Welcome to LooksRare 👀💎

> LooksRare is the community-first NFT marketplace that actively rewards traders, collectors and creators for active participation.

🖼️👌 By NFT people, for NFT people

This is who we are.

Part of being community-first means that LooksRare rewards, empowers, and gives back to its users. **100% of fees** generated by the protocol are used as rewards to encourage active participation.

🍴 Not a fork

LooksRare's smart contracts are custom-built within a modular system that enables new features to be rolled out over time – without compromising security – thanks to standardized signatures that clearly define the execution scope.

This means we can offer new types of interactions, like these (rolling out soon):

- **Collection offer** - Make an offer that covers all NFTs in a selected collection
- **Trait offer** - Offer to buy any NFT with a specific trait in a collection e.g. BAYC w/ Gold Fur
- **Multi-cancelation** - Cancels multiple open orders in one transaction

🦖 Not a dinosaur

From utilizing some of the world's most powerful search infra, down to implementing on-chain standards like EIP-712 and ERC-2981, LooksRare's technical architecture (contracts, database, API, frontend, search) has been designed from the floor up for **scalability, speed and security** using the latest and greatest tech.

We've also made sure to retain compatibility for deploying on ETH scaling solutions when they're ready for the masses.

LFG 💦

Head to https://LooksRare.org to start your NFT journey.

Looking for more? Explore the docs, follow the LooksRare Twitter, or join the LooksRare community on Discord!

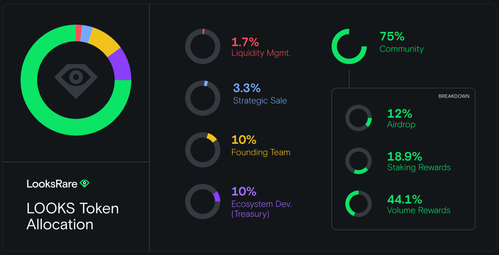
LOOKS Tokenomics

About LOOKS

LOOKS is the native, fungible token of the LooksRare protocol.

It's a transferable representation of the utility functions specified in LooksRare's protocol and code, where it's designed solely to be used as an interoperable utility token.

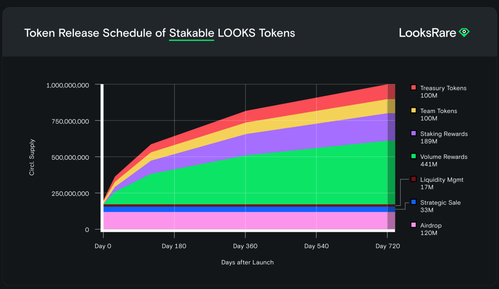
LOOKS allocation



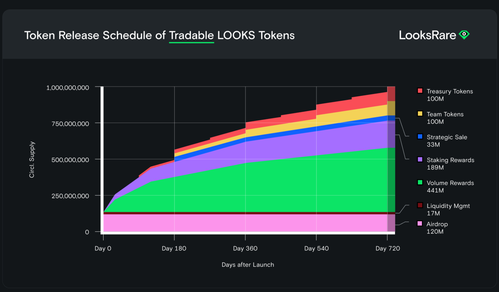
	Percentage of supply	Total LOOKS
Airdrop	12%	120,000,000
Strategic Sale	3.3%	33,083,003
Liquidity Management	1.7%	16,916,997
Volume Rewards	44.1%	441,000,000
Staking Rewards	18.9%	189,000,000
Founding Team	10%	100,000,000
Treasury	10%	100,000,000
Total Supply	100%	1,000,000,000

LOOKS emission schedule

Release schedule of usable LOOKS



Release schedule of tradable LOOKS



LOOKS governance (coming in the future)

Looking ahead, we hope that the community of LOOKS holders is comprised of a diverse field of developers, professionals and supporters of the project to develop the LooksRare platform (including without limitation experts in software development, blockchain technology, cryptography, artificial intelligence, law or finance), which will be able to share and exchange balanced views on the overall direction of the project.

To promote decentralised community governance for the protocol, LOOKS would allow holders to propose and vote on governance proposals to determine future features, upgrades and/or parameters of the LooksRare protocol, or provide feedback, with voting weight calculated in proportion to the amount of tokens held. The right to vote is restricted solely to voting on features of the LooksRare protocol; it does not entitle LOOKS holders to vote on the operation and management of the Company, its affiliates, or their assets or the disposition of such assets to token holders, or select the board of directors or similar bodies of these entities, or determine the development direction of these entities, nor does LOOKS constitute any equity interest in any of these entities or any collective investment scheme; the arrangement is not intended to be any form of joint venture or partnership.

After governance launch there will be no individual or corporate entity or other active promoter, sponsor, or group or affiliated party that maintains sole control over the LooksRare protocol. The LooksRare protocol would gradually introduce mechanisms (with certain voting and technical safeguards) enabling any party unaffiliated with the LooksRare team to modify/upgrade the functionality of the smart contracts or protocol, or to participate in other essential task and responsibilities. In this way, future changes to the system or network would be driven by active participants of the protocols themselves (instead of the original LooksRare team) based on community voting.

Further information on the LOOKS token

Important to note is that LOOKS doesn't represent any shareholding, participation, right, title, or interest in the protocol, the Distributor, their respective affiliates, or any other company, enterprise or undertaking.

LOOKS doesn't entitle token holders to any promise of fees, dividends, revenue, profits or investment returns. It isn't intended to constitute a security in any relevant jurisdiction.

LOOKS can only be used on the LooksRare platform, and ownership carries no rights, express or implied, other than the right to use LOOKS as a means to enable usage of and interaction within the LooksRare protocol. The secondary market pricing of LOOKS is not dependent on the LooksRare team's efforts, and there is no token functionality or scheme designed to control or manipulate such secondary pricing.

Critically, LOOKS is designed to encourage ecosystem participation — so users need to participate to enjoy rewards in the form of LOOKS and other digital assets.

Update to Team / Treasury Fund Management

`Update Feb 4, 2022`

During the initial setup of the Team and Treasury tokens, there was an oversight in the amount of Pre-minted LOOKS tokens sent to the respective wallets.

- The Team vesting contract (0x332580e0da5b5072ff5d5b73a494a65bb99744d8) was meant to receive 20M LOOKS tokens, but instead received 10M LOOKS:
  https://etherscan.io/tx/0xf5a003b5a82afd39d2f8d1466b7b670e4d821adce4502de62de2ed36360ef367

- The Treasury vesting contract (0x6757f73cddf4c16171281ff869e620c6ce30e12b) was meant to receive 10M LOOKS tokens, but instead received 20M LOOKS:
  https://etherscan.io/tx/0x5f90fa71e3b631d497739aede84b48d3a51c5a918c342b4a4108f1bc3f00024e2

While **this does not affect the current vesting schedule for team and treasury tokens**, it does mean we will need to make a final adjustment at the end of the 721 day LOOKS emission period, as the Team wallet address will end up with 90M LOOKS and the Treasury wallet will end up with 110M LOOKS, when it should be 100M for each.

An immediate impact is that the WETH fee sharing amount that is distributed to the Team and Treasury address each day is affected. i.e. The Treasury will receive more WETH than intended and the Team will receive less WETH than intended. To resolve this, we will continue to make periodic adjustments to the Team and Treasury WETH balances by transferring between the two, so that each address receives the WETH that it should have received if the oversight mentioned above had not occurred. An example of this can be seen in this transaction, where we transferred **3,850** WETH from the Treasury multi-sig wallet to the Team multi-sig wallet:
https://etherscan.io/tx/0xdee1d14528431ce44d7d70fe758f0b1fb3fa05025ebc1ca36f82592c5b0d6d5

- Team vesting contract 0x332580e0da5b5072ff5d5b73a494a65bb99744d8
- Team multisig safe 0xF6aF7dA33f86F138Cc7DBE3a970De19O5Da5d1E8
- Treasury vesting contract 0x6757f73cddf4c16171281ff869e620c6ce30e12b
- Treasury multisig safe 0xC8C57e4C73c71f72cA0a7eO43E5D2D144F98ef13

*Last updated on **Aug 28, 2023***

Audits

Trail of Bits

LooksRare exchange

Coverage:

- LooksRare Exchange
- Manager Contracts (CurrencyManager, ExecutionManager, and RoyaltyFeeManager)
- Execution Strategies
- Exchange Libraries
- Exchange—Royalty Fee Helper Contracts
- Exchange—Transfer Manager Contracts
- Token Rewards—TokenDistributor
- Token Rewards—FeeSharingSystem
- Token Rewards—LooksRareToken

Link to audit

Addressing the findings

PeckShield

LooksRare reward protocols

Contracts in scope:

1. FeeSharingSystem
2. TokenDistributor
3. LooksRareToken

Link to audit

Auto-compounding protocol

Contracts in scope:

1. AggregatorFeeSharing

Link to audit

Spearbit

Audit of V2 contracts

Link to audit

Last updated on ***Aug 28, 2023***

Trading Rewards

> **NOTE**
> As of September 2023, Trading Rewards have been discontinued in favor of a different approach to rewards. The information below is preserved for archival purposes.

Trade any NFT on LooksRare.

Earn LOOKS tokens.

Simple.

Trading Rewards are a core part of LooksRare's token economy as we seek to become the world's most liquid marketplace for NFTs.

> `UPDATE` 🎉 **On October 27, 2022, LooksRare released an update to protocol fees, royalties, and trading rewards. Sellers will now receive 95% of trading rewards. For more information please check out our blog post!** 🎉

❓ How do Trading Rewards work?

Users who trade NFTs on LooksRare earn Trading Rewards in LOOKS — LooksRare's platform token.

Both **buyers** and **sellers** receive Trading Rewards, based on their trading volume (though this doesn't include private sales).

Trading Rewards are **calculated daily** and are assigned to users **two hours after the end of each day**. The entire schedule of Trading Rewards emissions will span **4,686,250 Ethereum blocks** (or approximately **721 days**, assuming 6,500 blocks per day), at which point all LOOKS token emissions will come to an end as the ecosystem becomes fully self-sustaining.

> Please also note that only trading volume resulting from orders created by the LooksRare Marketplace is included in the daily Trading Rewards calculation. Trades matched from custom orders (e.g., created by private client libraries or other unofficial marketplaces) are not treated as LooksRare Marketplace trades — so they won't receive Trading Rewards.

> Trading via an NFT marketplace aggregator? Get in touch with the provider to find out how Trading Rewards are handled.

🥧 How many LOOKS tokens are allocated for Trading Rewards?

Trading Rewards are earned over **four phases** with differing reward rates, based on a schedule of **6,500 Ethereum blocks per day.**

Phase	Length	LOOKS per Day for Trading Rewards
A	195,000 blocks (30 days)	2,866,500.00 LOOKS
B	585,000 blocks (90 days)	1,361,587.50 LOOKS
C	1,560,000 blocks (240 days)	537,468.75 LOOKS
D	2,346,250 blocks (361 days)	286,650.00 LOOKS

> **LISTING REWARDS UPDATES**
> Of the amount allocated to Trading Rewards daily, 50,000 LOOKS have been allocated to Listing Rewards.
>
> As of March 15th, 2023 the Listing Rewards program has been discontinued, and the 50,000 LOOKS in allocated rewards per period will be sent to a dedicated Gnosis Safe. Full details here.

🧮 How are Trading Rewards calculated?

Trading Rewards are calculated daily, based on each user's trading volume on LooksRare. They're derived as a percentage of the total platform trading volume, excluding private sales.

$$\text{User A's trading rewards on Day 1} = \frac{a}{b} \times c$$

Where:

- a = User A's trading volume during Day 1
- b = Total platform trading volume during Day 1
- c = LOOKS rewards per day

Example User A trades 10 ETH worth of NFTs on Day 1, a period which saw 10,000 ETH in total volume across the platform.

Based on the above, on Day 1, User A would receive:

$$\frac{10}{10,000} \times 2,866,500 = 2,866.5 \text{ LOOKS}$$

How can I claim Trading Rewards?

You can claim your rewards any time between **09:00 AM and 11:00 AM (UTC)** via the Rewards page.

Claiming is suspended daily for a period of two hours between **09:00 AM and 11:00 AM (UTC)** while that day's Trading Rewards are calculated. Unclaimed rewards can be claimed at a later date — so you don't need to rush to claim them every day.

✅ What collections are eligible for Trading Rewards?

All of them! The only exceptions are pieces that are traded in private sales.

Previously, Trading Rewards were limited to only collections with 1,000 ETH trading volume on LooksRare or higher, but this limitation has now been removed.

🧑‍🌾 What happens if there's an abundance of volume from token farming?

The Trading Rewards system has been designed and iterated upon to encourage genuine trading of NFTs, and heavily dissuade users from trading for the sole purpose of farming LOOKS rewards:

- Each trade on LooksRare incurs a platform fee of 0.5%
- Available rewards are fixed and distributed based on traders' contribution to the *total* trading volume — meaning there's no guarantee on the amount of rewards that a farmer can collect on any given day

For instance, if multiple individuals try to farm LOOKS, daily rewards will be split between them. Once trading fees are factored in, they'll likely operate at a loss.

Further, the aforementioned update to make *all* collections eligible for trading rewards further reduces the efficacy of reward farming, by increasing the overall pool of trading reward recipients.

🎟️ Will I receive Trading Rewards if I buy an NFT using an aggregator like Genie or OpenSea Pro?

Yes! You can collect rewards from those platforms directly, instead of via LooksRare's Trading Rewards page.

When you use an aggregator, it's the aggregator's contract that interacts with LooksRare's Marketplace contract. As a result, the volume generated is actually attributed to their contract — meaning they need to claim the Trading Rewards.

Last updated on ***Oct 1, 2023***

Protocol Rewards

➕ Commit LOOKS, earn LOOKS

When you commit LOOKS, you earn additional LOOKS — on top of WETH-denominated rewards generated from trades. Prefer to leave your committed LOOKS for the long-term? You can **auto-compound** them, too.

🥧 How many LOOKS tokens are allocated for Protocol Rewards?

Protocol Rewards will be earned over **four phases** with differing reward rates, based on a schedule of **6,500 Ethereum blocks per day**.

Phase	Length	LOOKS per Block	LOOKS per Day
A	195,000 blocks (30 days)	189.00 LOOKS	1,228,500.00
B	585,000 blocks (90 days)	89.775 LOOKS	583,537.50
C	1,560,000 blocks (240 days)	35.4375 LOOKS	230,343.75
D	2,346,250 blocks (361 days)	18.90 LOOKS	122,850.00

🧮 How are Protocol Rewards calculated?

The following formula is applied to calculate the Protocol Rewards (per block) available to a given user.

$$\frac{\text{LOOKS committed by user}}{\text{Total amount of committed LOOKS}} \times \text{LOOKS per block}$$

Example

- User A commits 10,000 LOOKS during Phase A
- The total amount of committed LOOKS is now 1,000,000 LOOKS

Based on the above, User A would receive:

$$\frac{10,000}{1,000,000} \times 189 = 1.89 \text{ LOOKS per block}$$

🔁 What does auto-compounding mean?

On many platforms, pending rewards must be manually committed by users if they want to increased their total share of committed tokens. However, this comes with gas fees and opportunity cost.

LooksRare enables **auto-compounding** for users who commit LOOKS. Meaning that LOOKS rewards received from committing are **automatically committed** (or *compounded*) to increase the total share — without the usual fees.

When users remove their tokens, compounded rewards and the original amount are withdrawn to the user's wallet.

!! But wait, there's more!

We mentioned the WETH rewards generated by trades earlier.

On every sale (except private ones), LooksRare collects a fixed fee of **0.5%**. All fees are consolidated every **6,500 blocks** and distributed to LOOKS stakers over the next **6,500 blocks**.

Once Phase D ends and LOOKS is no longer emitted, these ETH-denominated rewards will make up the entirety of the Protocol Rewards.

Active vs passive committing

So, the rewards for each day are calculated at the end of the previous period. Then, they're split between **active** and **passive** committers.

🏃 What's the difference?

Active committers earn additional LOOKS when they commit LOOKS. *Passive* committers do not.

But don't worry, you're probably an active one. Active committers are those whose tokens are fully unlocked. Passive committers, on the other hand, have their tokens locked for trading, but unlocked for committing. Usually, these are tokens coming from the Team, Treasury and Strategic Sale allocations (read more about their unlock schedules here).

The formulas

At the start of every 6,500 block period, a portion of the WETH fees collected from the previous 6,500 blocks are sent to the passive committing address. The amount is calculated as:

$$\frac{A}{B} \times C$$

Where:

- A = The total amount of LOOKS passively committed at the start of the period
- B = The total amount of LOOKS committed at the start of the period (both passive and active)
- C = The WETH fees accumulated from the previous 6,500 blocks

The remaining WETH is then sent to the active committing pool.

Fee sharing (for each 6,500-block period) is calculated for active committers as follows:

$$\frac{\text{Total WETH collected as fees in the prior 6500 block period}}{6500}$$

Each user's amount of staked LOOKS at each block is then compared against the total amount of LOOKS staked at each block, with this being done at every block within the 6,500 block period to find the total amount of WETH rewards received.

Example time! Let's assume that LooksRare has 10,000 WETH of non-private sale volume in a period of 6,500 blocks. It would collect 50 WETH in fees, based on its 0.5% rate.

At Block #6,500, User A commits 10,000 LOOKS, preparing to start earning WETH at Block #6,501.

At Block #6,501:

- the amount of passive tokens committed is 10% of the total amount of tokens staked — meaning that 5 WETH is sent to the passive committing address

- The remaining 45 WETH is put into the active committing pool, with 0.00692307692 WETH to be earned per block for the next 6,500 blocks (45/6,500)

- The total amount of LOOKS committed by **all** users is 1,000,000 LOOKS

As a result, User A will be allocated $\frac{10,000}{1,000,000} \times 0.00692307692 = 0.00006923076$ WETH at block #6501.

At Block #6,502:

- Assuming User A's stake doesn't change, but an additional 500,000 LOOKS are committed, then User A will be allocated $\frac{10,000}{1,500,000} \times 0.00692307692 = 0.00004615384$

This calculation will then continue for another 6,498 blocks (as the example covers a 6,500-block period), with User A earning the amount specified in each example per block.

Meanwhile, the WETH fees generate during this period will be used as rewards to distribute in the next phase.

🤔 Frequently asked questions

How often can I claim my rewards?

As often as you want! However, bear in mind that you'll need to pay an Ethereum gas fee for each claim.

What happens if I forget to claim my WETH rewards for today?

Nothing at all. They'll be there whenever you want to collect them. You committed LOOKS will continue to accumulate rewards, so you can leave it until gas fees are low or until it becomes worthwhile to claim.

Why don't private sales count towards Protocol Rewards?

Unlike other marketplaces, LooksRare doesn't take a cut on private sales — so the volume generated by these doesn't contribute to the pile allocated to LOOKS stakers.

*Last updated on **Aug 28, 2023***

Gems

While a Season's running, your NFT listings can earn **Gems**.

At the end of the Season, you'll cash these Gems in for your share of the **LOOKS Airdrop Pot.**

⏱️ How Long Does a Season Last?

It depends. The inaugural Season One will run for **one month**, at which point a **5,000,000 LOOKS** Airdrop Pot will be distributed to Gems holders.

💎 What's a Gem Worth?

That depends, too. There isn't a fixed number of Gems available, so your LOOKS reward is calculated at the end of the Season with the following formula:

$$\frac{\text{Your Gems}}{\text{Total Number of Gems}} \times \text{Airdrop Pot Amount}$$

📈 How Do I Get a Gem Boost?

The higher on the Gem Leaderboard you are, the higher your Gem Boost — a multiplier that increases the amount of Gems you earn on every item:

- **Ranks 1-10:** 2.5x boost
- **Ranks 11-25:** 2x boost
- **Ranks 26-50:** 1.5x boost
- **Ranks 51-100:** 1.2x boost

✖️ What's the Gems Multiplier?

How you list counts.

We get it — the more marketplaces you put your item up for sale on, the better your chances of selling your piece.

But we're the jealous type. So, we reward lower listings on LooksRare — and penalize lower ones on other marketplaces. Suppose you're listing BAYC #10,001 on several marketplaces:

- If the listing is lowest on LooksRare, you'll get a **5x multiplier**
- If it's equal to its price on other marketplaces, you'll get **no multiplier**
- But if it's higher on LooksRare than it is on other marketplaces, you'll get penalized with an **0.5x multiplier**

❓ Will I get a 5x Multiplier if I only list my item on LooksRare?

Yes! Provided, of course, that the price isn't set too far above the floor and that the collection is eligible.

❓ Why list on LooksRare?

Because the fees are much lower. We charge a flat 0.5% on transactions — but with your LOOKS rebate, your effective fees are basically zero.

✅ What Collections Are Eligible for Rewards?

Eligible collections are clearly labeled. But, in case you were wondering, here's the criteria we use:

- Collections must be tradeable on LooksRare
- Collections must be verified on LooksRare
- Collections can't exceed 100,000 items
- Collections can't have 'sub-collections' within the same contract
- Collections must adhere to the ERC-721 format

⚠️ Don't list too high!

To reduce the opportunities for abuse, we can't share the exact figures here. However, **listings that are too high above the floor price may not earn Gems.** The system rewards the provision of liquidity, so there needs to be a reasonable chance that items sell.

*Last updated on **Aug 28, 2023***

Backend Engineer (GraphQL Focus)

General information

- **Salary Range**: Competitive. Negotiable
- **Position**: Full-time
- **Location**: Remote
- **Timezone**: Any

Your responsibilities

- Work with the team to create new offerings for the GraphQL API (Used internally)
- Collaborate cross-functional partners to ensure the highest quality of deliverables.
- Enthusiastically contribute and innovate with a small team to help build ground-breaking product features.
- Constantly planning ways to improve our API performance and offerings.

Requirements

- Fluent in english
- 3 years of experience using JavaScript & NodeJS
- 1 year of experience using TypeScript
- Strong understanding experience working with of GraphQL
- Strong experience working with PostgreSQL databases

How to apply

Send a copy of your resume (*required*), along with any additional information about yourself which may help us to make our decision to **hiring@looksrare.org**

NOTES FOR APPLICANTS

- LooksRare is a decentralized platform and thus also employs the team in a decentralized way, contributing as individuals towards the protocol
- Salary is paid in crypto (LOOKS or WETH).
- Salary proposal may include a vested allocation of LOOKS.

*Last updated on **Aug 28, 2023***

Backend Engineer (Indexer)

General information

- **Salary Range**: Competitive. Negotiable
- **Position**: Full-time
- **Location**: Remote
- **Timezone**: Any

Your responsibilities

- Development of new features on LooksRare
- Collaborate with cross-functional partners to ensure the highest quality of deliverables.
- Constantly strategise and implement ways to improve the LooksRare indexer
- Enthusiastically contribute and innovate with a small team to help build ground-breaking product features.

Requirements

- Fluent in English
- 3 years of experience using Java/Spring, Multithreading
- Knowledge of RabbitMQ/Kafka/Redis/Postgres
- AWS, Docker
- Experience building RESTful services
- Experience with integration with third parties

How to apply

Send a copy of your resume (*required*), along with any additional information about yourself which may help us to make our decision to **hiring@looksrare.org**

NOTES FOR APPLICANTS

- LooksRare is a decentralized platform and thus also employs the team in a decentralized way, contributing as individuals towards the protocol
- Salary is paid in crypto (LOOKS or WETH).
- Salary proposal may include a vested allocation of LOOKS.

::::

*Last updated on **Sep 22, 2023***

Backend Engineer (Rest API Focus)

General information

- **Salary Range**: Competitive. Negotiable
- **Position**: Full-time
- **Location**: Remote
- **Timezone**: Any

Your responsibilities

- Work with the team to create new offerings for the Public Rest API
- Collaborate cross-functional partners to ensure the highest quality of deliverables.
- Enthusiastically contribute and innovate with a small team to help build ground-breaking product features.
- Constantly planning ways to improve our API performance and offerings.
- Faciliatate requirements from external partners to improve our API offerings.

Requirements

- Fluent in english
- 3 years of experience using JavaScript & NodeJS
- 1 year of experience using TypeScript
- Strong understanding experience working with of GraphQL
- Strong experience working with PostgreSQL databases

How to apply

Send a copy of your resume (*required*), along with any additional information about yourself which may help us to make our decision to **hiring@looksrare.org**

NOTES FOR APPLICANTS

- LooksRare is a decentralized platform and thus also employs the team in a decentralized way, contributing as individuals towards the protocol
- Salary is paid in crypto (LOOKS or WETH).
- Salary proposal may include a vested allocation of LOOKS.

*Last updated on **Aug 28, 2023***

Blockchain Engineer (Solidity)

General information

- **Salary Range**: Competitive. Negotiable
- **Position**: Full-time
- **Location**: Remote
- **Timezone**: Any

Your responsibilities

- Conceptualize, create, develop, and release highly efficient production-grade smart contracts for EVM-based networks that match business requirements
- Integrate risk systems for deployed smart contracts to debug issues that may arise on mainnet environments
- Support cross-functional teams to review specs and provide insights
- Write unit & integration tests for smart contracts and other parts of the technical stack
- Manage subgraphs and work with backend/frontend teams to integrate smart contracts into the entire infrastructure

Requirements

- 3+ years of experience in software development or similar
- 1+ year experience in developing decentralized applications ("dApps") with experience deployment complex smart contracts on blockchains
- Deep understanding of the Ethereum Virtual Machine and at least one high-level language development language (e.g., Solidity)
- Good knowledge of EIP and ERC Ethereum standards
- Knowledge of TypeScript/JavaScript

Good to have

- Familiarity with Assembly languages and their use for low-level optimizations
- Experience with Hardhat & Foundry frameworks and working knowledge of Ethers.js
- Understanding of L2 solutions, including non-EVM frameworks (e.g., Starknet)

How to apply

Send a copy of your resume (*required*), along with any additional information about yourself which may help us to make our decision to **hiring@looksrare.org**

NOTES FOR APPLICANTS

- LooksRare is a decentralized platform and thus also employs the team in a decentralized way, contributing as individuals towards the protocol
- Salary is paid in crypto (LOOKS or WETH).
- Salary proposal may include a vested allocation of LOOKS.

*Last updated on **Aug 28, 2023***

Data Engineer

General information

- **Salary Range**: Competitive. Negotiable
- **Position**: Full-time
- **Location**: Remote
- **Timezone**: Any

Your responsibilities

- Build out data infrastructure that will help to serve as a foundation for dashboards, data warehousing, machine learning and ab testing.
- Work with cross functional teams to ensure that all requirements are understood and being satisfied.
- Make use of a mix of on and offchain data to serve in making business decisions and marketing.

Requirements

- 3+ years in data engineering, data science or similar fields.
- Work independently and proactively
- Able to write well tested and high quality code

How to apply

Send a copy of your resume (*required*), along with any additional information about yourself which may help us to make our decision to **hiring@looksrare.org**

NOTES FOR APPLICANTS

- LooksRare is a decentralized platform and thus also employs the team in a decentralized way, contributing as individuals towards the protocol
- Salary is paid in crypto (LOOKS or WETH).
- Salary proposal may include a vested allocation of LOOKS.

*Last updated on **Aug 28, 2023***

Database Administrator (DBA)

General information

- **Salary Range**: Competitive. Negotiable
- **Position**: Full-time
- **Location**: Remote
- **Timezone**: Any

Your responsibilities

- Work closely with the tech and product teams in order to maintain the DB and create new models according to the team needs
- Constantly looking for new ways to optimise and improve database performance
- Maintain a high quality standard across the codebase

Requirements

- Fluent in english
- Experience working on high volume databases
- Experience working on high availability systems
- Relevant experiences with Postgres and SQL

Nice to have

- Understanding of web3, nfts, and how a blockchain works
- Some knowledge in Javascript

How to apply

Send a copy of your resume (*required*), along with any additional information about yourself which may help us to make our decision to **hiring@looksrare.org**

NOTES FOR APPLICANTS

- LooksRare is a decentralized platform and thus also employs the team in a decentralized way, contributing as individuals towards the protocol
- Salary is paid in crypto (LOOKS or WETH).
- Salary proposal may include a vested allocation of LOOKS.

*Last updated on **Aug 28, 2023***

Frontend Engineer

General information

- **Salary Range**: Competitive. Negotiable
- **Position**: Full-time
- **Location**: Remote
- **Timezone**: Any

Your responsibilities

- Development of new features on LooksRare
- Collaborate with the design team and other cross-functional partners to ensure the highest quality of deliverables.
- Accurately and efficiently translate designs from Figma onto the LooksRare site.
- Enthusiastically contribute and innovate with a small team to help build ground-breaking product features.

Requirements

- Fluent in english
- A good understanding of blockchain and specifically the Ethereum blockchain
- 3+ years of experience using JavaScript/Typescript
- Extensive knowledge of React JS
- A good understanding of NextJS and Ethers.js
- Able to write structured and clean code
- Experience with modern frontend build pipelines.
- Experience with GraphQL

How to apply

Send a copy of your resume (*required*), along with any additional information about yourself which may help us to make our decision to **hiring@looksrare.org**

NOTES FOR APPLICANTS

- LooksRare is a decentralized platform and thus also employs the team in a decentralized way, contributing as individuals towards the protocol
- Salary is paid in crypto (LOOKS or WETH).
- Salary proposal may include a vested allocation of LOOKS.

*Last updated on **Aug 28, 2023***

Growth Marketing Specialist

General information

- **Salary Range**: Competitive. Negotiable
- **Position**: Full-time
- **Location**: Remote
- **Timezone**: Any

Your responsibilities

- Lead the conceptualization and implementation of growth marketing strategy and plans, in line with project goals
- Create, manage and experiment with new user acquisition campaigns
- Develop and execute online and offline campaigns, including education, gamification, digital marketing, brand partnerships, etc
- Work with cross-functional teams to establish growth strategies to support project goals
- Oversee SEO/SEM and ensure respective strategies are effectively implemented.
- Identify, develop, and optimize growth channels to drive active user growth
- Analyze marketing data and user feedback (campaign results, conversion rates, traffic, etc.) to help shape future marketing strategies
- Work with and provide support to community admins in order to better understand community/user needs and create a positive feedback loops with cross functional teams
- Keep up to date with market developments and competitor strategies.

Requirements

- 5+ years experience in growth marketing
- Solid project management skills: ability to lead and collaborate with cross-functional partners, prioritize high impact activities, and keep complex projects moving forward
- Strong growth track record, with samples and results of past campaigns
- Excellent analytical skills and leverage data, metrics, analytics, and consumer behavior trends to generate creative ideas
- Good knowledge of various aspects of marketing, including brand marketing, PR, content management, digital marketing & performance marketing
- Highly goal-oriented individual
- Strong knowledge of the NFT ecosystem and crypto as a whole

How to apply

Send a copy of your resume (*required*), along with any additional information about yourself which may help us to make our decision to **hiring@looksrare.org** – as a growth specialist, you should already know we want to see statistics!

NOTES FOR APPLICANTS

- LooksRare is a decentralized platform and thus also employs the team in a decentralized way, contributing as individuals towards the protocol
- Salary is paid in crypto (LOOKS or WETH).
- Salary proposal may include a vested allocation of LOOKS.

*Last updated on **Aug 28, 2023***

Lead QA Engineer/SDET

General information

- **Salary Range**: Competitive. Negotiable
- **Position**: Full-time
- **Location**: Remote
- **Timezone**: Any

Your responsibilities

- Understand the product and system from start to finish
- Grow the LooksRare QA/SDET team
- Help to set goals/targets for the QA/SDET team
- Create a full testing setup using a local blockchain (hardhat)
- Write appropriate end-to-end tests to protect the product from regression bugs
- Consistently plan to upgrade and improve existing testing infrastructure

Requirements

- Fluent in english
- Able to work autonomously without much guidance
- A good understanding of blockchain and specifically the Ethereum blockchain
- Highly skilled in software development
- Strong testing knowledge and experience
- 3+ years of experience in QA/SDET roles
- Thorough understanding of NFT marketplaces such as LooksRare and OpenSea
- Able to write structured and clean code
- Experience with modern frontend build pipelines.
- Experience with GraphQL

How to apply

Send a copy of your resume (*required*), along with any additional information about yourself which may help us to make our decision to **hiring@looksrare.org**

NOTES FOR APPLICANTS

- LooksRare is a decentralized platform and thus also employs the team in a decentralized way, contributing as individuals towards the protocol
- Salary is paid in crypto (LOOKS or WETH).
- Salary proposal may include a vested allocation of LOOKS.

