Djwebdapp V1.0.0

Thomas Binetruy

2023-07-31

Abstract

In this paper, DjWebDapp is introduced, a set of Django applications to simplify multi blockchain smart contract deployment orchestration, indexers and end-to-end testing. It describes the queue based mechanism to deploy smart contracts, the indexing API put in place allowing developpers to normalize smart contract data in a database, the provider architecture put in place in order to handle multiple blockchains and the reasoning behind these choices. Finally, various examples of applications built on Djwebdapp are shown.

1 Introduction

Developing applications making use of the blockchain can be very time consuming as it consists in writing information to and reading information from the blockcain.

Indeed, for many applications, it is impossible to retrieve data to be shown on a frontend directly. As an example, historical prices on a decentralized exchange are not stored on-chain, and as such, require a service that will index each swaps, store them in a database, and expose them to a frontend via an API. Similarly, in many NFT applications, frontends would like to show the assets owned by a particular wallet. Although this information is stored on-chain, it is subperformant to request it from the smart contract directly since for storage effeciency reasons at the smart contract level, this data is not easily made available to the user and requesting this information would require iterating over all tokens to retrieve their owners. Therefore, applications develop services that index mint and transfer calls to re-create the ledger off chain and expose it via an API. In both of these examples, an NFT platform and a defi application, frontends usually allow to search and filter through assets, which is usually not possible with on-chain calls as sometimes, the metadata to filter against is not on the blockchain but stored elsewhere.

Some applications also require some automated way to inject transactions into the blockchain, which can fail due to network events (the node is not reachable) and on-chain events such as reorgs. In both these cases, a system in charge of writing transactions on chain requires monitoring these transactions in order to ensure that they were indeed injected into the blockchain. Such systems include claimable reward systems on blockchain games, trading or liquidation bots, cross-chain bridges and oracles providing offchain data to on-chain smart contracts.

Finally, smart contract based blockchains work in a similar fashion, but have all specific SDKs that one has to learn in order to manipulate, making for heterogenous development from across blockchains.

The present paper presents DjWebDapp, an open-source Django [8] application developped at Pyratzlabs, a Paris-based Web3 startup studio, and used by some of our startups that allow to address each of the above challenges summarized as follows:

1. provide a uniform API across blockchains;
2. provide abstractions to index smart contracts and normalize their data in a reorg-robust fashion;
3. provide a queue based transaction injection API with a retry mechanism to simplify automating the deployment of smart contracts and their interactions.

The goal for DjWebDapp is therefore to provide a unified framework allowing to simplify the development of blockchain-based applications, hence the name:

1. "Dj": stands for Django, the underlying Python framework DjWebDapp is built on;
2. "Web": stands for Webapp, applictions build on the World Wide Web;
3. "Dapp": stands for Decentralized-Application (DApp), applications build on a Blockchain.

Indeed, DjWebDapp aims at builds on top of Django and enhancing its already wide set of features in order to make it Web3 ready.

2 The Django Web Framework

Bulding web application backends can be very time consuming. In order to ease their development, many frameworks were built in order to encourage rapid development. These provide the necessary abstractions to simplify common routine tasks such as implementing authentication and APIs, interacting with the database using an ORM with a migration system, or adding a caching system.

Different languages have their own frameworks, such as Laravel for PHP, Ruby on Rails for Ruby, DotNet for C#, Spring for Java to name a few. The Python most succesful equivalent is Django (used at Instgram, Pinterest, Udemi, Robinhood, Opensea, Backmarket).

Not only is Django widely used and battle-tested at mega-corps in the Tech sector and thus highly maintained, it is Python based meaning that it benefits from its entire ecosystem, large standard library and is easy to learn. For this reason, it was preferred to its equivalents in other languages.

Therefore, DjWebDapp aims at building on Python and Django's ecosystems and providing a unified API for blockchain web application development.

3 High-level Architecture

A typical Django project is structured with a server serving the Django application and exposing its route on ports 80 (HTTP) and 443 (HTTPS). Django stores its persistent data in a database, and, as it has various backend, many database engines can be used and abstracted away by a uniform API for storing, mutating and migrating the database. At Pyratzlabs, we favor PostgreSQL [15] to be used on production systems as it is Free and Open Source Software (FOSS), scales well, and is well supported by Django.

On top of these two services, DjWebDapp implements three new servicese, implemented as Django management commands:

1. the indexer: which queries for new blocks continually from a blockchain node and stores the raw transactions targetting contracts to index: its sender, arguments, events and method name;

2. the normalizer: which normalizes indexed transactions into database tables more suitable for querying;

3. the spooler: which deploys queued transactions in an asynchronous manner.

Since all these services are implemented as Django management commands, they communicate with each other and the core Django application served over the internet through the database using Django's ORM. Therefore, the DjWebDapp services are Django services started differently, they do not serve any endpoints. However, it becomes trivial to serve normalized smart contract data and inject transactions on chain via an API.

