PROJETO NIKE TOKEN

Sejam bem vindes ao **Projeto Nike Token**. Estamos muito ansiosos para a Data lançamento do nosso projeto. Abaixo segue uma breve explicação de como funcionário o Token e o Metaverso Nike.

Bom, primeiro, já informamos que nossa moeda é uma circular bem rara, por tanto, será de uma certa forma exclusividade ao Meteverso que estamos construindo. Nele, as pessoas poderão se divertir, ter sua coleção inédita e ainda fazer "call" em tempo real com os amigos com o Voice, além de tudo isso claro, vai poder ter suas recompensas e participar de Battle Royale PVP com o Play2Earn.

O metaverso:

Um mundo cheio de lugares com passagens gratuitas e pagas para cultivar o melhor no mundo virtual. Onde poderá encontrar pessoas, ir em lojas, restaurantes, casas, metro e muito mais. Tudo isso dentro do Explore Meta.

Call:

As pessoas além te poderem conversar por chat online, poderão também escolher conversar por voz, tendo uma interação mais ampla com o jogo, invés de digitar, pode ir falando e explorando mais o jogo. O modo Call também estará incluso no Battle Royale.

Battle Royale:

O modo inovador onde teremos o P2P misturado com Play2Earn. Entrando nesse modo, a plataforma irá sortear o outro jogador que irá disputar a partida com você. Assim que sorteado os adversários terão um tempo de 30segundos para se equiparem e começar a batalha. É um jogo bem simples mas que é promissor, simplesmente porque a pessoa irá se equipar com os Tênis disponibilizados Gratuitamente (nesse caso terão um Nível de poder fixado) ou comprados do nosso Marketplace (nesse caso, poderá ser comprado Tênis já em upgrade, ou Tênis para efetuar upgrade, vai de sua escolha). Assim que a partida iniciar, os mesmo irão disputar uma corrida (Atletismo). O vencedor além de acumular pontos de experiências, medalhas e pontos de partida ranqueada, ganhará também NIKECASH, um coin utilizado dentro do jogo para trocar pelo Token NIKE ou fazer upgrade em seu Avatar, como estilos de roupa,

visual, potencializar o nível do Tênis e demais. No caso do Token NIKE, o mesmo poderá ser transferido para as plataformas de Trader e vendido. Lembrando que o modo Battle Royale P2P Play2Earn é liberado para o modo gratuito e pago. No modo gratuito os jogadores terão apenas a quantia de NIKECASH diferenciada com o modo pago. Modo Gratuito terá por partida ganha 20 NIKECASH. Modo Pago terá por partida ganha 200 NIKECASH.

Marketplace:

O marketplace terá dentre muitos equipamento, mas do mais importante terá os Tênis, onde quanto maior o nível dele, maior sua chance de vitória. Nosso marketplace será também liberado para jogadores vendeder os Tênis após equipa-los com os níveis avançados. Assim liberando uma estratégia maior ainda de ganho, liberando assim o modo NFT dentro do nosso marketplace.

Com tudo, estamos muito felizes em fazer parte dessa nova era e estarmos integrando ao Metaverso. Esperamos que gostem.

Informações legais e adendos sobre Token NIKE

Segue abaixo as informações necessário sobre o Token Nike, para ser lançado dentro da plataforma Binance.

Totais de Token NIKE: 40.000 Unidades

Totais para circulação/volume trader: 30.000 Unidades

Totais para projetos de doações, ongs e demais: 5.000 Unidades

Totais para apoiadores do projeto: 5.000 Unidades

Valor inicial de cada NIKE: USD $10 (DEZ Dólares Americanos)

Volume de Margem Promissora Após lançamento do Jogo **NIKE RUN BATTLE:** 50.000% rentável

Logo Token NIKE

Nome Token NIKE na Binance:

NIKE

Propósito do Token NIKE:

Buscarmos capitalização no mercado para ajuda humanitária e gerar entretenimento surgindo da nova era Metaverso.