*Last updated on **Aug 28, 2023***

Search Engine Marketing Specialist

General information

- **Salary Range**: Competitive. Negotiable
- **Position**: Full-time
- **Location**: Remote
- **Timezone**: Any

Your responsibilities

- Lead LooksRare's SEO/SEM efforts from the ground up
- Provide technical SEO guidance to the product team
- Assist a content lead in planning an seo-focused content strategy
- Actively monitor and assess competitor search rankings
- Create and execute a data-driven strategy to outrank competitors
- Set up and manage organic traffic dashboards inclusive on LooksRare and competitors
- Help plan and initiate link-building campaigns to promote site growth
- Monitor the technical SEO health of LooksRare

Requirements

- Fluent in english
- 3+ years of experience as a Search Engine Marketing Specialist or similar role
- Experience with planning and implementing a successful SEO strategy with proven success
- Proven SEM experience and success managing PPC campaigns across Google etc
- Proficiency with website analytics tools such as Google Analytics
- Well-versed in performance marketing, conversion, and online customer acquisition
- Familiarity with A/B testing and other testing methods
- Ability to analyze data and provide evidence-based recommendations

How to apply

Send a copy of your resume (*required*), along with any additional information about yourself which may help us to make our decision to **hiring@looksrare.org**

NOTES FOR APPLICANTS

- LooksRare is a decentralized platform and thus also employs the team in a decentralized way, contributing as individuals towards the protocol
- Salary is paid in crypto (LOOKS or WETH).
- Salary proposal may include a vested allocation of LOOKS.

*Last updated on **Aug 28, 2023***

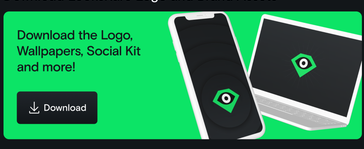
Brand, Logos & Usage

> **READ ME**
> Please read the usage guidelines fully before you use our logo and brand assets.

These guidelines cover our brand assets, including but not limited to:

- The names "LooksRare" and "LOOKS Token"
- The LooksRare logo (text and icon)
- The LOOKS diamond-eye icon

Download LooksRare Logo and Brand Assets



This ZIP file contains:

- LooksRare logos
- Device wallpapers / brand images
- A community social media kit so you can rep the project on Discord, Twitter, Instagram... and other NFT marketplaces!

💾 Download the Assets

Intro

LooksRare is pretty open minded and flexible with our icon, but not our name.

We encourage creators to make derivative artworks based on the icon, and tbh we wanna see the community get real weird with it. If you're selling derivative works, please give attribution and make sure you follow the guidelines here carefully so we don't have a disagreement. If it's dope, @ us on social so we don't miss it!

BUT, our logo and name are important, copyrighted parts of our brand, and we must protect them. So let's go over a few ground rules.

What Can I Do?

✅👍 You Can:

- ✅ Recreate, remix, create derivative artworks featuring the diamond icon, with attribution.
- ✅ Buy and sell said derivatives, if you give attribution.

🚫👎 You Can't:

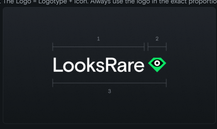
- 🚫 Use our Brand Assets (logo, icon, name,) or any derivative works in any product/brand name, brand logo, etc. that can conflict or compete with LooksRare. For example:
  - Don't make an NFT marketplace called "LooksExpensive"
  - Don't make a digital art brand or product with a similarly-shaped diamond logo with an eye, even if it's colored yellow.
  - Don't make a product called "LooksRare Tools" or similar that may be misinterpreted as having an official relationship to LooksRare.
- 🚫 Use or remix any of our Brand Assets to imply an official relationship, partnership, or other endorsement where there is none.
- 🚫 Use the name "LooksRare" in full in social media channels in a way that implies any official relationship to us. If you make a community group, Twitter account etc., make it clear that it's not officially tied to LooksRare.
- 🚫 Falsely claim or suggest any sort of official relationship, partnership, or other endorsement in any way.

If in doubt, reach out to design@looksrare.org

Logo Guidelines

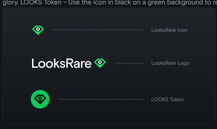
Logo Anatomy

1. Logotype. Set in Basis Grotesque using Camel Case for the L and R. There is no spaces in the LooksRare name.
2. Icon. The eye watches you as you paperhand your JPGs.
3. The Logo = Logotype + Icon. Always use the logo in the exact proportion provided.



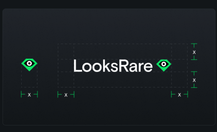
Variations

Icon - Can be used on its own to represent the LooksRare platform or brand. Logo - This is the LooksRare logo in all its glory. LOOKS Token - Use the icon in black on a green background to represent the token.



Logo Clear Space

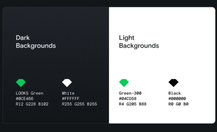
Give the logo breathing room. Keep a space around the logo that's equal to at least the width of the diamond icon (x) on all sides.



The downloadable assets have the correct proportions already, but anyway; the clear space around the diamond icon must be at least 1/4 the width of the icon (y/4).

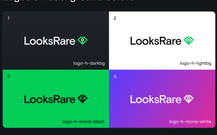
Logo Colors

Green is LooksRare's brand color. It's vibrant, saturated, slightly acidic, and generally stands out because it's great. We like the green.



When you're using the LooksRare logos, ***never*** adapt or recolor the files manually; just use the correct version provided for your use case. The RGB values and hex codes are shown just for your reference.

Logos on background colors



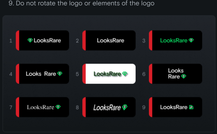
Make sure the logo and icon are always fully visible against the background. 1 & 2 - Use the colored logos on monochrome backgrounds 3 & 4 - Use the monochrome logos on colored backgrounds

We use a different shade of green on different backgrounds, and a different version of the icon.

Logo Restrictions

Common sense stuff really.

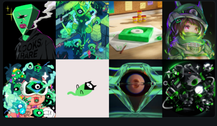
1. Do not move the icon to the left
2. Do not remove the icon
3. Do not change the text color
4. Do not add a space between "Looks" and "Rare"
5. Do not add drop shadows or distort the text
6. Do not try to create a stacked version of the logo (Use the Icon alone if the full logo isn't suitable for your use case)
7. Do not change the typeface
8. Do not skew or distort the dimensions of the logo
9. Do not rotate the logo or elements of the logo



Flexibility

OK, we just got through some examples of what not to do with our logo specifically in official comms, but LooksRare's brand should stand as a symbol for the Web3 community. In that spirit, we welcome any and all artistic expressions that incorporate the LooksRare brand faithfully. Show your skills 👀💚

Artwork featured below by @tara_547, @PonderingDraw, @adiganz, @Souii_Arts, @PhuTwenty_, @sammjeenn, @basherbasir6, @surlyanoom!



Again, if you've got any questions, shoot us an email at design@looksrare.org

*Last updated on **Aug 26, 2023***

Business

For business partnerships, please submit a request via this form.

Official Business Partnership Contacts

For any business partnership discussions, please ensure that you are only communicating with the following team members from LooksRare:

Leeroy - BD Manager

- Discord: Leeroy#9875
- Telegram: @LeeroyLooksRare
- Email: leeroy@looksrare.org

*Last updated on **Aug 28, 2023***

LOOKS Staking to Earn Platform Fees

(Archived. See here for updated data)

> **TL;DR**: LOOKS token stakers earn 75-100% of the trading fees on LooksRare.

LooksRare collects a basic sales fee of 2% (in WETH) on all NFT sales excluding private sales.

The WETH collected from these fees are consolidated at the end of each recurring 6,500 Ethereum block period (roughly 24 hours) and then **distributed to LOOKS stakers in a linear format per block over the next 6,500 block period.**

How often can I claim my WETH rewards?

Any time you want. There's no limit on the number of times you can claim your rewards each day, but just be cautious of the gas fees you are paying to claim.

What happens if I forget to claim my WETH rewards for today?

No need to worry. As long as your LOOKS remain staked it will continue to accumulate even if you don't claim every day. You can leave it until gas fees are low or you've accumulated enough to make it worth claiming.

Why don't private sales count to volume calculations for fee sharing?

Unlike other marketplaces, LooksRare doesn't take a fee on private sales, so the volume generated by private sales doesn't contribute anything to the WETH that's earned by LOOKS stakers.

Active and Passive Staking

The rewards for each day are calculated at the end of the previous period, then split between Active and Passive stakers before they're distributed.

> The main difference between Active and Passive stakers is that **LOOKS in passive staking do NOT earn additional LOOKS while staked.**

Active stakers are the majority of stakers, whose staked LOOKS tokens are fully unlocked.

Passive stakers are stakers whose LOOKS tokens are locked for trading but unlocked for staking: namely Team, Treasury and Strategic sale tokens (which unlock over time - read more).

At the start of each 6,500 block period, a portion of the WETH fees collected from the previous 6,500 blocks are sent to the Passive Staking address. The amount is calculated as:

$$\frac{a}{b} \times c$$

Where

- a = Total amount of LOOKS staked passively at start of period
- b = Total amount of LOOKS staked at start of period (passive + active)
- c = Amount of WETH fees from previous 6,500 blocks

The remaining WETH is then sent to the Active Staking Pool.

How is WETH fee sharing calculated for Active LOOKS stakers?

For each 6,500 block period, the amount of WETH rewards to be distributed in each block to active stakers is calculated by:

$$\frac{\text{Total WETH collected as fees in the prior 6500 block period}}{6500}$$

Each user's amount of staked LOOKS at each block is then compared against the total amount of LOOKS staked at each block, with this being done at every block within the 6,500 block period to find the total amount of WETH rewards received.

Example

- Let's assume LooksRare processes 10,000 WETH of non-private sale volume in the first 6,500 blocks after launch, collecting total fees of 200 WETH.
- At block #6500, User A stakes 10,000 LOOKS in preparation to start earning WETH at block #6501.
- At block #6501,
  - The amount of passive tokens staked is 10% of the total amount of tokens staked, meaning 20 WETH is sent to the passive staking address.
  - The remaining 180 WETH is put into the active LOOKS staking pool, with 0.027692 WETH to be earned per block for 6,500 blocks (180 / 6,500).
  - The total amount of LOOKS staked from ALL users is 1,000,000 LOOKS.
  - This means that User A will be allocated: $\frac{10,000}{1,000,000} \times 0.027692 = 0.00027692$ WETH at block #6501.
- At block #6502,
  - Assuming User A's stake does not change, but the total amount of LOOKS staked from all users increases to 1,500,000 LOOKS, then User A will be allocated: $\frac{10,000}{1,500,000} \times 0.027692 = 0.00018641$ WETH at block #6502.
- This calculation each block will then continue for another 6,498 blocks (since our example covers a 6,500 block period), with 0.027692 WETH as the amount to be earned per block.
- The WETH fees generated from trades on the LooksRare platform in the 6,500-block period between block #6501 and block #13000 will then be used as the rewards to be distributed linearly over the next period, which will be from block #13001 and #19500.

*Last updated on **Aug 28, 2023***

Listing Rewards Archived

LooksRare's Listing Rewards program enables NFT collectors and traders to earn LOOKS tokens just by listing NFTs for sale on LooksRare.

Traders receive points for their eligible listings every 10 minutes, which are converted into LOOKS once daily.

What are the criteria for listings to earn points?

Users will receive Listing Reward points **every 10 minutes** for all items that they have listed for sale on LooksRare that meet these criteria:

- The listed NFT is from one of the top 5 eligible collections ranked by rolling 24 hour volume (ranking updates every 10 minutes)
- The listing is an active and executable order
- The listing price is **within 1.4x** of the global collection floor price (the lowest floor price on either OpenSea or LooksRare)
- Listing must be within the max limit of 200 eligible listings per collection at each snapshot
- Listings must be live for a minimum of 10 minutes in order to start accumulating points
- Listings must not expire within 1 hour of each snapshot in order to qualify for points for that snapshot

Which collections are eligible?

In addition to the below criteria for individual listings, the collections must meet these criteria to be eligible for listing rewards.

- Total item count of 100,000 or less
- Conform to ERC-721 token standard. Unfortunately, ERC-1155s and other standards introduce too many complications in the calculation methodology.

Note: Collections identified to be artificially increasing trading volume for the sole purpose of increasing their collection volume rank will have their eligibility removed. The eligibility of creator portal contracts from various marketplaces has also been removed.

When are points counted and distributed?

Users receive points every 10 minutes for listings that meet the criteria above.

Each day's points will start being counted from 9:00 AM (UTC) and run for 144 snapshots every 10 minutes until 9:00 AM (UTC) the next day.

At the end of each day, each user's total points balance will be used to determine their LOOKS rewards for that day. This is calculated as: (Alice's total points for the day / All users' points for the day) * Total LOOKS allocated for listing rewards.

Each day's LOOKS rewards will be claimable at 11:00 AM (UTC), at the same time as the trading rewards claim.

How are points calculated?

The amount of points that users will receive for each listing are relative to the floor multiple at the time of each snapshot. The formula is as follows:

- Listed price is ≤ 1.1x floor price = 10x points
- Listed price is > 1.1x but ≤ 1.2x floor price = 4x points
- Listed price is > 1.2x but ≤ 1.3x floor price = 2x points
- Listed price is > 1.3x but ≤ 1.4x floor price = 1x points

For example, if Azuki is the Rank 1 Collection by 24hr rolling volume (500 base points per listing) and the the floor price of Azuki is 30 ETH, a user that lists at 32 ETH will receive 5,000 points at each snapshot and a user that lists at 34 ETH will receive 800 points at each snapshot so long as they are within the 200 lowest listings of that collection.

Every 10 minutes, LooksRare will update the collection leaderboard on the listing rewards page to show the Top 5 ranked eligible collections by rolling 24 hour volume on OpenSea. The corresponding points per item listed in each collection will be applied based on the respective collection rank.

Collections ranked by rolling 24 hour volume	Base Points per item listed at every 10 minute snapshot
Rank 1 Collection	500
Rank 2 Collection	360
Rank 3 Collection	200
Rank 4 Collection	100
Rank 5 Collection	50

Example

The Top 5 ranked ERC721 collections are listed on the listing rewards page. For the purposes of this example, the top 5 ranked projects by volume at **10:00 AM (UTC)** are listed below, and 100,000 LOOKS are allocated to daily listing rewards.

This means that at exactly 10:00 AM (UTC), each BAYC NFT listed at 140 ETH or below on LooksRare will earn their owners **500 points**.

Rank	Collection	24 hour rolling volume (ETH)	Current Floor Price (ETH)	Listing Price (ETH) to be eligible for points	200th lowest listing price (ETH)	Points per item listed at every 10 minute snapshot
1	Bored Ape Yacht Club	3,096	100	≤140	145	500
2	Azuki	2,390	16	≤22.4	22	360
3	Clone X	2,032	14	≤19.6	18	200
4	Doodles	1,066	12	≤16.8	15	100
5	Cool Cats	941	10	≤14	16	50

As an example if at 9:58 AM (UTC), Bob, a LooksRare user, listed 3 BAYC for 120 ETH, 2 Azuki for 25 ETH, 1 Clone X for 19 ETH, and 10 Cool Cats for 11 ETH each, Bob would earn (3 x 500 x 4) + (10 x 50 x 10) = **10,400 points at the 10:00 AM (UTC) snapshot**. Note that the 2 Azukis did not earn points for this snapshot as they were listed above 1.4x floor price and the Clone X did not earn points as it was not within the lowest 200 listings for that collection. Also note that the listed BAYC received a multiple boost of 4x points for being listed at 1.2x floor price, while the listed Cool Cats received a multiple boost of 10x points for being listed at 1.1x floor price.

If the floor price and volume rankings do not change for the **next 10 minutes**, Bob will earn **another** 10,400 points at the 10:10 AM (UTC) snapshot.

At 11:00 AM (UTC), Bob decides to come back and check how his listing rewards are going. He notices that the rankings have changed due to some sales that occurred in the past hour. The Azuki collection has increased in volume to be the number 1 ranked collection and every listed Azuki within 1.4x of the floor price at this snapshot will now earn at least 200 points for their owners. Similarly, Cool Cats increased in volume to be rank #4 and owners will now earn at least 100 points for each listed Cool Cat. As both BAYC and Doodles were pushed down in rank, they will earn less per NFT listed in this snapshot.

Rank	Collection	24 hour rolling volume (ETH)	Current Floor Price (ETH)	Listing Price (ETH) to be eligible for points	200th lowest listing price (ETH)	Points per item listed at every 10 minute snapshot
1	**Azuki**	3,255	18	≤25.2	22	500
2	Bored Ape yacht Club	3,096	100	≤140	145	360
3	Clone X	2,032	14	≤19.6	18	200
4	**Cool Cats**	1,066	10	≤14	16	100
5	Doodles	1,066	12	≤16.8	15	50

For the 11:00 AM (UTC) snapshot, Bob will receive (2 x 500 x 1) + (3 x 360 x 4) + (10 x 100 x 10) = **12,200 points**. Note that because the Azuki floor increased to 18 ETH at the time of this snapshot, Bob's original listings of 25 ETH are now within the 1.4x floor price range, and will thus earn points at a 1x points multiple.

At **9:00 AM (UTC)** the next morning, Bob sees that he has accumulated a total of 150,000 points for the previous day, as well as 30,000,000 being the combined number of points from all users the previous day. Bob is now able to see that his rewards for the previous day are 500 LOOKS, calculated by (150,000 / 30,000,000) * 100,000.

*Last updated on **Aug 28, 2023***

Liquidity Provider Rewards

`Update Feb 8, 2022`

As announced here, the liquidity rewards program will be discontinued starting from 2022/02/15, 10:15 AM (UTC) and the tokens will be reallocated to be used for long-lasting Uniswap V3 liquidity managed by the team.

- **We will be discontinuing the Liquidity Rewards Program** at Ethereum Block 14211012 on Day 37 (Approximately 2022/02/15, 10:15 AM (UTC)), leaving 2,660,000 LOOKS to reallocate to V3.
- We'll **reallocate the leftover tokens to pair with ETH to add as V3 liquidity** ourselves. The ETH for the LP will come from directly from our Treasury.

LP token stakers can continue to earn rewards for another week. We are giving you a one-week heads up before implementing the change to give you enough time to reassess your positions going forward.

If you haven't unstaked your LP tokens by the time rewards end, don't sweat. The pool will be open and available on the Rewards page for everyone to withdraw from for the foreseeable future.

We understand that this may be an inconvenience for those who provided liquidity for the sake of earning LP rewards on a longer timeframe, but we are confident that this change is for the best of both the project and LOOKS holders/traders long-term.

`Original Article`

LooksRare is incentivizing liquidity for the LOOKS token by giving rewards for users that stake LOOKS-ETH UniV2 LP tokens.

The initial liquidity program will continue for **500,000 blocks** (approximately 77 days) with the rate of rewards set at **10 LOOKS per block** for LP stakers. We will assess the performance and requirement of a further liquidity program after the 500,000 blocks have passed.

How are liquidity provider rewards calculated?

User A's LP rewards per block are calculated by:

$$\frac{\text{User A's LP tokens staked}}{\text{Total amount of LP tokens staked}} \times \text{LOOKS Rewarded per block}$$

Example

- User A stakes 100 LOOKS/ETH LP tokens on Day 5
- The total amount of total staked LOOKS/ETH LP tokens is now 10,000

Based on the above, User A would receive: $\frac{1,000}{10,000} \times 10 = 1$ LOOKS per block.

How do I set up liquidity on UniV2?

To learn how to add liquidity on Uniswap V2, read our guide on adding liquidity

*Last updated on **Aug 28, 2023***

LooksRare Protocol Overview

What is the LooksRare's V2 Protocol?

LooksRare's V2 Protocol consists of a hybrid off-chain/on-chain system where taker orders are matched with maker orders to execute trades for NFTs.

It is a non-custodial exchange where orders require approval of the tokens (fungible and non-fungible) being transferred in a trade. Off-chain orders ("Maker orders") are EIP-712 signatures that are stored off-chain.

Order matching

A transaction consists of a bilateral exchange of either ETH or an ERC-20 token against one or multiple NFT tokens (e.g. ERC-721, ERC-1155) with a specific amount at a specific price.

> In the protocol, the bid user spends fungible tokens to acquire NFT assets. On the other hand, the ask user sells NFT assets for fungible tokens.

Trades exist as a combination of taker/maker and bid/ask.

The `MakerBid` is a `Maker` object with the `quoteType` set to 0 (Bid) and the `MakerAsk` is a `Maker` object with the `quoteType` set to 1 (Ask).

While both `TakerAsk` and `TakerBid` are standard `Taker` objects defining the `recipient` of the asset (if matched against a `MakerAsk`) or funds (if matched against a `MakerBid`)

- **MakerBid** (`Maker`) — a passive order that exists off-chain where the user wishes to acquire an NFT using a specific ERC-20 token.
- **MakerAsk** (`Maker`) — a passive order that exists off-chain where the user wishes to sell an NFT for a specific ERC-20 token.
- **TakerBid** (`Taker`) — an order that is executed on-chain and matched against a `MakerAsk`, e.g., the taker accepts the offer from the maker and buys the NFT for the ETH or the ERC20 token specified.
- **TakerAsk** (`Taker`) — an order that is executed on-chain and matched against a `MakerBid`, e.g., the taker accepts the offer from the maker and sells the NFT for the ETH or the ERC20 token specified.

See Order Types for more details on each order type.

Execution strategies

Order matching relies on the concept of execution strategies that define the rules of how trades must be matched: (1) `price`, (2) `itemIds`, (3) `amounts`, (4) whether the trade invalidates the maker order (`isNonceInvalidated`). The strategies are mapped with a `strategyId`.

LooksRare's execution strategies enable the integration of customizable sets of rules, which can be expanded over time to effortlessly incorporate new functionalities, such as dutch auctions, into the protocol. A user can make use of a certain strategy by setting the `strategyId` in the `Maker` object to the id of the desired strategy. Then the `ExecutionManager` uses the `strategyId` to route the order details to the correct strategy for execution.

Each strategy consists of a contract function that takes two order inputs (`MakerAsk` w/ `TakerBid` or `MakerBid` w/ `TakerAsk`) and returns a 4-tuple:

1. The total `price` of the transaction.
2. An array containing the `itemIds` being transacted.
3. An array containing the `amounts` of each token id being transacted (ERC-721: 1 /ERC-1155: 1+).
4. The `isNonceInvalidated` flag defines whether the trade invalidates the maker order.

> In the Solidity codebase, standard transactions use the native strategy with `strategyId = 0`. Standard transactions means the maker decides the price, the itemIds, and the amounts. The transaction is atomic and it cannot be partially filled.

LooksRare's execution strategy design can allow for more advanced and innovative order types, such as:

- **Collection orders** — a `MakerBid` can be executable against any `TakerAsk` within a designated collection (e.g., *"I wish to buy an asset from the BAYC collection for 40 WETH"*).
- **Trait-based orders** — (pending implementation) through the use of cryptographic proofs (e.g., *"I wish to buy an asset from the BAYC collection for 55 WETH if its metadata for hat is a party hat"*).
- **USD based listings** — (pending implementation) enable the user to define the price in USD and receive the equivalent amount in ETH at the time of the sale. This is achieved with the use of the Chainlink ETH/USD price oracle to auto-convert at the transaction time when the `TakerBid` user executes the transaction.
- **Listing relative to the floor price** — (pending implementation) a strategy where the user can list an asset to be always a certain amount away from the floor. This is achieved with the use of the Chainlink NFT Floor Price Feed to calculate the final sale price based on the premium or discount defined by the `Maker`.
- **Bid for assets with an id within a certain range** — (pending implementation) a strategy to limit the `MakerBid` validity to a certain range of token ids.

Protocol and creator fees

On each trade, trading fees are taken from the execution price and can exist as (1) protocol fee and/or (2) creator fee. Protocol fees are always deducted from the final amount before transferring the profit to the Ask.

We currently have a 0.5% protocol fee and a 0% creator fee.

Protocol fees are present at the strategy level and include the following parameters:

- `standardProtocolFeeBp`: standard protocol fee for a strategy (in basis point).
- `minTotalFeeBp`: minimum total fee (in basis point).
- `maxProtocolFeeBp`: the maximum boundary if it is adjusted.

Creator fees can be assigned a specific on-chain logic that exists as a stand-alone contract.

- Creator rebate model: a fixed royalty applies (e.g. 0.5%)
- Full royalties model: full royalties apply

Two sources are used to determine creator fee recipients:

- LooksRare's RoyaltyFeeRegistry
- EIP-2981

There is no "slippage parameter" but instead an adjustable limit is set by the owner at the protocol level. If the creator fee is higher than a defined percentage, it reverts.

The protocol fee parameters for a strategy can be updated by the protocol owner but the `newStandardProtocolFeeBp` (and `newMinTotalFeeBp`) must be inferior to the `maxProtocolFeeBp` defined initially at the time of the strategy deployment.

What is the core architecture?

The LooksRare Protocol consists of a stand-alone contract (`LooksRareProtocol`) that coordinates with other smart contracts designed to provide instructions on how trades must be executed, verifies the validity of strategies and currencies, whether collection rebates must be paid (and to who), and how tokens must be transferred.

LooksRare's v2 Protocol core architecture consists of:

- **Currency manager** (`CurrencyManager`) — handles an allowlist of currencies that are valid for transactions within the protocol.
- **Execution manager** (`ExecutionManager`) — handles the execution by routing the maker order and taker order to the correct strategy implementation.
- **Nonce manager** (`NonceManager`) — handles the nonce logic that is used for invalidating maker orders that exist off-chain.
- **Transfer selector for NFT contracts** (`TransferSelectorNFT`) — handles the logic for transferring assets based on the `assetType` and for supporting non-compliant asset types or new standards.
- **Creator fee manager** (`CreatorFeeManager`) — provides a stand-alone logic that tells if creator fees should be collected and what amount. It supports ERC-2981 ("NFT Royalty Standard") and our own royalty registry (`RoyaltyFeeRegistry`).
- **Affiliate manager** (`AffiliateManager`) — handles the management of affiliates for the LooksRare protocol.

Other design decisions

Protocol upgradeability

LooksRare v2 Protocol's modular design allows the upgradeability of specific parts of the exchange **without any need for proxies**, circuit breakers (e.g., the possibility to discontinue a strategy or a currency), and the rollout of new features continuously (e.g., support of new order types).

Advanced cancellation features with a triple-nonce system

- **Global user bid/ask nonces** — each user address is mapped to one ask nonce and one bid nonce. The nonce on the maker order struct must match the signer's value on the chain. This can be used for canceling all pending maker bids or/and pending maker asks.
- **Order nonce** — used to know whether an order with this nonce can be filled, is being filled or has been filled/cancelled. If an order has started "partial filling", it is impossible to execute other orders sharing this order nonce.
- **Subset nonce** — comes in addition to the order nonce, and serves a different purpose. They can be used to group a set of arbitrary orders under the same subset nonce. A subset nonce can only be cancelled manually. When a subset nonce is invalidated, all the orders sharing this subset nonce become invalid.

This system allows for innovative features such as OCO to replace passive orders (with more aggressive orders). The protocol also supports batch cancellation for orders and other "cancel-all" features.

See more details about our triple-nonce system Triple-nonce System

Partial fills

The LooksRare Protocol supports partial fills. The strategy contract decides whether to invalidate the order nonce. For instance, a strategy could include a stand-alone logic allowing multiple fills.

If the strategy does not invalidate the order nonce, it will instead map the order hash to the user's `orderNonce`. This prevents the execution of another maker order, which may share the same order nonce.

- Each order isn't atomic anymore, meaning that a strategy can include stand-alone logic allowing multiple fills.
- Events for (`TakerBid` & `TakerAsk`) reflect whether the order is invalidated.

Affiliate program

The LooksRare Protocol implements an affiliate system. All functions for executing orders accept an affiliate address, if set, the system would give a percent of the protocol fee from the trade to the affiliate defined in the call.

The affiliate rates are defined in basis points and are managed in the `AffiliateManager` contract.

- Each "execution" function at the protocol level has an affiliate system.
- Each affiliate address has a mapping with the percentage it can obtain for referring new users.
- When a transaction occurs with a valid affiliate address specified, there is an event issued (`AffiliatePayment`).

Batch listing in a single signature

A signature can include an arbitrarily large number of maker orders. To do so, users can sign a Merkle root from a tree that is computed using different order hashes (derived from multiple `MakerBid` and/or `MakerAsk`).

- With the protocol, one signature can include an arbitrarily large number of maker orders.
- Users can sign a Merkle root that contains order hashes (vs. signing the order hash).
- The computation of the Merkle tree occurs at the application level (with the SDK). For more information, see LooksRare SDK v2

Bundle orders

Users can submit `MakerBid` or `MakerAsk` orders on a set of `itemIds` with specific `amounts`.

- Bundles can be created only for a single collection address. A bundle can contain multiple `itemIds` and multiple `amounts` (for ERC-1155).
- Bundles can only be executed for EIP-2981 if all `itemIds` share the same royalty fee structure (i.e. same creator percent and same recipient).

Multi-purchase in one transaction

The LooksRare Protocol enables the matching of multiple pairs of `Maker` and `Taker` in a single transaction natively. The only requirement is that it must be using the same currency. This offers considerable savings in transaction gas.

The execution can either be atomic or non-atomic.

- **Atomic**: if one purchase fails, the whole transaction is reverted and the execution fails. No purchase occurred.
- **Non-atomic**: if one or more purchase fails, the transaction will not revert and the execution will be partially successful, meaning not all the `Maker` were able to be successfully matched to a `Taker`.

The protocol also offers benefits to users such as the human-readability of the signed message thanks to the use of EIP-712 signatures but it also supports EIP-1271 as a signature validation method for smart contracts.

Audit Report and Security Assurance

At LooksRare security is a top priority. The smart contracts have been rigorously audited by reputable third-party auditors to ensure the highest level of safety and reliability. These audits assess the contracts for potential vulnerabilities, and compliance with best practices in the blockchain industry.

We are committed to maintaining the highest security standards and to continuously improve our smart contract infrastructure responsibly. Below the audits reports for the LooksRare Protocol (v2).

Spearbit

> Audit Report: https://github.com/spearbit/portfolio/blob/master/pdfs/LooksRare-Spearbit-Security-Review.pdf

Last updated on Aug 28, 2023

Orders Types

Introduction

LooksRare's exchange is built on a hybrid system (off-chain/on-chain), which incorporates off-chain signatures (these are called Maker Orders and use the `Maker` schema defined below) and on-chain orders (called Taker Orders which use the `Taker` schema). This design is similar to other NFT marketplaces (e.g., OpenSea, Rarible) running on the Ethereum blockchain.

LooksRare uses exclusively **EIP-712 signatures**, which provide key benefits for users such as greater readability for humans in the message they sign.

A maker order (`Maker`) is a passive order, which can be executed (if not cancelled) against a taker order (`Taker`) after it is signed. Since the execution of a maker order is done once a taker order matches on-chain, network gas fees are never paid by the maker user.

Signatures for the LooksRare exchange all contain a set of parameters that are defined in a Solidity struct named `Maker`. While Taker Orders are standardized in a Solidity struct named `Taker`. For more information about the protocol see the LooksRare Protocol Overview.

Overview of parameters in a Maker

The `Maker` struct contains 15 distinct parameters:

- `quoteType` — Whether the order is an Ask (sending a passive order to sell a NFT) or a Bid (sending a passive order to buy an NFT).
- `globalNonce` — The global user order nonce for maker orders.
- `subsetNonce` — The subset nonce (shared across bid/ask maker orders).
- `orderNonce` — The order nonce (it can be shared across bid/ask maker orders).
- `strategyId` — The id of the strategy used in the maker order.
- `collectionType` — The collection type (i.e. ERC721 or ERC1155).
- `collection` — The collection's address.
- `currency` — The currency address for this order.
- `signer` — The signer's address.
- `startTime` — The start time for the order (in epoch format).
- `endTime` — The end time for the order (in epoch format).
- `price` — Minimum price for maker ask, maximum price for maker bid.
- `itemIds` — Array of item ids.
- `amounts`— Array of amounts.
- `additionalParameters` — Can contain additional parameters that are not used for standard orders.

Breakdown of parameters

`quoteType`

> Type: `BigNumber`

Like most exchange marketplaces & protocols (including non-NFT marketplaces), an order can be of two distinct types: Ask or Bid.

In the LooksRare protocol, `Ask` means the user is selling a NFT whereas `Bid` means the user is buying a NFT.

