# Enjin Documentation

# Enjin Documentation

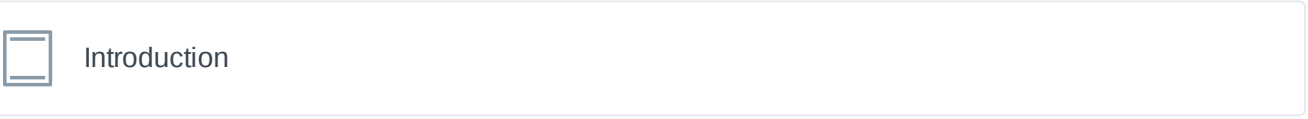
## Enjin Docs Navigation

A page where users can find quick links to specific sections in our documentation!

## Enjin Documentation Pages Summary & Quick Links

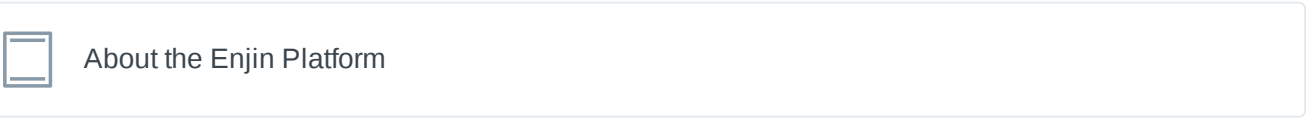
### Introduction

Cover the basics in our documentation so devs can get up-to-speed with blockchain terms!



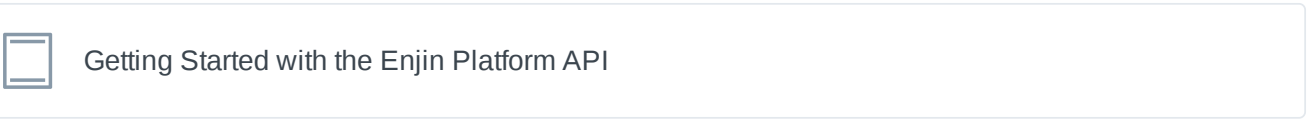

### About the Enjin Platform

Learn more about the platform and our amazing products!



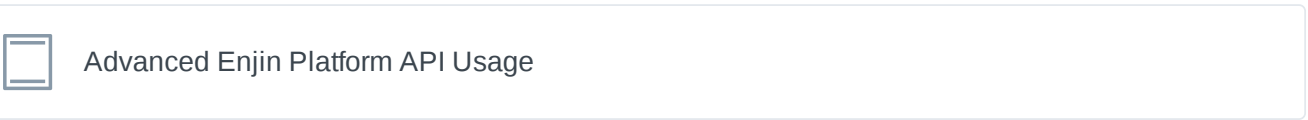

### Getting started with the Enjin Platform API

Straightforward guides that will help you get started with the API.




### Advanced Enjin Platform API Usage

Learn more about advanced usage of the Platform API.



### EnjinX API

EnjinX API

### Wallet Daemon

Want to learn more about the Wallet Daemon and how you can use it to automate transactions?

Wallet Daemon

### Enjin Platform SDKs

Our SDKs section covers everything you need to know about our SDKs and how to use it from scratch!

Enjin Platform SDKs

### Downloads

| Resource | Download Link |
| --- | --- |
| C# SDK | https://github.com/enjin/enjin-csharp-sdk |
| C++ SDK | https://github.com/enjin/enjin-cpp-sdk |
| Java SDK | https://github.com/enjin/enjin-java-sdk |
| Wallet Daemon | https://cdn.enjin.io/downloads/enjin-wallet-daemon/latest |
| EnjinCraft Plugin | https://www.spigotmc.org/resources/enjincraft.794 |
| Wallet App | https://enjin.io/products/wallet |

# Introduction

# About Enjin

Learn more about Enjin's history across Gaming and Blockchain history.

Enjin was founded in 2009 by Maxim Blagov and Witek Radomski.

In the same year, the company launched the Enjin Network, a gaming community platform that has since grown to over 20 million registered users across 250,000 gaming communities.

In 2017 following a successful ICO that raised $18.9 million, Enjin established itself as a leading blockchain ecosystem developer, building a suite of user-first blockchain products that enable anyone to easily manage, explore, distribute, and integrate blockchain assets.

Early 2018 saw the release of Enjin’s first blockchain product, the Enjin Wallet, which boasts over 1 million downloads on Google Play and the App Store.

The first public showcase of the Enjin ecosystem took place at GDC 2018, followed by appearances at E3, Unity Unite events, and other numerous gaming and blockchain conferences in the USA, France, Germany, South Korea, Singapore, and more countries across the globe.

In mid-2018, Enjin launched an Early Adopter Program geared toward game developers, which closed in 2019. Currently, there are 16 studios are using the Enjin Platform to create and integrate blockchain technology into 40 games, apps, and websites.

In August 2018, Enjin kick-started the world’s first decentralized gaming multiverse by creating the first set of blockchain assets usable across multiple games and facilitating collaboration between six Early Adopter studios that agreed to implement them. The gaming multiverse has since grown to 30 games.

November 2018 brought the launch of the Enjin Mintshop, an on-demand blockchain asset creation service, followed by Enjin Beam, a QR-powered blockchain asset distribution service. Shortly after in December 2018, the company launched EnjinX, an ad-free, user-friendly blockchain explorer.

In March 2019, Enjin launched the Testnet version of their blockchain game development platform, the Enjin Platform, as well as the Blockchain SDK for Unity. This was followed by the release of Enjin's Java SDK alongside "EnjinCraft," a demo Minecraft server create to showcase the robust blockchain integration made possible by Enjin's upcoming Minecraft plugin.

In June 2019, exactly one year after Enjin CTO Witek Radomski pushed the first version of it to Ethereum's Github repository, the ERC-1155 Multi Token Standard reached final status and was adopted by Ethereum as an official token standard.

September 2019 brought the release of the much-awaited Enjin Marketplace, a hub for trading ERC-1155 digital assets, powered by the Enjin Explorer and Wallet.

In February 2020, the Enjin Platform was released on Ethereum Mainnet. allowing anyone to mint blockchain assets with Enjin Coin.

Since the completion of its ICO, Enjin's community has grown to a consolidated 600,000+ engaged members on Telegram, Twitter, Reddit, Facebook, and other messaging and social platforms.

You can learn more about our updated timeline here - https://enjin.io/timeline

## Why use Enjin?

Built on top of a robust on-chain infrastructure and comprised of the Enjin Platform, Enjin Marketplace, Wallet, Beam, and other tools and services, the Enjin ecosystem enables game and app developers to increase revenue, gain a competitive edge, and innovate in previously impossible ways.

Forged in gaming, Enjin's tools and services can also be used by companies of all sizes and industries seeking to create blockchain products or utilize tokenized digital assets as part of their acquisition, retention, engagement, and monetization strategies.

The Enjin Platform is a blockchain PaaS (Platform as a Service) that allows you to create and manage blockchain games without the complexity of building and maintaining the infrastructure typically associated with developing and launching a blockchain game, it is a robust, flexible, powerful suite of tools and services for creating groundbreaking blockchain games, it is an all-in-one blockchain game development platform comprised of Blockchain SDKs, Platform API, Wallet Daemon, and Efinity.


The Enjin Platform allows you to -

- **Utilize Innovative Crowdfunding Models:** Create your characters, weapons, real estate, and other gaming assets in advance and offer them to gamers via traditional crowdfunding platforms or a stand-alone website.

- **Enable Player-Driven Value Creation:** Enable your users to modify, craft, upgrade, build, and trade blockchain-based gaming assets which can then gain intrinsic, real-life value of their own based on their history, stats, input, and player customization.
- **Gain User Loyalty and Trust:** Give each player an unbreakable bond of trust: true ownership of digital gaming assets value-backed with ENJ. This makes your users not only more loyal but also more likely to make a purchasing decision.
- **Recapture Lost Revenue:** Eliminate or regulate gray market trading, which can amount to over 40% in lost revenue. You can then monetize all trading—whether online, in-game, or peer-to-peer—via implementing blockchain-enforced trading fees.
- **Build and Partner with Gaming Multiverses:** Build a gaming multiverse of your own, enabling players to use and even level up their characters and items across sequels or entirely different games—or collaborate with other studios to do the same.
- **Reduce Player Churn:** Gaming assets are owned by gamers, and safely kept in their private Enjin Wallet. Players must approve each transaction via their wallet, thus making any hacking or in-game fraud impossible—and in turn reducing player churn.

But not just with games, the Enjin Platform can also empower a few other use-cases, which are -

- Art: Turned into NFTs, your artwork can be easily displayed in virtual galleries—or anywhere in the Metaverse; it can also double as proof of ownership that is connected to the physical art piece
- Sports: Mint NFT Collectibles, own Sports moments and memorabilia, engage your sponsors, and create memories that truly last a lifetime.
- Music: Truly own music, create and produce music using new technology and have exclusive rights; turning it into a powerful token
- Real-World: Tokenize anything from old comics and rare paintings to real state and physical products.
- Collectibles: Create easily tradable, programmable, scarce digital collectibles that tell a meaningful story.

Today, there are millions of ENJ & JENJ locked in blockchain-based digital assets.

## What is Ethereum?

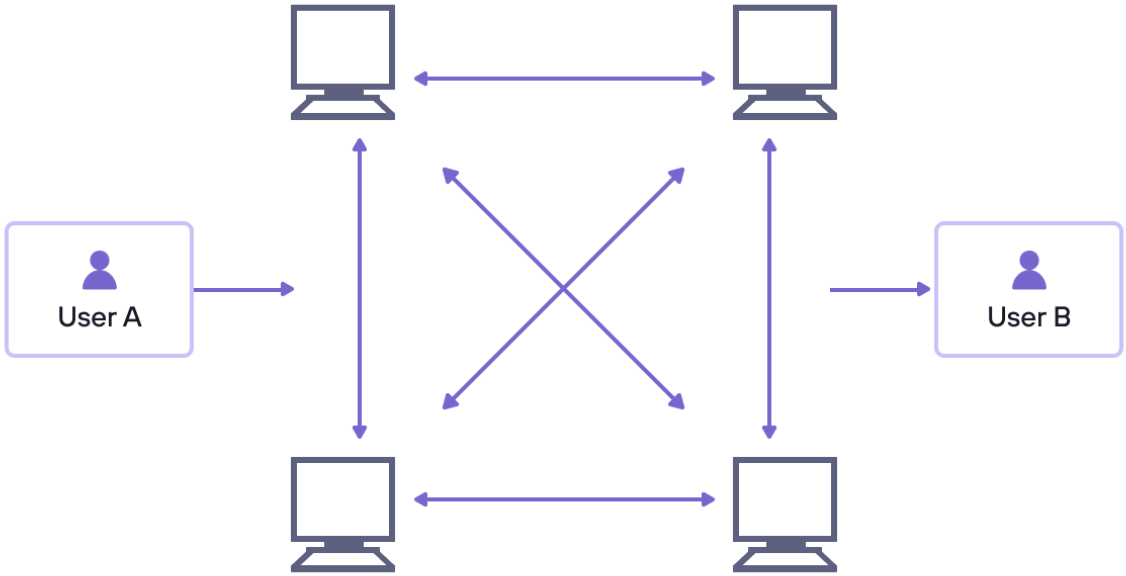
Learn more about Ethereum.

In the Ethereum blockchain, there's a single, canonical computer called the Ethereum Virtual Machine, or the EVM, whose state everyone on the Ethereum network agrees on. Everyone who participates in the Ethereum blockchain (nodes) keeps a copy of the current state of this specific computer. Any participant can broadcast and send a request for this computer to perform arbitrary computations. Whenever such a request is sent, other actors in the network can verify, validate and proceed with this specific computation. This specific execution causes a state change in the EVM, which is committed and propagated throughout the network.

Requests for computations in the blockchain are called transaction requests; the record of every transaction

and the current state of the EVM are stored on the blockchain, which is also stored and agreed upon by all nodes.

Cryptographic mechanisms ensure that once a transaction has been verified as valid and added to the blockchain, it cannot be altered later. The same mechanism also ensures that all transactions are signed and executed with appropriate permissions, for example, no one should be able to send funds from Witek's account, except for Witek himself.




Blockchain structure

**What is ETH?**

Ether (ETH) is the main cryptocurrency token of the Ethereum Network. The purpose of ETH is to allow market computations; this market provides an economic incentive to participants to verify, execute transactions, and provide computational resources to the network.

Any participant in the blockchain who is broadcasting a transaction must also provide an amount of ETH to the network. This amount will be awarded to the actor who's verifying this transaction, executing it, and broadcasting it to the network.

The amount of ETH paid while performing a transaction in the blockchain is equivalent to the time required to perform the computation in the blockchain. These ETH bounties also prevent malicious actors from flooding the network by requesting the execution of infinite computations or other scripts as these participants must pay for the computation time.

**What are Smart Contracts?**

As a basic example, smart contracts work like sale scripts; when called with specific parameters, they can perform specific actions attached to that smart contract should pre-existing conditions be fulfilled. For example, a sales smart contract could mint and assign the ownership of a digital asset if the caller sends ETH to a specific recipient.

Any developer can create and deploy a smart contract that would be public to the network. With the Enjin Platform, there is no need to deploy developer smart contracts to create and mint NFTs as this is going to be handled by interactions between the dev wallet and Enjin's ERC-1155 smart contract.

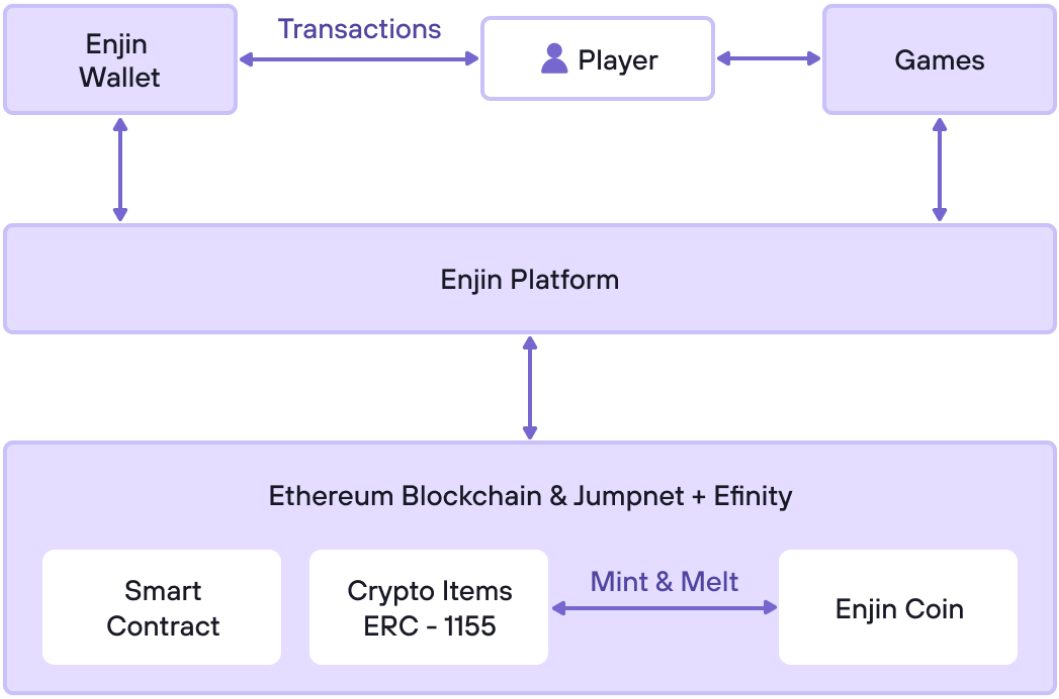
> For more information on the Ethereum blockchain, we would also recommend reaching out to the Ethereum foundation docs on this link here.

## What is Enjin Coin?

Learn more about Enjin Coin (ENJ), the gold standard for digital assets.

Enjin Coin (ENJ) is an Ethereum-based cryptocurrency used to directly back the value of next-generation blockchain assets.

Every digital asset created with the Enjin Platform is an ERC-1155 token backed by ENJ. The native ERC-1155 token standard was created by Enjin's Co-founder & CTO Witek Radomski. To date, over 1 billion Enjin-powered blockchain assets have already been created.



Enjin Platform interaction structure

Minting blockchain tokens with ENJ provides a variety of benefits to creators and users:

- Infusing tokens with residual value
- Ensures the transparency and scarcity of tokens
- Gives tokens instant liquidity
- Enables the utility of tokens in games and applications
- Anti-inflationary
- Melting feature that allows users to destroy blockchain tokens at any time in order to retrieve the ENJ value from such token.

> ⓘ You can find more details about Enjin Coin through our blockchain explorer page here

> ⓘ If you wish to purchase Enjin from a trustworthy exchange, you can also reach out to our website with a list of exchanges that has Enjin Coin listed.

## What is an NFT?

Learn more about NFTs and the importance of NFTs to our ecosystem.

