@pigi Documentation

Plasma Group

Dec 09, 2019

Plasma Explainers

Hello and welcome to Plasma Group! We're building free and open software to scale blockchains right now. Right now we're mainly focused on building an easy to use general purpose plasma chain.

Just like any big software endeavor, the Plasma Group ecosystem is composed of a lot of moving parts. In an effort to keep this ecosystem maintainable, we're documenting as much as we possibly can. A lot of our documentation is about Plasma Group projects, but there's also a lot about plasma, layer 2, and blockchains in general.

CHAPTER 1

Understanding: Plasma Chains vs. Sidechains

People often talk about plasma chains and sidechains like they're the same thing. Sometimes people even refer to plasma chains as sidechains. However, **plasma chains and sidechains are very different!** It's really important to understand these differences because plasma chains and sidechains make different promises about the safety of your funds.

1.1 Sidechains

1.1.1 The Basics

The idea of the "sidechain" was first popularized by this paper published in 2014. First applied mainly to Bitcoin, the sidechain concept was basically to run another blockchain *alongside* some other "main" blockchain. These two blockchains could then talk to each other in a special way that made is possible for assets to move between the two chains.

Let's take a look at how this might look in the world of Ethereum. If we want to create an Ethereum sidechain, we first have to create another blockchain. We're going to create an Ethereum clone for the sake of this thought experiment.

Setting up an Ethereum clone is really simple. You'd just need to run any standard Ethereum client (like Parity or Geth) and set it up to create a new blockchain instead of connecting to an existing one. You'd also need a "consensus mechanism," which basically just means you need a way to create new blocks. In theory you could use Proof-of-Work, the same system Ethereum uses, but for now let's just assign ourselves the sole power to create blocks (basically "Proof-of-Authority").

Now you'd just need some way for assets to move between the two blockchains. Usually, this is done by creating a smart contract on Ethereum. When users want to move assets from Ethereum onto your sidechain, they deposit those assets into a smart contract sitting on Ethereum. You'd watch for these deposits on Ethereum and re-create those assets on your sidechain. Similarily, when users want to move assets from your sidechain back onto Ethereum, you delete those assets from your sidechain and allow the user to unlock the asset again on Ethereum. It's really as simple as that!

1.1.2 The Pros

If you think about what we just described, there's really no reason why the person who originally deposited some asset also has to be the same person to withdraw the asset. This is what makes sidechains so cool – assets can be moved around a lot before they're withdrawn. Even though we might've made dozens of transactions on the sidechain, only two transactions (the deposit and the withdrawal) ever occur on Ethereum. Since transactions on the sidechain are almost always cheaper than transactions on Ethereum, we get scalability!

1.1.3 The Cons

If you think about the thing we just described, you might see some flaws. Remember that we gave you the sole power to create new blocks. What happens if you stop producing blocks altogether? Or even worse, what happens if you stop allowing anyone to withdraw funds from the sidechain?

It's completely possible for you to do both of these things. Usually this is somewhat mitigated by creating a sidechain with a more robust consensus mechanism. For example, you could copy Ethereum's Proof-of-Work.

Unfortunately this still doesn't fix all of the problems with sidechains. There's a reason why transactions on the sidechain are cheaper than transactions on Ethereum. When you're paying fees on a blockchain, you're paying the miners who keep the blockchain secure. Generally speaking, the more you pay in fees, the more security you get. If the sidechain had just as much hash power as Ethereum (so the same level of security), transactions on the sidechain would cost pretty much the same as transactions on Ethereum.

All of this means that, in general, if a sidechain is cheaper than Ethereum then it's going to be (proportionally) less secure than Ethereum. **If the sidechain fails (meaning the consensus mechanism gets compromised), you could lose all of your funds.** So it's all about the amount of risk you're willing to take. You might feel comfortable putting 1 ETH on a sidechain but not 100 ETH.

1.2 Plasma Chains

1.2.1 The Basics

Plasma chains were popularized by the plasma paper published in 2016. In a nutshell, plasma chains are *sort of* like sidechains, except they trade off some utility for extra security.

Just like sidechains, plasma chains have a consensus mechanism that creates blocks. However, unlike sidechains, the "root" of each plasma chain block is published to Ethereum. Block "roots" are basically little pieces of information that users can use to prove things about the contents of those blocks. For example, a user could use a block root to prove that they made a transaction in that specific block.

Todo: Write an article about how users can use a block root to prove things about the contents of that block.

1.2.2 The Pros

Plasma chain block roots act sort of like "save points" in the blockchain. Remember that one of the major cons of sidechains is that sidechain consensus mechanisms can stop producing blocks and lock everyone's funds up forever. Since it's possible for users to use block roots to show that they received funds on the plasma chain, plasma doesn't have this problem! If the plasma chain consensus mechanism stops creating blocks, users can use the block roots to make *claims* to Ethereum ("I claim I had 10 ETH on the plasma chain and I want to withdraw it.").