4 Blockchain Providers

One of the main goal of DjWebDapp is to be blockchain agnostic. As such, it needs to provide its own abstractions over SDKs used to interact with various blockchain to provide developpers with a unified API to interact with the different chains.

The Strategy pattern, which DjWebDapp implements, is used to define a common interface to all supported algorithms [6]. An abstract `Provider` class is used to define a set of necessary routines to index and deploy on-chain transactions, which are to

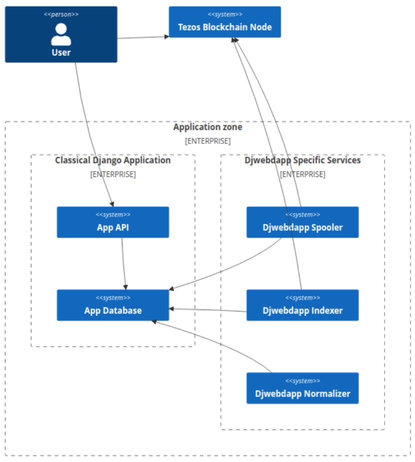

Figure 1: Typical Djwebdapp Services Architecture

be subclassed for each supported blockchain by wrapping Python SDKs for on-chain interactions. In particular, Pytezos [5] is used in the `TezosProvider` and Web3.py [9] is used for EVM compatible chains in `EthereumProvider`. Since blockchains have their own specific way of representing their blocks and transactions, the `Provider` class defined methods to be implemented by subclasses which will be used by common method implementations using these.

Moreover, multiple providers can easily be implemented for the same blockchain. It is for example possible that one would not want to index or deploy transactions by querying the blocktchain directly, but rather, by going through an alternate service such as Alchemy, Moralis and Venly for Ethereum or Tzkt for Tezos. In such a scenario, one would subclass `Provider` and implement their own custom logic to accustom their needs.

As a matter of fact, and as will be presented in the next sections, the `TezosProvider` and `EthereumProvider` classes implement different strategies for indexing: in the Tezos case, transactions are queried at each block, parsed, and stored along with their events; whereas in the Ethereum case, it is events that are queried from the nodes, parsed and stored, from which transactions are then stored as well. This is in parts due to the representation of internal transactions by each blockchain on one hand, and due to the query optimisations that could be made with these strategies on the other.

5 Data model

This section describes the data model used by DjWebDapp to operate. Indeed, although the ultimate goal of DjWebDapp is to allow developpers to index, normalize and deploy smart contract transactions, some elements need to be defined first. A transaction needs

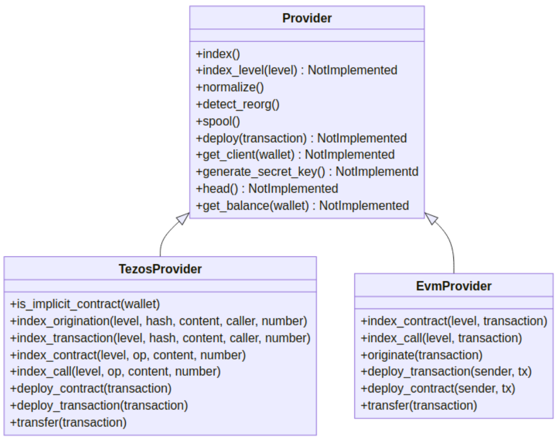
Provider
+index()
+index_level(level) : NotImplemented
+normalize()
+detect_reorg()
+spool()
+deploy(transaction) : NotImplemented
+get_client(wallet) : NotImplemented
+generate_secret_key() : NotImplementd
+head() : NotImplemented
+get_balance(wallet) : NotImplemented
TezosProvider
+is_implicit_contract(wallet)
+index_origination(level, hash, content, caller, number)
+index_transaction(level, hash, content, caller, number)
+index_contract(level, op, content, number)
+index_call(level, op, content, number)
+deploy_contract(transaction)
+deploy_transaction(transaction)
+transfer(transaction)
EvmProvider
+index_contract(level, transaction)
+index_call(level, transaction)
+originate(transaction)
+deploy_transaction(sender, tx)
+deploy_contract(sender, tx)
+transfer(transaction)

Figure 2: Provider class hierarchy

a blockchain to be injected into, a blockchain node to query, and accounts to deploy transactions with.

5.1 Blockchain

The most elementary table is that describing what blockchain is being operated on. It keeps track of essential data such as the level of the last indexed block, how many blocks are necessary for a transaction to be considered “confirmed”, the provider that is to be used when working with this blockchain object, along with an extra JSON field to allow adding some information without needing to subclass the blockchain object and commonly used helpers.

For flexibility reasons, the node associated with this blockchain object is not defined the the `Blockchain` table but in a new `Node` table defining a foreign key relationship with the blockchain it should be associated with. Indeed, a blockchain is composed of many nodes, some of which may temporarily fail. It is thus useful to define fallback nodes to use when others are down. It is the provider’s job to define a load balancing strategy to use when deciding which node to query. A `priority` attribute is defined on `Node` objects to help the load balancer determine which nodes to prioritize when querying no-chain data.

