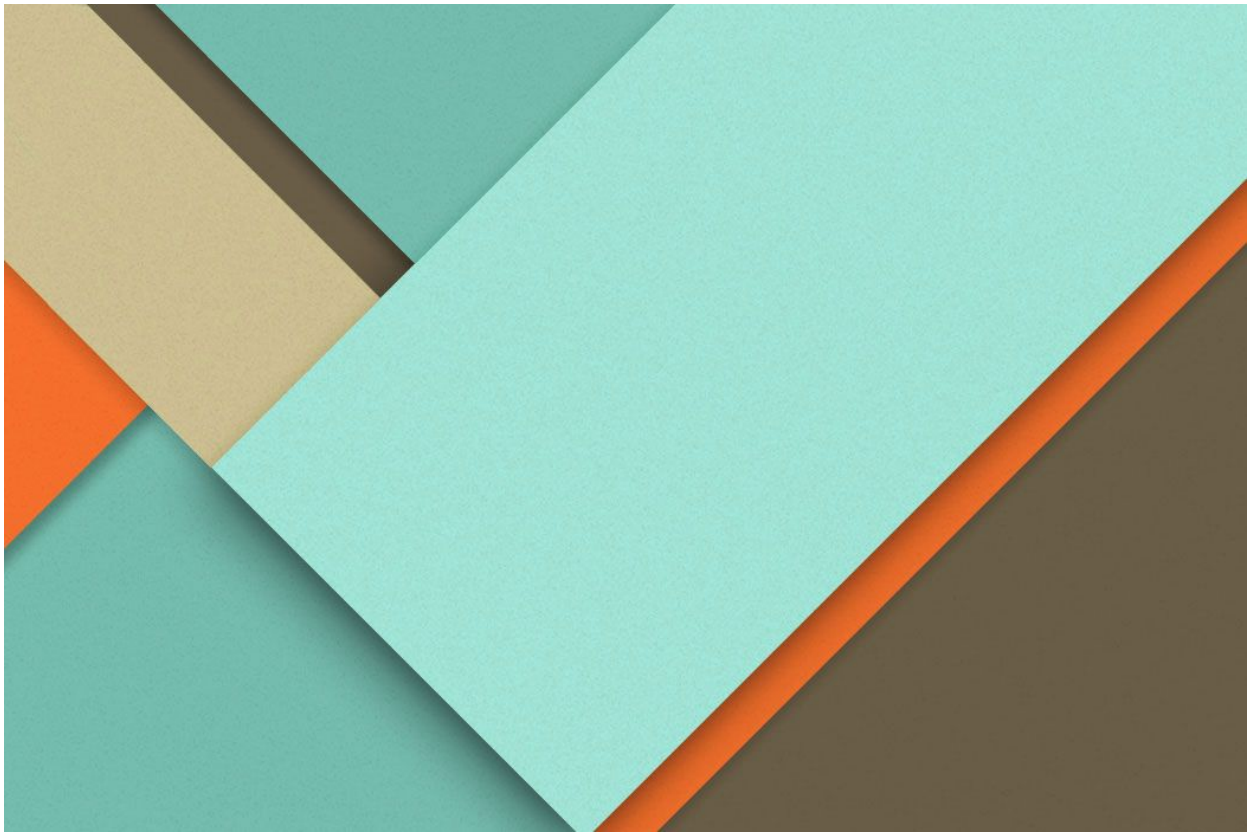


Blockchain Explorer

12/10/2016

Mahesh R. Gutala
DTCC
18301 Bermuda Isle Dr
Tampa, FL 33647

Revisions

Date	Revision	Description	Author
12/10/2016	0.1	First Draft	Mahesh Gutala
01/10/2017	0.2	Added data points. Changed to use Jade templates.	Mahesh Gutala
01/23/2017	0.3	Fixed issues found during review	Mahesh Gutala
02/03/2017	0.4	Added metadata support	Mahesh Gutala

Overview

Blockchain explorer provides a dashboard for viewing information about transactions, blocks, node logs, statistics, and smart contracts available on the network. Users will be able to query for specific blocks or transactions and view the complete details. Blockchain explorer can also be integrated with any authentication/authorization platforms (commercial/open source) and will provided appropriate functionality based on the privileges available to the user.

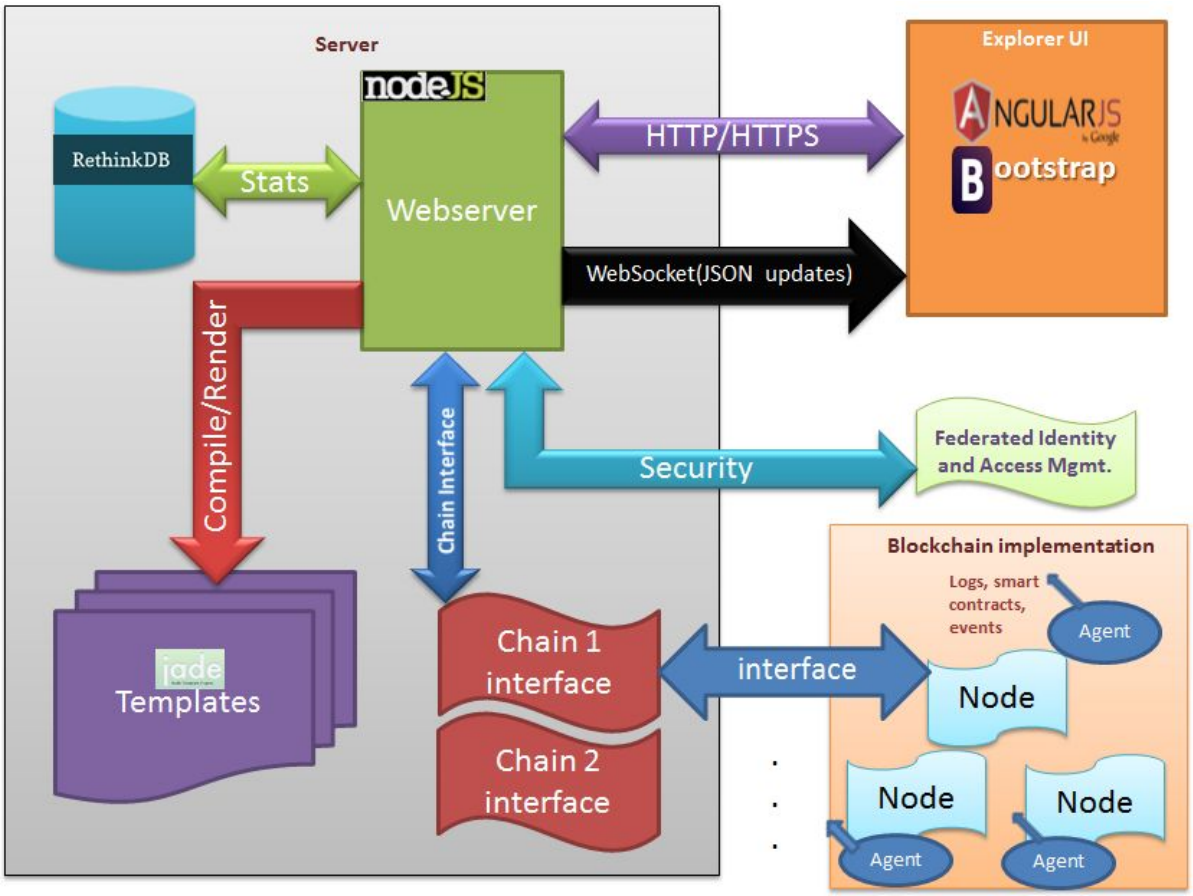
Architecture of the blockchain explorer, API needed to integrate with blockchain platforms is defined clearly later in this document.