Value	Description
0	Maker order is a `Bid`
1	Maker order is a `Ask`

`globalNonce`

> Type: `BigNumber`

The `globalNonce` field represent the global user order nonce for the maker order. Each user address is mapped to one ask nonce and one bid nonce. The nonce on the maker order struct must match the signer's value on the chain. This can be used for canceling all pending maker bids or/and pending maker asks.

`subsetNonce`

> Type: `BigNumber`

The `subsetNonce` is assigned a to each order. A subset nonce can only be cancelled manually. It cancels all orders sharing this subset and can be shared for bid and ask maker orders.

`orderNonce`

> Type: `BigNumber`

The `orderNonce` is used to know whether an order with this nonce can be filled, is being filled, or has been filled/cancelled. If an order has started "partial filling", it is impossible to execute other orders sharing this order nonce.

`strategyId`

> Type: `BigNumber`

The `strategyId` represent the id of the strategy being used in the maker order. Strategies can only be valid in the LooksRare protocol if they are implemented in the `StrategyManager` contract. If the execution strategy is not included, the trade cannot occur (it reverts). For more details, see LooksRare Protocol - Execution Strategies

`collectionType`

> Type: `BigNumber`

The `collectionType` field represent the type of collection included in the `Maker` object.

Value	Description
0	ERC-721
1	ERC-1155

`collection`

> Type: `string`

The `collection` field represent the address of the asset's collection.

`currency`

> Type: `string`

The `currency` field represent the address of the currency to execute the trade. Currencies can only be valid in the LooksRare Protocol if they are allowed by the `CurrencyManager` contract. If the currency is not allowed, the trade cannot occur (it reverts).

`signer`

> Type: `string`

The `signer` field represent the address used to EIP-712 sign the `Maker` object.

`startTime`

> Type: `BigNumber`

The `startTime` represents the epoch timestamp of a maker order from where its execution can be valid.

`endTime`

> Type: `BigNumber`

The `endTime` represents the epoch timestamp of a maker order from where its execution stops being valid.

`price`

> Type: `BigNumber`

The price for the order **in WEI**. It is a `uint256` value, which doesn't contain any decimal. Ethereum does not support float numbers. Price must be displayed as a number in "BigNumber" formatting.

Desired price	Actual filed value
0.1 ETH	100000000000000000
1 ETH	1000000000000000000
100 ETH	100000000000000000000

Converters can be useful to convert WETH (or other assets with **18 decimals**) to wei.

> Never sign a message with a price displayed as "10." if you are looking to sell an item for 10 WETH, you would end up selling your item for 100 wei!

`itemIds`

> Type: `BigNumber[]`

The `itemIds` field is an array containing the token id of each asset.

> For collection orders, this field is not used in the execution and set it at 0 if you use the exchange frontend available at looksrare.org.

`amounts`

> Type: `BigNumber[]`

The `amounts` field is an array defining the amount of asset to transfer for each `itemId`. must be `1` for ERC-721, while it can be `1+` for ERC-1155.

`additionalParameters`

> Type: `string[]`

The `additionalParameters` field is used for additional parameters specific to complex orders that are less frequent (e.g., private sale, Dutch auctions, and more complex order types). The set of additional parameters is represented as `bytes`, where parameters are concatenated.

Further reading:

> For more details, please visit this website to read more about digital signatures.

Overview of parameters in a Taker

The `Taker` struct contains 2 distinct parameters:

- `recipient` — The address which will receive the asset (if matched with a `MakerAsk`) or the profits (if matched with a `MakerBid`).
- `additionalParameters` — Can contain additional parameters that are not used for standard orders.

Solidity Structs & Enums

When working with EIP-712 signatures and interacting with contracts it can be useful to understand how certain data structures are defined at within the solidity smart contracts. Here are the main data structures you will have to work with when interacting with the LooksRare protocol.

The `Maker` struct as defined at the contract level:

```
struct Maker {
    QuoteType quoteType;
    uint256 globalNonce;
    uint256 subsetNonce;
    uint256 orderNonce;
    uint256 strategyId;
    CollectionType collectionType;
    address collection;
    address currency;
    address signer;
    uint256 startTime;
    uint256 endTime;
    uint256 price;
    uint256[] itemIds;
    uint256[] amounts;
    bytes additionalParameters;
}
```

The `Taker` struct as defined at the contract level:

```
struct Taker {
    address recipient;
    bytes additionalParameters;
}
```

The `QuoteType` enum as defined at the contract level:

```
enum QuoteType {
    Bid,
    Ask
}
```

The `CollectionType` enum as defined at the contract level:

```
enum CollectionType {
    ERC721,
    ERC1155
}
```

Last updated on ***Aug 28, 2023***

Execution Strategies

Order matching relies on a concept of execution strategies that define the rules of how trades must be matched: (1) `price`, (2) `itemIds`, (3) `amounts`, (4) whether the trade invalidates the maker order (`isNonceInvalidated`). The strategies are mapped with a `strategyId`, which is defined by the signer at the creation of a maker order.

LooksRare's execution strategies enable the integration of customizable sets of rules, which can be expanded over time to effortlessly incorporate new functionalities, such as dutch auctions, into the protocol. The strategy is defined in the maker order through the use of a field named `strategyId`, which contains the strategy id used by the `ExecutionManager` to handle and route the order details to the correct strategy for execution.

1. The total `price` of the transaction.
2. An array containing the `itemIds` being transacted.
3. An array containing the `amounts` of each token id being transacted (ERC-721: 1 /ERC-1155: 1+).
4. The `isNonceInvalidated` flag defines whether the trade invalidates the maker order.

> In the Solidity codebase, standard transactions use the native strategy with `strategyId = 0`. Standard transactions means the maker decides the price, the itemIds, and the amounts. The transaction is atomic and it cannot be partially filled.

Strategy IDs List

`strategyId`	Description
0	Standard strategy
1	Collection offer strategy

Advanced strategies documentation

In this section, we will document how to take advantage of non-standard strategies which require `additionalParameters` or the user of a Merkle tree.

Collection offer

When creating a collection offer the only differences from a standard `MakerBid` relies on the `strategyId` being set to `1` and the `itemIds` array set to `[0]`.

As shown below.

```
const makerBid: Maker = {
    ...
    strategyId: 1
    itemIds: [0]
    ...
}
```

When accepting a collection offer the tokenId of the asset being sold needs to be encoded in the `additionalParameters` of the `TakerAsk`. That can be achieved by using the `encodeParams` function from the LooksRare SDK v2.

As shown below.

```
const takerAsk: Taker = {
    ...
    additionalParameters: encodeParams([TOKEN_ID], ["uint256"])
}
```

An alternative to the `encodeParams` included within our SDK is to do the following.

```
const takerAsk: Taker = {
    ...
    additionalParameters: ethers.utils.defaultAbiCoder.encode(["uint256"], [TOKEN_ID]);
}
```

The `TOKEN_ID` placeholder should be replaced with the token id of the asset being sold.

Last updated on ***Aug 28, 2023***

Triple-nonce System

The LooksRare Protocol implements a triple-nonce system which enables for advanced cancellation features. Nonces are used to check orders validity, to keep track of executed orders and are specific to each user; but they also enable features, like specific or global orders cancellation.

The three nonce are:

Global nonce (`globalNonce`)

Each user address is mapped to one ask nonce and one bid nonce. The nonce on the maker order struct must match the signer's value on the chain. This can be used for canceling all pending maker bids or/and pending maker asks.

- There are 2 global nonces per user: bid nonce (when `quoteType = 0`) and ask nonce (when `quoteType = 1`).
- They are used to cancel all the bids and/or all the asks for a user.
- For an order, if its **order nonce** value equals the signer's **global nonce** value, then the order is valid. Else, it is not executable.
- Users can increment their bid and/or ask global nonces to cancel all/part of their orders.

> EXAMPLE
>
> User `0x54BE3a794282C030b15E43aE2bB182E14c409C5e` has a `globalNonce` (Ask) = 2.
>
> The user has a signed `MakerAsk` with a `orderNonce = 1` → **This order is invalid**
>
> The user has a signed `MakerAsk` with a `orderNonce = 2` → **This order is valid**
>
> The user has a signed `MakerAsk` with a `orderNonce = 3` → **This order is invalid**

Order nonce (`orderNonce`)

The order nonce is used to know whether an order can be filled, is being filled, or has been filled/cancelled. If an order has started "partial filling", it is impossible to execute other orders sharing this order nonce. The order nonce also allows the implementation of OCO orders.

A user can invalidate a order nonce (or a list of order nonces) manually in order to cancel the orders associated with these nonces.

In the context of strategies that enable partial fills, if the transaction does not fill fully the maker order, the user's order nonce gets mapped to the order hash.

- The order nonce is mapped with a bytes32 value.
- Each order nonce is mapped to `bytes32(0)` by default.
- When a maker order with the nonce is fully executed, the order nonce gets mapped to `keccak256("ORDER_NONCE_EXECUTED")`.
- Each order object signed by the user has to contain an order nonce.

> EXAMPLE
>
> User `0x54BE3a794282C030b15E43aE2bB182E14c409C5e` creates two maker orders with the same `orderNonce = 2`. The first order can be partially filled.
>
> The `orderNonce` 2 is by default mapped to `bytes32(0)` → **Both orders are valid**
>
> The first order gets partially filled, the `orderNonce` 2 is mapped to the first order's hash → **The second order that shares the same order nonce cannot be executed.**

Subset nonce (`subsetNonce`)

Subsets nonces are part of the order object signed by the user. They come in addition to the order nonce, and serve a different purpose. They are used to group a set of arbitrary orders under the same subset nonce.

Each order is assigned a subset nonce. A subset nonce can only be cancelled manually. When a subset nonce is invalidated, all the orders sharing this subset nonce become invalid. Subset nonces can be shared across `MakerAsk` and `MakerBid`.

The protocol is agnostic of how orders are grouped together with subset nonces. It can be enforced at the application level, for example, to group all the user's orders from a single collection under the same subset nonce. Or it can be customized by the user to create custom subsets.

> EXAMPLE
>
> User `0x54BE3a794282C030b15E43aE2bB182E14c409C5e` creates 10 orders, all with the same `subsetNonce = 2`.
>
> If 1 order is executed → **All the other 9 orders remain valid.**
>
> User invalidates the `subsetNonce` 2 → **All the 10 orders become invalid.**

*Last updated on **Aug 28, 2023***

Deployed Contract Addresses

The following page lists our current and past Solidity smart contracts.

LooksRare Protocol (v2)

LooksRareToken

Mainnet: 0xf4d2888d29d722226fafa5d9b24f9164c092421e

LooksRareProtocol

Mainnet: 0x0000000000E655fAe4d56241588680F86E3b2377

Goerli: 0x35C2215F2FFe8917B06454eEEaba189877F200cf

TransferManager

Mainnet: 0x0000000000060C4Ca14CfC4325359062ace33Fe3D

Goerli: 0xC20E0CeAD98abBBEb626B77efbBDc1E5D781f90c

OrderValidatorV2A

Mainnet: 0x2a784a5b5C8AE0bd738FBc67E4C069dB4F4961B7

Goerli: 0x7454Cc9AEB024bcE6A2CDC49ad4733B4D8215fb8

StrategyCollectionOffer

Mainnet: 0x0000000000E58692Bf5fA347312da78cA8Bfc8d2

Goerli: 0x17094C21e1936056Fcae6DE32bd6FB015dB743F7

RoyaltyFeeRegistry

Mainnet: 0x55010472a93921a117aad9b055c14106Oc8d8022

Goerli: 0x1240Sdb79325DO6a973aD913D6e9BdA1343cD526

RoyaltyFeeSetter

Mainnet: 0x66466107d9cae4da0176e699406419003f3c27a8

Goerli: 0x73d392242617F27DF51E9cd7B8B2A0e435abCa06

ProtocolFeeRecipient

Mainnet: 0x1838De7d4e4e42c8eB7b204A91e28E9fad14F536

TradingRewardsDistributor

Mainnet: 0x0554f068365eD43dcC98dcd7Fd7A8208a5638C72

FeeSharingSystem

Mainnet: 0x8cD7254A1D759EFA08eC7c3291B2E85c5dCC12ce

AggregatorFeeSharingWithUniswapV3

Mainnet: 0x3ab16Af1315dc6C95F83Cbf622fecF98D00fd9ba

UniswapV3Pool

Mainnet: 0xDC00bA87Cc2D99468f7f34BC04CBf72E111A32f7

LooksRareAirdrop

Mainnet: 0xA35dce3eOE6ceb67a30b8D7f4aEe721C949B5970

Timelock

Mainnet: 0xa6a96fa698a6d5afcef6e8efeacaae7ee43f2486

Deprecated Protocol (v1)

The following contracts were being used in the previous version of the protocol.

LooksRareExchange

> This contract is deprecated and no longer maintained by LooksRare.

Mainnet: 0x59728544b08ab483533076417fbbb2fdOb17ce3a

Goerli: 0xD11246647 1b5438C1ca2D21869420Oe49d81DO47

StrategyStandardSaleForFixedPrice

> This contract is deprecated and no longer maintained by LooksRare.

Mainnet: 0x579af6FD30BF83a5AcOD636bc619f98DBdeb930c

Goerli: 0x6ACbeb7f6E225FbCOD1CEe27a40ADC49E7277E57

StrategyAnyItemFromCollectionForFixedPrice

> This contract is deprecated and no longer maintained by LooksRare.

Mainnet: 0x09F93623019049c76209C26517AcC2aF9d49c69b

Goerli: 0xAfb81825DCO765OOBF19CBf69d443684F2EBd3f6

CurrencyManager

> This contract is deprecated and no longer maintained by LooksRare.

Mainnet: 0xC881ADdf409eE2C4b6bBc8B607c2C5CAFaB93d25

Goerli: 0xAC56740Fa471C0266B59B101C76Ab771f199F2eA

ExecutionManager

> This contract is deprecated and no longer maintained by LooksRare.

Mainnet: 0x9cc58bf22a173c0fa8791c13df396d18185d62b2

Goerli: 0x972B3A1A89ABb5e489163345E3C44Ab1bB813986

OrderValidatorV1B

> This contract is deprecated and no longer maintained by LooksRare.

Mainnet: 0xdC56dE26C64B14f1A8e95582A16b6A9AEfEEd395

Goerli: 0xdD6178143FbOa16DEC9261498F6007a7468D7Aea

RoyaltyFeeManager

> This contract is deprecated and no longer maintained by LooksRare.

Mainnet: 0xCBfebA41C3e69d24B5C8bO4Ed60C42CC5D883620

Goerli: 0x37Ac5Af9eE9394F4356E0C7aaBeF3c93731A5c90

TransferManagerERC721

> This contract is deprecated and no longer maintained by LooksRare.

Mainnet: 0xf42aa99F011A1fA7CDA90E5E98b277E306BcA83e

Goerli: 0xF8C81f3ae82b6EFC9154c69E3db57fD4da57aB6E

TransferManagerERC1155

> This contract is deprecated and no longer maintained by LooksRare.

Mainnet: 0xFED24eC7E22f573c2e08AEF55aA6797Ca2b3A051

Goerli: 0xF2ae42e871937F4e9ffb394C5A814357C16eO6d6

TransferManagerNonCompliantERC721

> This contract is deprecated and no longer maintained by LooksRare.

Mainnet: 0x3e538190635f51435298ee58a7984961120510a1

Goerli: 0x1c90A2f30090a79E0dee770161C1879b31B6F74d

TransferSelectorNFT

> This contract is deprecated and no longer maintained by LooksRare.

Mainnet: 0x9ba628f27aac9b2d78a9f2bf40a8a6df4ccd9e2c

Goerli: 0x1b03835903D54dd4DB41F04B07188C0f7645edD7

UniswapV2Pool

> This contract is deprecated and no longer maintained by LooksRare.

Mainnet: 0x4b5Ab61593A2401B1075b90c04cBCDD3F87CE011

Last updated on ***Aug 28, 2023***

AffiliateManager

This contract handles the management of affiliates for the LooksRare protocol.

isAffiliateProgramActive

```
bool isAffiliateProgramActive
```

Whether the affiliate program is active.

affiliateController

```
address affiliateController
```

Address of the affiliate controller.

affiliateRates

```
mapping(address => uint256) affiliateRates
```

It tracks the affiliate rate (in basis point) for a given affiliate address. The basis point represents how much of the protocol fee will be shared to the affiliate.

constructor

```
constructor(address _owner) public
```

Constructor

Parameters

Name	Type	Description
_owner	address	Owner address

updateAffiliateRate

```
function updateAffiliateRate(address affiliate, uint256 bp) external
```

This function allows the affiliate controller to update the affiliate rate (in basis point).

Parameters

Name	Type	Description
affiliate	address	Affiliate address
bp	uint256	Rate (in basis point) to collect (e.g. 100 = 1%) per referred transaction

updateAffiliateController

```
function updateAffiliateController(address newAffiliateController) external
```

This function allows the owner to update the affiliate controller address.

Only callable by owner.

Parameters

Name	Type	Description
newAffiliateController	address	New affiliate controller address

updateAffiliateProgramStatus

```
function updateAffiliateProgramStatus(bool isActive) external
```

This function allows the owner to update the affiliate program status.

Only callable by owner.

Parameters

Name	Type	Description
isActive	bool	Whether the affiliate program is active

BatchOrderTypehashRegistry

The contract generates the batch order hash that is used to compute the digest for signature verification.

hashBatchOrder

```
function hashBatchOrder(bytes32 root, uint256 proofLength) public pure returns (bytes32 batchOrder
```

This function returns the hash of the concatenation of batch order type hash and merkle root.

Parameters

Name	Type	Description
root	bytes32	Merkle root
proofLength	uint256	Merkle proof length

Return Values

Name	Type	Description
batchOrderHash	bytes32	The batch order hash

_getBatchOrderTypehash

```
function _getBatchOrderTypehash(uint256 height) internal pure returns (bytes32 typehash)
```

It looks like this for each height

```
height == 1: BatchOrder(Maker[2] tree)Maker(uint8 quoteType,uint256 globalNonce,uint256 subsetNonc

height == 2: BatchOrder(Maker[2][2] tree)Maker(uint8 quoteType,uint256 globalNonce,uint256 subsetN

height == n: BatchOrder(Maker[2]...[2] tree)Maker(uint8 quoteType,uint256 globalNonce,uint256 subs
```

*Last updated on **Aug 28, 2023***

CreatorFeeManagerWithRebates

This contract returns the creator fee address and the creator rebate amount.

STANDARD_ROYALTY_FEE_BP

```
uint256 STANDARD_ROYALTY_FEE_BP
```

Standard royalty fee (in basis point).

royaltyFeeRegistry

```
contract IRoyaltyFeeRegistry royaltyFeeRegistry
```

Royalty fee registry interface.

constructor

```
constructor(address _royaltyFeeRegistry) public
```

Constructor

Parameters

Name	Type	Description
_royaltyFeeRegistry	address	Royalty fee registry address.

viewCreatorFeeInfo

```
function viewCreatorFeeInfo(address collection, uint256 price, uint256[] itemIds) external view re
```

This function returns the creator address and calculates the creator fee amount.

Parameters

Name	Type	Description
collection	address	Collection address
price	uint256	Transaction price
itemIds	uint256[]	Array of item ids

Return Values

Name	Type	Description
creator	address	Creator address
creatorFeeAmount	uint256	Creator fee amount

CreatorFeeManagerWithRoyalties

This contract returns the creator fee address and the creator fee amount.

royaltyFeeRegistry

```
contract IRoyaltyFeeRegistry royaltyFeeRegistry
```

Royalty fee registry interface.

constructor

```
constructor(address _royaltyFeeRegistry) public
```

Constructor

Parameters

Name	Type	Description
_royaltyFeeRegistry	address	Royalty fee registry address.

viewCreatorFeeInfo

```
function viewCreatorFeeInfo(address collection, uint256 price, uint256[] itemIds) external view re
```

This function returns the creator address and calculates the creator fee amount.

> There are two on-chain sources for the royalty fee to distribute.
>
> 1. RoyaltyFeeRegistry: It is an on-chain registry where creator fee is defined for all items of a collection.
> 2. ERC2981: The NFT Royalty Standard where royalty fee is defined at an itemId level in a collection.
>
> The on-chain logic looks up the registry first. If it does not find anything, it checks if a collection supports ERC2981. If so, it fetches the proper royalty information for the itemId.
>
> For a bundle that contains multiple itemIds (for a collection using ERC2981), if the royalty fee/recipient differ among the itemIds part of the bundle, the trade reverts. This contract DOES NOT enforce any restriction for extremely high creator fee, nor verifies the creator fee fetched is inferior to the total price.
>
> If any contract relies on it to build an on-chain royalty logic, it should implement protection against:
>
> 1. high royalties
> 2. potential unexpected royalty changes that can occur after the creation of the order.

Parameters

Name	Type	Description
collection	address	Collection address
price	uint256	Transaction price
itemIds	uint256[]	Array of item ids

Return Values

Name	Type	Description
creator	address	Creator address
creatorFeeAmount	uint256	Creator fee amount

CurrencyManager

This contract manages the list of valid fungible currencies.

isCurrencyAllowed

```
mapping(address => bool) isCurrencyAllowed
```

It checks whether the currency is allowed for transacting.

constructor

```
constructor(address _owner) public
```

Constructor

Parameters

Name	Type	Description
_owner	address	Owner address

updateCurrencyStatus

```
function updateCurrencyStatus(address currency, bool isAllowed) external
```

This function allows the owner to update the status of a currency.

Only callable by owner.

Parameters

Name	Type	Description
currency	address	Currency address (address(0) for ETH)
isAllowed	bool	Whether the currency should be allowed for trading

*Last updated on **Aug 28, 2023***

ExecutionManager

This contract handles the execution and resolution of transactions. A transaction is executed on-chain when an off-chain maker order is matched by on-chain taker order of a different kind. For instance, a taker ask is executed against a maker bid (or a taker bid against a maker ask).

protocolFeeRecipient

```
address protocolFeeRecipient
```

Protocol fee recipient.

maxCreatorFeeBp

```
uint16 maxCreatorFeeBp
```

Maximum creator fee (in basis point).

creatorFeeManager

```
contract ICreatorFeeManager creatorFeeManager
```

Creator fee manager.

constructor

```
constructor(address _owner, address _protocolFeeRecipient) public
```

Constructor

Parameters

Name	Type	Description
_owner	address	Owner address
_protocolFeeRecipient	address	Protocol fee recipient address

updateCreatorFeeManager

```
function updateCreatorFeeManager(address newCreatorFeeManager) external
```

This function allows the owner to update the creator fee manager address.

Only callable by owner.

Parameters

Name	Type	Description
newCreatorFeeManager	address	Address of the creator fee manager

updateMaxCreatorFeeBp

```
function updateMaxCreatorFeeBp(uint16 newMaxCreatorFeeBp) external
```

This function allows the owner to update the maximum creator fee (in basis point).

The maximum value that can be set is 25%. Only callable by owner.

Parameters

Name	Type	Description
newMaxCreatorFeeBp	uint16	New maximum creator fee (in basis point)

updateProtocolFeeRecipient

```
function updateProtocolFeeRecipient(address newProtocolFeeRecipient) external
```

This function allows the owner to update the protocol fee recipient.

Only callable by owner.

Parameters

Name	Type	Description
newProtocolFeeRecipient	address	New protocol fee recipient address

_executeStrategyForTakerOrder

```
function _executeStrategyForTakerOrder(struct OrderStructs.Taker takerOrder, struct OrderStructs.M
```

This function is internal and is used to execute a transaction initiated by a taker order.

Parameters

Name	Type	Description
takerOrder	struct OrderStructs.Taker	Taker order struct (taker specific parameters for the execution)
makerOrder	struct OrderStructs.Maker	Maker order struct (maker specific parameter for the execution)
sender	address	The address that sent the transaction

Return Values

Name	Type	Description
itemIds	uint256[]	Array of item ids to be traded
amounts	uint256[]	Array of amounts for each item id
recipients	address[2]	Array of recipient addresses
feeAmounts	uint256[3]	Array of fee amounts
isNonceInvalidated	bool	Whether the order's nonce will be invalidated after executing the order

InheritedStrategy

This contract handles the verification of parameters for standard transactions. It does not verify the taker struct's itemIds and amounts array as well as minPrice (taker ask) / maxPrice (taker bid) because before the taker executes the transaction and the maker itemIds/amounts/price should have already been confirmed off-chain.

A standard transaction (bid or ask) is mapped to strategyId = 0.

_verifyItemIdsAndAmountsEqualLengthsAndValidAmounts

```
function _verifyItemIdsAndAmountsEqualLengthsAndValidAmounts(uint256[] amounts, uint256[] itemIds)
```

This function is internal and is used to validate the parameters for a standard sale strategy when the standard transaction is initiated by a taker bid.

Parameters

Name	Type	Description
amounts	uint256[]	Array of amounts
itemIds	uint256[]	Array of item ids

*Last updated on **Aug 28, 2023***

LooksRareProtocol

This contract is the core smart contract of the LooksRare protocol ("v2"). It is the main entry point for users to initiate transactions with taker orders and manage the cancellation of maker orders, which exist off-chain.

WETH

```
address WETH
```

Wrapped ETH.

chainId

```
uint256 chainId
```

Current chainId.

domainSeparator

```
bytes32 domainSeparator
```

Current domain separator.

constructor

```
constructor(address _owner, address _protocolFeeRecipient, address _transferManager, address _weth
```

Constructor

Parameters

Name	Type	Description
_owner	address	Owner address
_protocolFeeRecipient	address	Protocol fee recipient address
_transferManager	address	Transfer manager address
_weth	address	Wrapped ETH address

executeTakerAsk

```
function executeTakerAsk(struct OrderStructs.Taker takerAsk, struct OrderStructs.Maker makerBid, b
```

This function allows a user to execute a taker ask (against a maker bid).

Parameters

Name	Type	Description
takerAsk	struct OrderStructs.Taker	Taker ask struct
makerBid	struct OrderStructs.Maker	Maker bid struct
makerSignature	bytes	Maker signature
merkleTree	struct OrderStructs.MerkleTree	Merkle tree struct (if the signature contains multiple maker orders)
affiliate	address	Affiliate address

executeTakerBid

```
function executeTakerBid(struct OrderStructs.Taker takerBid, struct OrderStructs.Maker makerAsk, b
```

This function allows a user to execute a taker bid (against a maker ask).

Parameters

Name	Type	Description
takerBid	struct OrderStructs.Taker	Taker bid struct
makerAsk	struct OrderStructs.Maker	Maker ask struct
makerSignature	bytes	Maker signature
merkleTree	struct OrderStructs.MerkleTree	Merkle tree struct (if the signature contains multiple maker orders)
affiliate	address	Affiliate address

executeMultipleTakerBids

```
function executeMultipleTakerBids(struct OrderStructs.Taker[] takerBids, struct OrderStructs.Maker
```

This function allows a user to batch buy with an array of taker bids (against an array of maker asks).

Parameters

Name	Type	Description
takerBids	struct OrderStructs.Taker[]	Array of taker bid structs
makerAsks	struct OrderStructs.Maker[]	Array of maker ask structs
makerSignatures	bytes[]	Array of maker signatures
merkleTrees	struct OrderStructs.MerkleTree[]	Array of merkle tree structs if the signature contains multiple maker orders
affiliate	address	Affiliate address
isAtomic	bool	Whether the execution should be atomic i.e. whether it should revert if 1 or more transactions fail

restrictedExecuteTakerBid

```
function restrictedExecuteTakerBid(struct OrderStructs.Taker takerBid, struct OrderStructs.Maker m
```

This function is used to do a non-atomic matching in the context of a batch taker bid.

This function is only callable by this contract. It is used for non-atomic batch order matching.

Parameters

Name	Type	Description
takerBid	struct OrderStructs.Taker	Taker bid struct
makerAsk	struct OrderStructs.Maker	Maker ask struct
sender	address	Sender address (i.e. the initial msg.sender)
orderHash	bytes32	Hash of the maker ask order

Return Values

Name	Type	Description
protocolFeeAmount	uint256	Protocol fee amount

updateDomainSeparator

```
function updateDomainSeparator() external
```

This function allows the owner to update the domain separator (if possible).

Only callable by owner. If there is a fork of the network with a new chainId, it allows the owner to reset the domain separator for the new chain id.

updateETHGasLimitForTransfer

```
function updateETHGasLimitForTransfer(uint256 newGasLimitETHTransfer) external
```

This function allows the owner to update the maximum ETH gas limit for a standard transfer.

Only callable by owner.

Parameters

Name	Type	Description
newGasLimitETHTransfer	uint256	New gas limit for ETH transfer

_executeTakerAsk

```
function _executeTakerAsk(struct OrderStructs.Taker takerAsk, struct OrderStructs.Maker makerBid,
```

This function is internal and is used to execute a taker ask (against a maker bid).

Parameters

Name	Type	Description
takerAsk	struct OrderStructs.Taker	Taker ask order struct
makerBid	struct OrderStructs.Maker	Maker bid order struct
orderHash	bytes32	Hash of the maker bid order

Return Values

Name	Type	Description
[0]	uint256	protocolFeeAmount Protocol fee amount

_executeTakerBid

```
function _executeTakerBid(struct OrderStructs.Taker takerBid, struct OrderStructs.Maker makerAsk,
```

This function is internal and is used to execute a taker bid (against a maker ask).

Parameters

Name	Type	Description
takerBid	struct OrderStructs.Taker	Taker bid order struct
makerAsk	struct OrderStructs.Maker	Maker ask order struct
sender	address	Sender of the transaction (i.e. msg.sender)
orderHash	bytes32	Hash of the maker ask order

Return Values

Name	Type	Description
[0]	uint256	protocolFeeAmount Protocol fee amount

_payProtocolFeeAndAffiliateFee

```
function _payProtocolFeeAndAffiliateFee(address currency, address bidUser, address affiliate, uint
```

This function is internal and is used to pay the protocol fee and affiliate fee (if any).

Parameters

Name	Type	Description
currency	address	Currency address to transfer (address(0) is ETH)
bidUser	address	Bid user address
affiliate	address	Affiliate address (address(0) if none)
totalProtocolFeeAmount	uint256	Total protocol fee amount (denominated in the currency)

_transferFungibleTokens

```
function _transferFungibleTokens(address currency, address sender, address recipient, uint256 amou
```

This function is internal and is used to transfer fungible tokens.

Parameters

Name	Type	Description
currency	address	Currency address
sender	address	Sender address
recipient	address	Recipient address
amount	uint256	Amount (in fungible tokens)

Last updated on Aug 28, 2023

NonceManager

This contract handles the nonce logic that is used for invalidating maker orders that exist off-chain. The nonce logic revolves around three parts at the user level: - order nonce (orders sharing an order nonce are conditional, OCO-like) - subset (orders can be grouped under a same subset) - bid/ask (all orders can be executed only if the bid/ask nonce matches the user's one on-chain) Only the order nonce is invalidated at the time of the execution of a maker order that contains it.

MAGIC_VALUE_ORDER_NONCE_EXECUTED

```
bytes32 MAGIC_VALUE_ORDER_NONCE_EXECUTED
```

Magic value nonce returned if executed (or cancelled).

userBidAskNonces

```
mapping(address => struct INonceManager.UserBidAskNonces) userBidAskNonces
```

This tracks the bid and ask nonces for a user address.

userOrderNonce

```
mapping(address => mapping(uint256 => bytes32)) userOrderNonce
```

This checks whether the order nonce for a user was executed or cancelled.

userSubsetNonce

```
mapping(address => mapping(uint256 => bool)) userSubsetNonce
```

This checks whether the subset nonce for a user was cancelled.

cancelOrderNonces

```
function cancelOrderNonces(uint256[] orderNonces) external
```

This function allows a user to cancel an array of order nonces.

It does not check the status of the nonces to save gas and to prevent revertion if one of the orders is filled in the same block.

Parameters

Name	Type	Description
orderNonces	uint256[]	Array of order nonces

cancelSubsetNonces

```
function cancelSubsetNonces(uint256[] subsetNonces) external
```

This function allows a user to cancel an array of subset nonces.

It does not check the status of the nonces to save gas.

Parameters

Name	Type	Description
subsetNonces	uint256[]	Array of subset nonces

incrementBidAskNonces

```
function incrementBidAskNonces(bool bid, bool ask) external
```

This function increments a user's bid/ask nonces.

