that read the state of another blockchain before acting on it.
- Smart contracts that wait until an event happens on another blockchain before acting on it.
- Asset encumbrance smart contracts which are smart contracts that lock up assets on Blockchain A with unlocking conditions depending on actions happening in Blockchain B.

h: Alternatively, any address from that assets cannot be recovered anymore can be used. A verifiable proof for the irreversible property of that address should be given.

8.6 3. Example Use Cases

Specific use cases that we intend to support. The core idea is to support as many use-cases as possible by enabling interoperability between a large variety of ledgers specific to certain mainstream or exotic use cases.

The following table summarizes the use cases that will be explained in more detail in the following sections. FA, NFA, and D denote a fungible asset, a non-fungible asset, and data, respectively.

8.6.1 3.1 Car Trade

Use Case Attribute Name	Use Case Attribute Value
Use Case Title	Car Trade
Use Case	TBD
Interworking patterns	TBD
Type of Social Interaction	TBD
Narrative	TBD
Actors	TBD
Goals of Actors	TBD
Success Scenario	TBD
Success Criteria	TBD
Failure Criteria	TBD
Prerequisites	TBD
Comments	

8.6.2 3.2 Electricity Trade

Use Case Attribute Name	Use Case Attribute Value
Use Case Title	Electricity Trade
Use Case	TBD
Interworking patterns	TBD
Type of Social Interaction	TBD
Narrative	TBD
Actors	TBD
Goals of Actors	TBD
Success Scenario	TBD
Success Criteria	TBD
Failure Criteria	TBD
Prerequisites	TBD
Comments	

8.6.3 3.3 Supply chain

Use Case Attribute Name	Use Case Attribute Value
Use Case Title	Supply Chain
Use Case	TBD
Interworking patterns	TBD
Type of Social Interaction	TBD
Narrative	TBD
Actors	TBD
Goals of Actors	TBD
Success Scenario	TBD
Success Criteria	TBD
Failure Criteria	TBD
Prerequisites	TBD
Comments	

8.6.4 3.4 Ethereum to Quorum Asset Transfer

Use Case Attribute Name	Use Case Attribute Value
Use Case Title	Ethereum to Quorum Escrowed Asset Transfer
Use Case	1. `User A` owns some assets on an Ethereum ledger2. `User A` asks `Exchanger` to exchange specified amount of assets on Ethereum ledger, and receives exchanged asset at the Quorum ledger.
Interworking patterns	Value transfer
Type of Social Interaction	Escrowed Asset Transfer
Narrative	A person (`User A`) has multiple accounts on different ledgers (Ethereum, Quorum) and he wishes to send some assets from Ethereum ledger to a Quorum ledger with considering conversion rate. The sent asset on Ethereum will be received by Exchanger only when he successfully received converted asset on Quorum ledger.
Actors	1. `User A`: The person or entity who has ownership of the assets asscociated with its accounts on ledger.
Goals of Actors	`User A` loses ownership of sent assets on Ethereum, but he will get ownnership of exchanged asset value on Quorum.
Success Scenario	Transfer succeeds without issues. Asset is available on both Ethereum and Quorum ledgers.
Success Criteria	Transfer asset on Quorum was successed.
Failure Criteria	Transfer asset on Quorum was failed.
Prerequisites	1. Ledgers are provisioned2. `User A` and `Exchanger` identities established on both ledgers3. `Exchanger` authorized business logic plugin to operate the account on Quorum ledger.4.`User A` has access to Hyperledger Cactus deployment
Comments	

8.6.5 3.5 Escrowed Sale of Data for Coins

W3C Use Case Attribute Name	W3C Use Case Attribute Value
Use Case Title	Escrowed Sale of Data for Coins
Use Case	1. `User A` initiates (proposes) an escrowed transaction with `User B`2. `User A` places funds, `User B` places the data to a digital escrow service.3. They both observe each other's input to the escrow service and decide to proceed.4. Escrow service releases the funds and the data to the parties in the exchange.
Type of Social Interaction	Peer to Peer Exchange
Narrative	Data in this context is any series of bits stored on a computer: * Machine learning model * ad-tech database * digital/digitized art * proprietary source code or binaries of software * etc.`User A` and B trade the data and the funds through a Hyperledger Cactus transaction in an atomic swap with escrow securing both parties from fraud or unintended failures.Through the transaction protocol's handshake mechanism, A and B can agree (in advance) upon* The delivery addresses (which ledger, which wallet)* the provider of escrow that they both trust* the price and currencyEstablishing trust (e.g. Is that art original or is that machine learning model has the advertised accuracy) can be facilitated through the participating DLTs if they support it. Note that `User A` has no way of knowing the quality of the dataset, they entirely rely on `User B`'s description of its quality (there are solutions to this problem, but it's not within the scope of our use case to discuss these).
Actors	1. `User A`: A person or business organization with the intent to purchase data.2. `User B`: A person or business entity with data to sell.
Goals of Actors	`User A` wants to have access to data for an arbitrary reason such as having a business process that can enhanced by it. `User B`: Is looking to generate income/profits from data they have obtained/created/etc.
Success Scenario	Both parties have signaled to proceed with escrow and the swap happened as specified in advance.
Success Criteria	`User A` has access to the data, `User B` has been provided with the funds.
Failure Criteria	Either party did not hold up their end of the exchange/trace.
Prerequisites	`User A` has the funds to make the purchase`User B` has the data that `User A` wishes to purchase.`User A` and B can agree on a suitable currency to denominate the deal in and there is also consensus on the provider of escrow.
Comments	Hyperledger Private Data: https://hyperledger-fabric.readthedocs.io/en/release-1.4/private_data_tutorial.html Besu Privacy Groups: https://besu.hyperledger.org/en/stable/Concepts/Privacy/Privacy-Groups/

8.6.6 3.6 Money Exchanges

Enabling the trading of fiat and virtual currencies in any permutation of possible pairs.

> On the technical level, this use case is the same as the one above and therefore the specific details were omitted.

8.6.7 3.7 Stable Coin Pegged to Other Currency

W3C Use Case Attribute Name	W3C Use Case Attribute Value
Use Case Title	Stable Coin Pegged to Other Currency
Use Case	1. `User A` creates their own ledger2. `User A` deploys Hyperledger Cactus in an environment set up by them.3. `User A` implements necessary plugins for Hyperledger Cactus to interface with their ledger for transactions, token minting and burning.
Type of Social Interaction	Software Implementation Project
Narrative	Someone launches a highly scalable ledger with their own coin called ExampleCoin that can consistently sustain throughput levels of a million transactions per second reliably, but they struggle with adoption because nobody wants to buy into their coin fearing that it will lose its value. They choose to put in place a two-way peg with Bitcoin which guarantees to holders of their coin that it can always be redeemed for a fixed number of Bitcoins/USDs.
Actors	`User A`: Owner and/or operator of a ledger and currency that they wish to stabilize (peg) to other currencies
Goals of Actors	1. Achieve credibility for their currency by backing funds.2. Implement necessary software with minimal boilerplate code (most of which should be provided by Hyperldger Cactus)
Success Scenario	`User A` stood up a Hyperledger Cactus deployment with their self-authored plugins and it is possible for end user application development to start by leveraging the Hyperledger Cactus REST APIs which now expose the functionalities provided by the plugin authored by `User A`
Success Criteria	Success scenario was achieved without significant extra development effort apart from creating the Hyperledger Cactus plugins.
Failure Criteria	Implementation complexity was high enough that it would've been easier to write something from scratch without the framework
Prerequisites	* Operational ledger and currency*Technical knowledge for plugin implementation (software engineering)
Comments	

> Sequence diagram omitted as use case does not pertain to end users of Hyperledger Cactus itself.

3.7.1 With Permissionless Ledgers (BTC)

A BTC holder can exchange their BTC for ExampleCoins by sending their BTC to `ExampleCoin Reserve Wallet` and the equivalent amount of coins get minted for them onto their ExampleCoin wallet on the other network.

An ExampleCoin holder can redeem their funds to BTC by receiving a Proof of Burn on the ExampleCoin ledger and getting sent the matching amount of BTC from the `ExampleCoin Reserve Wallet` to their BTC wallet.

3.7.2 With Fiat Money (USD)

Very similar idea as with pegging against BTC, but the BTC wallet used for reserves gets replaced by a traditional bank account holding USD.

8.6.8 3.8 Healthcare Data Sharing with Access Control Lists

W3C Use Case Attribute Name	W3C Use Case Attribute Value
Use Case Title	Healthcare Data Sharing with Access Control Lists
Use Case	1. `User A` (patient) engages in business with `User B` (healthcare provider)2. `User B` requests permission to have read access to digitally stored medical history of `User A` and write access to log new entries in said medical history.3.`User A` receives a prompt to grant access and allows it.4. `User B` is granted permission through ledger specific access control/privacy features to the data of `User A`.
Type of Social Interaction	Peer to Peer Data Sharing
Narrative	Let's say that two healthcare providers have both implemented their own blockchain based patient data management systems and are looking to integrate with each other to provide patients with a seamless experience when being directed from one to another for certain treatments. The user is in control over their data on both platforms separately and with a Hyperledger Cactus backed integration they could also define fine grained access control lists consenting to the two healthcare providers to access each other's data that they collected about the patient.
Actors	* `User A`: Patient engaging in business with a healthcare provider* `User B`: Healthcare provider offering services to `User A`. Some of said services depend on having access to prior medical history of `User A`.
Goals of Actors	* `User A`: Wants to have fine grained access control in place when it comes to sharing their data to ensure that it does not end up in the hands of hackers or on a grey data market place.`User B`
Success Scenario	`User B` (healthcare provider) has access to exactly as much information as they need to and nothing more.
Success Criteria	There's cryptographic proof for the integrity of the data. Data hasn't been compromised during the sharing process, e.g. other actors did not gain unauthorized access to the data by accident or through malicious actions.
Failure Criteria	`User B` (healthcare provider) either does not have access to the required data or they have access to data that they are not supposed to.
Prerequisites	`User A` and `User B` are registered on a ledger or two separate ledgers that support the concept of individual data ownership, access controls and sharing.
Comments	It makes most sense for best privacy if `User A` and `User B` are both present with an identity on the same permissioned, privacy-enabled ledger rather than on two separate ones. This gives `User A` an additional layer of security since they can know that their data is still only stored on one ledger instead of two (albeit both being privacy-enabled)

8.6.9 3.9 Integrate Existing Food Traceability Solutions

W3C Use Case Attribute Name	W3C Use Case Attribute Value
Use Case Title	Food Traceability Integration
Use Case	1. `Consumer` is evaluating a food item in a physical retail store. 2. `Consumer` queries the designated end user application designed to provide food traces. 3. `Consumer` makes purchasing decision based on food trace.
Type of Social Interaction	Software Implementation Project
Narrative	Both `Organization A` and `Organization B` have separate products/services for solving the problem of verifying the source of food products sold by retailers.A retailer has purchased the food traceability solution from `Organization A` while a food manufacturer (whom the retailer is a customer of) has purchased their food traceability solution from `Organization B`.The retailer wants to provide end to end food traceability to their customers, but this is not possible since the chain of traceability breaks down at the manufacturer who uses a different service or solution. `Cactus` is used as an architectural component to build an integration for the retailer which ensures that consumers have access to food tracing data regardless of the originating system for it being the product/service of `Organization A` or `Organization B`.
Actors	`Organization A`, `Organization B` entities whose business has to do with food somewhere along the global chain from growing/manufacturing to the consumer retail shelves. `Consumer`: Private citizen who makes food purchases in a consumer retail goods store and wishes to trace the food end to end before purchasing decisions are finalized.
Goals of Actors	`Organization A`, `Organization B`: Provide `Consumer` with a way to trace food items back to the source.`Consumer`: Consume food that's been ethically sourced, treated and transported.
Success Scenario	`Consumer` satisfaction increases on account of the ability to verify food origins.
Success Criteria	`Consumer` is able to verify food items' origins before making a purchasing decision.
Failure Criteria	`Consumer` is unable to verify food items' origins partially or completely.
Prerequisites	1. `Organization A` and `Organization B` are both signed up for blockchain enabled software services that provide end to end food traceability solutions on their own but require all participants in the chain to use a single solution in order to work.2. Both solutions of `Organization A` and `B` have terms and conditions such that it is possible technically and legally to integrate the software with each other and `Cactus`.
Comments	

8.6.10 3.10 End User Wallet Authentication/Authorization

W3C Use Case Attribute Name	W3C Use Case Attribute Value
Use Case Title	Wallet Authentication/Authorization
Use Case	1. `User A` has separate identities on different permissioned and permissionless ledgers in the form of private/public key pairs (Public Key Infrastructure).2. `User A` wishes to access/manage these identities through a single API or user interface and opts to on-board the identities to a `Cactus` deployment.3. `User A` performs the on-boarding of identities and is now able to interact with wallets attached to said identities through `Cactus` or end user applications that leverage `Cactus` under the hood (e.g. either by directly issuing API requests or using an application that does so.
Type of Social Interaction	Identity Management
Narrative	End user facing applications can provide a seamless experience connecting multiple permissioned (or permissionless) networks for an end user who has a set of different identity proofs for wallets on different ledgers.
Actors	`User A`: The person or entity whose identities get consolidated within a single `Cactus` deployment
Goals of Actors	`User A`: Convenient way to manage an array of distinct identities with the trade-off that a `Cactus` deployment must be trusted with the private keys of the identities involved (an educated decision on the user's part).
Success Scenario	`User A` is able to interact with their wallets without having to access each private key individually.
Success Criteria	`User A`'s credentials are safely stored in the `Cactus` keychain component where it is the least likely that they will be compromised (note that it is never impossible, but least unlikely, definitely)
Failure Criteria	`User A` is unable to import identities to `Cactus` for a number of different reasons such as key format incompatibilities.
Prerequisites	1. `User A` has to have the identities on the various ledgers set up prior to importing them and must have access to the private
Comments	

8.6.11 3.11 Blockchain Migration

Use Case Attribute Name	Use Case Attribute Value
Use Case Title	Blockchain Migration
Use Case	1. `Consortium A` operates a set of services on the source blockchain.2. `Consortium A` decides to use another blockchain instance to run its services. 3. `Consortium A` migrates the existing assets and data to the target blockchain.4. `Consortium A` runs the services on the target blockchain.
Interworking patterns	Value transfer, Data transfer
Type of Social Interaction	Asset and Data Transfer
Narrative	A group of members (`Consortium A`) that operates the source blockchain (e.g., Hyperledger Fabric instance) would like to migrate the assets, data, and functionality to the target blockchain (e.g., Hyperledger Besu instance) to expand their reach or to benefits from better performance, lower cost, or enhanced privacy offered by the target blockchain. In the context of public blockchains, both the group size and the number of services could even be one. For example, a user that runs a Decentralized Application (DApp) on a publication blockchain wants to migrate DApp's assets, data, and functionality to a target blockchain that is either public or private.The migration is initiated only after all members of `Consortium A` provide their consent to migrate. During the migration, assets and data that are required to continue the services are copied to the target blockchain. A trusted intermediatory (e.g., Oracle) is also authorized by the members of `Consortium A` to show the current state of assets and data on source blockchain to the target blockchain. Assets on the source blockchain are burned and smart contracts are destructed during the migration to prevent double-spending. Proof-of-burn is verified on the target blockchain before creating the assets, smart contracts, or their state using the following process: 1. `Consortium A` requests smart-contract on the target blockchain (via `Cactus`) to transfer their asset/data, which will then wait until confirmation from the smart contract on the source blockchain. 2. `Consortium A` requests smart contract on source blockchain (via `Cactus`) to burn their asset/data. 3. Smart contract on source blockchain burns the asset/data and notifies that to the smart contract on the target blockchain. 4. Given the confirmation from Step 3, the smart contract on target blockchain creates asset/data. After the migration, future transactions are processed on the target blockchain. In contrast, requests to access historical transactions are directed to the source blockchain. As assets are burned and smart contracts are destructed, any future attempt to manipulate them on the source blockchain will fail.Regardless of whether the migration involves an entire blockchain or assets, data, and smart contracts of a DApp, migration requires lots of technical effort and time. The `Blockchain Migration` feature from `Cactus` can provide support for doing so by connecting source and target blockchains; proving values and proof-of-burn of assets, smart contracts, and data imported from the source blockchain to the target; and then performing the migration task.
Actors	1. `Consortium A`: The group of entities operating on the source blockchain who collectively aims at performing the migration to the target blockchain.
Goals of Actors	1. `Consortium A`: Wants to be able to operate its services on the target blockchain while gaining its benefits such as better performance, lower cost, or enhanced privacy.
Success Scenario	Asset and data (including smart contracts) are available on the target blockchain, enabling `Consortium A`'s services to operate on the target blockchain.
Suc-	Migration starts at a set time, and all desired assets and data, as well as their histroy have been migrated

Motivation

The suitability of a blockchain platform regarding a use case depends on the underlying blockchain properties. As blockchain technologies are maturing at a fast pace, in particular private blockchains, its properties such as performance (i.e., throughput, latency, or finality), transaction fees, and privacy might change. Also, blockchain platform changes, bug fixes, security, and governance issues may render an existing application/service suboptimal. Further, business objectives such as the interest to launch own blockchain instance, partnerships, mergers, and acquisitions may motivate a migration. Consequently, this creates an imbalance between the user expectations and the applicability of the existing solution. It is, therefore, desirable for an organization/consortium to be able to replace the blockchain providing the infrastructure to a particular application/service.

Currently, when a consortium wants to migrate the entire blockchain or user wants to migrate a DApp on a public blockchain (e.g., the source blockchain became obsolete, cryptographic algorithms are no longer secure, and business reasons), the solution is to re-implement business logic using a different blockchain platform, and arbitrary recreate the assets and data on the target blockchain, yielding great effort and time, as well as losing blockchain properties such as immutability, consistency, and transparency. Data migrations have been performed before on public blockchains [2, 3, 4] to render flexibility to blockchain-based solutions. Such work proposes data migration capabilities and patterns for public, permissionless blockchains, in which a user can specify requirements and scope for the blockchain infrastructure supporting their service.

3.11.1 Blockchain Data Migration

Data migration corresponds to capturing the subset of assets and data on a source blockchain and constructing a representation of those in a target blockchain. Note that the models underlying both blockchains do not need to be the same (e.g., world state model in Hyperledger Fabric vs account-balance model in Ethereum). For migration to be effective, it should be possible to capture the necessary assets and data from the source blockchain and to write them on the target blockchain.

3.11.2 Blockchain Smart Contract Migration

The task of migrating a smart contract comprises the task of migrating the smart contract logic and data embedded in it. In specific, the data should be accessible and writeable on another blockchain. When the target blockchain can execute a smart contract written for the source blockchain, execution behaviour can be preserved by redeploying the same smart contract (e.g., reusing a smart contract written for Ethereum on Hyperldger Besu) and recreating its assets and data. If code reuse is not possible (either at source or binary code level), the target blockchain's virtual machine should support the computational complexity of the source blockchain (e.g., one cannot migrate all Ethereum smart contracts to Bitcoin, but the other way around is feasible). Automatic smart contract migration (with or without translation) yields risks for enterprise blockchain systems, and thus the solution is nontrivial [4].

3.11.3 Semi-Automatic Blockchain Migration

By expressing a user's preferences in terms of functional and non-functional requirements, `Cactus` can recommend a set of suitable blockchains, as the target for the migration. Firstly, a user could know in real-time the characteristics of the target blockchain that would influence the migration decision. For instance, the platform can analyze the cost of writing data to Ethereum, Ether:USD exchange rate, average inter-block time, transaction throughput, and network hash rate [3]. Based on those characteristics and user-defined requirements, `Cactus` proposes a migration with indicators such as predicted cost, time to complete the migration, and the likelihood of success. For example, when transaction inclusion time or fee on Ethereum exceeds a threshold, `Cactus` may choose Polkadot platform, as it yields lower transaction inclusion time or fee. `Cactus` then safely migrate assets, data, and future transactions from Ethereum to Polkadot, without compromising the solution in production. This feature is more useful for public blockchains.

8.7 4. Software Design

8.7.1 4.1. Principles

4.1.1. Wide support

Interconnect as many ecosystems as possible regardless of technology limitations

4.1.2. Plugin Architecture from all possible aspects

Identities, DLTs, service discovery. Minimize how opinionated we are to really embrace interoperability rather than silos and lock-in. Closely monitor community feedback/PRs to determine points of contention where core Hyperledger Cactus code could be lifted into plugins. Limit friction to adding future use cases and protocols.

4.1.3. Prevent Double spending Where Possible

Two representations of the same asset do not exist across the ecosystems at the same time unless clearly labelled as such [As of Oct 30 limited to specific combinations of DLTs; e.g. not yet possible with Fabric + Bitcoin]

4.1.4 DLT Feature Inclusivity

Each DLT has certain unique features that are partially or completely missing from other DLTs. Hyperledger Cactus - where possible - should be designed in a way so that these unique features are accessible even when interacting with a DLT through Hyperledger Cactus. A good example of this principle in practice would be Kubernetes CRDs and operators that allow the community to extend the Kubernetes core APIs in a reusable way.

4.1.5 Low impact

Interoperability does not redefine ecosystems but adapts to them. Governance, trust model and workflows are preserved in each ecosystem Trust model and consensus must be a mandatory part of the protocol handshake so that any possible incompatibilities are revealed up front and in a transparent way and both parties can "walk away" without unintended loss of assets/data. The idea comes from how the traditional online payment processing APIs allow merchants to specify the acceptable level of guarantees before the transaction can be finalized (e.g. need pin, signed receipt, etc.). Following the same logic, we shall allow transacting parties to specify what sort of consensus, transaction finality, they require. Consensus requirements must support predicates, e.g. "I am on Fabric, but will accept Bitcoin so long X number of blocks were confirmed post-transaction" Requiring KYC (Know Your Customer) compliance could also be added to help foster adoption as much as possible.

4.1.6 Transparency

Cross-ecosystem transfer participants are made aware of the local and global implications of the transfer. Rejection and errors are communicated in a timely fashion to all participants. Such transparency should be visible as trustworthy evidence.

4.1.7 Automated workflows

Logic exists in each ecosystem to enable complex interoperability use-cases. Cross-ecosystem transfers can be automatically triggered in response to a previous one. Automated procedure, which is regarding error recovery and exception handling, should be executed without any interruption.

4.1.8 Default to Highest Security

Support less secure options, but strictly as opt-in, never opt-out.

4.1.9 Transaction Protocol Negotiation

Participants in the transaction must have a handshake mechanism where they agree on one of the supported protocols to use to execute the transaction. The algorithm looks an intersection in the list of supported algorithms by the participants.

4.1.10 Avoid modifying the total amount of digital assets on any blockchain whenever possible

We believe that increasing or decreasing the total amount of digital assets might weaken the security of blockchain, since adding or deleting assets will be complicated. Instead, intermediate entities (e.g. exchanger) can pool and/or send the transfer.

4.1.11 Provide abstraction for common operations

Our communal modularity should extend to common mechanisms to operate and/or observe transactions on blockchains.

4.1.12 Integration with Identity Frameworks (Moonshot)

Do not expend opinions on identity frameworks just allow users of `Cactus` to leverage the most common ones and allow for future expansion of the list of supported identity frameworks through the plugin architecture. Allow consumers of `Cactus` to perform authentication, authorization and reading/writing of credentials.

Identity Frameworks to support/consider initially:

- Hyperledger Indy (Sovrin)
- DIF
- DID

8.7.2 4.2 Feature Requirements

4.2.1 New Protocol Integration

Adding new protocols must be possible as part of the plugin architecture allowing the community to propose, develop, test and release their own implementations at will.

4.2.2 Proxy/Firewall/NAT Compatibility

Means for establishing bidirectional communication channels through proxies/firewalls/NAT wherever possible

4.2.3 Bi-directional Communications Layer

Using a blockchain agnostic bidirectional communication channel for controlling and monitoring transactions on blockchains through proxies/firewalls/NAT wherever possible.

- Blockchains vary on their P2P communication protocols. It is better to build a modular method for sending/receiving generic transactions between trustworthy entities on blockchains.

4.2.4 Consortium Management

Consortiums can be formed by cooperating entities (person, organization, etc.) who wish to contribute hardware/network resources to the operation of a `Cactus` cluster.

What holds the consortiums together is the consensus among the members on who the members are, which is defined by the nodes' network hosts and public keys. The keys are produced from the Secp256k1 curve.

The consortium plugin(s) main responsibility is to provide information about the consortium's members, nodes and public keys. `Cactus` does not prescribe any specific consensus algorithm for the addition or removal of consortium members, but rather focuses on the technical side of making it possible to operate a cluster of nodes under the ownership of separate entities without downtime while also keeping it possible to add/remove members. It is up to authors of plugins who can implement any kind of consortium management functionality as they see fit. The default implementation that Cactus ships with is in the `cactus-plugin-consortium-manual` package which - as the name implies - leaves the achievement of consensus to the initial set of members who are expected to produce the initial set of network hosts/public keys and then configure their Cactus nodes accordingly. This process and the details of operation are laid out in much more detail in the dedicated section of the manual consortium management plugin further below in this document.

After the forming of the consortium with it's initial set of members (one or more) it is possible to enroll or remove certain new or existing members, but this can vary based on different implementations.

8.7.3 4.3 Working Policies

1. Participants can insist on a specific protocol by pretending that they only support said protocol only.
2. Protocols can be versioned as the specifications mature
3. The two initially supported protocols shall be the ones that can satisfy the requirements for Fujitsu's and Accenture's implementations respectively

8.8 5. Architecture

8.8.1 5.1 Deployment Scenarios

Hyperledger Cactus has several integration patterns as the following.