Both `Blockchain` and `Node` objects have a mutable attribute `is_active` that allows them to be activated and deactivated at will.

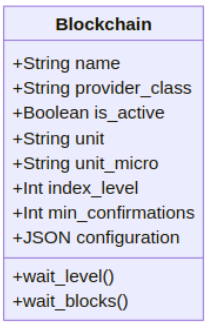

Figure 3: Blockchain class diagram

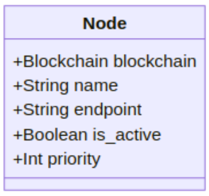

Figure 4: Blockchain node class diagram

5.2 Account

In order to interact with the blockchain, one needs an account, otherwise named as a wallet. DjWebDapp defines for that purpose an `Account` database table that provides the necessary information needed to interact with other accounts and smart contracts on the blockchain they are assigned to. This model symmetrically encrypts the account’s secret key using Django’s fernet fields app [17]. This app has the benefits of being able to rotate the encryption key by providing multiple keys in the `settings.FERNET_KEYS` list and fallbacking to `settings.SECRET_KEY` if none is provided. This ensures that should the secret key be compromised, updating the encryption scheme remains possible.

Morever, DjWebDapp leverage Django’s `User` model and allows associating many blockchain accounts to any user accounts, thus integrating with the rest of the Django user management system (such as authentication, management groups, . . .).

Since accounts have a relation with the blockchain model they are attached to, they also benefit from all of the provider’s available methods to interact with the chain. This provider is made accessible via a directly accessible python attribute property without having to specify the provider used at the account level. This property will also load the account’s private key when calling the provider’s blockchain client allowing for cleaner code since this pattern is often made use of.

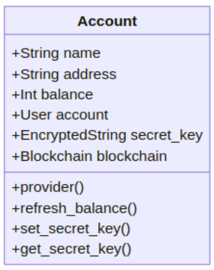

Figure 5: Account class diagram

5.3 Transactions

The final abstraction needed by DjWebDapp to function is that of transactions. Since these can differ between blockchains, it is possible to leverage Django’s model inheritance to simplify working with multiple blockchains. Indeed, Tezos and Ethereum have significantly different transactional models. In particular, on

Tezos, it is possible to have transactions bulks initialized by an external/implicit account where all transactions share the same hash and have different nonces, whereas on Ethereum this is not possible. Moreover, on Tezos, a smart contract call can create explicit on-chain internal transactions whereas on Ethereum, internal transactions are not included in the blockchain [2]. Hence, it is simple on Tezos to model internal smart contract calls at the database level with a foreign key on a single parent transaction whereas on Ethereum it is far from trivial, although possible by modifying the EVM clients to parse smart contract bytecode execution differently to account for the special instructions initializing cross-contract calls.

Moreover, EVM smart contracts being compiled in un-readable bytecode, require an ABI to decode smart contract call arguments, storages and events; whereas on Tezos, this is trivial due to the readable nature of the Michelson Bytecode. Hence, it makes sense to account for these differences at the database model level by inheriting from the `Transaction` class to make special `TezosTransaction` and `EthereumTransaction` models that allow for blockchain specific transactional attributes.

Both `TezosTransaction` and `EthereumTransaction` have Django model proxy models to differentiate between contracts and contract method calls and provide convinience methods for both where needed. Indeed, both contracts and method calls are modelled at the database level by a `Transaction` (or one of its subclasses) and these proxy methods help making code more readable.

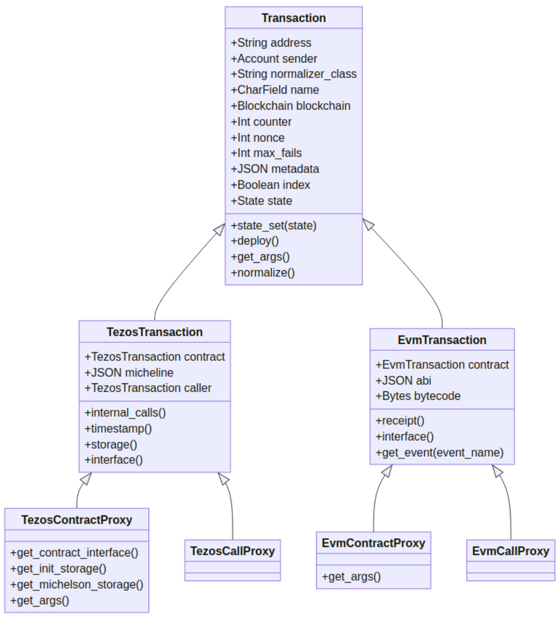

Figure 6: Transaction class hierarchy

5.4 Deployment graph

Sometimes, smart contract applications need to be divided accross multiple contracts that need to be deployed and configured with each other. For example, if one implements an NFT crowdsale contract that will mint contracts from an NFT contract, there will be a need to deploy both contracts and configure their permissions to ensure that they can communiacte together. Configuring these permissions will usually consist in passing the address of the NFT smart contract to the crowdsale contract and vice-versa. In this case, it is possible to define a deployment graph $G = (V, E)$ where the vertices V represent deployment and configuration transactions and edges E, the dependencies between these transactions. As will be explained in more detail in section 6.1, the configuration transactions cannot be injected until the smart contracts are deployed since they require the smart contract addresses that will only be known once those are deployed.