The logic for computing the quasi-random number is inspired by Seaport v1.2. The pseudo-randomness allows non-deterministic computation of the next ask/bid nonce. A deterministic increment would make the cancel-all process non-effective in certain cases (orders signed with a greater ask/bid nonce). The same quasi-random number is used for incrementing both the bid and ask nonces if both values are incremented in the same transaction. If this function is used twice in the same block, it will return the same quasiRandomNumber but this will not impact the overall business logic.

Parameters

Name	Type	Description
bid	bool	Whether to increment the user bid nonce
ask	bool	Whether to increment the user ask nonce

StrategyManager

This contract handles the addition and the update of execution strategies.

strategyInfo

```
mapping(uint256 => struct IStrategyManager.Strategy) strategyInfo
```

This returns the strategy information for a strategy id.

constructor

```
constructor(address _owner) public
```

Constructor

Parameters

Name	Type	Description
_owner	address	Owner address

addStrategy

```
function addStrategy(uint16 standardProtocolFeeBp, uint16 minTotalFeeBp, uint16 maxProtocolFeeBp,
```

This function allows the owner to add a new execution strategy to the protocol.

Strategies have an id that is incremental. Only callable by owner.

Parameters

Name	Type	Description
standardProtocolFeeBp	uint16	Standard protocol fee (in basis point)
minTotalFeeBp	uint16	Minimum total fee (in basis point)
maxProtocolFeeBp	uint16	Maximum protocol fee (in basis point)
selector	bytes4	Function selector for the strategy
isMakerBid	bool	Whether the function selector is for maker bid
implementation	address	Implementation address

updateStrategy

```
function updateStrategy(uint256 strategyId, bool isActive, uint16 newStandardProtocolFee, uint16 n
```

This function allows the owner to update parameters for an existing execution strategy.

Only callable by owner.

Parameters

Name	Type	Description
strategyId	uint256	Strategy id
isActive	bool	Whether the strategy must be active
newStandardProtocolFee	uint16	New standard protocol fee (in basis point)
newMinTotalFee	uint16	New minimum total fee (in basis point)

TransferManager

This contract provides the transfer functions for ERC721/ERC1155 for contracts that require them. Collection type "0" refers to ERC721 transfer functions. Collection type "1" refers to ERC1155 transfer functions.

"Safe" transfer functions for ERC721 are not implemented since they come with added gas costs to verify if the recipient is a contract as it requires verifying the receiver interface is valid.

hasUserApprovedOperator

```
mapping(address => mapping(address => bool)) hasUserApprovedOperator
```

This returns whether the user has approved the operator address. The first address is the user and the second address is the operator (e.g. LooksRareProtocol).

isOperatorAllowed

```
mapping(address => bool) isOperatorAllowed
```

This returns whether the operator address is allowed by this contract's owner.

constructor

```
constructor(address _owner) public
```

Constructor

Parameters

Name	Type	Description
_owner	address	Owner address

transferItemsERC721

```
function transferItemsERC721(address collection, address from, address to, uint256[] itemIds, uint
```

This function transfers items for a single ERC721 collection.

Parameters

Name	Type	Description
collection	address	Collection address
from	address	Sender address
to	address	Recipient address
itemIds	uint256[]	Array of itemIds
amounts	uint256[]	Array of amounts

transferItemsERC1155

```
function transferItemsERC1155(address collection, address from, address to, uint256[] itemIds, uin
```

This function transfers items for a single ERC1155 collection.

It does not allow batch transferring if from = msg.sender since native function should be used.

Parameters

Name	Type	Description
collection	address	Collection address
from	address	Sender address
to	address	Recipient address
itemIds	uint256[]	Array of itemIds
amounts	uint256[]	Array of amounts

transferBatchItemsAcrossCollections

```
function transferBatchItemsAcrossCollections(struct ITransferManager.BatchTransferItem[] items, ad
```

This function transfers items across an array of collections that can be both ERC721 and ERC1155.

Parameters

Name	Type	Description
items	struct ITransferManager.BatchTransferItem[]	Array of BatchTransferItem
from	address	Sender address
to	address	Recipient address

grantApprovals

```
function grantApprovals(address[] operators) external
```

This function allows a user to grant approvals for an array of operators. Users cannot grant approvals if the operator is not allowed by this contract's owner.

Each operator address must be globally allowed to be approved.

Parameters

Name	Type	Description
operators	address[]	Array of operator addresses

revokeApprovals

```
function revokeApprovals(address[] operators) external
```

This function allows a user to revoke existing approvals for an array of operators.

Each operator address must be approved at the user level to be revoked.

Parameters

Name	Type	Description
operators	address[]	Array of operator addresses

allowOperator

```
function allowOperator(address operator) external
```

This function allows an operator to be added for the shared transfer system. Once the operator is allowed, users can grant NFT approvals to this operator.

Only callable by owner.

Parameters

Name	Type	Description
operator	address	Operator address to allow

removeOperator

```
function removeOperator(address operator) external
```

This function allows the user to remove an operator for the shared transfer system.

Only callable by owner.

Parameters

Name	Type	Description
operator	address	Operator address to remove

TransferSelectorNFT

This contract handles the logic for transferring non-fungible items.

transferManager

```
contract TransferManager transferManager
```

Transfer manager for ERC721 and ERC1155.

constructor

```
constructor(address _owner, address _protocolFeeRecipient, address _transferManager) public
```

Constructor

Parameters

Name	Type	Description
_owner	address	Owner address
_protocolFeeRecipient	address	Protocol fee recipient address
_transferManager	address	Address of the transfer manager for ERC721/ERC1155

_transferNFT

```
function _transferNFT(address collection, enum CollectionType collectionType, address sender, addr
```

This function is internal and used to transfer non-fungible tokens.

Parameters

Name	Type	Description
collection	address	Collection address
collectionType	enum CollectionType	Collection type (e.g. 0 = ERC721, 1 = ERC1155)
sender	address	Sender address
recipient	address	Recipient address
itemIds	uint256[]	Array of itemIds
amounts	uint256[]	Array of amounts

BaseStrategy

isLooksRareV2Strategy

```
function isLooksRareV2Strategy() external pure returns (bool)
```

This function acts as a safety check for the protocol's owner when adding new execution strategies.

Return Values

Name	Type	Description
[0]	bool	

_validateAmountNoRevert

```
function _validateAmountNoRevert(uint256 amount, enum CollectionType collectionType) internal pure
```

This is equivalent to if (amount == 0 || (amount != 1 && collectionType == 0)) { return (0, OrderInvalid.selector); } OrderInvalid_error_selector is a left-padded 4 bytes. If the error selector is returned instead of reverting, the error selector needs to be right-padded by 28 bytes. Therefore it needs to be left shifted by 28 x 8 = 224 bits.

© 2023 LooksRare. Built with Docusaurus.

BaseStrategyChainlinkMultiplePriceFeeds

This contract allows a strategy to store Chainlink price feeds for price retrieval.

priceFeeds

```
mapping(address => address) priceFeeds
```

This maps the collection address to a Chainlink price feed address. All prices returned from the price feeds must have 18 decimals.

PriceFeedUpdated

```
event PriceFeedUpdated(address collection, address priceFeed)
```

It is emitted when a collection's price feed address is added.

Parameters

Name	Type	Description
collection	address	NFT collection address
priceFeed	address	Chainlink price feed address

constructor

```
constructor(address _owner) public
```

Constructor

Parameters

Name	Type	Description
_owner	address	Owner address

setPriceFeed

```
function setPriceFeed(address _collection, address _priceFeed) external
```

This function allows the owner to set an NFT collection's Chainlink price feed address.

Only callable by owner. Once the price feed is set for a collection, it is not possible to adjust it after.

Parameters

Name	Type	Description
_collection	address	NFT collection address
_priceFeed	address	Chainlink price feed address

BaseStrategyChainlinkPriceLatency

This contract allows the owner to define the maximum acceptable Chainlink price latency.

maxLatency

```
uint256 maxLatency
```

Maximum latency accepted after which the execution strategy rejects the retrieved price.

```
    For ETH, it cannot be higher than 3,600 as Chainlink will at least update the
    price every 3,600 seconds, provided ETH's price does not deviate more than 0.5%.

    For NFTs, it cannot be higher than 86,400 as Chainlink will at least update the
    price every 86,400 seconds, provided ETH's price does not deviate more than 2%.
```

constructor

```
constructor(address _owner, uint256 _maxLatency) public
```

Constructor

Parameters

Name	Type	Description
_owner	address	Owner address
_maxLatency	uint256	Maximum price latency allowed

© 2023 LooksRare. Built with Docusaurus.

StrategyChainlinkFloor

These strategies allow a seller to make a floor price + premium ask and a buyer to make a floor price - discount collection bid. Currently Chainlink only has price feeds for ERC721 tokens, but these strategies can also support ERC1155 tokens as long as the trade amount is 1.

WETH

```
address WETH
```

Wrapped ether (WETH) address.

constructor

```
constructor(address _owner, address _weth) public
```

Constructor

Parameters

Name	Type	Description
_owner	address	Owner address
_weth	address	Address of WETH

executeFixedPremiumStrategyWithTakerBid

```
function executeFixedPremiumStrategyWithTakerBid(struct OrderStructs.Taker takerBid, struct OrderS
```

This function validates the order under the context of the chosen strategy and returns the fulfillable items/amounts/price/nonce invalidation status. This strategy looks at the seller's desired execution price in ETH (floor + premium) and minimum execution price and chooses the higher price.

The client has to provide the bidder's desired premium amount in ETH from the floor price as the additionalParameters.

Parameters

Name	Type	Description
takerBid	struct OrderStructs.Taker	Taker bid struct (taker bid-specific parameters for the execution)
makerAsk	struct OrderStructs.Maker	Maker ask struct (maker ask-specific parameters for the execution)

Return Values

Name	Type	Description
price	uint256	The final execution price
itemIds	uint256[]	The final item ids to be traded
amounts	uint256[]	The corresponding amounts for each item id. It should always be 1 for any collection type.
isNonceInvalidated	bool	Whether the order's nonce will be invalidated after executing the order

executeBasisPointsPremiumStrategyWithTakerBid

```
function executeBasisPointsPremiumStrategyWithTakerBid(struct OrderStructs.Taker takerBid, struct
```

This function validates the order under the context of the chosen strategy and returns the fulfillable items/amounts/price/nonce invalidation status. This strategy looks at the seller's desired execution price in ETH (floor * (1 + premium)) and minimum execution price and chooses the higher price.

The client has to provide the bidder's desired premium basis points from the floor price as the additionalParameters.

Parameters

Name	Type	Description
takerBid	struct OrderStructs.Taker	Taker bid struct (taker bid-specific parameters for the execution)
makerAsk	struct OrderStructs.Maker	Maker ask struct (maker ask-specific parameters for the execution)

Return Values

Name	Type	Description
price	uint256	The final execution price
itemIds	uint256[]	The final item ids to be traded
amounts	uint256[]	The corresponding amounts for each item id. It should always be 1 for any collection type.
isNonceInvalidated	bool	Whether the order's nonce will be invalidated after executing the order

executeFixedDiscountCollectionOfferStrategyWithTakerAsk

```
function executeFixedDiscountCollectionOfferStrategyWithTakerAsk(struct OrderStructs.Taker takerAs
```

This function validates the order under the context of the chosen strategy and returns the fulfillable items/amounts/price/nonce invalidation status. This strategy looks at the bidder's desired execution price in ETH (floor - discount) and maximum execution price and chooses the lower price.

The client has to provide the bidder's desired discount amount in ETH from the floor price as the additionalParameters.

Parameters

Name	Type	Description
takerAsk	struct OrderStructs.Taker	Taker ask struct (taker ask-specific parameters for the execution)
makerBid	struct OrderStructs.Maker	Maker bid struct (maker bid-specific parameters for the execution)

Return Values

Name	Type	Description
price	uint256	The final execution price
itemIds	uint256[]	The final item ids to be traded
amounts	uint256[]	The corresponding amounts for each item id. It should always be 1 for any collection type.
isNonceInvalidated	bool	Whether the order's nonce will be invalidated after executing the order

executeBasisPointsDiscountCollectionOfferStrategyWithTakerAsk

```
function executeBasisPointsDiscountCollectionOfferStrategyWithTakerAsk(struct OrderStructs.Taker t
```

This function validates the order under the context of the chosen strategy and returns the fulfillable items/amounts/price/nonce invalidation status. This strategy looks at the bidder's desired execution price in ETH (floor * (1 - discount)) and maximum execution price and chooses the lower price.

The client has to provide the bidder's desired discount basis points from the floor price as the additionalParameters.

Parameters

Name	Type	Description
takerAsk	struct OrderStructs.Taker	Taker ask struct (taker ask-specific parameters for the execution)
makerBid	struct OrderStructs.Maker	Maker bid struct (maker bid-specific parameters for the execution)

Return Values

Name	Type	Description
price	uint256	The final execution price
itemIds	uint256[]	The final item ids to be traded
amounts	uint256[]	The corresponding amounts for each item id. It should always be 1 for any collection type.
isNonceInvalidated	bool	Whether the order's nonce will be invalidated after executing the order

isMakerOrderValid

```
function isMakerOrderValid(struct OrderStructs.Maker makerOrder, bytes4 functionSelector) external
```

Validate *only the maker* order under the context of the chosen strategy. It does not revert if the maker order is invalid. Instead it returns false and the error's 4 bytes selector.

Parameters

Name	Type	Description
makerOrder	struct OrderStructs.Maker	Maker struct (maker specific parameters for the execution)
functionSelector	bytes4	Function selector for the strategy

Return Values

Name	Type	Description
isValid	bool	Whether the maker struct is valid
errorSelector	bytes4	If isValid is false, it returns the error's 4 bytes selector

_calculateFixedPremium

```
function _calculateFixedPremium(uint256 floorPrice, uint256 premium) internal pure returns (uint25
```

There is no fixed premium verification. We will let the runtime reverts the transaction if there is an overflow.

_calculateBasisPointsPremium

```
function _calculateBasisPointsPremium(uint256 floorPrice, uint256 premium) internal pure returns (
```

There is no basis points premium verification. We will let the runtime reverts the transaction if there is an overflow.

_calculateFixedDiscount

```
function _calculateFixedDiscount(uint256 floorPrice, uint256 discount) internal pure returns (uint
```

There is a fixed discount verification, unlike the premium calculation. This is not to catch an underflow, but to demonstrate the fact that there is a limit to the discount while the only limit to the premium is the size of uint256.

_calculateBasisPointsDiscount

```
function _calculateBasisPointsDiscount(uint256 floorPrice, uint256 discount) internal pure returns
```

There is a basis points discount verification (max 99%), unlike the premium calculation. This is not to catch an underflow, but to demonstrate the fact that there is a limit to the discount while the only limit to the premium is the size of uint256.

StrategyChainlinkUSDDynamicAsk

This contract allows a seller to sell an NFT priced in USD and the receivable amount to be in ETH.

ETH_USD_PRICE_FEED_DECIMALS

```
uint256 ETH_USD_PRICE_FEED_DECIMALS
```

It is possible to call priceFeed.decimals() to get the decimals, but to save gas, it is hard coded instead.

WETH

```
address WETH
```

Wrapped ether (WETH) address.

priceFeed

```
contract AggregatorV3Interface priceFeed
```

ETH/USD Chainlink price feed

constructor

```
constructor(address _owner, address _weth, address _priceFeed) public
```

Constructor

Parameters

Name	Type	Description
_owner	address	Owner address
_weth	address	Wrapped ether address
_priceFeed	address	Address of the ETH/USD price feed

executeStrategyWithTakerBid

```
function executeStrategyWithTakerBid(struct OrderStructs.Taker takerBid, struct OrderStructs.Maker
```

This function validates the order under the context of the chosen strategy and returns the fulfillable items/amounts/price/nonce invalidation status. This strategy looks at the seller's desired sale price in USD and minimum sale price in ETH, converts the USD value into ETH using Chainlink's price feed and chooses the higher price.

The client has to provide the seller's desired sale price in USD as the additionalParameters

Parameters

Name	Type	Description
takerBid	struct OrderStructs.Taker	Taker bid struct (taker bid-specific parameters for the execution)
makerAsk	struct OrderStructs.Maker	Maker ask struct (maker ask-specific parameters for the execution)

isMakerOrderValid

```
function isMakerOrderValid(struct OrderStructs.Maker makerAsk, bytes4 functionSelector) external v
```

Validate *only the maker* order under the context of the chosen strategy. It does not revert if the maker order is invalid. Instead it returns false and the error's 4 bytes selector.

Parameters

Name	Type	Description
makerAsk	struct OrderStructs.Maker	
functionSelector	bytes4	Function selector for the strategy

Return Values

Name	Type	Description
isValid	bool	Whether the maker struct is valid
errorSelector	bytes4	If isValid is false, it returns the error's 4 bytes selector

StrategyCollectionOffer

This contract offers execution strategies for users to create maker bid offers for items in a collection. There are two available functions: 1. executeCollectionStrategyWithTakerAsk --> it applies to all itemIds in a collection 2. executeCollectionStrategyWithTakerAskWithProof --> it allows adding merkle proof criteria. The bidder can only bid on 1 item id at a time. 1. If ERC721, the amount must be 1. 2. If ERC1155, the amount can be greater than 1.

Use cases can include trait-based offers or rarity score offers.

executeCollectionStrategyWithTakerAsk

```
function executeCollectionStrategyWithTakerAsk(struct OrderStructs.Taker takerAsk, struct OrderStr
```

This function validates the order under the context of the chosen strategy and returns the fulfillable items/amounts/price/nonce invalidation status. This strategy executes a collection offer against a taker ask order without the need of merkle proofs.

Parameters

Name	Type	Description
takerAsk	struct OrderStructs.Taker	Taker ask struct (taker ask-specific parameters for the execution)
makerBid	struct OrderStructs.Maker	Maker bid struct (maker bid-specific parameters for the execution)

executeCollectionStrategyWithTakerAskWithProof

```
function executeCollectionStrategyWithTakerAskWithProof(struct OrderStructs.Taker takerAsk, struct
```

This function validates the order under the context of the chosen strategy and returns the fulfillable items/amounts/price/nonce invalidation status. This strategy executes a collection offer against a taker ask order with the need of a merkle proof.

The transaction reverts if the maker does not include a merkle root in the additionalParameters.

Parameters

Name	Type	Description
takerAsk	struct OrderStructs.Taker	Taker ask struct (taker ask-specific parameters for the execution)
makerBid	struct OrderStructs.Maker	Maker bid struct (maker bid-specific parameters for the execution)

isMakerOrderValid

```
function isMakerOrderValid(struct OrderStructs.Maker makerBid, bytes4 functionSelector) external p
```

Validate *only the maker* order under the context of the chosen strategy. It does not revert if the maker order is invalid. Instead it returns false and the error's 4 bytes selector.

Parameters

Name	Type	Description
makerBid	struct OrderStructs.Maker	
functionSelector	bytes4	Function selector for the strategy

Return Values

Name	Type	Description
isValid	bool	Whether the maker struct is valid
errorSelector	bytes4	If isValid is false, it returns the error's 4 bytes selector

StrategyDutchAuction

This contract offers a single execution strategy for users to create Dutch auctions.

executeStrategyWithTakerBid

```
function executeStrategyWithTakerBid(struct OrderStructs.Taker takerBid, struct OrderStructs.Maker
```

This function validates the order under the context of the chosen strategy and returns the fulfillable items/amounts/price/nonce invalidation status. The execution price set by the seller decreases linearly within the defined period.

The client has to provide the seller's desired initial start price as the additionalParameters.

Parameters

Name	Type	Description
takerBid	struct OrderStructs.Taker	Taker bid struct (taker ask-specific parameters for the execution)
makerAsk	struct OrderStructs.Maker	Maker ask struct (maker bid-specific parameters for the execution)

isMakerOrderValid

```
function isMakerOrderValid(struct OrderStructs.Maker makerAsk, bytes4 functionSelector) external p
```

Validate *only the maker* order under the context of the chosen strategy. It does not revert if the maker order is invalid. Instead it returns false and the error's 4 bytes selector.

Parameters

Name	Type	Description
makerAsk	struct OrderStructs.Maker	
functionSelector	bytes4	Function selector for the strategy

Return Values

Name	Type	Description
isValid	bool	Whether the maker struct is valid
errorSelector	bytes4	If isValid is false, it returns the error's 4 bytes selector

StrategyItemIdsRange

This contract offers a single execution strategy for users to bid on a specific amount of items in a range bounded by 2 item ids.

executeStrategyWithTakerAsk

```
function executeStrategyWithTakerAsk(struct OrderStructs.Taker takerAsk, struct OrderStructs.Maker
```

This function validates the order under the context of the chosen strategy and returns the fulfillable items/amounts/price/nonce invalidation status. With this strategy, the bidder picks a item id range (e.g. 1-100) and a seller can fulfill the order with any tokens within the specified id range.

Parameters

Name	Type	Description
takerAsk	struct OrderStructs.Taker	Taker ask struct (taker ask-specific parameters for the execution)
makerBid	struct OrderStructs.Maker	Maker bid struct (maker bid-specific parameters for the execution)

isMakerOrderValid

```
function isMakerOrderValid(struct OrderStructs.Maker makerBid, bytes4 functionSelector) external p
```

Validate *only the maker* order under the context of the chosen strategy. It does not revert if the maker order is invalid. Instead it returns false and the error's 4 bytes selector.

Parameters

Name	Type	Description
makerBid	struct OrderStructs.Maker	
functionSelector	bytes4	Function selector for the strategy

Return Values

Name	Type	Description
isValid	bool	Whether the maker struct is valid
errorSelector	bytes4	If isValid is false, it returns the error's 4 bytes selector

LooksRare Aggregator Overview

The LooksRare aggregator is a decentralized system that streamlines the process of executing multiple orders from different sources in a gas-efficient manner within a single transaction. This system is built upon a set of Solidity smart contracts, with the main contract being the `LooksRareAggregator`.

One of the primary goals of the LooksRare aggregator is to minimize the gas consumption associated with executing multiple orders. To achieve this, the system takes an array of orders and delegates the execution to the source-specific proxy. This modular design also enables easy updates and improvements while maintaining the system's security and integrity.

This aggregation allows users to access a wide range of orders in a single platform enabling batch orders and partial fills in a gas effective manner, streamlining the trading experience and reducing the need to interact with multiple platforms individually.

As a decentralized system built on smart contracts, the LooksRare aggregator prioritizes security and transparency. The smart contracts are verified and have been audited, and adhere to the highest security standards. For more details on the contracts and audits see the sections below.

What is the core architecture?

The LooksRare aggregator consists of the `LooksRareAggregator` contract and a set of source-specific proxy contracts which delegate the execution to the relevant protocol. Alongside the `ERC20EnabledLooksRareAggregator`, which transfers the ERC20 tokens in the defined amount to the aggregator contract for orders that use ERC20 tokens instead of ETH as the currency.

The aggregator's core architecture consists of:

- **LooksRare Aggregator** (`LooksRareAggregator`) — an NFT liquidity aggregator that allows anyone to buy ERC721/ERC1155 tokens from multiple marketplaces in a single transaction. Users can only execute trades through proxies that have been whitelisted by the contract owner.
- **ERC20 LooksRare Aggregator** (`ERC20EnabledLooksRareAggregator`) — the entrypoint for purchases that are denominated in ERC20 tokens instead of native ETH. It still executes trades through `LooksRareAggregator`. The purpose of having an extra aggregator is to prevent malicious proxies from stealing users' ERC20 tokens in case LooksRareAggregator's ownership is compromised.
- **Proxies** (`LooksRareV2Proxy`, `SeaportProxy`) — the contracts which delegate the execution to the relevant protocol and handle partial filling, for non-atomic orders.

Deployed contracts

LooksRareAggregator

Mainnet: 📃 0x00000000005228B791a99a61f36A130d50600106

Goerli: 📃 0x00000000005228B791a99a61f36A130d50600106

ERC20EnabledLooksRareAggregator

Mainnet: 📃 0x0000000000a35231D7706BD1eE827d43245655aB

Goerli: 📃 0x0000000000a35231D7706BD1eE827d43245655aB

LooksRareV2Proxy (v2)

Mainnet: 📃 0x000000000074f2e99d7602fCA3cf0ffdCa906495

Goerli: 📃 0x5AAf7A47A96f4695b4c5F4d4706C04ae606FA59f

SeaportProxy (1.5)

Mainnet: 📃 0x000000000055d65008F1dFf7167f24E70DB431F6

Goerli: 📃 0x000000000055d65008F1dFf7167f24E70DB431F6

Audit Reports and Security Assurance

At LooksRare security is a top priority. The smart contracts have been rigorously audited by reputable third-party auditors to ensure the highest level of safety and reliability. These audits assess the contracts for potential vulnerabilities, and compliance with best practices in the blockchain industry.

We are committed to maintaining the highest security standards and to continuously improve our smart contract infrastructure responsibly. Below the audits reports for the LooksRare aggregator.

Code4rena

> Audit Report: https://code4rena.com/reports/2022-11-looksrare

Peckshield

> Audit Report: https://github.com/peckshield/publications/blob/master/audit_reports/PeckShield-Audit-Report-LooksRare-Aggregator-v1.0.pdf

Aggregator SDK

We made available an SDK to easily interact with the aggregator. The LooksRare Aggregator SDK is public and available on GitHub and as an NPM Package.

*Last updated on **Aug 28, 2023***

LooksRare Raffles Overview

🎟 LooksRare Raffles, Explained

The LooksRare Raffles system is a decentralized application built on the Ethereum blockchain, empowering users to create, manage and participate in transparent, on-chain raffles. Winners are chosen through a **provbably fair**, trustless mechanism — built with Chainlink's Verifiable Random Function (VRF). Jump to How are winners selected for more information on this.

Raffles can be hosted either by the LooksRare team, or by users. Both can be viewed and interacted with via looksrare.org, but anyone is free to spin up their own user interface anywhere else.

The system is designed to handle a wide range of tokens as prizes:

- Native **ETH**
- Fungible tokens (**ERC20**)
- Non-fungible tokens (**ERC721**)
- Semi-fungible tokens (**ERC1155**)

> ⓘ NOTE
> At present, Community Raffles (i.e., those created by users) only support ERC-721 NFTs. However, this will change in future iterations.

For more in-depth documentation of the Raffle system see LooksRare Raffles Smart Contract

Audit Report and Security Assurance

At LooksRare security is a top priority. The raffle smart contract has been rigorously audited by a reputable third-party auditor to ensure the highest level of safety and reliability. The audit assess the contract for potential vulnerabilities, and compliance with best practices in the blockchain industry.

We are committed to maintaining the highest security standards and to continuously improve our smart contract infrastructure responsibly. Below the audits reports for the LooksRare Raffle.

PeckShield

Audit Report Raffle V2:

> https://github.com/peckshield/publications/tree/master/audit_reports/PeckShield-Audit-Report-RaffleV1-v1.0.pdf

Audit Report Raffle V1:

> https://github.com/peckshield/publications/tree/master/audit_reports/PeckShield-Audit-Report-Raffle-v1.0.pdf

How to create and start a raffle

A raffle can be created by anyone via the Raffle smart contract by calling the `createRaffle` function.

When creating a raffle you can define multiple parameters like the cutoff time, the minimum number of entries required, the maximum number of entries per participant, the prizes and pricing options.

Before creating a raffle ensure all the necessary approvals to the Transfer Manager are sorted. The prizes will be transferred to the contract at creation.

How are winners selected?

Once a raffle has reached the minimum amount of entries, the system will send a request to Chainlink VRF requesting a provably fair and verifiable random number. For each request, Chainlink VRF generates one or more random values and cryptographic proof of how those values were determined.

The proof is published and verified on-chain. This process ensures that results cannot be tampered with or manipulated by any single entity including oracle operators, validators, users, or smart contract developers.

Methodology for Selecting Winners

Let's take an example where we have a total of 106 winners - one tier 0 prize, five tier 1 prizes, and one hundred tier 2 prizes.

1. We initiate the process by generating a random number using Chainlink's VRF.
2. This random number is then utilized to select the first winner, who receives the tier 0 prize.
3. To maintain randomness for subsequent selections, we then hash the initial random number using the `keccak256` function, producing a new random number.
4. The second winner is then chosen using this new random number, and they are awarded the tier 1 prize.
5. In an event where a selected entry has already won, then the seed is rehashed instead to find the next winning entry.
6. This process is repeated for each remaining prize, moving from lower to higher tiers (with lower tiers being more valuable), until all the prizes have been distributed.

How to cancel a raffle

A raffle can be canceled by if the minimum number of entries required for the raffle to be successful (`minimumEntries`) is not reached by the cutoff time (`cutoffTime`).

When a raffle is canceled, the owner can withdraw the prizes, while participants can request refunds for their entries.

Additionally, the contract owner has the ability to cancel a raffle in case a Chainlink VRF request (initiated during the raffle drawing process) gets stuck for more than a day.

Typically, a Chainlink VRF request takes only a few minutes, depending on network conditions. This provision serves as an extra safeguard.

LooksRare Subgraph Overview

What is a subgraph?

Using The Graph, a subgraph defines the data that is indexed from the Ethereum blockchain and how it is stored. Once it is deployed, it is used to form a part of a global graph of blockchain data.

> To learn more about subgraphs, read the introduction from The Graph.

Subgraphs can be queried using GraphQL.

How many subgraphs are deployed?

There are three subgraphs that can be used:

- **Exchange v1** (deprecated) — Built using data from events emitted by the LooksRare Exchange v1.
- **LOOKS Distribution** — Built using data from events emitted by several contracts, which all tie to the LOOKS token (e.g., airdrop contract, staking contracts).
- **Royalty Fee Registry** — Built using data from events emitted by the Royalty Fee Registry smart contract.

> These subgraphs are **expected to be modified in the future,** with changes being reflected on this documentation! As such, please be aware that future changes may break or remove currently implemented functions, etc.

How are subgraphs hosted?

Subgraphs are currently hosted on The Graph's "Hosted Service". In the future, these three subgraphs are expected to be deployed on The Graph's decentralized network that operates with curators, distributors, and indexers.

*Last updated on **Aug 28, 2023***

LooksRare Raffles Smart Contract

TL;DR: The raffle contract allows the creation of a new raffle with parameters such as cutoff time, minimum entries, maximum entries per participant, fees, and prizes. The prizes are deposited into the raffle at creation, and participants can enter the raffles by purchasing entries. Once the raffle is concluded, the winners are selected with the help of randomness provided by Chainlink VRF, after which the winners can claim their prizes. The contract also provides functionalities for the raffle owner to claim the collected fees and for the participants to withdraw or rollover their entry fees to any currently open raffles with the same fee currency in case the raffle is cancelled.

Functions

The contract includes various functions for interacting with the raffle system, such as creating a raffle (`createRaffle`), depositing prizes (`depositPrizes`), entering a raffle (`enterRaffles`), selecting winners (`selectWinners`), claiming prizes (`claimPrize(s)`), claiming fees (`claimFees`), cancelling a raffle (`cancel`), claiming a refund (`claimRefund`), rollover entries (`rollover`), setting the protocol fee (`setProtocolFeeBp`), setting the protocol fee recipient (`setProtocolFeeRecipient`), updating the status of currencies (`updateCurrenciesStatus`), toggling the contract's paused status (`togglePaused`), and various getter functions to retrieve information about raffles, such as the winners, pricing options, prizes, and entries.

Events

Events are emitted for various actions, such as buying an entry, refunding an entry, updating the raffle status, claiming prizes, claiming fees, sending a randomness request etc.

Subgraph

Our Raffle Subgraph indexes the events emitted by this contract.

More details about this subgraph can also be found in the Raffle Subgraph documentation.

Deployed contracts addresses (v2)

Chain	Raffle v2	Transfer Manager
Mainnet	`0x0000000000aDEaD599C11A0C9a7475B67852c1D0`	`0x00000000000ea4af05656C17b90f4d64AdD29e1d`
Goerli	`0x5A6451Eb148054a17262e56EF4e55203C36f095a`	`0xb737687983D6CcB4003A72731885454864Ecba9d`

Table of content

This page includes the following sections.

- Raffle System Overview
- Raffle Lifecycle
- Creating a raffle
- Entering a raffle
- Completing a raffle
- Selecting the winners
- Claiming the prizes
- Claiming the fees
- Cancelling a raffle
- Withdrawing the prizes
- Claiming a refund
- Rollover entries
- Collecting the protocol fees
- Read-only functions
- Raffle Object Data Types
- Other Data Types

Raffle System Overview

The Raffle system is designed to create and manage raffles on-chain and makes use of Chainlink's Verifiable Random Function (VRF) to provide a secure and provably fair mechanism for drawing winners.