- Note: In the following description, **Value (V)** means numerical assets (e.g. money). **Data (D)** means non-numerical assets (e.g. ownership proof). Ledger 1 is source ledger, Ledger 2 is destination ledger.

No.	Name	Pattern	Consistency
1.	value transfer	V -> V	check if V1 = V2 (as V1 is value on ledger 1, V2 is value on ledger 2)
2.	value-data transfer	V -> D	check if data transfer is successful when value is transferred
3.	data-value transfer	D -> V	check if value transfer is successful when data is transferred
4.	data transfer	D -> D	check if all D1 is copied on ledger 2 (as D1 is data on ledger 1, D2 is data on ledger 2)
5.	data merge	D <-> D	check if D1 = D2 as a result (as D1 is data on ledger 1, D2 is data on ledger 2)

There's a set of building blocks (members, nodes, API server processes, plugin instances) that you can use when defining (founding) a consortium and these building blocks relate to each other in a way that can be expressed with an entity relationship diagram which can be seen below. The composability rules can be deducted from how the diagram elements (entities) are connected (related) to each other, e.g. the API server process can have any number of plugin instances in it and a node can contain any number of API server processes, and so on until the top level construct is reached: the consortium.

> Consortium management does not relate to achieving consensus on data/transactions involving individual ledgers, merely about consensus on the metadata of a consortium.

Now, with these composability rules in mind, let us demonstrate a few different deployment scenarios (both expected and exotic ones) to showcase the framework's flexibility in this regard.

5.1.1 Production Deployment Example

Many different configurations are possible here as well. One way to have two members form a consortium and both of those members provide highly available, high throughput services is to have a deployment as shown on the below figure. What is important to note here is that this consortium has 2 nodes, 1 for each member and it is irrelevant how many API servers those nodes have internally because they all respond to requests through the network host/web domain that is tied to the node. One could say that API servers do not have a distinguishable identity relative to their peer API servers, only the higher-level nodes do.

5.1.2 Low Resource Deployment Example

This is an example to showcase how you can pull up a full consortium even from within a single operating system process (API server) with multiple members and their respective nodes. It is not something that's recommended for a production grade environment, ever, but it is great for demos and integration tests where you have to simulate a fully functioning consortium with as little hardware footprint as possible to save on time and cost.

The individual nodes/API servers are isolated by listening on seperate TCP ports of the machine they are hosted on:

8.8.2 5.2 System architecture and basic flow

Hyperledger Cactus will provide integrated service(s) by executing ledger operations across multiple blockchain ledgers. The execution of operations are controlled by the module of Hyperledger Cactus which will be provided by vendors as the single Hyperledger Cactus Business Logic plugin. The supported blockchain platforms by Hyperledger Cactus can be added by implementing new Hyperledger Cactus Ledger plugin. Once an API call to Hyperledger Cactus framework is requested by a User, Business Logic plugin determines which ledger operations should be executed, and it ensures reliability on the issued integrated service is completed as expected. Following diagram shows the architecture of Hyperledger Cactus based on the discussion made at Hyperledger Cactus project calls. The overall architecture is as the following figure.

5.2.1 Definition of key components in system architecture

Key components are defined as follows:

- **Business Logic Plugin**: The entity executes business logic and provide integration services that are connected with multiple blockchains. The entity is composed by web application or smart contract on a blockchain. The entity is a single plugin and required for executing Hyperledger Cactus applications.
- **CACTUS Node Server**: The server accepts a request from an End-user Application, and return a response depending on the status of the targeted trade. Trade ID will be assigned when a new trade is accepted.
- **End-user Application**: The entity submits API calls to request a trade, which invokes a set of transactions on Ledger by the Business Logic Plugin.
- **Ledger Event Listener**: The standard interface to handle various kinds of events(LedgerEvent) regarding asynchronous Ledger operations. The LedgerEvent will be notified to appropriate business logic to handle it.
- **Ledger Plugin**: The entity communicates Business Logic Plugin with each ledger. The entity is composed by a validator and a verifier as follows. The entity(s) is(are) chosen from multiple plugins on configuration.
- **Service Provider Application**: The entity submits API calls to control the cmd-api-server when it is enabling/disabling Ledger plugins, or shutting down the server. Additional commands may be available on Admin API since **Server controller** is implementation-dependent.
- **Validator**: The entity monitors transaction records of Ledger operation, and it determines the result(success, failed, timeouted) from the transaction records. Validator ensure the determined result with attaching digital signature with "Validator key" which can be verified by "Verifier".
- **Validator Server**: The server accepts a connection from Verifier, and it provides Validator API, which can be used for issuing signed transactions and monitoring Ledger behind it. The LedgerConnector will be implemented for interacting with the Ledger nodes.
- **Verifier**: The entity accepts only sucussfully verified operation results by verifying the digital signature of the validator. Verifier will be instantiated by calling the VerifierFactory#create method with associated with the Validator to connect. Each Verifier may be temporarily enabled or disabled. Note that "Validator" is apart from "Verifier" over a bi-directional channel.
- **Verifier Registry**: The information about active Verifier. The VerifierFactory uses this information to instantiate Verifier for the Business Logic Plugin.

5.2.2 Bootstrapping Cactus application

Key components defined in 4.2.1 becomes ready to serve Cactus application service after following procedures:

1. Start `Validator`: The `Validator` of `Ledger Plugin` which is chosen for each `Ledger` depending the platform technology used (ex. Fabric, Besu, etc.) will be started by the administrator of `Validator`. `Validator` becomes ready status to accept connection from `Verifier` after initialization process is done.
2. Start `Business Logic Plugin` implementation: The administrator of Cactus application service starts `Business Logic Plugin` which is implemented to execute business logic(s). `Business Logic Plugin` implementation first checks availability of depended `Ledger Plugin(s)`, then it trys to enable each `Ledger Plugin` with customized profile for actual integrating `Ledger`. This availability checks also covers determination on the status of connectivity from `Verifier` to `Validator`. The availability of each `Ledger` is registered and maintained at `Cactus Routing Interface`, and it allows bi-directional message communication between `Business Logic Plugin` and `Ledger`.

5.2.3 Processing Service API call

`Service API call` is processed as follows:

- **Step 1**: "Application user(s)" submits an API call to "Cactus routing interface".
- **Step 2**: The API call is internally routed to "Business Logic Plugin" by "Cactus Routing Interface" for initiating associated business logic. Then, "Business Logic Plugin" determines required ledger operation(s) to complete or abort a business logic.
- **Step 3**" "Business Logic Plugin" submits API calls to request operations on "Ledger(s)" wrapped with "Ledger Plugin(s)". Each API call will be routed to designated "Ledger Plugin" by "Routing Interface".
- **Step 4**: "Ledger Plugin" sends an event notification to "Business Logic Plugin" via "Cactus Routing Interface", when its sub-component "Verifier" detect an event regarding requested ledger operation to "Ledger".
- **Step 5**: "Business Logic Plugin" receives a message from "Ledger Plugin" and determines completion or continuous of the business logic. When the business logic requires to continuous operations go to "Step 3" ,or end the process.

8.8.3 5.3 APIs and communication protocols between Cactus components

API for Service Application, communication protocol for business logic plugin to interact with "Ledger Plugins" will be described in this section.

5.3.1 Cactus Service API

Cactus Service API is exposed to Application user(s). This API is used to request for initializing a business logic which is implemented at **Business Logic Plugin**. It is also used for making inquery of execution status and final result if the business logic is completed.

Following RESTful API design manner, the request can be mapped to one of CRUD operation with associated resource 'trade'.

The identity of User Application is authenticated and is applied for access control rule(s) check which is implemented as part of **Business Logic Plugin**.

NOTE: we are still open to consider other choose on API design patterns, such as gRPC or GraphQL.

Open Endpoints

Open endpoints require no authentication

- Login : `POST /api/v1/bl/login`

Restricted Endpoints

Restricted endpoints requre a valid Token to be included in the headder of the request. A Token can be acquired by calling Login.

- Request Execution of Trade(instance of business logic) : `POST /api/v1/bl/trades/`
- Show Current Status of Trade : `GET /api/v1/bl/trades/(id)`
- Show Business Logics : `GET /api/v1/bl/logics/`
- Show Specification of Business Logic : `GET /api/v1/bl/logics/(id)`
- Register a Wallet : `POST /api/v1/bl/wallets/`
- Show Wallet List : `GET /api/v1/bl/wallets/`
- Update Existing Wallets : `PUT /api/v1/bl/wallets/(id)`
- Delete a Wallet : `DELETE /api/v1/bl/walllets/(id)`

NOTE: resource `trade` and `logic` are cannot be updated nor delete

5.3.2 Ledger plugin API

Ledger plugin API is designed for allowing **Business Logic Plugin** to operate and/or monitor Ledger behind the components of **Verifier** and **Validator**.

Validator provides a common set of functions that abstract communication between **Verifier** and **Ledger**. Please note that Validator will not have any privilege to manipulate assets on the Ledger behind it. **Verifier** can receive requests from **Business Logic Plugin** and reply responses and events asynchronously.

APIs of Verifier and Validator are described as the following table:

No.	Component	API Name	Input	Description
1.	Verifier	sendAsyncRequest	`contract` (object)`method` (object)`args` (object)	Sends an asynchronous request to the validator
2.	Verifier	sendSyncRequest	`contract` (object)`method` (object)`args` (object)	Sends a synchronous request to the validator
3.	Verifier	startMonitor	`id` (string)`eventListener` (VerifierEventListener)	Request a verifier to start monitoring ledger
4.	Verifier	stopMonitor	`id` (string)	Request a verifier to stop monitoring ledger
5.	Validator	sendAsyncRequest	`args` (Object)	Called when the validator receives an asynchronous operation request from the verifier
6.	Validator	sendSyncRequest	`args` (Object)	Called when the validator receives a synchronous operation request from the verifier
7.	Validator	startMonitor	`cb` (function)	Called when monitoring ledger is needed
8.	Validator	stopMonitor	none	Called when monitoring ledger is no longer needed

The detail information is described as following:

- `packages/cactus-cmd-socketio-server/src/main/typescript/verifier/LedgerPlugin.ts`
  - interface `IVerifier`

```
interface IVerifier {
  // BLP -> Verifier
  sendAsyncRequest(contract: Object, method: Object, args: Object): Promise
↪<void>;
  sendSyncRequest(contract: Object, method: Object, args: Object): Promise<any>
↪;
  startMonitor(id: string, options: Object, eventListener:␣
↪VerifierEventListener): Promise<void>;
  stopMonitor(id: string): void;
}
```

    * function `sendAsyncRequest(contract: object, method: object, args: object): Promise<void>`
      - description:
      - Send a request to the validator (and the ledger behind it) that takes time to finish e.g. writing to the ledger.
      - input parameters:
      - `contract` (Object): specify the smart contract
      - `method` (Object): name of the method
      - `args` (Object): arguments to the method
      - returns:
      - `Promise<void>`: `resolve()` when it successfully sent the request to the validator, `reject()` otherwise.

    * function `sendSyncRequest: Promise<any>`
        - description:
        - Send a request to the validator, e.g. searching a datablock.
        - input parameters:
        - `contract` (Object): specify the smart contract
        - `method` (Object): name of the method
        - `args` (Object): arguments to the method
        - returns:
        - `Promise<any>`: search result is returned.
    * function `startMonitor(id: string, options: Object, eventListener: VerifierEventListener): Promise<void>`
        - description:
        - Request the verifier to start monitoring ledger.
        - input parameters:
        - `id` (string): a user (Business Logic Plugin) generated string to identify the ledgerEvent object.
        - `options` (Object): parameters to monitor functionality in the validator. specify `{}` if no options are necessary
        - `eventListener` (VerifierEventListener): the callback function of this object is called when there are new blocks written to the ledger.
        - returns:
        - `Promise<void>`: `resolve()` when it successfully started monitoring, `reject()` otherwise.
    * function `stopMonitor(id: string): void`
        - description:
        - Request the verifier to remove an `eventListener` from event monitors list.
        - input parameter:
        - `id` (string): a string identifying the ledgerEvent.
        - returns:
        - none

- interface `IValidator`

```
interface IValidator {
  // Verifier -> Validator
  sendAsyncRequest(args: Object): Object;
  sendSyncRequest(args: Object): Object;
  startMonitor(cb: function): Promise<void>;
  stopMonitor(): void;
}
```

    * function `sendAsyncRequest(args: Object): Object`
        - description:

- Send a request to the ledger that takes time to finish e.g. writing to the ledger. The implementer of a validator for new distributed ledger technology (DLT) is expected to extract parameters from `args` object and cal the API (of the target DLT).
- input parameter:
- `args` (Ojbect): parameters of verifier's `sendAsyncRequest` API are included in this object.
- returns:
- Object
- Editor's Note: check return type

* function `sendSyncRequest(args: Object): Object`
  - description:
  - Send a request to a validator, e.g. searching a datablock.
  - input parameter:
  - `args` (Object): parameters of verifier's `sendSyncRequest` API are inlucded in this object.
  - returns:
  - Object: result of the requested operation.
* function `startMonitor(cb: function): Promise<void>`
  - description:
  - Start monitoring of the ledger. The implementer of a validator for new distributed ledger technology (DLT) is expected to start monitoring ledger events of the target DLT. If there are any new block written to the ledger, the monitoring code should call `cb(data)`. The parameter to the `cb` is the new data from the ledger.
  - input parameter:
  - `cb` (function): callback function called when there are new data in the ledger.
  - returns:
  - Promise: `resolve()` when it successfully started monitorinig the ledger, `reject()` otherwise.
* function `stopMonitor(void): void`:
  - description:
  - Stop monitoring the ledger.
  - input parameter:
  - none
  - returns:
  - none

5.3.3 Exection of "business logic" at "Business Logic Plugin"

The developper of **Business Logic Plugin** can implement business logic(s) as codes to interact with **Ledger Plugin**. The interaction between **Business Logic Plugin** and **Ledger Plugin** includes:

- Submit a transaction request on targeted **Ledger Plugin**
- Make a inquery to targeted **Ledger Plugin** (ex. account balance inquery)
- Receive an event message, which contains transaction/inquery result(s) or error from **Ledger Plugin**

NOTE: The transaction request is prepared by **Business Logic Plugin** using transaction template with given parameters

The communication protocol between Business Logic Plugin, Verifier, and Validator as following:

8.8.4 5.4 Technical Architecture

5.4.1 Monorepo Packages

Hyperledger Cactus is divided into a set of npm packages that can be compiled separately or all at once.

All packages have a prefix of `cactus-*` to avoid potential naming conflicts with npm modules published by other Hyperledger projects. For example if both Cactus and Aries were to publish a package named `common` under the shared `@hyperledger` npm scope then the resulting fully qualified package name would end up being (without the prefix) as `@hyperledger/common` but with prefixes the conflict can be resolved as `@hyperledger/cactus-common` and `@hyperledger/aries-common`. Aries is just as an example here, we do not know if they plan on releasing packages under such names, but it also does not matter for the demonstration of ours.

Naming conventions for packages:

- cmd-* for packages that ship their own executable
- All other packages should be named preferably as a single English word suggesting the most important feature/responsibility of the package itself.

5.4.1.1 cmd-api-server

A command line application for running the API server that provides a unified REST based HTTP API for calling code. Contains the kernel of Hyperledger Cactus. Code that is strongly opinionated lives here, the rest is pushed to other packages that implement plugins or define their interfaces. Comes with Swagger API definitions, plugin loading built-in.

> By design this is stateless and horizontally scalable.

The main responsibilities of this package are:

5.4.1.1.1 Runtime Configuration Parsing and Validation

The core package is responsible for parsing runtime configuration from the usual sources (shown in order of precedence):

- Explicit instructions via code (`config.setHttpPort(3000);`)
- Command line arguments (`--http-port=3000`)
- Operating system environment variables (`HTTP_PORT=3000`)
- Static configuration files (config.json: `{ "httpPort": 3000 }`)

The Apache 2.0 licensed node-convict library to be leveraged for the mechanical parts of the configuration parsing and validation: https://github.com/mozilla/node-convict

5.4.1.1.2 Configuration Schema - API Server

To obtain the latest configuration options you can check out the latest source code of Cactus and then run this from the root folder of the project on a machine that has at least NodeJS 10 or newer installed:

```
$ date
Mon 18 May 2020 05:09:58 PM PDT

$ npx ts-node -e "import {ConfigService} from './packages/cactus-cmd-api-server/src/main/
↪typescript/config/config-service'; console.log(ConfigService.getHelpText());"

Order of precedent for parameters in descdending order: CLI, Environment variables,␣
↪Configuration file.
Passing "help" as the first argument prints this message and also dumps the effective␣
↪configuration.

Configuration Parameters
========================

  plugins:
                Description: A collection of plugins to load at runtime.
                Default:
                Env: PLUGINS
                CLI: --plugins
  configFile:
                Description: The path to a config file that holds the configuration␣
↪itself which will be parsed and validated.
                Default: Mandatory parameter without a default value.
                Env: CONFIG_FILE
                CLI: --config-file
  cactusNodeId:
                Description: Identifier of this particular Cactus node. Must be unique␣
↪among the total set of Cactus nodes running in any given Cactus deployment. Can be any␣
↪string of characters such as a UUID or an Int64
                Default: Mandatory parameter without a default value.
                Env: CACTUS_NODE_ID
                CLI: --cactus-node-id
  logLevel:
                Description: The level at which loggers should be configured. Supported␣
↪values include the following: error, warn, info, debug, trace
                Default: warn
                Env: LOG_LEVEL
                CLI: --log-level
  cockpitHost:
                Description: The host to bind the Cockpit webserver to. Secure default␣
↪is: 127.0.0.1. Use 0.0.0.0 to bind for any host.
                Default: 127.0.0.1
                Env: COCKPIT_HOST
                CLI: --cockpit-host
```

(continues on next page)

(continued from previous page)

```
  cockpitPort:
                Description: The HTTP port to bind the Cockpit webserver to.
                Default: 3000
                Env: COCKPIT_PORT
                CLI: --cockpit-port
  cockpitWwwRoot:
                Description: The file-system path pointing to the static files of web␣
↪application served as the cockpit by the API server.
                Default: packages/cactus-cmd-api-server/node_modules/@hyperledger/cactus-
↪cockpit/www/
                Env: COCKPIT_WWW_ROOT
                CLI: --cockpit-www-root
  apiHost:
                Description: The host to bind the API to. Secure default is: 127.0.0.1.␣
↪Use 0.0.0.0 to bind for any host.
                Default: 127.0.0.1
                Env: API_HOST
                CLI: --api-host
  apiPort:
                Description: The HTTP port to bind the API server endpoints to.
                Default: 4000
                Env: API_PORT
                CLI: --api-port
  apiCorsDomainCsv:
                Description: The Comma seperated list of domains to allow Cross Origin␣
↪Resource Sharing from when serving API requests. The wildcard (*) character is␣
↪supported to allow CORS for any and all domains, however using it is not recommended␣
↪unless you are developing or demonstrating something with Cactus.
                Default: Mandatory parameter without a default value.
                Env: API_CORS_DOMAIN_CSV
                CLI: --api-cors-domain-csv
  publicKey:
                Description: Public key of this Cactus node (the API server)
                Default: Mandatory parameter without a default value.
                Env: PUBLIC_KEY
                CLI: --public-key
  privateKey:
                Description: Private key of this Cactus node (the API server)
                Default: Mandatory parameter without a default value.
                Env: PRIVATE_KEY
                CLI: --private-key
  keychainSuffixPrivateKey:
                Description: The key under which to store/retrieve the private key from␣
↪the keychain of this Cactus node (API server)The complete lookup key is constructed␣
↪from the ${CACTUS_NODE_ID}${KEYCHAIN_SUFFIX_PRIVATE_KEY} template.
                Default: CACTUS_NODE_PRIVATE_KEY
                Env: KEYCHAIN_SUFFIX_PRIVATE_KEY
                CLI: --keychain-suffix-private-key
  keychainSuffixPublicKey:
                Description: The key under which to store/retrieve the public key from␣
↪the keychain of this Cactus node (API server)The complete lookup key is constructed␣
↪from the ${CACTUS_NODE_ID}${KEYCHAIN_SUFFIX_PRIVATE_KEY} template.
```

(continues on next page)

(continued from previous page)

```
                Default: CACTUS_NODE_PUBLIC_KEY
                Env: KEYCHAIN_SUFFIX_PUBLIC_KEY
                CLI: --keychain-suffix-public-key
```

5.4.1.1.3 Plugin Loading/Validation

Plugin loading happens through NodeJS's built-in module loader and the validation is performed by the Node Package Manager tool (npm) which verifies the byte level integrity of all installed modules.

5.4.1.2 core-api

Contains interface definitions for the plugin architecture and other system level components that are to be shared among many other packages. `core-api` is intended to be a leaf package meaning that it shouldn't depend on other packages in order to make it safe for any and all packages to depend on `core-api` without having to deal with circular dependency issues.

5.4.1.3 API Client

Javascript API Client (bindings) for the RESTful HTTP API provided by `cmd-api-server`. Compatible with both NodeJS and Web Browser (HTML 5 DOM + ES6) environments.

5.4.1.4 keychain

Responsible for persistently storing highly sensitive data (e.g. private keys) in an encrypted format.

For further details on the API surface, see the relevant section under `Plugin Architecture`.

5.4.1.5 tracing

Contains components for tracing, logging and application performance management (APM) of code written for the rest of the Hyperledger Cactus packages.

5.4.1.6 audit

Components useful for writing and reading audit records that must be archived longer term and immutable. The latter properties are what differentiates audit logs from tracing/logging messages which are designed to be ephemeral and to support technical issues not regulatory/compliance/governance related issues.

5.4.1.7 document-storage

Provides structured or unstructured document storage and analytics capabilities for other packages such as `audit` and `tracing`. Comes with its own API surface that serves as an adapter for different storage backends via plugins. By default, `Open Distro for ElasticSearch` is used as the storage backend: https://aws.amazon.com/blogs/aws/new-open-distro-for-elasticsearch/

> The API surface provided by this package is kept intentionally simple and feature-poor so that different underlying storage backends remain an option long term through the plugin architecture of `Cactus`.

5.4.1.8 relational-storage

Contains components responsible for providing access to standard SQL compliant persistent storage.

> The API surface provided by this package is kept intentionally simple and feature-poor so that different underlying storage backends remain an option long term through the plugin architecture of `Cactus`.

5.4.1.9 immutable-storage

Contains components responsible for providing access to immutable storage such as a distributed ledger with append-only semantics such as a blockchain network (e.g. Hyperledger Fabric).

> The API surface provided by this package is kept intentionally simple and feature-poor so that different underlying storage backends remain an option long term through the plugin architecture of `Cactus`.

5.4.2 Deployment Diagram

Source file: `./docs/architecture/deployment-diagram.puml`

5.4.3 Component Diagram

5.4.4 Class Diagram

5.4.5 Sequence Diagram - Transactions

TBD

8.8.5 5.5 Transaction Protocol Specification

5.5.1 Handshake Mechanism

TBD

5.5.2 Transaction Protocol Negotiation

Participants in the transaction must have a handshake mechanism where they agree on one of the supported protocols to use to execute the transaction. The algorithm looks an intersection in the list of supported algorithms by the participants.

Participants can insist on a specific protocol by pretending that they only support said protocol only. Protocols can be versioned as the specifications mature. Adding new protocols must be possible as part of the plugin architecture allowing the community to propose, develop, test and release their own implementations at will. The two initially supported protocols shall be the ones that can satisfy the requirements for Fujitsu's and Accenture's implementations respectively. Means for establishing bi-directional communication channels through proxies/firewalls/NAT wherever possible

8.8.6 5.6 Plugin Architecture

Since our goal is integration, it is critical that `Cactus` has the flexibility of supporting most ledgers, even those that don't exist today.

> A plugin is a self contained piece of code that implements a predefined interface pertaining to a specific functionality of `Cactus` such as transaction execution.

Plugins are an abstraction layer on top of the core components that allows operators of `Cactus` to swap out implementations at will.

> Backward compatibility is important, but versioning of the plugins still follows the semantic versioning convention meaning that major upgrades can have breaking changes.

Plugins are implemented as ES6 modules (source code) that can be loaded at runtime from the persistent data store. The core package is responsible for validating code signatures to guarantee source code integrity.

An overarching theme for all aspects that are covered by the plugin architecture is that there should be a dummy implementation for each aspect to allow the simplest possible deployments to happen on a single, consumer grade machine rather than requiring costly hardware and specialized knowledge.

> Ideally, a fully testable/operational (but not production ready) `Cactus` deployment could be spun up on a developer laptop with a single command (an npm script for example).

5.6.1 Ledger Connector Plugins

Success is defined as:

1. Adding support in `Cactus` for a ledger invented in the future requires no `core` code changes, but instead can be implemented by simply adding a corresponding connector plugin to deal with said newly invented ledger.
2. Client applications using the REST API and leveraging the feature checks can remain 100% functional regardless of the number and nature of deployed connector plugins in `Cactus`. For example: a generic money sending application does not have to hardcode the supported ledgers it supports because the unified REST API interface (fed by the ledger connector plugins) guarantees that supported features will be operational.