### What is a crypto "Token"?

The term tokens refer to a digital currency or how cryptocurrencies are defined in a specific blockchain environment/project. A token represents a set of rules encoded in a smart contract and belongs to a specific blockchain address, a great example of blockchain tokens would be ETH (Ethereum) and BTC (Bitcoin). In the context of the Enjin Platform, our main Tokens are Enjin & Efinity, which are used to power up our environment and make sure our vision and mission come to fruition.

### What is an NFT?

NFTs (Non-Fungible Tokens) are unique digital assets created on the blockchain. They can be everything from gaming items to digital art, to sports collectibles and real-world assets. NFTs are based on a concept of true ownership, where your asset is stored in your blockchain wallet address and no one would be able to take it from you. NFTs can be used in apps, games, websites and even real life, the limit to its usage is based on the imagination of the NFT creator. Every NFT is unique, which means that every NFT is different and this is where the non-fungible meaning comes from.

Fungible Tokens (FTs) are blockchain assets replaceable by another identical item; they are mutually interchangeable. Gold coins, mana gems, and resources like iron, stone, and wood are fungible. One gold coin is equal to another gold coin; one piece of iron is the same as another piece of iron. These assets are identical, in terms of NFTs, the logic would be the complete opposite, where NFTs are created to represent unique tokens, as an example, rare swords, a car you named after something you like, and your own character in a game can be NFTs since these are unique.

**What is the ERC-1155 standard?**

Created by Enjin CTO Witek Radomski, ERC-1155 is an Ethereum standard (a format of standardized coding rules and data structures) that allows for infinite amounts of both fungible (identical) and non-fungible (unique) tokens in a single deployed smart contract.

Tokens standards like ERC-20 and ERC-721 require a separate contract to be deployed for each token type or collection. This places a lot of redundant bytecode on the Ethereum blockchain and limits certain functionality by the nature of separating each token contract into its own permissioned address. With the rise of blockchain games and platforms like Enjin Coin, game developers may be creating thousands of token types, and a new type of token standard is needed to support them. However, ERC-1155 is not specific to games and many other applications can benefit from this flexibility.

ERC-1155 is also referred to as the “Multi Token Standard.”

It was adopted by Ethereum as an official token standard in June 2019.

**Features**

- **Multiple Tokens:** Developers can define and configure both fungible and non-fungible tokens within a single smart contract.
- **Advanced Capabilities:** Users can trade, destroy, use, upgrade, combine, rent, loan, and lose their assets.
- **Gas Saving:** Using the Multi Token Standard can cut gas fees by up to 90% when minting new tokens.
- **Atomic Swaps:** Enables swaps of any number of tokens in two simple steps.
- **Multi-Transfers:** This Allows users to send tokens to multiple recipients in a single transaction.


The ERC-1155 standard

**Reference links for ERC-1155**

- GitHub: https://eips.ethereum.org/EIPS/eip-1155
- Original issue thread: https://github.com/ethereum/EIPs/issues/1155

**How is an NFT created in the context of the Enjin Platform?**

By abstracting the Ethereum back-end interactions, developers and users can create their NFTs by only focusing on what is important for them, without needing to understand how the Enjin Smart Contract works, and without any sort of coding expertise needed.

To create an NFT in the Enjin Platform, a developer would need to have an amount of ENJ or JENJ, create an Enjin Platform account, create a collection and link your Enjin Wallet to that specific collection, after doing that, the process of creating the NFT would only involve a few parameters presented by the Enjin Platform, such as the amount of NFTs being created, Enjin backing, transfer fees and a few other parameters that we will cover in the Enjin Platform API tutorials.

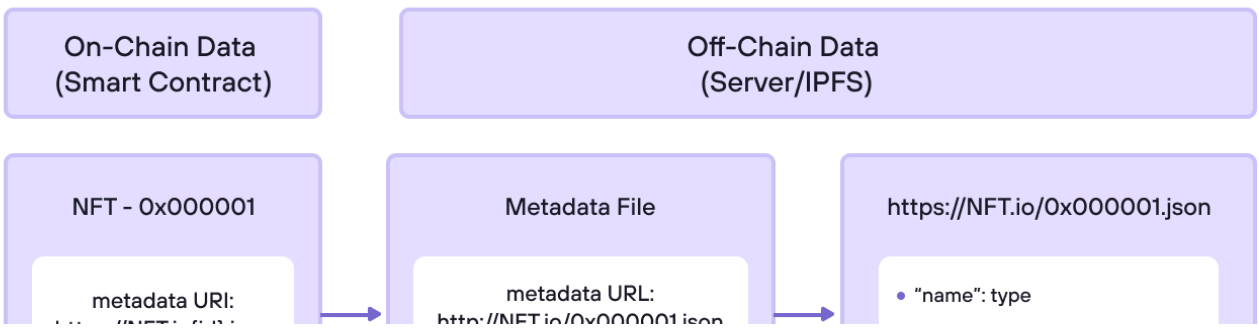
**What is Metadata?**

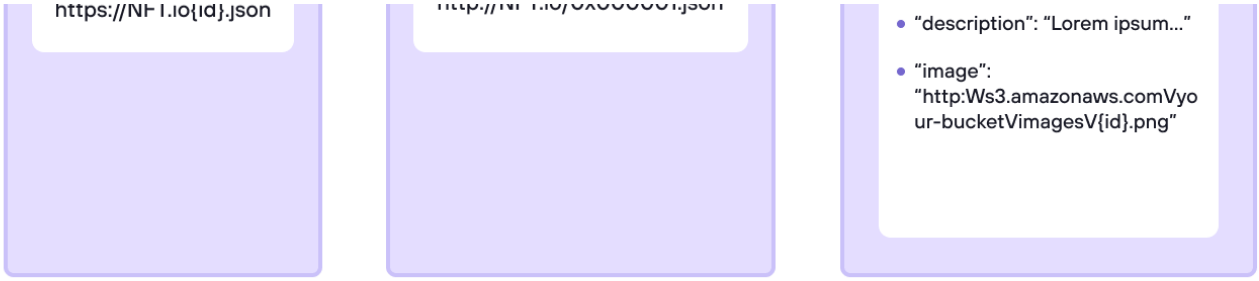
Metadata, in the context of programming, refers to data of the containers of data, as a description of a resource, structural information about an object, or reference such as the contents of any piece of information.

In the context of the FT & NFT world, metadata refers to a piece of data that gives the context of what an FT or NFT actually is, each NFT in the Enjin Platform contains a "metadata" parameter that it can be referred to, which gives information on what that specific NFT is. As an example, this specific metadata file can contain a name, image, description, and other rich information about that specific NFT.

With the Enjin Platform, developers are able to store images, gifs, and videos as a way to enrich the NFTs created and improve the user experience.

This information is stored on a public online repository off-chain so that information can be fetched and read by users and any supported platform that may interact with that specific NFT.






# What is Jumpnet?

Instant, secure, carbon-negative, and free on-chain transactions for NFTs and Enjin Coin.

### What is Jumpnet?

**JumpNet** is a private (PoA) method of consensus blockchain that allows instant and gasless transactions on-chain.

PoA stands for Proof of Authority, which is an algorithm used within Jumpnet to deliver fast transactions through a consensus mechanism based on identity. With PoA, transactions and blocks are all validated by approved validators, in the case of Jumpnet, the Enjin team is the validator and does not require any sort of monitoring besides maintaining the nodes operational.

JENJ is a private version of Enjin Coin (ENJ) which operates on the JumpNet network. Every NFT you create on JumpNet is backed by JENJ.

By moving Enjin Coin (ENJ) from Ethereum to JumpNet, you will be able to do the following without gas fees:

- Send and receive ENJ and ERC-1155 tokens
- Create and mint ERC-1155 tokens via the **Enjin Platform**
- Trade ERC-1155 tokens on the **Enjin Marketplace**
- Distribute ENJ and ERC-1155 tokens via **QR codes**
- Automatically distribute ERC-1155 tokens from your app or game

### What is JENJ?

JENJ is the equivalent of ENJ on the JumpNet Network. With JENJ, you can create, mint and distribute NFTs backed by JENJ, all free and instant. To acquire JENJ, you will need to convert ENJ over to JumpNet via your Enjin Wallet.

> ⓘ If you wish to convert some of your ENJ to JENJ, you can follow this guide here.

# What is Goerli?

Learn more about Goerli Testnet and how it relates to the Enjin Platform.

### What is Goerli?

The Goerli (Görli) blockchain is a PoA (Proof-of-Authority) public cross-client testing blockchain, created and maintained by the Ethereum community to assist the Ethereum development, with Goerli, developers are allowed to perform blockchain development testing before deploying their smart contracts on the Ethereum Mainnet. It was created in 2018 during ETHBerlin.

### How does the Enjin Platform interact with Goerli?

The Enjin Platform utilizes Goerli to allow developers to integrate and test their integrations within the Enjin Platform using a safe and cost-free environment to prevent loss of funds and to test possible use-case scenarios that are only available in the Enjin Mainnet platform or through Jumpnet.

To interact with the Enjin Platform Goerli environment, you can access this link -

https://goerli.cloud.enjin.io

Please note that in order to interact with the Goerli environment, you would need to have some GENJ and GETH, which can be acquired here -

### Goerli Faucets

For GENJ - enj.in/support

For GETH - https://faucets.chain.link/goerli

> ⓘ After you have received the Testnet tokens in your Enjin Wallet, you would also need to enable developer mode in the Enjin Wallet app, which can be done by following this guide. This is also necessary in order to interact with the Testnet Environments in the Enjin Platform.

# Common Blockchain & Enjin Platform Terms

Learn more about the common and most used terms across the Blockchain and the Enjin Platform as well.

This specific section covers the meaning behind common terms used across the Enjin Platform as well as Blockchain terms that could help developers understand a bit more about the technology and get familiarized with the common wording on blockchain development.

**Blockchain Glossary**

**Enjin Platform Glossary**

# Blockchain Glossary

Learn more about common Blockchain Terms.

Here's a list of Common Blockchain & Crypto related terms used in the space that should come in handy while interacting with the Enjin Platform and with Blockchain as a whole.

### Blockchain

A blockchain is a decentralized system that records transactions made in Bitcoin or any other cryptocurrency, maintained by several computers linked in a peer-to-peer network, it can be seen as a ledger that facilitates the process of recording transactions and tracking assets.

### Block

A Block, or Blocks, is a data structure within the blockchain where transactions are recorded permanently, blocks are used to record all recent transactions that haven't been validated by the network, and once the data is completely validated, the block is closed and a new block is generated.

### ETH

Ethereum is the main cryptocurrency token of the Ethereum blockchain. ETH is used to pay for all types of transactions and interactions you make on the Ethereum blockchain. Examples include sending tokens, purchasing NFTs, performing exchanges, etc.

### EVM

Ethereum Virtual Machine (EVM) is an engine that acts as a decentralized computer having multiple executable projects. It acts as the virtual machine which is the core foundation of Ethereum's entire operating structure.

### Gas Fees

Gas fees are incentives made as payments from blockchain users for the computing energy required to process and validate transactions on the blockchain performed by the blockchain miners.

### Seed Phrase

A Seed Phrase or Recovery Phrase is a series of words generated by blockchain wallets that grant you access to the cryptocurrencies associated with that specific blockchain wallet. It should never be shared with 3rd party actors and should be maintained in a safe place to avoid loss of funds.

### Smart Contract

As a basic example, smart contracts work like sale scripts that when called with specific parameters can perform specific actions attached to that smart contract if pre-existing conditions are fulfilled. For example, a sales smart contract could mint and assign the ownership of a digital asset if the caller sends ETH to a specific recipient.

### DApp

A DApp (Decentralized Application) is an application that operates autonomously, through the use of smart contracts on the blockchain, just like conventional apps, a DApp provides utility to users who interact with the applications on the blockchain.

### Account

A blockchain account is a digital wallet that allows users to store and manage their cryptocurrencies in a safe way. One must not mistake a blockchain account with a blockchain imported on a blockchain wallet application as these are different and blockchain wallet applications work only as a window to your account which is stored in the blockchain.

### Private Key

A private key is a number used in cryptography standards, similar to a password. private keys are used to create signatures that can be verified without compromising the integrity of the private keys.

### Address

A blockchain address is a string of text that holds the location of a particular wallet on the blockchain, with a blockchain address, users can locate their funds and receive funds from other users or platforms across the blockchain.

### Node

A blockchain node is an open-source P2P protocol that allows developers to communicate with each other through the network and broadcast information regarding transactions and blocks. A node can be a computer or a server.

**Miner**

A blockchain miner is a blockchain actor who is able to confirm and validate transactions in the blockchain by running a node.

**Nonce**

Transactions can only be mined and executed sequentially on the blockchain; multiple transactions cannot be mined with the same nonce, and a nonce cannot be skipped. Since a used nonce cannot be confirmed again, the process prevents the possibility of replay attacks, where the recipient can rebroadcast the same signed transaction repeatedly to drain the sender's wallet.

**Hash**

A hash function turns any text input into a string of bytes with a specific length and structure on the blockchain, the result of that conversion is called a hash value that is used to generate unique and non-repeatable identifiers.

**FT**

FTs (Fungible tokens) are stackable blockchain tokens that have a quantity and optional decimal places. An example of a fungible token is a twenty-dollar bill - each bill is worth the same amount as another twenty-dollar bill.

**NFT**

NFTs (Non-fungible tokens) are unique digital assets created on the blockchain. They can be everything from gaming items and digital art, to sports collectibles and real-world assets. Today, NFTs are fueling the rise of new economic models and interconnected digital realities.

**ERC20**

The ERC-20 introduces a standard for Fungible Tokens, in other words, they have a property that makes each Token be exactly the same (in type and value) as another Token. For example, an ERC-20 Token acts just like the ETH, meaning that 1 Token is and will always be equal to all the other Tokens.

**ERC721**

The ERC-721 introduces a standard for NFT, in other words, this type of Token is unique and can have a different value than another Token from the same Smart Contract, maybe due to its age, rarity, or even something else like its visual**.**

**ERC1155**

A standard interface for contracts that manage multiple token types. A single deployed contract may include any combination of fungible tokens, non-fungible tokens, or other configurations (e.g. semi-fungible tokens).

**PoA**

PoA stands for Proof of Authority, which is an algorithm used by some blockchains such as Jumpnet to deliver fast transactions through a consensus mechanism based on identity on a distributed network.

### PoW

PoW stands for Proof of Work, which is a process that allows the blockchain to remain operational by using the process of mining, or recording transactions difficult.  The idea behind proof of work blockchains is to prevent manipulation of the network by establishing large energy and hardware control requirements to be able to process transactions.

### PoS

PoS stands for Proof of Stake, which is a process that allows a class of consensus mechanisms for blockchains that work by selecting validators in proportion to their quantity of holdings in the associated cryptocurrency.

### tps

TPS stands for Transactions Per Second, which is literally the number of transactions that a network is capable of processing each second while producing new blocks and confirming transactions.

### Stake

Staking is the process of participating in transaction validation or lending tokens in order to receive staking rewards from a blockchain protocol or an exchange.

## Enjin Platform Glossary

Learn more about common Enjin Platform Terms.

Here's a list of Common Enjin Platform-related terms used in the space that should come in handy while interacting with the Enjin Platform and with Blockchain as a whole.

### Enjin Platform

The Enjin Platform is a blockchain PaaS (Platform as a Service) that allows you to create and manage blockchain games without the complexity of building and maintaining the infrastructure typically associated with developing and launching a blockchain game.

### Wallet Daemon

The Wallet Daemon is a tool that you can use to automate the authorization of transaction requests to and from the Enjin Platform.

Without the Wallet Daemon, you would need to sign (authorize) every in-game blockchain transaction via the Enjin Wallet (e.g., sending a sword to a player).

Wallet Daemon manages an Ethereum address linked to an Enjin Platform identity. When a transaction is submitted on the Enjin Platform, the Wallet Daemon receives that transaction signs it, and sends it back to the Enjin Platform.

### Query

A query is a special type of object that clients can execute against a server to return information like a list of addresses or tokens available in the Enjin Platform.

### Mutation

A mutation is similar in structure and purpose to the queries. Whereas the query type defines entry points for read operations, the Mutation type defines entry points for write operations like creating, minting, and setting metadata to a token in the Enjin Platform.

### GraphiQL

GraphQL is an open-source data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data.

### GraphiQL Playground

GraphQL Playground is a graphical, interactive, in-browser GraphQL IDE, created by Prisma and based on GraphiQL. It's used by the Enjin Platform as a way to improve the experience of developers while interacting with the Enjin Platform before heading towards a software-based GraphQL IDE.

### Platform Schema

The Enjin Platform Schema is a set of parameters and variables that can be processed by the GraphQL language for devs to integrate the Enjin Platform into their projects with ease.

### Wallet App

The Enjin Wallet is a secure, feature-packed, and convenient cryptocurrency and blockchain asset wallet built for traders, developers, and gamers.

### Enjin

Is a suite of integrated products that make non-fungible tokens (NFTs) easy for individuals, businesses, and developers.

### Jumpnet

**JumpNet** is a private (PoA) method of consensus blockchain that allows instant and gasless transactions on-chain.

### ENJ

Enjin Coin (ENJ) is an Ethereum-based cryptocurrency used to back the value of next-gen fungibles and NFTs.