Effectively, this means that plasma chains are safer than sidechains by design. Your funds are only ever at risk if *Ethereum* fails, but you probably have bigger problems. Simply stated, a plasma chain is as secure as the main chain consensus mechanism, whereas as sidechain is only as secure as its own consensus mechanism. This convenient property also means that the plasma chain can use really simple consensus mechanisms (like just a single authority!) and still be safe.

Todo: Link out to the operator explainer page.

1.2.3 The Cons

So plasma chains give us cheap transactions that are as secure as the main blockchain. But what's the catch?

Well, when we're using a *sidechain* we have to trust the sidechain consensus mechanism. If that mechanism fails, we're out of luck anyway. That trust makes it possible to do really complex things because we also implicitly trust that the sidechain will be around in the future.

On a *plasma chain*, we keep funds more secure by not making that assumption. We always have to assume that the plasma chain consensus mechanism could fail at any moment and need to design around that. This adds extra restrictions to the things that are possible on a plasma chain.

Take, for example, a very long (let's say 1 year) timelock contract. You could definitely put that contract on a sidechain if you trust that the sidechain will be around in a year. But since we *don't* trust that the plasma chain will be around in year (even if it's the exact same consensus mechanism!), we need to think a little bit outside of the box. We basically need to make sure that if the consensus mechanism fails, we have a way to move the *entire timelock contract* back onto Ethereum. Luckily that's not so difficult, but it's more complex than it would be on the sidechain.

Things get really complex when it's not so clear how the thing on the plasma chain gets to move back onto Ethereum. A timelock contract that's just holding your money makes sense because it seems obvious that *you* should be able to move the contract. But what if we're talking about a timelock contract that's holding money for 100 people at once? Now it's not so clear anymore.

Put simply, the major con of plasma chains is that you can't really do the same complex operations that you could do on sidechains. Importantly, though, the *reason* you can't do these complex things is because you're taking more precautions in order to ensure that your funds stay safe.

CHAPTER 2

Understanding: The Plasma Chain Operator

When learning about plasma you'll eventually run into the idea of the plasma chain 'operator'. For the most part, the operator is exactly what it sounds like — a single entity that's responsible for aggregating transactions into blocks and then publishing those blocks to some other "main" blockchain (like Ethereum). Blockchains are only usable if new blocks are being created, so, by producing these new blocks, the operator is quite literally responsible for keeping the whole plasma chain running.

This might be a little confusing at first. Initial reactions are usually something along the lines of, "What? Plasma chains have operators? Aren't blockchains supposed to be decentralized?" Well, it turns out that the answer is, as you might've guessed, "Kinda."

2.1 Back to Basics

Let's go back to basics and talk a little bit about why most blockchains use decentralized block production mechanisms in the first place. Blockchains are big logs of things that have happened, broken into concrete and ordered time-steps we call blocks. Sometimes these "things that have happened" are simple - "A sent X amount of money to B" - sometimes they're more complex - "Kitty A mated with Kitty B and created Kitty C". No matter what these events are, we usually want to make sure that we have a few key properties:

1. No one should be able to re-write history.
2. No one should be blocked from making transactions.

Let's imagine we have a blockchain run by a single person. For the sake of argument, assume that you *must* use the blockchain run by that person for some reason. Well, unfortunately it's quite easy for that person to break the first property. If that person says "transaction X happened" and then later says "transaction X never happened", there's not much you can really do. *You* know that the transaction happened, but the blockchain itself doesn't.

It's also really easy for that person to break the second property. If they don't want you to send transactions, they can just refuse to add any transactions that come from you. Had a bunch of money on that blockchain? Too bad, you're not getting it back.

2.2 Plasma Magic

The problems with blockchains run by a single person are why blockchains usually have fancy mechanisms that ensure that it's extremely expensive to rewrite history. It's also why blockchains usually have lots of different people who can create blocks – no single person can stop someone from making transactions. So why can we have a single person running plasma chain? It's because we cheat (sort of).

In plasma world, we get the first property by taking advantage of the "main blockchain" that we were talking about earlier. Plasma chain operators need to publish a block "commitment" (sort of like a very compressed version of the block) to the main blockchain for every block they produce. A smart contract on the main blockchain ensures that the operator can never publish the same block twice. As long as the main blockchain has the first property, so does the plasma chain! There's no way for the operator to re-write history unless they can re-write history on the main chain.

The second property (blocking users from transacting) is where things get interesting. Unfortunately, it's still possible for the operator to censor transactions from anyone they want. *However*, this is where the magic of plasma comes in. Plasma chains are designed in a way that **no matter what, a user can always withdraw their money** from the plasma chain back to the main chain. Even if the operator is actively trying to steal money from you, you'll still be able to get it back. Being censored obviously isn't great, but it's not as bad when you can always take your money somewhere else.

2.3 Decentralizing the Operator

The one thing that wasn't really mentioned here is the fact that the "operator" can actually consist of multiple people making decisions about what blocks to publish. This could be as simple as a having a few designed people who take turns making blocks, or as complex as a Proof-of-Stake system that selects block producers randomly. Either way, it's complicated than just having a single person run everything, but it's probably the way to go for projects that want to get rid of censorship. At the same time, it tends to be unimportant research-wise whether the operator is a single person or many people. As a result, you'll often see people just assuming the operator is a single person for simplicity.

