EOSIO Web App Development

Who am I?

Jeffrey Smith

I am a software engineer
on the blockchain team at
block.one.

You can message me at:
jeffrey.smith@block.one

EOSJS

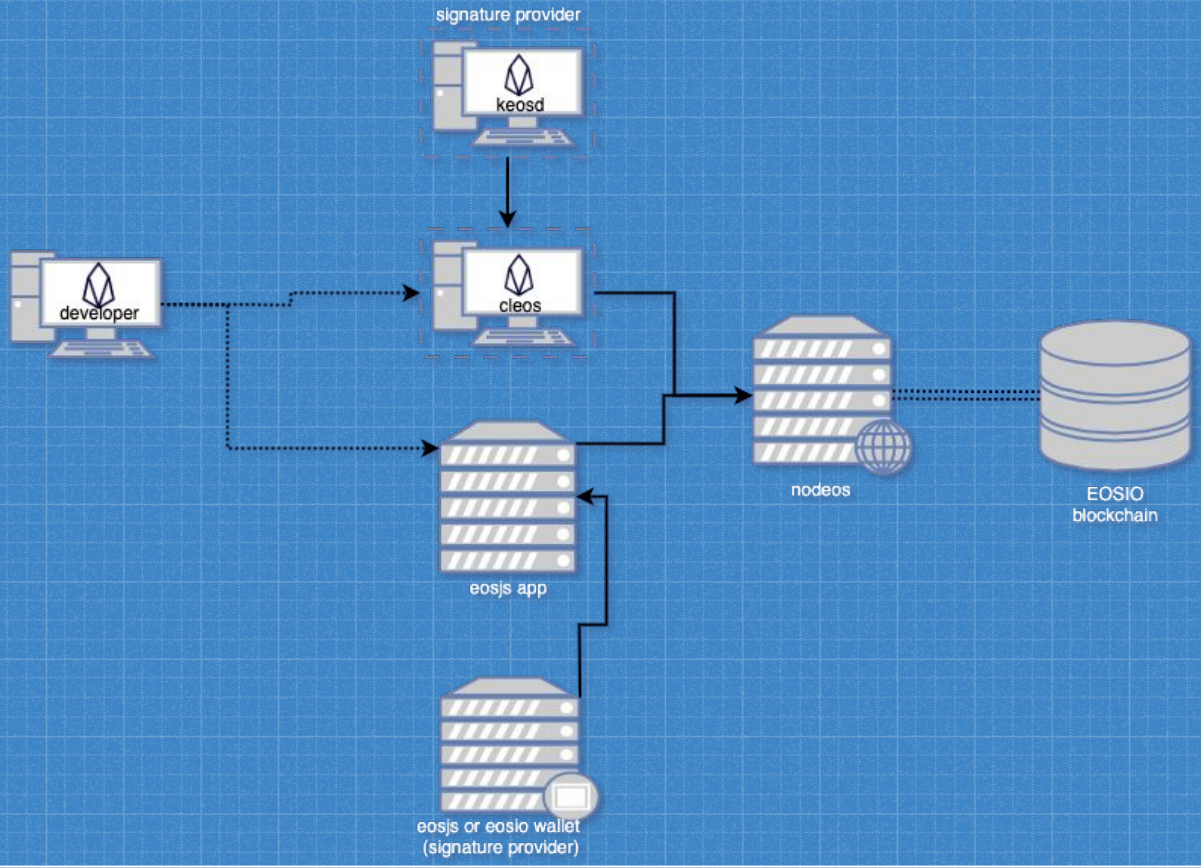
What is it and why do you need it.

EOSJS

EOSJS is a Javascript library that supports signing and broadcasting transactions, as well as interacting with other parts of the EOSIO RPC API.

It is one of the main ways of developing a Web Application that interacts with an EOSIO based blockchain.

EOSJS


signature provider
keosd
developer
cleos
nodeos
EOSIO
blockchain
eosjs app
eosjs or eosio wallet
(signature provider)

EOSJS

eosjs.Api

This class exposes some high level, convenience methods that are needed for basic interactions with an EOSIO blockchain.