**JENJ**

JENJ is the Jumpnet version of ENJ tokens.

**Efinity**

Efinity is a scalable, decentralized, and cross-chain token infrastructure built to enable users, developers, and enterprises to harness NFTs with zero friction. Built on Polkadot, *Efinity* will serve as the infrastructure for the decentralized, cross-chain Metaverse.

**EFI**

EFI is based on the Paratoken Standard, a new standard for interoperable, cross-chain tokens—across the entire Polkadot ecosystem, and even external blockchains. EFI is the core utility token on Efinity—it's used to pay for transferring, bridging, and trading tokens. Everyone who participates in the Efinity network—from collators who help maintain it, to users who trade or bid on NFTs—is rewarded with EFI.

**Goerli**

The Goerli Blockchain is a PoA (Proof-of-Authority) public testing blockchain, created and maintained by a consortium of Ethereum Developers to assist the Ethereum development community.

**Metadata**

The concept of Metadata applied in blockchain for NFTs is the content and description of the content stored on an NFT in the blockchain, it can be used to refer to a file that will give a name, description, image and attributes to an FT or NFT.

**IPFS**

The InterPlanetary File System is a protocol and peer-to-peer network for storing and sharing data in a distributed file system. IPFS uses content-addressing to uniquely identify each file in a global namespace connecting all computing devices.

**Platform**

The Term Platform refers to the Enjin Platform, which is a service that allows you to create and manage blockchain games without the complexity of building and maintaining the infrastructure typically associated with developing and launching a blockchain game.

**Token ID**

Token ID is a numerical identifier generated by the Enjin Platform to differentiate tokens and make sure tokens are unique and traceable within the Enjin Ecosystem.

### Token Index

A token Index is also a numerical identifier exclusive to NFTs, which is generated by the Enjin Platform to differentiate NFTs that were created under the same Token ID.

### Minting

Minting is the process of creating blockchain assets infused with Enjin Coin (ENJ).

### Melting

Melting is the process of destroying a blockchain asset to retrieve the ENJ from within. The percentage of ENJ a user can retrieve by melting an item can be between 50% and 100%. This value is determined by the developer, who receives the remainder of the melted value (the melt fee).

# About the Enjin Platform

## What is the Enjin Platform?

The complete tech stack for NFTs.

### About the Enjin Platform

The Enjin Platform is a blockchain PaaS (Platform as a Service) that allows you to create and manage blockchain games without the complexity of building and maintaining the infrastructure typically associated with developing and launching a blockchain game.

With the Enjin Platform, game developers are able to integrate blockchain gaming into their projects with ease, by building exactly what you need with a flexible API and using our SDKs and comprehensive documentation along the way.

The Enjin Platform API allows you to:

**Distribute your tokens -** Send tokens to up to 150 users per transaction.

**Trigger Events** - Trigger your NFTs to send based on game, app, or website events.

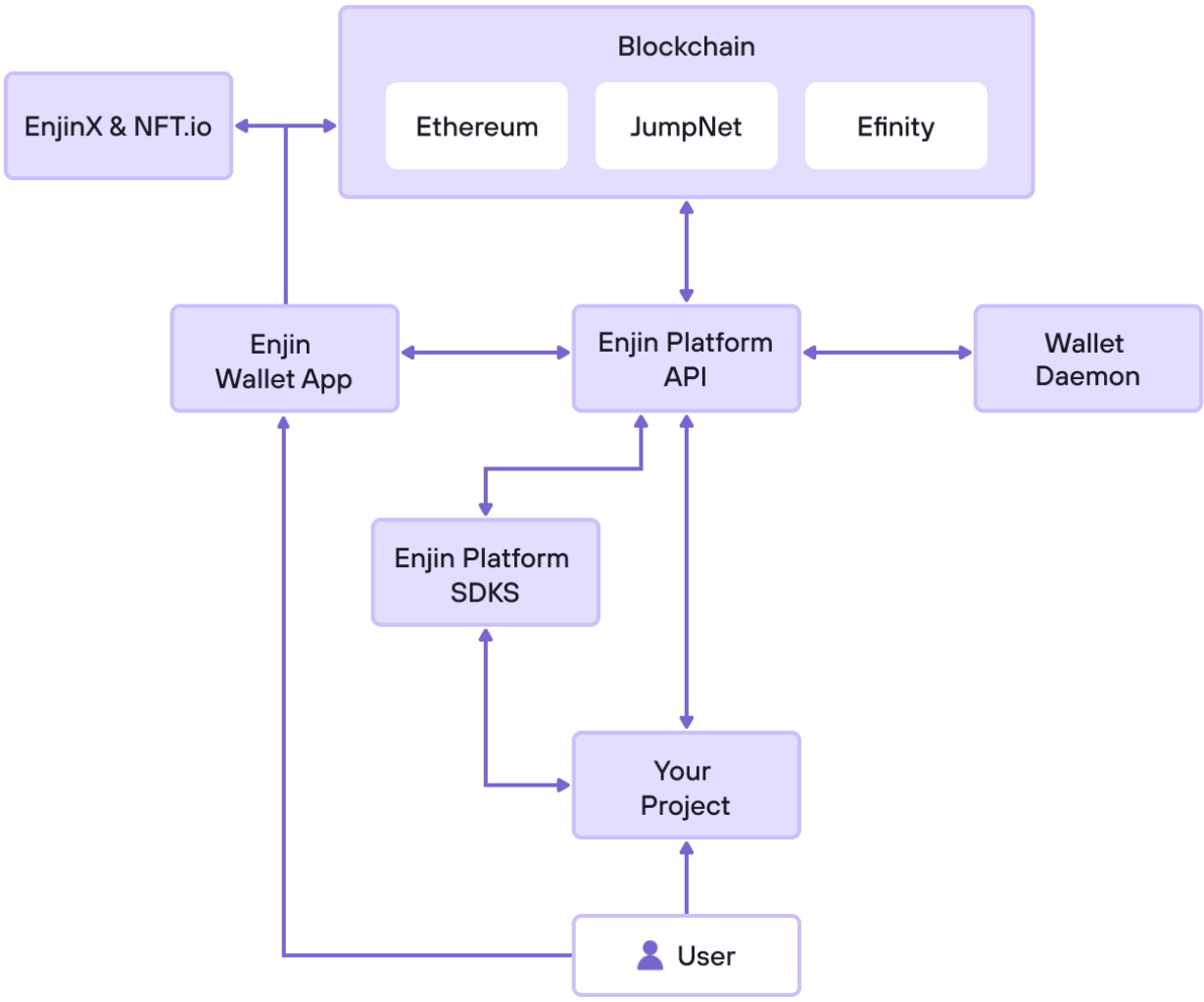
**Enable in-app Trading** - Allow users to perform NFT trades directly in your game or application.

**Wallet Linking** - With the linking feature, users are able to link their Enjin Wallet directly into your game or project with ease.

**Automate Signing** - Use a simple app to automate the transaction signing process

**Get Data** - With the Enjin API, you are able to get robust on-chain data about your users and their wallets.

### The Enjin Platform Structure




The Platform—visual minting interface, API, SDKs, and other development tools—are completely free to use. In order to mint NFTs on Ethereum, you'll need both its native currency (ether), in order to pay network fees, and some Enjin Coin, as NFT minting material.

Minting on JumpNet only requires a nominal amount of Enjin Coin as all transactions on the Jumpnet blockchain do not require any amount of ETH to be processed based on the premise of a PoA blockchain.

## What is Enjin Wallet, EnjinX & NFT.io?

Learn more about the Enjin Wallet, EnjinX & NFT.io

#### Enjin Wallet

The Enjin Wallet is a secure, feature-packed, and convenient cryptocurrency and blockchain asset wallet built for traders, developers, and gamers.

While interacting with the Enjin Platform, the Enjin Wallet can be used by developers as the go-to wallet while creating their games, NFT projects, or anything else that is provided by the Enjin team. For users, the Enjin Wallet can be used to store, trade, and transfer your NFTs without any difficulties. It can also be used to store a wide range of cryptocurrencies.

**The Enjin Wallet is available for Android and iOS:**

- **Download from Google Play:** https://enj.li/android-wallet
- **Download from the App Store:** https://enj.li/iOS-wallet

**What is EnjinX?**

EnjinX is an ad-free, user-friendly Ethereum & Jumpnet Blockchain explorer and marketplace.

With EnjinX, you can explore Ethereum, ERC-20, and ERC-1155 blockchain tokens and blockchain data.

While Interacting with the Enjin Platform, EnjinX can be used by game developers to fetch information regarding transactions performed by your project as well as listing assets for sale so users can purchase them directly from the EnjinX Marketplace.

For Users, EnjinX can be used to look up transactions performed on the ETH blockchain as well as transactions performed using Jumpnet. Users can also purchase and sell their assets directly through the EnjinX Marketplace.

ETH Mainnet EnjinX - https://enjinx.io

Jumpnet EnjinX - https://jumpnet.enjinx.io

> ⓘ For further information regarding EnjinX, please refer to our help center articles here.

**What is NFT.io?**

NFT.io is the global NFT MarketPlace created by the Enjin team on the Ethereum Blockchain, NFT.io is also powered by Efinity, our scaling solution. It supports ERC-1155 digital assets Fungible (FTs) and Non-Fungible Tokens (NFTs) and ERC-721 assets.

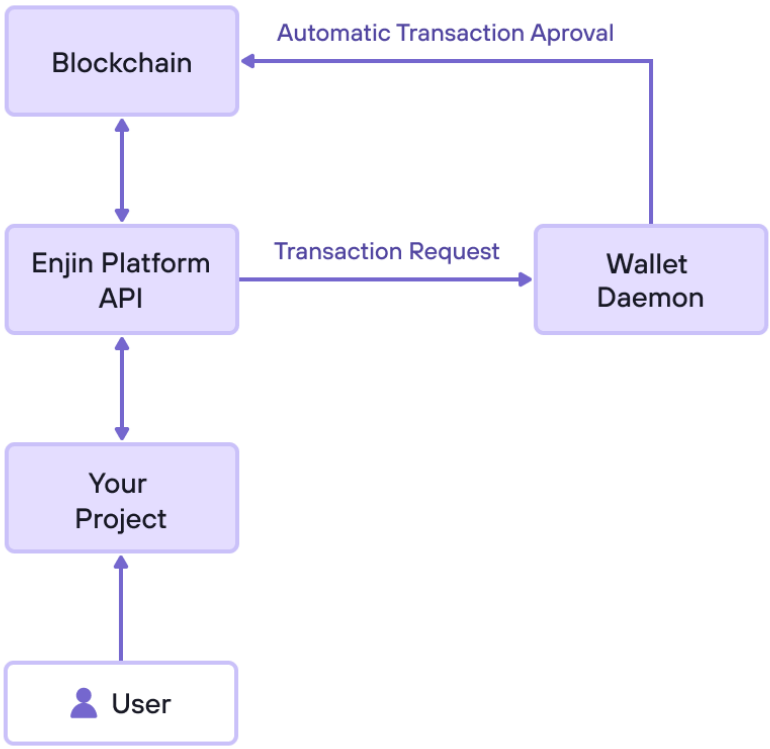
> ⓘ For further information regarding NFT.io, please refer to the NFT.io website here.

## What is the Enjin Wallet Daemon?

The wallet daemon allows you to automate the transaction signing process, so all of your blockchain transactions are actioned instantly, creating a persistent bridge between your game and the blockchain, and ensuring your players can enjoy a seamless and fluid gaming experience.

In the context of the Enjin Platform, The Enjin Wallet Daemon is a utility tool that manages a blockchain

wallet address linked to an Enjin Platform user. When a transaction is submitted on the Enjin Platform, the wallet daemon receives that transaction, signs it, and sends it back to the Trusted Cloud.




From the Diagram Above, devs can understand a bit more about the interaction of the Enjin Wallet Daemon through the Enjin Platform, where it interacts with the Platform API bilaterally by signing transactions automatically and broadcasting them to the blockchain.

## What is Enjin Beam?

> ⓘ **Notice:** Enjin Beam is currently in early access. To request access, please contact support.

Enjin Beam is a QR code-powered NFT distribution service created by the Enjin team to create, expand and engage with your audience.

Enjin Beam allows you to:

1. Easily create and generate QR codes in a few simple steps.
2. Mass-distribute NFTs in the form of a QR code - allowing only for a simple scan.
3. Build up your presence and create a user-friendly experience to reach anyone, from all different backgrounds (even non-crypto/blockchain gurus).

## Types of Beams

### Single-use QRs

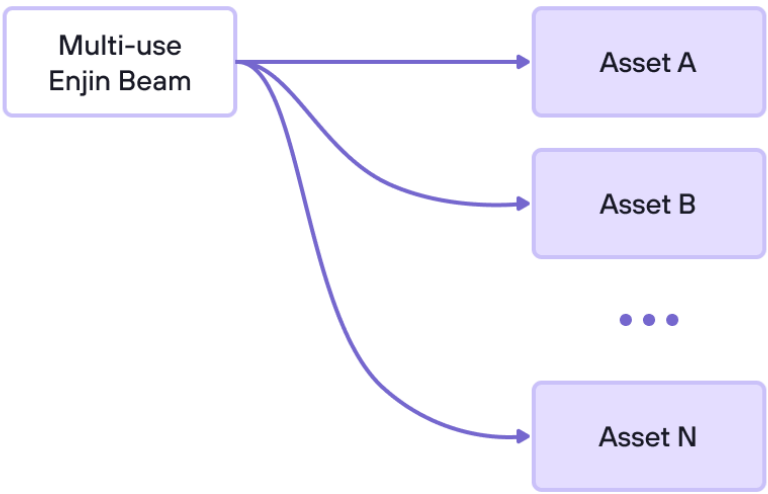
Single-use QRs can be used for one-time claims of an individual asset.

A unique QR code will be generated for each individual asset you want to distribute. Once a single-use QR code is scanned, it is considered depleted and cannot be scanned again.


### Multi-use QRs

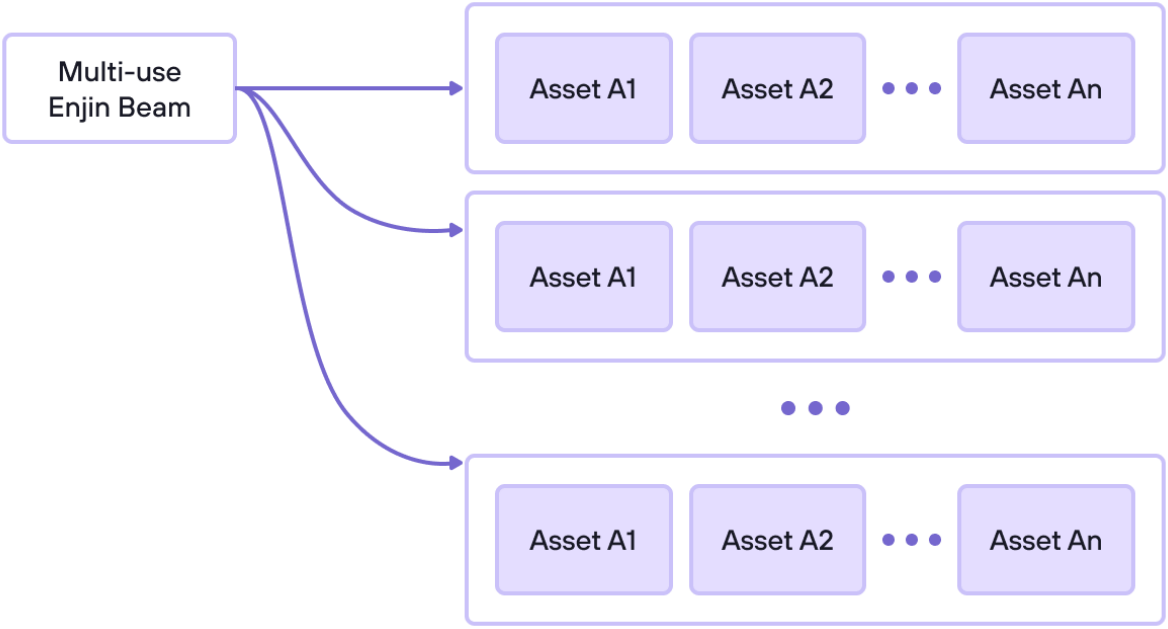
Multi-use QRs can be used to distribute multiple assets using the same QR code.

This allows you to create a single QR code to be displayed on a channel where multiple users are expected to scan the same QR code.




Multi-use Enjin beam with different asset types of singular quantity. All users receive a different asset.


Multi-use Enjin beam with multiple quantities of singular asset type. All users receive the same blockchain asset.