Goals

1. To implement a generic Blockchain explorer web application which is easy to install and can be used with different Blockchain platforms.
2. Use latest tools and technologies that make the explorer easy to implement, maintain and extend.
3. Easily installable package available through standard package managers for most popular platforms.

Architecture and components

The diagram below represents the proposed architecture, various frameworks and tools that will be used.



Webserver

Node.js will the backend framework for implementing the server-side components. Express Node.js framework will be used for the web application. These tools are used as they are robust, easy to develop and maintain.

Web UI

AngularJS will be used to implement the front-end framework. AngularJS features like data-binding and directives greatly help in developing reusable components and modular code. Bootstrap will be used for front-end for its rich UI and responsive features.

Websockets

Websocket API will be used to push information from the server to the clients. Information about new transactions, blocks, node logs etc. are pushed via Websocket API. This is a convenient API that reduces load on clients and server.

Jade Templates

Jade is the templating engine that will be used for customizing the pages based on user permissions and roles.

RethinkDB

RethinkDB will be the datastore for the explorer. The information about blocks, transactions, smart contracts etc will be stored in this database. This is a great database for real-time web applications as the updates will be pushed from database instead of the application polling for data. Live statistics will be implemented using ReQL feature of the database. ReQL is a SQL-like language for querying data.

Security

User identity and access management will be implemented using a federated security repository. This will be a facade for security implementations from different Blockchain platforms.

Blockchain implementation

Each Blockchain implementation additionally will have an agent which will be an external software component that will provide updates on transactions, blocks, node logs and smart contracts to the explorer web server. This can be via Websockets.

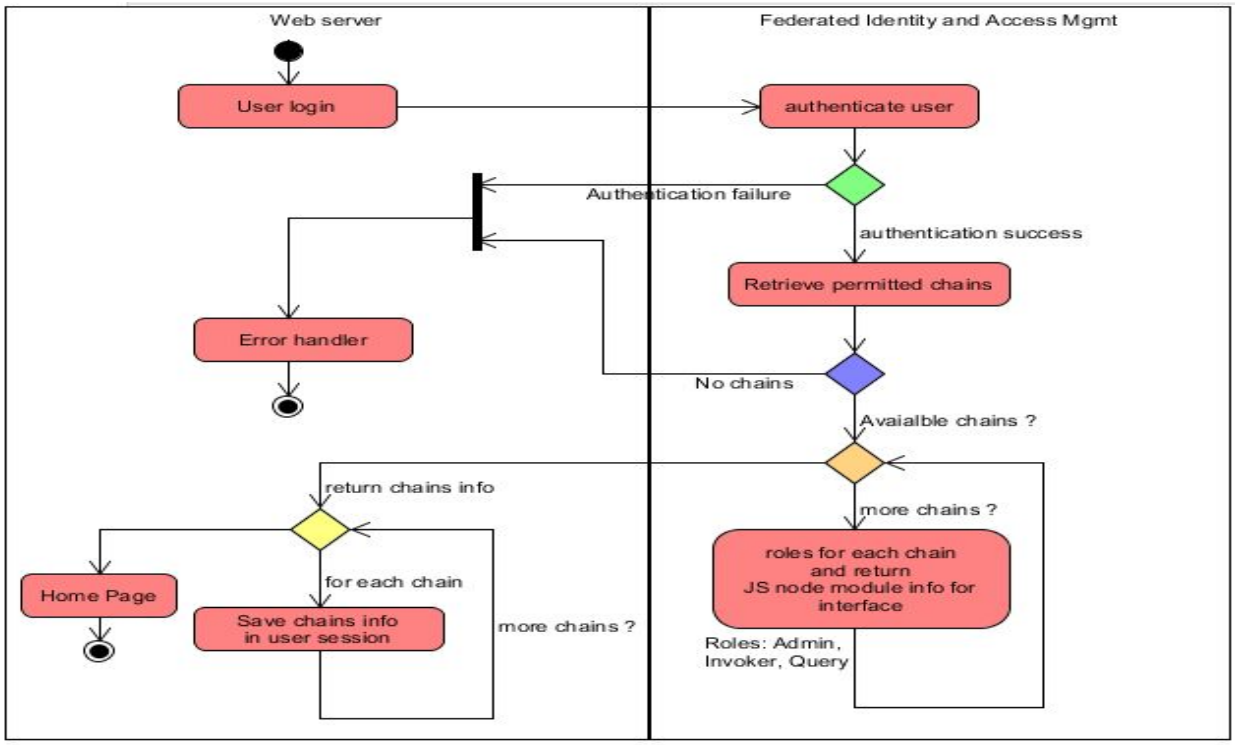
API

I. User login, Available chains and Roles

User is presented with a login page. User credentials are passed to the federated identity and access management(IAM). User is authenticated and authorization check done for the available chains. For each authorized chain, available roles(Admin, Invoker and Query) are determined. Metadata for each authorized chain such as node module to load, URL to retrieve node module is returned to

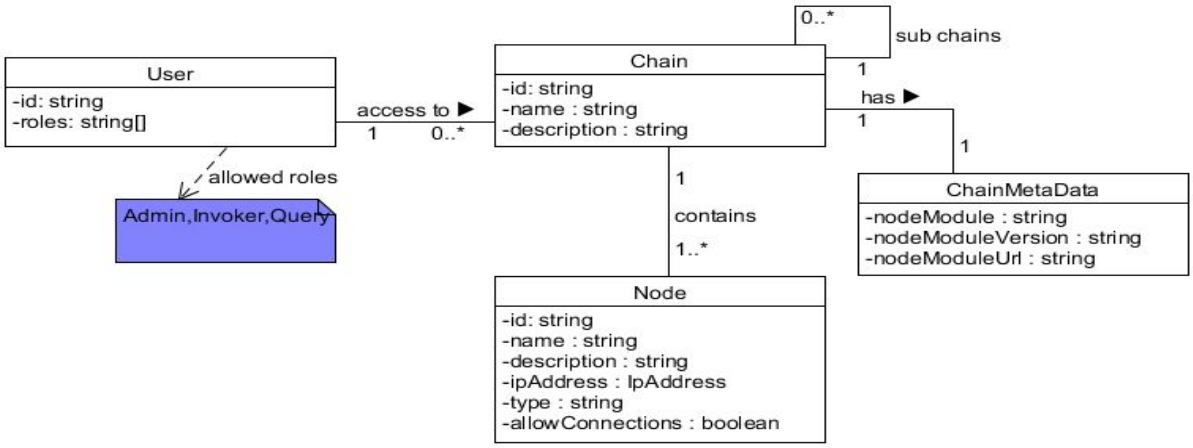
the server. If the module is not already loaded, web server uses this information to dynamically load the node module.