CHAPTER 3

Generalized Plasma State Spec

- **Data Structures**
  - `stateID: uint` - the index of unique, non-fungible states in a plasma chain
  - **`stateObject` struct:**
    - `predicate: address` - the state's predicate ruleset
    - `parameters: bytes` - input parameters to the predicate ruleset
  - **`stateUpdate` struct**
    - `state: stateObject` - the new state for this range of `stateID` s
    - `start: uint` - the start of the range of `stateID` s
    - `end: uint` - the end of the range of `stateID` s
    - `plasmaBlockNumber: uint` - the plasma block in which a committment was made
    - `plasmaContract: address` - the address of the plasma contract the state update is meant for
  - **`stateUpdateWitness` struct**
    - `inclusionProof: bytes[]` - array of sibling nodes forming the Merkle inclusion proof
  - **`deposit` struct:**
    - `state: stateObject` - the initial state of the deposited coin
    - `start: uint` - the first `stateID` of the deposit. The plasma contract stores a mapping from `depositEnd->deposit`
    - `precedingPlasmaBlockNumber: uint` - the most recent plasma block leading up to the deposit.
  - **`exitableRange` struct:**

        * `start: uint` - The first `stateID` of a still-exitable range. The plasma contract stores a mapping from `cexitableRangeEnd->exitableRange`
        * `isSet: bool` - whether or not this value in the mapping has been initialized. Needed because EVM can't differentiate between a mapping set to 0 and an unset mapping.
    - **`exit` struct:**
        * `update: stateUpdate` - the state being claimed.
        * `exitStart` - the start of the update's subrange claiming to be undeprecated.
        * `exitEnd` - the end of the update's subrange claiming to be undeprecated.
        * `ethBlockRedeemable: uint` - when the `exit`'s dispute period expires.
        * `numChallenges: uint` - the number of pending challenges preventing a `exit`'s redemption.
    - **`challenge` struct:**
        * `earlierExitID: uint` - the earlier undeprecated `exit` being used to challenge an invlaid `exit`.
        * `laterExitID: uint` - the challenged `exit`.
- **Commitment Contract**
    - **Public Variables:**
        * `operator: public(address)` - the operator's address.
        * `nextPlasmaBlockNumber: public(uint256)` - what the block number of the next block committment will be.
        * `lastPublish: public(uint256)` - the Ethereum block number of the last plasma block.
        * `blockHashes: public(map(uint256, bytes32))` - the blocks submitted so far.
    - **Methods:**
        * `setup` - Used to initiate the contract with collator pubkey
        * `commitBlock`: allows the operator to submit sequential blocks
        * `verifyUpdate(update: stateUpdate, subject: address, stateUpdateWitness: bytes):` returns bool whether a given `commitment` was made on behalf of a given `subject` address (this is the plasma contract for e.g. a specific ERC20), at a the given `plasmaBlockNumber`, based on a valid `commitmentWitness`
    - **Block Structure/Proof Validity, as checked by `verifyUpdate`:**
        * merkle node format: `[hash: bytes32][subject: address][index: bytes16]`
        * merkle parent function: `parent(leftSibling, rightSibling) = [sha256([leftSibling][rightSibling])][rightSibling.subject][rightSibling.index]`
        * merkle proof format: `siblingMerkleNodes[]` - array of the siblings going up the branch
        * branch validity requires that left `siblingMerkleNodes` going up the branch are monotonically decreasin
        * **For a given `commitment`, `subject`, and `commitmentWitness`, where the `leaf` is the bottommost no**

        - `leaf.subject == subject`
        - `stateUpdate.end <= leaf.index`
        - `stateUpdate.subject = leaf.subject`
        - `stateUpdate.start >= START` where `START` is either the merkle proof's deepest left sibling, or `0` if none exist (this is only the case for the ‘‘0‘‘th element in the tree)
    * **leaf nodes are parsed to `[hash(state)][subject][state.end]`**
        - NOTE: for any nodes in the tree whose sibling has the same `subject` address, we may remove the address for efficiency, as long as the above conditions are met as if the `subject` is prepended to the index. This is definitely an optimization to consider down the line!