Because the features of different ledgers can be very diverse, the plugin interface has feature checks built into allowing callers/client applications to **determine programmatically, at runtime** if a certain feature is supported or not on a given ledger.

```
export interface LedgerConnector {
  // method to verify a signature coming from a given ledger that this connector is␣
↪responsible for connecting to.
  verifySignature(message, signature): Promise<boolean>;

  // used to call methods on smart contracts or to move assets between wallets
  transact(transactions: Transaction[]);

  getPermissionScheme(): Promise<PermissionScheme>;

  getTransactionFinality(): Promise<TransactionFinality>;

  addForeignValidator(): Promise<void>;
}

export enum TransactionFinality {
  GUARANTEED = "GUARANTEED",
  NOT_GUARANTEED = "NOT_GUARANTEED"
}

export enum PermissionScheme {
  PERMISSIONED = "PERMISSIONED",
  PERMISSIONLESS = "PERMISSIONLESS"
}
```

5.6.1.1 Ledger Connector Besu Plugin

This plugin provides `Cactus` a way to interact with Besu networks. Using this we can perform:

- Deploy Smart-contracts through bytecode.
- Build and sign transactions using different keystores.
- Invoke smart-contract functions that we have deployed on the network.

5.6.1.2 Ledger Connector Fabric Plugin

This plugin provides `Cactus` a way to interact with Fabric networks. Using this we can perform:

- Deploy Golang chaincodes.
- Make transactions.
- Invoke chaincodes functions that we have deployed on the network.

5.6.1.3 Ledger Connector Quorum Plugin

This plugin provides `Cactus` a way to interact with Quorum networks. Using this we can perform:

- Deploy Smart-contracts through bytecode.
- Build and sign transactions using different keystores.
- Invoke smart-contract functions that we have deployed on the network.

5.6.2 HTLCs Plugins

Provides an API to deploy and interact with Hash Time Locked Contracts (HTLC), used for the exchange of assets in different blockchain networks. HTLC use hashlocks and timelocks to make payments. Requires that the receiver of a payment acknowledge having received this before a deadline or he will lose the ability to claim payment, returning this to rhe payer.

5.6.2.1 HTLC-ETH-Besu Plugin

For the network Besu case, this plugin uses Leger Connector Besu Plugin to deploy an HTLC contarct on the network and provides an API to interact with the HTLC ETH swap contracts.

5.6.2.2 HTLC-ETH-ERC20-Besu Plugin

For the network Besu case, this plugin uses Leger Connector Besu Plugin to deploy an HTLC and ERC20 contarct on the network and provides an API to interact with this. This plugin allow `Cactus` to interact with ERC-20 tokens in HTLC ETH swap contracts.

5.6.3 Identity Federation Plugins

Identity federation plugins operate inside the API Server and need to implement the interface of a common PassportJS Strategy: https://github.com/jaredhanson/passport-strategy#implement-authentication

```
abstract class IdentityFederationPlugin {
  constructor(options: any): IdentityFederationPlugin;
  abstract authenticate(req: ExpressRequest, options: any);
  abstract success(user, info);
  abstract fail(challenge, status);
  abstract redirect(url, status);
  abstract pass();
  abstract error(err);
}
```

5.6.3.1 X.509 Certificate Plugin

The X.509 Certificate plugin facilitates clients authentication by allowing them to present a certificate instead of operating with authentication tokens. This technically allows calling clients to assume the identities of the validator nodes through the REST API without having to have access to the signing private key of said validator node.

PassportJS already has plugins written for client certificate validation, but we go one step further with this plugin by providing the option to obtain CA certificates from the validator nodes themselves at runtime.

5.6.4 Key/Value Storage Plugins

Key/Value Storage plugins allow the higher-level packages to store and retrieve configuration metadata for a `Cactus` cluster such as:

- Who are the active validators and what are the hosts where said validators are accessible over a network?
- What public keys belong to which validator nodes?
- What transactions have been scheduled, started, completed?

```
interface KeyValueStoragePlugin {
  async get<T>(key: string): Promise<T>;
  async set<T>(key: string, value: T): Promise<void>;
  async delete<T>(key: string): Promise<void>;
}
```

5.6.5 Serverside Keychain Plugins

The API surface of keychain plugins is roughly the equivalent of the key/value *Storage* plugins, but under the hood these are of course guaranteed to encrypt the stored data at rest by way of leveraging storage backends purpose built for storing and managing secrets.

Possible storage backends include self hosted software [1] and cloud native services [2][3][4] as well. The goal of the keychain plugins (and the plugin architecture at large) is to make `Cactus` deployable in different environments with different backing services such as an on-premise data center or a cloud provider who sells their own secret management services/APIs. There should be a dummy implementation as well that stores secrets in-memory and unencrypted (strictly for development purposes of course). The latter will decrease the barrier to entry for new users and would be contributors alike.

Direct support for HSM (Hardware Security Modules) is also something the keychain plugins could enable, but this is lower priority since any serious storage backend with secret management in mind will have built-in support for dealing with HSMs transparently.

By design, the keychain plugin can only be used by authenticated users with an active `Cactus` session. Users secrets are isolated from each other on the keychain via namespacing that is internal to the keychain plugin implementations (e.g. users cannot query other users namespaces whatsoever).

```
interface KeychainPlugin extends KeyValueStoragePlugin {
}
```

[1] https://www.vaultproject.io/ [2] https://aws.amazon.com/secrets-manager/ [3] https://aws.amazon.com/kms/ [4] https://azure.microsoft.com/en-us/services/key-vault/

5.6.6 Manual Consortium Plugin

This plugin is the default/simplest possible implementation of consortium management. It delegates the initial trust establishment to human actors to be done manually or offline if you will.

Once a set of members and their nodes were agreed upon, a JSON document containing the consortium metadata can be constructed which becomes an input parameter for the `cactus-plugin-consortium-manual` package's implementation. Members bootstrap the consortium by configuring their Cactus nodes with the agreed upon JSON document and start their nodes. Since the JSON document is used to generate JSON Web Signatures (JWS) as defined by RFC 7515 it is important that every consortium member uses the same JSON document representing the consortium.

> Attention: JWS is not the same as JSON Web Tokens (JWT). JWT is an extension of JWS and so they can seem very similar or even indistinguishable, but it is actually two separate things where JWS is the lower level building block that makes JWT's higher level use-cases possible. This is not related to Cactus itself, but is important to be mentioned since JWT is very well known among software engineers while JWS is a much less often used standard.

Example of said JSON document (the `"consortium"` property) as passed in to the plugin configuration can be seen below:

```
{
            "packageName": "@hyperledger/cactus-plugin-consortium-manual",
            "options": {
                "keyPairPem": "-----BEGIN PRIVATE KEY-----\nREDACTED\n-----END PRIVATE␣
↪KEY-----\n",
                "consortium": {
                    "name": "Example Cactus Consortium",
                    "id": "2ae136f6-f9f7-40a2-9f6c-92b1b5d5046c",
                    "mainApiHost": "http://127.0.0.1:4000",
                    "members": [
                        {
                            "id": "b24f8705-6da5-433a-b8c7-7d2079bae992",
                            "name": "Example Cactus Consortium Member 1",
                            "nodes": [
                                {
                                    "nodeApiHost": "http://127.0.0.1:4000",
                                    "publicKeyPem": "-----BEGIN PUBLIC KEY-----\
↪nMFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEtDeq7BgpelfsX7WKiSb7Lhxp8VeS6YY/\
↪nInbYuTgwZ8ykGs2Am2fM03aeMX9pYEzaeOVRU6ptwaEBFYX+YftCSQ==\n-----END PUBLIC KEY-----\n"
                                }
                            ]
                        }
                    ]
                }
            }
        }
```

The configuration above will cause the `Consortium JWS` REST API endpoint (callable via the API Client) to respond with a consortium JWS that looks similar to what is pasted below.

Code examples of how to use the API Client to call this endpoint can be seen at `./packages/cactus-cockpit/src/app/consortium-inspector/consortium-inspector.page.ts`

```
{
    "payload":
↪"eyJjb25zb3J0aXVtIjp7ImlkIjoiMmFlMTM2ZjYtZjlmNy00MGEyLTlmNmMtOTJiMWI1ZDUwNDZjIiwibWFpbkFwaUhvc3QiOiJo
↪",
```

(continues on next page)

(continued from previous page)


```
    "signatures": [
        {
            "protected":
↪"eyJpYXQiOjE1OTYyNDQzMzQ0NTksImp0aSI6IjM3NmJjMzk0LTBlYWMtNDcwZi04NjliLThkYWIzNDRmNmY3MiIsImlzcyI6Ikh5
↪",
            "signature": "ltnDyOe9WSdCk6f5Op8XlcnFoXUp3yJZgImsAvERnxWM-
↪eeL6eX0MnCtfC5r3q6knt4kTTaUv8536SMCka_YyA"
        }
    ]
}
```


The same JWS after being decoded looks like this:


```
{
    "payload": {
        "consortium": {
            "id": "2ae136f6-f9f7-40a2-9f6c-92b1b5d5046c",
            "mainApiHost": "http://127.0.0.1:4000",
            "members": [
                {
                    "id": "b24f8705-6da5-433a-b8c7-7d2079bae992",
                    "name": "Example Cactus Consortium Member 1",
                    "nodes": [
                        {
                            "nodeApiHost": "http://127.0.0.1:4000",
                            "publicKeyPem": "-----BEGIN PUBLIC KEY-----\
↪nMFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEtDeq7BgpelfsX7WKiSb7Lhxp8VeS6YY/\
↪nInbYuTgwZ8ykGs2Am2fM03aeMX9pYEzaeOVRU6ptwaEBFYX+YftCSQ==\n-----END PUBLIC KEY-----\n"
                        }
                    ]
                }
            ],
            "name": "Example Cactus Consortium"
        }
    },
    "signatures": [
        {
            "protected": {
                "iat": 1596244334459,
                "jti": "376bc394-0eac-470f-869b-8dab344f6f72",
                "iss": "Hyperledger Cactus",
                "alg": "ES256K"
            },
            "signature": "ltnDyOe9WSdCk6f5Op8XlcnFoXUp3yJZgImsAvERnxWM-
↪eeL6eX0MnCtfC5r3q6knt4kTTaUv8536SMCka_YyA"
        }
    ]
}
```


The below sequence diagram demonstrates a real world example of how a consortium between two business organizations (who both operate their own distributed ledgers) can be formed manually and then operated through the plugin discussed here. There's many other ways to perform the initial agreement that happens offline, but a concrete, non-generic example is provided here for ease of understanding:

5.6.7 Test Tooling

Provides `Cactus` with a tool to test the diferent plugins of the proyect.

8.9 6. Identities, Authentication, Authorization

`Cactus` aims to provide a unified API surface for managing identities of an identity owner. Developers using the `Cactus` Service API for their applications can support one or both of the below requirements:

1. Applications with a focus on access control and business process efficiency (usually in the enterprise)
2. Applications with a focus on individual privacy (usually consumer-based applications)

The following sections outline the high-level features of `Cactus` that make the above vision reality.

An end user (through a user interface) can issue API requests to

- register a username+password account (with optional MFA) **within** `Cactus`.
- associate their wallets to their `Cactus` account and execute transactions involving those registered wallet (transaction signatures performed either locally or remotely as explained above).
- execute a trade which executes a set of transactions across integrated Ledgers. `Cactus` may also executes recovery transaction(s) when the trade was failed with some reason. For example, recovery transactions may be executed to reverse executed transaction result using intermediate account which provide escrow trading service.

8.9.1 6.1 Definition of Identities in Cactus

Various identities are used at Cactus Service API.

Cactus user ID

- ID for user (behind web service application) to execute a Service API call.
- Service provider assign the role(s) and access right(s) of user in integrated service as part of `Business Logic Plugin`.
- The user can add Wallet(s) which is associated with account address and/or key.

Wallet ID

- ID for the user identity which is associated with authentication credential at integrated `Ledger`.
- It is recommended to store temporary credential here allowing minimal access to operate `Ledger` instead of giving full access with primary secret.
- Service API enables user to add/update/delete authentication credential for the Wallet.

Ledger ID

- ID for `Ledger Plugin` which is used at Wallet
- `Ledger ID` is assigned by administrator of integrated service, and provided for user to configure their own Wallet settings.
- The connectivity settings associated with the `Ledger ID` is also configured at `Ledger Plugin` by the administrator.

Business Logic ID

- ID for business logic to be invoked by Cactus user.

- Each business logic should be implemented to execute necessary transactions on integrated `Ledgers` without any interaction with user during its execution.
- Business logic may require user to setup access permission with storing credential before executing business logic call.

8.9.2 6.2 Transaction Signing Modes, Key Ownership

An application developer using `Cactus` can choose to enable users to sign their transactions locally on their user agent device without disclosing their private keys to `Cactus` or remotely where `Cactus` stores private keys server-side, encrypted at rest, made decryptable through authenticating with their `Cactus` account. Each mode comes with its own pros and cons that need to be carefully considered at design time.

6.2.1 Client-side Transaction Signing

Usually a better fit for consumer-based applications where end users have higher expectation of individual privacy.

Pros

- Keys are not compromised when a `Cactus` deployment is compromised
- Operator of `Cactus` deployment is not liable for breach of keys (same as above)
- Reduced server-side complexity (no need to manage keys centrally)

Cons

- User experience is sub-optimal compared to sever side transaction signing
- Users can lose access permanently if they lose the key (not acceptable in most enterprise/professional use cases)

6.2.2 Server-side Transaction Signing

Usually a better fit for enterprise applications where end users have most likely lowered their expectations of individual privacy due to the hard requirements of compliance, governance, internal or external policy enforcement.

Pros

- Frees end users from the burden of managing keys themselves (better user experience)
  - Improved compliance, governance

Cons

- Server-side breach can expose encrypted keys stored in the keychain

---

8.9.3 6.3 Open ID Connect Provider, Identity Provider

`Cactus` can authenticate users against *third party Identity Providers* or serve as an *Identity Provider* itself. Everything follows the well-established industry standards of Open ID Connect to maximize information security and reduce the probability of data breaches.

8.9.4 6.4 Server-side Keychain for Web Applications

There is a gap between traditional web/mobile applications and blockchain applications (web 2.0 and 3.0 if you will) authentication protocols in the sense that blockchain networks rely on private keys belonging to a Public Key Infrastructure (PKI) to authenticate users while traditional web/mobile applications mostly rely on a centralized authority storing hashed passwords and the issuance of ephemeral tokens upon successful authentication (e.g. successful login with a password). Traditional (Web 2.0) applications (that adhering security best practices) use server-side sessions (web) or secure keychains provided by the operating system (iOS, Android, etc.) The current industry standard and state of the art authentication protocol in the enterprise application development industry is Open ID Connect (OIDC).

To successfully close the gap between the two worlds, `Cactus` comes equipped with an OIDC identity provider and a server-side key chain that can be leveraged by end user applications to authenticate once against Hyperledger Cactus and manage identities on other blockchains through that single Hyperledger Cactus identity. This feature is important for web applications which do not have secure offline storage APIs (HTML localStorage is not secure).

Example: A user can register for a Hyperledger Cactus account, import their private keys from their Fabric/Ethereum wallets and then have access to all of those identities by authenticating once only against `Cactus` which will result in a server-side session (HTTP cookie) containing a JSON Web Token (JWT).

> Native mobile applications may not need to use the server-side keychain since they usually come equipped with an OS provided one (Android, iOS does).

In web 2.0 applications the prevalent authentication/authorization solution is Open ID Connect which bases authentication on passwords and tokens which are derived from the passwords. Web 3.0 applications (decentralized apps or *DApps*) which interact with blockchain networks rely on private keys instead of passwords.

8.10 7. Terminology

Application user: The user who requests an API call to a Hyperledger Cactus application or smart contract. The API call triggers the sending of the transaction to the remote ledger.

Hyperledger Cactus Web application or Smart contract on a blockchain: The entity executes business logic and provide integration services that include multiple blockchains.

Tx verifier: The entity verifies the signature of the transaction data transmitted over the secure bidirectional channel. Validated transactions are processed by the Hyperledger Cactus Web application or Smart Contract to execute the integrated business logic.

Tx submitter: The entity submits the remote transaction to the API server plug-in on one of the ledgers.

API Server: A Cactus package that is the backbone of the plugin architecture and the host component for all plugins with the ability to automaticaly wire up plugins that come with their own web services (REST or asynchronous APIs through HTTP or WebSockets for example)

Cactus Node: A set of identically configured API servers behind a single network host where the set size is customizable from 1 to infinity with the practical maximum being much lower of course. This logical distinction between node and API server is important because it allows consortium members to abstract away their private infrastructure details from the public consortium definition.

For example if a consortium member wants to have a highly available, high throughput service, they will be forced to run a cluster of API servers behind a load balancer and/or reverse proxy to achieve these system properties and their API servers may also be in an auto-scaling group of a cloud provider or (in the future) even run as Lambda functions. To avoid having to update the consortium definition (which requires a potentially costly consesus from other members) every time let's say an auto-scaling group adds a new API server to a node, the consortium member can define their presence in the consortium by declaring a single `Cactus Node` and then customize the underlying deployment as they see fit so long as they ensure that the previously agreed upon keys are used by the node and it is indeed accessible through the network host as declared by the `Cactus Node`. To get a better understanding of the various, near-infinite deplyoment scenarios, head over to the Deployment Scenarios sub-section of the Architecture top level section.

Validator: A module of Hyperledger Cactus which verifies validity of transaction to be sent out to the blockchain application.

Lock asset: An operation to the asset managed on blockchain ledger, which disable further operation to targeted asset. The target can be whole or partial depends on type of asset.

Abort: A state of Hyperledger Cactus which is determined integrated ledger operation is failed, and Hyperledger Cactus will execute recovery operations.

Integrated ledger operation: A series of blockchain ledger operations which will be triggered by Hyperledger Cactus. Hyperledger Cactus is responsible to execute 'recovery operations' when 'Abort' is occurred.

Restore operation(s): Single or multiple ledger operations which is executed by Hyperledger Cactus to restore the state of integrated service before start of integrated operation.

End User: A person (private citizen or a corporate employee) who interacts with Hyperledger Cactus and other ledger-related systems to achieve a specific goal or complete a task such as to send/receive/exchange money or data.

Business Organization: A for-profit or non-profit entity formed by one or more people to achieve financial gain or achieve a specific (non-financial) goal. For brevity, *business organization* may be shortened to *organization* throughout the document.

Identity Owner: A person or organization who is in control of one or more identities. For example, owning two separate email accounts by one person means that said person is the identity owner of two separate identities (the email accounts). Owning cryptocurrency wallets (their private keys) also makes one an identity owner.

Identity Secret: A private key or a password that - by design - is only ever known by the identity owner (unless stolen).

Credentials: Could mean `user a` authentication credentials/identity proofs in an IT application or any other credentials in the traditional sense of the word such as a proof that a person obtained a bachelor's degree or a PhD.

Ledger/Network/Chain: Synonymous words meaning referring largely to the same thing in this paper.

OIDC: Open ID Connect authentication protocol

PKI: Public Key Infrastructure

MFA: Multi Factor Authentication

8.11 8. Related Work

Blockchain interoperability is emerging as one of the crucial features of blockchain technology A recent survey classifies blockchain interoperability studies in three categories: Cryptocurrency-directed interoperability approaches, Blockchain Engines, and Blockchain Connectors [5]. Each category is further divided into sub-categories based on defined criteria. Each category serves particular use cases.

Category	Subcategory	Main use case
Cryptocurrency-directed Approaches	Sidechains	Scalability, asset exchange
	Notary Schemes	Cryptocurrency exchanges
	Hashed timelocks	Cryptocurrency trading
	Combined	Enabling cross-chain assets
Blockchain Engines	-	Creation of customized blockchains
Blockchain Connectors	Trusted Relays	Efficient interoperation
	Blockchain-Agnostic	General protocols
	Blockchain of Blockchains	Cross-blockchain dApps
	Blockchain Migrators	Risk reduction

Cryptocurrency-directed interoperability approaches identify and define different strategies for chain interoperability across public blockchains, most of them implementing cryptocurrencies.

Blockchain engines are frameworks that provide reusable data, network, consensus, incentive, and contract layers for the creation of customized blockchains, serving general use-cases. Emerging blockchains are, for example, the Cosmos Network and Polkadot.

The Blockchain Connector category is composed of interoperability solutions that are not cryptocurrency-directed or blockchain engines. Several sub-categories exist: Trusted Relays, Blockchain Agnostic Protocols, Blockchain of Blockchains, and Blockchain Migrators".

While Hyperledger Cactus has caracteristics from the the three categories, it can be considered a Blockchain Connector (namely a Trusted Relay). In particular, Cactus focuses on providing multiple use case scenarios via a trusted consortium. Trusted relays allow the discovery of the target blockchains, appearing often in a permissioned blockchain environment, where cross-blockchain transactions are routed by trusted escrow parties. Thus, Cactus supports developers at building cross-chain dApps.

Depending on the validator plugin, the trust on the relay can be decentralized, making Cactus a decentralized, general-purpose, trustless relay. The blockchain migrator feature paves the way for building a solution that performs data migration across blockchains.

8.12 9. References

1: Heterogeneous System Architecture - Wikipedia, Retrieved at: 11th of December 2019

2: E Scheid and Burkhard Rodrigues, B Stiller. 2019. Toward a policy-based blockchain agnostic framework. 16th IFIP/IEEE International Symposium on Integrated Network Management (IM 2019) (2019)

3: Philipp Frauenthaler, Michael Borkowski, and Stefan Schulte. 2019. A Framework for Blockchain Interoperability and Runtime Selection.

4: H.M.N. Dilum Bandara, Xiwei Xu, and Ingo Weber. 2020. Patterns for blockchain data migration. European Conf. on Pattern Languages of Programs 2020 (EuroPLoP 2020).

5: Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A Surveyon Blockchain Interoperability: Past, Present, and Future Trends,"arXiv,2020. [Online]. Available: http://arxiv.org/abs/2005.14282

8.13 10. Recommended Reference

Please use the following BibTex entry to cite this whitepaper:

@article{hyperledgercactus, title={Hyperledger Cactus Whitepaper v0.1}, author={Montgomery, Hart and Borne-Pons, Hugo and Hamilton, Jonathan and Bowman, Mic and Somogyvari, Peter and Fujimoto, Shingo and Takeuchi, Takuma and Kuhrt, Tracy and Belchior, Rafael}, journal={URL: https://github.com/hyperledger/cactus/blob/main/whitepaper/whitepaper. md}, year={2020} }

CHAPTER

NINE

REGULATORY AND INDUSTRY INITIATIVES READING LIST

A non-exhaustive list of industry standards and regulations/regulatory plans that one could read through if they are trying to get a comprehensive picture of what it takes from the non-technical standpoint to interoperate existing productive systems in finance, trade and more.

While these documents are not dealing with technological problems, even just the awareness of their existence could still add tremendous value to someone who is looking to implement some sort of integration/interoperability between different DLTs or a DLT and any pre-existing centralized system.

9.1 MICA European crypto assets regulation (coming into force by 2023)

https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A52020PC0593

9.2 Pilot Regime for DLT market infrastructure

https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:52020PC0594&from=EN

9.3 Paperless trade framework (UN / UNESCAP more on APAC now but supposed to expand in EU) for bilateral cooperation

https://readiness.digitalizetrade.org/

9.4 ICC Digital Standard Initiative (Private sector)

https://iccwbo.org/media-wall/news-speeches/digital-trade-standards-initiative-launches-under-the-umbrella-of-icc/

9.5 BAFT Digital Ledger Payment Commitment - Technical and Business Best Practices (Private sector - Banking Industry)

https://www.baft.org/wp-content/uploads/2021/03/baft-dlpc-technical-bps-v1-1.pdf

https://www.baft.org/wp-content/uploads/2021/03/baft-dlpc-business-bps-v1-1.pdf

9.6 DCSA Digital Container Shipping Association (Private sector - shipping industry)

https://dcsa.org/dcsa-publishes-standards-for-the-bill-of-lading/

9.7 MSB Networked Supervision

https://www.occ.gov/news-issuances/news-releases/2021/nr-occ-2021-2.html

9.8 OCC Interpretative letter on the use of DA for payments

https://www.occ.gov/news-issuances/news-releases/2021/nr-occ-2021-2a.pdf

9.9 Open Digital Asset Protocol from IETF (Thanks to Rafael Belchior for these)

https://datatracker.ietf.org/doc/draft-hargreaves-odap/ https://datatracker.ietf.org/doc/draft-belchior-gateway-recovery/

There are some more standards in section 6.2 of https://deepai.org/publication/a-survey-on-blockchain-interoperability-past-present-and-future-trends

CHAPTER

TEN

CACTUS COMPONENTS

This section contains the components to form Hyperledger Cactus.

10.1 @hyperledger/cactus-api-client

- Summary
- Usage
  - Routing to Cactus Node with connector to specific ledger
    - Leverage the `ConsortiumDatabase` for discovery
    - Use a provided `mainApiHost` and `ledgerId`
    - Use the API host of a node directly
- Public API Surface
  - `DefaultConsortiumProvider`
  - `ApiClient`

10.1.1 Summary

The Hyperledger Cactus API Client package is designed to be a generic extension with convenience features wrapped around the [**typescript-axios** flavored API clients](https://github.com/OpenAPITools/openapi-generator/blob/v5.2.1/docs/generators/typescript-axios.md) that we auto-generate and ship with each web service-enabled plugin such as the API clients of the

- **Manual Consortium Plugin** Typescript Axios API Client
- **Besu Connector** Typescript Axios API Client
- **Corda Connector** Typescript Axios API Client
- **Fabric Connector** Typescript Axios API Client
- **Quorum Connector** Typescript Axios API Client
- **API Server** Typescript Axios API Client
- **Vault Keychain Plugin** Typescript Axios API Client

The code generation for the listed code folders above is done by the OpenAPI Generator tool that can convert OpenAPI V3 json specifications of ours straight into the program code of the API clients.