II. User login, Available chains and Roles Activity Diagram



Activity Diagram

III. User login, Available chains and Roles Data Model

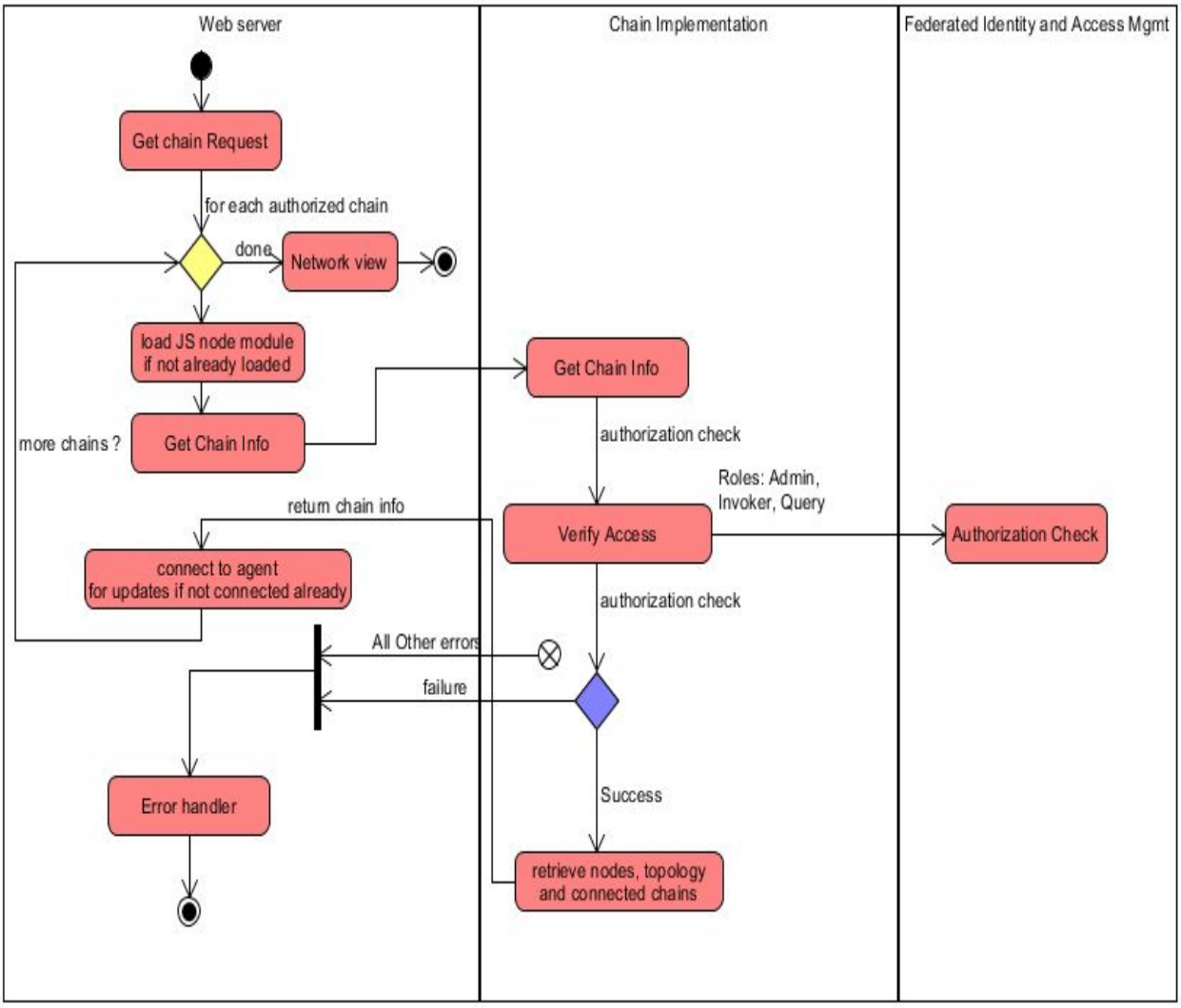


Data points

IV. Get Chain Info

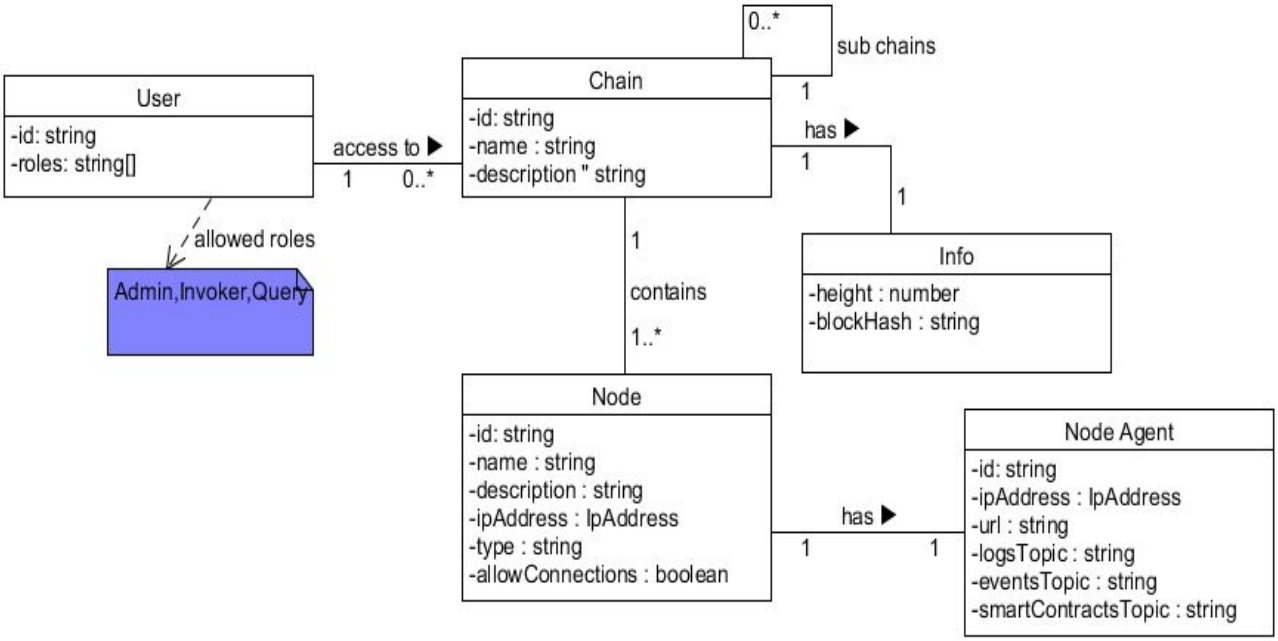
Get Chain request retrieves chain and its associated nodes information. It also retrieves information for all connected chains. The information returned will be the chain identity, current chain height, latest block information, IP address for each node, identity of each node and node agent port for updates.