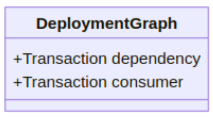

Figure 7: Dependency class diagram

6 DjWebDapp services

DjWebDapp is composed of three services. The spooler deploys asynchronously queued transactions, the indexer stores transactions and events targetting predefined contracts and the normalizer normalizes indexed transactions in a more suitable database schema.

6.1 Spooler

Deploying and interacting with contracts in an automated manner is particularly useful in the context of blockhain development.

Deploying a single smart contract is not difficult in itself, however, orchestrating the deployment of mutually dependent smart contracts can be challenging. Indeed, these deployment scripts are usually written in a synchronous way, meaning that a network error can result in needing to rerun the script from where it failed, potentially needing to copy/paste smart contract addresses in the script manually. DjWebDapp provides a mechanism to deploy smart contracts asynchronously, via a queue, automating retrying transactions in the correct order automatically.

Similarly, smart contract interaction automation; as can be necessary when developping cross-chain bridges, liquidation bots or oracles; can easily be implemented using DjWebdapp, using the same patterns as those used for deploying smart contracts.

Finally, unlike with most synchronous smart contract deployment or interaction scripts, since DjWebdapp uses a queue at the Django ORM level, all transactions and their states are stored in the database by default allowing for easy monitoring.

6.1.1 Deployment algorithms

Transactions to be deployed can be one of three types:

1. a smart contract deployment: the source code and initial storage are required;
2. a smart contract call: the target contract address and parameters are required along with an optional `amount` parameter, allowing to send native token to the target smart contract;
3. a native token transfer between external addresses: the amount of native token to transfer is necessary.

The field `Transaction.kind` allows one to specify what kind of transaction is being deployed. It can also be infered from the provided fields with the rules defined above if left out by the developper.

When creating a `Transaction` object, it is naturally stored in the database. Hence, the `Transaction` table can be considered a queue the spooler service can continually query for transactions to deploy. All transactions that are to be deployed by the spooler have one of two states:

1. `Transaction.state == DEPLOY`, set by the user when creating a transaction to deploy.
2. `Transaction.state == RETRY` , set by the spooler service when the transaction failed. The parameter `Transaction.max_fails` can be set to control the number of times a transaction is to be retried by the spooler before setting its state to `"aborted"`.

To speed up transaction spooling, it is necessary for the service to parallelize transaction deployments. However, due to the nature of how transactions are constructed, it is not always possible to parallelize transactions. Indeed, the transaction contains a number, called a **nonce**, which is required and incremented after each transactions made by a wallet. As such, when parallelizing trannsactions, it is not possible to inject two transactions with the same nonce. However, since nonces are predictable, it is possible , when spooling n transactions, to prefill nonces for transactions of the same wallet. However, if one of these transactions fails, all others will as well as they will not contain the proper nonces which are part of the transaction execution validation tests [16]. To remedee this constraint, the DjWebDapp spooler only parallelizes transactions that are from distinct sender wallets.

Finally, the DjWebDapp spooler introduced the notion of **dependency graph** when deploying transactions. Consider the problem of deploying a set of smart contracts such as an NFT platform consisting of an NFT contract and a crowdsale contract. At least three transactions are necessary:

1. deploy the NFT contract;
2. deploy the crowdsale contract with the NFT contract address contained in the initial storage;
3. configure the NFT contract to allow transactions incoming from the crowdsale contract.

Naively, one could deploy these three transactions chronologically and store them in the database accordingly. However, how should the case where one of the two first transactions fail be handled? Should the transaction to retry be retried until it is aborted? In the event that it fails until abortion, should transaction three be deployed? How should the spooler know the relation between these transaction, as other unrelated transactions to deploy could also be queued. Indeed, it is possible that the backend also maintains an oracle and needs to continually deploy transactions for the oracle to be up to date.

There is a natural dependency that's relating how these transactions should be deployed, and DjWebDapp allows one to define these dependencies at deployment to prevent the spooler from deploying transactions in the wrong order while not blocking unrelated transaction to spool. Hence, at deployement, the spooler will look for transactions dependencies by inspecting the `DeploymentGraph` table. It can then generate a topological order on this graph and deploy transactions in the proper order. Since topological sorts only exist on directed acyclic graphs [13], the `DeploymentGraph` graph object will not allow introducing cyclic dependencies. Both the topological sort and the acyclicity checks are implemented using NetworkX [11]. Moreover, in the current implementation of DjWebDapp, a transaction can only be part of a single graph, generalizing this to multiple graphs is left for future research. Indeed, currently, no applications have been found to require transactions to be part of multiple graphs on one hand, and since such an implementation has performances impacts on the other hand, it was decided to restrict transactions to a single graph. Note that, if a transaction has a failed dependency, its state is set to `HELD` in order to prevent it from being spooled at the next iteration.