Código Programação Token NIKE:

```
pragma solidity 0.5.16;

interface IBEP20 {
  /**
   * @dev Returns the amount of tokens in existence.
   */
  function totalSupply() external view returns (uint256);

  /**
   * @dev Returns the token decimals.
   */
  function decimals() external view returns (uint8);

  /**
   * @dev Returns the token symbol.
   */
  function symbol() external view returns (string memory);

  /**
  * @dev Returns the token name.
  */
```

```
  function name() external view returns (string memory);

  /**
   * @dev Returns the bep token owner.
   */
  function getOwner() external view returns (address);

  /**
   * @dev Returns the amount of tokens owned by `account`.
   */
  function balanceOf(address account) external view returns (uint256);

  /**
   * @dev Moves `amount` tokens from the caller's account to `recipient`.
   *
   * Returns a boolean value indicating whether the operation succeeded.
   *
   * Emits a {Transfer} event.
   */
  function transfer(address recipient, uint256 amount) external returns (bool);

  /**
   * @dev Returns the remaining number of tokens that `spender` will be
   * allowed to spend on behalf of `owner` through {transferFrom}. This is
   * zero by default.
   *
   * This value changes when {approve} or {transferFrom} are called.
   */
  function allowance(address _owner, address spender) external view returns
(uint256);

  /**
   * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
   *
   * Returns a boolean value indicating whether the operation succeeded.
   *
   * IMPORTANT: Beware that changing an allowance with this method brings the
risk
   * that someone may use both the old and the new allowance by unfortunate
   * transaction ordering. One possible solution to mitigate this race
   * condition is to first reduce the spender's allowance to 0 and set the
   * desired value afterwards:
   * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
   *
   * Emits an {Approval} event.
```

```
   */
  function approve(address spender, uint256 amount) external returns (bool);

  /**
   * @dev Moves `amount` tokens from `sender` to `recipient` using the
   * allowance mechanism. `amount` is then deducted from the caller's
   * allowance.
   *
   * Returns a boolean value indicating whether the operation succeeded.
   *
   * Emits a {Transfer} event.
   */
  function transferFrom(address sender, address recipient, uint256 amount)
external returns (bool);

  /**
   * @dev Emitted when `value` tokens are moved from one account (`from`) to
   * another (`to`).
   *
   * Note that `value` may be zero.
   */
  event Transfer(address indexed from, address indexed to, uint256 value);

  /**
   * @dev Emitted when the allowance of a `spender` for an `owner` is set by
   * a call to {approve}. `value` is the new allowance.
   */
  event Approval(address indexed owner, address indexed spender, uint256 value);
}

/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
contract Context {
  // Empty internal constructor, to prevent people from mistakenly deploying
  // an instance of this contract, which should be used via inheritance.
  constructor () internal { }
```

```
  function _msgSender() internal view returns (address payable) {
    return msg.sender;
  }

  function _msgData() internal view returns (bytes memory) {
    this; // silence state mutability warning without generating bytecode - see
https://github.com/ethereum/solidity/issues/2691
    return msg.data;
  }
}

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
  /**
   * @dev Returns the addition of two unsigned integers, reverting on
   * overflow.
   *
   * Counterpart to Solidity's `+` operator.
   *
   * Requirements:
   * - Addition cannot overflow.
   */
  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    require(c >= a, "SafeMath: addition overflow");

    return c;
  }

  /**
   * @dev Returns the subtraction of two unsigned integers, reverting on
   * overflow (when the result is negative).
   *
```

```
   * Counterpart to Solidity's `-` operator.
   *
   * Requirements:
   * - Subtraction cannot overflow.
   */
  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    return sub(a, b, "SafeMath: subtraction overflow");
  }

  /**