V. Get Chain Info Activity Diagram



Activity Diagram

VI. Get Chain Info Data Model

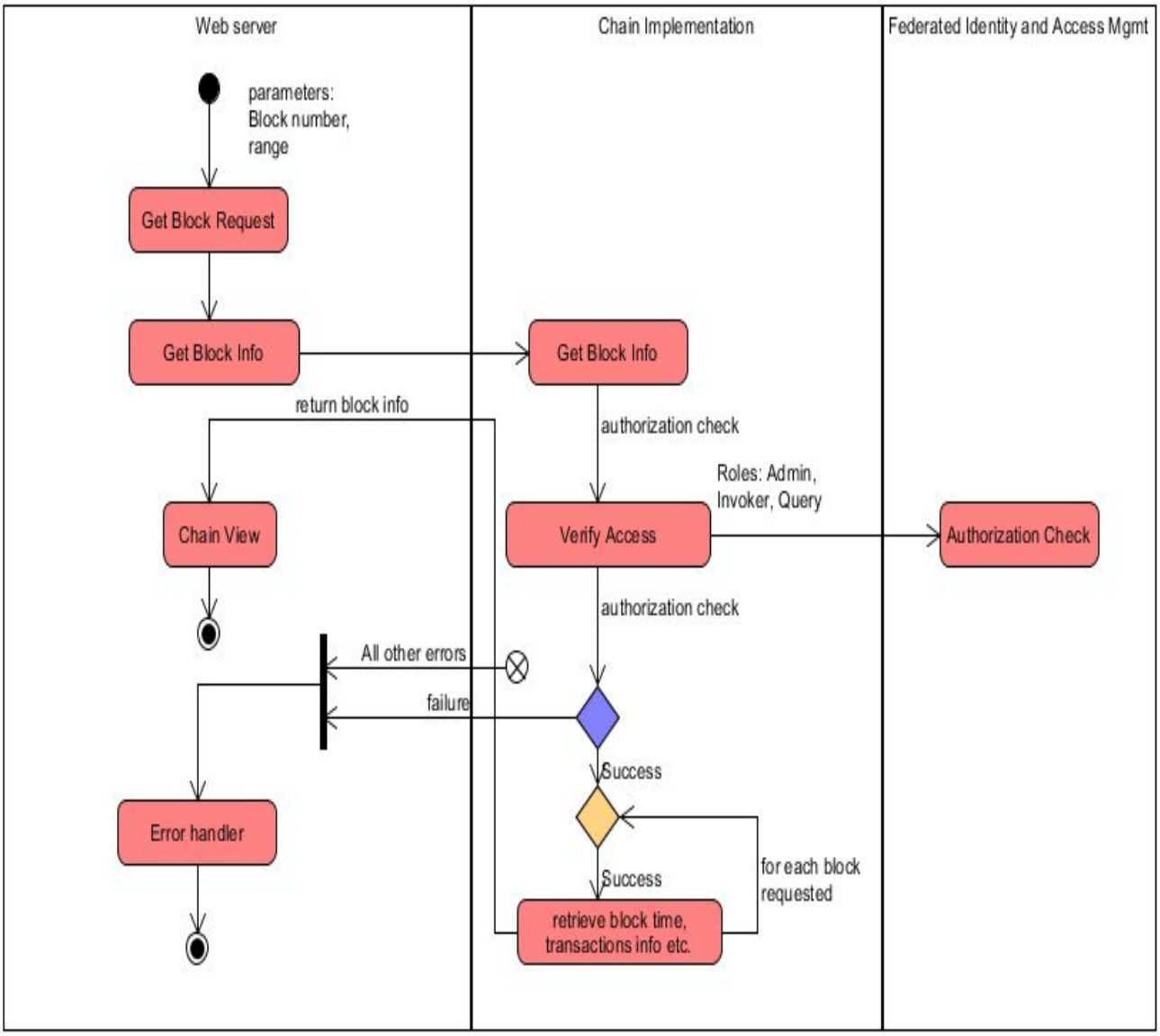


Data points

VII. Get Blocks

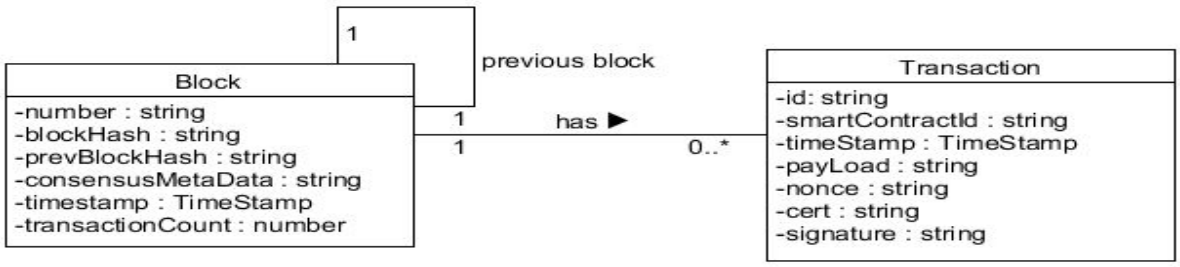
Get block information for specified block numbers or range. This will be used on the Web client to retrieve a specific block or range for view. Block information includes Block hash, timestamp and list of transactions. Transaction info includes transaction id, smart contract id, hashes etc.

VIII. Get Blocks Activity Diagram



Activity Diagram

IX. Get Blocks Data Model

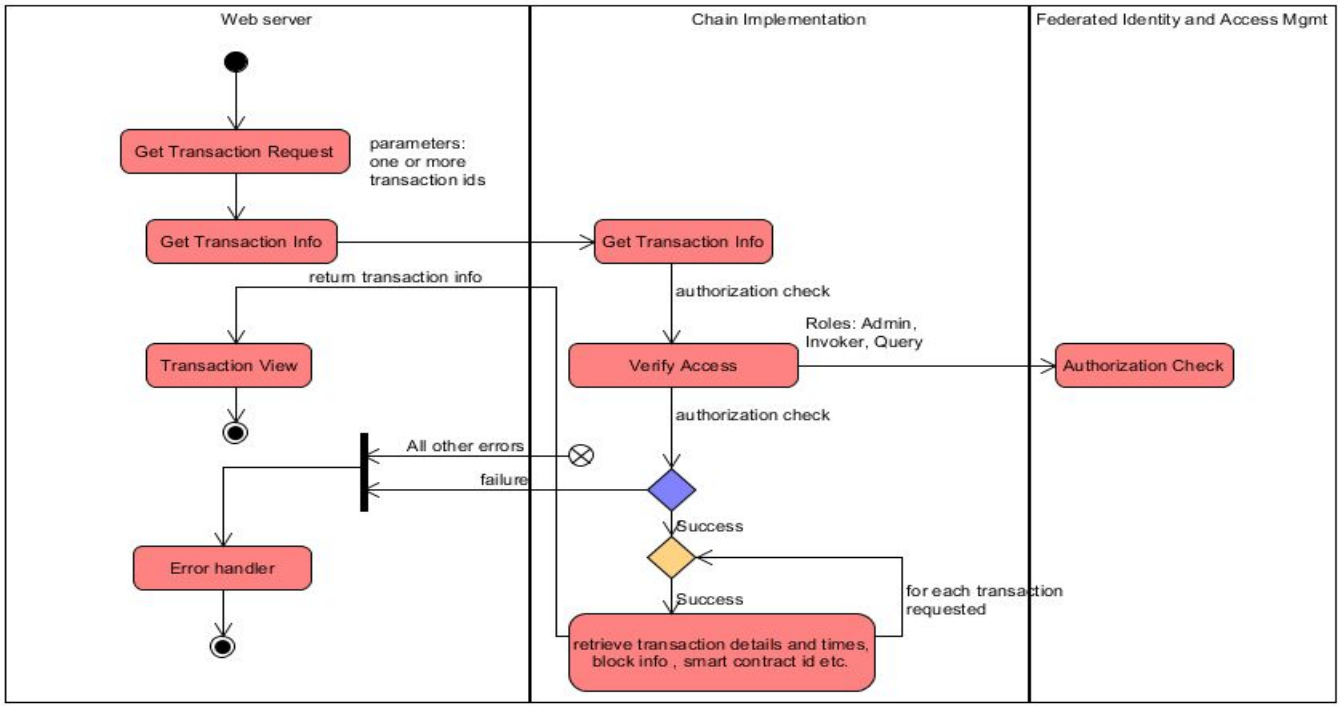


Data points

X. Get Transactions

Retrieve transactions information based on user search or view details of a transaction.