Finally, once a transaction has been deployed, its state is set to `WATCHING`. Indeed, the only way to know if a transaction has been properly included in a block, is to index the blockchain by polling each block and look for the transaction hash in these blocks. Therefore, when a transaction is deployed, it is actually sent to the mempool awaiting to be included in a block. It is the job of the indexer service, presented in the next section, to look for these transactions and update the transaction state to `DONE`.

In the algorithms formalizing the above, $\mathbb{B}$ and $\mathbb{T}$ denote the set of all blockchain and transactions in the database respectively.

6.2 Indexer

DjWebDapp provides an indexer service, its purpose is to query store all raw transactions and events emitted to and from some predefined set of addresses to watch. This is useful for two reasons:

Algorithm 1 Deploy transaction

```
Require: $t \in \mathbb{T}$
  success ← inject($t$)
  if success then
    setState($t$, WATCHING)
  else
    setNFails($t$, getNFails($t$) + 1)
    if getNFails($t$) $\geq$ getMaxFails($t$) then
      setState($t$, ABORTED)
    else
      setState($t$, RETRYING)
    end if
  end if
```

Algorithm 2 Fetch transaction dependencies

```
Require: $t \in \mathbb{T}$
  $g$ ← getDependencyGraph($t$)
  $s$ ← topoSort($g$)
  for $t_i$ in $s$ do
    if $t_i == t$ then
      return $t$
    else if getState($t$) in [DEPLOY, RETRY] then
      return $t_i$
    else if getState($t$) in [ABORTED, HELD] then
      return None
    end if
  end for
```

Algorithm 3 Deploy transactions

```
Require: $b \in \mathbb{B}$
  $\mathbf{t_p}$ ← fetchPendingTransactions($b$)
  $\mathbf{t_d}$ ← filterDistinctSender($t_p$)
  $\mathbf{t}$ ← map($t_d$, fetchDependency)
  for $t_i$ in distinct($t$) do
    if $t_i \neq$ None then
      $p$ ← createSubprocess()
      runProcess($p$, $\lambda \rightarrow$ deploy($t_i$))
    else
      setState($t$, HELD)
    end if
  end for
```

1. When deploying a blockchain transaction, be it a smart contract deployment, call or transfer, the only way to determine that the transaction succeeded is to find it in a block since, by definition, the blockchain is only made up of its blocks' contents.

2. Developping blockchain based applications require showing blockchain data to the user. It is much more efficient to query it from some cache rather than from a node itself.

The indexer service stores all indexed transactions in the same `Transaction` objects that were used to deploy transactions in the previous section. It is the provider's job to index, and therefore to store the transaction in the appropriate subclass: the `EthereumProvider` will store indexed transactions in `EthereumTransaction` objects whereas the `TezosProvider` will store them in `TezosTransaction`.

The blockchain provider wil set indexed transactions to one of two states: `CONFIRMING` and `DONE`. Indeed, since there can be on-chain effects called **reorgs** that can reshuffle transactions in the best case [7], and remove them from the chain in the worst case. This is due to the decentralized nature of the blockcain where latency can cause multiple blocks to be mined at the same time, but not propagate at the same speed on the network, in which case, the node that mined the transaction on the longest chain wins [12]. The DjWebDapp indexer therefore needs to account for such on-chain effects, and update the state of thransactions that were indexed to `DELETED` allowing application backends and frontends to update in such a way as to only consider on-chain information.

Since thes on-chain events happen freqently, blocks are considered to be confirmed (with a state set to `DONE`) once their level is lower by some delta, defined in `Blockchain.min_confirmations`, from the blockchain head. Hence, when the indexer service indexes transactions, it reflects this block difference by chosing between these two states.

It shall be seen in section 6.3 that the distinction between `CONFIRMING` and `DONE` transactions can be useful to simplify normalizing smart contract data when idempotency of the normalizer is not possible or too time-consuming to implement properly.

Note that not all blockchains have the require the same number of blocks to consider a transaction to be confirmed. Although Nakamoto-style consensus mechanisms such as Bitcoin's or Ethereum's have a decreasing probability of transactions being reorgs based on their confirmation level, Tezos' new Tenderbake algorithm guarantees transactions older than two blocks to never be reorged [3].

6.2.1 Algorithms

Let $b \in \mathbb{B}$ be a blockchain to index. b_n denotes the n^{th} block already indexed where b_0 is the indexed block with the highest level, b_1 is the indexed block with the second highest level and so on.

Let us define two helper functions:

- hash(b_n) retrieves the hash for indexed block n.
- queryHash(b, n) queries the blockchain node for block n's hash.

As such, for a given block level, the hash at the time of the last index can be compared with the current hash. If these hashes differ, it means that the indexed block was reorged and should be invalidated. Since all indexed transactions have a relation on the block object they were indexed in, their states can efficiently be updated to `DELETED` in the event of a reorg.