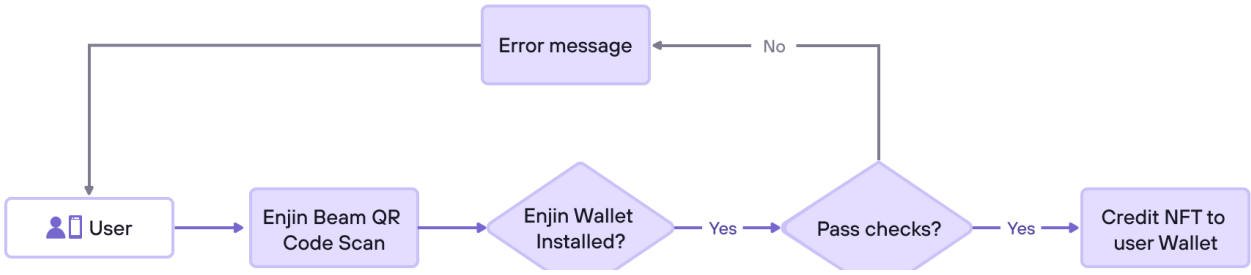


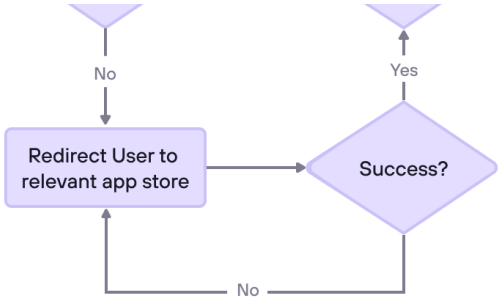

Multi-use Enjin beam with multiple quantity of multiple asset types.

The probability of a user receiving an asset depends on the number of assets held in the Beam; the fewer of an asset there is, the less likely a user is to receive it, and vice-versa.

### User Journey

The user does not need to have the Enjin Wallet pre-installed to proceed with a Beam claim.






## Beam Parameters

| Parameter | Description |
| --- | --- |
| IP restriction (Boolean) | If true, users will be restricted from claiming multiple items from the same IP address. |
| SMS validation (Boolean) | If true, SMS code verification of the Enjin Wallet w be required prior to claiming assets. |
| Password verification (Boolean) | If true, users will be required to enter (or set up) their Enjin Wallet password to claim assets. |
| Randomized sending order (Boolean) | If true, items in the Beam will be distributed in a randomized sequence. If disabled, items will be distributed in sequential order as specified. |
| Claim limit per device (Integer) | Specify the maximum number of claims possible through an individual Beam for any given device. |
| Beam Tags (Optional) | You are able to assign tags to individual Beams to easily categorize and analyze claim data. |
| Beam Metadata (Optional) | You are able to assign custom metadata to your Beams ( `key` and corresponding `value` parameters) |
| Reveal Codes | Blockchain assets containing reveal codes can be minted and sent via Enjin Beam. When unveiled, these hidden messages can deliver exclusive benefits such as discounts, event tickets, early access, gift cards, and more. |

# What is GraphQL and how does it interact with the Enjin Platform API?

Learn more about GraphQL and how the Enjin Platfom uses it through our API.

GraphQL is a modern query language for APIs that allows you to define the data structure of your queries and ask for exactly what you want and nothing more.

GraphQL queries access not just the properties of one resource but also smoothly follow references between them. While typical REST APIs require loading from multiple URLs, GraphQL APIs get all the data your app needs in a single request.

### Operations

**Query:** READ operations are performed by GraphQL queries, these do not change data.

**Mutation:** You will use mutations to perform all other operations to modify data.

### Object Types

Object types are sets of fields that are used to define the set of data you can query from the API.

```
query {

}

mutation {

}
```

### Fields

Fields are used to ask for specific object properties.

Each object has fields that can be queried by name in order to query for the properties you need.

```
query {
 EnjinToken {
    id
 }
}
```

### Arguments

You can determine the return value of a query by passing arguments to it. This narrows down the results and allows you to only get what you're after.

In the following example, the object is `GetAsset`, the requested field is `id`, the arguments are `id`,

`name`, `createdAt`, `updatedAt`, `wallet` and `ethAddress`.

```
query {
  GetAsset(id: "708000000000088e") {
    id
    name
    createdAt
    updatedAt
    wallet {
      ethAddress
    }
  }
}
```

> ⓘ For further reference on how GraphQL works, you can also refer to their documentation here

#### GraphiQL: GraphQL's Visual Interface

Probably the most user-friendly feature of GraphQL is its visual interface, an in-browser tool for writing, validating, and testing GraphQL queries.

Before you query the API, it's recommended to run your queries through the visual interface to make sure they are correct and the data being returned is the data you expect.

**You can use the following GraphiQL web interfaces to interact with the Enjin API:**

- **Ethereum (Mainnet)** API (GraphiQL): **https://cloud.enjin.io/graphql/playground**
- **Goerli (Testnet)** API (GraphiQL): **https://kovan.cloud.enjin.io/graphql/playground**
- **JumpNet** API (GraphiQL): **https://jumpnet.cloud.enjin.io/graphql/playground**

#### GraphiQL Desktop App

You can also download the desktop version of GraphiQL to interact with the Enjin API.

**Download for Windows: https://www.electronjs.org/apps/graphiql**

Here are the endpoints to use within the desktop app:

- **Ethereum (Mainnet)** GraphiQL Endpoint: **https://cloud.enjin.io/graphql**
- **Goerli (Testnet)** GraphiQL Endpoint: **https://goerli.cloud.enjin.io/graphql/playground**
- **JumpNet** API (GraphiQL): **https://jumpnet.cloud.enjin.io/graphql**

#### Postman API Platform Desktop App

You can also download the desktop version of **Postman** to interact with the Open Platform API.

**Download for Windows: https://dl.pstmn.io/download/latest/win64**

# Getting Started with the Enjin Platform API

## Creating your Enjin Platform Account

Learn how to create your Enjin Platform Account.

If you are new to the Enjin Platform, this article will guide you through creating an account.

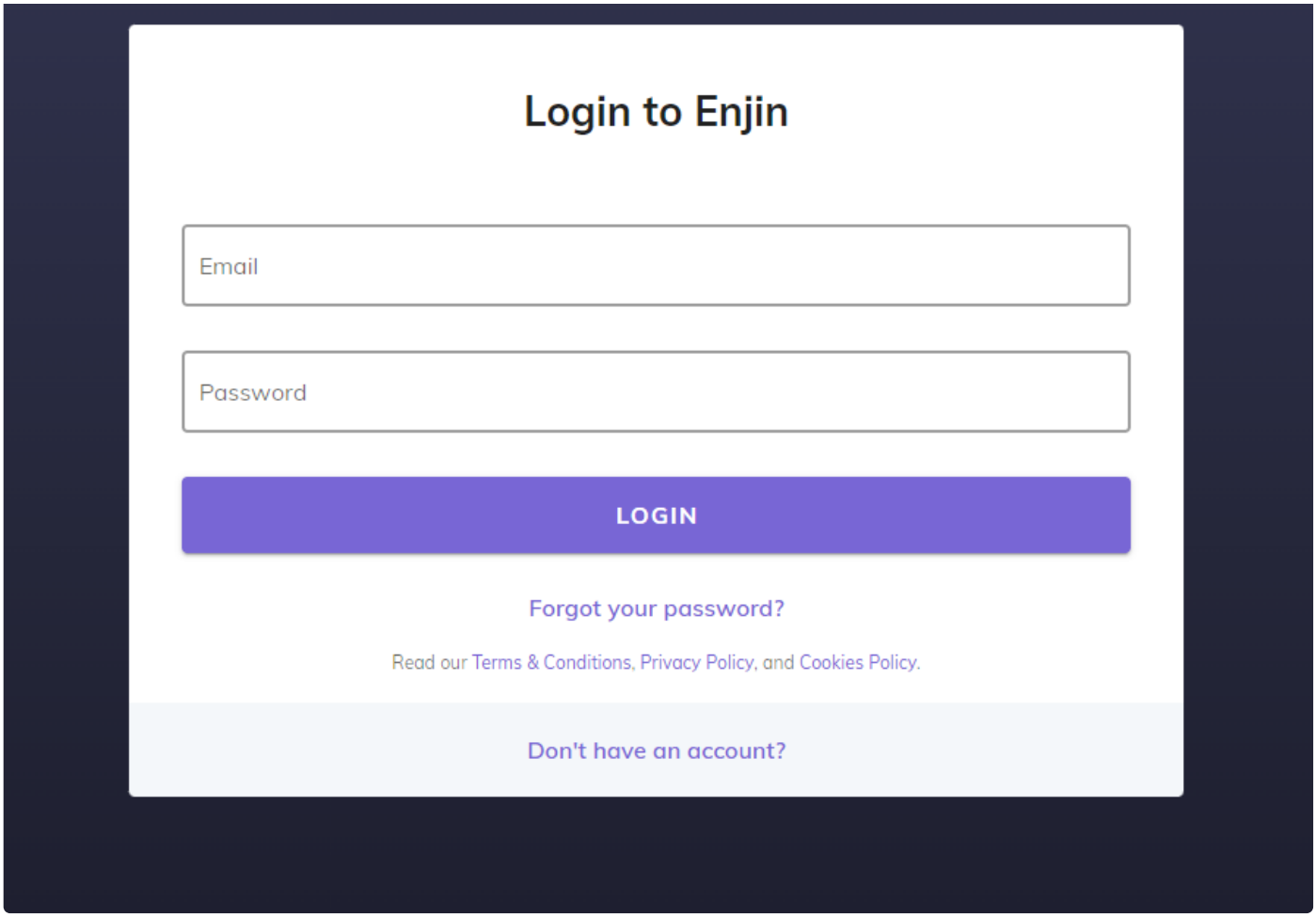
To create an account, register at -

- ETH Mainnet - https://cloud.enjin.io
- Jumpnet - https://jumpnet.cloud.enjin.io
- Goerli Testnet - https://goerli.cloud.enjin.io

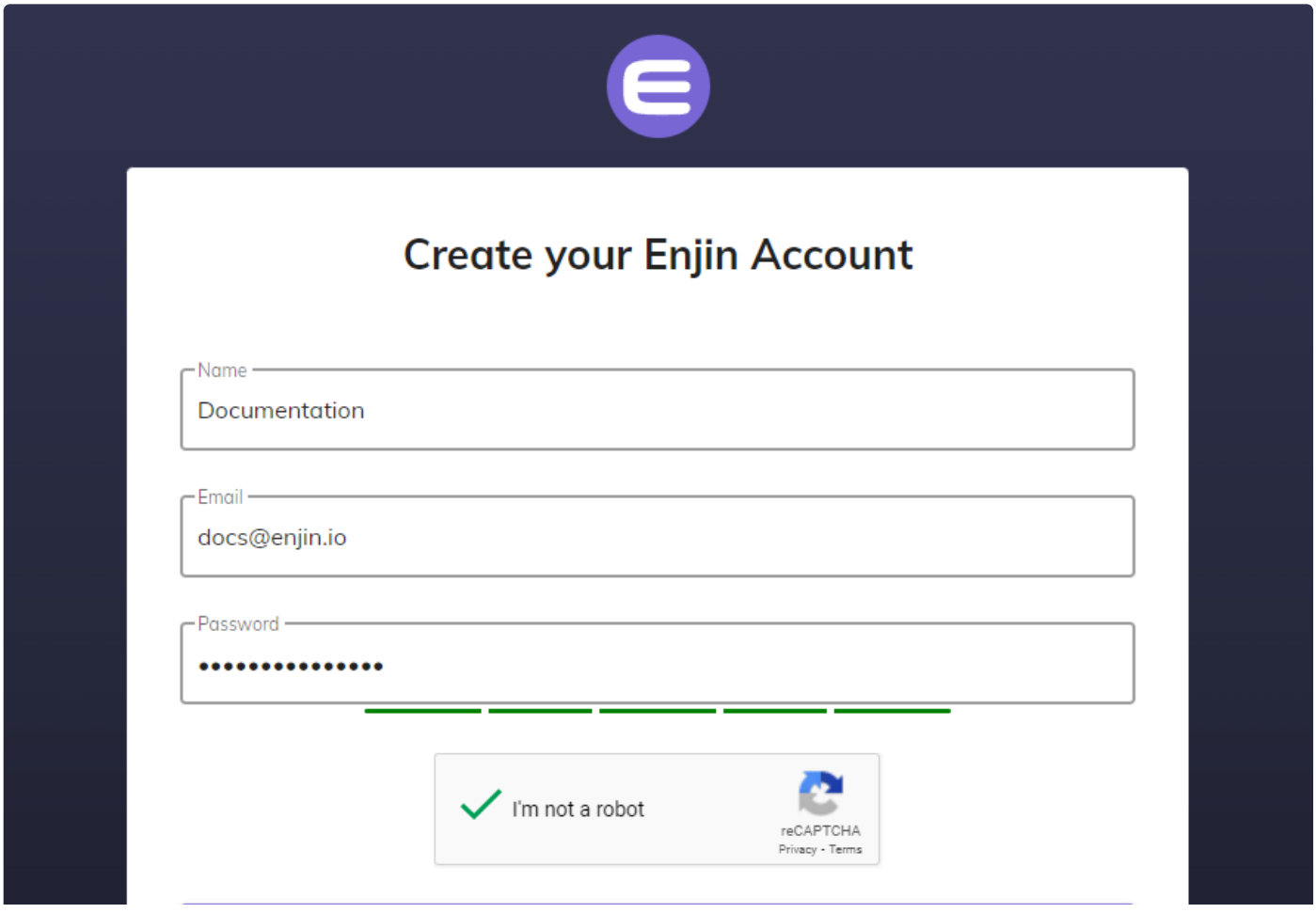
> Keep in mind that you would need an account for each environment, you can still use the same e-mail across any of the environments, but you would still need to register manually through each environment that you are planning on using.

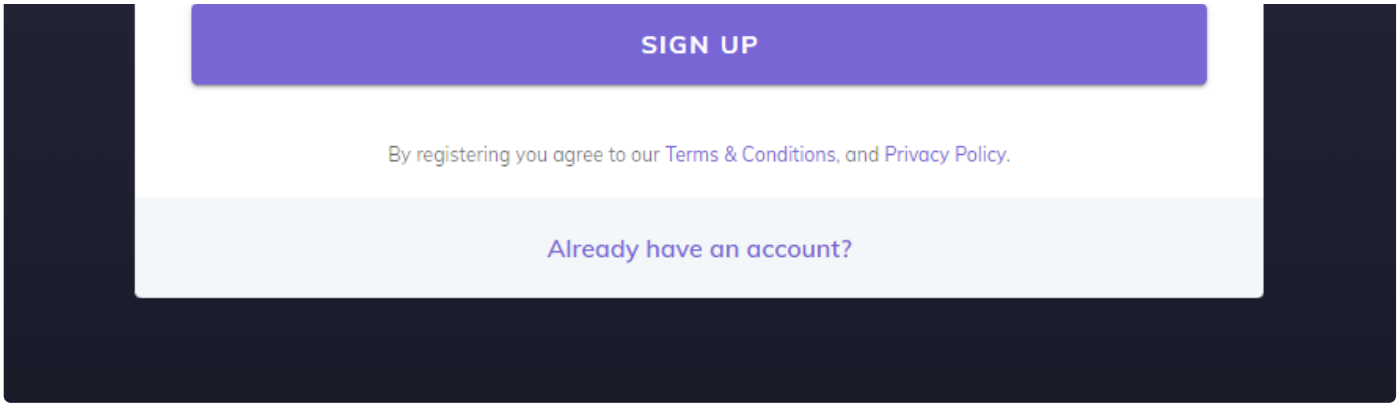
As a suggestion, we would recommend creating your first account using a Testnet Environment, as you can get familiarized with the Enjin API and mint Testnet assets without needing to spend any Mainnet tokens while paying for transactions or backing your assets with real ENJ.

1. Reach out to the environment you are planning on creating your account and click on the "Don't have an account?" button.