The system allows users to create their own raffles *(currently only at the contract level)*. The system is also designed to handle a wide range of tokens as prizes: ETH, ERC20, ERC721 and ERC1155. A raffle can have at most 200 winners and a max of 200 prizes.

Here is a brief description of how it works:

1. **Raffle Creation**: A user (the raffle owner) creates a raffle by specifying the raffle parameters, such as the cut-off time, the minimum number of entries required to draw the raffle, the prize(s), the protocol fee, the pricing options for tickets, and the number of winners. The prizes are automatically deposited at this stage (ensure the Transfer Manager approvals are sorted). The raffle's status is set to `Open`.
2. **Raffle Entry**: Participants can enter the raffle by purchasing entries. The number of tickets purchased determines the participant's chances of winning.
3. **Raffle Completion**: Once a raffle's total purchased entries reach the minimum number of entries required, the raffle's status is set to `Drawing`, and the randomness request is sent. The raffle's owner has also the option to draw the winners if the minimum number of entries is not met after the cutoff time.
4. **Randomness Received**: Once Chainlink fulfills the randomness request, the raffle's status is set to `RandomnessFulfilled` and the winners can be selected.
5. **Select Winners**: The winners are selected using the randomness previously received by calling the `selectWinners` function. The raffle's status is set to `Drawn`.
6. **Claiming Prizes**: Once the status is set to `Drawn`, the winners can claim their prizes. Prizes are transferred securely from the contract to the winner's address.
7. **Claim Fees**: The raffle's owner can claim their fees once the winners have been selected. The raffle's status is set to `Complete` (winners can continue to claim their prizes). At this stage, the protocol fee is also calculated and added to the total `protocolFeeRecipientClaimableFees`.

Raffle Lifecycle

A raffle goes through several stages in its lifecycle:

1. **Open**: The raffle is initialized with the specified parameters, the prizes get deposited, and the raffle is open for participants to buy entries.
2. **Drawing**: The last entry has been sold, and the randomness request has been sent to Chainlink.
3. **RandomnessFulfilled**: Chainlink has fulfilled the randomness request.
4. **Drawn**: The winners of the raffle have been determined.
5. **Complete**: The raffle is set to `Complete` once the fees have been withdrawn.

- **Refundable**: The raffle was cancelled because it didn't reach the minimum number of entries before the `cutoffTime`, or if Chainlink did not fulfil the randomness request within the limit (one day). In this case, the owner can withdraw the prizes, and participants can request a refund for their entries.
- **Cancelled**: The raffle's owner withdrawn the prizes after canceling the raffle (based on the criteria outlined above). In this case, the participants can still request a refund for their entries (if applicable).

> INFO
>
> As long as there are more entries than prizes, the raffle's owner can draw the winners even tho the minimum entries are not met by calling `drawWinners(uint256 raffleId)`.

Creating a raffle

Before creating a raffle, ensure to sort all the approvals needed to deposit the prizes, by approving ERC tokens to be transferred by the Transfer Manager. It's also required to grant approval to the Raffle V2 contract to request the transfer via the Transfer Manager. That can be done by calling `grantApprovals()` with the Raffle V2 contract address as the parameter.

Raffles are created using the `createRaffle(CreateRaffleCalldata calldata params)` function. This function accepts a `CreateRaffleCalldata` struct as a parameter, which includes the following properties:

- `cutoffTime`: The time after which no more entries can be accepted for the raffle. Must be at least 1 day in the future and within 1 week.
- `isMinimumEntriesFixed`: Specifies whether the minimum number of entries also acts as a maximum number of entries or allows for an additional batch of entries to be purchased even if the total would exceed the `minimumEntries` amount.
- `minimumEntries`: The minimum number of entries required to draw the raffle.
- `maximumEntriesPerParticipant`: The maximum number of entries that a single participant can buy.
- `protocolFeeBp`: The protocol fee in basis points.
- `feeTokenAddress`: The address of the token to be used as a fee.
- `prizes`: An array of `Prize` structs representing the prizes to be distributed.
- `pricingOptions`: An array of `PricingOption` structs representing the pricing options for the raffle.

PricingOption

Each raffle can have between 1 to 5 pricing options. The `PricingOption` must respect the following rules:

1. The first pricing option's `entriesCount` (`pricingOption[0].entriesCount`), must divide the `minimumEntries` number evenly and be greater than zero.
2. The `entriesCount` of each `pricingOption` must be evenly divisible by `pricingOption[0].entriesCount`.
3. The price of all pricing options must be divisible by the number of entries it provides.
4. The entries given must increase as pricing options are added; they can't be less or equal to the previous `pricingOption` entries.
5. The price of the `pricingOption` must be greater than the previous `pricingOption`.
6. The price per entry must be lower than the previous `pricingOption`.

If the raffle is successfully created, the function will return the `raffleId` and the prizes defined will be transferred automatically.

Data Types

`CreateRaffleCalldata` is a struct used when creating a raffle, including the cutoff time, whether the minimum number of entries is a strict limit, the minimum entries, the maximum entries per participant, the protocol fee, the fee token address, the prizes, and the pricing options.

```
struct CreateRaffleCalldata {
    uint40 cutoffTime;
    bool isMinimumEntriesFixed;
    uint40 minimumEntries;
    uint40 maximumEntriesPerParticipant;
    uint16 protocolFeeBp;
    address feeTokenAddress;
    Prize[] prizes;
    PricingOption[] pricingOptions;
}
```

Events Emitted

- `RaffleStatusUpdated(raffleId, RaffleStatus.Open)`

Entering a raffle

Participants can enter a raffle using the `enterRaffles(EntryCalldata[] calldata entries)` function. This function accepts an array of `EntryCalldata` structs, each containing the `raffleId` and the pricing option index/id that the participant wishes to enter, alongside the count/amount being bought and the recipient address (optional, defaults to the `msg.sender`). The function checks if the raffle is still open, if the participant has not exceeded the maximum entries, and if they have enough balance to pay for the entries. If all checks pass, the entries are added to the raffle.

Data Types

`EntryCalldata` is a struct used when making entries into a raffle, containing the raffle ID and the index of the pricing option.

```
struct EntryCalldata {
    uint256 raffleId;
    uint256 pricingOptionIndex;
    uint40 count;
    address recipient;
}
```

Events Emitted

- `EntrySold(raffleId, msg.sender, recipient, entriesCount, price)`

Completing a raffle

When the last entry is sold (as defined in `raffle.minimumEntries`), the raffle status will be automatically set to `Drawing`, and a randomness request will be sent to Chainlink VRF. Once Chainlink has fulfilled the randomness request, the raffle's status will be set to `RandomnessFulfilled`.

Events Emitted

Once the randomness request is sent, the following events are emitted:

- `RaffleStatusUpdated(raffleId, RaffleStatus.Drawing)`
- `RandomnessRequested(raffleId, requestId)`

Once the randomness request has been fulfilled, the following events are emitted:

- `RaffleStatusUpdated(raffleId, RaffleStatus.RandomnessFulfilled)`

Selecting the winners

The winners are drawn using the `selectWinners(uint256 raffleId)` function, which accepts the `raffleId` as a parameter. It checks if the raffle is in the `Drawing` status and if Chainlink VRF has fulfilled the randomness request. If all checks pass, the function draws the winners for the raffle.

Methodology for Selecting Winners

Let's take an example where we have a total of 106 winners – one tier 0 prize, five tier 1 prizes, and one hundred tier 2 prizes.

1. We initiate the process by generating a random number using Chainlink's VRF.
2. This random number is then utilized to select the first winner, who receives the tier 0 prize.
3. To maintain randomness for subsequent selections, we then hash the initial random number using the `keccak256` function, producing a new random number.
4. The second winner is then chosen using this new random number, and they are awarded the tier 1 prize.
5. In an event where a selected entry has already won, then the seed is rehashed instead to find the next winning entry.
6. This process is repeated for each remaining prize, moving from lower to higher tiers (with lower tiers being more valuable), until all the prizes have been distributed.

Events Emitted

- `RaffleStatusUpdated(raffleId, RaffleStatus.Drawn)`

Claiming the prizes

The winners will be able to claim their prizes once the raffle's status is set to `Drawn` by calling either the `claimPrizes(ClaimPrizesCalldata[] calldata claimPrizesCalldata)` function or the `claimPrize(uint256 raffleId, uint256 winnerIndex)` function. The functions accepts an array of `ClaimPrizesCalldata` structs, each containing a `raffleId` and the index(es) of the prize(s) won (based upon the `raffle.winners` array) for the `claimPrizes` case or just the `raffleId` and `winnerIndex` in case of a single claim. If all the checks pass, the function caller will receive the prizes won. A winner can claim multiple prizes from multiple raffles in a single transaction.

Data Types

`ClaimPrizesCalldata` is a struct used when claiming prizes, including the `raffleId` and the indices of the winners (based on the `Winner[]` array defined in `Raffle` object, see Raffle Object Data Types).

```
struct ClaimPrizesCalldata {
    uint256 raffleId;
    uint256[] winnerIndices;
}
```

Events Emitted

- `PrizesClaimed(raffleId, winnerIndices)` if `claimPrizes` was used.
- `PrizeClaimed(raffleId, winnerIndex)` if `claimPrize` was used.

Claiming the fees

Once the raffle's status is set to `Drawn` either to raffle's owner or the protocol owner can call the `claimFees(uint256 raffleId)` function, which accepts the `raffleId` as a parameter. This function will transfer the raffle profits to the raffle's owner and calculate the protocol fee. The protocol fees won't be transferred at this time, but instead, the value will be added to the `protocolFeeRecipientClaimableFees[currency]` object. The protocol owner can then call the `claimProtocolFees(address currency)` at any time to collect the fees accrued so far.

Events Emitted

- `RaffleStatusUpdated(raffleId, RaffleStatus.Complete)`
- `FeesClaimed(raffleId, claimableFees)`

Cancelling a raffle

The raffle's owner can cancel the raffle if the minimum amount of entries has not been reached by the cutoff time (status `Open` and `timestamp > cutoffTime`). This can be done by calling the `cancel(uint256 raffleId)` function, which accepts the `raffleId` as a parameter. The contract's owner can also call `cancelAfterRandomnessRequest(uint256 raffleId)` if Chainlink does not fulfil the randomness request within the limit (one day).

Once one or more raffles are cancelled, the participants can get a refund of their entries, by calling `claimRefund(uint256[] calldata raffleIds)`.

After a raffle is cancelled, the status will be set to `Refundable` meaning the raffle's owner can withdraw the prizes by calling `withdrawPrizes(uint256 raffleId)`. Once the prizes have been withdrawn the status is ultimately set to `Cancelled`. The participants will be able to claim a refund or rollover the entries with either of those raffle statuses.

Events Emitted

- `RaffleStatusUpdated(raffleId, RaffleStatus.Refundable)` (cancelled, but prizes not yet withdrawn)
- `RaffleStatusUpdated(raffleId, RaffleStatus.Cancelled)` (cancelled and prizes withdrawn)

Withdrawing the prizes

Once the raffle has been cancelled and the status is set to `Refundable` the raffle's owner can withdraw the prizes deposited by calling `withdrawPrizes(uint256 raffleId)`, which accepts the `raffleId` as a parameter. This function will transfer all the deposited prizes back to the owner and set the raffle's status to `Cancelled`.

Events Emitted

- `RaffleStatusUpdated(raffleId, RaffleStatus.Cancelled)`

Claiming a refund

Once a raffle has been cancelled, the participants can request a refund for their entries by calling the `claimRefund(uint256[] calldata raffleIds)` function, which accepts an array of `raffleId`. It's possible to request refunds for multiple raffles in one call.

Events Emitted

- `EntryRefunded(raffleId, msg.sender, amountPaid)`

Rollover entries

Once a raffle has been cancelled, the participants can rollover their entries by calling `rollover(uint256[] calldata refundableRaffleIds, EntryCalldata[] calldata entries)`, which accepts an array of raffles ids (`refundableRaffleIds`) to be refunded and an array of `EntryCalldata` which are the entries to be bought in the new raffle. The fee token address for all the raffles involved **must be the same**.

Events Emitted

- `EntrySold(raffleId, msg.sender, recipient, entriesCount, price)`

Collecting the protocol fees

Protocol fees are collected on each raffle creation and are set as a percentage of the total value of the raffle. The protocol fee is set in basis points (bps), where 1 bps is equal to 0.01%. The protocol fee is collected in the token specified by `feeTokenAddress` during raffle creation.

The collected fees are stored in the contract and can be withdrawn by the contract owner to a specified address using the `claimProtocolFees(address currency)` function. The function checks if the caller is the contract owner and if there are any fees to withdraw. If all checks pass, the fees are transferred to the specified address.

Read-only functions

Function	Input	Output	Description
`rafflesCount`	`N/A`	`uint256`	The number of raffles created.
`raffles`	`raffleId`	`Raffle`	The raffle object, see Raffle Object Data Types.
`rafflesParticipantsStats`	`raffleId`, `address`	`ParticipantStats`	The participants stats of the raffles, see Other Data Types.
`isCurrencyAllowed`	`address`	`bool`	It checks whether the currency is allowed.
`randomnessRequests`	`requestId`	`RandomnessRequest`	The randomness requests, see Other Data Types.
`protocolFeeRecipient`	`N/A`	`address`	The protocol fee recipient.
`protocolFeeBp`	`N/A`	`uint16`	The protocol fee in basis points.
`getWinners`	`raffleId`	`Winner[]`	It calculates and returns the winners for a specific `raffleId` once the randomness request has been satisfied even if the function `selectWinners` hasn't been executed yet.
`getPrizes`	`raffleId`	`Prize[]`	It returns the raffle's prizes for a specific `raffleId`.
`getEntries`	`raffleId`	`Entry[]`	It returns the raffle's entries for a specific `raffleId`.
`getPricingOptions`	`raffleId`	`PricingOption[5]`	It returns the raffle's pricing options for a specific `raffleId`.

Raffle Object Data Types

`Raffle` is a struct representing a raffle, including the owner's address, the status of the raffle, the cutoff time, the drawn time, the minimum entries, the maximum entries per participant, the protocol fee, the claimable fees, the pricing options, the prizes, the entries, and the winners.

```
struct Raffle {
    address owner;
    RaffleStatus status;
    bool isMinimumEntriesFixed;
    uint40 cutoffTime;
    uint40 drawnAt;
    uint40 minimumEntries;
    uint40 maximumEntriesPerParticipant;
    address feeTokenAddress;
    uint16 protocolFeeBp;
    uint208 claimableFees;
    PricingOption[] pricingOptions;
    Prize[] prizes;
    Entry[] entries;
    Winner[] winners;
}
```

`RaffleStatus` is an enum with the different possible statuses of a raffle

```
enum RaffleStatus {
  None,
  Open,
  Drawing,
  RandomnessFulfilled,
  Drawn,
  Complete,
  Refundable,
  Cancelled,
}
```

`PricingOption` is a struct that defines a pricing option, including the number of entries that can be purchased for the given price and the price itself.

```
struct PricingOption {
    uint40 entriesCount;
    uint208 price;
}
```

PricingOption

Each raffle can have between 1 to 5 pricing options. The `PricingOption` must respect the following rules:

1. The first pricing option's `entriesCount` (`pricingOption[0].entriesCount`), must divide the `minimumEntries` number evenly and be greater than zero.
2. The `entriesCount` of each `pricingOption` must be evenly divisible by `pricingOption[0].entriesCount`.
3. The price of all pricing options must be divisible by the number of entries it provides.
4. The entries given must increase as pricing options are added; they can't be less or equal to the previous `pricingOption` entries.
5. The price of the `pricingOption` must be greater than the previous `pricingOption`.
6. The price per entry must be lower than the previous `pricingOption`.

`Prize` is a struct defining a prize, including the number of winners, the type of the prize, the tier of the prize, the address of the prize, the `prizeId`, and the amount of the prize.

```
struct Prize {
    uint40 winnersCount;
    uint40 cumulativeWinnersCount;
    TokenType prizeType;
    uint8 prizeTier;
    address prizeAddress;
    uint256 prizeId;
    uint256 prizeAmount;
}
```

`Entry` is a struct representing an entry in a raffle, including the participant's address and the cumulative number of entries in the raffle.

```
struct Entry {
  uint40 currentEntryIndex;
  address participant;
}
```

`Winner` is a struct representing a raffle winner, including the participant's address, whether the prize has been claimed, the index of the prize, and the index of the winning entry.

```
struct Winner {
    address participant;
    bool claimed;
    uint8 prizeIndex;
    uint40 entryIndex;
}
```

Other Data Types

`ParticipantStats` is a struct representing the statistics of a participant, including the amount paid, the number of entries, and whether the participant has been refunded.

```
struct ParticipantStats {
    uint208 amountPaid;
    uint40 entriesCount;
    bool refunded;
}
```

`RandomnessRequest` is a struct representing a randomness request, including whether the request exists, the raffleId, and the random words returned by Chainlink VRF.

```
struct RandomnessRequest {
    bool exists;
    uint80 raffleId;
    uint256 randomWord;
}
```

LooksRare Raffles Subgraph

This Subgraph is designed to facilitate interaction with the LooksRare raffles system. It enables you to efficiently query and manipulate data related to raffles, participants, entries, prizes, pricing, currencies, winners, and transactions that occur within the LooksRare raffle smart contract.

Raffle subgraph (Mainnet): https://thegraph.com/hosted-service/subgraph/looksrare/looksrare-raffle

Raffle subgraph (Goerli): https://thegraph.com/hosted-service/subgraph/looksrare/raffle

Schema

The main entities in the LooksRare Raffle Subgraph are:

- Raffle
- Participant
- Entry
- Prize
- Pricing
- Currency
- Winner
- Transaction

Entities Schema

More information about each field can also be found in the Raffle Smart Contract documentation.

RaffleVersion

The `RaffleVersion` is an enum. The options are `V1` (0) or `V2` (1).

```
enum RaffleVersion {
    v1
    v2
}
```

RaffleStatus

The `RaffleStatus` is an enum representing all the possible statuses of a raffle.

```
enum RaffleStatus {
    None
    Created  # Only on v1 raffles
    Open
    Drawing
    RandomnessFulfilled
    Drawn
    Complete
    Refundable
    Cancelled
    unknown
}
```

TokenType

The `TokenType` is an enum representing all supported token types.

```
enum TokenType {
    ERC721
    ERC1155
    ETH
    ERC20
    unknown
}
```

Raffle

A `Raffle` represents a unique raffle event in the system.

- **id** The unique identifier for the raffle (`ID!`).
- **raffleId** The `RaffleId` used in the smart contract (`BigInt!`).
- **currency** The `Currency` entity associated with the raffle (`Currency!`).
- **owner** The address of the raffle owner (`Bytes!`).
- **status** The current status of the raffle (`RaffleStatus!` enum).
- **lastStatusUpdate** The timestamp of the last status update (`BigInt!`).
- **statusLog** An array of `RaffleStatusLog` entities associated with the raffle (`[RaffleStatusLog!]!`).
- **drawAt** The timestamp when the winners have been determined (`BigInt`).
- **cutoffTime** The timestamp when the raffle is no longer accepting entries (`BigInt!`).
- **minimumEntries** The minimum number of entries required for the raffle to complete (`BigInt!`).
- **maximumEntriesPerParticipant** The maximum number of entries allowed per participant (`BigInt!`).
- **isMinimumEntriesFixed** Whether the raffle allows the total entries to exceed the minimum amount defined (`Boolean!`).
- **protocolFee** Total Prize Pool; the fair price in the Raffle's currency value of every prize added up (`BigDecimal!`).
- **currentPool** The current pool value of the raffle (`BigDecimal!`).
- **version** The raffle contract version (`RaffleVersion!`).
- **pricing** A list of `Pricing` entities (pricing options) for the raffle (`[Pricing!]!`).
- **prizes** A list of prizes associated with the raffle (`[Prize!]!`).
- **totalUsers** The total number of users who have entered the raffle (`BigInt!`).
- **totalTickets** The total number of tickets sold in the raffle (`BigInt!`).
- **tickets** A list of `Entry` entities for the raffle (`[Entry!]!`).
- **totalWinners** The total number of winners in the raffle (`BigInt!`).
- **winners** A list of `Winner` entities in the raffle (`[Winner!]`).
- **transactions** A list of `Transaction` entities associated with the raffle (`[Transaction!]`).

Currency

A `Currency` entity represents a currency used for raffle entries.

- **id** The currency's address (`ID!`).
- **name** The name of the currency (`String!`).
- **symbol** The currency's symbol (`String!`).
- **decimals** The number of decimal places for the currency (`BigInt!`).
- **isAllowed** Indicates whether the currency is allowed for raffle entries (`Boolean!`).

Pricing

A `Pricing` entity represents a pricing tier in a raffle.

- **id** The unique identifier for the pricing option (`ID!`).
- **raffle** The raffle associated with the pricing tier (`Raffle!`).
- **totalEntries** The number of entries granted by this pricing option (`BigInt!`).
- **price** The price for this pricing option (`BigDecimal!`).

Entry

An `Entry` represents a participant's entry into a raffle.

- **id** The unique identifier for the entry (`ID!`).
- **raffle** The `Raffle` entity associated with the entry (`Raffle!`).
- **participant** The `Participant` entity who purchased the entry (`Participant!`).
- **totalTickets** The total number of tickets purchased in this entry (`BigInt!`).
- **totalPrice** The total price paid for the entry (`BigDecimal!`).
- **refunded** Indicates whether the entry has been refunded (`Boolean!`).
- **winners** A list of `Winner` entities associated with this entry (`[Winner!]`).
- **transactions** A list of `Transaction` entities associated with this entry (`[Transaction!]`).

Participant

A `Participant` represents a user who participates in one or more raffles.

- **id** The participant's address (`ID!`).
- **totalRaffles** The total number of raffles the participant has entered (`BigInt!`).
- **totalTickets** The total number of tickets the participant has bought across all raffles (`BigInt!`).
- **tickets** A list of entries associated with the participant (`[Entry!]`).
- **winners** A list of winning entries associated with the participant (`[Winner!]`).
- **transactions** A list of transactions associated with the participant (`[Transaction!]`).

Transaction

A `Transaction` entity represents a transaction associated with a raffle entry.

- **id** The transaction's hash (`ID!`).
- **hash** The transaction hash (`Bytes!`).
- **raffle** The Raffle entity associated with the transaction (`Raffle!`).
- **participant** The Participant entity who initiated the transaction (`Participant!`).
- **entry** The Entry entity associated with the transaction (`Entry`).
- **entriesCount** The number of entries purchased in the transaction (`BigInt!`).
- **gasLimit** The gas limit for the transaction (`BigInt!`).
- **gasPrice** The gas price for the transaction (`BigDecimal!`).
- **block** The block number in which the transaction was included (`BigInt!`).
- **timestamp** The timestamp of the block containing the transaction (`BigInt!`).

Prize

A `Prize` represents a prize in a raffle.

- **id** The unique identifier for the prize (`ID!`).
- **prizeId** The prize's id in the smart contract (`BigInt!`).
- **raffle** The raffle associated with the prize (`Raffle!`).
- **tier** The prize tier (`BigInt!`).
- **type** The type of token used for the prize (`TokenType!` enum).
- **address** The address of the prize's contract (`Bytes`, nullable).
- **tokenId** The identifier of the specific token within the given contract (`BigInt`, nullable).
- **amount** The amount of the prize (`BigInt`, nullable).
- **totalWinners** The total number of winners for this prize (`BigInt!`).
- **totalWinnersCumulative** The cumulative number of winners for this prize tier and all previous tiers (`BigInt!`).

Winner

A `winner` entity represents a winning entry in a raffle.

- **id** The unique identifier for the winning entry (`ID!`).
- **raffle** The `Raffle` entity associated with the winning entry (`Raffle!`).
- **winnerIndex** The index of the winner in the winners array (`BigInt!`).
- **participant** The `Participant` entity with the details of the winner (`Participant!`).
- **entryIndex** The index of the winning entry in the entries array (`BigInt!`).
- **entry** The `Entry` entity associated with this entry (`Entry!`).
- **prizeIndex** The index of the prize won in the prizes array (`BigInt!`).
- **prize** The `Prize` entity with the details of the prize won by the winner (`Prize!`).
- **claimed** Indicates whether the prize has been claimed (`Boolean!`).
- **claimedHash** The transaction's hash of the prize claim event (`Bytes`).

RaffleStatusLog

A RoundStatusLog entity represents a log of a round's status change.

- **id** The log's id (`ID!`).
- **raffle** The `Raffle` entity associated with the status log (`Raffle!`).
- **status** The status of the raffle (`RaffleStatus!`).
- **transaction** The transaction hash that caused the status to change (`Bytes!`).
- **timestamp** The timestamp when the status was updated (`BigInt!`).
- **block** The block number when the status was updated (`BigInt!`).

Examples

These examples cover various use cases, including fetching raffle details, participant information, prizes, and more. Note that this is not an exhaustive list. You can build your own queries with the GraphiQL Explorer found on the Raffle Subgraph page.

Querying Raffles

Fetch a Raffle by ID

Get raffle details by ID, including status, cutoff time, minimum entries, and maximum entries per participant.

```
query {
  raffle(id: "1") {
    id
    status
    cutoffTime
    minimumEntries
    maximumEntriesPerParticipant
  }
}
```

List All Raffles

Retrieve a list of all raffles, including a pagination option.

```
query {
  raffles(first: 10, skip: 0) {
    id
    status
    cutoffTime
    minimumEntries
    maximumEntriesPerParticipant
  }
}
```

Querying Participants

Fetch a Participant by ID

Get participant details by ID, including total raffles, total tickets, and associated entries.

```
query {
  participant(id: "0x123...") {
    id
    totalRaffles
    totalTickets
    tickets {
      id
      raffle {
        id
      }
    }
  }
}
```

Querying Entries

List All Entries

Retrieve a list of all entries, including a pagination option.

```
query {
  entries(first: 10, skip: 0) {
    id
    raffle {
      id
    }
    participant {
      id
    }
    totalTickets
    totalPrice
  }
}
```

Fetch an Entry by ID

Get entry details by ID, including the raffle, participant, total tickets, and total price.

```
query {
  entry(id: "1-0x123...") {
    id
    raffle {
      id
    }
    participant {
      id
    }
    totalTickets
    totalPrice
  }
}
```

Querying Prizes

Fetch a Prize by ID

Get prize details by ID, including raffle, tier, type, token, and winners.

```
query {
  prize(id: "1-0") {
    id
    raffle {
      id
    }
    tier
    type
    tokenId
    address
    totalWinners
  }
}
```

Querying Transactions

Fetch a Transaction by ID

Get transaction details by ID, including hash, raffle, participant, entry, gas limit, gas price, block number, and block timestamp.

```
query {
  transaction(id: "0x123...") {
    id
    hash
    raffle {
      id
    }
    participant {
      id
    }
    entry {
      id
    }
    gasLimit
    gasPrice
    block
    timestamp
  }
}
```

Last updated on Aug 28, 2023

LooksRare YOLO Overview

Introducing YOLO

The YOLO system is designed to operate a transparent, fair, and decentralized game directly on the Ethereum blockchain. It utilizes smart contracts to handle the process, ensuring trust and integrity. The YOLO system is structured around a series of 'rounds'. Each round represents a single iteration of the game, where participants can enter by making deposits (`ETH`/`ERC-721`/`ERC-20`), and winners are selected at the end.

The likelihood of a participant winning is equivalent to the share of the total pot they deposited. Each deposit value is calculated, and each participant is allocated a number of entry spots proportional to the value deposited. The YOLO contract relies on a few external oracles: Chainlink for fair randomness, Reservoir for ERC-721 pricing data and Uniswap for ERC-20 pricing data.

NOTE: All NFTs in a collection will be valued at the floor price of the collection; rarity is not accounted for.

Frequently Asked Questions (FAQs)

1. How does the YOLO system handle participant deposits?

The value of each deposit from a participant is obtained either directly for `ETH` or via one of the external oracles for ERC-20/ERC-721. Once the value is determined, it gets divided by the `valuePerEntry` defined in the contract and the number of entries is then allocated to the depositor. This way, each depositor has a number of entries proportional to the total value of the assets deposited.

NOTE: All NFTs in a collection will be valued at the floor price of the collection; rarity is not accounted for.

2. How does a round completes?

A round completes when the time has elapsed and there are at least two participants, or when the maximum number of participants or deposits per round is reached; whichever occurs first.

3. How is the winner of a round selected?

The winner is selected based on a provably fair random number generated by the Chainlink oracle. This ensures that the drawing process is fair and resistant to manipulation.

For each request, Chainlink VRF generates a random values and cryptographic proof of how the value were determined. The proof is published and verified on-chain. This process ensures that results cannot be tampered with or manipulated by any single entity including oracle operators, validators, users, or smart contract developers.

The random number is used to select the winning entry, and the depositor associated with that entry is the winner of the round.

4. How does the YOLO drawing process ensure fairness?

The contract employs Chainlink's Verifiable Random Function (VRF) to generate a provably fair random number for a YOLO's round draw. This ensures that the process is fair, transparent, and tamper-proof.

For more details, see the Chainlink VRF Documentation.

5. How does the contract interact with external oracles?

The YOLO contract relies on 3 external oracles:

- The Chainlink oracle to request a provably fair random value.
- The Reservoir oracle to validate the collection floor price for an `ERC-721`.
- The Uniswap oracle to get the price of an `ERC-20`.

6. What is the lifecycle of a YOLO round?

A round starts in the `Open` status, where it can accept new deposits. Once the cutoff time is reached, the round progresses to the `Drawing` status while waiting for the Chainlink oracle to provide the randomness requested. Once Chainlink fulfils the request, the round moves to the `Drawn` status and a winner is declared. If certain conditions aren't met, the round can be `Cancelled`.

Reasons for a round being cancelled are:

- There were less than two participants when the cutoff time was reached.
- The randomness request sent to Chainlink has not been fulfilled within 1 day.
- As soon as a round is completed (`Drawn`) or cancelled (`Cancelled`) a new one will start.

7. What happens if a round does not have enough participants within the expected time frame?

If a round doesn't reach at least 2 distinct participants within the cutoff time, the round will be cancelled.

8. What happens when a round is cancelled?

If a round is cancelled, the system will immediately start a new round. The depositors will be able to withdraw their deposits on any cancelled round. Reasons for a round being cancelled are:

- There were less than two participants when the cutoff time was reached.
- The randomness request sent to Chainlink has not been fulfilled within 1 day.

9. How is the price for an ERC-20 token determined?

An ERC-20 token price is determined by relying on the Uniswap oracle.

The YOLO contract requests the TWAP (time-weighted average price) from Uniswap.

10. How is the price for an ERC-721 token determined?

The asset's price is determined by relying on the Reservoir oracle to validate the collection floor price on-chain. The necessary data can be retrieved via their API (see get collection floor with the configurations set to 1 day TWAP). Then the contract verifies the signature to ensure Reservoir provided the data and that the request is not expired (older than `signatureValidityPeriod`).

NOTE: All NFTs in a collection will be valued at the floor price of the collection; rarity is not accounted for.

11. Are the prices fetched at each deposit?

The price will be fetched only when a collection or currency (`tokenAddress`) is deposited for the first time. Then the same value will be used for the duration of the round if more of those assets are deposited.

12. Is the contract audited?

The YOLO smart contract has been rigorously audited by a reputable third-party auditor to ensure the highest level of safety and reliability. The audit assesses the contract for potential vulnerabilities and compliance with best practices in the blockchain industry.

Audit Report:

> https://github.com/peckshield/publications/tree/master/audit_reports/PeckShield-Audit-Report-LooksRare-YOLO-v1.0.pdf

Last updated on ***Aug 28, 2023***

LooksRare YOLO Smart Contract

TL;DR: The YOLO system is designed to operate a transparent, fair, and decentralized game directly on the Ethereum blockchain. It utilizes smart contracts to handle the process, ensuring trust and integrity. The YOLO system is structured around a series of 'rounds'. Each round represents a single iteration of the game, where participants can enter by making deposits (`ETH`/`ERC-721`/`ERC-20`), and winners are selected at the end.

The likelihood of a participant winning is equivalent to the share of the total pot they deposited. Each deposit value is calculated, and each participant is allocated a number of entry spots proportional to the value deposited. The YOLO contract relies on a few external oracles: Chainlink for fair randomness, Reservoir for ERC-721 pricing data and Uniswap for ERC-20 pricing data.