The most important method that you will need is:

- **Api.transact**
  - This method signs a transaction, and optionally broadcasts it to the chain. This is the most important method to use, as it handles a lot of the complexity of signing and transaction formation/serialization for you.

EOSJS

eosjs.JsonRpc

This class exposes methods that are useful for directly querying the Nodeos RPC API.

Some important methods:

- **JsonRpc.get_info**
  - This method communicates with the chain to get information such as the head block number, chain id, etc.
- **JsonRpc.get_currency_balance**
  - Retrieves the balance of an account for a given currency.
- **JsonRpc.get_table_rows**
  - Returns an object containing rows from the specified table. We will cover this in more detail later in the talk.

EOSJS

Other Classes/Methods

These classes should mostly serve as an implementation detail, but should you have need of more advanced capabilities:

- **SerialBuffer**
  - This class handles serializing/deserializing data to and from the format Nodeos expects.
- **JsSignatureProvider**
  - Handles the action of signing a transaction with specified key(s).
- **Api.getAbi**
  - Returns an object containing the abi for a specific account.
- **Api.serializeTransaction**
  - Turns a JS object into a binary representation of the transaction.
- **Api.pushSignedTransaction**
  - Pushes a signed, serialized transaction to the chain.

Quick JS Refresher

Quick recap of JS asynchronous code

Async Javascript - Promises

- The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.[1]

- Promises are typically used for network/IO operations that can be done in a background “thread”.

- EOSJS makes use of Promises for many of its functions.
  - **Always** check the return type of a function and handle the Promise correctly to prevent race conditions!

Async Javascript - Promises vs Callbacks

- Before Promises, the only way to perform async code was using callbacks.

```
function getGoogle() {
    request('http://www.google.com', function(err, res) {
        response.json(res, function(err, data) {
            console.log(data)
        })
    })
}
```

```
function getGoogle() {
    fetch('http://www.google.com')
      .then(response => response.json())
      .then(data => console.log(data));
      .catch(() => console.error("error!"))
}
```

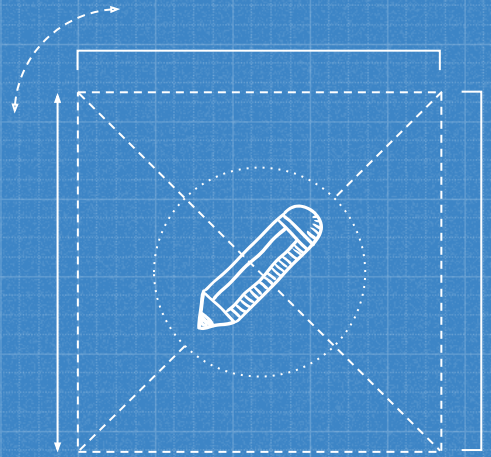
Async Javascript - async/await

- Javascript recently introduced the async/await syntax.
- This allows Promises to be written in a more sequential form, leading to code that looks closer to languages like Java/C#/etc.

Async Javascript - Promises vs Async/Await

```
function getGoogle() {
    fetch('http://www.google.com')
      .then(response => response.json())
      .then(data => console.log(data));
      .catch(() => console.error("error!"))
}
```

```
async function getGoogle() {
    try {
        const res = await fetch('http://www.google.com')
        const data = await response.json()
        console.log(data)
    } catch(e) {
        console.error("error!")
    }
}
```



Addressbook

A smart contract that acts as an address book and showcases actions and tables

Addressbook - Initializing EOSJS

```
// The JsonRpc class is used for communicating directly with a node's RPC API.
const rpc = new eosjs_jsonrpc.JsonRpc('http://127.0.0.1:8888');

// The JsSignatureProvider handles using private keys to sign transactions.
const signatureProvider = new eosjs_jssig.JsSignatureProvider(privateKeys);

// The Api class pulls together the JsonRpc and JsSignatureProvider
// to provide easy to use methods for signing and broadcasting transactions
const api = new eosjs_api.Api({ rpc, signatureProvider });
```