Algorithm 4 Index blockchain

```
Require: $b \in \mathbb{B}$
  invalidateReorgedBlocks($b$)
  $l \leftarrow$ level($b_0$) or 0
  $h \leftarrow$ level(queryHeadBlock($b$))
  while $l \leq h$ do
      indexLevel($b, l$)
      $l \leftarrow l + 1$
      setIndexLevel($b, l$)
  end while
```

Algorithm 5 Invalidate reorged blocks

```
Require: $b \in \mathbb{B}$
Ensure: $l$, the highest non-reorged indexed block level
  $l \leftarrow$ level($b_0$)
  while hash($b_l$) $\neq$ queryHash($b, l$) do
      deleteBlock($b_l$)
      $l \leftarrow l - 1$
  end while
  setIndexLevel($b, l$)
```

Indexing strategies Indexing a blockchain is in itself simple, one needs to query each blocks individually and parse its information. However, on-chain information querying is blockchain specific. As a result, the same strategy cannot optimally be applied accross blockchain. Thanks to DjWebDapp's strategy pattern based provider architecture, it is possible to account for these differences.

Tezos Indexing the Tezos blockchain is very simple thanks to its RPC API. Querying a block returns all of the needed information to reconstruct the ledger in a single API call. This call returns all transactions, internal transactions and events emitted by each of these. As a result, the indexing routine implemented by `TezosProvider` is trivial, it consists in querying the block, iterating on all its transactions (internal transactions included) and storing transactions to index.

The benefits of the Tezos transaction execution model is that calls cross-contract calls architecture makes it trivial to parse internal transactions [10]. Consider the case where an NFT smart contract needs to be indexed, not only must direct calls to this smart contract need to be indexed, but also calls made by other contracts as weel. Indeed, if one needs to index the owners of each NFTs in this contract, then should one of the owner transfer his token via a marketplace smart contract, this transaction should be detected by the indexer. DjWebDapp's `TezosProvider` gaurantees that such internal transactions are correctly indexed using a single RPC call per block.

Algorithm 6 Index level Tezos

```
Require: $b \in \mathbb{B}, l \in \mathbb{N}$
  $\mathbf{c} \leftarrow$ getIndexedContracts($b$)          ▷ List of indexed contracts
  $\mathbf{a} \leftarrow$ getContractAddresses($\mathbf{c}$)      ▷ List of indexed contract addresses
  $b_n \leftarrow$ queryBlock($b, l$)
  for $t$ in getTransactions($b_n$) do
      $a_s \leftarrow$ getSenderAddress($t$)
      $a_d \leftarrow$ getDestinationAddress($t$)
      if $a_s \in \mathbf{a}$ or $a_d \in \mathbf{a}$ then
          $\beta \leftarrow$ saveBlock($b_n$)
          saveTransaction($t, \beta$)
      end if
  end for
```

Ethereum Unlike on Tezos, Ethereum nodes do not respond with the entire block information in a single call. In particular, it does not return events when querying the for a block, nor does it return internal transactions modelling cross-contract calls. Due to this second constraint, it is costly to compute these internal transactions [1] and much more efficient to rely on events, since they are stored on-chain and query-able.

EVM nodes provide the `eth_getLogs` endpoint that retrieve all events emitted for a list of smart contract addresses and event names over a range of blocks, in a single call. DjWebDapp makes use of this endpoint to query n blocks at once and index the transactions along with their events.

Algorithm 7 Index level Ethereum

```
Require: $b \in \mathbb{B}, l \in \mathbb{N}$
  $\mathbf{c} \leftarrow$ getIndexedContracts($b$)          ▷ List of indexed contracts
  $\mathbf{a} \leftarrow$ getContractAddresses($\mathbf{c}$)      ▷ List of indexed contract addresses
  $\mathbf{e} \leftarrow$ queryEvents($b, l, a$)
  for $e_i$ in $e$ do
      $h \leftarrow$ getTransactionHash($e$)
      $n \leftarrow$ getEventName($e$)
      $a \leftarrow$ getDestinationAddress($e$)
      $v \leftarrow$ getEventValue($e$)
      $b_l \leftarrow$ saveBlock($l$)
      $t \leftarrow$ saveTransaction ($a, h, b_l$)
      saveEvent($t, n, v$)
  end for
```

6.3 Normalizer

Arguably the most commonly needed service in a blockchain-based application: data normalization. Since DjWebDapp's indexer service only stores raw transactions and that applications often cannot query the blockchain data in its native format, data normalization is needed. Inspired by DipDup's [4] API, DjWebDapp allows one to attach an `Indexer`-inheriting class to a `Transaction` class. When this transaction is of kind `CONTRACT`, then each of this class' methods will be called when a transaction method or event is indexed.

Note that it is necessary to implement normalizers with idempotent methods such that normalized data can represent on-chain data accuratly in case of reorgs. Indeed, if a reorged transaction was already normalized, it should not affect the final result if it gets re-included in a future block an normalized again.