The above means that the `ApiClient` class is not the one containing the implementation responsible for executing all the supported API calls by a Cactus node (which would make it a monolith, something that we try to avoid as it is the opposite of a flexible plugin architecture)

For example you can use the `@hyperledger/cactus-api-client` node package to perform Cactus node discovery based on ledger IDs (that can be obtained from the `ConsortiumDatabase` as defined by the generated models of the `@hyperledger/cactus-core-api` package.

> While you can generate API Clients for the Cactus API specifications in any supported langauge of the OpenAPI Generator the features provided by this package will have to be developed separately (if not already done by the Cactus maintainers). Currently the only implementation of the abstract API Client and its features (node discovery) is in Typescript (e.g. the `@hyperledger/cactus-api-client` package).

10.1.2 Usage

Routing to Cactus Node with connector to specific ledger

Let's say you have a consortium with several members who all have their own ledgers deployed as well. The `ConsortiumDatabase` will therefore contain the entities pertaining to these entities (such as the ledgers or the members themselves) meaning that if you are developing an application that needs to perform operations on one of the ledgers in the consortium then you have a couple of different ways of obtaining an API client to do just that:

Leverage the `ConsortiumDatabase` for discovery

```
import { ApiClient } from "@hyperledger/cactus-api-client";

import { ConsortiumDatabase, Ledger, LedgerType } from "@hyperledger/cactus-core-api";

import { PluginRegistry } from "@hyperledger/cactus-core";

import { DefaultApi as QuorumApi } from "@hyperledger/cactus-plugin-ledger-connector-
↪quorum";

const mainFn = async () => {
  const ledgerId = "theIdOfYourLedgerInTheConsortiumDatabase";

  // How you obtain a consortium provider is dependent on which consortium
  // plugin you use and your exact deployment scenario
  const consortiumProvider: IAsyncProvider<ConsortiumDatabase> = ...;
  const consortiumDatabase: ConsortiumDatabase = await consortiumProvider.get();
  const consortium = consortiumDatabase.consortium[0];

  const mainApiClient = new ApiClient({ basePath: consortium.mainApiHost });

  // This client is now configured to point to a node that has a connector to
  // the ledger referenced by `ledgerId`
  const apiClient = await mainApiClient.ofLedger(ledgerId, QuorumApi);

  // Use the client to perform any supported operation on the ledger
};

mainFn();
```

Use a provided `mainApiHost` and `ledgerId`

```
import { ApiClient } from "@hyperledger/cactus-api-client";

import { ConsortiumDatabase, Ledger, LedgerType } from "@hyperledger/cactus-core-api";

import { PluginRegistry } from "@hyperledger/cactus-core";

import { DefaultApi as QuorumApi } from "@hyperledger/cactus-plugin-ledger-connector-
↪quorum";

const mainFn = async () => {
  const ledgerId = "theIdOfYourLedgerInTheConsortiumDatabase";
  const consortiumMainApiHost = "https://cactus.example.com";

  const mainApiClient = new ApiClient({ basePath: consortiumMainApiHost });

  // This client is now configured to point to a node that has a connector to
  // the ledger referenced by `ledgerId`
  const apiClient = await mainApiClient.ofLedger(ledgerId, QuorumApi);
}

mainFn();
```

Use the API host of a node directly

```
import { ApiClient } from "@hyperledger/cactus-api-client";

import { ConsortiumDatabase, Ledger, LedgerType } from "@hyperledger/cactus-core-api";

import { PluginRegistry } from "@hyperledger/cactus-core";

import { DefaultApi as QuorumApi } from "@hyperledger/cactus-plugin-ledger-connector-
↪quorum";

const mainFn = async () => {
  const nodeApiHost = "https://my-node.cactus.example.com";

  const mainApiClient = new ApiClient({ basePath: nodeApiHost });

  // This client is now configured to point to a node that has a connector to the ledger␣
↪of your choice
  const apiClient = await mainApiClient.extendWith(QuorumApi);
}

mainFn();
```

10.1.3 Public API Surface

DefaultConsortiumProvider

Builds the default Consortium provider that can be used by this object to retrieve the Cactus Consortium metadata object when necessary (one such case is when we need information about the consortium nodes to perform routing requests to a specific ledger via a connector plugin, but later other uses could be added as well).

The DefaultConsortiumProvider class leverages the simplest consortium plugin that we have at the time of this writing: @hyperledger/cactus-plugin-consortium-manual which holds the consortium metadata as pre-configured by the consortium operators.

The pattern we use in the ApiClient class is that you can inject your own `IAsyncProvider<Consortium>` implementation which then will be used for routing information and in theory you can implement completely arbitrary consortium management in your own consortium plugins which then Cactus can use and leverage for the routing. This allows us to support any exotic consortium management algorithms that people may come up with such as storing the consortium definition in a multi-sig smart contract or have the list of consortium nodes be powered by some sort of automatic service discovery or anything else that people might think of.

ApiClient

Class responsible for providing additional functionality to the DefaultApi classes of the generated clients (OpenAPI generator / typescript-axios).

Each package (plugin) can define it's own OpenAPI spec which means that they all can ship with their own `DefaultApi` class that is generated directly from the respective OpenAPI spec of the package/plugin.

The functionality provided by this class is meant to be common traints that can be useful for all of those different `DefaultApi` implementations.

One such common trait is the client side component of the routing that decides which Cactus node to point the `ApiClient` towards (which is in itself ends up being the act of routing).

@see — https ://github.com/OpenAPITools/openapi-generator/blob/v5.0.0-beta2/modules/openapi-generator/src/main/resources/typescript-axios/apiInner.mustache#L337

@see — https ://github.com/OpenAPITools/openapi-generator/blob/v5.0.0/docs/generators/typescript-axios.md

10.2 @hyperledger/cactus-cmd-api-server

- Summary
- Usage
  - Basic Example
  - Remote Plugin Imports at Runtime Example
  - Complete Example
- Deployment Scenarios
  - Production Deployment Example
  - Low Resource Deployment Example
- Containerization
  - Building the container image locally

  - Running the container image locally
  - Testing API calls with the container
- Prometheus Exporter
  - Usage Prometheus
  - Prometheus Integration
  - Shutdown Hook
  - Helper code - response.type.ts - data-fetcher.ts - metrics.ts
- FAQ
  - What is the difference between a Cactus Node and a Cactus API Server?
  - Is the API server horizontally scalable?
  - Does the API server automatically protect me from malicious plugins?
  - Can I use the API server with plugins deployed as a service?

10.2.1 Summary

This package is part of the Hyperledger Cactus blockchain integration framework and is used as a shell/container of sort for housing different Cactus plugins (which all live in their own npm packages as well).

The API server gives you for free the following benefits, should you choose to use it:

1. Automatic wiring of API endpoints for Cactus plugins which implement the `IPluginWebService` Typescript interface
2. Lightweight inversion of control container provided to plugins in the form of the `PluginRegistry` so that plugins can depend on each other in a way that each plugin instance can be uniquely identified and obtained by other plugins. A great example of this in action is ledger connector plugins frequently using the `PluginRegistry` to look up instances of keychain plugins to get access to secrets that are needed for the connector plugins to accomplish certain tasks such as cryptographically signing some information or SSH-ing into a server instance in order to upload and deploy binary (or otherwise) artifacts of smart contracts.

10.2.2 Usage

Like with most parts of the framework in Cactus, using the `ApiServer` is optional.

To see the `ApiServer` in action, the end to end tests of the framework are a great place to start. A few excerpts that regularly occur in said tests can be seen below as well for the reader's convenience.

One of our design principles for the framework is **secure by default** which means that the API servers

1. assumes TLS is enabled by default and
2. cross-origin resource sharing is disabled completely

Basic Example

```
#!/usr/bin/env node

import { ApiServer } from "../api-server";
import { ConfigService } from "../config/config-service";
import { Logger, LoggerProvider } from "@hyperledger/cactus-common";

const log: Logger = LoggerProvider.getOrCreate({
  label: "cactus-api",
  level: "INFO",
});

const main = async () => {
  const configService = new ConfigService();
  const config = await configService.getOrCreate();
  const serverOptions = config.getProperties();

  LoggerProvider.setLogLevel(serverOptions.logLevel);

  if (process.argv[2].includes("help")) {
    const helpText = ConfigService.getHelpText();
    console.log(helpText);
    log.info(`Effective Configuration:`);
    log.info(JSON.stringify(serverOptions, null, 4));
  } else {
    const apiServer = new ApiServer({ config: serverOptions });
    await apiServer.start();
  }
};

export async function launchApp(): Promise<void> {
  try {
    await main();
    log.info(`Cactus API server launched OK `);
  } catch (ex) {
    log.error(`Cactus API server crashed: `, ex);
    process.exit(1);
  }
}

if (require.main === module) {
  launchApp();
}
```

Remote Plugin Imports at Runtime Example

```
import { PluginImportType, PluginImportAction } from "@hyperledger/cactus-core-api";
import { ApiServer } from "@hyperledger/cactus-cmd-api-server";
import { ConfigService } from "@hyperledger/cactus-cmd-api-server";
import { Logger, LoggerProvider } from "@hyperledger/cactus-common";

const main = async () => {

  const configService = new ConfigService();
  const apiServerOptions = await configService.newExampleConfig();
  // If there is no configuration file on the file system, just set it to empty string
  apiServerOptions.configFile = "";
  // Enable CORS for
  apiServerOptions.apiCorsDomainCsv = "your.domain.example.com";
  apiServerOptions.apiPort = 3000;
  apiServerOptions.cockpitPort = 3100;
  apiServerOptions.grpcPort = 5000;
  // Disble TLS (or provide TLS certs for secure HTTP if you are deploying to production)
  apiServerOptions.apiTlsEnabled = false;
  apiServerOptions.plugins = [
    {
      // npm package name of the plugin you are installing
      // Since this will be imported at runtime, you are responsible for
      // installing the package yourself prior to launching the API server.
      packageName: "@hyperledger/cactus-plugin-keychain-vault",
      // The REMOTE value means that a different plugin factory will be imported and
      // called to obtain the plugin instance. This way plugins can support them
      // being imported by the API server regardless of the language the plugin
      // was written in.
      type: PluginImportType.REMOTE,
      // The INSTALL value means that the plugin will be installed instead of
      // only instantiate it
      action: PluginImportAction.INSTALL,
      // The options that will be passed in to the plugin factory
      options: {
        keychainId: "_keychainId_",
        instanceId: "_instanceId_",
        remoteConfig: configuration,
      },
    },
  ];
  const config = await configService.newExampleConfigConvict(apiServerOptions);

  const apiServer = new ApiServer({
    config: config.getProperties(),
  });

  // start the API server here and you are ready to roll
};

export async function launchApp(): Promise<void> {
  try {
```

(continues on next page)

(continued from previous page)

```
    await main();
    log.info(`Cactus API server launched OK `);
  } catch (ex) {
    log.error(`Cactus API server crashed: `, ex);
    process.exit(1);
  }
}

if (require.main === module) {
  launchApp();
}
```

Complete Example

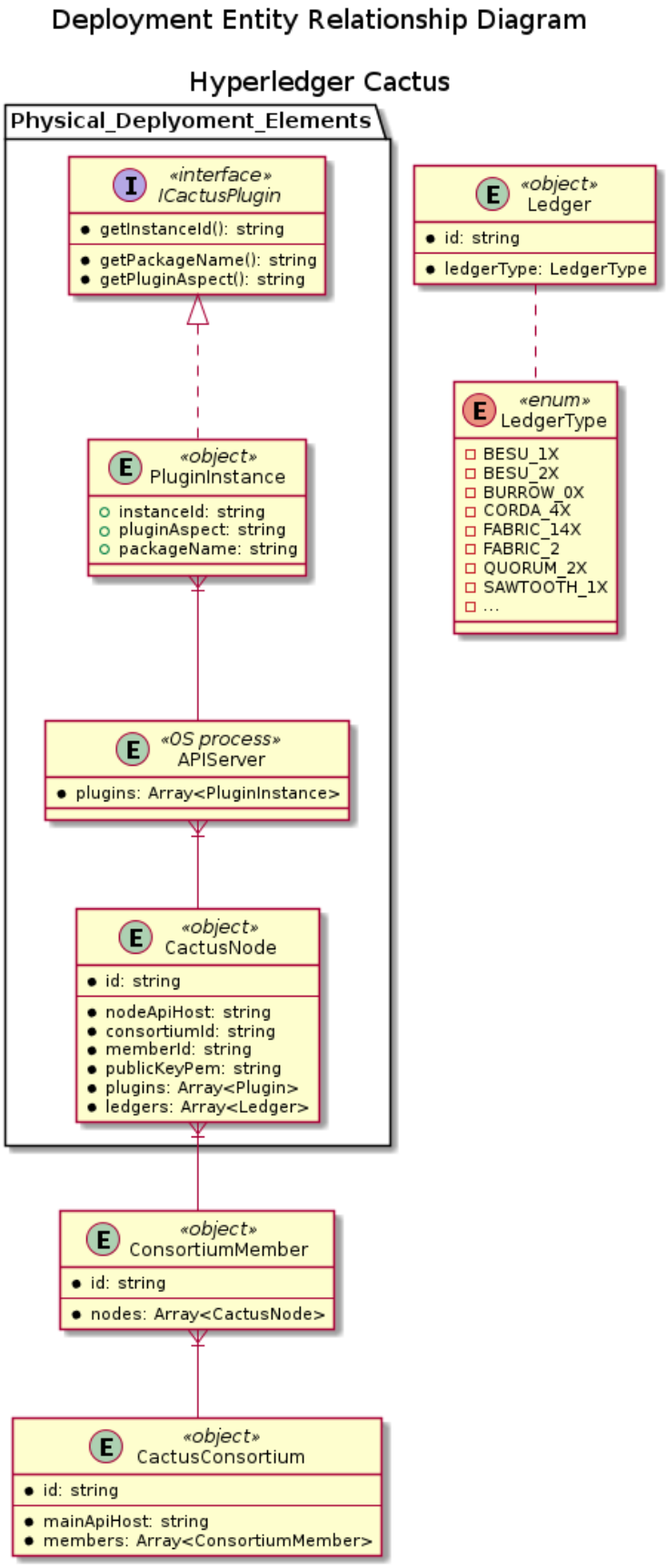
For a complete example of how to use the API server, read all the code of the supply chain example's backend package:

https://github.com/hyperledger/cactus/tree/main/examples/cactus-example-supply-chain-backend/src/main/typescript

10.2.3 Deployment Scenarios

There's a set of building blocks (members, nodes, API server processes, plugin instances) that you can use when defining (founding) a consortium and these building blocks relate to each other in a way that can be expressed with an entity relationship diagram which can be seen below. The composability rules can be deducted from how the diagram elements (entities) are connected (related) to each other, e.g. the API server process can have any number of plugin instances in it and a node can contain any number of API server processes, and so on until the top level construct is reached: the consortium.

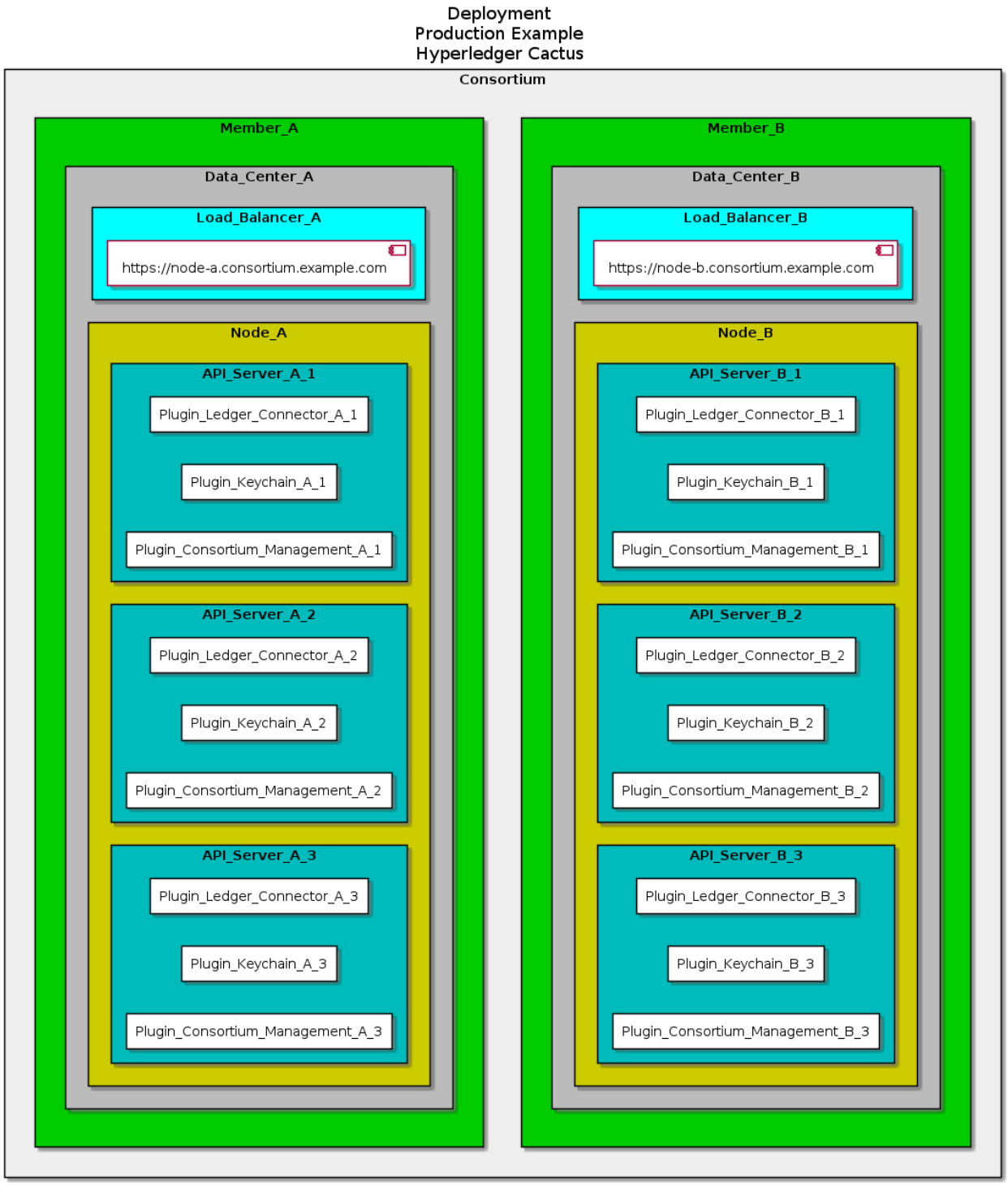
> Consortium management does not relate to achieving consensus on data/transactions involving individual ledgers, merely about consensus on the metadata of a consortium.


Deployment Entity Relationship Diagram
Hyperledger Cactus
Physical_Deplyoment_Elements
«interface»
ICactusPlugin
getInstanceId(): string
getPackageName(): string
getPluginAspect(): string
«object»
Ledger
id: string
ledgerType: LedgerType
«enum»
LedgerType
BESU_1X
BESU_2X
BURROW_0X
CORDA_4X
FABRIC_14X
FABRIC_2
QUORUM_2X
SAWTOOTH_1X
...
«object»
PluginInstance
instanceId: string
pluginAspect: string
packageName: string
«OS process»
APIServer
plugins: Array<PluginInstance>
«object»
CactusNode
id: string
nodeApiHost: string
consortiumId: string
memberId: string
publicKeyPem: string
plugins: Array<Plugin>
ledgers: Array<Ledger>
«object»
ConsortiumMember
id: string
nodes: Array<CactusNode>
«object»
CactusConsortium
id: string
mainApiHost: string
members: Array<ConsortiumMember>

Now, with these composability rules in mind, let us demonstrate a few different deployment scenarios (both expected and exotic ones) to showcase the framework's flexibility in this regard.

Production Deployment Example

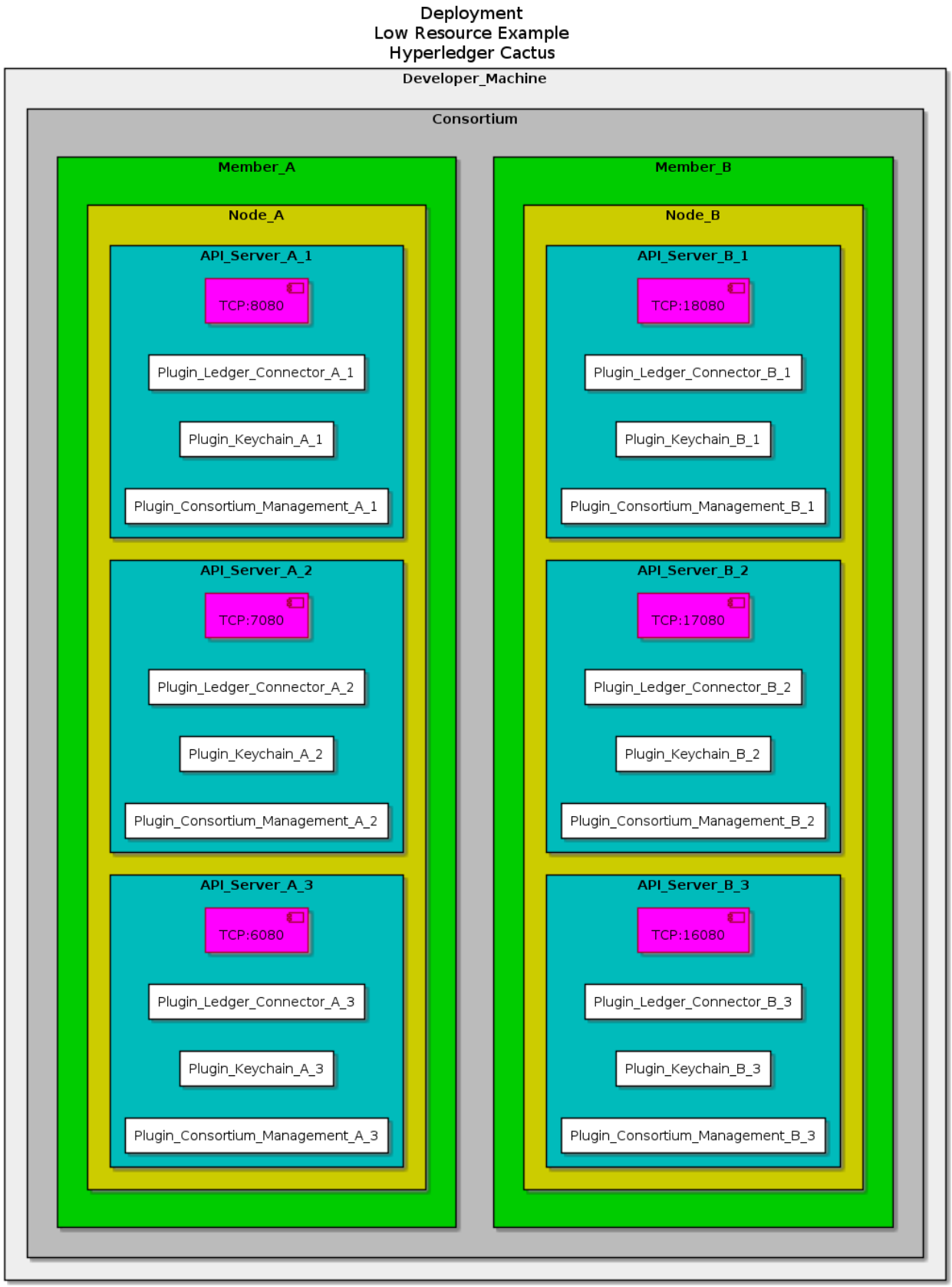
Many different configurations are possible here as well. One way to have two members form a consortium and both of those members provide highly available, high throughput services is to have a deployment as shown on the below figure. What is important to note here is that this consortium has 2 nodes, 1 for each member and it is irrelevant how many API servers those nodes have internally because they all respond to requests through the network host/web domain that is tied to the node. One could say that API servers do not have a distinguishable identity relative to their peer API servers, only the higher-level nodes do.


Deployment
Production Example
Hyperledger Cactus
Consortium
Member_A
Data_Center_A
Load_Balancer_A
https://node-a.consortium.example.com
Node_A
API_Server_A_1
Plugin_Ledger_Connector_A_1
Plugin_Keychain_A_1
Plugin_Consortium_Management_A_1
API_Server_A_2
Plugin_Ledger_Connector_A_2
Plugin_Keychain_A_2
Plugin_Consortium_Management_A_2
API_Server_A_3
Plugin_Ledger_Connector_A_3
Plugin_Keychain_A_3
Plugin_Consortium_Management_A_3
Member_B
Data_Center_B
Load_Balancer_B
https://node-b.consortium.example.com
Node_B
API_Server_B_1
Plugin_Ledger_Connector_B_1
Plugin_Keychain_B_1
Plugin_Consortium_Management_B_1
API_Server_B_2
Plugin_Ledger_Connector_B_2
Plugin_Keychain_B_2
Plugin_Consortium_Management_B_2
API_Server_B_3
Plugin_Ledger_Connector_B_3
Plugin_Keychain_B_3
Plugin_Consortium_Management_B_3

Low Resource Deployment Example

This is an example to showcase how you can pull up a full consortium even from within a single operating system process (API server) with multiple members and their respective nodes. It is not something that's recommended for a production grade environment, ever, but it is great for demos and integration tests where you have to simulate a fully functioning consortium with as little hardware footprint as possible to save on time and cost.