   * @dev Returns the subtraction of two unsigned integers, reverting with custom
message on
   * overflow (when the result is negative).
   *
   * Counterpart to Solidity's `-` operator.
   *
   * Requirements:
   * - Subtraction cannot overflow.
   */
  function sub(uint256 a, uint256 b, string memory errorMessage) internal pure
returns (uint256) {
    require(b <= a, errorMessage);
    uint256 c = a - b;

    return c;
  }

  /**
   * @dev Returns the multiplication of two unsigned integers, reverting on
   * overflow.
   *
   * Counterpart to Solidity's `*` operator.
   *
   * Requirements:
   * - Multiplication cannot overflow.
   */
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    // Gas optimization: this is cheaper than requiring 'a' not being zero, but
the
    // benefit is lost if 'b' is also tested.
    // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
    if (a == 0) {
      return 0;
    }
```

```
    uint256 c = a * b;
    require(c / a == b, "SafeMath: multiplication overflow");

    return c;
  }

  /**
   * @dev Returns the integer division of two unsigned integers. Reverts on
   * division by zero. The result is rounded towards zero.
   *
   * Counterpart to Solidity's `/` operator. Note: this function uses a
   * `revert` opcode (which leaves remaining gas untouched) while Solidity
   * uses an invalid opcode to revert (consuming all remaining gas).
   *
   * Requirements:
   * - The divisor cannot be zero.
   */
  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    return div(a, b, "SafeMath: division by zero");
  }

  /**
   * @dev Returns the integer division of two unsigned integers. Reverts with
custom message on
   * division by zero. The result is rounded towards zero.
   *
   * Counterpart to Solidity's `/` operator. Note: this function uses a
   * `revert` opcode (which leaves remaining gas untouched) while Solidity
   * uses an invalid opcode to revert (consuming all remaining gas).
   *
   * Requirements:
   * - The divisor cannot be zero.
   */
  function div(uint256 a, uint256 b, string memory errorMessage) internal pure
returns (uint256) {
    // Solidity only automatically asserts when dividing by 0
    require(b > 0, errorMessage);
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold

    return c;
  }

  /**
```

```
   * @dev Returns the remainder of dividing two unsigned integers. (unsigned
integer modulo),
   * Reverts when dividing by zero.
   *
   * Counterpart to Solidity's `%` operator. This function uses a `revert`
   * opcode (which leaves remaining gas untouched) while Solidity uses an
   * invalid opcode to revert (consuming all remaining gas).
   *
   * Requirements:
   * - The divisor cannot be zero.
   */
  function mod(uint256 a, uint256 b) internal pure returns (uint256) {
    return mod(a, b, "SafeMath: modulo by zero");
  }

  /**
   * @dev Returns the remainder of dividing two unsigned integers. (unsigned
integer modulo),
   * Reverts with custom message when dividing by zero.
   *
   * Counterpart to Solidity's `%` operator. This function uses a `revert`
   * opcode (which leaves remaining gas untouched) while Solidity uses an
   * invalid opcode to revert (consuming all remaining gas).
   *
   * Requirements:
   * - The divisor cannot be zero.
   */
  function mod(uint256 a, uint256 b, string memory errorMessage) internal pure
returns (uint256) {
    require(b != 0, errorMessage);
    return a % b;
  }
}

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
```

```
 */
contract Ownable is Context {
  address private _owner;

  event OwnershipTransferred(address indexed previousOwner, address indexed
newOwner);

  /**
   * @dev Initializes the contract setting the deployer as the initial owner.
   */
  constructor () internal {
    address msgSender = _msgSender();
    _owner = msgSender;
    emit OwnershipTransferred(address(0), msgSender);
  }

  /**
   * @dev Returns the address of the current owner.
   */
  function owner() public view returns (address) {
    return _owner;
  }

  /**
   * @dev Throws if called by any account other than the owner.
   */
  modifier onlyOwner() {
    require(_owner == _msgSender(), "Ownable: caller is not the owner");
    _;
  }

  /**