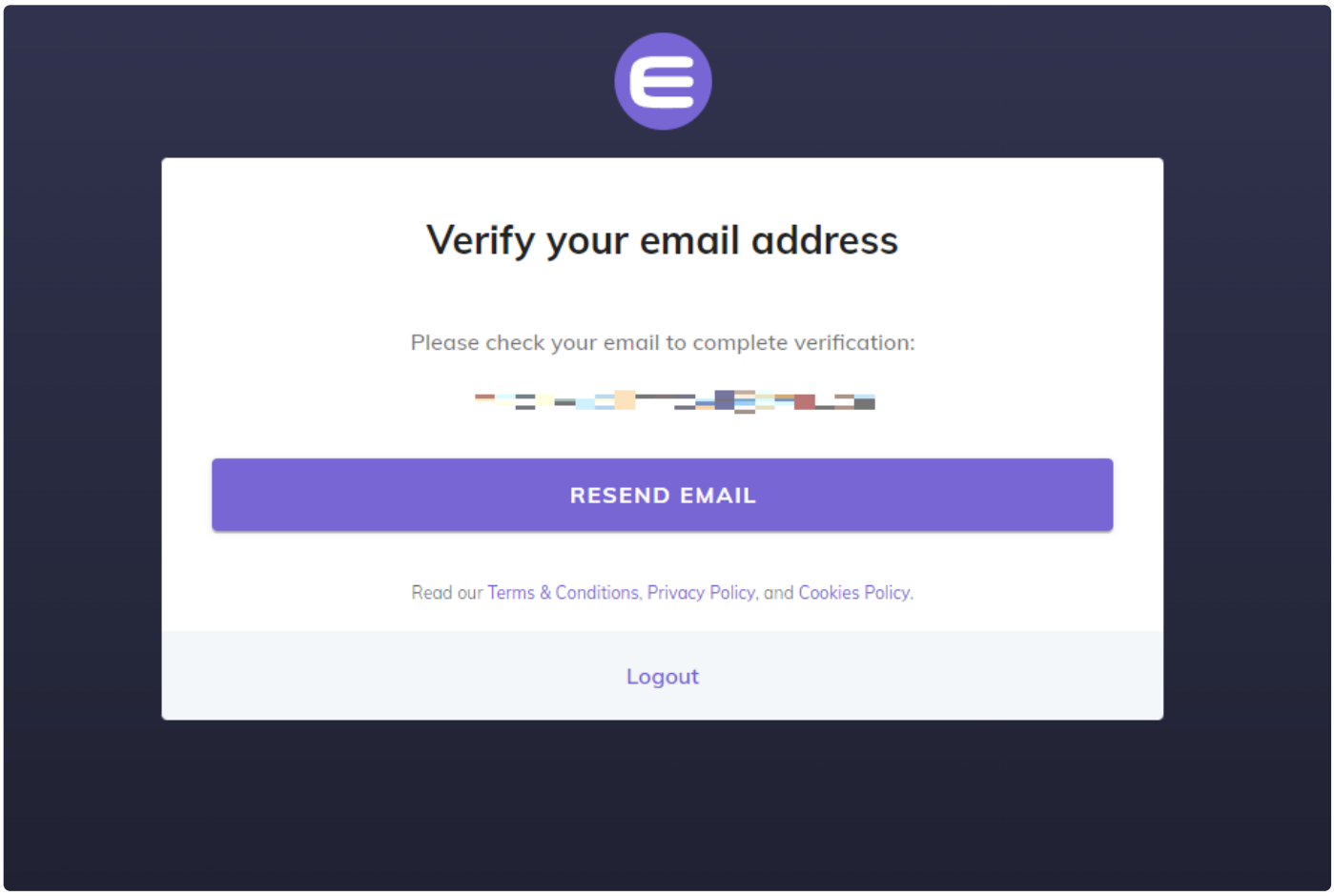



2. Enter details about your account, verify the captcha, and click on the "sign up" button.

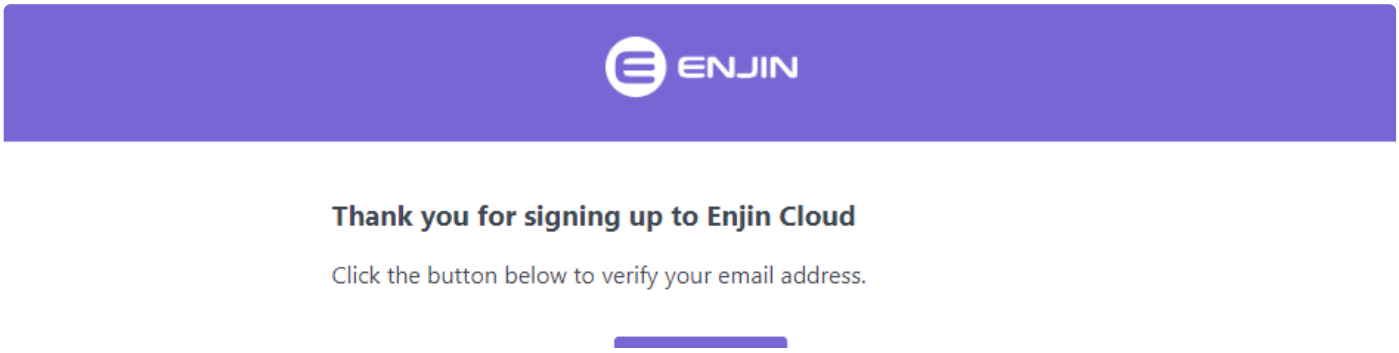


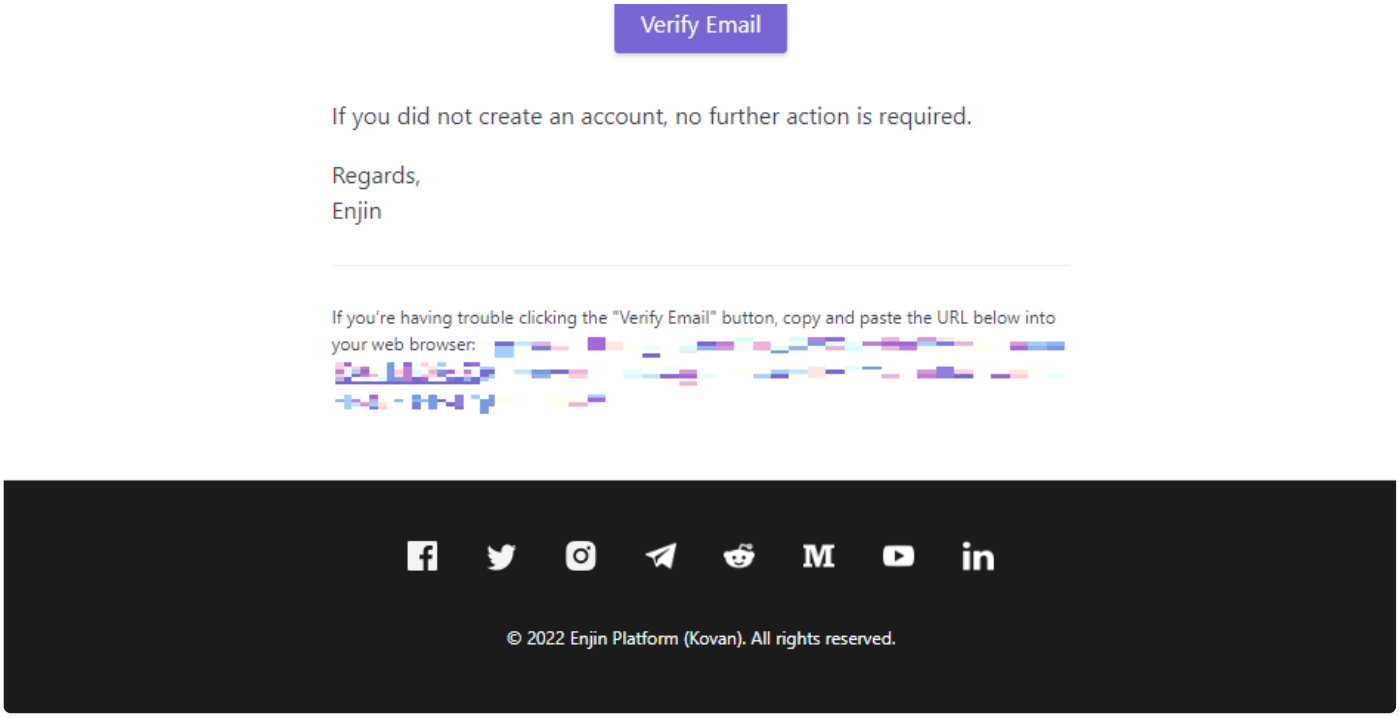



3. After clicking on the "sign up" button, you will need to visit your e-mail account in order to confirm the account creation, make sure to check under your trash and spam inboxes if you don't see the confirmation e-mail in your inbox.




4. The confirmation e-mail should look like this, click on the confirmation e-mail link and you should be redirected to the Enjin Platform page.



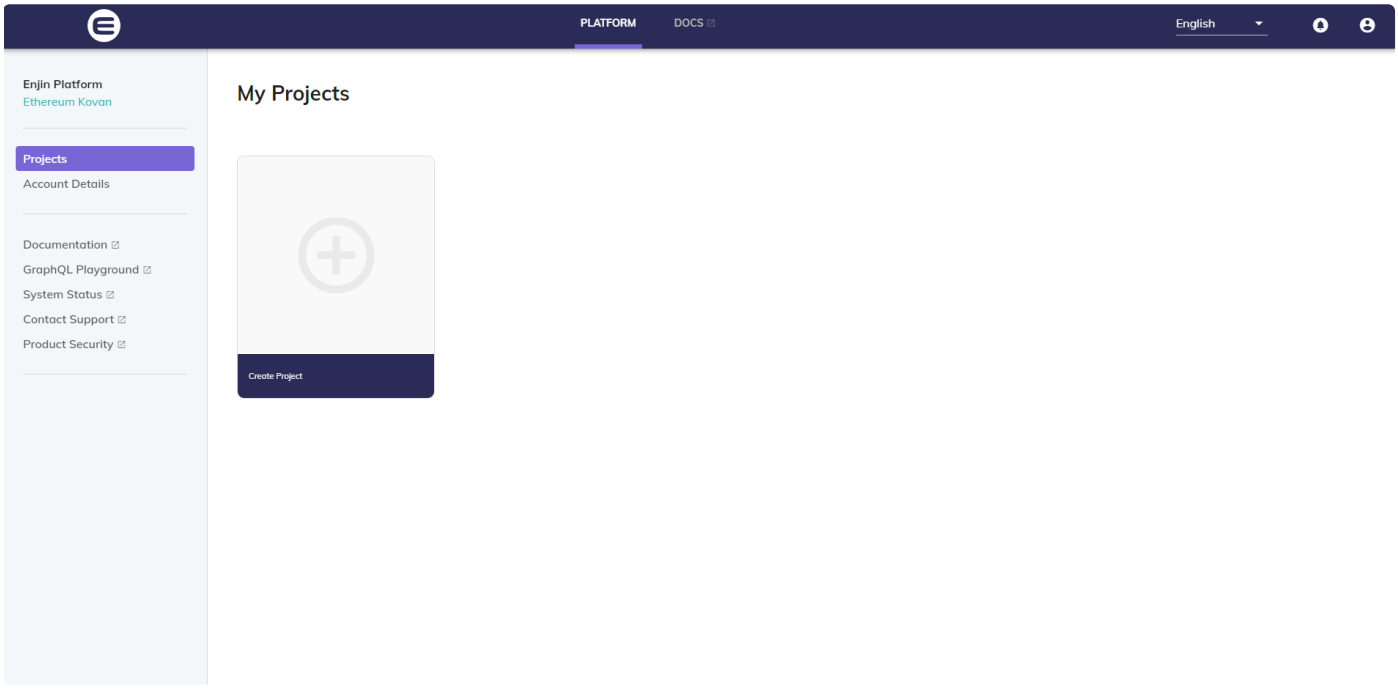



# Creating your Enjin Platform Collection

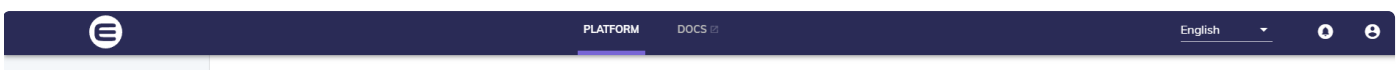
Learn how to create your Enjin Platform collection before starting with your integration.

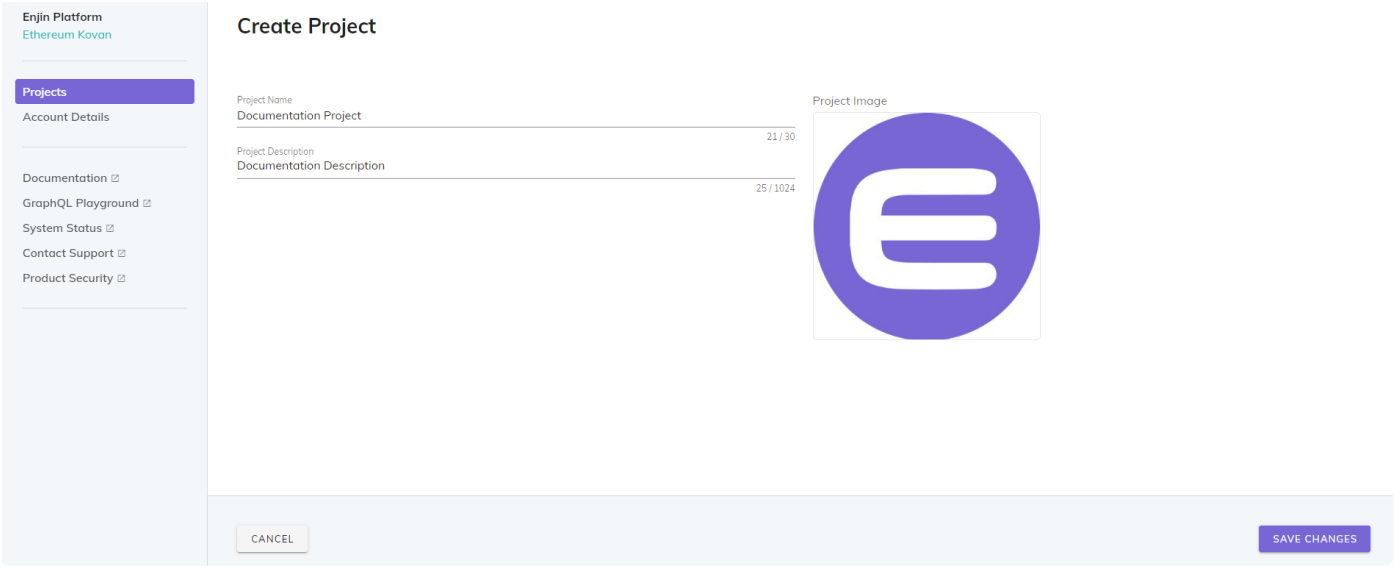
### Creating your Enjin Platform Collection

To create your Enjin Platform Collection, click on the "Create Project" button to begin the process.




Choose a name, description, and image for your project. Then, save your changes.



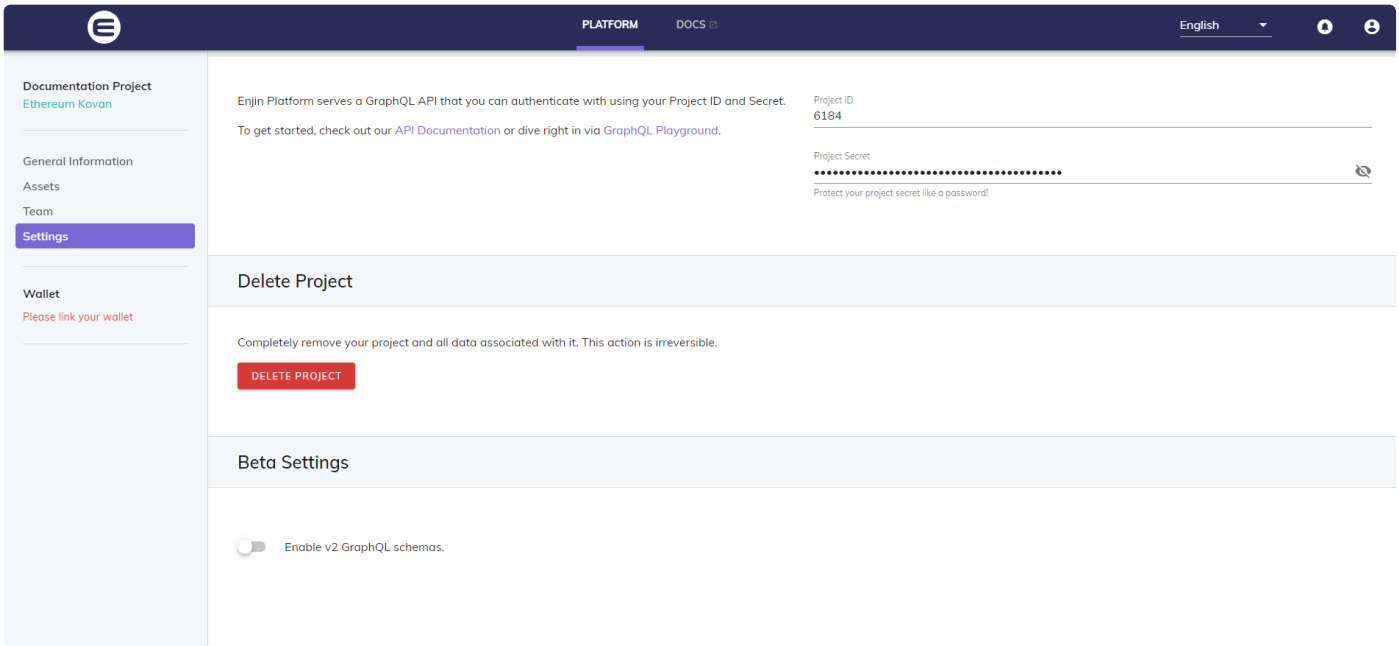



You will receive a message confirming your created project. You will then be redirected to your created Project in the General Information tab.

You can edit your project at any time by selecting it in the General Information tab.

### Deleting a Project

To delete your project, navigate to the project settings you wish to delete and click "Delete Project." You will be prompted to confirm the deletion.




Once confirmed, your project will be deleted and will no longer be listed on the Enjin Platform panel or EnjinX. Any assets that were created under this project will be moved to a general "Other (No Project)" collection.

Note that deleting a project does **not** delete any assets.