- **Plasma Contract**
  - **Public Variables:**
    * `self.commitmentAddress` - where the operator is submitting commitments
    * `self.tokenAddress` - the ERC20 contract of for this plasma contract (we'll have one contract per token)
    * `self.deposits[end: uint] -> deposit` - mapping of all deposits to `deposit` structs
    * `self.exitableRanges[end: uint] -> exitableRange` - mapping of all the unclaimed ranges ("states still in the plasma chain")
    * `self.exits[exitID] -> exit` - all of the current exits
    * `self.challenges[challengeID] -> challenge` - all of the current challenges on exits
    * `self.DISPUTE_PERIOD: uint` - the minimum dispute period before a claim can be redeemed
  - **Public methods:**
    * **`deposit(amount, state)`**
        - Deposits specify an initial state and the amount of money being deposited into that state
        - adds to `self.deposits`
        - extends `self.claimableRanges` so that the state is now claimable
    * **`exitStateUpdate(exitStart: uint, exitEnd: uint, update: stateUpdate, u`**
        - `assert verifyUpdate(update, self.address, stateUpdateWitness)`
        - `assert exitStart >= update.start`
        - `assert exitEnd <= update.end`
        - `assert update.state.predicate.can_initiate_exit(update, initiationWitness)`
        - if so, adds a new exit to `self.exits`

- sets the exit's `ethBlockRedeemable` to: `eth.block + self.CHALLENGE_PERIOD + state.predicateAddress.getAdditionalLockup(update)`

* **`exitDeposit(exitStart: uint, exitEnd: uint, depositEnd: uint, claimabi`**

- both of the above store an `exit` struct in `self.exits[self.exitNonce]` and increment `self.exitNonce`.
- sets the claim's `ethBlockRedeemable` to: `eth.block + self.CHALLENGE_PERIOD + state.predicateAddress.getAdditionalLockup(state)`
- In this case, the `update.plasmaBlockNumber` comes from the `deposit.precedingPlasmaBlockNumber`

* **`challengeExit(earlierExitID, laterExitID)` - allows users to challenge a later exit with a**

- **this is the way we challenge exits if the operator commits some a state with something undprecate**

  `earlierExitID`'s claimed range intersects that of `laterExitID`

  `earlierExitID.update.plasmaBlockNumber < laterExitID.update.plasmaBlockNumber`

  `eth.block < laterExit.ethBlockRedeemable`

- **if so, it does the following:**

  create a `challenge` object in `self.challenges[challengeNonce]`

  increment `challengeNonce`

  increase the `laterExit.ethBlockRedeemable` to `earlierExit.ethBlockRedeemable` if the latter is bigger

  increment `challengedClaim.numChallenges`

* **`cancelDeprecatedExit(stateID: uint, exitID: uint, deprecationWitness: by`**

  - `exit = self.exits[exitID]`
  - `assert exit.update.predicateAddress.verifyDeprecation(stateID, exit.update, deprecationWitness)`
  - if so, clears the exit, deleting it from the `self.exits` mapping

* **`removeChallenge(challengeID: uint)` - allows users to remove a challenge**

  - checks that the `self.challenges[challengeID].earlierExit` has been revoked, i.e. that its key is no longer set to a value in self.exits[]
  - if so, decrements the `self.exits[self.challenges[challengeID].laterExitID].numChallenges` and then clears/deletes `self.challenges[challengeID]`

* **`finalizeExit(exitID, exitableRangeEnds)`**

  - asserts `exit`'s numChallenges = 0

        - tries `isRangeClaimable` for the various `claimableRangeEnds`, reverts if none pass the check
        - asserts the current `eth.block >= exit.ethBlockRedeemable`
        - approves the ERC20 claim amount (`=start-end`) to be transferred by the `exit.update.state.predicateAddress`
        - calls `finalizeExit(update)` on the `update.state.predicateAddress`

- **Predicate interface**
  - **Public methods/interface:**
    * `verifyDeprecation(stateID: uint, update: stateUpdate, deprecationWitness: bytes) -> bool` - returns true/false whether a given `deprecationWitness` is valid (if true the exit may be cancelled)
    * **`finalizeExit(update: stateUpdate)` - called once a claim on a state is redeemed on the**
        - in principle, this can do anything, but will almost always call the `ERC20.transferFrom` function to the tune of `exit.start - exit.end`, either to itself to initiate an additional dispute period, or to some ultimate beneficiary as devised from the `exit.update.state.parameters`
    * `canInitiateExit(update: stateUpdate, initiationWitness: bytes) -> bool` - returns true/false whether a claimant is eligible to submit an exit on a given state
    * `getAdditionalDisputePeriod(update: stateUpdate)` - returns an additional number of ETH blocks which must elapse, in addition to the standard `plasmaContract.DISPUTE_PERIOD`, before the exit may be redeemed
- **Predicate Examples**
  - **Simple Ownership**
    * **`struct ownershipDeprecationWitness:`**
        - `newStateUpdate: stateUpdate`
        - `newUpdateWitness: stateUpdateWitness`
        - `signature: signature`
    * **`public function verifyDeprecation(stateID: uint, update: stateUpdate,`**
      ```
      assert verifyUpdate(deprecationWitness.
      newStateUpdate, revocationWitness.newUpdateWitness)
      assert verifySignature(revocationWitness.
      newStateUpdate, signature) = update.state.owner
      ```
    * **`public function finalizeExit(exit: exit):`**
      ```
      redeemedAmount: uint = exit.end - exit.start
      #length of sequential stateIDs claimed          ERC20.
      transferFrom(self.address, exit.update.state.owner,
      )
      ```
    * **`public function canInitiateExit(update: stateUpdate, initiationWitnes`**
      assert tx.sender = commitment.state.parameters.owner‘‘
  - Multisig