Addressbook - Creating an Entry

```
async function create_entry() {
  const form_info = get_form_info();
  try {
    const result = await api.transact({
      actions: [{
        account: 'addressbook',
        name: 'upsert',
        authorization: [{
          actor: form_info.user,
          permission: 'active',
        }],
        data: {
          user: form_info.user,
          first_name: form_info.first_name,
          last_name: form_info.last_name,
          age: form_info.age,
          street: form_info.street,
          city: form_info.city,
          state: form_info.state,
        },
      }]
    }, {
      blocksBehind: 3,
      expireSeconds: 30,
    });
    show_logs(result);
  } catch (e) {
    show_error(e);
  }
}
```

Addressbook - Creating an Entry

```
actions: [{                    // actions contains an array of transaction objects.
  account: 'addressbook',
  name: 'upsert',
  authorization: [{
    actor: form_info.user,
    permission: 'active',
  }],
  data: {
    user: form_info.user,
    first_name: form_info.first_name,
    last_name: form_info.last_name,
    age: form_info.age,
    street: form_info.street,
    city: form_info.city,
    state: form_info.state,
  },
}]
```

Addressbook - Creating an Entry

```
account: 'addressbook',   // the account the contract is deployed on.
name: 'upsert',           // the name of the action.
authorization: [{
  actor: form_info.user,  // the user authorizing the action.
  permission: 'active',   // the permission used for authorizing the action.
}],
```

Addressbook - Creating an Entry

```
data: {                                  // transaction data, changes per transaction.
  user: form_info.user,
  first_name: form_info.first_name,
  last_name: form_info.last_name,
  age: form_info.age,
  street: form_info.street,
  city: form_info.city,
  state: form_info.state,
},
```

Addressbook - Creating an Entry - TAPoS

- TAPoS is beyond the scope of this presentation, more info can be found in the whitepaper.
- The example contains sane defaults that should be sufficient for most applications.

```
}, {
  blocksBehind: 3,
  expireSeconds: 30,
});
```

Addressbook - Erasing an Entry

```
async function erase_entry() {
  const user = get_erase_user();

  try {
    const result = await api.transact({
      actions: [{
        account: 'addressbook',
        name: 'erase',
        authorization: [{
          actor: user,
          permission: 'active',
        }],
        data: {
          user,
        },
      }]
    }, {
      blocksBehind: 3,
      expireSeconds: 30,
    });
    show_logs(result);
  } catch (e) {
    show_error(e);
  }
}
```

Addressbook - Viewing the Data

```
async function get_table_all_elements() {
  const result = await rpc.get_table_rows({
    json: true,              // Get the response as json (if false, returns serialized form)
    code: 'addressbook',     // Contract that we target
    scope: 'addressbook',    // Account that owns the data
    table: 'people',         // Table name (as defined by eosio::multi_index<your_table_name here, ...>
    limit: 10,               // Maximum number of rows that we want to get
    reverse: false,          // Optional: Get reversed data
    show_payer: false        // Optional: Show ram payer
  });
  console.log(result);
  generate_table(result);
}
```

Addressbook - Viewing the Data with Filtering

```
async function get_table_by_bound(lower_bound) {
  const result = await rpc.get_table_rows({
    json: true,                 // Get the response as json
    code: 'addressbook',        // Contract that we target
    scope: 'addressbook',       // Account that owns the data
    table: 'people',            // Table name (as defined by eosio::multi_index<your_table_name_here, ...>
    limit: 10,                  // Maximum number of rows that we want to get
    table_key: 'byage',         // The key of the seconday index (as defined by indexed_by<your_key_name_here, ...>
    index_position: 2,          // The index position to query. The primary key is considered index 1.
    key_type: 'i64',            // The type of the secondary index key.
                                // (Can be i64, i128, i256, float64, float128, ripemd160, sha256)
    lower_bound: lower_bound,   // By setting lower_bound, only values >= lower_bound will be returned
    reverse: false,             // Optional: Get reversed data
    show_payer: false,          // Optional: Show ram payer
  });
  generate_table(result);
}
```

Thanks!

ANY QUESTIONS?

You can email me at:
jeffrey.smith@block.one