The individual nodes/API servers are isolated by listening on seperate TCP ports of the machine they are hosted on:


Deployment
Low Resource Example
Hyperledger Cactus
Developer_Machine
Consortium
Member_A
Node_A
API_Server_A_1
TCP:8080
Plugin_Ledger_Connector_A_1
Plugin_Keychain_A_1
Plugin_Consortium_Management_A_1
API_Server_A_2
TCP:7080
Plugin_Ledger_Connector_A_2
Plugin_Keychain_A_2
Plugin_Consortium_Management_A_2
API_Server_A_3
TCP:6080
Plugin_Ledger_Connector_A_3
Plugin_Keychain_A_3
Plugin_Consortium_Management_A_3
Member_B
Node_B
API_Server_B_1
TCP:18080
Plugin_Ledger_Connector_B_1
Plugin_Keychain_B_1
Plugin_Consortium_Management_B_1
API_Server_B_2
TCP:17080
Plugin_Ledger_Connector_B_2
Plugin_Keychain_B_2
Plugin_Consortium_Management_B_2
API_Server_B_3
TCP:16080
Plugin_Ledger_Connector_B_3
Plugin_Keychain_B_3
Plugin_Consortium_Management_B_3

10.2.4 Containerization

Building the container image locally

In the Cactus project root say:

```
DOCKER_BUILDKIT=1 docker build -f ./packages/cactus-cmd-api-server/Dockerfile . -t cas -
↪t cactus-api-server
```

Build with a specific version of the npm package:

```
DOCKER_BUILDKIT=1 docker build --build-arg NPM_PKG_VERSION=main -f ./packages/cactus-cmd-
↪api-server/Dockerfile . -t cas -t cactus-api-server
```

Running the container image locally

Before running the examples here you need to build the image locally. See section Building the container image locally for details on how to do that.

Once you've built the container, the following commands should work:

- Launch container - no plugins, default configuration

```
docker run \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  cas
```

- Launch container with plugins of your choice (keychain, consortium connector, etc.)

```
  docker run \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  cas \
    ./node_modules/.bin/cactusapi \
    --plugins='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-fabric
↪", "type": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.
↪hyperledger.cactus.plugin_import_action.INSTALL",  "options": { "connectionProfile
↪": {}, "instanceId": "some-unique-instance-id"}}]'
```

- Launch container with plugin configuration as an **environment variable**:

```
docker run \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  --env PLUGINS='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-besu
↪", "type": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.
↪hyperledger.cactus.plugin_import_action.INSTALL",  "options": {"rpcApiHttpHost":
↪"http://localhost:8545", "instanceId": "some-unique-besu-connector-instance-id"}}]
↪' \
  cas
```

- Launch container with plugin configuration as a **CLI argument**:

```
docker run \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  cas \
    ./node_modules/.bin/cactusapi \
    --plugins='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-besu",
↪"type": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.
↪hyperledger.cactus.plugin_import_action.INSTALL",  "options": {"rpcApiHttpHost":
↪"http://localhost:8545", "instanceId": "some-unique-besu-connector-instance-id"}}]
↪'
```

- Launch container with **configuration file** mounted from host machine:

```
echo '[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-besu", "type":
↪"org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.
↪cactus.plugin_import_action.INSTALL",  "options": {"rpcApiHttpHost": "http://
↪localhost:8545", "instanceId": "some-unique-besu-connector-instance-id"}}]' >␣
↪cactus.json

docker run \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  --mount type=bind,source="$(pwd)"/cactus.json,target=/cactus.json \
  cas \
    ./node_modules/.bin/cactusapi \
    --config-file=/cactus.json
```

Testing API calls with the container

Don't have a Besu network on hand to test with? Test or develop against our Besu All-In-One container!

1. Terminal Window 1 (Ledger)

```
docker run --publish 8545:8545 hyperledger/cactus-besu-all-in-one:latest
```

2. Terminal Window 2 (Cactus API Server)

```
docker run \
  --network host \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  --env PLUGINS='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-besu
↪", "type": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.
↪hyperledger.cactus.plugin_import_action.INSTALL",  "options": {"rpcApiHttpHost":
↪"http://localhost:8545", "instanceId": "some-unique-besu-connector-instance-id"}}]
↪' \
  cas
```

3. Terminal Window 3 (curl - replace eth accounts as needed)

```
curl --location --request POST 'http://127.0.0.1:4000/api/v1/plugins/@hyperledger/
↪cactus-plugin-ledger-connector-besu/run-transaction' \
--header 'Content-Type: application/json' \
--data-raw '{
    "web3SigningCredential": {
      "ethAccount": "627306090abaB3A6e1400e9345bC60c78a8BEf57",
      "secret": "c87509a1c067bbde78beb793e6fa76530b6382a4c0241e5e4a9ec0a0f44dc0d3",
      "type": "PRIVATE_KEY_HEX"
    },
    "consistencyStrategy": {
      "blockConfirmations": 0,
      "receiptType": "NODE_TX_POOL_ACK"
    },
    "transactionConfig": {
      "from": "627306090abaB3A6e1400e9345bC60c78a8BEf57",
      "to": "f17f52151EbEF6C7334FAD080c5704D77216b732",
      "value": 1,
      "gas": 10000000
    }
}'
```

4. The above should produce a response that looks similar to this:


```
{
    "success": true,
    "data": {
        "transactionReceipt": {
            "blockHash":
↪"0x7c97c038a5d3bd84613fe23ed442695276d5d2df97f4e7c4f10ca06765033ffd",
            "blockNumber": 1218,
            "contractAddress": null,
            "cumulativeGasUsed": 21000,
            "from": "0x627306090abab3a6e1400e9345bc60c78a8bef57",
            "gasUsed": 21000,
            "logs": [],
            "logsBloom":
↪"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
            "status": true,
            "to": "0xf17f52151ebef6c7334fad080c5704d77216b732",
            "transactionHash":
↪"0xc7fcb46c735bdc696d500bfc70c72595a2b8c31813929e5c61d9a5aec3376d6f",
            "transactionIndex": 0
        }
    }
}
```

10.2.5 Prometheus Exporter

This class creates a prometheus exporter, which scrapes the total Cactus node count.

Usage Prometheus

The prometheus exporter object is initialized in the `ApiServer` class constructor itself, so instantiating the object of the `ApiServer` class, gives access to the exporter object. You can also initialize the prometheus exporter object seperately and then pass it to the `IApiServerConstructorOptions` interface for `ApiServer` constructor.

`getPrometheusMetricsV1` function returns the prometheus exporter metrics, currently displaying the total plugins imported, which currently refreshes to match the plugin count, everytime `setTotalPluginImports` method is called.

Prometheus Integration

To use Prometheus with this exporter make sure to install Prometheus main component. Once Prometheus is setup, the corresponding scrape_config needs to be added to the prometheus.yml

```
- job_name: 'consortium_manual_exporter'
  metrics_path: /api/v1/api-server/get-prometheus-exporter-metrics
  scrape_interval: 5s
  static_configs:
    - targets: ['{host}:{port}']
```

Here the `host:port` is where the prometheus exporter metrics are exposed. The test cases (For example, packages/cactus-plugin-consortium-manual/src/test/typescript/unit/consortium/get-node-jws-endpoint-v1.test.ts) exposes it over `0.0.0.0` and a random port(). The random port can be found in the running logs of the test case and looks like (42379 in the below mentioned URL) `Metrics URL: http://0.0.0.0:42379/api/v1/api-server/get-prometheus-exporter-metrics/get-prometheus-exporter-metrics`

Once edited, you can start the prometheus service by referencing the above edited prometheus.yml file. On the prometheus graphical interface (defaulted to http://localhost:9090), choose **Graph** from the menu bar, then select the **Console** tab. From the **Insert metric at cursor** drop down, select **cactus_api_server_total_plugin_imports** and click **execute**

Shutdown Hook

The API config contains a flag:

```
{
  "enableShutdownHook": true
}
```

This allows for graceful shutdown of the API server after a SIGINT via cli CTRL + C. This hook can be disabled by passing in false either via the TypeScript constructor or the JSON config file.

Helper code

response.type.ts

This file contains the various responses of the metrics.

data-fetcher.ts

This file contains functions encasing the logic to process the data points.

metrics.ts

This file lists all the prometheus metrics and what they are used for.

10.2.6 FAQ

What is the difference between a Cactus Node and a Cactus API Server?

The node is what has an identity within your PKI and can be made up of 1-N API server instances that all share the same configuration/identity of the node. See deployment scenarios above for a much more detailed explanation.

Is the API server horizontally scalable?

Yes, 100%. Keep in mind though that the API server can be loaded up with arbitrary plugins meaning that if you write a plugin that has a central database that can only do 1 transaction per second, then it will not help you much that the API server itself is horizontally scalable because deploying a thousand instances of the API server will just result in you having a thousand instances of your plugin all waiting for that underlying database with its 1 TPS throughput hogging your system. When we say that the API server is horizontally scalable, we mean that the API server itself is designed not to have any such state mentioned in the example above. You are responsible for only deploying plugins in the API server that are horizontally scalable as well. In short, your whole system is only horizontally scalable if all components of it are horizontally scalable.

Does the API server automatically protect me from malicious plugins?

No. If you install a third-party plugin that wasn't vetted by anyone and that plugin happens to have malicious code in it to steal your private keys, it can do so. You are responsible for making sure that the plugins you install have no known security vulnerabilities or backdoors e.g. they are considered "secure". The double quotes around "secure" is meant to signify the fact that no software is ever really truly secure, just merely lacking of known vulnerabilities at any given point in time.

Can I use the API server with plugins deployed as a service?

Yes. You can deploy your plugin written in any language, anywhere as long as it is accessible over the network and does come with a Typescript API client that you can use to install into the API server as a proxy for an in-process plugin implementation.

10.3 @hyperledger/cactus-common

TODO: description

10.3.1 Usage

```
// TODO: DEMONSTRATE API
```

10.4 @hyperledger/cactus-core

This module is responsible for providing the backbone for the rest of the packages when it comes to the features of Cactus.

The main difference between this and the `cactus-common` package is that this one does not guarantee it's features to work in the browser.

The main difference from the `cactus-core-api` package is that this is meant to contain actual implementations while `cactus-core-api` is meant to be strictly about defining interfaces. Based on that latter constraint we may move the `PluginRegistry` out of that package and into this one in the near future.

10.4.1 Usage

```
// TODO: DEMONSTRATE API
```

10.5 @hyperledger/cactus-plugin-keychain-vault

10.5.1 Prometheus Exporter

This class creates a prometheus exporter, which scrapes the transactions (total transaction count) for the use cases incorporating the use of Keychain vault plugin.

Usage

The prometheus exporter object is initialized in the `PluginKeychainVault` class constructor itself, so instantiating the object of the `PluginKeychainVault` class, gives access to the exporter object. You can also initialize the prometheus exporter object seperately and then pass it to the `IPluginKeychainVaultOptions` interface for `PluginKeychainVault` constructor.

`getPrometheusMetricsV1` function returns the prometheus exporter metrics, currently displaying the total key count, which currently increments everytime the `set()` and `delete()` method of the `PluginKeychainVault` class is called.

Prometheus Integration

To use Prometheus with this exporter make sure to install Prometheus main component. Once Prometheus is setup, the corresponding scrape_config needs to be added to the prometheus.yml

```
- job_name: 'keychain_vault_exporter'
  metrics_path: api/v1/plugins/@hyperledger/cactus-plugin-keychain-vault/get-prometheus-
↪exporter-metrics
  scrape_interval: 5s
  static_configs:
    - targets: ['{host}:{port}']
```

Here the `host:port` is where the prometheus exporter metrics are exposed. The test cases (For example, packages/cactus-plugin-keychain-vault/src/test/typescript/integration/plugin-keychain-vault.test.ts) exposes it over `0.0.0.0` and a random port(). The random port can be found in the running logs of the test case and looks like (42379 in the below mentioned URL) `Metrics URL: http://0.0.0.0:42379/api/v1/plugins/@hyperledger/cactus-plugin-keychain-plugin/get-prometheus-exporter-metrics`

Once edited, you can start the prometheus service by referencing the above edited prometheus.yml file. On the prometheus graphical interface (defaulted to http://localhost:9090), choose **Graph** from the menu bar, then select the **Console** tab. From the **Insert metric at cursor** drop down, select **cactus_keychain_vault_total_key_count** and click **execute**

Helper code

response.type.ts

This file contains the various responses of the metrics.

data-fetcher.ts

This file contains functions encasing the logic to process the data points

metrics.ts

This file lists all the prometheus metrics and what they are used for.

10.6 @hyperledger/cactus-plugin-ledger-connector-besu

This plugin provides `Cactus` a way to interact with Besu networks. Using this we can perform:

- Deploy Smart-contracts through bytecode.
- Build and sign transactions using different keystores.
- Invoke smart-contract functions that we have deployed on the network.

10.6.1 Summary

- Getting Started
- Architecture
- Usage
- Prometheus Exporter
- Runing the tests
- Built With
- Contributing
- License
- Acknowledgments

10.6.2 Getting Started

Clone the git repository on your local machine. Follow these instructions that will get you a copy of the project up and running on your local machine for development and testing purposes.

Prerequisites

In the root of the project to install the dependencies execute the command:

```
npm run configure
```

Compiling

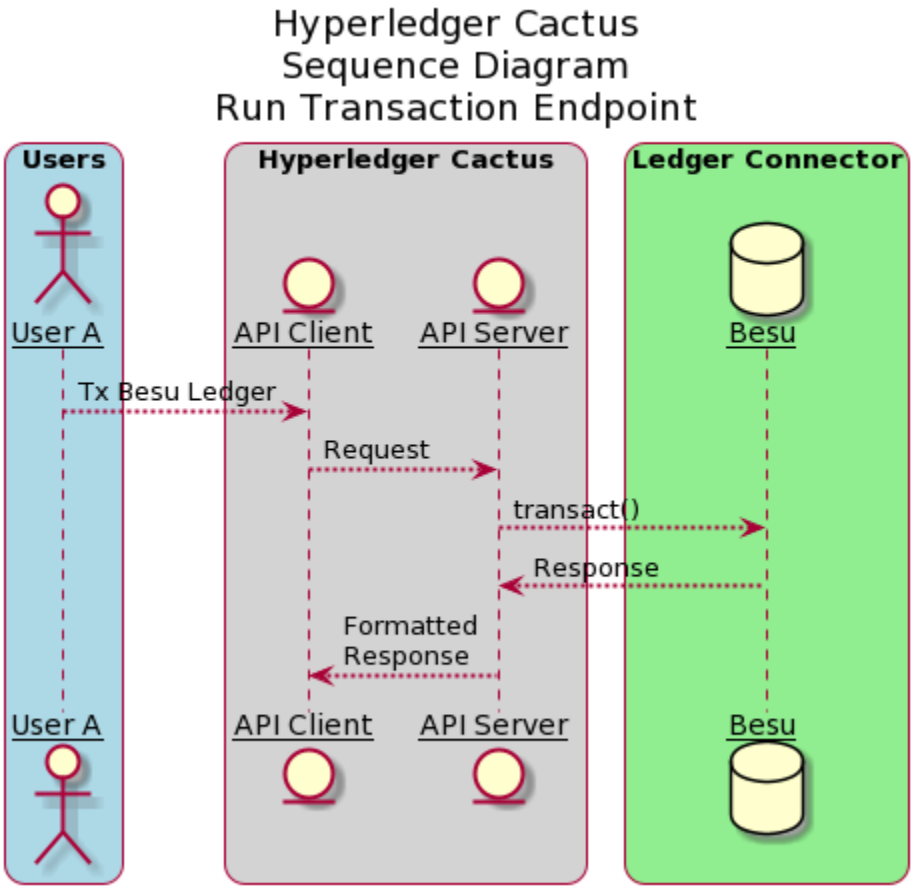
In the project root folder, run this command to compile the plugin and create the dist directory:

```
npm run tsc
```

Architecture

The sequence diagrams for various endpoints are mentioned below

run-transaction-endpoint

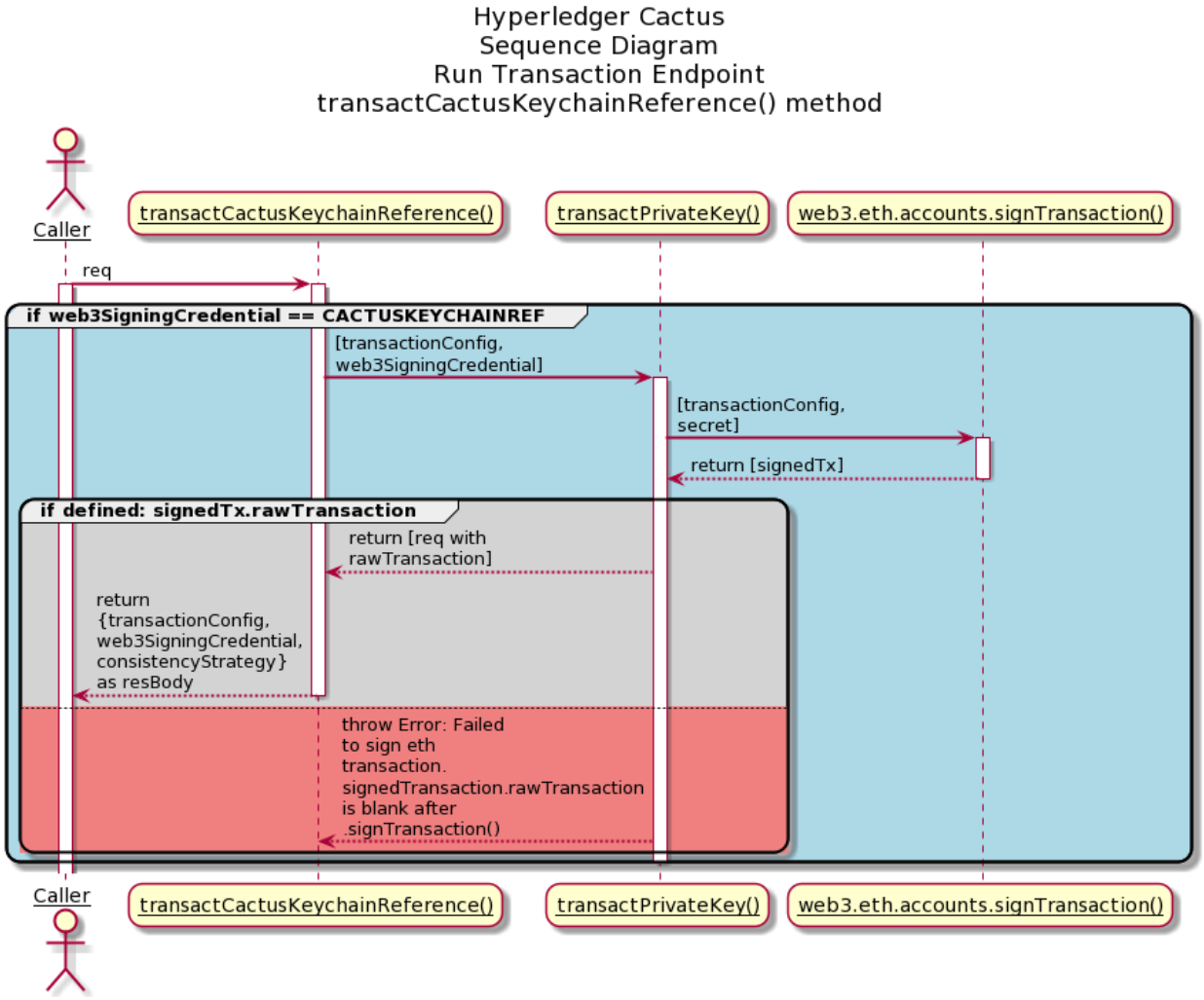


The above diagram shows the sequence diagram of run-transaction-endpoint. User A (One of the many Users) interacts with the API Client which in turn, calls the API server. API server then executes transact() method which is explained in detailed in the subsequent

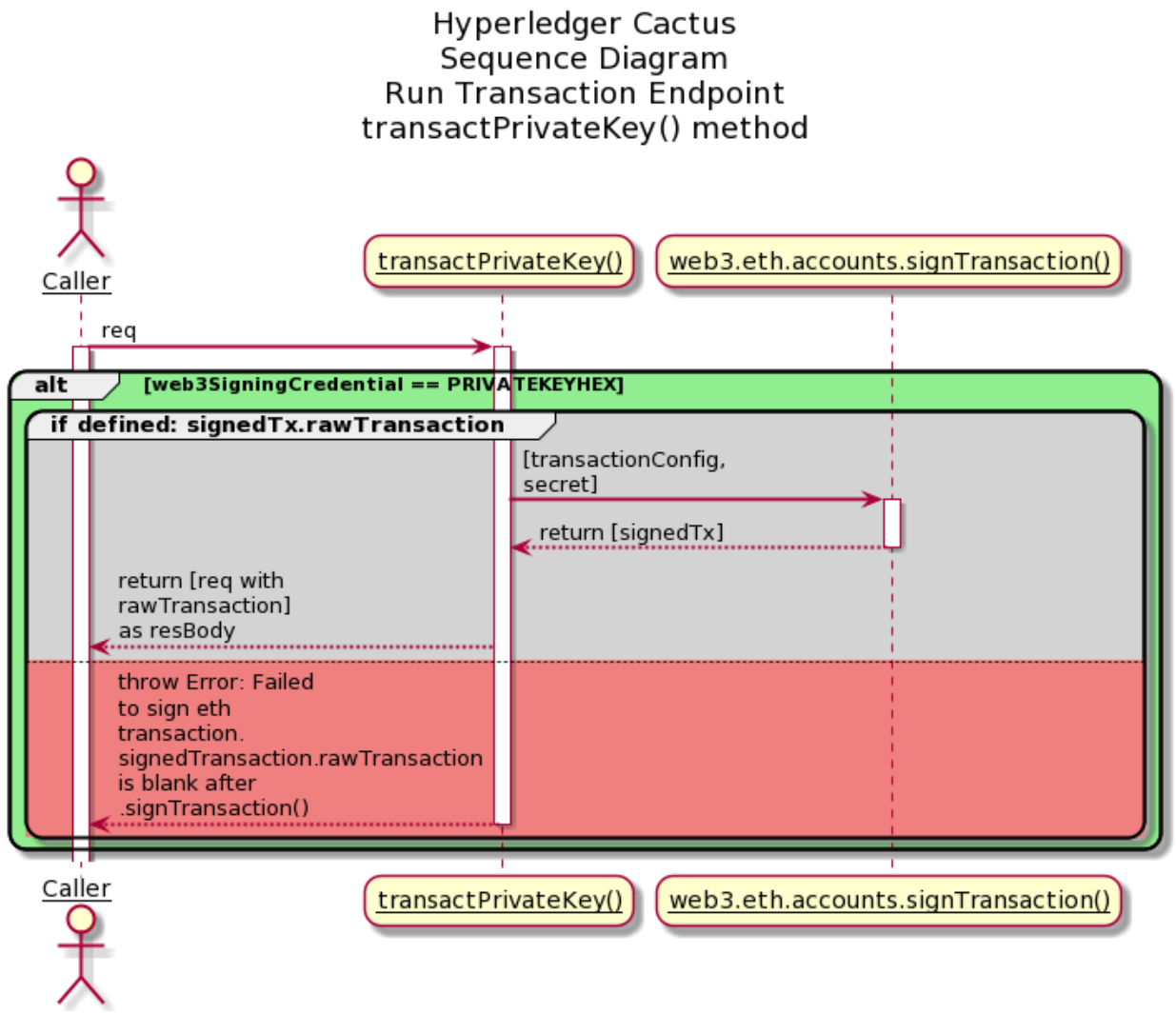

Hyperledger Cactus
Sequence Diagram
Run Transaction Endpoint
transact() method
Caller
«class»
PluginLedgerConnectorBesu
transact(RunTransactionRequest)
alt
[web3SigningCredential == CACTUSKEYCHAINREF]
transactCactusKeychainRef(RunTransactionRequest)
RunTransactionResponse
return
RunTransactionResponse
[web3SigningCredential == PRIVATEKEYHEX]
if defined: signedTx.rawTransaction
transactPrivateKey(RunTransactionRequest)
RunTransactionResponse
return
RunTransactionResponse
throw Error: Failed
to sign eth
transaction.
signedTransaction.rawTransaction
is blank after
.signTransaction()
[web3SigningCredential == NONE]
if defined: req.transactionConfig.rawTransaction
transactSigned(RunTransactionRequest)
RunTransactionResponse
return
RunTransactionResponse
throw Error:
Expected
pre-signed raw
transaction
[default]
throw Error:
Unrecognized
Web3SigningCredentialType
Caller
«class»
PluginLedgerConnectorBesu

diagrams.