- Atomic Swap
- **Basic Payment Channel**
    - **struct `stateChannelParameters`:**
        - `participants: address[]` - array of pubkeys participating in the channel
        - `openingUpdatesHash: bytes32` - a hash of all the `stateUpdate` objects which must be made for the channel to be considered successfully "opened"
        - `failedOpeningRecipient: address` - the person to send money to if the opening failed, i.e. the above commitments weren't made
        - `onChainChannel: address` - the on-chain payment channel to send the money to if channel isn't closed out on-chain
        - `callData: bytes[]` - the instantiation data passed to the `onChainChannel`
    - **struct `stateChannelDeprecationWitness`**
        - `closureUpdates: stateUpdate[]` - array of the states agreed to close on
        - `closureUpdateWitnesses: stateUpdateWitness[]` - array of the proofs that the above updates were made
        - `closureApprovals: signature[]` - array of signatures by each of the `state.parameters.participants` on `hash(closureUpdates)` agreeing to close
    - public `self.successfulOpenings[upeningUpdatesHash] -> bool` - mapping of whether or not a given `openingUpdatesHash` was successfully made
    - **public `proveOpenings(openingUpdates: commitment[], openingWitnesses: state`**
        - allows users to prove that a state channel was successfully opened by validating all opening inclusions
        - asserts that `verifyUpdate` for each `openingUpdate` state and its witness
        - if so, sets ‘‘self.successfulOpenings[hash(openingUpdates) = true]
    - **struct `openingExitStatus` - the struct used if an open channel is being exited because of an unsucces**
        - `totalCoins` - the total number of coins entered into the payment channel
        - `redeemedCoins` - the total number of coins whose claims have been redeemed so far
    - public `self.openingExitsInProgress[upeningUpdatesHash:bytes32] -> openingExitStatus` - mapping of "in progress" exits on opened channels
    - **`verifyDeprecation`**
        - asserts that `self.openingClaimsInProgress[update.parameters.openingUpdatesHash)].redeemedCoins == 0` – if any of the opening state has been redeemed, all state must be redeemed from the openings, no revocation is valid.

- asserts that `verifyUpdate` for each commitment in the revocation witness
- asserts that each `state.parameters.participants` signed off on `hash(closureUpdates)`

* **`finalizeExit`**
  - checks whether the channel was successfully opened: `assert self.successfulOpenings[openingUpdatesHash]`
  - **If it was:**

    `self.openingExitsInProgress[openingUpdatesHash].redeemedCoins += exit.end - exit.start`

    let `exitInProgress = self.openingExitsInProgress[openingCommitments`

    if `exitInProgress.redeemedCoins == exitInProgress.totalCoins`, then forward the `totalCoins` to the `exit.update.parameters.onChainChannel(exit.update.parameters.callData)` – the opening has been fully claimed and the on-chain channel may take over.
  - Otherwise, not all money in the channel has been redeemed from the plasma contract yet, so we must wait.

– **L1<>L2 liquidity predicate (swap PETH for ETH)**

* **struct `tradeParameters`:**
  - `tradeID: uint` - a unique ID for the trade
  - `seller: address`
  - `saleAmount: uint` - the amount of ETH the coins are being sold for
* **struct `trade`**
  - `ethSender: address`
  - `targetPlasmaBlock: uint`
* mapping `self.trades[tradeID][ethRecipient][amount] -> trade` maps the unique aspects of the trade to the sender and intended block of the new ownership state committment
* **public method: `submitTrade(tradeID: bytes32, ethRecipient: address, targ`**
  - assert that the next plasma block is the `targetPlasmaBlock`
  - assert that `self.trades[tradeID: bytes32][ethRecipient: address][tx.value: uint]` is unset
  - **if not:**

    set the value with `trade.ethSender = tx.Sender` and `trade.targetPlasmaBlock = targetPlasmaBlock`

    forward the ETH to `ethRecipient`
* **`verifyDeprecation`**
  - **`deprecationWitness` consists of:**

    a valid `newStateUpdate`, satisfying:

`.start` and `.end` equalling the deprecated `stateUpdate.start` and `.end`

the existance of an entry in `self.trades[stateUpdate.parameters.tr`

the `ethSender` in that entry being the `newStateUpdate.parameters.owner`

the `newStateUpdate.plasmaBlockNumber == trade.targetPlasmaBlock`

* **`finalizeExit`**
  - **checks for the existence of an entry in `self.Trades[exit.state.parameters.`**

    if it exists, send to that `trade.ethSender`

    otherwise, send back to `redeemedState.parameters.seller`

CHAPTER 4

Contributing to @pigi

Welcome! A huge thank you for your interest in contributing to Plasma Group. Plasma Group is an open source initiative developing a simple and well designed plasma implementation. If you're looking to contribute to a Plasma Group project, you're in the right place! It's contributors like you that make open source projects work, we really couldn't do it without you.