Since read-time idempotent normalizers are hard to implement, DjWebDapp's indexer distinguishes between indexed and confirmed transactions. This allows one to configure the normalizer such that it only normalizes confirmed transactions, preventing one from reorgs. On Tezos for example, since deterministic finality is guaranteed after two confirmation blocks [3], configuring the normalizer such that it only processes two blocks old transactions allows one to implement non-idempotency in the normalizer. Note however that event in such a case, idempotent normalizers allow for simpler re-indexation of a smart contract.

7 End-toEnd Testing WebDapps

End-to-end testing is often necessary for large applications to be continuously deployed. Since DjWebDapp provides an API for deploying, indexing and normalizing data in a already well-furnished web framework, it is possible to end-to-end test blockchain applications build on top of DjWebDapp without mocking third party services. Indeed, since the providers shiped with DjWebDapp only require a blockchain node to communicate with, and that those are available as stand-alone docker containers in most cases, it is easy to embark them in continuous integration pipelines. Deploying and normalizing a simple contract or a complex set of contracts can be implemented in a few line of end-to-end testable code running in a CI, without requiring mocking close-source dependencies.

8 Conclusion

DjWebDapp provides the necessary tools to create blockchain-agnostic decentralized apps. Years of iteration on similar problems have converged towards this unified, blockchain-agnostic, framework. Without reinvienting the whell, DjWebDapp merely builds on top of it, as a Django app.

Indeed, the blockchain is fully abstracted away with DjWebDapp, deploying and normalizing transactions is reduced to interacting with the Django ORM. Athough this paper focused on the framework internals, and required some domain-specific language, the goal for DjWebDapp is that no developper should need to understand these to develop a productive WebDapp. Up to a certain limit, to develop a performant app: a Django developper should have some SQL notions, and a Tezos smart contract should know have some basic Michelson knowledge. Similarly, a blockchain based application developper should have some blockchain intuition. Thus, it is rather the technicalities of the blockchain that DjWebDapp abstracts away more so than its fundamental concepts. As such, a neophyte should be able to develop WebDapps without requiring a thorough understanding of the underlying.

In other words, DjWebDapp should be considered a blockchain application kernel, providing an API comfortable enough to ease development. This kernel has been developped and tested, in production, at Pyratzlabs' and some of the startups it incubates. It has been developped from and for ground data and asks only to be built upon.

It is hoped that the provider architecture will be sufficiently flexible for applications to integrate different backends than blockchain nodes. Indeed, it is absolutely possible to spool and index using a different API than the node's, such a Venly's or Alchemy's for example. Indeed, applications making use of theses third party API to deploy and index transactions, still need to keep a version in their database for efficiency reasons. Hence DjWebDapp does not aim at replacing Venly, Tzkt, Alchemy, but merely integrate with them, while providing cost efficient methods to bypass them.

In order to facilitate this, Pyratzlabs will open-source many of the routines built on DjWebDapp that it has been developping, such as metadata stardard handling, token deployement via an API, and much more.

References

[1] Inc. Alchemy. *What are Internal Transactions?* 2023. URL: `https://docs.alchemy.com/docs/what-are-internal-transactions` (visited on 06/25/2021).

[2] Paul Apivat. *Learn Foundational Ethereum Topics with SQL.* 2021. URL: `https://ethereum.org/en/developers/tutorials/learn-foundational-ethereum-topics-with-sql/` (visited on 06/20/2021).

[3] Lăcrămioara Astefănoaei et al. "Tenderbake - A Solution to Dynamic Repeated Consensus for Blockchains". In: ().

[4] Baking Bad. *DipDup.* Version 6.5.7. June 20, 2023. URL: `https://github.com/dipdup-io/dipdup`.

[5] Baking Bad. *Pytezos.* Version 3.10.0. June 20, 2023. URL: `https://pytezos.org/`.

[6] Renu Bala and Kapil Kumar Kaswan. “Strategy Design Pattern”. In: *Global Journal of Computer Science and Technology: Software & Data Engineering* 14 (2014), pp. 34–38.

[7] corwintines. *Ethereum proof-of-stake attack and defense*. 2023. URL: `https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/attack-and-defense/` (visited on 06/25/2021).

[8] Django Software Foundation. *Django*. Version 4.1. June 20, 2023. URL: `https://djangoproject.com`.

[9] Ethereum Foundation. *Web3.py*. Version 6.5.0. June 20, 2023. URL: `https://github.com/ethereum/web3.py`.

[10] Tezos Foundation. *Michelson: the language of Smart Contracts in Tezos*. 2023. URL: `https://tezos.gitlab.io/active/michelson.html#inter-transaction-semantics` (visited on 06/25/2021).

[11] Aric Hagberg, Pieter Swart, and Daniel Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[12] Satoshi Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: ().

[13] Chaoyi Pang et al. “Topological sorts on DAGs”. In: *Information Processing Letters* 115 (20144), pp. 298–301.

[14] James Pic and Thomas Binetruy. *djWebdApp*. 2023. URL: `https://djwebdapp.readthedocs.io/en/latest/` (visited on 06/25/2021).