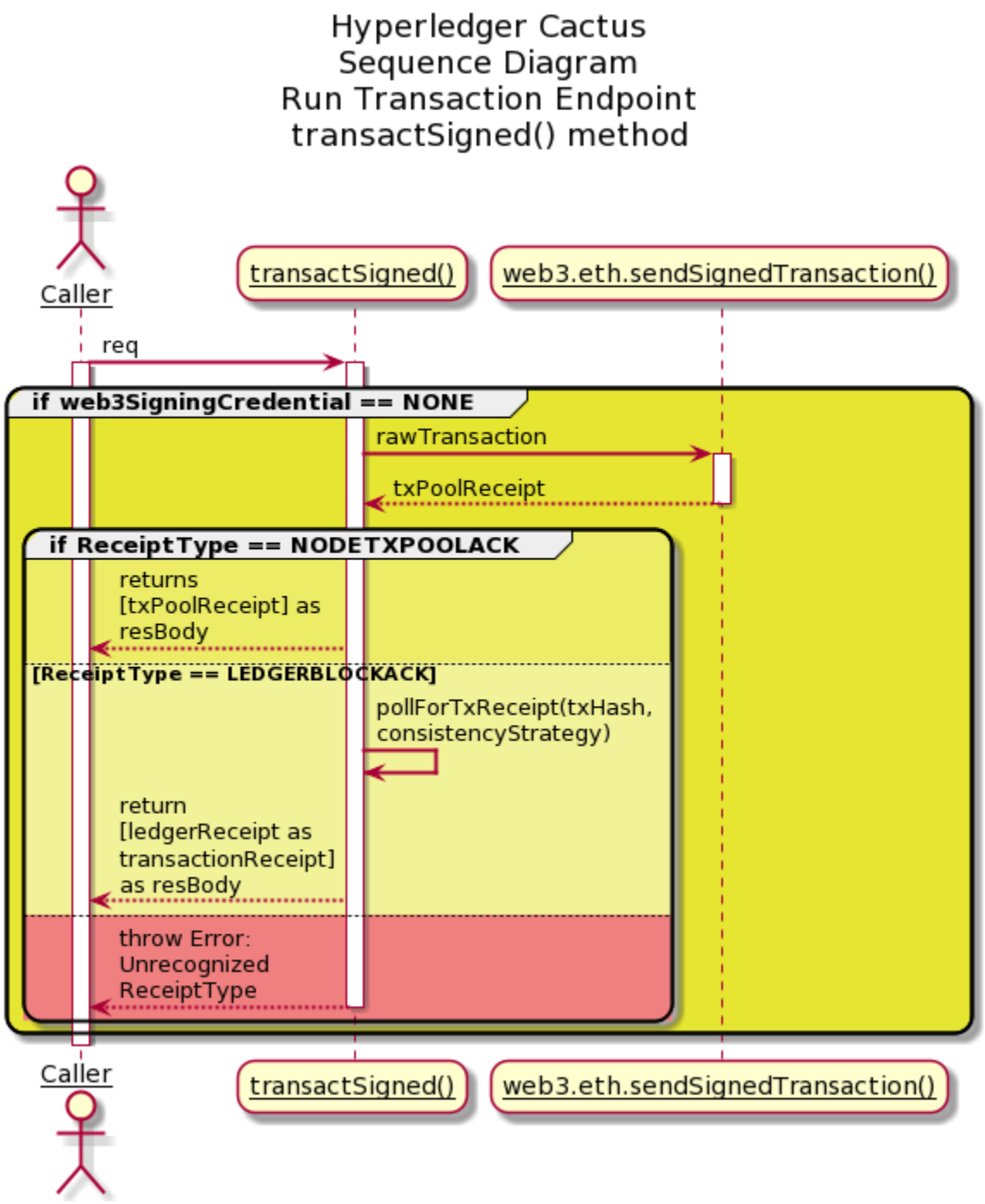
The above diagram shows the sequence diagraom of transact() method of the PluginLedgerConnectorBesu class. The caller to this function, which in reference to the above sequence diagram is API server, sends RunTransactionRequest object as an argument to the transact() method. Based on the type of Web3SigningCredentialType, corresponsing responses are sent back to the caller.



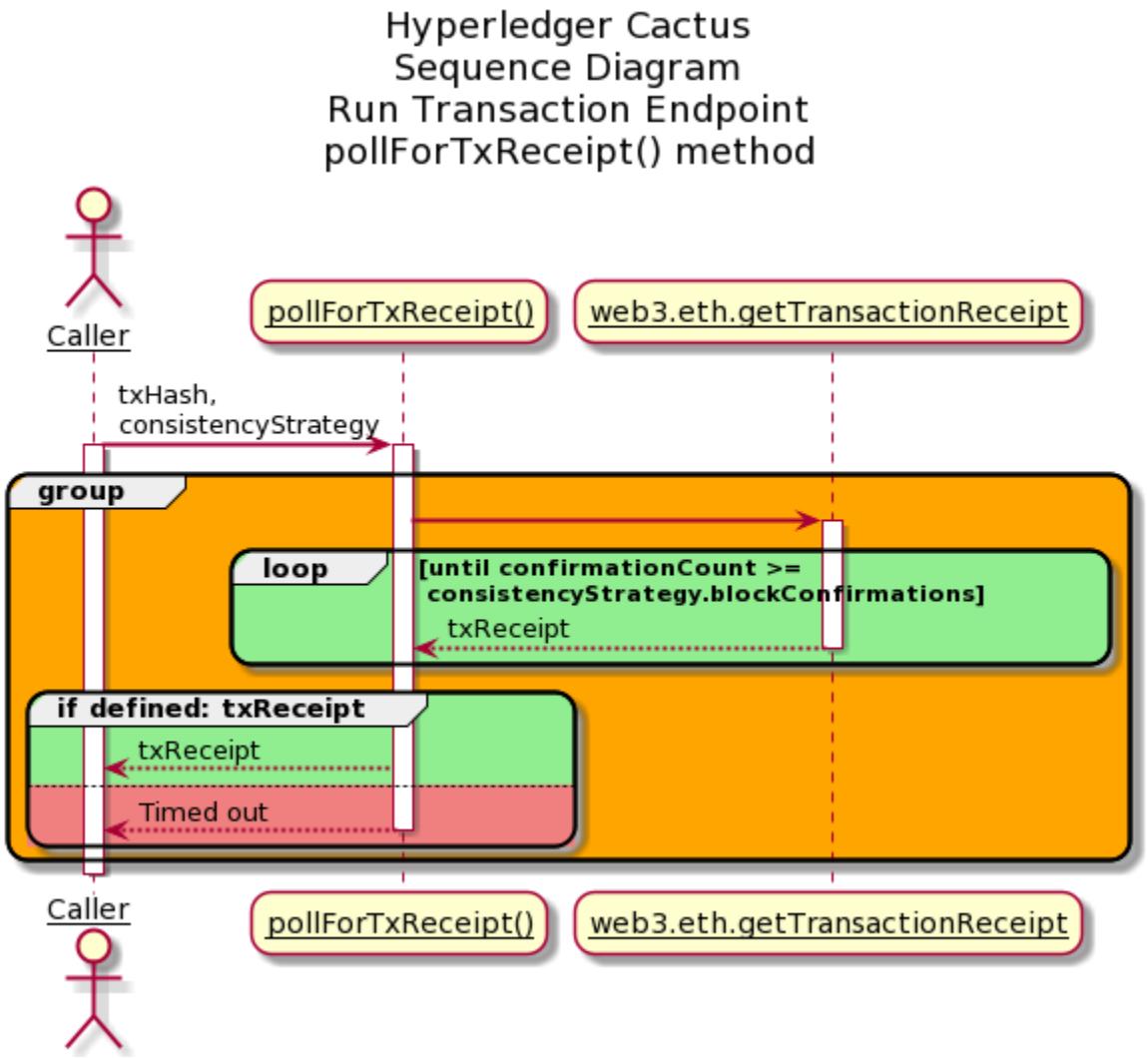
The above diagram shows transactCactusKeychainReference() method being called by the transact() method of the PluginLedgerConnector class when the Web3SigningCredentialType is CACTUSKEYCHAINREF. This method inturn calls transactPrivateKey() which calls the signTransaction() method of web3 library.



The above diagram shows transactPrivateKey() method being called by the transact() method of the PluginLedgerConnector class when the Web3SigningCredentialType is PRIVATEKEYHEX. This method then calls the signTransaction() method of the web3 library.



The above diagram shows transactSigned() method being called by the transact() method of the PluginLedgerConnector class when the Web3SigningCredentialType is NONE. This method calls the sendSignedTransaction() of the web3 library and then calls pollForTxReceipt() method.



The above diagram shows pollForTxReceipt() method which is called by the transactSigned() method as described in the previous sequence diagram. This method waits for the block confirmation in a loop and then sends the corresponding response back to the caller.

Usage

To use this import public-api and create new **PluginFactoryLedgerConnector**. Then use it to create a connector.

```
const factory = new PluginFactoryLedgerConnector({
    pluginImportType: PluginImportType.LOCAL,
  });
  const connector: PluginLedgerConnectorBesu = await factory.create({
    rpcApiHttpHost,
    instanceId: uuidv4(),
    pluginRegistry: new PluginRegistry(),
  });
```

You can make calls through the connector to the plugin API:

```
async invokeContract(req: InvokeContractV1Request):Promise<InvokeContractV1Response>;
async transactSigned(rawTransaction: string): Promise<RunTransactionResponse>;
async transactPrivateKey(req: RunTransactionRequest): Promise<RunTransactionResponse>;
async transactCactusKeychainRef(req: RunTransactionRequest):Promise
↪<RunTransactionResponse>;
async deployContract(req: DeployContractSolidityBytecodeV1Request):Promise
↪<RunTransactionResponse>;
async signTransaction(req: SignTransactionRequest):Promise<Optional
↪<SignTransactionResponse>>;
```

Call example to deploy a contract:

```
const deployOut = await connector.deployContract({
  web3SigningCredential: {
    ethAccount: firstHighNetWorthAccount,
    secret: besuKeyPair.privateKey,
    type: Web3SigningCredentialType.PrivateKeyHex,
  },
  bytecode: SmartContractJson.bytecode,
  gas: 1000000,
});
```

The field "type" can have the following values:

```
enum Web3SigningCredentialType {
    CACTUSKEYCHAINREF = 'CACTUS_KEYCHAIN_REF',
    GETHKEYCHAINPASSWORD = 'GETH_KEYCHAIN_PASSWORD',
    PRIVATEKEYHEX = 'PRIVATE_KEY_HEX',
    NONE = 'NONE'
}
```

Transaction Privacy Feature

Private transactions using Besu are currently enabled.

The privateFor and privateFrom fields must be populated, more information about Besu Private Transactions here.

Call example to deploy a private contract:

```
  const deployOut = await connector1.deployContract({
    bytecode: SmartContract.bytecode,
    contractAbi: SmartContract.abi,
    contractName: SmartContract.contractName,
    constructorArgs: [],
    privateTransactionConfig: {
      privateFrom: SendingTesseraPublicKey,
      privateFor: [
        Member1TesseraPrivateKey,
        Member2TesseraPrivateKey,
      ],
    },
    web3SigningCredential: {
      secret:SendingBesuPrivateKey,
```

(continues on next page)

(continued from previous page)

```
      type: Web3SigningCredentialType.PrivateKeyHex,
    },
    gas: 3000000
  });
```

Extensive documentation and examples in the readthedocs (WIP)

Building/running the container image locally

In the Cactus project root say:

```
DOCKER_BUILDKIT=1 docker build -f ./packages/cactus-plugin-ledger-connector-besu/
↪Dockerfile . -t cplcb
```

Build with a specific version of the npm package:

```
DOCKER_BUILDKIT=1 docker build --build-arg NPM_PKG_VERSION=0.4.1 -f ./packages/cactus-
↪plugin-ledger-connector-besu/Dockerfile . -t cplcb
```

Running the container

Launch container with plugin configuration as an **environment variable**:

```
docker run \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  --env PLUGINS='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-besu",
↪"type": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.
↪cactus.plugin_import_action.INSTALL",  "options": {"rpcApiHttpHost": "http://
↪localhost:8545", "instanceId": "some-unique-besu-connector-instance-id"}}]' \
  cplcb
```

Launch container with plugin configuration as a **CLI argument**:

```
docker run \
  --rm \
  --publish 3000:3000 \
   --publish 4000:4000 \
  cplcb \
    ./node_modules/.bin/cactusapi \
    --plugins='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-besu", "type
↪": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.
↪cactus.plugin_import_action.INSTALL",  "options": {"rpcApiHttpHost": "http://
↪localhost:8545", "instanceId": "some-unique-besu-connector-instance-id"}}]'
```

Launch container with **configuration file** mounted from host machine:

```
echo '[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-besu", "type": "org.
↪hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.cactus.plugin_
↪import_action.INSTALL",  "options": {"rpcApiHttpHost": "http://localhost:8545",
↪"instanceId": "some-unique-besu-connector-instance-id"}}]' > cactus.json

docker run \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  --mount type=bind,source="$(pwd)"/cactus.json,target=/cactus.json \
  cplcb \
    ./node_modules/.bin/cactusapi \
    --config-file=/cactus.json
```

Testing API calls with the container

Don't have a Besu network on hand to test with? Test or develop against our Besu All-In-One container!

Terminal Window 1 (Ledger)

```
docker run -p 0.0.0.0:8545:8545/tcp  -p 0.0.0.0:8546:8546/tcp  -p 0.0.0.0:8888:8888/tcp ␣
↪-p 0.0.0.0:9001:9001/tcp  -p 0.0.0.0:9545:9545/tcp hyperledger/cactus-besu-all-in-
↪one:latest
```

Terminal Window 2 (Cactus API Server)

```
docker run \
  --network host \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  --env PLUGINS='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-besu",
↪"type": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.
↪cactus.plugin_import_action.INSTALL",  "options": {"rpcApiHttpHost": "http://
↪localhost:8545", "instanceId": "some-unique-besu-connector-instance-id"}}]' \
  cplcb
```

Terminal Window 3 (curl - replace eth accounts as needed)

```
curl --location --request POST 'http://127.0.0.1:4000/api/v1/plugins/@hyperledger/cactus-
↪plugin-ledger-connector-besu/run-transaction' \
--header 'Content-Type: application/json' \
--data-raw '{
    "web3SigningCredential": {
      "ethAccount": "627306090abaB3A6e1400e9345bC60c78a8BEf57",
      "secret": "c87509a1c067bbde78beb793e6fa76530b6382a4c0241e5e4a9ec0a0f44dc0d3",
      "type": "PRIVATE_KEY_HEX"
    },
    "consistencyStrategy": {
      "blockConfirmations": 0,
      "receiptType": "NODE_TX_POOL_ACK"
    },
```

(continues on next page)

(continued from previous page)

```
    "transactionConfig": {
      "from": "627306090abaB3A6e1400e9345bC60c78a8BEf57",
      "to": "f17f52151EbEF6C7334FAD080c5704D77216b732",
      "value": 1,
      "gas": 10000000
    }
}'
```

The above should produce a response that looks similar to this:

```
{
    "success": true,
    "data": {
        "transactionReceipt": {
            "blockHash":
↪"0x7c97c038a5d3bd84613fe23ed442695276d5d2df97f4e7c4f10ca06765033ffd",
            "blockNumber": 1218,
            "contractAddress": null,
            "cumulativeGasUsed": 21000,
            "from": "0x627306090abab3a6e1400e9345bc60c78a8bef57",
            "gasUsed": 21000,
            "logs": [],
            "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
            "status": true,
            "to": "0xf17f52151ebef6c7334fad080c5704d77216b732",
            "transactionHash":
↪"0xc7fcb46c735bdc696d500bfc70c72595a2b8c31813929e5c61d9a5aec3376d6f",
            "transactionIndex": 0
        }
    }
}
```

10.6.3 Prometheus Exporter

This class creates a prometheus exporter, which scrapes the transactions (total transaction count) for the use cases incorporating the use of Besu connector plugin.

Prometheus Exporter Usage

The prometheus exporter object is initialized in the `PluginLedgerConnectorBesu` class constructor itself, so instantiating the object of the `PluginLedgerConnectorBesu` class, gives access to the exporter object. You can also initialize the prometheus exporter object seperately and then pass it to the `IPluginLedgerConnectorBesuOptions` interface for `PluginLedgerConnectoBesu` constructor.

`getPrometheusMetricsV1` function returns the prometheus exporter metrics, currently displaying the total transaction count, which currently increments everytime the `transact()` method of the `PluginLedgerConnectorBesu` class is called.

Prometheus Integration

To use Prometheus with this exporter make sure to install Prometheus main component. Once Prometheus is setup, the corresponding scrape_config needs to be added to the prometheus.yml

```
- job_name: 'besu_ledger_connector_exporter'
  metrics_path: api/v1/plugins/@hyperledger/cactus-plugin-ledger-connector-besu/get-
↪prometheus-exporter-metrics
  scrape_interval: 5s
  static_configs:
    - targets: ['{host}:{port}']
```

Here the `host:port` is where the prometheus exporter metrics are exposed. The test cases (For example, packages/cactus-plugin-ledger-connector-besu/src/test/typescript/integration/plugin-ledger-connector-besu/deploy-contract/deploy-contract-from-json.test.ts) exposes it over `0.0.0.0` and a random port(). The random port can be found in the running logs of the test case and looks like (42379 in the below mentioned URL) `Metrics URL: http://0.0.0.0:42379/api/v1/plugins/@hyperledger/cactus-plugin-ledger-connector-besu/get-prometheus-exporter-metrics`

Once edited, you can start the prometheus service by referencing the above edited prometheus.yml file. On the prometheus graphical interface (defaulted to http://localhost:9090), choose **Graph** from the menu bar, then select the **Console** tab. From the **Insert metric at cursor** drop down, select **cactus_besu_total_tx_count** and click **execute**

Helper code

response.type.ts

This file contains the various responses of the metrics.

data-fetcher.ts

This file contains functions encasing the logic to process the data points

metrics.ts

This file lists all the prometheus metrics and what they are used for.

10.6.4 Running the tests

To check that all has been installed correctly and that the pugin has no errors run the tests:

- Run this command at the project's root:

```
npm run test:plugin-ledger-connector-besu
```

10.6.5 Contributing

We welcome contributions to Hyperledger Cactus in many forms, and there's always plenty to do!

Please review CONTIRBUTING.md to get started.

10.6.6 License

This distribution is published under the Apache License Version 2.0 found in the LICENSE file.

10.6.7 Acknowledgments

10.7 @hyperledger/cactus-plugin-ledger-connector-corda

10.7.1 Table of Contents

10.7.2 Summary

The Corda connector is written in Kotlin and ships as a Spring Boot JVM application that accepts API requests and translates those into Corda RPC calls.

Deploying the Corda connector therefore involves also deploying the mentioned JVM application **in addition** to deploying the Cactus API server with the desired plugins configured.

10.7.3 Concepts

Contract Invocation JSON DSL

One of our core design principles for Hyperledger Cactus is to have low impact deployments meaning that changes to the ledgers themselves should be kept to a minimum or preferably have no need for any at all. With this in mind, we had to solve the challenge of providing users with the ability to invoke Corda flows as dynamically as possible within the confines of the strongly typed JVM contrasted with the weakly typed Javascript language runtime of NodeJS.

Corda might release some convenience features to ease this in the future, but in the meantime we have the *Contract Invocation JSON DSL* which allows developers to specify truly arbitrary JVM types as part of their contract invocation arguments even if otherwise these types would not be possible to serialize or deserialize with traditional tooling such as the excellent Jackson JSON Java library or similar ones.

Expressing Primitive vs Reference Types with the DLS

The features of the DSL include expressing whether a contract invocation parameter is a reference or a primitive JVM data types. This is a language feature that Javascript has as well to some extent, but for those in need of a refresher, here's a writeup from a well known Q/A website that I found on the internet: What's the difference between primitive and reference types?

To keep it simple, the following types are primitive data types in the Java Virtual Machine (JVM) and everything else not included in the list below can be safely considered a reference type:

- boolean
- byte
- short
- char
- int
- long
- float
- double

If you'd like to further clarify how this works and feel like an exciting adventure then we recommend that you dive into the source code of the deserializer implementation of the JSON DSL and take a look at the following points of interest in the code located there:

- `val exoticTypes: Map<String, Class<*>>`
- `fun instantiate(jvmObject: JvmObject)`

Flow Invocation Types

Can be **dynamic** or **tracked dynamic** and the corresponding enum values are defined as:

```
/**
 * Determines which flow starting method will be used on the back-end when invoking the␣
↪flow. Based on the value here the plugin back-end might invoke the rpc.
↪startFlowDynamic() method or the rpc.startTrackedFlowDynamic() method. Streamed␣
↪responses are aggregated and returned in a single response to HTTP callers who are not␣
↪equipped to handle streams like WebSocket/gRPC/etc. do.
```

(continues on next page)

(continued from previous page)

```
 * @export
 * @enum {string}
 */
export enum FlowInvocationType {
    TRACKEDFLOWDYNAMIC = 'TRACKED_FLOW_DYNAMIC',
    FLOWDYNAMIC = 'FLOW_DYNAMIC'
}
```

Official Corda Java Docs - startFlowDynamic()

Official Corda Java Docs - startTrackedFlowDynamic()

10.7.4 Usage

Take a look at how the API client can be used to run transactions on a Corda ledger: `packages/cactus-plugin-ledger-connector-corda/src/test/typescript/integration/jvm-kotlin-spring-server.test.ts`

Invoke Contract (flow) with no parameters

Below, we'll demonstrate invoking a simple contract with no parameters.

The contract source:

```
package com.example.organization.samples.application.flows;

class SomeCoolFlow {
  // constructor with no arguments
  public SomeCoolFlow() {
    this.doSomething();
  }

  public doSomething(): void {
    throw new RuntimeException("Method not implemented.");
  }
}
```

Steps to build your request:

1. Find out the fully qualified class name of your contract (flow) and set this as the value for the request parameter `flowFullClassName`
2. Decide on your flow invocation type which largely comes down to answering the question of: Does your invocation follow a request/response pattern or more like a channel subscription where multiple updates at different times are streamed to the client in response to the invocation request? In our example we assume the simpler request/response communication pattern and therefore will set the `flowInvocationType` to `FlowInvocationType.FLOWDYNAMIC`
3. Invoke the flow via the API client with the `params` argument being specified as an empty array `[]`

```
import { DefaultApi as CordaApi } from "@hyperledger/cactus-plugin-ledger-connector-
↪corda";
import { FlowInvocationType } from "@hyperledger/cactus-plugin-ledger-connector-
↪corda";
```

(continues on next page)

(continued from previous page)

```
const apiUrl = "your-cactus-host.example.com"; // don't forget to specify the port␣
↪if applicable
const apiClient = new CordaApi({ basePath: apiUrl });

const res = await apiClient.invokeContractV1({
  flowFullClassName: "com.example.organization.samples.application.flows.
↪SomeCoolFlow",
  flowInvocationType: FlowInvocationType.FLOWDYNAMIC,
  params: [],
  timeoutMs: 60000,
});
```

Invoke Contract (flow) with a single integer parameter

Below, we'll demonstrate invoking a simple contract with a single numeric parameter.

The contract source:

```
package com.example.organization.samples.application.flows;

class SomeCoolFlow {
  // constructor with a primitive type long argument
  public SomeCoolFlow(long myParameterThatIsLong) {
    // do something with the parameter here
  }
}
```

Steps to build your request:

1. Find out the fully qualified class name of your contract (flow) and set this as the value for the request parameter `flowFullClassName`
2. Decide on your flow invocation type. More details at Invoke Contract (flow) with no parameters
3. Find out what is the fully qualified class name of the parameter you wish to pass in. You can do this be inspecting the sources of the contract itself. If you do not have access to those sources, then the documentation of the contract should have answers or the person who authored said contract. In our case here the fully qualified class name for the number parameter is simply `long` because it is a primitive data type and as such these can be referred to in their short form, but the fully qualified version also works such as: `java.lang.Long`. When in doubt about these, you can always consult the official java.lang.Long Java Docs After having determined the above, you can construct your first `JvmObject` JSON object as follows in order to pass in the number `42` as the first and only parameter for our flow invocation:

```
params: [
  {
    jvmTypeKind: JvmTypeKind.PRIMITIVE,
    jvmType: {
      fqClassName: "long",
    },
    primitiveValue: 42,
  }
]
```

4. Invoke the flow via the API client with the `params` populated as explained above:

```
import { DefaultApi as CordaApi } from "@hyperledger/cactus-plugin-ledger-connector-
↪corda";
import { FlowInvocationType } from "@hyperledger/cactus-plugin-ledger-connector-
↪corda";

// don't forget to specify the port if applicable
const apiUrl = "your-cactus-host.example.com";
const apiClient = new CordaApi({ basePath: apiUrl });

const res = await apiClient.invokeContractV1({
  flowFullClassName: "com.example.organization.samples.application.flows.
↪SomeCoolFlow",
  flowInvocationType: FlowInvocationType.FLOWDYNAMIC,
  params: [
    {
      jvmTypeKind: JvmTypeKind.PRIMITIVE,
      jvmType: {
        fqClassName: "long",
      },
      primitiveValue: 42,
    }
  ],
  timeoutMs: 60000,
});
```

Invoke Contract (flow) with a custom class parameter

Below, we'll demonstrate invoking a contract with a single class instance parameter.

The contract sources:

```
package com.example.organization.samples.application.flows;

// contract with a class instance parameter
class BuildSpaceshipFlow {
  public BuildSpaceshipFlow(SpaceshipInfo buildSpecs) {
    // build spaceship as per the specs
  }
}
```

```
package com.example.organization.samples.application.flows;

// The type that the contract accepts as an input parameter
class SpaceshipInfo {
  public SpaceshipInfo(String name, Integer seatsForHumans) {
  }
}
```

Assembling and Sending your request:

Invoke the flow via the API client with the `params` populated as shown below.