We don't just need people who can contribute code. We need people who can run this code for themselves and break it. We need people who can report bugs, request new features, and leave helpful comments. **We need you!**

We're always available to answer your questions and to help you become a contributor! You can reach out to any of the members of Plasma Group on GitHub, or send us an email at contributing@plasma.group.

Here at Plasma Group we're trying to foster an inclusive, welcoming, and accessible open source ecosystem. The best open source projects are those that make contributing an easy and rewarding experience. We're trying to follow those best practices by maintaining a series of resources for contributors to Plasma Group repositories.

If you're a new contributor to `@pigi/core`, please read through the following information. These resources will help you get started and will help you better understand what we're building.

4.1 Contributing Guide and Code of Conduct

Plasma Group follows a Contributing Guide and Code of Conduct adapted slightly from the Contributor Covenant. **All contributors are expected to read through this guide.** We're here to cultivate a welcoming and inclusive contributing environment. Every new contributor needs to do their part to uphold our community standards.

4.2 Getting Started as a Contributor

4.2.1 Requirements and Setup

Cloning the Repo

Before you start working on a Plasma Group project, you'll need to clone our GitHub repository:

```
git clone git@github.com:plasma-group/pigi.git
```

Now, enter the repository.

```
cd pigi
```

Node.js

Most of the Plasma Group projects are Node.js applications. You'll need to install `Node.js` for your system before continuing. We've provided a detailed explanation of now to install Node.js on Windows, Mac, and Linux.

Yarn

We're using a package manager called Yarn. You'll need to install Yarn before continuing.

Installing Dependencies

`@pigi` projects make use of several external packages.

Install all required packages with:

```
yarn install
```

4.3 Building

`@pigi` provides convenient tooling for building a package or set of packages.

Build all packages:

```
yarn run build
```

Build a specific package or set of packages:

```
PKGS=your,packages,here yarn run build
```

4.4 Linting

Clean code is the best code, so we've provided tools to automatically lint your projects.

Lint all packages:

```
yarn run lint
```

Lint a specific package or set of packages:

```
PKGS=your,packages,here yarn run lint
```

We've also provided tools to make it possible to automatically fix any linting issues. It's much easier than trying to fix issues manually.

Fix all packages:

```
yarn run fix
```

Fix a specific package or set of packages:

```
PKGS=your,packages,here yarn run fix
```

4.5 Running Tests

`@pigi` projects usually makes use of a combination of Mocha (a testing framework) and Chai (an assertion library) for testing.

Run all tests:

```
yarn test
```

Run tests for a specific package or set of packages:

```
PKGS=your,packages,here yarn test
```

Contributors: remember to run tests before submitting a pull request! Code with passing tests makes life easier for everyone and means your contribution can get pulled into this project faster.

CHAPTER 5

Miscellaneous Reference

This page provides a series of miscellaneous reference articles that can be helpful when installing Plasma Group components.

5.1 Running a Terminal

Before you keep going, it's probably good to become familiar with using the terminal on your computer. Here are some resources for getting started:

- Windows: Command Prompt: What It Is and How to Use It
- MacOS: Introduction to the Mac OS X Command Line
- Linux: How to Start Using the Linux Terminal

5.2 Installing Git

`git` is an open source version control system. You don't really need to know how it works, but you *will* need it in order to install most Plasma Group components.

5.2.1 Windows

Atlassian has a good tutorial on installing `git` on Windows. It's basically just installing an `.exe` and running a setup wizard.

5.2.2 MacOS

Installing `git` on a Mac is pretty easy. You basically just need to type `git` into your terminal. If you have `git` installed, you'll see a bunch of output. Otherwise, you'll get a pop-up asking you to install some command-line tools (including `git`).

5.2.3 Linux

Installing `git` on Linux is also pretty easy. However, the exact install process depends on your distribution. Here's a guide for installing `git` on some popular distributions.

5.3 Installing Node.js

Most of the Plasma Group apps are built in JavaScript and make use of a tool called Node.js. In order to run our tools, you'll need to make sure that you've got `Node.js` installed.

Here's a list of ways to install `Node.js` on different operating systems:

5.3.1 Windows

If you're on a windows computer, you can download the latest Long-term Support (LTS) version of `Node.js` here. You'll just need to install the `.msi` file that `Node.js` provides and restart your computer.

5.3.2 MacOS

You have some options if you want to install `Node.js` on a Mac. The simplest way is to download the `.pkg` file from the `Node.js` downloads page. Once you've installed the `.pkg` file, run this command on your terminal to make sure everything is working properly:

```
node -v
```

If everything is working, you should see a version number pop up that looks something like this:

```
v10.15.1
```

Homebrew

Note: If you've already installed `Node.js` with the above steps, you can skip this section!

You can also install `Node.js` using Homebrew. First, make sure Homebrew is up to date:

```
brew update
```

Now just install `Node.js`:

```
brew install node
```

5.3.3 Linux

There are different ways to install `Node.js` depending on your Linux distribution. Here's an article that goes through installing `Node.js` on different distributions.