NOTE: All NFTs in a collection will be valued at the floor price of the collection; rarity is not accounted for.

Functions

The contract includes various functions for interacting with the YOLO system, such as depositing into the current round (`deposit`), drawing the winners (`drawWinner`), claiming the prizes (`claimPrizes`), cancelling a round if less than two addresses deposited (`cancel`), withdrawing the deposit if cancelled (`withdrawDeposits`) and cancelling a round (if less than two addresses entered) and depositing to a new one (`cancelCurrentRoundAndDepositToTheNextRound`).

Events

Events are emitted for various actions, such as depositing, withdrawing a deposit, updating the round status, claiming prizes, sending a randomness request etc.

Subgraph Our YOLO Subgraph indexes the events emitted by this contract.

More details about this subgraph can also be found in the YOLO Subgraph documentation.

Deployed contracts addresses

Contract	Mainnet	Goerli
YOLO	`0x00000000007767d79f9F4aA1Ff0d71b8E2E4a231`	`0x6D0aD9426525f771b41F70d30521a1149D`
TransferManager	`0x00000000000ea4af05656C17b90f4d64AdD29e1d`	`0xb737687983D6CcB4003A727318B5454864`
PriceOracle	`0x00000000000A95dBfC66D37F3FC5E597C0b03Daf`	`0xdaFdD4Fa5A8d87eFE22fF230dA4F5be887`

Table of content

This page includes the following sections.

- YOLO System Overview
- YOLO configuration
- Deposit assets
- Draw winner
- Claim prizes
- Cancel a round
- Withdraw deposits
- Variables update

YOLO System Overview

The YOLO system is designed to run rounds continuously based on the roundDuration value defined. Users can enter by depositing one or more types of assets (ETH/ERC-721/ERC-20) and receive entries in exchange, the round can successfully end by either reaching the maximum number of participants or deposits allowed per round (maximumNumberOfParticipants/maximumNumberOfDeposits), and there are at least two participants. Once that threshold is reached, a randomness request is sent to Chainlink and the value returned is used to randomly select the round winner. After the winner is selected, a new round is started.

Here is a brief description of how it works:

The contract is initialised with the rounds configurations: These values define things like the duration of each round (`roundDuration`), the limits on participants per round and deposits (`maximumNumberOfDepositsPerRound`/`maximumNumberOfParticipantsPerRound`) and the various oracles addresses and details. The contract owner can adjust these values.

1. **The round is started**: The status is set to `Open`. At this point, people can deposit assets and currencies and receive entries equivalent to the total value of the assets deposited. The Reservoir and Uniswap oracles are used to calculate the assets' value. The price is fetched only with the first deposit, and it will stay the same for the whole round.
2. **The round is successfully completed**: The status is set to `Drawing`. The randomness request is sent to Chainlink; this can take a few minutes to complete based on the network status.
   - **The randomness request is completed**: The status is set to `Drawn`. The value returned by Chainlink is used to select the winner of the last round.
3. **The round does not reach two participants**: The status is set to `Cancelled`. The deposits can be withdrawn by calling `withdrawDeposits` or rolled over to the next round when cancelling a round the current by calling `cancelCurrentRoundAndDepositToTheNextRound`.

After a round is completed or cancelled, a new one is started.

YOLO configuration

When the contract is first deployed, there are a set of parameters that are required within the constructor and govern the logic of each round. Most of these values can be updated by the contract owner.

Here is the `ConstructorCalldata` struct:

```
struct ConstructorCalldata {
    // The following values can be updated after the contract has been deployed
    address owner;
    address operator;
    uint40 maximumNumberOfDepositsPerRound;
    uint40 maximumNumberOfParticipantsPerRound;
    uint40 roundDuration;
    uint256 valuePerEntry;
    address protocolFeeRecipient;
    uint16 protocolFeeBp;
    address reservoirOracle;
    address erc20Oracle;
    uint40 signatureValidityPeriod;

    // The following values cannot be updated after the contract has been deployed
    bytes32 keyHash;
    uint64 subscriptionId;
    address vrfCoordinator;
    address weth;
    address transferManager;
}
```

Definitions

- `owner`: The contract's owner can update all the values outlined above except the currencies allowlist.
- `operator`: The contract's operator can update the allowed currencies.
- `maximumNumberOfDepositsPerRound`: The maximum number of deposits per round.
- `maximumNumberOfParticipantsPerRound`: The maximum number of participants per round.
- `roundDuration`: The duration of each round in seconds. It must be at most 1 hour.
- `valuePerEntry`: The value in ETH of one entry.
- `protocolFeeRecipient`: The address which will receive the protocol fee.
- `protocolFeeBp`: The protocol fee in bps, taken from the total value of a round. At most 2500 (25%).
- `reservoirOracle`: The address of Reservoir's price oracle. erc20Oracle: The address of Uniswap's price oracle.
- `signatureValidityPeriod`: The time in seconds that a given Reservoir signature will be valid for.
- `keyHash`/`subscriptionId`/`vrfCoordinator`: Chainlinks configuration variables, see Random Numbers: Using Chainlink VRF for more details.
- `weth`: The WETH address. Used for the protocol fee transfer function in case the ETH transfer fails.
- `transferManager`: The Transfer Manager contract's address. Used to transfer the assets. For more details, see the transfer manager contract interface. Approvals must be granted to this contract by the participants.

As soon as the contract is deployed, a round will start.

Events emitted

- `ERC20OracleUpdated(address erc20Oracle)`
- `MaximumNumberOfDepositsPerRoundUpdated(uint40 maximumNumberOfDepositsPerRound)`
- `MaximumNumberOfParticipantsPerRoundUpdated(uint40 maximumNumberOfParticipantsPerRound)`
- `ProtocolFeeBpUpdated(uint16 protocolFeeBp)`
- `ProtocolFeeRecipientUpdated(address protocolFeeRecipient)`
- `ReservoirOracleUpdated(address reservoirOracle)`
- `RoundDurationUpdated(uint40 roundDuration)`
- `SignatureValidityPeriodUpdated(uint40 signatureValidityPeriod)`
- `ValuePerEntryUpdated(uint256 valuePerEntry)`
- `RoundStatusUpdated(roundId, RoundStatus.Open)`

Deposit assets

Once a round has started, users can deposit one or more assets/currencies (ETH/ERC-20/ERC-721) by calling the function deposit(uint256 roundId, DepositCalldata[] calldata deposits). This function accepts a roundId and an array of DepositCalldata. Not needed for ETH deposits.

The DepositCalldata struct includes the following properties:

- `tokenType`: Either 1 for ERC-20 or 2 for ERC-721 (ETH does not need a DepositCalldata to be provided, the value sent alongside the transaction will be automatically deposited as ETH).
- `tokenAddress`: The asset's address.
- `tokenIdsOrAmounts`: The asset's ids for ERC-721 or the amount for ERC-20.
  - If the type is ERC-20 the tokenIdsOrAmounts length must be 1.
- `reservoirOracleFloorPrice`: An object of type ReservoirOracleFloorPrice that can be retrieved via the Reservoir API (see the section below for more details) with the following properties:
  - `id`: The response's id.
  - `payload`: The response's payload.
  - `timestamp`: The response's timestamp.
  - `signature`: The response's signature.

How to retrieve the floor price data via the Reservoir API

The signed collection's floor price data can be retrieved via the Reservoir API.

- Use the Collection floor endpoint.
- Set the kind to TWAP.
- Set the twapSeconds to 86400.
- Set the useNonFlaggedFloorAsk to false.

`DepositCalldata` struct:

```
struct ReservoirOracleFloorPrice {
    bytes32 id;
    bytes payload;
    uint256 timestamp;
    bytes signature;
}
```

`ReservoirOracleFloorPrice` struct:

```
struct DepositCalldata {
    TokenType tokenType;
    address tokenAddress;
    uint256[] tokenIdsOrAmounts;
    ReservoirOracleFloorPrice reservoirOracleFloorPrice;
}
```

NOTE: All NFTs in a collection will be valued at the floor price of the collection; rarity is not accounted for.

Events emitted

- `Deposited(msg.sender, roundId, totalEntriesCount)`
- `RandomnessRequested(roundId, requested)` if the deposit reached the limit per round
- `RoundStatusUpdated(roundId, RoundStatus.Drawing)` if the deposit reached the limit per round

Draw winner

Anyone can trigger for the winner to be drawn if the round's `cutoffTime` has been reached but not the maximum participants/deposits threshold, as long as there are at least 2 participants. This can be achieved by calling the function `drawWinner()`.

Events emitted

- `RandomnessRequested(roundId, requested)`
- `RoundStatusUpdated(roundId, RoundStatus.Drawing)`

Claim prizes

Once a round has been completed successfully, the winner can claim the prizes by calling the function `claimPrizes(ClaimPrizesCalldata[] calldata claimPrizesCalldata)`; it can be used to claim multiple rounds at once. The function accepts an array of `ClaimPrizesCalldata`.

The `ClaimPrizesCalldata` struct includes the following properties:

- `roundId`: The id of the round won.
- `prizeIndices`: An array of indices of prizes to claim.

`ClaimPrizesCalldata` struct:

```
struct ClaimPrizesCalldata {
    uint256 roundId;
    uint256[] prizeIndices;
}
```

NOTE: When calling the claim function, it might be required to send some ETH to cover the protocol fee, if the pot won does not have enough ETH prizes for that.

The function `getClaimPrizesPaymentRequired(ClaimPrizesCalldata[] calldata claimPrizesCalldata)` accepts the same input as the **claimPrizes** function, and it returns the amount of ETH to send along when claiming the prizes (this function does not validate **claimPrizesCalldata** for duplicated entries, **it's the caller responsibility to ensure correct usage to avoid incorrect values)**.

Events emitted

- `PrizesClaimed(perRoundClaimPrizesCalldata.roundId, msg.sender, prizeIndices)`

Cancel a round

Anyone can cancel a round if one of the following two conditions are met:

- The round's `cutoffTime` has been reached, and there are less than 2 participants.
- The randomness request to Chainlink has not been fulfilled within 1 day.

There are 3 different cancel functions:

- `cancel()`
- `cancelCurrentRoundAndDepositToTheNextRound(DepositCalldata[] calldata deposits)`
- `cancelAfterRandomnessRequest()`

The `cancelCurrentRoundAndDepositToTheNextRound` function requires a `DepositCalldata` array.

For more details on `DepositCalldata`, see the deposit assets section.

Events emitted

- `RoundStatusUpdated(roundId, RoundStatus.Cancelled)` current round cancelled
- `RoundStatusUpdated(roundId, RoundStatus.Open)` new round started
- `Deposited(msg.sender, roundId, totalEntriesCount)` if the deposit has been rolled over.

Withdraw deposits

After a round has been cancelled, the participants can withdraw their deposits by calling `withdrawDeposits(uint256 roundId, uint256[] calldata depositIndices)`.

The function accepts a `roundId` and an array of deposit indices.

Events emitted

- `DepositsWithdrawn(roundId, msg.sender, depositIndices)`

Variables update

The contract has a group of functions which are used to update some of the variables initially set at deployment. See YOLO configuration for more details.

Update Functions

Function Name	Function Input	Access control
`updateRoundDuration`	`uint40 _roundDuration`	`owner`
`updateSignatureValidityPeriod`	`uint40 _signatureValidityPeriod`	`owner`
`updateValuePerEntry`	`uint256 _valuePerEntry`	`owner`
`updateProtocolFeeRecipient`	`address _protocolFeeRecipient`	`owner`
`updateProtocolFeeBp`	`uint16 _protocolFeeBp`	`owner`
`updateMaximumNumberOfDepositsPerRound`	`uint40 _maximumNumberOfDepositsPerRound`	`owner`
`updateMaximumNumberOfParticipantsPerRound`	`uint40 _maximumNumberOfParticipantsPerRound`	`owner`
`updateReservoirOracle`	`address _reservoirOracle`	`owner`
`updateERC20Oracle`	`address _erc20Oracle`	`owner`
`updateCurrenciesStatus`	`address[] currencies, bool isAllowed`	`operator`

Last updated on ***Aug 28, 2023***

LooksRare YOLO Subgraph

Introduction

The LooksRare YOLO Subgraph is designed to index data from the YOLO contract deployed on the Ethereum network. The YOLO contract is a system where users can deposit different types of tokens (`ETH`/`ERC-20`/`ERC-721`) to participate in YOLO rounds.

The LooksRare YOLO Subgraph organizes its data into several core entities. These entities include:

- `YOLO`: Represents the overall YOLO state.
- `Round`: Contains information about a specific YOLO round.
- `Entry`: Describes a single user's participation in a round.
- `Deposit`: Represents a deposit made by a user in a round.
- `Transaction`: Represents an Ethereum transaction associated with a YOLO event.
- `RoundStatusLog`: Contains logs for the status changes of a round.

If you're unfamiliar with GraphQL, you may want to check out an Introduction to GraphQL.

The YOLO Subgraph is located at the following endpoint: YOLO Subgraph.

Entities Schema

More information about each field can also be found in the YOLO Smart Contract documentation.

Entry

An Entry entity represents an entry in a round.

- `id`: The entry's id (`ID!`).
- `depositor`: The address of the depositor (`Bytes!`).
- `round`: The Round entity associated with the entry (`Round!`).
- `totalNumberOfEntries`: The total number of entries for the depositor in the round (`BigInt!`).
- `deposits`: A list of Deposit entities associated with this entry (`[Deposit!]!`)

Round

A Round entity represents a round of the YOLO.

- `id`: The round's id (`ID!`).
- `roundId`: The `roundId` within the contract (`BigInt!`).
- `status`: The status of the round (`Int!`).
- `lastStatusUpdate`: The timestamp of the last status update (`BigInt`).
- `statusLog`: A list of RoundStatusLog entities associated with the round (`[RoundStatusLog!]!`).
- `winner`: The address of the round's winner (`Bytes`).
- `cutofftime`: The cutoff time for the round (`BigInt!`).
- `drawnAt`: The timestamp when the winner was drawn (`BigInt`).
- `drawnHash`: The transaction hash where the winner was drawn (`Bytes`).
- `numberOfParticipants`: The number of participants in the round (`BigInt!`).
- `maximumNumberOfDeposits`: The maximum number of deposits for the round (`BigInt!`).
- `maximumNumberOfParticipants`: The maximum number of participants for the round (`BigInt!`).
- `valuePerEntry`: The value per entry for the round (`BigInt!`).
- `numberOfEntries`: The number of entries in the round (`BigInt!`).
- `deposits`: A list of Deposit entities associated with the round (`[Deposit!]!`).
- `transactions`: A list of Transaction entities associated with the round (`[Transaction!]!`).

RoundStatusLog

A RoundStatusLog entity represents a log of a round's status change.

- `id`: The log's id (`ID!`).
- `round`: The Round entity associated with the status log (`Round!`).
- `status`: The status of the round (`Int!`).
- `transaction`: The transaction hash that caused the status to change (`Bytes!`).
- `timestamp`: The timestamp when the status was updated (`BigInt!`).
- `block`: The block number when the status was updated (`BigInt!`).

YOLO

A YOLO entity represents the overall YOLO game.

- `id`: The YOLO entity id (`ID!`).
- `totalRounds`: The total number of rounds in the YOLO (`BigInt!`).

Deposit

A Deposit entity represents a deposit made in a round.

- `id`: The deposit's id (`ID!`).
- `round`: The Round entity associated with the deposit (`Round!`).
- `entry`: The Entry entity associated with the deposit (`Entry!`).
- `transaction`: The Transaction entity associated with the deposit (`Transaction!`).
- `indice`: The index of the deposit (`BigInt!`).
- `tokenType`: The type of token deposited (`Int!`).
- `tokenAddress`: The address of the token (`Bytes!`).
- `tokenId`: The id of the token (`BigInt!`).
- `tokenAmount`: The amount of tokens deposited (`BigInt!`).
- `depositor`: The address of the depositor (`Bytes!`).
- `numberOfEntries`: The number of entries corresponding to the deposit (`BigInt!`).
- `claimed`: Indicates whether the deposit has been claimed (`Boolean!`).
- `withdrawn`: Indicates whether the deposit has been withdrawn (`Boolean!`).
- `claimedHash`: The transaction hash when the deposit was claimed (`Bytes`).

Transaction

A Transaction entity represents a transaction in a round.

- `id`: The transaction's hash (`ID!`).
- `hash`: The transaction's hash (`Bytes!`).
- `round`: The Round entity associated with the transaction (`Round!`).
- `depositor`: The address of the depositor (`Bytes!`).
- `deposits`: A list of Deposit entities associated with the transaction (`[Deposit!]!`).
- `totalNumberOfEntries`: The total number of entries associated with the transaction (`BigInt!`).
- `gasLimit`: The gas limit of the transaction (`BigInt!`).
- `gasPrice`: The gas price of the transaction (`BigInt!`).
- `block`: The block number when the transaction was made (`BigInt!`).
- `timestamp`: The timestamp when the transaction was made (`BigInt!`).

Examples

Querying YOLO total rounds

This query retrieves the `totalRounds` of the YOLO game.

```
{
  yolos {
    totalRounds
  }
}
```

Querying a Round

This query retrieves a specific `Round` by its id.

```
{
  round(id: "4") {
    id
    roundId
    status
    lastStatusUpdate
    cutofftime
    deposits {
      tokenType
      depositor
      tokenAddress
      tokenAmount
    }
    numberOfParticipants
    maximumNumberOfDeposits
    maximumNumberOfParticipants
    valuePerEntry
    numberOfEntries
    drawnAt
    winner
  }
}
```

Querying an Entry

This query retrieves a specific `Entry` by its id.

```
{
  entry(id: "4-0x992c659e3a3a2a7902c145441c77e8c216a1332f") {
    id
    depositor
    round {
      roundId
      drawnAt
      cutofftime
    }
    totalNumberOfEntries
  }
}
```

Querying a Deposit

This query retrieves a specific `Deposit` by its id.

```
{
  deposit(id: "4-2") {
    id
    round {
      roundId
      lastStatusUpdate
    }
    entry {
      id
      totalNumberOfEntries
    }
    transaction {
      hash
    }
    indice
    tokenType
    tokenAddress
    tokenId
    tokenAmount
    depositor
    numberOfEntries
    claimed
    withdrawn
    claimedHash
  }
}
```

Querying a Transaction

This query retrieves a specific `Transaction` by its id.

```
{
  transaction(
    id: "0x06332cd7009ac080ddb6af6d86e2a07560c667c6971d29e89e5481edc43d38fa"
  ) {
    id
    hash
    round {
      roundId
    }
    depositor
    totalNumberOfEntries
    gasLimit
    gasPrice
    block
    timestamp
  }
}
```

Querying a RoundStatusLog

This query retrieves a specific `RoundStatusLog` by its id.

```
{
  roundStatusLog(id: "4-2") {
    id
    round {
      roundId
    }
    status
    transaction
    timestamp
    block
  }
}
```

Last updated on Aug 28, 2023

Public API Documentation

IMPORTANT INFORMATION

LooksRare API is currently in BETA and will be undergoing regular maintenance to increase data offerings and improve performance.

LooksRare provides the API free of charge, and ask that you provide attribution to LooksRare on your platform (website and/or app) in return.

There should be a link to LooksRare from the NFTs or relative data that you display from the LooksRare API.

Please refer to Brand, Logos & Usage page for images that you can use for LooksRare attribution.

LooksRare reserves the right to revoke or restrict API access in the event that the above terms are not adhered to.

Public API Documentation

LooksRare Public API Docs: https://looksrare.dev/

General FAQs

What chains are available via the API?

Ethereum Mainnet for production and *Goerli* for development.

> Mainnet: `https://api.looksrare.org/api/`
>
> Goerli: `https://api-goerli.looksrare.org/api/`

Do I need an API key?

A key is only required for requests sent to mainnet. For testing and development you can use our Goerli API without needing an API key.

Is there a cache on the API?

Yes, we currently have a 10-second API cache in place.

Can I request a higher limit?

It is possible to request a higher rate limit; however, this process requires additional information, and further requirements must be met for the request to be successful. It is highly unlikely that we will provide higher rate limits for applications still under development.

To request a higher limit, please see Can I contact LooksRare about the API?

Can I create listings through the API?

Yes, you can create listing via the Public API (see https://looksrare.dev/reference/createorder). An API key will be required on mainnet.

How can I obtain an API Key?

Please see Can I contact LooksRare about the API?

Does LooksRare have an SDK?

Yes, we provide an SDK to enhance the developers' experience.

> LooksRare SDK v2: https://github.com/LooksRare/sdk-v2

Can I contact LooksRare about the API?

Yes, we are always happy to help. We have a Developers Discord available at discord.gg/LooksRareDevelopers

*Last updated on **Aug 28, 2023***

LooksRare SDK Cancel Orders

⚠️ These code snippets are just examples and the data should never be used as is ⚠️

How to cancel orders

Cancel orders using order nonces

This method is used to invalidate the `orderNonce`. If multiple maker orders share the same nonce, they will all be cancelled and non-executable. An `orderNonce` is also invalidated once an order with that nonce is executed.

```
import { LooksRare, ChainId } from "@looksrare/sdk-v2";

const lr = new LooksRare(ChainId.MAINNET, provider, signer);

// Cancel order nonce 0
const tx = await lr.cancelOrders([0]).call();
const receipt = await tx.wait();

// Cancel order nonce 0 and 12
const tx = await lr.cancelOrders([0, 12]).call();
const receipt = await tx.wait();
```

Cancel orders using subset nonces

The `subsetNonce` can only be invalidated/cancelled manually. The purpose of the `subsetNonce` is to allow the user to arbitrarily group different orders under the same `subsetNonce`, for example, all the orders from the same collection could have the same `subsetNonce`, allowing for a more targeted cancel functionality.

This is pending implementation within our frontend, it should only be used by users who have a thorough understanding of its functionality.

Important to note

If you cancel the `subsetNonce` with a value of 0, that address will be unable to create orders via our website as this feature is currently not implemented and orders are only created with the `subsetNonce` set to 0. If you plan to make use of the `subsetNonce` functionalities, consider using a value greater than 0.

```
import { LooksRare, ChainId } from "@looksrare/sdk-v2";

const lr = new LooksRare(ChainId.MAINNET, provider, signer);

// Cancel order with the subset nonce 1
const tx = await lr.cancelSubsetOrders([1]).call();
const receipt = await tx.wait();

// Cancel order with the subset 1 and 12
const tx = await lr.cancelSubsetOrders([1, 12]).call();
const receipt = await tx.wait();
```

Cancel all your bids and/or all your asks

This function can be used to cancel all the sender's bids or all the sender's asks, or both in a single call. The following example showcases all the possible combinations.

```
import { LooksRare, ChainId } from "@looksrare/sdk-v2";

const lr = new LooksRare(ChainId.MAINNET, provider, signer);

// Cancel all bids
const tx = await lr.cancelAllOrders(true, false).call();
const receipt = await tx.wait();

// Cancel all asks
const tx = await lr.cancelAllOrders(false, true).call();
const receipt = await tx.wait();

// Cancel all bids and asks
const tx = await lr.cancelAllOrders(true, true).call();
const receipt = await tx.wait();
```

For more details on our triple-nonce system, see the related documentation: Triple-nonce system

Need help?

You can reach out to the LooksRare team via our Developers Discord: https://discord.gg/LooksRareDevelopers

LooksRare SDK Collection Order

⚠️ These code snippets are just examples and the data should never be used as is ⚠️

How to create a collection offer (maker bid with collection strategy)

The code snippet below is an example of how to create a maker bid using the `@looksrare/sdk-v2` library.

> A collection order is just a maker bid order using the `StrategyType.collection`.

The main steps are:

1. Initialize a LooksRare class instance by providing the chain id, RPC provider and a signer.
2. Use the `createMakerCollectionOffer` method to create a maker bid with the parameters of your order.
3. Check and grant necessary approvals for transferring assets.
4. Sign the maker bid order with `signMakerOrder` method.

> The `orderNonce` has to be retrieved via our Public API, see get order nonce.

Here is an example:

```
import { parseEther } from "ethers";
import { LooksRare, ChainId, CollectionType, StrategyType } from "@looksrare/sdk-v2";

const lr = new LooksRare(ChainId.MAINNET, provider, signer);

const { maker, isCurrencyApproved, isBalanceSufficient } = await lr.createMakerCollectionOffer({
  collection: "0x0000000000000000000000000000000000000000", // Collection address
  collectionType: CollectionType.ERC721,
  subsetNonce: 0, // keep 0 if you don't know what it is used for
  orderNonce: 0, // You need to retrieve this value from the API
  endTime: Math.floor(Date.now() / 1000) + 86400, // If you use a timestamp in ms, the function wi
  price: parseEther("1"), // Be careful to use a price in wei, this example is for 1 ETH
  amounts: [1], // Use it for listing multiple ERC-1155 (Optional, Default to [1])
  startTime: Math.floor(Date.now() / 1000), // Use it to create an order that will be valid in the
});

// Approve spending of the currency used for bidding
if (!isCurrencyApproved) {
  const tx = await lr.approveErc20(lr.addresses.WETH);
  await tx.wait();
}

// Checks if the WETH balance is enough to cover the bid
if (!isBalanceSufficient) {
  throw new Error(`WETH balance too low.`);
}

// Sign your maker order
const signature = await lr.signMakerOrder(maker);
```

> Once, the maker bid for your collection offer has been created, the approvals sorted and the order signed, you will have to send it along with the signature to the `POST /api/v2/orders` endpoint. For more details and examples, see create order).

How to execute a collection offer

`createTakerForCollectionOrder` is just a convenient wrapper around `createTaker`.

```
import { LooksRare, ChainId } from "@looksrare/sdk-v2";

const lr = new LooksRare(ChainId.MAINNET, provider, signer);

// To be done only once the first a user is interacting with the V2.
// It will grant the Exchange contract with the right to use your collections approvals done on th
await lr.grantTransferManagerApproval().call();
await setApprovalForAll(signer, maker.collection, lr.addresses.TRANSFER_MANAGER_V2);

const taker = lr.createTakerForCollectionOrder(maker, TOKEN_ID); // Change the token id

const { call } = lr.executeOrder(maker, taker, signature);
const tx = await call();
const receipt = await tx.wait();
```

Need help?

You can reach out to the LooksRare team via our Developers Discord: https://discord.gg/LooksRareDevelopers

LooksRare SDK Create MakerAsk

⚠️ These code snippets are just examples and the data should never be used as is ⚠️

How to create a maker ask order

The code snippet below is an example of how to create a maker ask using the `@looksrare/sdk-v2` library.

The main steps are:

1. Initialize a LooksRare class instance by providing the chain id, RPC provider and a signer.
2. Use the `createMakerAsk` method to create a maker ask with the parameters of your order.
3. Check and grant necessary approvals for transferring assets.
4. Sign the maker ask order with `signMakerOrder` method.

> The `orderNonce` has to be retrieved via our Public API, see get order nonce.

Here is an example:

```
import { parseEther } from "ethers";
import { LooksRare, ChainId, CollectionType, StrategyType } from "@looksrare/sdk-v2";

const lr = new LooksRare(ChainId.MAINNET, provider, signer);

const { maker, isCollectionApproved, isTransferManagerApproved } = await lr.createMakerAsk({
  collection: "0x0000000000000000000000000000000000000000", // Collection address
  collectionType: CollectionType.ERC721,
  strategyId: StrategyType.standard,
  subsetNonce: 0, // keep 0 if you don't know what it is used for
  orderNonce: 0, // You need to retrieve this value from the API
  endTime: Math.floor(Date.now() / 1000) + 86400, // If you use a timestamp in ms, the function wi
  price: parseEther("1"), // Be careful to use a price in wei, this example is for 1 ETH
  itemIds: [0], // Token id of the NFT(s) you want to sell, add several ids to create a bundle
  amounts: [1], // Use it for listing multiple ERC-1155 (Optional, Default to [1])
  startTime: Math.floor(Date.now() / 1000), // Use it to create an order that will be valid in the
});

// Grant the TransferManager the right the transfer assets on behalf od the LooksRareProtocol
if (!isTransferManagerApproved) {
  const tx = await lr.grantTransferManagerApproval().call();
  await tx.wait();
}

// Approve the collection items to be transferred by the TransferManager
if (!isCollectionApproved) {
  const tx = await lr.approveAllCollectionItems(maker.collection);
  await tx.wait();
}

// Sign your maker order
const signature = await lr.signMakerOrder(maker);
```

> Once, the maker ask for your collection offer has been created, the approvals sorted and the order signed, you will have to send it along with the signature to the `POST /api/v2/orders` endpoint. For more details and examples, see create order).

Need help?

You can reach out to the LooksRare team via our Developers Discord: https://discord.gg/LooksRareDevelopers

LooksRare SDK Create MakerBid

⚠️ These code snippets are just examples and the data should never be used as is ⚠️

How to create a maker bid order

The code snippet below is an example of how to create a maker bid using the `@looksrare/sdk-v2` library.

The main steps are:

1. Initialize a LooksRare class instance by providing the chain id, JSON-RPC provider and a signer.
2. Use the `createMakerBid` method to create a maker bid with the parameters of your order.
3. Check and grant necessary approvals for transferring assets.
4. Sign the maker bid order with `signMakerOrder` method.

> The `orderNonce` has to be retrieved via our Public API, see get order nonce.

Here is an example:

```
import { ethers } from "ethers";
import { LooksRare, ChainId, CollectionType, StrategyType } from "@looksrare/sdk-v2";

const lr = new LooksRare(ChainId.MAINNET, provider, signer);

const { maker, isCurrencyApproved, isBalanceSufficient } = await lr.createMakerBid({
  collection: "0x0000000000000000000000000000000000000000", // Collection address
  collectionType: CollectionType.ERC721,
  strategyId: StrategyType.standard,
  subsetNonce: 0, // keep 0 if you don't know what it is used for
  orderNonce: 0, // You need to retrieve this value from the API
  endTime: Math.floor(Date.now() / 1000) + 86400, // If you use a timestamp in ms, the function wi
  price: parseEther("1"), // Be careful to use a price in wei, this example is for 1 ETH
  itemIds: [0], // Token id of the NFT you want to buy
  amounts: [1], // Use it for listing several ERC-1155 (Optional, Default to [1])
  startTime: Math.floor(Date.now() / 1000), // Use it to create an order that will be valid in the
});

// Approve spending of the currency used for bidding
if (!isCurrencyApproved) {
  const tx = await lr.approveErc20(lr.addresses.WETH);
  await tx.wait();
}

// Checks if the WETH balance is enough to cover the bid
if (!isBalanceSufficient) {
  throw new Error(`WETH balance too low.`);
}

// Sign your maker order
const signature = await lr.signMakerOrder(maker);
```

> Once, the maker bid for your collection offer has been created, the approvals sorted and the order signed, you will have to send it along with the signature to the `POST /api/v2/orders` endpoint. For more details and examples, see create order).

Need help?

You can reach out to the LooksRare team via our Developers Discord: https://discord.gg/LooksRareDevelopers

LooksRare SDK Execute Trade

⚠️ These code snippets are just examples and the data should never be used as is ⚠️

How to create a Taker order and execute a trade

Trades are executed on-chain by matching a `Maker` order with a `Taker` order. The maker order can be retrieved from the API, see get v2 orders for more details. While the taker order can be obtained by calling the `createTaker` method as shown here:

```
import { LooksRare, ChainId } from "@looksrare/sdk-v2";

const lr = new LooksRare(ChainId.MAINNET, provider, signer);

// The recipient address is optional, if you don't provide it will use your signer address
const takerOrder = lr.createTaker(makerOrder, recipientAddress);
```

From the API response, you will also get the `signature`, which is necessary to execute the trade on-chain. To execute the trade, you can call the `executeOrder` method passing the `Maker`, `Taker` and the `signature`. The method will return a contract call. Here is an example:

```
const { call } = lr.executeOrder(makerOrder, takerOrder, signature);
const tx = await call();
const receipt = await tx.wait();
```

Need help?

You can reach out to the LooksRare team via our Developers Discord: https://discord.gg/LooksRareDevelopers

LooksRare SDK Check Validity

⚠️ These code snippets are just examples and the data should never be used as is ⚠️

Verify order validity

The LooksRare SDK also provides a method to validate your order. It can be used as follows:

```
import { LooksRare, ChainId } from "@looksrare/sdk-v2";

const lr = new LooksRare(ChainId.MAINNET, provider, signer);

const validatorCodes = await lr.verifyMakerOrders([makerOrder], [signature]);
```

To see all the possible validation codes, see the `OrderValidatorCode` enum located in src/types.ts.