Key thing notice here is that we now have a class instance as a parameter for our contract (flow) invocation so we have to describe how this class instance itself will be instantiated by providing a nested array of parameters via the `jvmCtorArgs` which stands for Java Virtual Machine Constructor Arguments meaning that elements of this array will be passed in dynamically (via Reflection) to the class constructor.

Java Equivalent

```
cordaRpcClient.startFlowDynamic(
  BuildSpaceshipFlow.class,
  new SpaceshipInfo(
    "The last spaceship you'll ever need.",
    10000000
  )
);
```

Cactus Invocation JSON DLS Equivalent to the Above Java Snippet

```
import { DefaultApi as CordaApi } from "@hyperledger/cactus-plugin-ledger-connector-corda
↪";
import { FlowInvocationType } from "@hyperledger/cactus-plugin-ledger-connector-corda";

// don't forget to specify the port if applicable
const apiUrl = "your-cactus-host.example.com";
const apiClient = new CordaApi({ basePath: apiUrl });

const res = await apiClient.invokeContractV1({
  flowFullClassName: "com.example.organization.samples.application.flows.
↪BuildSpaceshipFlow",
  flowInvocationType: FlowInvocationType.FLOWDYNAMIC,
  params: [
    {
      jvmTypeKind: JvmTypeKind.REFERENCE,
        jvmType: {
        fqClassName: "com.example.organization.samples.application.flows.SpaceshipInfo",
      },

      jvmCtorArgs: [
        {
          jvmTypeKind: JvmTypeKind.PRIMITIVE,
          jvmType: {
            fqClassName: "java.lang.String",
          },
          primitiveValue: "The last spaceship you'll ever need.",
        },
        {
          jvmTypeKind: JvmTypeKind.PRIMITIVE,
          jvmType: {
            fqClassName: "java.lang.Long",
          },
          primitiveValue: 10000000000,
        },
      ],
    }
  ],
```

(continues on next page)

(continued from previous page)

```
  timeoutMs: 60000,
});
```

Custom Configuration via Env Variables

```
{
  "cactus": {
    "corda": {
      "node": {
        "host": "localhost"
      },
      "rpc": {
        "port": 10006,
        "username": "user1",
        "password": "test"
      }
    }
  }
}
```

```
SPRING_APPLICATION_JSON='{"cactus":{"corda":{"node": {"host": "localhost"}, "rpc":{"port
↪": 10006, "username":"user1", "password": "test"}}}}' gradle test
```

```
{
  "flowFullClassName" : "net.corda.samples.example.flows.ExampleFlow${"$"}Initiator",
  "flowInvocationType" : "FLOW_DYNAMIC",
  "params" : [ {
    "jvmTypeKind" : "PRIMITIVE",
    "jvmType" : {
      "fqClassName" : "java.lang.Integer"
    },
    "primitiveValue" : 42,
    "jvmCtorArgs" : null
  }, {
    "jvmTypeKind" : "REFERENCE",
    "jvmType" : {
      "fqClassName" : "net.corda.core.identity.Party"
    },
    "primitiveValue" : null,
    "jvmCtorArgs" : [ {
      "jvmTypeKind" : "REFERENCE",
      "jvmType" : {
        "fqClassName" : "net.corda.core.identity.CordaX500Name"
      },
      "primitiveValue" : null,
      "jvmCtorArgs" : [ {
        "jvmTypeKind" : "PRIMITIVE",
        "jvmType" : {
          "fqClassName" : "java.lang.String"
        },
```

(continues on next page)

(continued from previous page)


```
        "primitiveValue" : "PartyB",
        "jvmCtorArgs" : null
      }, {
        "jvmTypeKind" : "PRIMITIVE",
        "jvmType" : {
          "fqClassName" : "java.lang.String"
        },
        "primitiveValue" : "New York",
        "jvmCtorArgs" : null
      }, {
        "jvmTypeKind" : "PRIMITIVE",
        "jvmType" : {
          "fqClassName" : "java.lang.String"
        },
        "primitiveValue" : "US",
        "jvmCtorArgs" : null
      } ]
    }, {
      "jvmTypeKind" : "REFERENCE",
      "jvmType" : {
        "fqClassName" : "org.hyperledger.cactus.plugin.ledger.connector.corda.server.
↪impl.PublicKeyImpl"
      },
      "primitiveValue" : null,
      "jvmCtorArgs" : [ {
        "jvmTypeKind" : "PRIMITIVE",
        "jvmType" : {
          "fqClassName" : "java.lang.String"
        },
        "primitiveValue" : "EdDSA",
        "jvmCtorArgs" : null
      }, {
        "jvmTypeKind" : "PRIMITIVE",
        "jvmType" : {
          "fqClassName" : "java.lang.String"
        },
        "primitiveValue" : "X.509",
        "jvmCtorArgs" : null
      }, {
        "jvmTypeKind" : "PRIMITIVE",
        "jvmType" : {
          "fqClassName" : "java.lang.String"
        },
        "primitiveValue" : "MCowBQYDK2VwAyEAoOv19eiCDJ7HzR9UrfwbFig7qcD1jkewKkkS4WF9kPA=
↪",
        "jvmCtorArgs" : null
      } ]
    } ]
  } ],
  "timeoutMs" : null
}
```

```
I 16:51:01 1 Client.main - nodeDiagnosticInfo=
{
  "version" : "4.6",
  "revision" : "85e387ea730d9be7d6dc2b23caba1ee18305af74",
  "platformVersion" : 8,
  "vendor" : "Corda Open Source",
  "cordapps" : [ {
    "type" : "Workflow CorDapp",
    "name" : "workflows-1.0",
    "shortName" : "Example-Cordapp Flows",
    "minimumPlatformVersion" : 8,
    "targetPlatformVersion" : 8,
    "version" : "1",
    "vendor" : "Corda Open Source",
    "licence" : "Apache License, Version 2.0",
    "jarHash" : {
      "offset" : 0,
      "size" : 32,
      "bytes" : "V7ssTw0etgg3nSGk1amArB+fBH8fQUyBwIFs0DhID+0="
    }
  }, {
    "type" : "Contract CorDapp",
    "name" : "contracts-1.0",
    "shortName" : "Example-Cordapp Contracts",
    "minimumPlatformVersion" : 8,
    "targetPlatformVersion" : 8,
    "version" : "1",
    "vendor" : "Corda Open Source",
    "licence" : "Apache License, Version 2.0",
    "jarHash" : {
      "offset" : 0,
      "size" : 32,
      "bytes" : "Xe0eoh4+T6fsq4u0QKqkVsVDMYSWhuspHqE0wlOlyqU="
    }
  } ]
}
```

10.7.5 Building Docker Image Locally

The `cccs` tag used in the below example commands is a shorthand for the full name of the container image otherwise referred to as `cactus-corda-connector-server`.

From the project root:

```
DOCKER_BUILDKIT=1 docker build ./packages/cactus-plugin-ledger-connector-corda/src/main-
↪server/ -t cccs
```

10.7.6 Example NodeDiagnosticInfo JSON Response


```
{
  "version": "4.6",
  "revision": "85e387ea730d9be7d6dc2b23caba1ee18305af74",
  "platformVersion": 8,
  "vendor": "Corda Open Source",
  "cordapps": [
    {
      "type": "Workflow CorDapp",
      "name": "workflows-1.0",
      "shortName": "Obligation Flows",
      "minimumPlatformVersion": 8,
      "targetPlatformVersion": 8,
      "version": "1",
      "vendor": "Corda Open Source",
      "licence": "Apache License, Version 2.0",
      "jarHash": {
        "bytes": "Vf9MllnrC7vrWxrlDE94OzPMZW7At1HhTETL/XjiAmc=",
        "offset": 0,
        "size": 32
      }
    },
    {
      "type": "CorDapp",
      "name": "corda-confidential-identities-4.6",
      "shortName": "corda-confidential-identities-4.6",
      "minimumPlatformVersion": 1,
      "targetPlatformVersion": 1,
      "version": "Unknown",
      "vendor": "Unknown",
      "licence": "Unknown",
      "jarHash": {
        "bytes": "nqBwqHJMbLW80hmRbKEYk0eAknFiX8N40LKuGsD0bPo=",
        "offset": 0,
        "size": 32
      }
    },
    {
      "type": "Contract CorDapp",
      "name": "corda-finance-contracts-4.6",
      "shortName": "Corda Finance Demo",
      "minimumPlatformVersion": 1,
      "targetPlatformVersion": 8,
      "version": "1",
      "vendor": "R3",
      "licence": "Open Source (Apache 2)",
      "jarHash": {
        "bytes": "a43Q/GJG6JKTZzq3U80P8L1DWWcB/D+Pl5uitEtAeQQ=",
        "offset": 0,
        "size": 32
      }
    },
```


(continues on next page)

(continued from previous page)


```
    {
      "type": "Workflow CorDapp",
      "name": "corda-finance-workflows-4.6",
      "shortName": "Corda Finance Demo",
      "minimumPlatformVersion": 1,
      "targetPlatformVersion": 8,
      "version": "1",
      "vendor": "R3",
      "licence": "Open Source (Apache 2)",
      "jarHash": {
        "bytes": "wXdD4Iy50RaWzPp7n9s1xwf4K4MB8eA1nmhPquTMvxg=",
        "offset": 0,
        "size": 32
      }
    },
    {
      "type": "Contract CorDapp",
      "name": "contracts-1.0",
      "shortName": "Obligation Contracts",
      "minimumPlatformVersion": 8,
      "targetPlatformVersion": 8,
      "version": "1",
      "vendor": "Corda Open Source",
      "licence": "Apache License, Version 2.0",
      "jarHash": {
        "bytes": "grTZzN71Cpxw6rZe/U5SB6/ehl99B6VQ1+ZJEx1rixs=",
        "offset": 0,
        "size": 32
      }
    }
  ]
}
```


10.7.7 Monitoring

Usage Prometheus

The prometheus exporter object is initialized in the `PluginLedgerConnectorCorda` class constructor itself, so instantiating the object of the `PluginLedgerConnectorCorda` class, gives access to the exporter object. You can also initialize the prometheus exporter object seperately and then pass it to the `IPluginLedgerConnectorCordaOptions` interface for `PluginLedgerConnectoCorda` constructor.

`getPrometheusExporterMetricsEndpointV1` function returns the prometheus exporter metrics, currently displaying the total transaction count, which currently increments everytime the `transact()` method of the `PluginLedgerConnectorCorda` class is called.

Prometheus Integration

To use Prometheus with this exporter make sure to install Prometheus main component. Once Prometheus is setup, the corresponding scrape_config needs to be added to the prometheus.yml

```
- job_name: 'corda_ledger_connector_exporter'
  metrics_path: api/v1/plugins/@hyperledger/cactus-plugin-ledger-connector-corda/get-
↪prometheus-exporter-metrics
  scrape_interval: 5s
  static_configs:
    - targets: ['{host}:{port}']
```

Here the `host:port` is where the prometheus exporter metrics are exposed. The test cases (For example, packages/cactus-plugin-ledger-connector-corda/src/test/typescript/integration/deploy-cordapp-jars-to-nodes.test.ts) exposes it over `0.0.0.0` and a random port(). The random port can be found in the running logs of the test case and looks like (42379 in the below mentioned URL) `Metrics URL: http://0.0.0.0:42379/api/v1/plugins/@hyperledger/cactus-plugin-ledger-connector-corda/get-prometheus-exporter-metrics`

Once edited, you can start the prometheus service by referencing the above edited prometheus.yml file. On the prometheus graphical interface (defaulted to http://localhost:9090), choose **Graph** from the menu bar, then select the **Console** tab. From the **Insert metric at cursor** drop down, select **cactus_corda_total_tx_count** and click **execute**

Helper code

response.type.ts

This file contains the various responses of the metrics.

data-fetcher.ts

This file contains functions encasing the logic to process the data points

metrics.ts

This file lists all the prometheus metrics and what they are used for.

10.8 @hyperledger/cactus-plugin-ledger-connector-fabric

10.8.1 Table of Contents

10.8.2 1. Usage

This plugin provides a way to interact with Fabric networks. Using this one can perform:

- Deploy smart contracts (chaincode).
- Execute transactions on the ledger.
- Invoke chaincode functions.

The above functionality can either be accessed by importing the plugin directly as a library (embedding) or by hosting it as a REST API through the Cactus API server

We also publish the Cactus API server as a container image to the GitHub Container Registry that you can run easily with a one liner. The API server is also embeddable in your own NodeJS project if you choose to do so.

1.1. Installation

npm

```
npm install @hyperledger/cactus-plugin-ledger-connector-fabric
```

yarn

```
yarn add @hyperledger/cactus-plugin-ledger-connector-fabric
```

1.2. Using as a Library

```
import {
  PluginLedgerConnectorFabric,
  DefaultEventHandlerStrategy,
} from "@hyperledger/cactus-plugin-ledger-connector-fabric";

const plugin = new PluginLedgerConnectorFabric({
  // See test cases for exact details on what parameters are needed
});

const req: RunTransactionRequest = {
  // See tests for specific examples on request properties
};

try {
  const res = await plugin.transact(req);
} catch (ex: Error) {
  // Make sure to handle errors gracefully (which is dependent on your use-case)
  console.error(ex);
  throw ex;
}
```

1.3. Using Via The API Client

Prerequisites

- A running Fabric ledger (network)
- You have a running Cactus API server on `$HOST:$PORT` with the Fabric connector plugin installed on it (and the latter configured to have access to the Fabric ledger from point 1)

```
import {
  PluginLedgerConnectorFabric,
  DefaultApi as FabricApi,
  DefaultEventHandlerStrategy,
} from "@hyperledger/cactus-plugin-ledger-connector-fabric";

// Step zero is to deploy your Fabric ledger and the Cactus API server
const apiUrl = `https://${HOST}:${PORT}`;

const config = new Configuration({ basePath: apiUrl });

const apiClient = new FabricApi(config);

const req: RunTransactionRequest = {
  // See tests for specific examples on request properties
};

try {
  const res = await apiClient.runTransactionV1(req);
} catch (ex: Error) {
  // Make sure to handle errors gracefully (which is dependent on your use-case)
```

(continues on next page)

(continued from previous page)

```
  console.error(ex);
  throw ex;
}
```

1.4. Signing Credentials with Hashicorp Vault

To support signing of message with multiple identity types

```
// vault server config for supporting vault identity provider
const vaultConfig:IVaultConfig = {
  endpoint : "http://localhost:8200",
  transitEngineMountPath: "/transit",
}
// web-socket server config for supporting vault identity provider
const webSocketConfig:IVaultConfig = {
  server: socketServer as FabricSocketServer
}
// provide list of identity signing to be supported
const supportedIdentity:FabricSigningCredentialType[] = [FabricSigningCredentialType.
↪VaultX509,FabricSigningCredentialType.WsX509,FabricSigningCredentialType.X509]
const pluginOptions:IPluginLedgerConnectorFabricOptions = {
  // other options
  vaultConfig : vaultConfig,
  webSocketConfig : webSocketConfig,
  supportedIdentity:supportedIdentity
  // .. other options
}
const connector: PluginLedgerConnectorFabric = new␣
↪PluginLedgerConnectorFabric(pluginOptions);
```

To enroll an identity

```
await connector.enroll(
        {
          keychainId: "keychain-identifier-for storing-certData",
          keychainRef: "cert-data-identifier",

          // require in case of vault
          type: FabricSigningCredentialType.VaultX509, // FabricSigningCredentialType.
↪X509
          vaultTransitKey: {
            token: "vault-token",
            keyName: "vault-key-label",
          },
          // required in case of web-socket server
          type: FabricSigningCredentialType.WsX509,
          webSocketKey: {
            signature: signature,
            sessionId: sessionId,
          },
        },
```

(continues on next page)

(continued from previous page)

```
        {
          enrollmentID: "client2",
          enrollmentSecret: "pw",
          mspId: "Org1MSP",
          caId: "ca.org1.example.com",
        },
      );
```

To Register an identity using register's key

```
const secret = await connector.register(
        {
          keychainId: "keychain-id-that-store-certData-of-registrar",
          keychainRef: "certData-label",

          // require in case of vault
          type: FabricSigningCredentialType.VaultX509, // FabricSigningCredentialType.
↪X509
          vaultTransitKey: {
            token: testToken,
            keyName: registrarKey,
          },
          // required in case of web-socket server
          type: FabricSigningCredentialType.WsX509,
          webSocketKey: {
            signature: signature,
            sessionId: sessionId,
          },
        },
        {
          enrollmentID: "client-enrollmentID",
          enrollmentSecret: "pw",
          affiliation: "org1.department1",
        },
        "ca.org1.example.com", // caID
      );
```

To transact with fabric

```
const resp = await connector.transact{
  signingCredential: {
          keychainId: keychainId,
          keychainRef: "client-certData-id",

          // require in case of vault
          type: FabricSigningCredentialType.VaultX509, // FabricSigningCredentialType.
↪X509
          vaultTransitKey: {
            token: testToken,
            keyName: registrarKey,
          },
          // required in case of web-socket server
```

(continues on next page)

(continued from previous page)

```
          type: FabricSigningCredentialType.WsX509,
          webSocketKey: {
            signature: signature,
            sessionId: sessionId,
          },
        },
        // .. other options
}
```

To Rotate the key

```
await connector.rotateKey(
  {
        keychainId: keychainId,
        keychainRef: "client-certData-id",
        type: FabricSigningCredentialType.VaultX509, // FabricSigningCredentialType.X509

        // require in case of vault
        vaultTransitKey: {
          token: testToken,
          keyName: registrarKey,
        },
        // key rotation currently not available using web-socket server
        // web-socket connection not used to manages external keys
        // user should re-enroll with new pub/priv key pair
  },
  {
      enrollmentID: string;
      enrollmentSecret: string;
      caId: string;
  }
)
```

Extensive documentation and examples in the readthedocs (WIP)

1.4.1. Identity Providers

Identity providers allows client to manage their private more effectively and securely. Cactus Fabric Connector support multiple type of providers. Each provider differ based upon where the private are stored. On High level certificate credential are stored as

```
{
  type: FabricSigningCredentialType;
  credentials: {
    certificate: string;
    // if identity type is IdentityProvidersType.X509
    privateKey?: string;
  };
  mspId: string;
}
```

Currently Cactus Fabric Connector supports following Identity Providers

- X509 : Simple and unsecured provider wherein `private` key is stored along with certificate in some `datastore`. Whenever connector require signature on fabric message, private key is brought from the `datastore` and message signed at the connector.
- Vault-X.509 : Secure provider wherein `private` key is stored with vault's transit transit engine and certificate in `certDatastore`. Rather then bringing the key to the connector, message digest are sent to the vault server which returns the `signature`.
- WS-X.509 : Secure provider wherein `private` key is stored with `client` and certificate in `certDatastore`. To get the fabric messages signed, message digest is sent to the client via `webSocket` connection opened by the client in the beginning (as described above)

1.4.2. Setting up a WS-X.509 provider

The following packages are used to access private keys (via web-socket) stored on a clients external device (e.g., browser, mobile app, or an IoT device...). -ws-identity: web-socket server that issues new ws-session tickets, authenticates incoming connections, and sends signature requests -ws-identity-client: backend connector to send requests from fabric application to ws-identity -ws-wallet: external clients crypto key tool: create new key pair, request session ticket and open web-socket connection with ws-identity

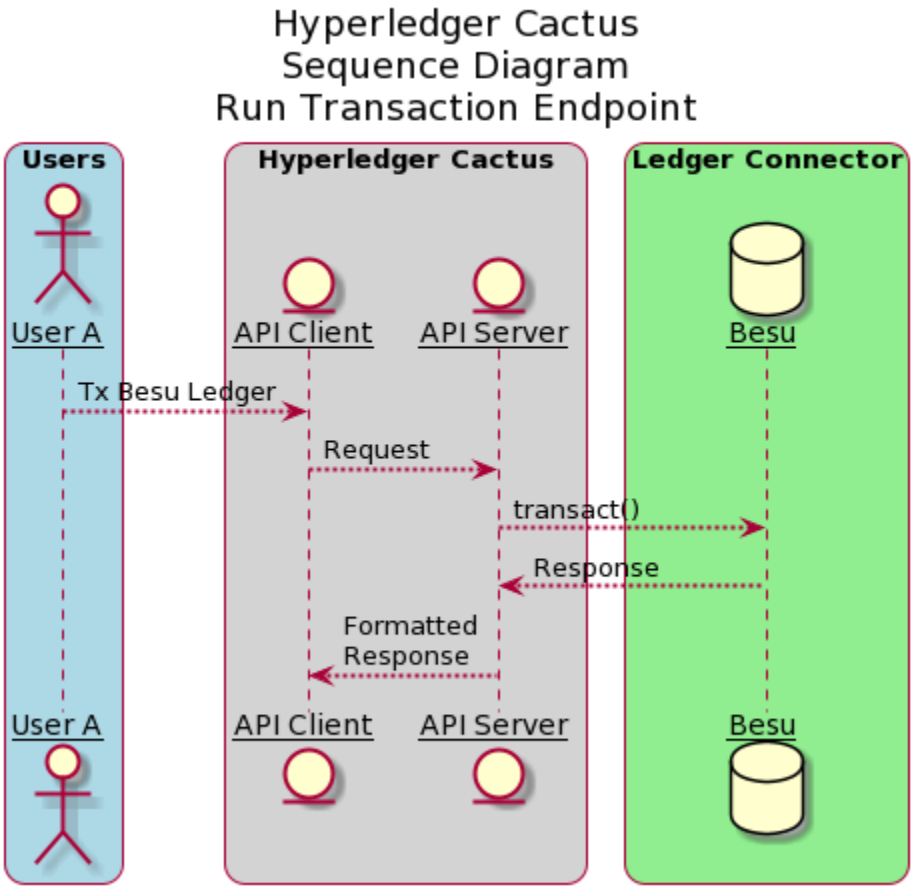
1.4.3. Building the ws-identity docker image

TBD

10.8.3 2. Architecture

The sequence diagrams for various endpoints are mentioned below

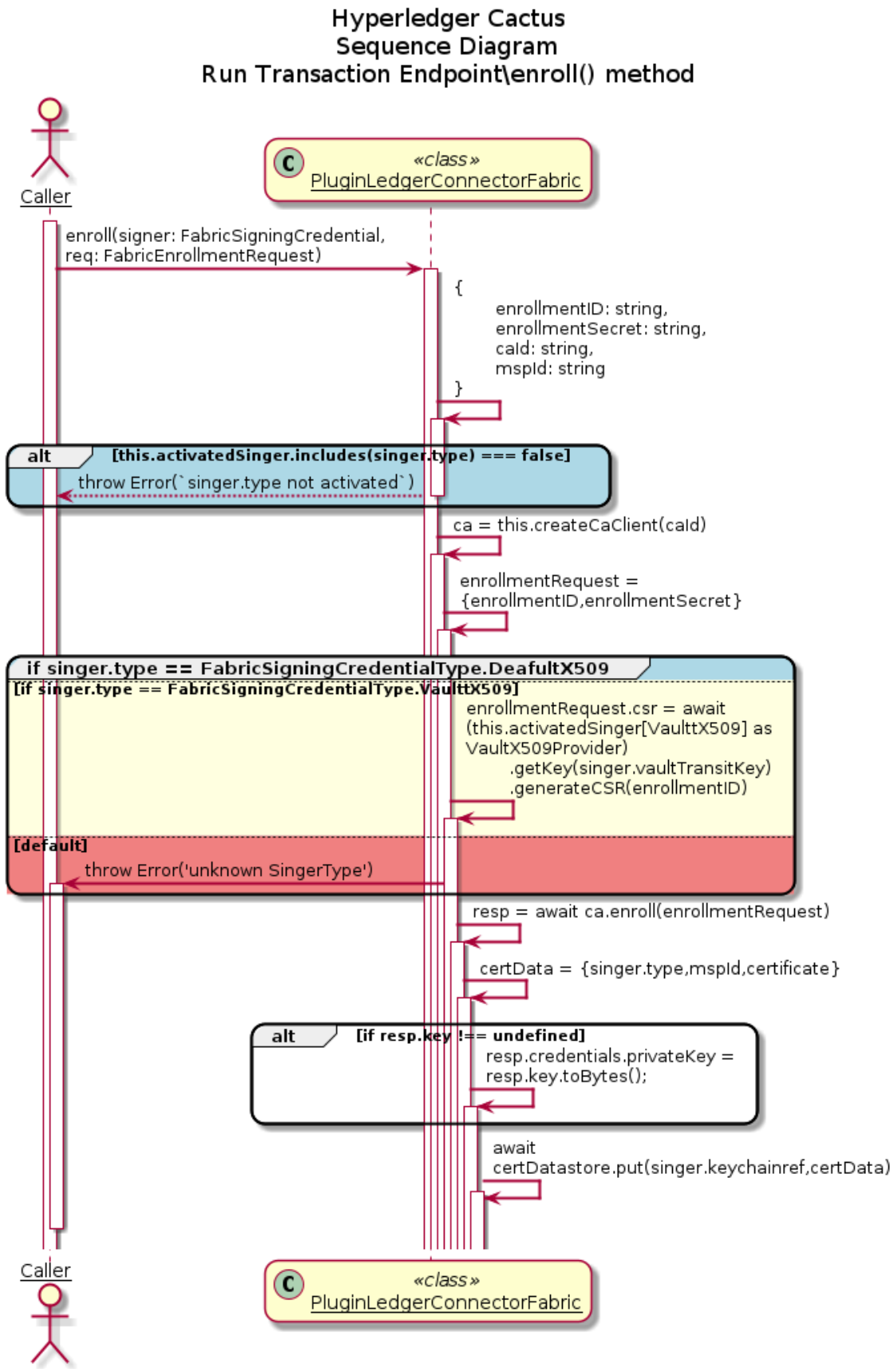
2.1. run-transaction-endpoint



The above diagram shows the sequence diagram of run-transaction-endpoint. User A (One of the many Users) interacts with the API Client which in turn, calls the API server. API server then executes transact() method which is explained in detailed in the subsequent diagram.