Once you delete a project, you will be unable to recover it! Any assets created under a deleted project will be moved to the "Other" collection.

# Enabling V.2 Schemas in your Project

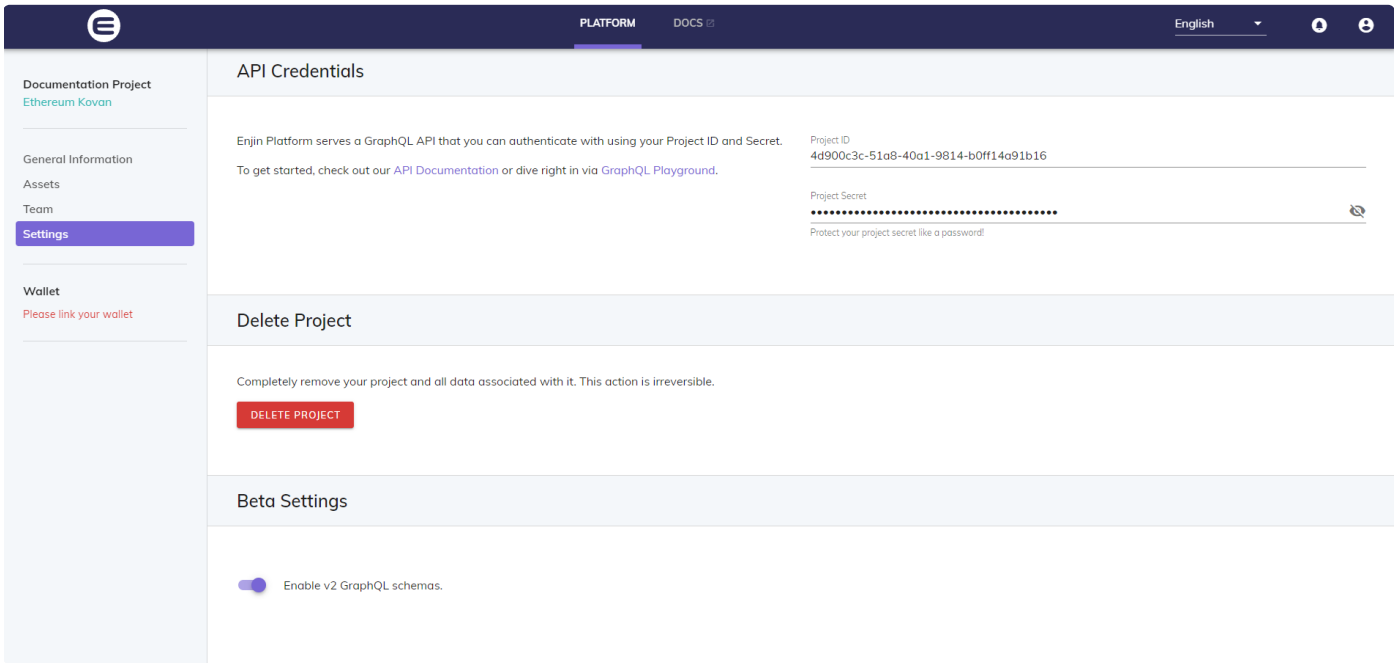
The V.2 schemas are our improved schemas based on the original Enjin API. The v2 schemas consist of the Project and Player.

The **project schema** has all of its queries and mutations scoped to a singular project. This means that a project access token has unfettered control over everything to do with itself, this includes player authentication.

The **player schema**, on the other hand, is restricted to a singular player for a given project. This means that they can only query/mutate data that relates directly to themselves. This means that you cannot, for example, fetch the wallet address of another player or retrieve a list of players for the project.

**The v2 schemas are available on all networks (Mainnet, Goerli, and JumpNet).**

1. Go to your current collection.
2. Go to "Settings" in the left-side panel.
3. Scroll down to 'Beta Settings'.
4. Toggle on to enable v2 schemas for your project.




# Creating Your Player

Learn how to create your player in order to start processing queries and mutations through the Enjin Platform API.

### Create your User/Player

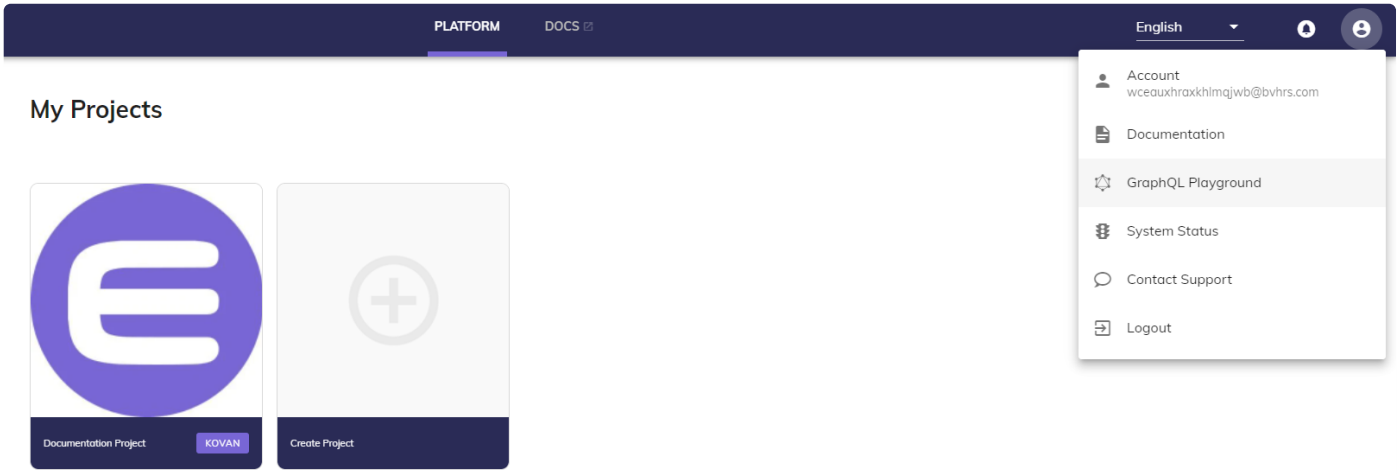
After you have created your Enjin Platform collection, it's time to create your user/player, this is mandatory if you are planning on running processes that need to be approved by your developer wallet.

To do this, reach out for the GraphQL Playground menu at the top right corner of your screen, or you can also access it through these links.

**Ethereum (Mainnet)** API (GraphiQL): **https://cloud.enjin.io/graphql/playground**

**Goerli (Testnet)** API (GraphiQL): **https://kovan.cloud.enjin.io/graphql/playground**

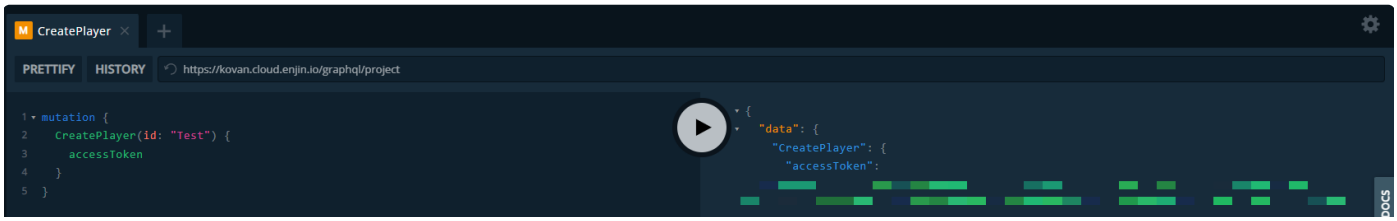
**JumpNet** API (GraphiQL): **https://jumpnet.cloud.enjin.io/graphql/playground**




After accessing the GraphQL Playground, select your project using the V.2 schemas and enter the following mutation, make sure to enter the username of your user/player credential.

```
mutation {
  CreatePlayer(id: "Your Username") {
    accessToken
  }
}
```

Using this query should return a unique **Access Token** for yourself. You will want to keep this Access Token server-side, and somewhere stored securely.



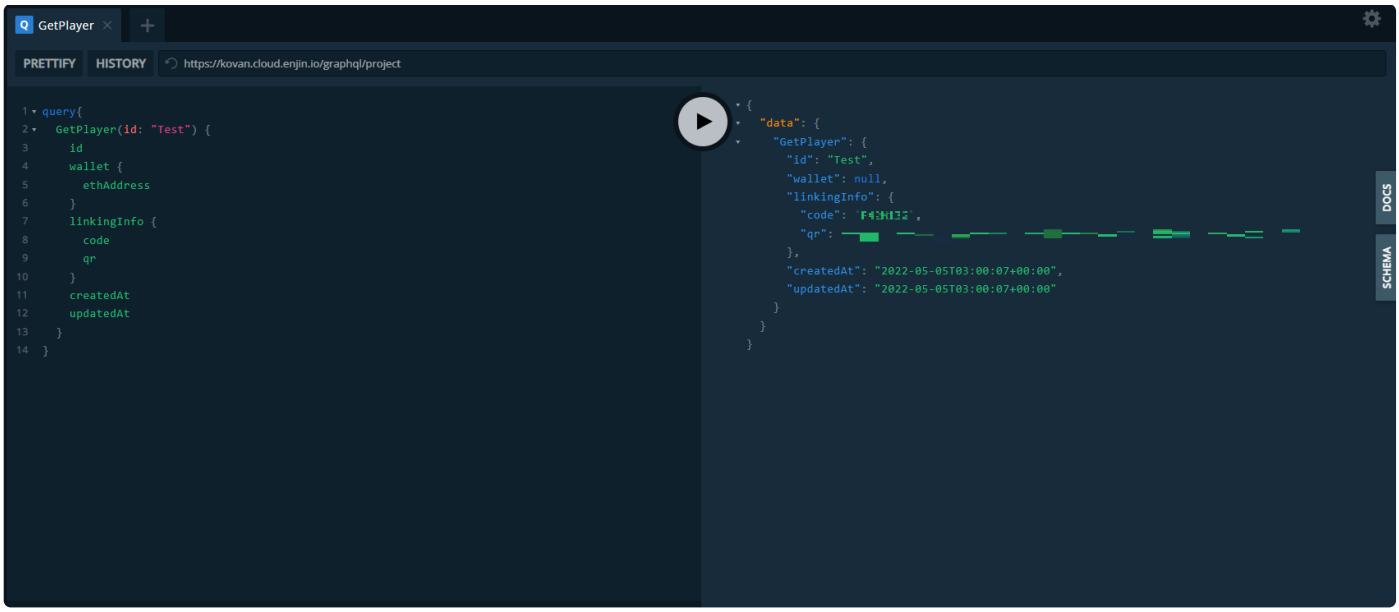
## Linking your Enjin Platform Account to your Enjin Wallet

### Link Your Enjin Wallet

From the account that you've generated from the previous step, you will want to link your project to your developer wallet. You can link your wallet and retrieve linking information by using the following query:

```
query{
  GetPlayer(id: "Your Username") {
    id
    wallet {
      ethAddress
    }
    linkingInfo {
      code
      qr
    }
    createdAt
    updatedAt
  }
}
```

Using the query above should return information such as your username, your wallet address (if already linked), your `linkingInfo` with the option to link with the unique code, or via QR.



- **Code:** You can link your wallet by going to the Enjin Wallet -> Linked Projects -> "+ Link Project" -> Select your Developer Wallet and enter the unique code to link.
- **QR:** You can link your wallet by going to the Enjin Wallet -> Linked Projects -> "+ Link Project" -> Select your Developer Wallet and scan the unique QR code that you've been provided.

Since we are passing requests through the platform itself, it's also required for you to link your project through the platform page, which can be done by following this guide here

Once you have linked your wallet to your project, you will be able to approve and process any request from the Enjin Platform and Enjin API that are programmatically sent from your project.

# Authentication

How to authenticate your requests on the Enjin Platform API.

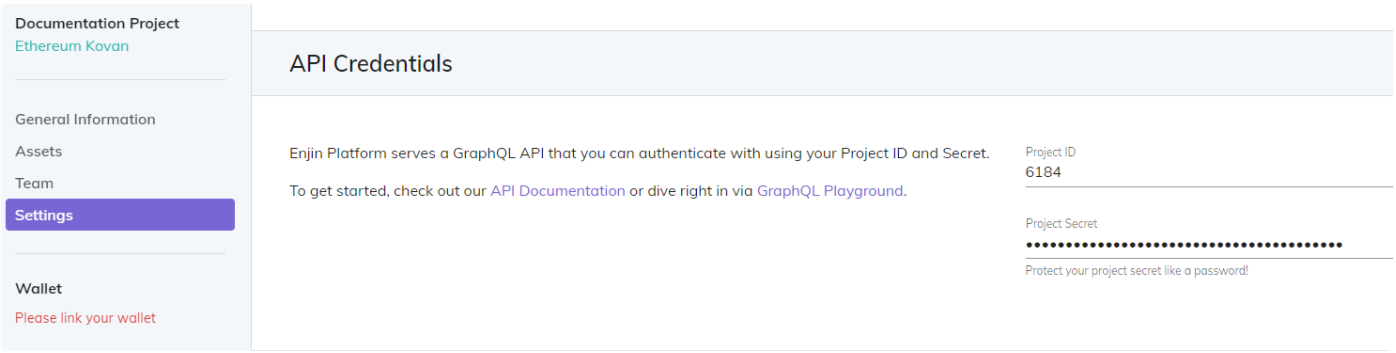
Now that you have already set up your project, we are going to go over the process of authenticating your requests.

This will allow you to programmatically view wallet inventory, mint assets from your wallet, send assets from your wallet, and request assets from users' wallets.

### Step 1: Find your App ID & AppSecret

Firstly, you will need to locate your **AppID** and **AppSecret** on the Enjin platform.

1. Go to cloud.enjin.io (if using Mainnet), goerli.cloud.enjin.io (if using Goerli Testnet), or jumpnet.cloud.enjin.io (if using JumpNet).
2. Select your **Project.**
3. Select **Settings** in the sidebar.




You will find your **AppID** and **AppSecret** in the **Settings** panel.

### Step 2: Generate your Authorization Token

To do this, reach out for the GraphQL Playground menu at the top right corner of your screen, or you can also

access it through these links.

**Ethereum (Mainnet)** API (GraphiQL): **https://cloud.enjin.io/graphql/playground**

**Goerli (Testnet)** API (GraphiQL): **https://goerli.cloud.enjin.io/graphql/playground**

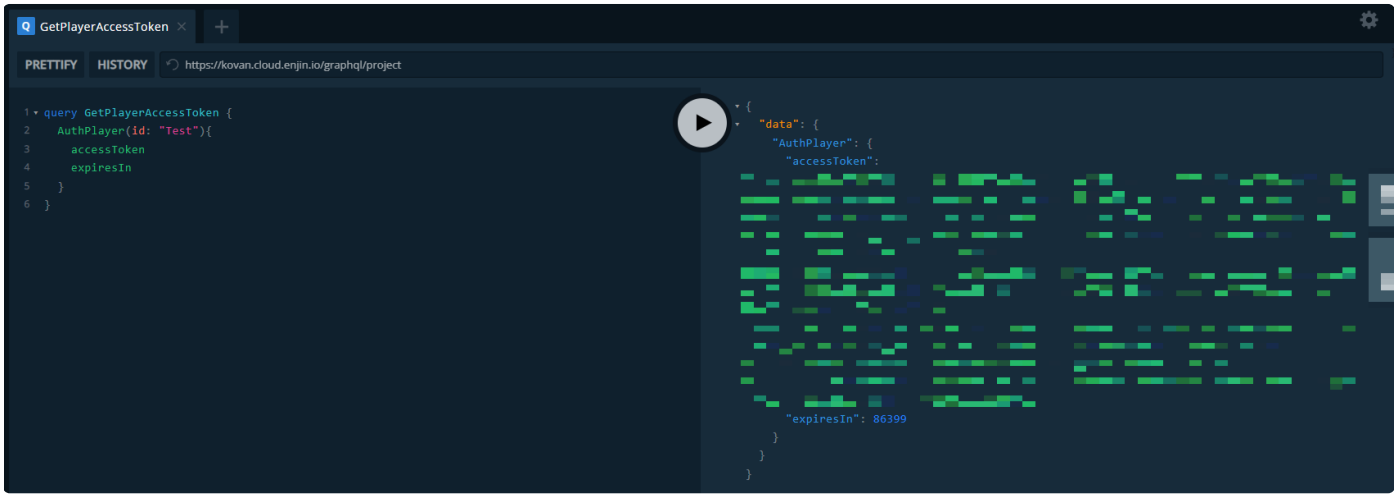
**JumpNet** API (GraphiQL): **https://jumpnet.cloud.enjin.io/graphql/playground**

Generate your unique authorization token by using the following query:

GraphQL

```
query GetPlayerAccessToken {
  AuthPlayer(id: "userName"){
    accessToken
    expiresIn
  }
}
```

Running this query should return your access Token and an expiration timeframe.




> ⚠ **IMPORTANT:** Only store your App Secret server side. Do not store your App Secret inside your executable file or hackers will be able to decompile your game and mint tokens on your behalf.