CHAPTER 6

Our Lovely Contributors

This is just a quick high-level overview of the different types of contributors we can expect to work on Plasma Group projects! All of our contributors should be treated with respect, and they're all equally important. It's important to understand the different types of people who might be helping out with Plasma Group projects so we can streamline the contributing experience (and make sure everyone feels the love they deserve!).

6.1 New Contributors

New Contributors are people who want to get involved with PG but might not know how yet. This is the "bucket" category for the remaining personas and New Contributors are eventually funnelled into one of the other categories. These users can be pretty much anyone - software engineers, crypto enthusiasts, family members, whoever.

Generally, we want to help new contributors become active contributors. For example, we might want to direct someone to resources that help them become a Beta Testing Contributor. This means that we should develop a set of strong resources where a new contributor can land and figure out where they want to help out.

6.2 Co-Developing Contributors

Co-Developing Contributors are people working on the PG code in any one of our repositories. These people are doing all sorts of work, like tackling issues, adding new features, or writing new documentation. This work is usually happening in the Plasma Group repositories.

These contributors want things to work on, and want a good experience while doing so. You wouldn't be working on a project if you couldn't figure out what to do or if you got a terrible response every time you tried to contribute!

Co-Developers are extremely important and we need to make sure that they're always treated with respect. We should be responding as quickly as possible, reviewing their PRs, and rewarding them for their hard work.

6.3 Beta Testing Contributors

Beta Testing Contributors are users helping us test our code, but not necessarily writing code. Often these contributors don't have the time to contribute lots of code because they're busy with other projects. However, these users still want to give back to the community by seeing how things work and playing with new features. These users want to make sure that their feedback is heard and that their time spent beta testing isn't for nothing. Adding features or fixing bugs quickly will make beta testers feel like there's constantly something new to explore.

Beta Testing Contributors should be treated as a great resource, because they are. Without them, we wouldn't find all of the various bugs we're inevitably going to have. There's no such thing as production-quality software without beta testing :-).

6.4 Passive Contributors

Passive Contributors are users who are interacting with PG but aren't actively working on issues in one of our GitHub repositories. These users are typically people building applications on top of our projects but not working on the underlying code. For example, this might be someone building a wallet using `@pigi/plasma-js`.

Most of the time, these users don't come in intending to make significant changes to the code that they're making use of - they usually just expect it to work. When they do interact with the codebase, it tends to be to report a bug or ask for a new feature. These users are important because they find bugs in production and request features that help us build things out for real use-cases.

6.5 External Contributors

There are lots of people who support us but aren't specifically helping out by making issues, tackling issues, or requesting features. This could be people on twitter discussing our work, sitting near us when we're working, or just providing moral support. We have to care for these people because they're what keep us motivated and pushing to build the best projects we can! Let them know they're appreciated because we wouldn't be here without them.

CHAPTER 7

Community Interactions

Community members are the single most important resource available to Plasma Group. With such a small set of core contributors, we can only hope to create the best possible plasma implementation by tapping into our community. Remember, we **cannot succeed without the support of our community**.

In the words of Eric Raymond:

> If you treat your beta-testers as if they're your most valuable resource, they will respond by becoming your most valuable resource.

It's **extremely** important that we keep this fact in mind during **each and every** interaction with a community member, on- or offline.

7.1 General Guidelines

1. **Treat all contributors with respect.**

Our contributors are taking time out of their busy days to help us make plasma a reality. That alone deserves a lot of respect! Our contributors are also real people with real lives. Plasma Group core contributors should treat all outside contributors with the same respect they show each other.

2. **Always thank contributors.**

Whether it's help with a bug report, feature request, or pull request, our contributors make a huge dent in the amount of work placed on the core team. Thank contributors for their help! It's nice and it's the right thing to do, but it's also important to make contributors feel appreciated. Most of the people helping us aren't doing it for the money, and they definitely don't want to receive a harsh response for trying to help.

Even if a user has submitted something like a duplicate issue, thank them for their assistance! They'll feel like they can help out without judgement.

3. **Show contributors that their contributions are making a difference.**

Contributors want to make a difference, and it's important that their impact is acknowledged. Again, always thank contributors for their help. Feel free to reach out to contributors privately and thank them for their contributions, especially after big releases.

Another great way to thank contributors for their work is to send GitCoin Kudos. Kudos are special ERC-721s that you can send to users via their GitHub usernames!

4. **Help contributors get involved.**

This might be the most important rule of them all. Always be focused on helping contributors get involved. Whether that's by Tweeting out GitHub issues, linking users to documentation, or writing better issues, your **number one focus** should be to build out the community of contributors.

CHAPTER 8

GitHub Interactions

Most of our interactions with community members happen through GitHub. As a result, it's important that the key GitHub interactions are well planned out. This section describes the most important interactions (Bug Reports, Feature Requests, and Pull Requests) and provides basic guidelines for how to best work with community members.

Remember that everyone who contributes to our projects is taking time out of their day. Always thank people for helping out, even if you're closing out an issue for being a duplicate of something else! The golden rule is the best rule :-).