   * @dev Leaves the contract without owner. It will not be possible to call
   * `onlyOwner` functions anymore. Can only be called by the current owner.
   *
   * NOTE: Renouncing ownership will leave the contract without an owner,
   * thereby removing any functionality that is only available to the owner.
   */
  function renounceOwnership() public onlyOwner {
    emit OwnershipTransferred(_owner, address(0));
    _owner = address(0);
  }

  /**
   * @dev Transfers ownership of the contract to a new account (`newOwner`).
```

```
   * Can only be called by the current owner.
   */
  function transferOwnership(address newOwner) public onlyOwner {
    _transferOwnership(newOwner);
  }

  /**
   * @dev Transfers ownership of the contract to a new account (`newOwner`).
   */
  function _transferOwnership(address newOwner) internal {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
  }
}

contract BEP20Token is Context, IBEP20, Ownable {
  using SafeMath for uint256;

  mapping (address => uint256) private _balances;

  mapping (address => mapping (address => uint256)) private _allowances;

  uint256 private _totalSupply;
  uint8 private _decimals;
  string private _symbol;
  string private _name;

  constructor() public {
    _name = "NIKE";
    _symbol = "NIKE";
    _decimals = 3;
    _totalSupply = 40000 * 10 ** 3;
    _balances[msg.sender] = _totalSupply;

    emit Transfer(address(0), msg.sender, _totalSupply);
  }

  /**
   * @dev Returns the bep token owner.
   */
  function getOwner() external view returns (address) {
    return owner();
  }
```

```
/**
 * @dev Returns the token decimals.
 */
function decimals() external view returns (uint8) {
  return _decimals;
}

/**
 * @dev Returns the token symbol.
 */
function symbol() external view returns (string memory) {
  return _symbol;
}

/**
* @dev Returns the token name.
*/
function name() external view returns (string memory) {
  return _name;
}

/**
 * @dev See {BEP20-totalSupply}.
 */
function totalSupply() external view returns (uint256) {
  return _totalSupply;
}

/**
 * @dev See {BEP20-balanceOf}.
 */
function balanceOf(address account) external view returns (uint256) {
  return _balances[account];
}

/**
 * @dev See {BEP20-transfer}.
 *
 * Requirements:
 *
 * - `recipient` cannot be the zero address.
 * - the caller must have a balance of at least `amount`.
 */
function transfer(address recipient, uint256 amount) external returns (bool) {
  _transfer(_msgSender(), recipient, amount);
```

```
    return true;
  }

  /**
   * @dev See {BEP20-allowance}.
   */
  function allowance(address owner, address spender) external view returns
(uint256) {
    return _allowances[owner][spender];
  }

  /**
   * @dev See {BEP20-approve}.
   *
   * Requirements:
   *
   * - `spender` cannot be the zero address.
   */
  function approve(address spender, uint256 amount) external returns (bool) {
    _approve(_msgSender(), spender, amount);
    return true;
  }

  /**
   * @dev See {BEP20-transferFrom}.
   *
   * Emits an {Approval} event indicating the updated allowance. This is not
   * required by the EIP. See the note at the beginning of {BEP20};
   *
   * Requirements:
   * - `sender` and `recipient` cannot be the zero address.
   * - `sender` must have a balance of at least `amount`.
   * - the caller must have allowance for `sender`'s tokens of at least
   * `amount`.
   */
  function transferFrom(address sender, address recipient, uint256 amount)
external returns (bool) {
    _transfer(sender, recipient, amount);
    _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount,
"BEP20: transfer amount exceeds allowance"));
    return true;
  }

  /**
   * @dev Atomically increases the allowance granted to `spender` by the caller.
```

```
   *
   * This is an alternative to {approve} that can be used as a mitigation for
   * problems described in {BEP20-approve}.
   *
   * Emits an {Approval} event indicating the updated allowance.
   *
   * Requirements:
   *
   * - `spender` cannot be the zero address.
   */
  function increaseAllowance(address spender, uint256 addedValue) public returns
(bool) {
    _approve(_msgSender(), spender,
_allowances[_msgSender()][spender].add(addedValue));
    return true;
  }

  /**