### Step 3: Use your App Secret

You can now use your AppSecret to run queries from your server by querying the Enjin API. You can do this either via GraphiQL PlayGround on Ethereum Mainnet, Enjin JumpNet, or on Goerli Testnet.

1. **Mainnet:** **cloud.enjin.io/graphql/playground**
2. **JumpNet:** **jumpnet.cloud.enjin.io/graphql/playground**

3. **Kovan: https://goerli.cloud.enjin.io/graphql/playground**

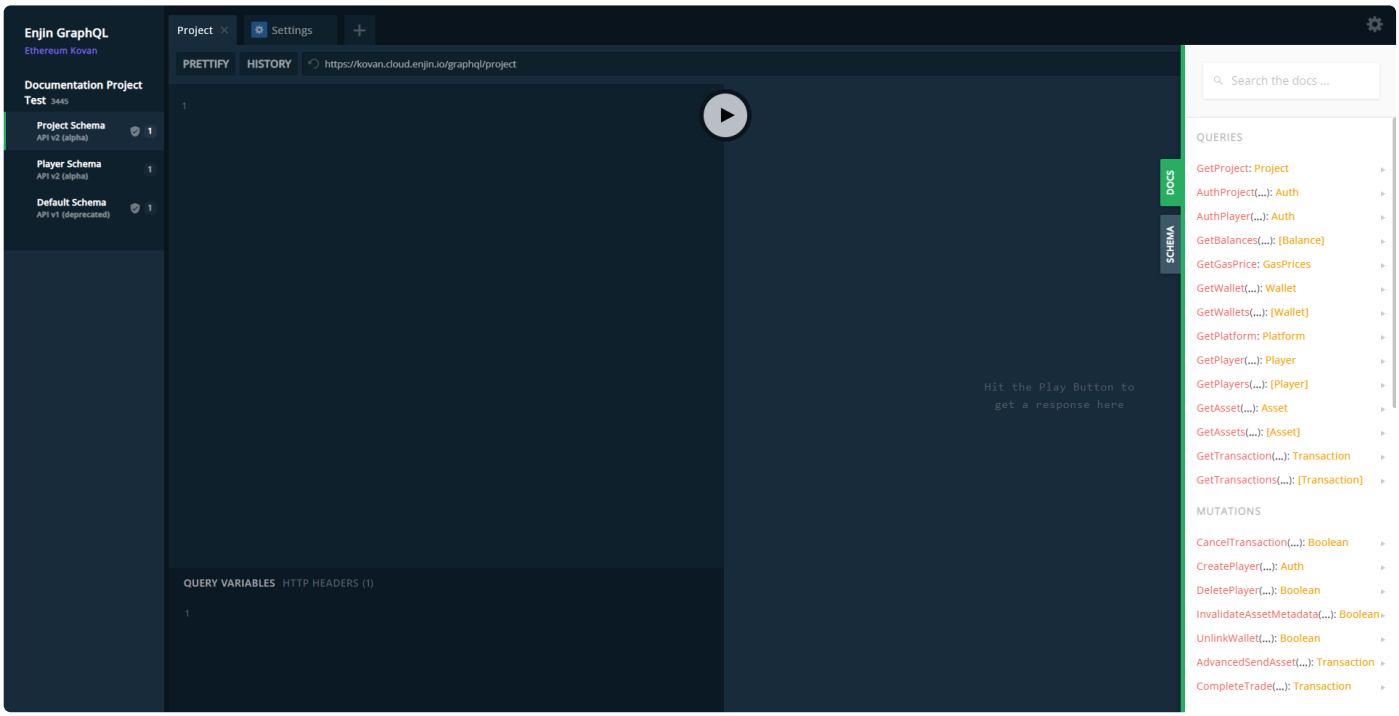
# Project & Player Requests

Learn more about Project and Player Requests

### Project Schema

The project schema has all of its queries and mutations scoped to a singular project. This means that a project access token has unfettered control over everything to do with itself, this includes player authentication.

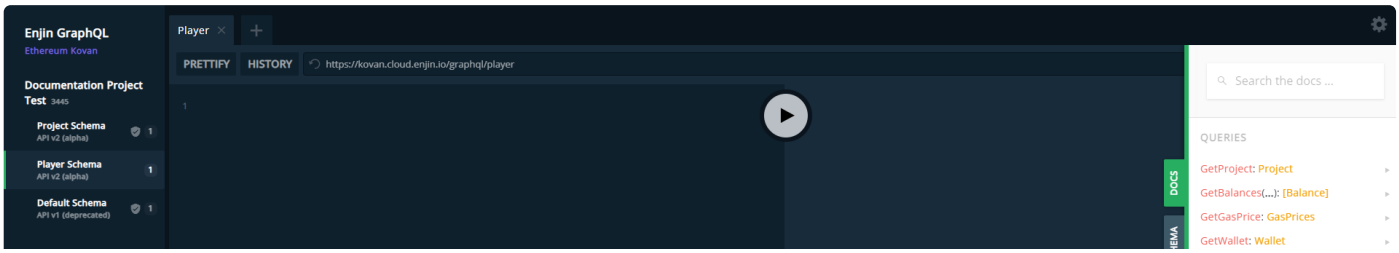
**A project** can do any task related to itself or its players (eg. the project can query any player's wallet). This can be seen as the project administrator.




### Player Schema

The player schema, on the other hand, is restricted to a singular player for a given project. This means that they can only query/mutate data that relates directly to themselves. This means that you cannot, for example, fetch the wallet address of another player or retrieve a list of players for the project.

**A player** can only do tasks related to itself (eg. Bob can only query their own wallet, cannot query Alice's wallet) and cannot create assets.




Project x Player Schemas Table.

The following table shows the differences between Project and Player schema permissions while running Queries and Mutations on GraphQL Playground.

Queries

| Query | Project | Player |
|---|---|---|
| `GetProject` | ✔Able to Run | ✔Able to Run |
| `AuthProject` | ✔Able to Run | Not able to F |
| `AuthPlayer` | ✔Able to Run | Not able to F |
| `GetBalances` | ✔Able to Run | ✔Able to Run |
| `GetGasPrice` | ✔Able to Run | ✔Able to Run |
| `GetWallet` | ✔Able to Run | ✔Able to Run |
| `GetWallets` | ✔Able to Run | Not able to F |
| `GetPlatform` | ✔Able to Run | ✔Able to Run |
| `GetPlayer` | ✔Able to Run | Not able to F |
| `GetAsset` | ✔Able to Run | ✔Able to Run |
| `GetAssets` | ✔Able to Run | ✔Able to Run |
| `GetTransaction` | ✔Able to Run | ✔Able to Run |
| `GetTransactions` | ✔Able to Run | ✔Able to Run |

Mutations

| Mutation | Project | Player |
| --- | --- | --- |
| `CancelTransaction` | ✔Able to Run | ✔Able to Run |
| `CreatePlayer` | ✔Able to Run | Not able to Run |
| `DeletePlayer` | ✔Able to Run | Not able to Run |
| `InvalidateAssetMetadata` | ✔Able to Run | Not able to Run |
| `UnlinkWallet` | ✔Able to Run | ✔Able to Run |
| `AdvancedSendAsset` | ✔Able to Run | ✔Able to Run |
| `CompleteTrade` | ✔Able to Run | Not able to Run |
| `CreateAsset` | ✔Able to Run | Not able to Run |
| `CreateTrade` | ✔Able to Run | Not able to Run |
| `DecreaseMaxMeltFee` | ✔Able to Run | Not able to Run |
| `DecreaseMaxTransferFee` | ✔Able to Run | Not able to Run |
| `MeltAsset` | ✔Able to Run | ✔Able to Run |
| `Message` | ✔Able to Run | ✔Able to Run |
| `MintAsset` | ✔Able to Run | Not able to Run |
| `ReleaseReserve` | ✔Able to Run | Not able to Run |
| `SendEnj` | ✔Able to Run | ✔Able to Run |
| `SendAsset` | ✔Able to Run | ✔Able to Run |
| `SetApprovalForAll` | ✔Able to Run | ✔Able to Run |
| `SetUri` | ✔Able to Run | Not able to Run |
| `SetMeltFee` | ✔Able to Run | Not able to Run |
| `SetTransferFee` | ✔Able to Run | Not able to Run |
| `SetTransferable` | ✔Able to Run | Not able to Run |
| `SetWhitelisted` | ✔Able to Run | Not able to Run |
| `UpdateName` | ✔Able to Run | Not able to Run |
| `ApproveEnj` | ✔Able to Run | ✔Able to Run |
| `ApproveMaxEnj` | ✔Able to Run | ✔Able to Run |

| | | |
|---|---|---|
| `ResetEnjApproval` | ✔Able to Run | ✔Able to Run |
| `BridgeAsset` | ✔Able to Run | ✔Able to Run |
| `BridgeAssets` | ✔Able to Run | ✔Able to Run |
| `BridgeClaimAsset` | ✔Able to Run | ✔Able to Run |

ⓘ Project and Player schemas are part of the V.2 Enjin API schemas, which can be enabled by accessing the Projects settings on your project page.

## Quick Start-up Guide to use Goërli

Learn more about how to set up your wallet to use and connect with to the Goërli testnet environment.

***Last update: 27/06/2022***

When we launched Kovan Testnet for Enjin products in early 2019, we wanted to enable developers to test and utilise Enjin products without using real-world funds. This is where the testnet environment comes into play.

**What is a Testnet Environment?**

A testnet environment is where you, as a blockchain developer, can test and experiment as much as you want using fake assets and money, this environment is solely used for testing your game or your project ahead of going into full development.

ⓘ We would **not** recommend using Goërli for the actual development integration of your project/game.

**Why should you use Goerli?**

We are migrating to Goërli due to Kovan (our previous testing environment) being officially deprecated effectively on the 25th of June. We would recommend using the Goërli network for all testing purposes.

**Key Notes:**

- Effective June 25th, 2022, you will no longer be able to access the Kovan Testnet environment(s);
- You will need to create a Goërli account here - https://goerli.cloud.enjin.io/signup
- Your account on Kovan will no longer be searchable.
- Enjin Platform: https://goerli.cloud.enjin.io/signup

- EnjinX: https://goerli.enjinx.io

**What's Next For You?**

The following guidelines and instructions will help you to get started with the Goërli environment for your testing.

**Pre-Requirements:**

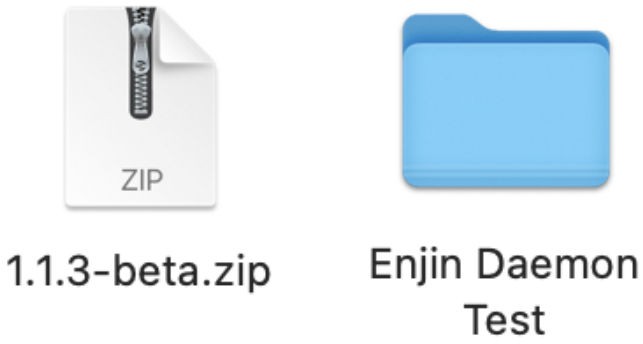
1. Create an account on the Goërli Enjin Platform here.
2. Download the Enjin Wallet Daemon here.
3. Acquire GENJ (Enjin Test Tokens on Goerli) from the Enjin Support team here.
4. Acquire GETH (ETH Test Tokens on Goerli) from the faucet here.

Step 1:

Download the Enjin Wallet Daemon from here.

Once downloaded, extract the contents of the .zip folder into another wallet. Name the folder to something simple (e.g. Enjin Daemon).

**Note:** The steps for this may look a bit different, depending on which operating system you're using (Linux, macOS,  or Windows). We also have the same steps for each operating system here.




Once done, in the unzipped folder, you will need to locate the `example-config.json` file. Duplicate the file and rename it to `config.json` . This will be your primary config file which you'll edit from.

You will have something like this:


In the `config.json` file, locate the `chain:` property.

You will need to change the value to `goerli` - this will allow the wallet to connect successfully to the Goerli platform.

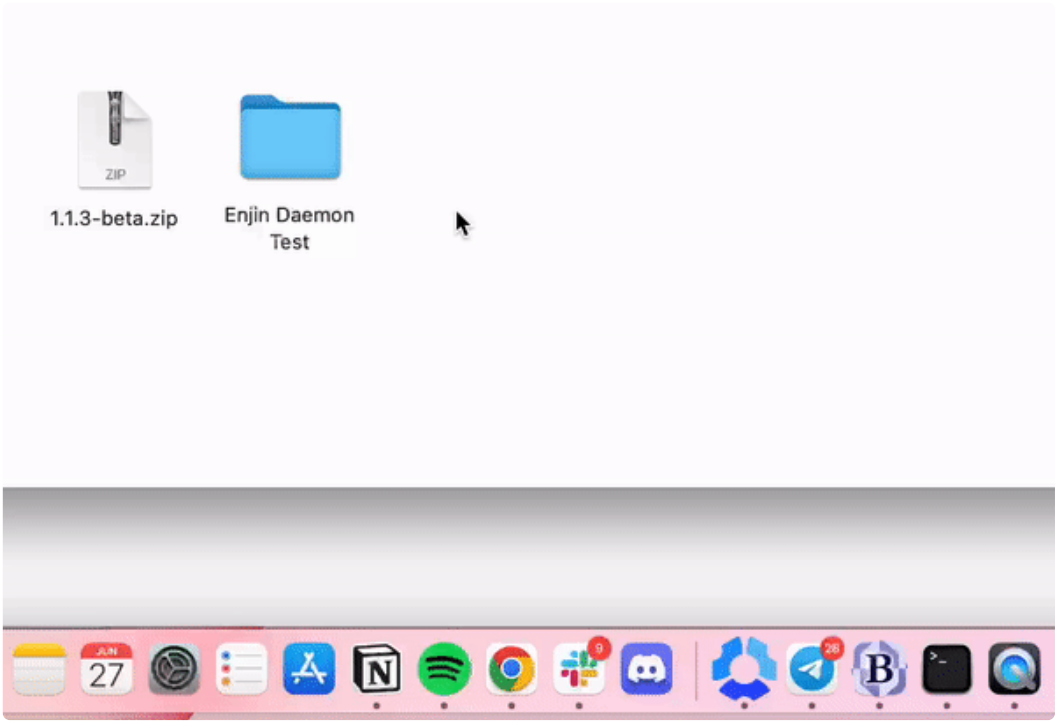
You should have something like this:

```
{
    "salt": "193e9997-5a10-4d9e-a829-69ddcf6cbf70",
    "keyIterations": 1000,
    "chain": "goerli",
    "enjinxEndpoint": "https://daemon.api.enjinx.io/",
    "minGasPrice": 1000000000,
    "maxGasPrice": 21000000000
}
```

Now you're set!

Step 2:

Drag your Enjin Daemon folder to your terminal, like the following example:




This will run the terminal with your daemon.

1. Run the following command to install the daemon:
2. `cd <CODE_FOLDER>\enjin-wallet-daemon-master npm install`
3. Once done, you can create a new wallet for your daemon. Run the following command:
4. `node src/main.js account new`
5. Set a password for your daemon. Make sure to set up a strong password and one that you can remember.

Step 3:

Once you have a new wallet created using the daemon, it's time to link your wallet to your project on Goërli. Create a project on the Goërli platform. If you're not familiar with creating projects, learn more in our guide here.

1. Go to the project's settings and grab the linking code that you're provided with.
2. Run the following command in the terminal to link your project to your wallet:
3. `node src/main.js link <linking code>`
4. Enter your password and your wallet will be successfully linked to your project.

You will see the following successful confirmation:

```
Password:************
    Enjin Wallet Daemon 1.1.3-beta
    account - created - (address: 0xa081093f3Fe18C2607F6eD63225fea6569cA92e3)
    account - nonce 20 - (address: 0xa081093f3Fe18C2607F6eD63225fea6569cA92e3)
    storage - Saving storage to: /Users/estherpartland/enjin-wallet-daemon/stora
ge.json
    wallet - Linking to new app using A5Q42V
    wallet - Getting TP ID...
    wallet - TP ID is 0
    wallet - Connecting to Platform: 0
    storage - Saving storage to: /Users/estherpartland/enjin-wallet-daemon/stora
ge.json
    identity - linking - (appId: 67)
    storage - Saving storage to: /Users/estherpartland/enjin-wallet-daemon/stora
ge.json
estherpartland@Esthers-iMac enjin-daemon %
```

Step 4:

Now that you've successfully linked your wallet to your project, you can start creating, minting, and distributing digital assets.

> **Note:** Any request you initiate, e.g. a create or mint transaction, the daemon will automatically sign and approve the transaction to execute on the network.

**Running into problems?**

If you're running into any problems or have any questions, please reach out to us at https://enjin.io/support

**FAQs:**

1. **What happened to Kovan?**
   - The Kovan Testnet has, unfortunately, shut down and is no longer being used. We have officially migrated over to Goërli testnet network.
2. **Have I lost all my Kovan assets?**
   - Yes, if you have had an account with test assets on Kovan, unfortunately, these are no longer available to acquire.
3. **Will I be able to transfer KENJ to GENJ?**
   - No, you won't be able to transfer KENJ to GENJ.
4. **How do I acquire GENJ and GETH?**
   - You can acquire GENJ by reaching out to Enjin Support here. You will also be able to acquire GETH from the faucet here.
5. **Does the Enjin Wallet mobile application support Goërli?**
   - At this moment in time, Goërli is not supported on the Enjin Wallet v1 app. However, we are looking into providing support for Goërli in the near future for our Enjin Wallet 2.0 app.