8.1 Bug Reports

1. **If a potential active contributor submits an issue, give them the resources to become an active contributor.**

A lot of the time, users submitting an issue aren't familiar with the underlying code. They're probably submitting an issue because they can't find an easy fix! If you just tell the user that you're "on it" or something similar, you're likely making the situation worse.

First, thank the contributor for reporting the issue in a timely manner. They're taking time out of their day to report a problem and, if they're like a lot of us, it's probably something that's caused them a headache. A quick acknowledgment will let them know that you're available, responsive, and are taking their issue seriously.

If the user hasn't provided enough information, make sure to politely ask for information that you think might be relevant. Always make sure the user provides steps to reproduce the problem so that it can be more easily solved. Unless you know otherwise, never assume that the user is particularly technical. Help them out by giving them the exact commands and relevant output that'll help solve the issue.

Todo: Create a basic diagnostics guide that helps users figure out what's going on with their projects.

---

2. **Get to the source of the issue.**

Next is getting to the source of the issue. This is where things get fun! The first thing you'll want to do is to assess the general difficulty of the underlying problem. Using the steps to reproduce, try to locate the problematic areas of the

software. Is it an encoding issue? Is the wrong thing being passed to another function? Is something undefined when it shouldn't be? Is it a problem that'll require fixes in ten different places?

Depending on the difficulty of the problem, you'll want to take different next steps. If the problem is relatively simple, great! This is an awesome opportunity to convert the potential contributor into an active one. If a problem is simple, unless the it's absolutely critical and needs an immediate fix, your top priority should be to give someone else the tools to solve the problem. Remember, letting potential contributors get their hands into some code is the best way to convert them to active contributors.

3. **Try to give people the tools to solve problems themselves.**

Giving someone the tools to solve the issue for themselves is a multi-step process. First, you'll want to provide the user with all the necessary background information. Explain the different relevant components and then explain what you think is probably the general cause of the issue. Feel free to tag another contributor if you think they'd have a better understanding of the problem. Your next steps depend on whether you know exactly what's causing the issue.

If you don't know exactly what's causing the issue, this is a good opportunity to ask a potential contributor if they'd like to step in and solve the issue! This might be the person who submitted the issue or it might be another contributor that you tag. If it's not the person who submitted the issue, try to think about which contributor would most benefit from working on the issue.

If you already know what's causing the issue, try to explain the cause in a very detailed manner. You can even go as far as pointing out specific lines that are causing problems or sketching out a potential solution. You don't necessarily want to entirely solve the problem for someone else, but you do want to give enough leads if possible.

8.2 Feature Requests

1. **Clarify the exact parameters of the feature request.**

Sometimes contributors are already very familiar with our codebase/functionality and have a clear idea of what new features they'd like. Other times contributors have a general idea of what they'd like, but we still need to figure out exactly how the feature will work. It's important to start off by clarifying exactly how the new feature will work. Clear feature requests are the best way to make sure that we're implementing exactly what was requested. We definitely don't want to spend a lot of time building something out that isn't useful!

2. **Figure out what changes would need to be made for the new feature.**

Understanding the scope and impact of any changes is necessary for figuring out a timeline. Once you've clarified exactly what feature is being requested, you can start figuring out what needs to be changed in order to add the feature. If you're familiar with the codebase, this is a great time to point to the files that would need to be changed. Otherwise, feel free to tag any other contributors who might have a better idea about the problem!

3. **Figure out a timeline for the new feature.**

We want to make sure that contributors who request new features get a timeline so they know how long they'll need to wait. Contributors might otherwise turn to another project with a more explicit roadmap. Try to guesstimate the amount of time that a feature will take to complete. Also think about where the feature fits in with other features because it might make more sense to release several updates simultaneously. Definitely discuss with other contributors if you're not sure on an exact timeline.

4. **Keep people updated with the status of a new feature.**

Figuring out an exact timeline is more of an art than a science. There are always unexpected things that might speed up (or slow down!) the addition of a new feature. It's important that we keep a feature request thread updated with the latest work on a feature.

8.3 Duplicate Requests

1. **Make sure the request is actually a duplicate.**

Sometimes we get duplicate bug reports or feature requests. This means that someone has created an issue that's already been created before. Duplicates are easy to handle well but they're also easy to handle poorly.

The very first thing you should do is make sure that the issue is actually a duplicate! Sometimes people submit very similar issues that have subtle differences. These subtle differences can actually have a huge impact on the required fixes.

2. **Be nice about it!**

Contributors who submit duplicates still did the same amount of work as the contributor who submitted the original issue. It's disappointing realizing that you did duplicate work, so make sure that they understand we still appreciate their work immensely. Maintainers too often treat duplicates like throwaway issues and close them without much interaction. Thank contributors for their work and direct them to the new thread so they can continue to help out.

3. **Close the duplicate issue and label it.**

This is the last step! Make sure to close the duplicate (but not before being nice) and label it as a duplicate for the future.