XI. Get Transactions Activity Diagram



Activity Diagram

XII. Get Transactions Data Model

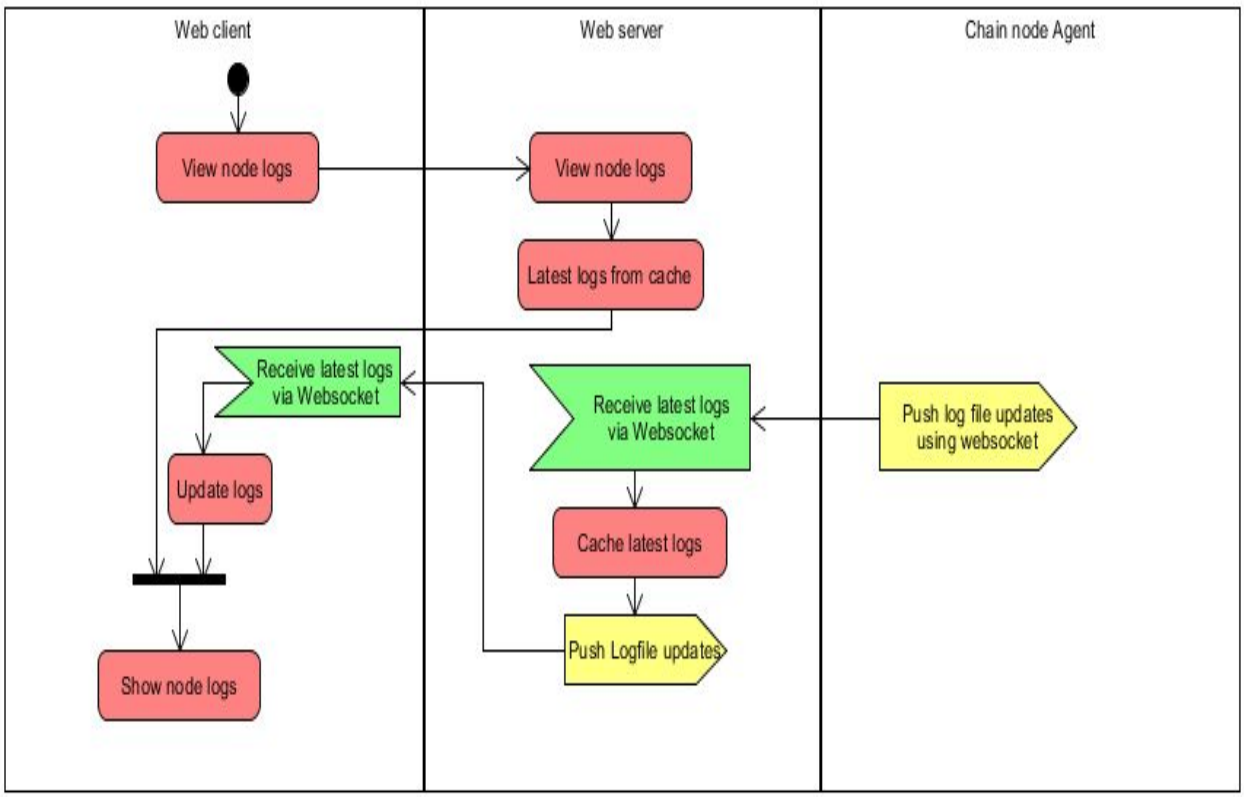
Transaction
-id: string -smartContractId : string -timeStamp : TimeStamp -payLoad : string -nonce : string -cert : string -signature : string

Data points

XIII. Node Logs

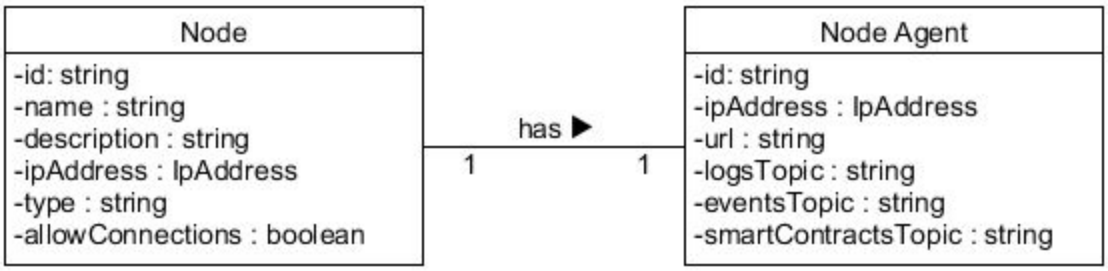
This is expected to be a functionality outside of the Chain implementation. Node agent runs independently, collects logs and publishes the information over Websockets(logsTopic) for webclient to display.

XIV. Node Logs Activity Diagram



Activity Diagram

XV. Node Logs Data Model

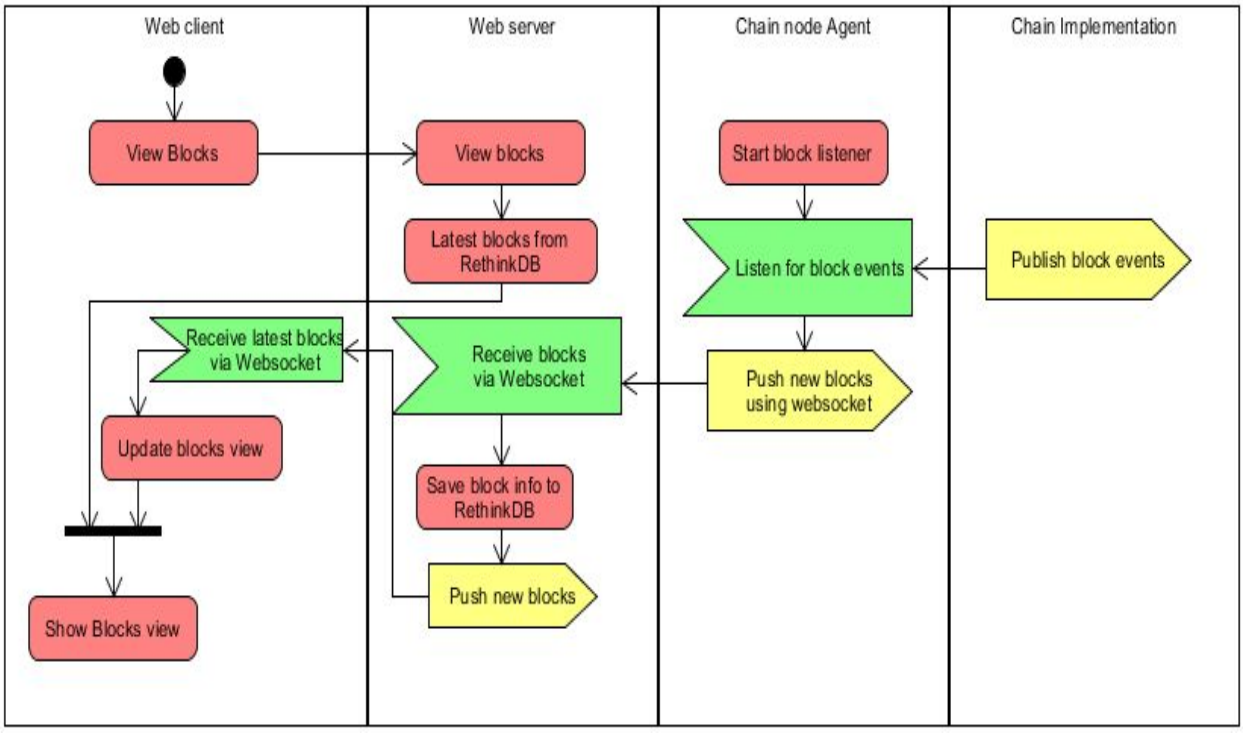


Data points

XVI. Block Events

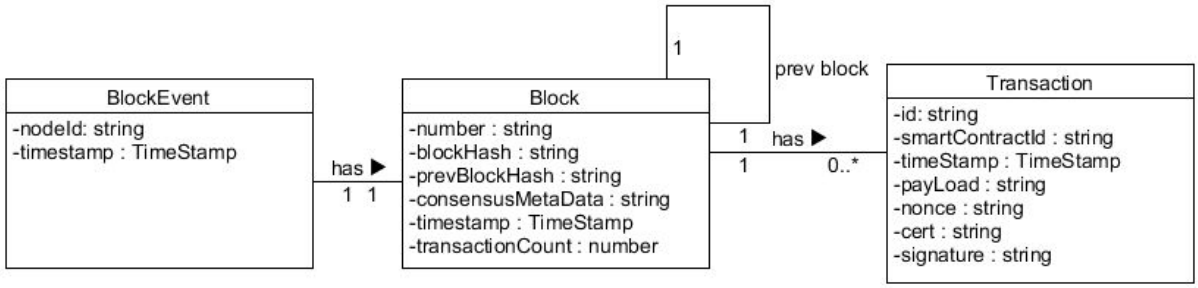
As new blocks are created, Chain implementation publishes events which are captured in the node agent and sent over to the server. Web server saves the block information to RethinkDB. Live statistics are built based on the block information and published to web client.