Need help?

You can reach out to the LooksRare team via our Developers Discord: https://discord.gg/LooksRareDevelopers

LooksRare SDK Merkle Tree Orders

⚠️ These code snippets are just examples and the data should never be used as is ⚠️

Sign multiple maker orders with a single signature

> **This functionality is for UI implementations only. If you are using a bot, don't use the `signMultipleMakerOrders` function, just loop over the `createMakerBid` and `createMakerAsk` functions.**

The code snippet below is an example of how to sign multiple orders with one signature using Merkle trees via the `@looksrare/sdk-v2` library.

NOTE: In this example we only used maker asks, for maker bids the approvals logic needs to be adjusted. See the documentation on creating maker bids for more details.

```
import { ethers } from "ethers";
import { LooksRare, ChainId, CollectionType, StrategyType } from "@looksrare/sdk-v2";

const lr = new LooksRare(ChainId.MAINNET, provider, signer);

const orders = [];

orders.push(await lr.createMakerAsk(...));
orders.push(await lr.createMakerAsk(...));

// Grant the TransferManager the right the transfer assets on behalf od the LooksRareProtocol. Onl
if (!orders[0].isTransferManagerApproved) {
    const tx = await lr.grantTransferManagerApproval().call();
    await tx.wait();
}

for (const order of orders) {
    // Approve the collection items to be transferred by the TransferManager
    if (!order.isCollectionApproved) {
        const tx = await lr.approveAllCollectionItems(order.maker.collection);
        await tx.wait();
    }
}

const { signature, merkleTreeProofs } = await lr.signMultipleMakerOrders(orders);
```

> The maker orders, merkleTreeProofs and signature will have be sent to the `POST /api/v2/orders/tree` endpoint. For more details and examples, see create Merkle tree order.

For more information on how to create your orders, see the createMakerAsk and createMakerBid documentation.

Need help?

You can reach out to the LooksRare team via our Developers Discord: https://discord.gg/LooksRareDevelopers

LooksRare Token & Staking Overview

What is the LooksRare Token ("LOOKS")?

The LooksRare token ("LOOKS") is the protocol token from the LooksRare ecosystem. It is an ERC-20 token deployed on the Ethereum blockchain.

LOOKS supply was pre-minted with 200 million tokens transferred at the deployment and has a **supply cap of 1 billion tokens**.

How is LOOKS distributed?

The LOOKS token's emission is controlled by the `TokenDistributor` contract that handles the issuance of freshly minted LOOKS tokens to two contracts:

1. Distributor contract (`TokenDistributor`) — It allows staking LOOKS to earn LOOKS tokens.
2. Splitter contract (`TokenSplitter`) — It receives LOOKS tokens that are distributed in an asynchronous way ("splitted" based on predefined weights) to:

- Trading rewards distributor contract (`TradingRewardsDistributor`).
- Treasury & team vesting contracts (`VestingContractWithFeeSharing`).

The LOOKS token is distributed to users in the LooksRare protocol to:

- Stakers (`FeeSharingSystem`) — LOOKS token holders can deposit LOOKS that are auto-compounded at each user interaction. This smart contract is an additional layer on top of the `TokenDistributor` contract, which also unlocks WETH rewards, stemming from the LooksRare exchange's protocol fee collected at every trade.
- Traders (`TradingRewardsDistributor`) - Traders can collect rewards using a rolling Merkle-tree based system that distributes daily LOOKS trading rewards based on volume activities (excluding private sale strategies). Computation of the trading rewards is done off-chain.
- Liquidity providers (`StakingPoolForUniswapV2Tokens`) – Liquidity providers can deposit liquidity in Uniswap V2 and receive LP tokens that they can stake to earn LOOKS tokens.

Some LOOKS tokens, part of the pre-minted tokens, are vested by strategic sale participants (`PrivateSaleWithFeeSharing`) that also collect WETH based on the prorata of the LOOKS tokens held in this contract.

Vesting contracts (`VestingContractWithFeeSharing`) also collect WETH when rewards are distributed to users based on the prorata of the LOOKS held in this contract relative to the total LOOKS eligible for fee-sharing (active + passive staking).

Finally, other preminted tokens (120 million) were distributed through the airdrop contract (`LooksRareAirdrop`), that relies on Merkle-tree computation, with an added requirement to submit a valid maker ask order.

For details on the token release schedule, please refer to the token economics page.

What is the circulating supply of the LOOKS token?

The circulating supply can be calculated such as:

```
circulatingSupply = totalSupply() - balanceOf(TokenSplitter) - balanceOf(VestingContractTeam) - balanceOf(VestingContractTreasury) - balanceOf(PrivateSaleWithFeeSharing)
```

What is the difference between active staking and passive staking for fee-sharing?

The WETH tokens earned through the platform's fee-sharing mechanism are collected differently by Active and Passive stakers:

- **LOOKS stakers ("Active fee-sharing")** — WETH are distributed daily to LOOKS stakers (`FeeSharingSystem`). The LOOKS stakers also receive LOOKS tokens.
- **Passive fee-sharing** — WETH are distributed daily to vesting contracts. Participants eligible to passive fee-sharing do not receive LOOKS tokens from the user emission schedule.

Every day, the previous' day protocol fees (in WETH) are distributed based on the total LOOKS staked (both active and passive).

The distribution of WETH for LOOKS active stakers is linearly distributed over the period (since active stakers are free to deposit and withdraw LOOKS anytime). Conversely, for passive stakers, the daily WETH distribution happens at once.

Other design decisions

The design of the smart contracts relies greatly on the previous work and algorithms developed by leading DeFi protocols, specifically Synthetix (SIP-63), SushiSwap (MasterChef), and Yearn (auto-compounding strategies).

*Last updated on **Aug 28, 2023***

AggregatorFeeSharingWithUniswapV3

An extra layer built on top of the `FeeSharingSystem` that sells WETH to LOOKS using the Uniswap V3 router. It is a yield aggregator that optimizes LOOKS staking.

> Prime shares represent the number of shares in the FeeSharingSystem. When not specified, shares represent secondary shares in this contract.

Methods

MAXIMUM_HARVEST_BUFFER_BLOCKS

```
function MAXIMUM_HARVEST_BUFFER_BLOCKS() external view returns (uint256)
```

Returns

Name	Type	Description
MAXIMUM_HARVEST_BUFFER_BLOCKS	uint256	maximum number of blocks between two harvests

MINIMUM_DEPOSIT_LOOKS

```
function MINIMUM_DEPOSIT_LOOKS() external view returns (uint256)
```

Returns

Name	Type	Description
MINIMUM_DEPOSIT_LOOKS	uint256	minimum amount for a single deposit (in LOOKS)

calculateSharePriceInLOOKS

```
function calculateSharePriceInLOOKS() external view returns (uint256)
```

Calculate price of one share (in LOOKS token). Share price is expressed times 1e18

Returns

Name	Type	Description
-	uint256	price of a (secondary) share in LOOKS

calculateSharePriceInPrimeShare

```
function calculateSharePriceInPrimeShare() external view returns (uint256)
```

Calculate price of one (secondary) share (in prime share). Share price is expressed times 1e18

Returns

Name	Type	Description
-	uint256	price of a (secondary) share in prime share

calculateSharesValueInLOOKS

```
function calculateSharesValueInLOOKS(address user) external view returns (uint256)
```

Calculate shares value of a user (in LOOKS)

Parameters

Name	Type	Description
user	address	address of the user

Returns

Name	Type	Description
-	uint256	value (in LOOKS) of the total shares of a user

canHarvest

```
function canHarvest() external view returns (bool)
```

Returns

Name	Type	Description
-	bool	whether harvests are triggered automatically at standard user action (i.e., deposits, withdraws)

checkAndAdjustLOOKSTokenAllowanceIfRequired

```
function checkAndAdjustLOOKSTokenAllowanceIfRequired() external nonpayable
```

Adjust allowance if necessary

Only callable by owner.

checkAndAdjustRewardTokenAllowanceIfRequired

```
function checkAndAdjustRewardTokenAllowanceIfRequired() external nonpayable
```

Adjust allowance if necessary

Only callable by owner.

deposit

```
function deposit(uint256 amount) external nonpayable
```

Deposit LOOKS tokens

There is a limit of 1 LOOKS per deposit to prevent potential manipulation of the shares

Parameters

Name	Type	Description
amount	uint256	amount to deposit (in LOOKS)

feeSharingSystem

```
function feeSharingSystem() external view returns (contract FeeSharingSystem)
```

Returns

Name	Type	Description
-	contract FeeSharingSystem	address of the `FeeSharingSystem` contract

harvestAndSellAndCompound

```
function harvestAndSellAndCompound() external nonpayable
```

Harvest pending WETH, sell them to LOOKS, and deposit LOOKS (if possible)

Only callable by owner. It is meant for emergency.

harvestBufferBlocks

```
function harvestBufferBlocks() external view returns (uint256)
```

Returns

Name	Type	Description
-	uint256	number of blocks between two consecutive harvests

lastHarvestBlock

```
function lastHarvestBlock() external view returns (uint256)
```

Returns

Name	Type	Description
lastHarvestBlock	uint256	block number of the last harvest

looksRareToken

```
function looksRareToken() external view returns (contract IERC20)
```

Returns

Name	Type	Description
-	contract IERC20	address of the `LooksRareToken`

maxPriceLOOKSInWETH

```
function maxPriceLOOKSInWETH() external view returns (uint256)
```

Returns

Name	Type	Description
maxPriceLOOKSInWETH	uint256	maximum price of 1 LOOKS (in WETH) times 1e18

owner

```
function owner() external view returns (address)
```

Returns the address of the current owner.

Returns

Name	Type	Description
owner	address	address of the current owner

pause

```
function pause() external nonpayable
```

Pause

Only callable by owner

paused

```
function paused() external view returns (bool)
```

Returns true if the contract is paused, and false otherwise.

Returns

Name	Type	Description
paused	bool	whether the contract is in pause status. If paused, new user deposits are suspended but withdrawals are available.

renounceOwnership

```
function renounceOwnership() external nonpayable
```

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.

rewardToken

```
function rewardToken() external view returns (contract IERC20)
```

Returns

Name	Type	Description
-	contract IERC20	address of the `rewardToken` (i.e., WETH)

startHarvest

```
function startHarvest() external nonpayable
```

Start automatic harvest/selling transactions

Only callable by owner

stopHarvest

```
function stopHarvest() external nonpayable
```

Stop automatic harvest transactions

Only callable by owner

thresholdAmount

```
function thresholdAmount() external view returns (uint256)
```

Returns

Name	Type	Description
-	uint256	threshold amount (in `rewardToken`) to trigger a sale on Uniswap V3

totalShares

```
function totalShares() external view returns (uint256)
```

Returns

Name	Type	Description
totalShares	uint256	number of total (secondary) shares

tradingFeeUniswapV3

```
function tradingFeeUniswapV3() external view returns (uint24)
```

Returns

Name	Type	Description
tradingFeeUniswapV3	uint24	trading fee for Uniswap v3 (e.g. 3000 ---> 0.30%, 500 ---> 0.05%)

transferOwnership

```
function transferOwnership(address newOwner) external nonpayable
```

Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the current owner.

Parameters

Name	Type	Description
newOwner	address	address of the new owner

uniswapRouter

```
function uniswapRouter() external view returns (contract ISwapRouter)
```

Returns

Name	Type	Description
-	contract ISwapRouter	address of the Uniswap V3 swap router

unpause

```
function unpause() external nonpayable
```

Unpause

Only callable by owner

updateHarvestBufferBlocks

```
function updateHarvestBufferBlocks(uint256 _newHarvestBufferBlocks) external nonpayable
```

Update harvest buffer block

Only callable by owner.

Parameters

Name	Type	Description
newHarvestBufferBlocks	uint256	buffer in blocks between two harvest operations

updateMaxPriceOfLOOKSInWETH

```
function updateMaxPriceOfLOOKSInWETH(uint256 _newMaxPriceLOOKSInWETH) external nonpayable
```

Update maximum price of LOOKS in WETH

Only callable by owner

Parameters

Name	Type	Description
newMaxPriceLOOKSInWETH	uint256	new maximum price of LOOKS in WETH times 1e18

updateThresholdAmount

```
function updateThresholdAmount(uint256 _newThresholdAmount) external nonpayable
```

Adjust threshold amount for periodic Uniswap v3 WETH to LOOKS conversion

Only callable by owner

Parameters

Name	Type	Description
newThresholdAmount	uint256	new threshold amount (in WETH)

updateTradingFeeUniswapV3

```
function updateTradingFeeUniswapV3(uint24 _newTradingFeeUniswapV3) external nonpayable
```

Adjust trading fee for Uniswap v3

Only callable by owner. Can only be 10,000 (1%), 3000 (0.3%), or 500 (0.05%).

Parameters

Name	Type	Description
newTradingFeeUniswapV3	uint24	new trading fee for Uniswap V3

userInfo

```
function userInfo(address) external view returns (uint256)
```

Parameters

Name	Type	Description
-	address	address of the user

Returns

Name	Type	Description
-	uint256	number of shares of the user

withdraw

```
function withdraw(uint256 shares) external nonpayable
```

Redeem shares for LOOKS tokens

Parameters

Name	Type	Description
shares	uint256	number of shares to redeem

withdrawAll

```
function withdrawAll() external nonpayable
```

Withdraw all shares of sender

Events

ConversionToLOOKS

```
event ConversionToLOOKS(uint256 amountSold, uint256 amountReceived)
```

Parameters

Name	Type	Description
amountSold	uint256	amount of tokens sold (in `rewardToken`)
amountReceived	uint256	amount of LOOKS received

Deposit

```
event Deposit(address indexed user, uint256 amount)
```

Parameters

Name	Type	Description
user `indexed`	address	address of the user
amount	uint256	amount deposited in LOOKS

FailedConversion

```
event FailedConversion()
```

HarvestStart

```
event HarvestStart()
```

HarvestStop

```
event HarvestStop()
```

NewHarvestBufferBlocks

```
event NewHarvestBufferBlocks(uint256 harvestBufferBlocks)
```

Parameters

Name	Type	Description
harvestBufferBlocks	uint256	number of blocks between two consecutive harvests

NewMaximumPriceLOOKSInWETH

```
event NewMaximumPriceLOOKSInWETH(uint256 maxPriceLOOKSInWETH)
```

Parameters

Name	Type	Description
maxPriceLOOKSInWETH	uint256	maximum price of LOOKS in WETH times 1e18

NewThresholdAmount

```
event NewThresholdAmount(uint256 thresholdAmount)
```

Parameters

Name	Type	Description
thresholdAmount	uint256	threshold amount (in `rewardToken`) to trigger a sale on Uniswap V3

NewTradingFeeUniswapV3

```
event NewTradingFeeUniswapV3(uint24 tradingFeeUniswapV3)
```

Parameters

Name	Type	Description
tradingFeeUniswapV3	uint24	trading fee for Uniswap V3

OwnershipTransferred

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

Parameters

Name	Type	Description
previousOwner `indexed`	address	address of the previous owner
newOwner `indexed`	address	address of the new owner

Paused

```
event Paused(address account)
```

Parameters

Name	Type	Description
account	address	-

Unpaused

```
event Unpaused(address account)
```

Parameters

Name	Type	Description
account	address	-

Withdraw

```
event Withdraw(address indexed user, uint256 amount)
```

Parameters

Name	Type	Description
user `indexed`	address	address of the user
amount	uint256	amount received (in LOOKS)

FeeSharingSetter

It receives LooksRare protocol fees and owns the FeeSharingSystem contract. It can plug to AMMs for converting all received currencies to WETH.

Methods

DEFAULT_ADMIN_ROLE

```
function DEFAULT_ADMIN_ROLE() external view returns (bytes32)
```

Returns

Name	Type	Description
DEFAULT_ADMIN_ROLE	bytes32	default admin role identifier

MAX_REWARD_DURATION_IN_BLOCKS

```
function MAX_REWARD_DURATION_IN_BLOCKS() external view returns (uint256)
```

Returns

Name	Type	Description
MAX_REWARD_DURATION_IN_BLOCKS	uint256	maximum reward duration that can be set (in blocks)

MIN_REWARD_DURATION_IN_BLOCKS

```
function MIN_REWARD_DURATION_IN_BLOCKS() external view returns (uint256)
```

Returns

Name	Type	Description
MIN_REWARD_DURATION_IN_BLOCKS	uint256	minimum reward duration that can be set (in blocks)

OPERATOR_ROLE

```
function OPERATOR_ROLE() external view returns (bytes32)
```

Returns

Name	Type	Description
OPERATOR_ROLE	bytes32	operator role identifier

addFeeStakingAddresses

```
function addFeeStakingAddresses(address[] _stakingAddresses) external nonpayable
```

Add staking addresses

Parameters

Name	Type	Description
_stakingAddresses	address[]	array of addresses eligible for fee-sharing only

convertCurrencyToRewardToken

```
function convertCurrencyToRewardToken(address token, bytes additionalData) external nonpayable
```

Convert currencies to reward token

Function only usable only for whitelisted currencies (where no potential side effect)

Parameters

Name	Type	Description
token	address	address of the token to sell
additionalData	bytes	additional data (e.g., slippage)

feeSharingSystem

```
function feeSharingSystem() external view returns (contract FeeSharingSystem)
```

Returns

Name	Type	Description
feeSharingSystem	contract FeeSharingSystem	address of the FeeSharingSystem

getRoleAdmin

```
function getRoleAdmin(bytes32 role) external view returns (bytes32)
```

Returns the admin role that controls `role`. See {grantRole} and {revokeRole}. To change a role's admin, use {_setRoleAdmin}.

Parameters

Name	Type	Description
role	bytes32	-

Returns

Name	Type	Description
-	bytes32	-

grantRole

```
function grantRole(bytes32 role, address account) external nonpayable
```

Grants `role` to `account`. If `account` had not been already granted `role`, emits a {RoleGranted} event. Requirements: - the caller must have `role`'s admin role.

Parameters

Name	Type	Description
role	bytes32	identifier of the role (e.g., OPERATOR_ROLE)
account	address	address of the account

hasRole

```
function hasRole(bytes32 role, address account) external view returns (bool)
```

Returns `true` if `account` has been granted `role`.

Parameters

Name	Type	Description
role	bytes32	identifier of the role (e.g., OPERATOR_ROLE)
account	address	address of the account

Returns

Name	Type	Description
-	bool	whether the account has been granted the role

lastRewardDistributionBlock

```
function lastRewardDistributionBlock() external view returns (uint256)
```

Returns

Name	Type	Description
-	uint256	block for the last reward distribution

looksRareToken

```
function looksRareToken() external view returns (contract IERC20)
```

Returns

Name	Type	Description
looksRareToken	contract IERC20	address of the LooksRareToken

nextRewardDurationInBlocks

```
function nextRewardDurationInBlocks() external view returns (uint256)
```

Returns

Name	Type	Description
nextRewardDurationInBlocks	uint256	next reward duration in blocks (that will apply for the next update of the rewards)

removeFeeStakingAddresses

```
function removeFeeStakingAddresses(address[] _stakingAddresses) external nonpayable
```

Remove staking addresses

Parameters

Name	Type	Description
_stakingAddresses	address[]	array of addresses eligible for fee-sharing only

renounceRole

```
function renounceRole(bytes32 role, address account) external nonpayable
```

Revokes `role` from the calling account. Roles are often managed via {grantRole} and {revokeRole}: this function's purpose is to provide a mechanism for accounts to lose their privileges if they are compromised (such as when a trusted device is misplaced). If the calling account had been revoked `role`, emits a {RoleRevoked} event. Requirements: - the caller must be `account`.

Parameters

Name	Type	Description
role	bytes32	identifier of the role (e.g., OPERATOR_ROLE)
account	address	address of the account

revokeRole

```
function revokeRole(bytes32 role, address account) external nonpayable
```

Revokes `role` from `account`. If `account` had been granted `role`, emits a {RoleRevoked} event. Requirements: - the caller must have `role`'s admin role.

Parameters

Name	Type	Description
role	bytes32	identifier of the role (e.g., OPERATOR_ROLE)
account	address	address of the account

rewardConvertor

```
function rewardConvertor() external view returns (contract IRewardConvertor)
```

Returns

Name	Type	Description
rewardConverter	contract IRewardConvertor	address of the reward convertor (it returns address(0) if not set)

rewardDurationInBlocks

```
function rewardDurationInBlocks() external view returns (uint256)
```

Returns

Name	Type	Description
rewardDurationInBlocks	uint256	current reward duration in blocks

rewardToken

```
function rewardToken() external view returns (contract IERC20)
```

Returns

Name	Type	Description
rewardToken	contract IERC20	address of the rewardToken (i.e., WETH)

setNewRewardDurationInBlocks

```
function setNewRewardDurationInBlocks(uint256 _newRewardDurationInBlocks) external nonpayable
```

Set new reward duration in blocks for next update

Parameters

Name	Type	Description
_newRewardDurationInBlocks	uint256	number of blocks for new reward period

setRewardConvertor

```
function setRewardConvertor(address _rewardConvertor) external nonpayable
```

Set reward convertor contract

Parameters

Name	Type	Description
_rewardConvertor	address	address of the reward convertor (set to null to deactivate)

supportsInterface

```
function supportsInterface(bytes4 interfaceId) external view returns (bool)
```

See {IERC165-supportsInterface}.

Parameters

Name	Type	Description
interfaceId	bytes4	-

Returns

Name	Type	Description
-	bool	whether it supports the interfaceId

tokenDistributor

```
function tokenDistributor() external view returns (contract TokenDistributor)
```

Returns

Name	Type	Description
tokenDistributor	contract TokenDistributor	address of the TokenDistributor

transferOwnershipOfFeeSharingSystem

```
function transferOwnershipOfFeeSharingSystem(address _newOwner) external nonpayable
```

Transfer ownership of fee sharing system

Parameters

Name	Type	Description
_newOwner	address	address of the new owner

updateRewards

```
function updateRewards() external nonpayable
```

Update the reward per block (in rewardToken)

It automatically retrieves the number of pending WETH and adjusts based on the balance of LOOKS in fee-staking addresses that exist in the set.

Events

ConversionToRewardToken

```
event ConversionToRewardToken(address indexed token, uint256 amountConverted, uint256 amountReceiv
```

Parameters

Name	Type	Description
token `indexed`	address	-
amountConverted	uint256	-
amountReceived	uint256	-

FeeStakingAddressesAdded

```
event FeeStakingAddressesAdded(address[] feeStakingAddresses)
```

Parameters

Name	Type	Description
feeStakingAddresses	address[]	-

FeeStakingAddressesRemoved

```
event FeeStakingAddressesRemoved(address[] feeStakingAddresses)
```

Parameters

Name	Type	Description
feeStakingAddresses	address[]	-

NewFeeSharingSystemOwner

```
event NewFeeSharingSystemOwner(address newOwner)
```

Parameters

Name	Type	Description
newOwner	address	address of the new owner

NewRewardConvertor

```
event NewRewardConvertor(address rewardConvertor)
```

Parameters

Name	Type	Description
rewardConvertor	address	-

NewRewardDurationInBlocks

```
event NewRewardDurationInBlocks(uint256 rewardDurationInBlocks)
```

Parameters

Name	Type	Description
rewardDurationInBlocks	uint256	-

RoleAdminChanged

```
event RoleAdminChanged(bytes32 indexed role, bytes32 indexed previousAdminRole, bytes32 indexed ne
```

Parameters

Name	Type	Description
role `indexed`	bytes32	-
previousAdminRole `indexed`	bytes32	-
newAdminRole `indexed`	bytes32	-

RoleGranted

```
event RoleGranted(bytes32 indexed role, address indexed account, address indexed sender)
```

Parameters

Name	Type	Description
role `indexed`	bytes32	-
account `indexed`	address	-
sender `indexed`	address	-

RoleRevoked

```
event RoleRevoked(bytes32 indexed role, address indexed account, address indexed sender)
```

Parameters

Name	Type	Description
role `indexed`	bytes32	-
account `indexed`	address	-
sender `indexed`	address	-

FeeSharingSystem

It handles the distribution of fees using WETH along with the auto-compounding of LOOKS.

Methods

PRECISION_FACTOR

```
function PRECISION_FACTOR() external view returns (uint256)
```

Returns

Name	Type	Description
PRECISION_FACTOR	uint256	-

calculatePendingRewards

```
function calculatePendingRewards(address user) external view returns (uint256)
```

Calculate pending rewards (WETH) for a user

Parameters

Name	Type	Description
user	address	address of the user

Returns

Name	Type	Description
-	uint256	pending rewards for the user

calculateSharePriceInLOOKS

```
function calculateSharePriceInLOOKS() external view returns (uint256)
```

Calculate price of one share (in LOOKS token) Share price is expressed times 1e18

Returns

Name	Type	Description
-	uint256	price of one share (in LOOKS token)

calculateSharesValueInLOOKS

```
function calculateSharesValueInLOOKS(address user) external view returns (uint256)
```

Calculate value of LOOKS for a user given a number of shares owned

Parameters

Name	Type	Description
user	address	address of the user

Returns

Name	Type	Description
-	uint256	estimated number of LOOKS for a user given the number of owned shares

currentRewardPerBlock

```
function currentRewardPerBlock() external view returns (uint256)
```

Returns

Name	Type	Description
-	uint256	current reward per block (in WETH)

deposit

```
function deposit(uint256 amount, bool claimRewardToken) external nonpayable
```

Deposit staked tokens (and collect reward tokens if requested)

Parameters

Name	Type	Description
amount	uint256	amount to deposit (in LOOKS)
claimRewardToken	bool	whether to claim reward tokens

harvest

```
function harvest() external nonpayable
```

Harvest reward tokens that are pending

lastRewardAdjustment

```
function lastRewardAdjustment() external view returns (uint256)
```

Returns

Name	Type	Description
lastRewardAdjustment	uint256	last reward adjustment

lastRewardBlock

```
function lastRewardBlock() external view returns (uint256)
```

Return last block where trading rewards were distributed

Returns

Name	Type	Description
lastRewardBlock	uint256	last block where trading rewards were distributed

lastUpdateBlock

```
function lastUpdateBlock() external view returns (uint256)
```

Returns

Name	Type	Description
lastUpdateBlock	uint256	last update block

looksRareToken

```
function looksRareToken() external view returns (contract IERC20)
```

Returns

Name	Type	Description
looksRareToken	contract IERC20	address of the LooksRare token

owner

```
function owner() external view returns (address)
```

Returns the address of the current owner.

Returns

Name	Type	Description
owner	address	address of the current owner

periodFinish

```
function periodFinish() external view returns (uint256)
```

Returns

Name	Type	Description
periodFinish	uint256	endBlock for the current reward period

renounceOwnership

```
function renounceOwnership() external nonpayable
```

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.

rewardPerTokenStored

```
function rewardPerTokenStored() external view returns (uint256)
```

Returns

Name	Type	Description
rewardPerTokenStored	uint256	current reward that has been stored per share

rewardToken

```
function rewardToken() external view returns (contract IERC20)
```

Returns

Name	Type	Description
rewardToken	contract IERC20	reward token address (WETH)

tokenDistributor

```
function tokenDistributor() external view returns (contract TokenDistributor)
```

Returns

Name	Type	Description
tokenDistributor	contract TokenDistributor	address of the `TokenDistributor`

totalShares

```
function totalShares() external view returns (uint256)
```

Returns

Name	Type	Description
totalShares	uint256	number of outstanding shares

transferOwnership

```
function transferOwnership(address newOwner) external nonpayable
```

Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the current owner.

Parameters

Name	Type	Description
newOwner	address	address of the new owner

updateRewards

```
function updateRewards(uint256 reward, uint256 rewardDurationInBlocks) external nonpayable
```

Update the reward per block (in rewardToken)

Only callable by owner. Owner is meant to be another smart contract.

Parameters

Name	Type	Description
reward	uint256	total reward distributed in the period
rewardDurationInBlocks	uint256	number of blocks in the period (e.g., 43200)

userInfo

```
function userInfo(address) external view returns (uint256 shares, uint256 userRewardPerTokenPaid,
```

Parameters

Name	Type	Description
-	address	address of the user

Returns

Name	Type	Description
shares	uint256	number of shares
userRewardPerTokenPaid	uint256	user reward per token paid
rewards	uint256	pending rewards

withdraw

```
function withdraw(uint256 shares, bool claimRewardToken) external nonpayable
```

Withdraw staked tokens (and collect reward tokens if requested)

Parameters

Name	Type	Description
shares	uint256	shares to withdraw
claimRewardToken	bool	whether to claim reward tokens

withdrawAll

```
function withdrawAll(bool claimRewardToken) external nonpayable
```

Withdraw all staked tokens (and collect reward tokens if requested)

Parameters

Name	Type	Description
claimRewardToken	bool	whether to claim reward tokens

Events

Deposit

```
event Deposit(address indexed user, uint256 amount, uint256 harvestedAmount)
```

Parameters

Name	Type	Description
user indexed	address	address of the user
amount	uint256	amount deposited
harvestedAmount	uint256	harvested amount (in WETH)

Harvest

```
event Harvest(address indexed user, uint256 harvestedAmount)
```

Parameters

Name	Type	Description
user indexed	address	address of the user
harvestedAmount	uint256	harvested amount (in WETH)

NewRewardPeriod

```
event NewRewardPeriod(uint256 numberBlocks, uint256 rewardPerBlock, uint256 reward)
```

Parameters

Name	Type	Description
numberBlocks	uint256	number of blocks for the new reward period
rewardPerBlock	uint256	reward per block (in WETH)
reward	uint256	total reward distributed over the period (in WETH)

OwnershipTransferred

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

Parameters

Name	Type	Description
previousOwner indexed	address	address of the previous owner
newOwner indexed	address	address of the new owner

Withdraw

```
event Withdraw(address indexed user, uint256 amount, uint256 harvestedAmount)
```

Parameters

Name	Type	Description
user indexed	address	address of the user
amount	uint256	amount withdrawn
harvestedAmount	uint256	harvested amount (in WETH)

LooksRareToken

Methods

SUPPLY_CAP

```
function SUPPLY_CAP() external view returns (uint256)
```

View supply cap

Returns

Name	Type	Description
SUPPLY_CAP	uint256	supply cap (set at 1 billion of tokens)

allowance

```
function allowance(address owner, address spender) external view returns (uint256)
```

See (ERC20-allowance)

Parameters

Name	Type	Description
owner	address	address of the owner
spender	address	address of the spender (e.g., FeeSharingSystem)

Returns

Name	Type	Description
allowance	uint256	amount of tokens that can be transferred by the spender

approve

```
function approve(address spender, uint256 amount) external nonpayable returns (bool)
```

See (ERC20-approve). Requirements: - `spender` cannot be the zero address

Parameters

Name	Type	Description
spender	address	address of the spender
amount	uint256	amount of tokens to approve

Returns

Name	Type	Description
-	bool	amount of tokens approved to be transferred by the spender

balanceOf

```
function balanceOf(address account) external view returns (uint256)
```

See (ERC20-balanceOf)

Parameters

Name	Type	Description
account	address	address of the account

Returns

Name	Type	Description
balance	uint256	balance of the account address

decimals

```
function decimals() external view returns (uint8)
```

Returns the number of decimals used to get its user representation. For example, if `decimals` equal `2`, a balance of `505` tokens should be displayed to a user as `5.05` (`505 / 10 ** 2`). Tokens usually opt for a value of 18, imitating the relationship between Ether and Wei. This is the value {ERC20} uses, unless this function is overridden; NOTE: This information is only used for display purposes: it in no way affects any of the arithmetic of the contract, including {ERC20-balanceOf} and {ERC20-transfer}.

Returns

Name	Type	Description
decimals	uint8	number of decimals ("18")

decreaseAllowance

```
function decreaseAllowance(address spender, uint256 subtractedValue) external nonpayable returns (
```

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 {ERC20-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`.

Parameters

Name	Type	Description
spender	address	address of the spender
subtractedValue	uint256	subtracted value

Returns

Name	Type	Description
-	bool	-

increaseAllowance

```
function increaseAllowance(address spender, uint256 addedValue) external nonpayable returns (bool)
```

Atomically increase 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 {ERC20-approve}. Emits an {Approval} event indicating the updated allowance. Requirements: - `spender` cannot be the zero address.