[15] PostgreSQL Global Development Group. *PostgreSQL*. Version 15.3.0. June 20, 2023. URL: `https://www.postgresql.org/`.

[16] Gavin Wood. “Ethereum: A secure decentralised generalised transaction ledger”. In: ().

[17] Yourlabs. *Django Fernet Fields*. Version 0.8.1. June 21, 2023. URL: `https://github.com/yourlabs/djfernet`.

A API

This appendix shows the basic example of deploying and normalizing a fungible token contract on both Tezos and Ethereum. Note how similar the API is, as most blockchain specifics were abstracted away by DjWebDapp.

A.1 Deploying contracts

Deploying a smart contract on Tezos and Ethereum is shown in this section. Note that in future works this should be abstracted away as enough information is provided, by Michelson source codes and the EVM ABI's, for these classes to be generated.

Nonetheless, DjWebdapp provides a mechanism to deploy transactions on the blockchain by interacting with the database through Django's ORM.

The full DjWebDapp documentation is available online [14].

A.1.1 Tezos

The **FA1.2** stadard defines a fungible token on Tezos. The `Fa12Contract` class demonstrates how one can keep track of token deployments easily through inheritance of the `TezosContract` class. The spooler will look for a file called `fa12.tz` and deploy it with the initial result calculated by `Fa12Contract.get_init_storage`.

```
class Fa12Contract(TezosContract):
    contract_file_name = "fa12.tz"

    admin = models.ForeignKey(Account)
    total_supply = models.BigIntegerField(
        default=0,
    )

    def get_init_storage(self):
        return {
            "admin": self.admin.address,
            "tokens" = {
                self.admin.address: \
                    self.total_supply,
            }
            "total_supply" = self.total_supply,
        }
```

A.1.2 Ethereum

The de-facto fungible token standard on Ethereum is the **ERC-20**. Below is shown a code example where the class `ERC20` inherits `EthereumContract`, provides its source code and ABI in the folder `ERC20`, and computes its initial storage.

```
class Erc20(EthereumContract):
    contract_name = 'ERC20'
    max_supply = models.BigIntegerField()

    def get_args(self):
        return (self.max_supply,)
```

A.2 Deploying smart contract calls

Similarly to smart contract deployment, interacting with a smart contract is implemented by subclassing `TezosCall` or `EthereumCall`, and defining a `get_args` method to compute the arguments to call the contract with.

Note that the call's target contract is typed at the database level through the `TezosCall.target_contract` foreign key overriding.

A.2.1 Tezos

```
class TransferCall(TezosCall):
    entrypoint = "transfer"

    target_contract = models.ForeignKey(
        Fa12Contract,
    )
    creditor = models.ForeignKey(Account)
    beneficiary = models.ForeignKey(Account)
    token_amount = models.BigIntegerField()

    def get_args(self):
        return {
            "from": self.creditor.address,
            "to": self.beneficiary.address,
            "value": self.token_amount,
        }
```

A.2.2 Ethereum

```
class TransferFromCall(EthereumCall):
    entrypoint = 'safeTransferFrom'

    target_contract = models.ForeignKey(CapsuleHD)
    from_address = models.CharField(max_length=255)
    to_address = models.CharField(max_length=255)
    token_id = models.BigIntegerField()

    def get_args(self):
        return [
            self.from_address,
            self.to_address,
            self.token_id,
        ]
```

A.3 Normalizing contracts

Normalizing smart contract transactions and events is done by subclassing `Normalizer` and naming methods according to method or event names (the methods are case sensitive). Below, an example of **FA1.2** or **ERC-20** normalization of the `mint` method. For idempotency, the `mint` call is normalized into a `ToknMovement` table. An API can now sum all of the `TokenMovement` entries, excluding those refering to a transaction with `state == DELETED` and return properly normalized on-chain data in a reorg-robust manner.

A.3.1 Tezos

The following normalizer leverages the fact that transactions are indexed by a `TezosProvider`, allowing the developer to retrieve the transaction arguments and normalize them.

```
class Fa2Normalizer(Normalizer):
    def mint(self, call, contract, *args, **kwargs):
        destination = Account.get_or_create(
            address=contract.args['to'],
        )
        TokenMovement.objects.get_or_create(
            source=contract,
            destination=destination,
            token_id=contract.args["id"],
            tx=call,
        )
```

A.3.2 Ethereum

In Ethereum however, it is much simpler to normalize event data since there is no optimal way to normalize cross-contract calls directly. Below is an example of an Ethereum contract normalizer operating on events. The handler method defined is named after the `Mint` event rathen than the `mint` method. And as such, is passed the event content an its argument.

```
class Erc20Normalizer(Normalizer):
    def Mint(self, event, contract, *args, **kwargs):
        destination = Account.get_or_create(
            address=event['to'],
        )
        TokenMovement.objects.get_or_create(
            source=contract,
            destination=destination,
            token_id=event['tokenId'],
            tx=call,
        )
```