XVII. Block Events Activity Diagrams



Activity Diagram

XVIII. Block Events Data Model

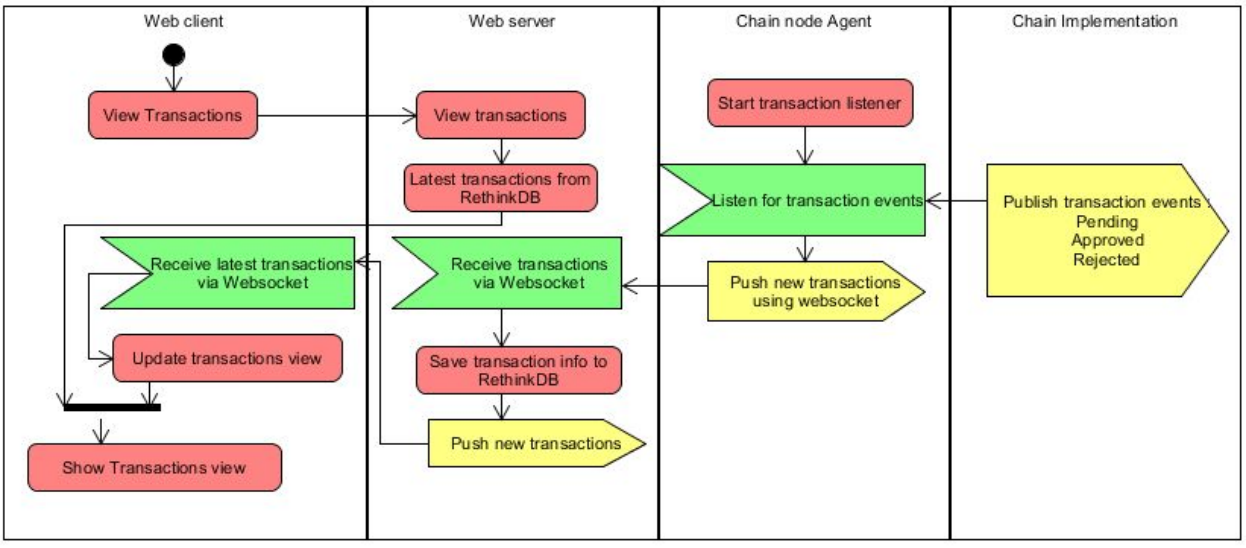


Data points

XIX. Transaction Events

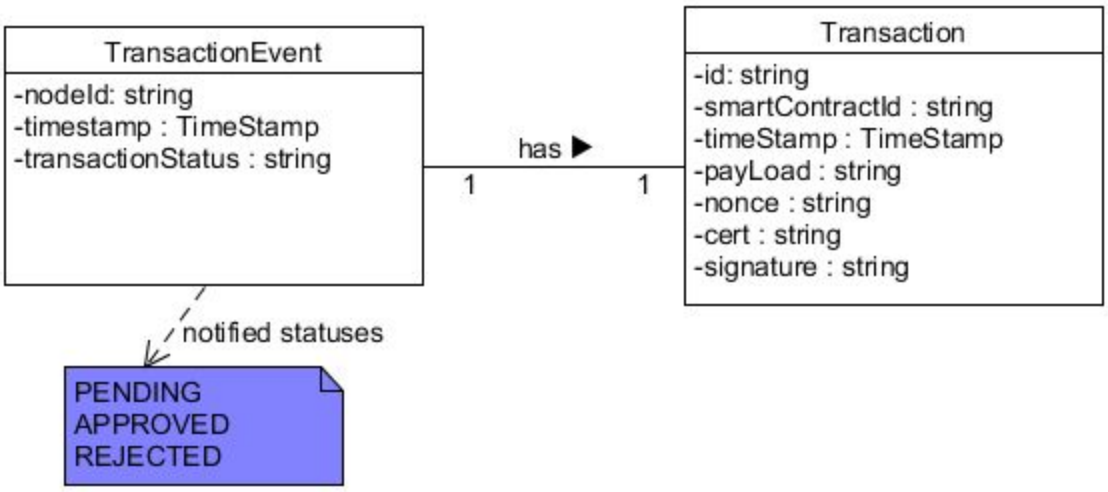
Chain implementation publishes information on each transaction when it is first submitted(pending) and when it gets approved or rejected. This information will be saved in the RethinkDB and used for building and publishing updates to the transactions view.