Parameters

Name	Type	Description
spender	address	address of the spender
addedValue	uint256	added value

Returns

Name	Type	Description
-	bool	-

mint

```
function mint(address account, uint256 amount) external nonpayable returns (bool status)
```

Mint LOOKS tokens

Parameters

Name	Type	Description
account	address	address to receive tokens
amount	uint256	amount to mint

Returns

Name	Type	Description
status	bool	true if mint is successful, false if not

name

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

Returns the name of the token.

Returns

Name	Type	Description
name	string	name of the token ("LooksRare Token")

owner

```
function owner() external view returns (address)
```

Returns the address of the current owner.

Returns

Name	Type	Description
owner	address	address of the current owner

renounceOwnership

```
function renounceOwnership() external nonpayable
```

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.

symbol

```
function symbol() external view returns (string)
```

Returns the symbol of the token, usually a shorter version of the name.

Returns

Name	Type	Description
symbol	string	symbol of the token ("LOOKS")

totalSupply

```
function totalSupply() external view returns (uint256)
```

See (ERC20-totalSupply).

Returns

Name	Type	Description
totalSupply	uint256	total supply of the token

transfer

```
function transfer(address recipient, uint256 amount) external nonpayable returns (bool)
```

See (ERC20-transfer). Requirements: - `recipient` cannot be the zero address. - the caller must have a balance of at least `amount`.

Parameters

Name	Type	Description
recipient	address	recipient address
amount	uint256	amount of tokens to transfer

Returns

Name	Type	Description
-	bool	whether the transaction succeeds

transferFrom

```
function transferFrom(address sender, address recipient, uint256 amount) external nonpayable retur
```

See (ERC20-transferFrom). Emits an {Approval} event indicating the updated allowance. This is not required by the EIP. See the note at the beginning of {ERC20}. 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`.

Parameters

Name	Type	Description
sender	address	sender address
recipient	address	recipient address
amount	uint256	amount of tokens transferred

Returns

Name	Type	Description
-	bool	whether the transaction succeeds

transferOwnership

```
function transferOwnership(address newOwner) external nonpayable
```

Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the current owner.

Parameters

Name	Type	Description
newOwner	address	address of the new owner

Events

Approval

```
event Approval(address indexed owner, address indexed spender, uint256 value)
```

Parameters

Name	Type	Description
owner `indexed`	address	owner address
spender `indexed`	address	spender address
value	uint256	amount of tokens approved

OwnershipTransferred

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

Parameters

Name	Type	Description
previousOwner `indexed`	address	address of the previous owner
newOwner `indexed`	address	address of the new owner

Transfer

```
event Transfer(address indexed from, address indexed to, uint256 value)
```

Parameters

Name	Type	Description
from `indexed`	address	address from where tokens were transferred
to `indexed`	address	address that received the tokens
value	uint256	amount of tokens transferred

TokenDistributor

It handles the distribution of LOOKS token. It auto-adjusts block rewards over a set number of periods.

Methods

NUMBER_PERIODS

```
function NUMBER_PERIODS() external view returns (uint256)
```

Returns

Name	Type	Description
_NUMBER_PERIODS	uint256	number of periods

PRECISION_FACTOR

```
function PRECISION_FACTOR() external view returns (uint256)
```

Returns

Name	Type	Description
PRECISION_FACTOR	uint256	precision factor

START_BLOCK

```
function START_BLOCK() external view returns (uint256)
```

Returns

Name	Type	Description
START_BLOCK	uint256	start block for the LOOKS staking

accTokenPerShare

```
function accTokenPerShare() external view returns (uint256)
```

Returns

Name	Type	Description
accTokenPerShare()	uint256	accumulated token per share

calculatePendingRewards

```
function calculatePendingRewards(address user) external view returns (uint256)
```

Calculate pending rewards for a user

Parameters

Name	Type	Description
user	address	address of the user

Returns

Name	Type	Description
-	uint256	pending rewards for a user

currentPhase

```
function currentPhase() external view returns (uint256)
```

Returns

Name	Type	Description
currentPhase	uint256	current phase of the LOOKS staking program

deposit

```
function deposit(uint256 amount) external nonpayable
```

Deposit staked tokens and compounds pending rewards

Parameters

Name	Type	Description
amount	uint256	amount to deposit (in LOOKS)

endBlock

```
function endBlock() external view returns (uint256)
```

Returns

Name	Type	Description
endBlock	uint256	end block for the token distribution

harvestAndCompound

```
function harvestAndCompound() external nonpayable
```

Compound based on pending rewards

lastRewardBlock

```
function lastRewardBlock() external view returns (uint256)
```

Returns

Name	Type	Description
lastRewardBlock	uint256	last reward block

looksRareToken

```
function looksRareToken() external view returns (contract ILooksRareToken)
```

Returns

Name	Type	Description
looksRareToken	contract ILooksRareToken	address of the LooksRareToken

rewardPerBlockForOthers

```
function rewardPerBlockForOthers() external view returns (uint256)
```

Returns

Name	Type	Description
rewardPerBlockForOthers	uint256	number of LOOKS tokens minted at each block in the current phase to the TokenSplitter

rewardPerBlockForStaking

```
function rewardPerBlockForStaking() external view returns (uint256)
```

Returns

Name	Type	Description
rewardPerBlockForStaking	uint256	number of LOOKS tokens that is minted at each block in the current phase and distributed for LOOKS staking

stakingPeriod

```
function stakingPeriod(uint256) external view returns (uint256 rewardPerBlockForStaking, uint256 r
```

Parameters

Name	Type	Description
stakingPeriod (0, 1 2, or 3)	uint256	staking period

Returns

Name	Type	Description
rewardPerBlockForStaking	uint256	number of LOOKS tokens that is minted at each block in the current phase and distributed for LOOKS staking
rewardPerBlockForOthers	uint256	number of LOOKS tokens that is minted at each block in the current phase to the TokenSplitter
periodLengthInBlock	uint256	length of the period (in blocks)

tokenSplitter

```
function tokenSplitter() external view returns (address)
```

Returns

Name	Type	Description
tokenSplitter	address	address of the TokenSplitter contract

totalAmountStaked

```
function totalAmountStaked() external view returns (uint256)
```

Returns

Name	Type	Description
totalAmountStaked	uint256	total amount of LOOKS tokens being staked in this contract

updatePool

```
function updatePool() external nonpayable
```

Update pool rewards

userInfo

```
function userInfo(address) external view returns (uint256 amount, uint256 rewardDebt)
```

Parameters

Name	Type	Description
-	address	address of the user

Returns

Name	Type	Description
amount	uint256	amount of tokens being staked
rewardDebt	uint256	reward debt

withdraw

```
function withdraw(uint256 amount) external nonpayable
```

Withdraw staked tokens and compound pending rewards

Parameters

Name	Type	Description
amount	uint256	amount to withdraw

withdrawAll

```
function withdrawAll() external nonpayable
```

Withdraw all staked tokens and collect tokens

Events

Compound

```
event Compound(address indexed user, uint256 harvestedAmount)
```

Parameters

Name	Type	Description
user `indexed`	address	-
harvestedAmount	uint256	-

Deposit

```
event Deposit(address indexed user, uint256 amount, uint256 harvestedAmount)
```

Parameters

Name	Type	Description
user `indexed`	address	-
amount	uint256	-
harvestedAmount	uint256	-

NewRewardsPerBlock

```
event NewRewardsPerBlock(uint256 indexed currentPhase, uint256 startBlock, uint256 rewardPerBlockF
```

Parameters

Name	Type	Description
currentPhase `indexed`	uint256	-
startBlock	uint256	-
rewardPerBlockForStaking	uint256	-
rewardPerBlockForOthers	uint256	-

Withdraw

```
event Withdraw(address indexed user, uint256 amount, uint256 harvestedAmount)
```

Parameters

Name	Type	Description
user `indexed`	address	-
amount	uint256	-
harvestedAmount	uint256	-

TokenSplitter

It splits LOOKS to team/treasury/trading volume reward accounts based on shares.

Methods

TOTAL_SHARES

```
function TOTAL_SHARES() external view returns (uint256)
```

Returns

Name	Type	Description
TOTAL_SHARES	uint256	total number of shares

accountInfo

```
function accountInfo(address) external view returns (uint256 shares, uint256 tokensDistributedToAc
```

Parameters

Name	Type	Description
account	address	account address

Returns

Name	Type	Description
shares	uint256	number of shares
tokensDistributedToAccount	uint256	total number of tokens distributed to account

calculatePendingRewards

```
function calculatePendingRewards(address account) external view returns (uint256)
```

Retrieve amount of LOOKS tokens that can be transferred

Parameters

Name	Type	Description
account	address	address of the account

Returns

Name	Type	Description
pendingRewards	uint256	amount of LOOKS tokens that can be transferred

looksRareToken

```
function looksRareToken() external view returns (contract IERC20)
```

Returns

Name	Type	Description
looksRareToken	contract IERC20	address of the LooksRareToken

owner

```
function owner() external view returns (address)
```

Returns the address of the current owner.

Returns

Name	Type	Description
owner	address	address of the current owner

releaseTokens

```
function releaseTokens(address account) external nonpayable
```

Release LOOKS tokens to the account

Parameters

Name	Type	Description
account	address	address of the account

renounceOwnership

```
function renounceOwnership() external nonpayable
```

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.

totalTokensDistributed

```
function totalTokensDistributed() external view returns (uint256)
```

Returns

Name	Type	Description
totalTokensDistributed	uint256	total amount of LOOKS tokens distributed

transferOwnership

```
function transferOwnership(address newOwner) external nonpayable
```

Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the current owner.

Parameters

Name	Type	Description
newOwner	address	address of the new owner

updateSharesOwner

```
function updateSharesOwner(address _newRecipient, address _currentRecipient) external nonpayable
```

Update share recipient

Parameters

Name	Type	Description
_newRecipient	address	address of the new recipient
_currentRecipient	address	address of the current recipient

Events

NewSharesOwner

```
event NewSharesOwner(address indexed oldRecipient, address indexed newRecipient)
```

Parameters

Name	Type	Description
oldRecipient `indexed`	address	old recipient address
newRecipient `indexed`	address	new recipient address

OwnershipTransferred

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

Parameters

Name	Type	Description
previousOwner `indexed`	address	address of the previous owner
newOwner `indexed`	address	address of the new owner

TokensTransferred

```
event TokensTransferred(address indexed account, uint256 amount)
```

Parameters

Name	Type	Description
account `indexed`	address	account address
amount	uint256	amount of LOOKS tokens transferred

Last updated on ***Aug 28, 2023***

TradingRewardsDistributor

It distributes LOOKS tokens with rolling Merkle airdrops.

Methods

BUFFER_ADMIN_WITHDRAW

```
function BUFFER_ADMIN_WITHDRAW() external view returns (uint256)
```

Returns

Name	Type	Description
BUFFER_ADMIN_WITHDRAW	uint256	buffer (in epoch time) for withdrawing LOOKS tokens

amountClaimedByUser

```
function amountClaimedByUser(address) external view returns (uint256)
```

Parameters

Name	Type	Description
user	address	address of the user

Returns

Name	Type	Description
-	uint256	total cumulative amount of LOOKS tokens claimed by the user address

canClaim

```
function canClaim(address user, uint256 amount, bytes32[] merkleProof) external view returns (bool)
```

Check whether it is possible to claim and how much based on previous distribution

Parameters

Name	Type	Description
user	address	address of the user
amount	uint256	amount to claim
merkleProof	bytes32[]	array with the merkle proof

Returns

Name	Type	Description
-	bool	whether the user can claim
-	uint256	how many tokens the user can receive

claim

```
function claim(uint256 amount, bytes32[] merkleProof) external nonpayable
```

Claim pending rewards

Parameters

Name	Type	Description
amount	uint256	amount to claim
merkleProof	bytes32[]	array containing the merkle proof

currentRewardRound

```
function currentRewardRound() external view returns (uint256)
```

Returns

Name	Type	Description
currentRewardRound	uint256	current reward round

hasUserClaimedForRewardRound

```
function hasUserClaimedForRewardRound(uint256, address) external view returns (bool)
```

Parameters

Name	Type	Description
rewardRound	uint256	reward round
user	address	address of the user

Returns

Name	Type	Description
-	bool	whether the user claimed at a given reward round

lastPausedTimestamp

```
function lastPausedTimestamp() external view returns (uint256)
```

Returns

Name	Type	Description
lastPausedTimestamp	uint256	timestamp (epoch) for the last paused

looksRareToken

```
function looksRareToken() external view returns (contract IERC20)
```

Returns

Name	Type	Description
looksRareToken	contract IERC20	address of the LooksRareToken

merkleRootOfRewardRound

```
function merkleRootOfRewardRound(uint256) external view returns (bytes32)
```

Parameters

Name	Type	Description
reward round	uint256	reward round

Returns

Name	Type	Description
merkleRoot	bytes32	merkle root used for the reward round

merkleRootUsed

```
function merkleRootUsed(bytes32) external view returns (bool)
```

Parameters

Name	Type	Description
merkleRoot	bytes32	merkle root

Returns

Name	Type	Description
-	bool	whether this merkle root has been used in this contract

owner

```
function owner() external view returns (address)
```

Returns the address of the current owner.

Returns

Name	Type	Description
owner	address	address of the current owner

pauseDistribution

```
function pauseDistribution() external nonpayable
```

Pause distribution

paused

```
function paused() external view returns (bool)
```

Returns true if the contract is paused, and false otherwise.

Returns

Name	Type	Description
-	bool	whether the contract is paused (true if paused, false if unpaused)

renounceOwnership

```
function renounceOwnership() external nonpayable
```

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.

transferOwnership

```
function transferOwnership(address newOwner) external nonpayable
```

Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the current owner.

Parameters

Name	Type	Description
newOwner	address	address of the new owner

unpauseDistribution

```
function unpauseDistribution() external nonpayable
```

Unpause distribution

updateTradingRewards

```
function updateTradingRewards(bytes32 merkleRoot) external nonpayable
```

Update trading rewards with a new merkle root

It automatically increments the currentRewardRound

Parameters

Name	Type	Description
merkleRoot	bytes32	root of the computed merkle tree

withdrawTokenRewards

```
function withdrawTokenRewards(uint256 amount) external nonpayable
```

Transfer LOOKS tokens back to owner

It is for emergency purposes

Parameters

Name	Type	Description
amount	uint256	amount to withdraw

Events

OwnershipTransferred

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

Parameters

Name	Type	Description
previousOwner `indexed`	address	address of the previous owner
newOwner `indexed`	address	address of the new owner

Paused

```
event Paused(address account)
```

Parameters

Name	Type	Description
account	address	-

RewardsClaim

```
event RewardsClaim(address indexed user, uint256 indexed rewardRound, uint256 amount)
```

Parameters

Name	Type	Description
user `indexed`	address	user address
rewardRound `indexed`	uint256	reward round
amount	uint256	amount of LOOKS tokens claimed

TokenWithdrawnOwner

```
event TokensWithdrawnOwner(uint256 amount)
```

Parameters

Name	Type	Description
amount	uint256	amount of LOOKS tokens withdrawn by the owner

Unpaused

```
event Unpaused(address account)
```

Parameters

Name	Type	Description
account	address	-

UpdateTradingRewards

```
event UpdateTradingRewards(uint256 indexed rewardRound)
```

Parameters

Name	Type	Description
rewardRound `indexed`	uint256	reward round

Last updated on Aug 28, 2022

LooksRare Exchange v1 Subgraph

Description

> ⚠ CAUTION
>
> **The LooksRare Exchange v1 is deprecated.**
>
> This Subgraph does not return live data since the Exchange v1 is not enabled anymore.

This subgraph can be used to access volume, royalties for collections, execution strategies, and users of the LooksRare Exchange v1.

How can I access?

- The Graph's Playground: https://thegraph.com/hosted-service/subgraph/looksrare/exchange
- Hosted Service Query endpoint: https://api.thegraph.com/subgraphs/name/looksrare/exchange
- Mainnet Query endpoint: `https://gateway.thegraph.com/api/<API_KEY>/subgraphs/id/HJaSxvt1vH2nawRS53fGdxSrBbUZnYh52V4k8MjiPa7t`

Learn more

- Queries Examples
- Subgraph Entities

Helpful Links

- Querying from an application
- Managing your API key & setting your indexer preferences

Last updated on ***Aug 28, 2023***

LOOKS Distribution Subgraph

Description

This subgraph tracks LOOKS and WETH distribution for all users of the LooksRare ecosystem from four different smart contracts:

- `FeeSharingSystem` — This contract handles the distribution of WETH (from fee-sharing) and auto-compounds LOOKS tokens across users.
- `LooksRareAirdrop` — This contract has distributed LOOKS tokens based on past volumes from an other NFT marketplace using a Merkle tree.
- `StakingPoolForUniswapV2LPTokens` — This contract previously distributed LOOKS tokens for Uniswap V2 LP tokens "staked" on the contract.
- `TradingRewardsDistributor` — This contract distributes LOOKS to users of the `LooksRareExchange` based on rolling Merkle trees based on daily trading volumes.

How can I access?

- The Graph's Playground: https://thegraph.com/hosted-service/subgraph/looksrare/looks-distribution
- Query endpoint: https://api.thegraph.com/subgraphs/name/looksrare/looks-distribution

Learn more

- Queries Examples
- Subgraph Entities

*Last updated on **Aug 28, 2023***

Exchange v1 Subgraph Examples

> ⚠ CAUTION
>
> **The LooksRare Exchange v1 is deprecated.**
>
> This Subgraph does not return live data since the Exchange v1 is not enabled anymore.

Sample Queries

Below are some sample queries to help you get started with the LooksRare Exchange Subgraph.

You can build your own queries using a GraphQL Explorer and enter your endpoint to limit your queries to the exact data desired.

Collection Totals

Description: Get the collections total stats

```
query collectionTotals(
  $collection: String = "0x23581767a106ae21c074b2276d25e5c3e136a68b"
) {
  collection(id: $collection) {
    id
    totalVolume
    totalRoyaltyPaid
    totalTransactions
  }
}
```

Collection Transactions

Description: Get the 10 most recent transactions for a collection.

```
query collectionTransactions(
  $collection: String = "0x23581767a106ae21c074b2276d25e5c3e136a68b"
) {
  transactions(
    where: { collection: $collection }
    orderBy: block
    orderDirection: desc
    first: 10
  ) {
    block
    date
    isTakerAsk
    price
    tokenId
    taker {
      id
    }
    maker {
      id
    }
  }
}
```

Collection Daily Stats

Description: Get the collection's daily metrics for the last 7 trading days (days will skip if no trades occur)

```
query collectionDailyStats(
  $collection: String = "0x23581767a106ae21c074b2276d25e5c3e136a68b"
) {
  collectionDailyDatas(
    where: { collection: $collection }
    orderBy: date
    orderDirection: desc
    first: 7
  ) {
    dailyRoyalty
    dailyVolume
    dailyTransactions
    collection {
      id
    }
  }
}
```

Collection Royalties

Description: Get the 10 most recent royalty payments to a collection.

```
query collectionRoyalties(
  $collection: String = "0x23581767a106ae21c074b2276d25e5c3e136a68b"
) {
  royaltyTransfers(
    where: { collection: $collection }
    first: 10
    orderBy: block
    orderDirection: desc
  ) {
    tokenId
    block
    amount
    id
  }
}
```

Last updated on ***Aug 28, 2023***

Exchange v1 Subgraph Entities

> ⚠ CAUTION
> **The LooksRare Exchange v1 is deprecated.**
>
> This Subgraph does not return live data since the Exchange v1 is not enabled anymore.

Subgraph Entities

- Subgraph Entities
- Collection
- ExecutionStrategy
- User
- CollectionDailyData
- ExchangeDailyData
- ExecutionStrategyDailyData
- UserDailyData
- RoyaltyTransfer
- Transaction

Collection

Field	Type	Description
id	ID!	
totalTransactions	BigInt!	Total transactions for the Collection
totalTakerBidTransactions	BigInt!	Total transactions where the taker accepted a listed NFT
totalTakerAskTransactions	BigInt!	Total transactions where the taker accepted an offer on an NFT
totalVolume	BigDecimal!	Total ETH Volume for the Collection
totalTakerBidVolume	BigDecimal!	Total ETH Volume where the taker accepted a listed NFT
totalTakerAskVolume	BigDecimal!	Total ETH Volume where the taker accepted an offer on an NFT
totalRoyaltyPaid	BigDecimal!	Total Royalties paid to the collection
dayData	CollectionDailyData!	Daily metrics for the collection
royaltyTransfers	RoyaltyTransfer!	Royalty transfers to the collection
transactions	Transaction!	Transactions for the collection

ExecutionStrategy

Field	Type	Description
id	ID!	_
protocolFee	BigInt!	Protocol fee (e.g., 500 --> 5%)
totalTransactions	BigInt!	Total number of transactions
totalTakerBidTransactions	BigInt!	Total number of taker bid transactions
totalTakerAskTransactions	BigInt!	Total number of taker ask transactions
totalVolume	BigDecimal!	Total volume (in ETH)
totalTakerBidVolume	BigDecimal!	Total taker bid volume (in ETH)
totalTakerAskVolume	BigDecimal!	Total taker ask volume (in ETH)
dayData	ExecutionStrategyDailyData!	Daily history for execution strategy
transactions	Transaction!	Transactions for execution strategy

User

Field	Type	Description
id	ID!	_
totalTransactions	BigInt!	Total transactions for a user
totalAskVolume	BigDecimal!	Total volume where the user sold an NFT
totalBidVolume	BigDecimal!	Total volume where the user bought an NFT
totalMakerVolume	BigDecimal!	Total volume where the user was the maker
totalTakerVolume	BigDecimal!	Total volume where the user was the taker
totalVolume	BigDecimal!	Total volume for the user
totalRoyaltyCollected	BigDecimal!	Total royalties collected by the user
dayData	UserDailyData!	Users daily metrics
makerTransactions	Transaction!	Users maker transactions
takerTransactions	Transaction!	Users taker transactions
royaltyPayments	RoyaltyTransfer!	Users royalty transactions

CollectionDailyData

Field	Type	Description
id	ID!	_
date	BigInt!	Start date (timestamp)
collection	Collection!	Collection info
dailyTransactions	BigInt!	Daily unique transactions
dailyTakerBidTransactions	BigInt!	Daily unique taker bid transactions
dailyTakerAskTransactions	BigInt!	Daily unique taker ask transactions
dailyVolume	BigDecimal!	Total volume for the day (in ETH)
dailyTakerBidVolume	BigDecimal!	Daily taker bid volume (in ETH)
dailyTakerAskVolume	BigDecimal!	Daily taker ask volume (in ETH)
dailyVolumeExcludingZeroFee	BigDecimal!	Daily volume (in ETH) excluding zero-fee strategy
dailyRoyalty	BigDecimal!	Royalty payment (in ETH)

ExchangeDailyData

Field	Type	Description
id	ID!	_
date	BigInt!	Start date (timestamp)
dailyUsers	BigInt!	Daily unique users
dailyCollections	BigInt!	Daily unique collections traded
dailyTransactions	BigInt!	Daily unique transactions
dailyTakerBidTransactions	BigInt!	Daily taker ask transactions
dailyTakerAskTransactions	BigInt!	Daily taker ask transactions
dailyVolume	BigDecimal!	Daily volume (in ETH)
dailyTakerBidVolume	BigDecimal!	Daily taker bid volume (in ETH)
dailyTakerAskVolume	BigDecimal!	Daily taker ask volume (in ETH)
dailyVolumeExcludingZeroFee	BigDecimal!	Daily volume (in ETH) excluding zero-fee strategy

ExecutionStrategyDailyData

Field	Type	Description
id	ID!	_
strategy	ExecutionStrategy!	Execution strategy
date	BigInt!	Start date (timestamp)
dailyTransactions	BigInt!	Daily unique transactions
dailyTakerBidTransactions	BigInt!	Daily unique taker bid transactions
dailyTakerAskTransactions	BigInt!	Daily unique taker bid transactions
dailyVolume	BigDecimal!	Daily volume (in ETH)
dailyTakerBidVolume	BigDecimal!	Daily taker bid volume (in ETH)
dailyTakerAskVolume	BigDecimal!	Daily taker ask volume (in ETH)

UserDailyData

Field	Type	Description
id	ID!	ID
date	BigInt!	Start date (timestamp)
user	User!	User
dailyTransactions	BigInt!	Daily unique transactions
dailyVolume	BigDecimal!	Daily volume (in ETH)
dailyVolumeExcludingZeroFee	BigDecimal!	Daily volume (in ETH) excluding zero-fee strategy

RoyaltyTransfer

Field	Type	Description
id	ID!	ID
date	BigInt!	Timestamp of Royalty Transfer
block	BigInt!	Block number
collection	Collection!	Collection
tokenId	BigInt!	Token Identifier
user	User!	User
amount	BigDecimal!	Amount received (in ETH)

Transaction

Field	Type	Description
id	ID!	ID
date	BigInt!	Date (timestamp)
block	BigInt!	Block number
collection	Collection!	Collection
isTakerAsk	Boolean!	Whether the transaction is a takerAsk
strategy	ExecutionStrategy!	Execution strategy
tokenId	BigInt!	Token Identifier
price	BigDecimal!	Price (in ETH)
maker	User!	Maker user
taker	User!	Taker user

Last updated on Aug 28, 2023

LOOKS Distribution Subgraph Examples

Sample Queries

Below are some sample queries to help you get started with the LOOKS Distribution Subgraph.

You can build your own queries using a GraphQL Explorer and enter your endpoint to limit your queries to the exact data desired.

Get Pending Rewards for User

`Note: user address should be all lowercase`

```
query GetPendingRewards($user: String = "0x123...") {
  user(id: $user) {
    tradingRewardsAmount
    listingRewardsAmount
    tradingRewardsLastClaimDate
    listingRewardsLastClaimDate
    feeSharingAdjustedDepositAmount
    aggregatorAdjustedDepositAmount
    feeSharingLastWithdrawDate
    feeSharingLastHarvestDate
    aggregatorLastDepositDate
  }
}
```

Get Recent Snapshots

Description: Get the recent user activity metric snapshots

```
query GetSnapshots {
  dailySnapshots(first: 500, orderBy: date, orderDirection: desc) {
    id
    date
    aggregatorActiveUsers
    aggregatorTotalStakedLOOKS
    feeSharingActiveUsers
    feeSharingTotalStakedLOOKS
  }
}
```

Get LOOKS Poundin'

Description: Get the recent conversions of ETH to Looks conversions from compounding staking

```
query QueryName {
  aggregatorConversions(first: 1000, orderBy: block, orderDirection: desc) {
    id
    block
    amountSold
    amountReceived
    priceOfETHInLOOKS
  }
}
```

*Last updated on **Aug 28, 2023***

LOOKS Distribution Subgraph Entities

LOOKS Distribution Subgraph Entities

- LOOKS Distribution Subgraph Entities
- User
- DailySnapshot
- Overview
- RewardPeriod
- AggregatorConversion

User

Field	Type	Description
id	ID!	User Id
airdropAmount	BigDecimal!	Airdrop amount claimed (LOOKS)
airdropClaimDate	BigInt!	Airdrop claim date
aggregatorAdjustedDepositAmount	BigDecimal!	Aggregator adjusted deposit amount (LOOKS)
aggregatorTotalCollectedLooks	BigDecimal!	Aggregator total LOOKS collected
aggregatorLastDepositDate	BigInt!	Aggregator last deposit date
aggregatorLastWithdrawDate	BigInt!	Aggregator last withdraw date
aggregatorIsActive	Boolean!	Checks wether aggrgator is active or not
feeSharingAdjustedDepositAmount	BigDecimal!	Fee sharing - Adjusted deposit amount (LOOKS)
feeSharingTotalCollectedLooks	BigDecimal!	Fee Sharing - Total LOOKS collected
feeSharingTotalCollectedWETH	BigDecimal!	Fee Sharing - Total WETH collected
feeSharingLastDepositDate	BigInt!	Fee Sharing - Last deposit date
feeSharingLastHarvestDate	BigInt!	Fee Sharing - Last harvest date
feeSharingLastWithdrawDate	BigInt!	Fee Sharing - Last withdraw date
feeSharingIsActive	Boolean!	Checks wether fee sharing is active or not
listingRewardsAmount	BigDecimal!	Listing Rewards - Total LOOKS collected
listingRewardsLastClaimDate	BigInt!	Listing Rewards - Last claim date
tradingRewardsAmount	BigDecimal!	Trading Rewards - Total LOOKS collected
tradingRewardsLastClaimDate	BigInt!	Trading Rewards - Last claim date
stakingPoolUniswapV2TotalCollectedLOOKS	BigDecimal!	UniswapV2 Staking - Total LOOKS collected
stakingPoolUniswapV2LastDepositDate	BigInt!	UniswapV2 Staking - Last deposit date
stakingPoolUniswapV2LastHarvestDate	BigInt!	UniswapV2 Staking - Last harvest date
stakingPoolUniswapV2LastWithdrawDate	BigInt!	UniswapV2 Staking - Last withdraw date

DailySnapshot

Field	Type	Description
id	ID!	Daily Snapshot Id
date	BigInt!	Daily Snapshot date
activeStakers	BigInt!	Number of unique active stakers (Aggregator + Fee Sharing)
newStakers	BigInt!	Daily new stakers (Aggregator + Fee Sharing)
removedStakers	BigInt!	Daily removed stakers (Aggregator + Fee Sharing)
aggregatorActiveUsers	BigInt!	Aggregator - Number of active users
aggregatorNewUsers	BigInt!	Aggregator - Daily new users
aggregatorRemovedUsers	BigInt!	Aggregator - Daily removed users
aggregatorDailyInflowLOOKS	BigDecimal!	Aggregator - Daily inflow (LOOKS)
aggregatorDailyOutflowLOOKS	BigDecimal!	Aggregator - Daily outflow (LOOKS)
aggregatorTotalStakedLOOKS	BigDecimal!	Aggregator - Total LOOKS staked
aggregatorTotalWETHSold	BigDecimal!	Aggregator - Total WETH sold
aggregatorTotalLOOKSReceived	BigDecimal!	Aggregator - Total LOOKS received from sales
aggregatorConversionCount	BigInt!	Aggregator - Number of conversions
feeSharingActiveUsers	BigInt!	Fee Sharing - Number active users
feeSharingNewUsers	BigInt!	Fee Sharing - Daily new users
feeSharingRemovedUsers	BigInt!	Fee Sharing - Daily removed users
feeSharingDailyInflowLOOKS	BigDecimal!	Fee Sharing - Daily inflow (LOOKS)
feeSharingDailyOutflowLOOKS	BigDecimal!	Fee Sharing - Daily outflow (LOOKS)
feeSharingTotalStakedLOOKS	BigDecimal!	Fee Sharing - Total LOOKS staked

Overview

Field	Type	Description
id	ID!	Overview Id
activeStakers	BigInt!	Number of unique active stakers (Aggregator + Fee Sharing)
aggregatorActiveUsers	BigInt!	Aggregator - Number of active users
aggregatorTotalStakedLOOKS	BigDecimal!	Aggregator - Total LOOKS staked
feeSharingActiveUsers	BigInt!	Fee Sharing - Number active users
feeSharingTotalStakedLOOKS	BigDecimal!	Fee Sharing - Total LOOKS staked

RewardPeriod

Field	Type	Description
id	ID!	ID (transaction hash)
block	BigInt!	Block number
date	BigInt!	Date (timestamp)
numberBlocks	BigInt!	Period length (in blocks)
reward	BigDecimal!	Reward amount for the period (in WETH)
rewardPerBlock	BigDecimal!	Reward per block for the period (in WETH)

AggregatorConversion

Field	Type	Description
id	ID!	ID (transaction hash)
block	BigInt!	Block number
date	BigInt!	Date (timestamp)
amountSold	BigDecimal!	Reward sold (in WETH)
amountReceived	BigDecimal!	Reward received (in LOOKS)
priceOfETHInLOOKS	BigDecimal!	Price of 1 WETH in LOOKS

Last updated on ***Aug 28, 2023***

Royalty Fee Registry Subgraph

Description

This subgraph tracks changes of royalty fee information from the RoyaltyFeeRegistry smart contract.

> NFT contracts that implement the ERC-2981 interface **are not registered in the RoyaltyFeeRegistry**.

How can I access?

- The Graph's Playground: https://thegraph.com/hosted-service/subgraph/looksrare/royalty-fee-registry
- Query endpoint: https://api.thegraph.com/subgraphs/name/looksrare/royalty-fee-registry

*Last updated on **Aug 28, 2023***

Previous
« LOOKS Distribution Subgraph Entities