Hyperledger Cactus
Sequence Diagram
Run Transaction Endpoint
transact() method
Caller
«class»
PluginLedgerConnectorFabric
transact(req: RunTransactionRequest)
req
{signingCredential, channelName,
contractName, invocationType, fnName as
methodName, params}
InMemoryWallet(new X509WalletMixin())
wallet
opts.pluginRegistry.findOneByKeychainID(signingCredential.keychainId)
keychain
keychain.get<string>(signingCredential.keychainRef)
fabricX509IdentityJson
JSON.parse(fabricX509IdentityJson)
identity
wait for
wallter.import(signingCredential.keychainRef,
identity)
wait for gateway.connect(connectionProfile,
gatewayOptions)
wait gateway.getNetwork(channelName)
network
network.getContract(contractName)
contract
if invocationType == FabricContractInvocationType.CALL
wait for
contract.evaluateTransaction(fnName,
...params)
out, success: true
[if invocationType == FabricContractInvocationType.SEND]
wait for contract.submitTransaction(fnName,
...params)
out, success: true
[default]
throw Error('unknown
FabricContractInvocationType')
out.toString("utf-8")
outUtf8
return { functionOutput, success } as res
Caller
«class»
PluginLedgerConnectorFabric

The above diagram shows the sequence diagraom of transact() method of the PluginLedgerConnectorFabric class. The caller to this function, which in reference to the above sequence diagram is API server, sends RunTransactionRequest object as an argument to the transact() method. Based on the invocationType (FabricContractInvocationType.CALL, FabricCOntractInvocationType.SEND), corresponding responses are send back to the caller.


Hyperledger Cactus
Sequence Diagram
Run Transaction Endpoint\enroll() method
Caller
«class»
PluginLedgerConnectorFabric
enroll(signer: FabricSigningCredential,
req: FabricEnrollmentRequest)
{
enrollmentID: string,
enrollmentSecret: string,
caId: string,
mspId: string
}
alt
[this.activatedSinger.includes(singer.type) === false]
throw Error(`singer.type not activated`)
ca = this.createCaClient(caId)
enrollmentRequest =
{enrollmentID,enrollmentSecret}
if singer.type == FabricSigningCredentialType.DeafultX509
[if singer.type == FabricSigningCredentialType.VaulttX509]
enrollmentRequest.csr = await
(this.activatedSinger[VaulttX509] as
VaultX509Provider)
.getKey(singer.vaultTransitKey)
.generateCSR(enrollmentID)
[default]
throw Error('unknown SingerType')
resp = await ca.enroll(enrollmentRequest)
certData = {singer.type,mspId,certificate}
alt
[if resp.key !== undefined]
resp.credentials.privateKey =
resp.key.toBytes();
await
certDatastore.put(singer.keychainref,certData)
Caller
«class»
PluginLedgerConnectorFabric

The above diagram shows the sequence diagraom of enroll() method of the PluginLedgerConnectorFabric class. The caller to this function, which in reference to the above sequence diagram is API server, sends Signer object along with EnrollmentRequest as an argument to the enroll() method. Based on the singerType (FabricSigningCredentialType.X509, FabricSigningCredentialType.VaultX509, FabricSigningCredentialType.WsX509), corresponding identity is enrolled and stored inside keychain.

10.8.4 3. Containerization

3.1. Building/running the container image locally

In the Cactus project root say:

```
DOCKER_BUILDKIT=1 docker build -f ./packages/cactus-plugin-ledger-connector-fabric/
↪Dockerfile . -t cplcb
```

Build with a specific version of the npm package:

```
DOCKER_BUILDKIT=1 docker build --build-arg NPM_PKG_VERSION=0.4.1 -f ./packages/cactus-
↪plugin-ledger-connector-fabric/Dockerfile . -t cplcb
```

3.2. Running the container

Launch container with plugin configuration as an **environment variable**:

```
docker run \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  --env PLUGINS='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-fabric",
↪"type": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.
↪cactus.plugin_import_action.INSTALL",  "options": {"instanceId": "some-unique-fabric-
↪connector-instance-id", "dockerBinary": "usr/local/bin/docker","cliContainerEnv": {
    "CORE_PEER_LOCALMSPID": "Org1MSP",
    "CORE_PEER_ADDRESS": "peer0.org1.example.com:7051",
    "CORE_PEER_MSPCONFIGPATH":
      "/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.
↪example.com/users/Admin@org1.example.com/msp",
    "CORE_PEER_TLS_ROOTCERT_FILE":
      "/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt",
    "ORDERER_TLS_ROOTCERT_FILE":
      "/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/
↪example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem"
  },
  "discoveryOptions": {
    "enabled": true,
    "asLocalhost: true"
  }
  }}}]' \
  cplcb
```

Launch container with plugin configuration as a **CLI argument**:

```
docker run \
  --rm \
  --publish 3000:3000 \
   --publish 4000:4000 \
  cplcb \
    ./node_modules/.bin/cactusapi \
    --plugins='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-fabric",
↪"type": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.
↪cactus.plugin_import_action.INSTALL",  "options": {"instanceId": "some-unique-fabric-
↪connector-instance-id", "dockerBinary": "usr/local/bin/docker","cliContainerEnv": {
    "CORE_PEER_LOCALMSPID": "Org1MSP",
    "CORE_PEER_ADDRESS": "peer0.org1.example.com:7051",
    "CORE_PEER_MSPCONFIGPATH":
      "/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.
↪example.com/users/Admin@org1.example.com/msp",
    "CORE_PEER_TLS_ROOTCERT_FILE":
      "/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt",
    "ORDERER_TLS_ROOTCERT_FILE":
      "/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/
↪example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem"
  },
  "discoveryOptions": {
    "enabled": true,
    "asLocalhost: true"
  }
  }}}]'
```

Launch container with **configuration file** mounted from host machine:

```
echo '[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-fabric", "type":
↪"org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.cactus.
↪plugin_import_action.INSTALL",  "options": {"instanceId": "some-unique-fabric-
↪connector-instance-id", "dockerBinary": "usr/local/bin/docker","cliContainerEnv": {
    "CORE_PEER_LOCALMSPID": "Org1MSP",
    "CORE_PEER_ADDRESS": "peer0.org1.example.com:7051",
    "CORE_PEER_MSPCONFIGPATH":
      "/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.
↪example.com/users/Admin@org1.example.com/msp",
    "CORE_PEER_TLS_ROOTCERT_FILE":
      "/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt",
    "ORDERER_TLS_ROOTCERT_FILE":
      "/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/
↪example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem"
  },
  "discoveryOptions": {
    "enabled": true,
    "asLocalhost: true"
  }
  }}}]' > cactus.json

docker run \
```

(continues on next page)

(continued from previous page)

```
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  --mount type=bind,source="$(pwd)"/cactus.json,target=/cactus.json \
  cplcb \
    ./node_modules/.bin/cactusapi \
    --config-file=/cactus.json
```

3.3. Testing API calls with the container

Don't have a fabric network on hand to test with? Test or develop against our fabric All-In-One container!

Terminal Window 1 (Ledger)

```
docker run --privileged -p 0.0.0.0:8545:8545/tcp  -p 0.0.0.0:8546:8546/tcp  -p 0.0.0.
↪0:8888:8888/tcp  -p 0.0.0.0:9001:9001/tcp  -p 0.0.0.0:9545:9545/tcp ghcr.io/
↪hyperledger/cactus-fabric-all-in-one:v1.0.0-rc.2
```

Terminal Window 2 (Cactus API Server)

```
docker run \
  --network host \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  --env PLUGINS='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-fabric",
↪"type": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.
↪cactus.plugin_import_action.INSTALL",  "options": {"instanceId": "some-unique-fabric-
↪connector-instance-id", "dockerBinary": "usr/local/bin/docker","cliContainerEnv": {
    "CORE_PEER_LOCALMSPID": "Org1MSP",
    "CORE_PEER_ADDRESS": "peer0.org1.example.com:7051",
    "CORE_PEER_MSPCONFIGPATH":
      "/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.
↪example.com/users/Admin@org1.example.com/msp",
    "CORE_PEER_TLS_ROOTCERT_FILE":
      "/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt",
    "ORDERER_TLS_ROOTCERT_FILE":
      "/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/
↪example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem"
  },
  "discoveryOptions": {
    "enabled": true,
    "asLocalhost: true"
  }
  }}}]' \
  cplcb
```

Terminal Window 3 (curl - replace eth accounts as needed)

```
curl --location --request POST 'http://127.0.0.1:4000/api/v1/plugins/@hyperledger/cactus-
↪plugin-ledger-connector-fabric/run-transaction' \
--header 'Content-Type: application/json' \
--data-raw '{
    channelName: "mychannel",
    contractName: "contract-example";
    invocationType: "FabricContractInvocationType.SEND";
    methodName: "example"
}'
```

The above should produce a response that looks similar to this:


```
{
    "success": true,
    "data": {
        "transactionReceipt": {
            "blockHash":
↪"0x7c97c038a5d3bd84613fe23ed442695276d5d2df97f4e7c4f10ca06765033ffd",
            "blockNumber": 1218,
            "contractAddress": null,
            "cumulativeGasUsed": 21000,
            "from": "0x627306090abab3a6e1400e9345bc60c78a8bef57",
            "gasUsed": 21000,
            "logs": [],
            "logsBloom":
↪"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
            "status": true,
            "to": "0xf17f52151ebef6c7334fad080c5704d77216b732",
            "transactionHash":
↪"0xc7fcb46c735bdc696d500bfc70c72595a2b8c31813929e5c61d9a5aec3376d6f",
            "transactionIndex": 0
        }
    }
}
```


10.8.5 4. Prometheus Exporter

This class creates a prometheus exporter, which scraps the transactions (total transaction count) for the use cases incorporating the use of Fabric connector plugin.

4.1. Usage Prometheus

The prometheus exporter object is initialized in the `PluginLedgerConnectorFabric` class constructor itself, so instantiating the object of the `PluginLedgerConnectorFabric` class, gives access to the exporter object. You can also initialize the prometheus exporter object seperately and then pass it to the `IPluginLedgerConnectorFabricOptions` interface for `PluginLedgerConnectoFabric` constructor.

`getPrometheusExporterMetricsEndpointV1` function returns the prometheus exporter metrics, currently displaying the total transaction count, which currently increments everytime the `transact()` method of the `PluginLedgerConnectorFabric` class is called.

4.2. Prometheus Integration

To use Prometheus with this exporter make sure to install Prometheus main component. Once Prometheus is setup, the corresponding scrape_config needs to be added to the prometheus.yml

```
- job_name: 'fabric_ledger_connector_exporter'
  metrics_path: api/v1/plugins/@hyperledger/cactus-plugin-ledger-connector-fabric/get-
↪prometheus-exporter-metrics
  scrape_interval: 5s
  static_configs:
    - targets: ['{host}:{port}']
```

Here the `host:port` is where the prometheus exporter metrics are exposed. The test cases (For example, packages/cactus-plugin-ledger-connector-fabric/src/test/typescript/integration/fabric-v1-4-x/run-transaction-endpoint-v1.test.ts) exposes it over `0.0.0.0` and a random port(). The random port can be found in the running logs of the test case and looks like (42379 in the below mentioned URL) `Metrics URL: http://0.0.0.0:42379/api/v1/plugins/@hyperledger/cactus-plugin-ledger-connector-fabric/get-prometheus-exporter-metrics`

Once edited, you can start the prometheus service by referencing the above edited prometheus.yml file. On the prometheus graphical interface (defaulted to http://localhost:9090), choose **Graph** from the menu bar, then select the **Console** tab. From the **Insert metric at cursor** drop down, select **cactus_fabric_total_tx_count** and click **execute**

4.3. Helper code

4.3.1. response.type.ts

This file contains the various responses of the metrics.

4.3.2. data-fetcher.ts

This file contains functions encasing the logic to process the data points

4.3.3. metrics.ts

This file lists all the prometheus metrics and what they are used for.

10.8.6 5. Contributing

We welcome contributions to Hyperledger Cactus in many forms, and there's always plenty to do!

Please review CONTIRBUTING.md to get started.

10.8.7 6. License

This distribution is published under the Apache License Version 2.0 found in the LICENSE file.

10.8.8 7. Acknowledgments

10.9 @hyperledger/cactus-plugin-ledger-connector-quorum

This plugin provides `Cactus` a way to interact with Quorum networks. Using this we can perform:

- Deploy Smart-contracts through bytecode.
- Build and sign transactions using different keystores.
- Invoke smart-contract functions that we have deployed on the network.

10.9.1 Summary

- Getting Started
- Usage
- Prometheus Exporter
- Runing the tests
- Contributing
- License
- Acknowledgments

10.9.2 Getting Started

Clone the git repository on your local machine. Follow these instructions that will get you a copy of the project up and running on your local machine for development and testing purposes.

Prerequisites

In the root of the project to install the dependencies execute the command:

```
npm run comfigure
```

Compiling

In the projects root folder, run this command to compile the plugin and create the dist directory:

```
npm run tsc
```

10.9.3 Usage

To use this import public-api and create new **PluginLedgerConnectorQuorum**.

```
  const connector: PluginLedgerConnectorQuorum = new PluginLedgerConnectorQuorum({
    instanceId: uuidV4(),
    rpcApiHttpHost,
    pluginRegistry: new PluginRegistry(),
  });
```

You can make calls through the connector to the plugin API:

```
async invokeContract(req: InvokeContractV1Request):Promise<InvokeContractV1Response>;
async transactSigned(rawTransaction: string): Promise<RunTransactionResponse>;
async transactPrivateKey(req: RunTransactionRequest): Promise<RunTransactionResponse>;
async transactCactusKeychainRef(req: RunTransactionRequest):Promise
↪<RunTransactionResponse>;
async deployContract(req: DeployContractSolidityBytecodeV1Request):Promise
↪<RunTransactionResponse>;
async signTransaction(req: SignTransactionRequest):Promise<Optional
↪<SignTransactionResponse>>;
```

Call example to deploy a contract:

```
const deployOut = await connector.deployContract({
  web3SigningCredential: {
    ethAccount: firstHighNetWorthAccount,
    secret: "",
    type: Web3SigningCredentialType.GETHKEYCHAINPASSWORD,
  },
  bytecode: ContractJson.bytecode,
  gas: 1000000,
});
```

The field "type" can have the following values:

```
enum Web3SigningCredentialType {
    CACTUSKEYCHAINREF = 'CACTUS_KEYCHAIN_REF',
    GETHKEYCHAINPASSWORD = 'GETH_KEYCHAIN_PASSWORD',
    PRIVATEKEYHEX = 'PRIVATE_KEY_HEX',
    NONE = 'NONE'
}
```

Extensive documentation and examples in the readthedocs (WIP)

10.9.4 Running the tests

To check that all has been installed correctly and that the pugin has no errors, there are two options to run the tests:

- Run this command at the project's root:

```
npm run test:plugin-ledger-connector-quorum
```

Building/running the container image locally

In the Cactus project root say:

```
DOCKER_BUILDKIT=1 docker build -f ./packages/cactus-plugin-ledger-connector-quorum/
↪Dockerfile . -t cplcb
```

Build with a specific version of the npm package:

```
DOCKER_BUILDKIT=1 docker build --build-arg NPM_PKG_VERSION=0.4.1 -f ./packages/cactus-
↪plugin-ledger-connector-quorum/Dockerfile . -t cplcb
```

Running the container

Launch container with plugin configuration as an **environment variable**:

```
docker run \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  --env PLUGINS='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-quorum",
↪"type": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.
↪cactus.plugin_import_action.INSTALL",  "options": {"rpcApiHttpHost": "http://
↪localhost:8545", "instanceId": "some-unique-quorum-connector-instance-id"}}]' \
  cplcb
```

Launch container with plugin configuration as a **CLI argument**:

```
docker run \
  --rm \
  --publish 3000:3000 \
   --publish 4000:4000 \
  cplcb \
    ./node_modules/.bin/cactusapi \
    --plugins='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-quorum",
↪"type": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.
↪cactus.plugin_import_action.INSTALL",  "options": {"rpcApiHttpHost": "http://
↪localhost:8545", "instanceId": "some-unique-quorum-connector-instance-id"}}]'
```

Launch container with **configuration file** mounted from host machine:

```
echo '[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-quorum", "type":
↪"org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.cactus.
↪plugin_import_action.INSTALL",  "options": {"rpcApiHttpHost": "http://localhost:8545",
↪"instanceId": "some-unique-quorum-connector-instance-id"}}]' > cactus.json

docker run \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  --mount type=bind,source="$(pwd)"/cactus.json,target=/cactus.json \
  cplcb \
    ./node_modules/.bin/cactusapi \
    --config-file=/cactus.json
```

Testing API calls with the container

Don't have a quorum network on hand to test with? Test or develop against our quorum All-In-One container!

Terminal Window 1 (Ledger)

```
docker run -p 0.0.0.0:8545:8545/tcp  -p 0.0.0.0:8546:8546/tcp  -p 0.0.0.0:8888:8888/tcp ␣
↪-p 0.0.0.0:9001:9001/tcp  -p 0.0.0.0:9545:9545/tcp hyperledger/cactus-quorum-all-in-
↪one:latest
```

Terminal Window 2 (Cactus API Server)

```
docker run \
  --network host \
  --rm \
  --publish 3000:3000 \
  --publish 4000:4000 \
  --env PLUGINS='[{"packageName": "@hyperledger/cactus-plugin-ledger-connector-quorum",
↪"type": "org.hyperledger.cactus.plugin_import_type.LOCAL", "action": "org.hyperledger.
↪cactus.plugin_import_action.INSTALL",  "options": {"rpcApiHttpHost": "http://
↪localhost:8545", "instanceId": "some-unique-quorum-connector-instance-id"}}]' \
  cplcb
```

Terminal Window 3 (curl - replace eth accounts as needed)

```
curl --location --request POST 'http://127.0.0.1:4000/api/v1/plugins/@hyperledger/cactus-
↪plugin-ledger-connector-quorum/run-transaction' \
--header 'Content-Type: application/json' \
--data-raw '{
    "web3SigningCredential": {
      "ethAccount": "627306090abaB3A6e1400e9345bC60c78a8BEf57",
      "secret": "c87509a1c067bbde78beb793e6fa76530b6382a4c0241e5e4a9ec0a0f44dc0d3",
      "type": "PRIVATE_KEY_HEX"
    },
    "consistencyStrategy": {
      "blockConfirmations": 0,
      "receiptType": "NODE_TX_POOL_ACK"
    },
```

(continues on next page)

(continued from previous page)

```
    "transactionConfig": {
      "from": "627306090abaB3A6e1400e9345bC60c78a8BEf57",
      "to": "f17f52151EbEF6C7334FAD080c5704D77216b732",
      "value": 1,
      "gas": 10000000
    }
}'
```

The above should produce a response that looks similar to this:


```
{
    "success": true,
    "data": {
        "transactionReceipt": {
            "blockHash":
↪"0x7c97c038a5d3bd84613fe23ed442695276d5d2df97f4e7c4f10ca06765033ffd",
            "blockNumber": 1218,
            "contractAddress": null,
            "cumulativeGasUsed": 21000,
            "from": "0x627306090abab3a6e1400e9345bc60c78a8bef57",
            "gasUsed": 21000,
            "logs": [],
            "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
            "status": true,
            "to": "0xf17f52151ebef6c7334fad080c5704d77216b732",
            "transactionHash":
↪"0xc7fcb46c735bdc696d500bfc70c72595a2b8c31813929e5c61d9a5aec3376d6f",
            "transactionIndex": 0
        }
    }
}
```


10.9.5 Prometheus Exporter

This class creates a prometheus exporter, which scrapes the transactions (total transaction count) for the use cases incorporating the use of Quorum connector plugin.

Prometheus Exporter Usage

The prometheus exporter object is initialized in the `PluginLedgerConnectorQuorum` class constructor itself, so instantiating the object of the `PluginLedgerConnectorQuorum` class, gives access to the exporter object. You can also initialize the prometheus exporter object seperately and then pass it to the `IPluginLedgerConnectorQuorumOptions` interface for `PluginLedgerConnectoQuorum` constructor.

`getPrometheusMetricsV1` function returns the prometheus exporter metrics, currently displaying the total transaction count, which currently increments everytime the `transact()` method of the `PluginLedgerConnectorQuorum` class is called.

Prometheus Integration

To use Prometheus with this exporter make sure to install Prometheus main component. Once Prometheus is setup, the corresponding scrape_config needs to be added to the prometheus.yml

```
- job_name: 'quorum_ledger_connector_exporter'
  metrics_path: api/v1/plugins/@hyperledger/cactus-plugin-ledger-connector-quorum/get-
↪prometheus-exporter-metrics
  scrape_interval: 5s
  static_configs:
    - targets: ['{host}:{port}']
```

Here the `host:port` is where the prometheus exporter metrics are exposed. The test cases (For example, packages/cactus-plugin-ledger-connector-quorum/src/test/typescript/integration/plugin-ledger-connector-quorum/deploy-contract/deploy-contract-from-json.test.ts) exposes it over `0.0.0.0` and a random port(). The random port can be found in the running logs of the test case and looks like (42379 in the below mentioned URL) `Metrics URL: http://0.0.0.0:42379/api/v1/plugins/@hyperledger/cactus-plugin-ledger-connector-quorum/get-prometheus-exporter-metrics`

Once edited, you can start the prometheus service by referencing the above edited prometheus.yml file. On the prometheus graphical interface (defaulted to http://localhost:9090), choose **Graph** from the menu bar, then select the **Console** tab. From the **Insert metric at cursor** drop down, select **cactus_quorum_total_tx_count** and click **execute**

Helper code

response.type.ts

This file contains the various responses of the metrics.

data-fetcher.ts

This file contains functions encasing the logic to process the data points

metrics.ts

This file lists all the prometheus metrics and what they are used for.

10.9.6 Running the tests

To check that all has been installed correctly and that the pugin has no errors, there are two options to run the tests:

- Run this command at the project's root:

```
npm run test:plugin-ledger-connector-quorum
```

10.9.7 Contributing

We welcome contributions to Hyperledger Cactus in many forms, and there's always plenty to do!

Please review CONTIRBUTING.md to get started.

10.9.8 License

This distribution is published under the Apache License Version 2.0 found in the LICENSE file.

10.9.9 Acknowledgments

10.10 @hyperledger/cactus-test-api-client

This is the test package for the package that's called `cactus-api-client`

10.10.1 Usage

```
// TODO: DEMONSTRATE API
```

10.10.2 FAQ

What is a dedicated test package for?

This is a dedicated test package meaning that it verifies the integration between two packages that are somehow dependent on each other and therefore these tests cannot be added properly in the child package due to circular dependency issues and it would not be fitting to add it in the parent because the child package's tests should not be held by the parent as a matter of principle.

10.11 @hyperledger/cactus-test-cmd-api-server

This is the test package for the package that's called `cactus-cmd-api-server`

10.11.1 Usage

```
// TODO: DEMONSTRATE API
```

10.11.2 FAQ

What is a dedicated test package for?

This is a dedicated test package meaning that it verifies the integration between two packages that are somehow dependent on each other and therefore these tests cannot be added properly in the child package due to circular dependency issues and it would not be fitting to add it in the parent because the child package's tests should not be held by the parent as a matter of principle.

10.12 @hyperledger/cactus-test-plugin-ledger-connector-quorum

10.12.1 Usage

```
// TODO: DEMONSTRATE API
```

10.12.2 FAQ

What is a dedicated test package for?

This is a dedicated test package meaning that it verifies the integration between two packages that are somehow dependent on each other and therefore these tests cannot be added properly in the child package due to circular dependency issues and it would not be fitting to add it in the parent because the child package's tests should not be held by the parent as a matter of principle.

10.13 @hyperledger/cactus-test-tooling

TODO: description

10.13.1 Usage

```
// TODO: DEMONSTRATE API
```

10.13.2 Docker image for the ws-identity server

A docker image of the ws-identity server is used to test integration of WS-X.509 credential type in the fabric connector plugin.

ws-identity includes A Docker file to build the image: clone the repo, install packages, build src and the image

```
npm install
npm run build
docker build . -t [image-name]
```

CHAPTER

ELEVEN

LEDGER SUPPORT FOR CONNECTORS

This section contains the ledger supported versions for connectors in Hyperledger Cactus.

11.1 Besu Support

Note: The deployContract feature is for development and test case authoring only, not recommended to be used in production environments for managing smart contracts.

11.2 Corda Support

Note: The deployContract feature is for development and test case authoring only, not recommended to be used in production environments for managing smart contracts.

11.3 Fabric Support

Note: The deployContract feature is for development and test case authoring only, not recommended to be used in production environments for managing smart contracts.

11.4 Iroha Support

Note: The deployContract feature not yet implemented since Iroha lacks full smart contract support during the initial development stage of the Iroha connector plugin.

11.5 Quorum Support

Note: The deployContract feature is for development and test case authoring only, not recommended to be used in production environments for managing smart contracts.

11.6 xDai Support

Note: The deployContract feature is for development and test case authoring only, not recommended to be used in production environments for managing smart contracts.