XX. Transaction Events Activity Diagram



Activity Diagram

XXI. Transaction Events Data Model

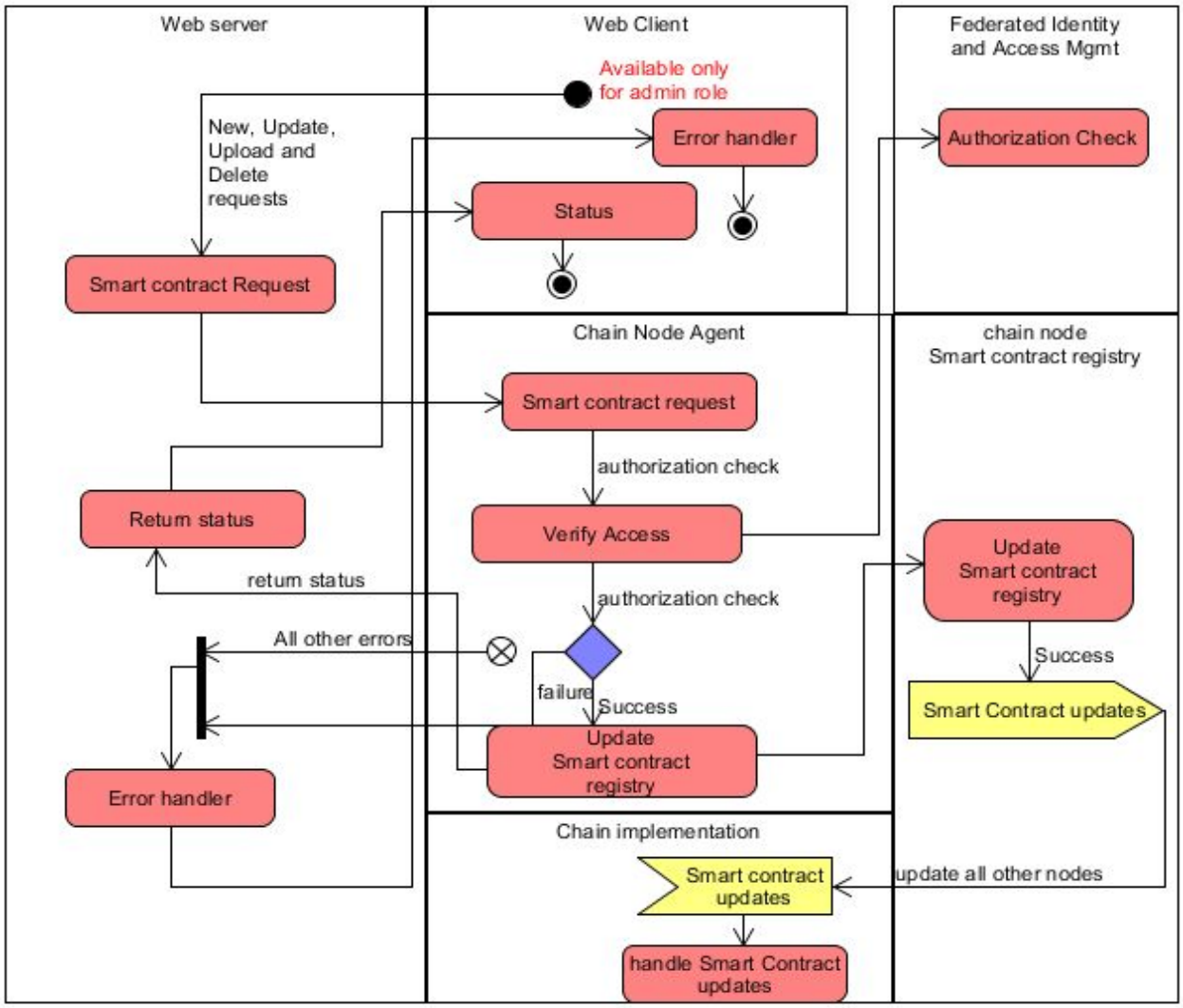


Data points

XXII. Smart Contracts(New, Update, Delete and Upload)

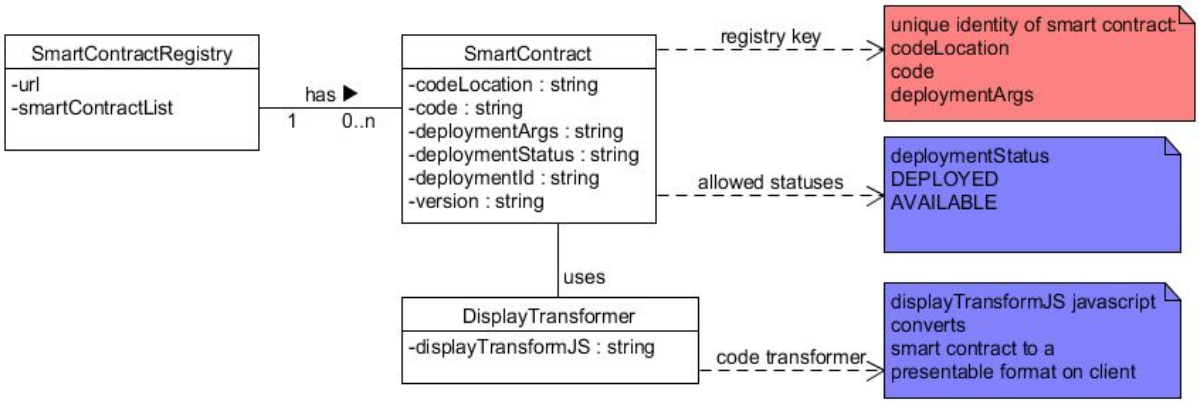
New or updates to smart contracts are handled external to chain implementation by node agent and coordinated by the registry. Updates of smart contracts are available only to users with Admin roles. Deploy, invoke and query are internal to chain implementation. When edits are done to smart contract code, changes will be communicated to one node but have to be persisted to the smart contract registry. Changes to smart contracts will be notified to Chain implementations to handle versioning, additions, updates and deletion internally.