   * @dev Atomically decreases the allowance granted to `spender` by the caller.
   *
   * This is an alternative to {approve} that can be used as a mitigation for
   * problems described in {BEP20-approve}.
   *
   * Emits an {Approval} event indicating the updated allowance.
   *
   * Requirements:
   *
   * - `spender` cannot be the zero address.
   * - `spender` must have allowance for the caller of at least
   * `subtractedValue`.
   */
  function decreaseAllowance(address spender, uint256 subtractedValue) public
returns (bool) {
    _approve(_msgSender(), spender,
_allowances[_msgSender()][spender].sub(subtractedValue, "BEP20: decreased
allowance below zero"));
    return true;
  }

  /**
   * @dev Creates `amount` tokens and assigns them to `msg.sender`, increasing
   * the total supply.
   *
   * Requirements
   *
```

```
   * - `msg.sender` must be the token owner
   */
  function mint(uint256 amount) public onlyOwner returns (bool) {
    _mint(_msgSender(), amount);
    return true;
  }

  /**
   * @dev Moves tokens `amount` from `sender` to `recipient`.
   *
   * This is internal function is equivalent to {transfer}, and can be used to
   * e.g. implement automatic token fees, slashing mechanisms, etc.
   *
   * Emits a {Transfer} event.
   *
   * Requirements:
   *
   * - `sender` cannot be the zero address.
   * - `recipient` cannot be the zero address.
   * - `sender` must have a balance of at least `amount`.
   */
  function _transfer(address sender, address recipient, uint256 amount) internal
{
    require(sender != address(0), "BEP20: transfer from the zero address");
    require(recipient != address(0), "BEP20: transfer to the zero address");

    _balances[sender] = _balances[sender].sub(amount, "BEP20: transfer amount
exceeds balance");
    _balances[recipient] = _balances[recipient].add(amount);
    emit Transfer(sender, recipient, amount);
  }

  /** @dev Creates `amount` tokens and assigns them to `account`, increasing
   * the total supply.
   *
   * Emits a {Transfer} event with `from` set to the zero address.
   *
   * Requirements
   *
   * - `to` cannot be the zero address.
   */
  function _mint(address account, uint256 amount) internal {
    require(account != address(0), "BEP20: mint to the zero address");

    _totalSupply = _totalSupply.add(amount);
```

```
    _balances[account] = _balances[account].add(amount);
    emit Transfer(address(0), account, amount);
  }

  /**
   * @dev Destroys `amount` tokens from `account`, reducing the
   * total supply.
   *
   * Emits a {Transfer} event with `to` set to the zero address.
   *
   * Requirements
   *
   * - `account` cannot be the zero address.
   * - `account` must have at least `amount` tokens.
   */
  function _burn(address account, uint256 amount) internal {
    require(account != address(0), "BEP20: burn from the zero address");

    _balances[account] = _balances[account].sub(amount, "BEP20: burn amount
exceeds balance");
    _totalSupply = _totalSupply.sub(amount);
    emit Transfer(account, address(0), amount);
  }

  /**
   * @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.
   *
   * This is internal function is equivalent to `approve`, and can be used to
   * e.g. set automatic allowances for certain subsystems, etc.
   *
   * Emits an {Approval} event.
   *
   * Requirements:
   *
   * - `owner` cannot be the zero address.
   * - `spender` cannot be the zero address.
   */
  function _approve(address owner, address spender, uint256 amount) internal {
    require(owner != address(0), "BEP20: approve from the zero address");
    require(spender != address(0), "BEP20: approve to the zero address");

    _allowances[owner][spender] = amount;
    emit Approval(owner, spender, amount);
  }
```

```
  /**
   * @dev Destroys `amount` tokens from `account`.`amount` is then deducted
   * from the caller's allowance.
   *
   * See {_burn} and {_approve}.
   */
  function _burnFrom(address account, uint256 amount) internal {
    _burn(account, amount);
    _approve(account, _msgSender(),
_allowances[account][_msgSender()].sub(amount, "BEP20: burn amount exceeds
allowance"));
  }
}
```

Contract Address:

0xdb9203901cD767856017b0174A4753Da90945e1d

Criadores Projeto:

Samuel Alves da Silva Soares

samuelalvess1998@gmail.com