XXIII. Smart Contracts(New, Update, Delete and Upload) Activity Diagram



Activity Diagram

XXIV. Smart Contracts(New, Update, Delete and Upload) Data Model

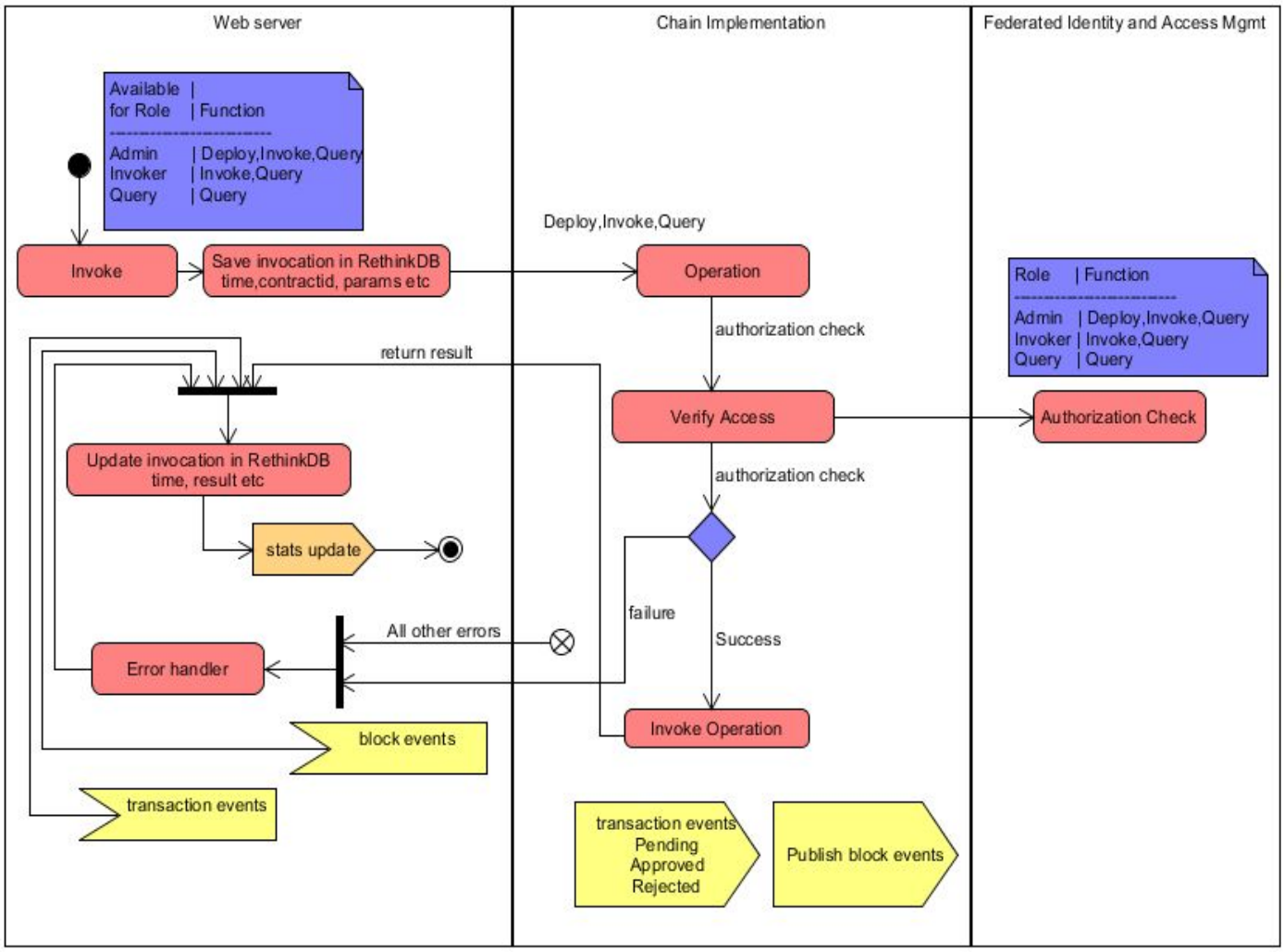


Data points

XXV. Smart Contracts(Deploy, Invoke and Query) and Live Statistics

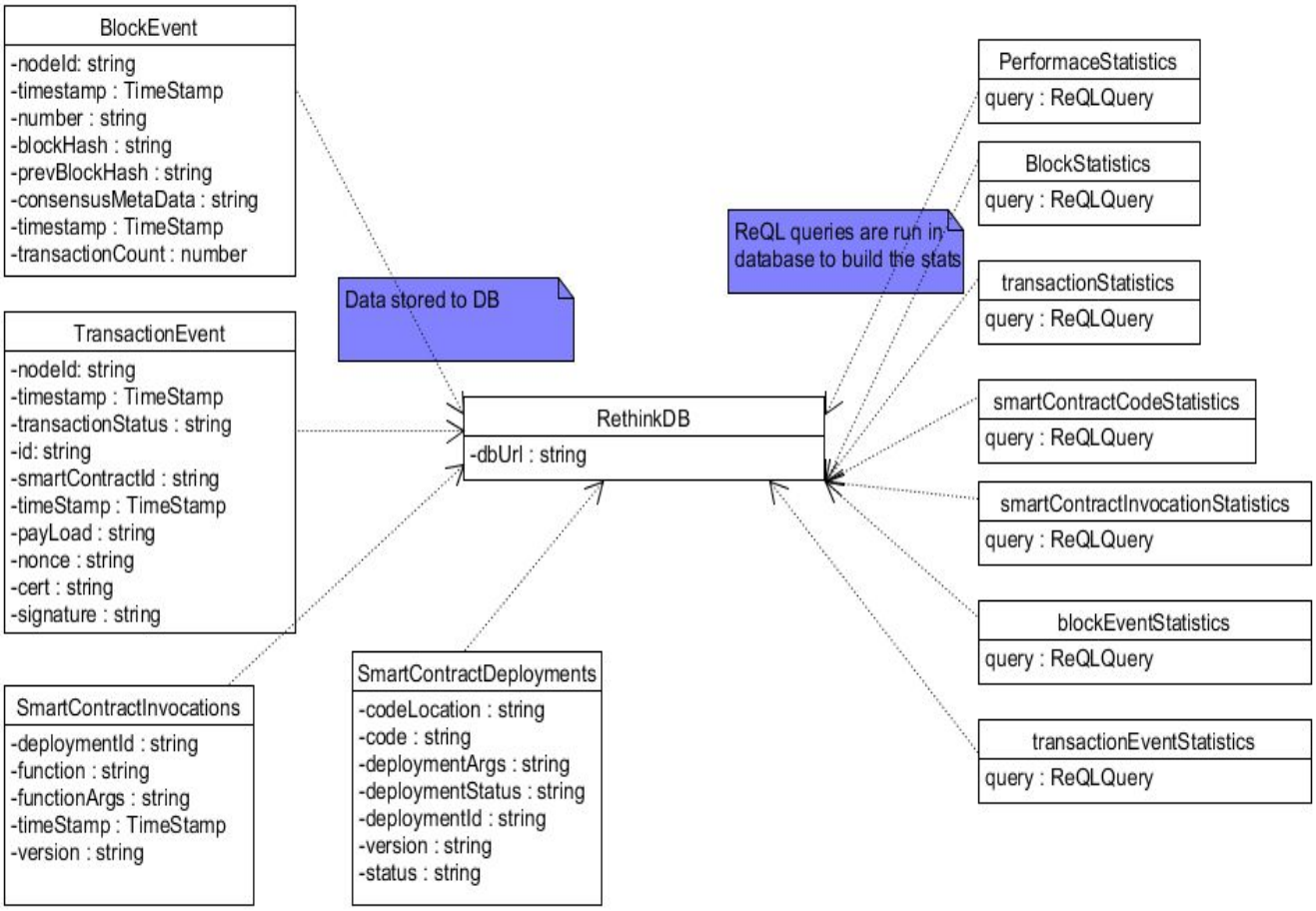
Smart contract operations are available only to certain roles as shown below. Each operation on a smart contract and associated events are persisted to the RethinkDB and live statistics are built and published to the web client. Transaction events are for new invocations, approvals and rejections. Each of this event is persisted and statistics are produced. Block events are also used to create stats for blocks.

XXVI. Smart Contracts(Deploy, Invoke and Query) and Live Statistics Activity Diagram



Activity Diagram

XXVII. Smart Contracts(Deploy, Invoke and Query) and Live Statistics Data Model



Data points

XXVIII. Ledger metadata for explorer customization

To support different ledger implementations, the interface node modules need to provide metadata for web client to display information like titles, descriptions etc. Also the metadata has boolean flags to indicate if a particular interface is available. Below is the expected format of metadata. None of the fields is mandatory. If a particular metadata is not provided, the interface is considered not available.

LedgerMetaData : {

title : string, /*Title to be displayed for the ledger implementation */

description : string,

securityEnabled : boolean, /* flag to determine role-based checks need to be done. Allowed roles are Admin, Invoker and Query */

ChainMetaData : {

title : string,

description : string,

available : boolean ,

FieldMetaData : [

{

fieldName : string,

filedDesc : string

fieldLabel : string

}

],

InfoMetaData : {

title : string,

description : string,

available : boolean ,

FieldMetaData : [

{

fieldName : string,

filedDesc : string

fieldLabel : string

}

]

},

NodeMetaData : {

title : string,

description : string,

available : boolean ,

```
            FieldMetaData : [

                {
                    fieldName : string,
                    filedDesc : string,
                    fieldLabel : string

                }
            ]

        },

        NodeAgentMetaData : {

            title : string,
            description : string,
            available : boolean ,
            nodeLogsAvailable : boolean, /* flag to check if node logs are available to be displayed on the
client */
            eventsAvailable : boolean, /* flag to check if block and transaction events are available */
            smartContractsAvailable : boolean, /* flag to check if smart contracts are available */

            FieldMetaData : [

                {
                    fieldName : string,
                    filedDesc : string,
                    fieldLabel : string

                }
            ]

        },

        /*Block meta data to describe the ledger implementations with transactions within the blocks*/
        BlockMetaData : {
            title : string,
            description : string,
```

```
            available : boolean ,
            FieldMetaData : [

                    {
                            fieldName : string,
                            filedDesc : string,
                            fieldLabel : string

                    }
            ],

            TransactionMetaData : {
                    title : string,
                    description : string,
                    available : boolean ,
                    FieldMetaData : [

                            {
                                    fieldName : string,
                                    filedDesc : string,
                                    fieldLabel : string

                            }
                    ]

            },
    },

    BlockEventMetaData : {
            title : string,
            description : string,
            available : boolean ,
            FieldMetaData : [

                    {
                            fieldName : string,
                            filedDesc : string,
```

```
                    fieldLabel : string

                }
            ],

            TransactionEventMetaData : {
                title : string,
                description : string,
                available : boolean ,
                FieldMetaData : [

                    {
                        fieldName : string,
                        filedDesc : string,
                        fieldLabel : string

                    }
                ]

            },
},

/*Transaction meta data to describe the ledger implementations with no blocks*/
TransactionMetaData : {
            title : string,
            description : string,
            available : boolean ,
            FieldMetaData : [

                {
                    fieldName : string,
                    filedDesc : string,
                    fieldLabel : string

                }
            ]
},
```

```
TransactionEventMetaData : {
        title : string,
        description : string,
        available : boolean ,
        FieldMetaData : [

                {
                        fieldName : string,
                        filedDesc : string,
                        fieldLabel : string

                }
        ]
},

SmartContractMetaData : {
        title : string,
        description : string,
        available : boolean ,
        addAvailable : boolean, /* Flag to check if smart contracts can be added from client */
        modifyAvailable : boolean, /* Flag to check if smart contracts can be modified from client */
        deleteAvailable : boolean, /* Flag to check if smart contracts can be deleted from client */
        FieldMetaData : [

                {
                        fieldName : string,
                        filedDesc : string,
                        fieldLabel : string

                }
        ],

        DeployOperationMetaData : {
                title : string,
                description : string,
                available : boolean ,
```

```
        FieldMetaData : [

                {
                        fieldName : string,
                        filedDesc : string,
                        fieldLabel : string

                }
        ]

},

InvokeOperationMetaData : {
        title : string,
        description : string,
        available : boolean ,
        FieldMetaData : [

                {
                        fieldName : string,
                        filedDesc : string,
                        fieldLabel : string

                }
        ]

},

QueryOperationMetaData : {
        title : string,
        description : string,
        available : boolean ,
        FieldMetaData : [

                {
                        fieldName : string,
                        filedDesc : string,
```

```
                                    fieldLabel : string

                                }
                            ]

                        },

                }


}
```