## Creating your first Blockchain Token

Learn how to create your first NFT using GraphiQL Playground.

Now that you have already learned how to create your Enjin Platform collection, player, and link your account to your Enjin Wallet, it's time to create your first blockchain token.

**What is a Blockchain Token?**

Tokens, also known as blockchain assets, are used to represent the identity of your items on the blockchain.

**Fungible Tokens (FT):**

Traditional currencies and cryptocurrencies are fungible; they are identical, interchangeable, and divisible. For currencies to work as a standard payment method, fungibility is essential.

Fungible tokens do not have a unique serial number or history; there is nothing to distinguish one from the next. For example, every $5 note is exactly the same and holds the same value. Every half of one fungible token is equal to two-quarters of another.

Fungible tokens are useful for things like currency, reward points, discounts, and promotional materials—any item that doesn't require a unique identity.

**Non-Fungible Tokens (NFT):**

A non-fungible token is a unique asset.

Non-fungible tokens are not divisible and are stored in the Enjin Wallet as separate tokens with individual data. However, non-fungible tokens are not always 100% unique. For example, a set of tokens may share the same name, description, and image, but they can still be non-fungible if they possess unique, distinguishing properties (identity, history, and metadata).

Non-fungible tokens are suitable for things like identification, certificates, rare items, collectibles, gaming characters—any asset that requires its own unique identity.

There are two types of data that can be attached to each token.

- **Blockchain Data** is committed permanently to the blockchain. The defining properties of a token, including its identity, settings, and ENJ-backed value can impact the demand for a token drastically. Therefore, much of this data can never be changed once committed to the blockchain. While some token settings can be updated by replacing old data with new data, the previous token settings will remain on record, viewable in the transaction history on the blockchain.
- **Metadata** is the human-readable information that your users will be able to see in your game or app and any other platform where they can see your token. This data can be updated at any time.

Blockchain Data: Changeable

| Blockchain Data | Changeable |
| --- | --- |
| Metadata URI | The metadata URI allows you to ad a URL that contains a JSON that describes the properties of your iten including images. |
| Transferable | Determines if items are able to be traded, or are bound to their owners (i.e. non-tradable).<br>• **Permanent**: Item is always able to be traded with others. This setting is not changeable once committed to a token.<br>• **Bound**: The item is always bound to the owner of the item.<br>• **Temporary**: The item is current tradable, but the creator can make it non-tradable at a future date. |
| | If using ENJ, multiply the value by 10^18 to include 18 decimals. Whe |

| | |
|---|---|
| `transferFeeSettings` : value | you first set a transfer fee, that settin becomes the maximum fee you can charge. However, you can lower a transfer fee at any time, at which point, you can then raise it back to the amount you initially set. |

**Blockchain Data: Permanent**

| Blockchain Data | Permanent |
|---|---|
| `totalSupply` : | This is how many of the items you want to exist. This limit can be broken or mean different things depending on the supply model you use above. |
| `initialReserve` : | This is how many items you want to pre-pay & Reserve to mint as part of the initial create operation by locking a bit of ENJ on it. Minting items will be deducted from this balance until it is exhausted. You have to pay for at least one item c creation. Having an initial reserve allows you to create your item without having to spend all the ENJ for your total supply upon creation. |
| `transferFeeSettings` :type | Indicate if a fee (in ENJ or crypto items) will be charged to the sender when they send this asset t another address. This fee will be sent to the asset creator.<br>• **None**: No Transfer fees are charged when thi item changes hands.<br>• **Per_Crypto_Item**: This refers to the transfer fee per asset in ENJ, which is cumulative based on the number of assets that are being sent.<br>• **Per_Transfer**: This refers to the transfer fee, per transfer, in ENJ. For example, if an Apple has a 0.1 ENJ fee per transfer and Simon sends 10 apples to user1, Simon would be charged 0.1 ENJ for the transaction that would go to the creator of the apple asset.<br>• **Ratio_Cut**: Note, to use ratio_cut, only fungib assets are allowed. A % cut of the total items i |

| | |
|---|---|
| | subtracted from the total for the creator, with th sender paying the total price.<br><br>• **Ratio_Extra**: Note, to use ratio_extra, only fungible assets are allowed. A tax that is charged ON TOP of everything. |
| `transferFeeSettings`: | The token ID of the token you want to use as the transfer fee. Use 0 if you want your users to pay you in Enjin Coin. |
| `meltValue` | The amount of ENJ you want to use per unit of the item you are creating. The more items of one type you are making, the less ENJ you need **per unit** o item. It's, also important to note that you would need to set a value by multiplying the value by 10^18 to include 18 decimals. The minimum amount of ENJ backing can be calculated using this formula: `0.1 * Math.sqrt(initialReserve)/ initialReserve` |
| `supplyModel`: | This is how the item pool behaves with respect to minting and melting.<br><br>The following are our current supply types:<br><br>• **Fixed**: You can have up to a TOTAL SUPPLY number of items in circulation at one time.<br>• **Settable**: This allows you to edit the total supply at any time.<br>• **Infinite**: You can mint as many items as you want, exceeding TOTAL SUPPLY.<br>• **Collapsing**: Once melted the items cannot be re-minted as this decreases the total supply of the token. |
| `meltFeeRatio` | This is the current percentage of ENJ that the player will receive upon melting the item. The remaining ENJ goes to the creator. |
| `nonFungible` | Whether the item is Non-Fungible or Fungible, a Boolean value. |

### Creating your first Blockchain Token

Now that you are familiarized with the possible parameters when creating your blockchain tokens, it's time for us to learn how to create it!

> If you wish to learn how to create your tokens using our minting panel, our amazing support team has created a series of guides in our help center here.

First, make sure that your wallet is linked to your player wallet and that you have enough KENJ and KETH if you are using a Testnet environment. It's also important to make sure that your wallet has developer mode enabled, if not, this guide here should assist.

First, head over to the GraphQL Playground and select the Project Schema of your collection, this is the mutation that we are going to run to create our first token template, you can set the parameters accordingly to your needs -

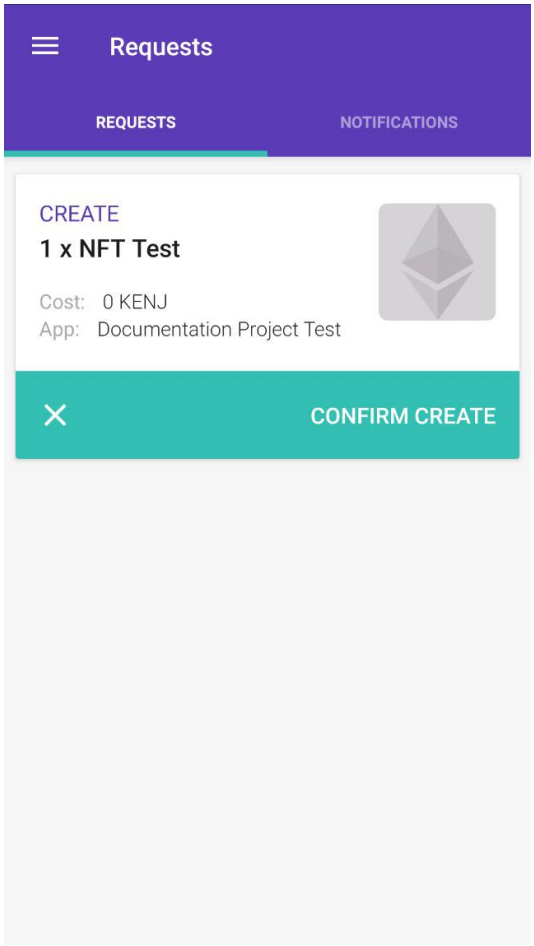
```
mutation {
  CreateAsset(
    name: "NFT Test"
    totalSupply: "1"
    initialReserve: "1"
    supplyModel: COLLAPSING
    meltValue: "1000000000000000000"
    meltFeeRatio: 0
    transferable: PERMANENT
    transferFeeSettings: {type:NONE}
    nonFungible: true
    wallet: "0xCc5e715008aceD612C00c0a4066AB7Da287cF0bB"
  ) {
    value
    id
    title
    state
    project{name}
  }
}
```

After hitting the play button on the playground, if all of your parameters are correct, this is the expected response that you should get from the platform.

```
{
  "data": {
    "CreateAsset": {
      "value": "1",
      "id": 98420,
      "title": "NFT Test",
      "state": "PENDING",
      "project": {
        "name": "Documentation Project Test"
      }
```

```
  } }
}
```

The request is now pending approval in the wallet, to approve it, we must head to the requests tab in the Enjin Wallet and approve the create request as shown here.




Once the request has been approved, we can run this specific query to check on the current status of the request with the id from our previous create asset response -

```
query {
  GetTransaction(id:98420) {
    id
    transactionId
    title
    type

    value
    state
    projectWallet
    asset{id, name}
  }
}
```

After hitting the play button on the playground, if all of your parameters are correct, this is the expected response that you should get from the platform.

```
{
  "data": {
    "GetTransaction": {
      "id": 98420,
      "transactionId": "0x5e0790f6f253e11dae566a23760da161931ab6305b863289e3bbe2725536abfb",
      "title": "NFT Test",
      "type": "CREATE",
      "value": "1",
      "state": "EXECUTED",
      "projectWallet": true,
      "asset": {
        "id": "70800000000029b7",
        "name": "NFT Test"
      }
    }
  }
}
```


The information that we will need to keep from this request is important for us to proceed with our next guide, in this case, make sure to keep the asset id close at heart as that will be necessary for us to set the asset metadata and mint the NFT to your Enjin Wallet.

## Setting Metadata on your first Enjin Token

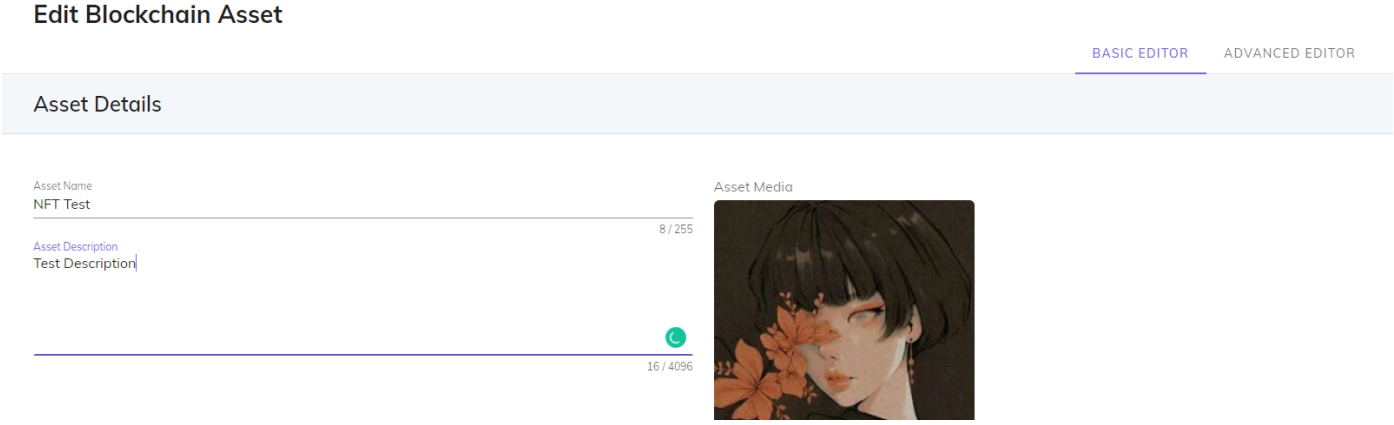
Learn how to set metadata on your first Enjin token.

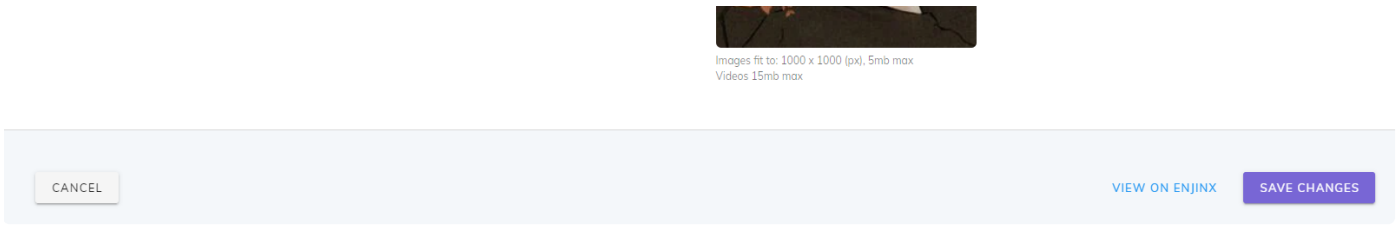
> For the Advanced Metadata guide, please refer to this link here.

### Basic Editor

Now that we have already created our token template with the id `70800000000029b7`, it is time for us to add a valid metadata file on it, which represents the off-chain data attached to the token, such as the name, image, and description.

This information can be stored using the Enjin asset editor as shown here






Your asset URI will then be customized to point towards Enjin's servers, where the metadata is stored.

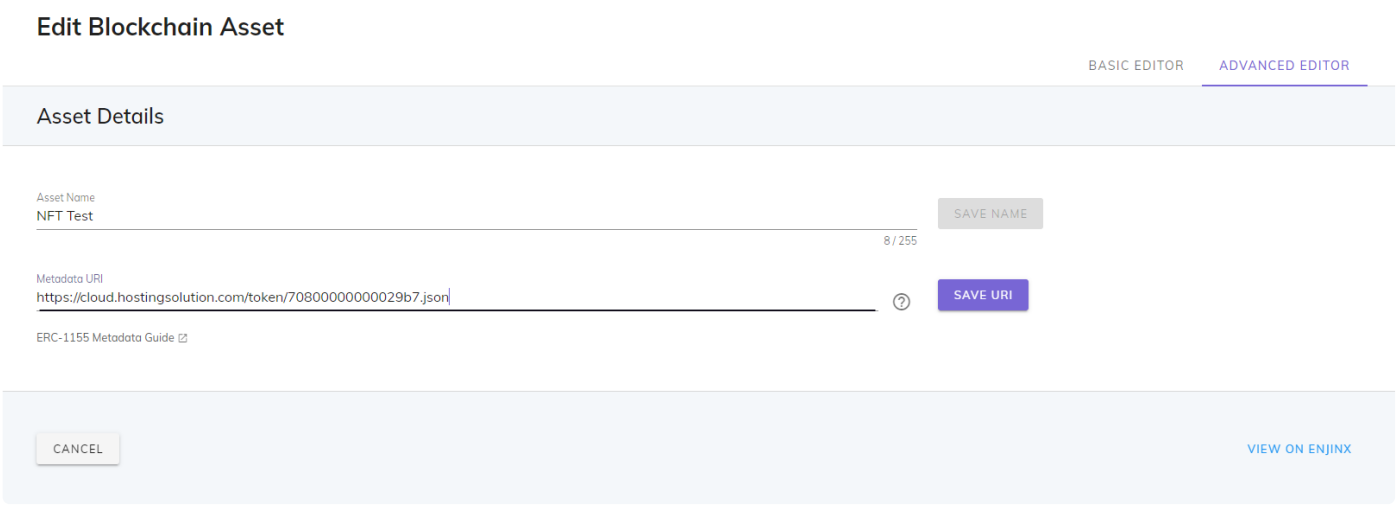
> ⓘ You can also upload .GIF and .MP4 files to your assets.

Requirements when using hosted metadata:

- 5Mb max for GIFs/regular images.
- 15Mb max for MP4 videos.

### Advanced Editor

If you want full control over your metadata, you can choose the URI using the `Advanced Editor`.




Your asset URI will then be customized to point toward the address you have specified and you will be able to customize the JSON file at your leisure.

Please note the following requirements when it comes to hosting your own metadata:

1. The link (to both metadata and image) must be publicly accessible to robots.
2. The URI must be set appropriately to the requested file.
3. The image must be that of a valid image file (the image must display online).
4. The JSON must conform with the JSON RFC standards. If it does not conform in any way, then your metadata won't be loaded by Enjin.
5. If in doubt, we recommend checking your metadata here, to make sure it's valid.

If you wish to create your own Metadata file, you will need to save it as a .json file.

Once you have your .json file uploaded with public read access, you can make the request to set the item URI (Uniform Resource Identifier).

Here is an example of a simple metadata schema:

```
{
 "name": "item_name",
 "description": "Description line 1.\nDescription line2.",
 "image": "/image.jpg"
}
```

### Specific Metadata URI

Any token ID may have a metadata URI that can be retrieved by calling `uri(_id)` on the ERC-1155 contract.

If an individual Non-Fungible token ID has a metadata URI defined, client apps should use this URI. If not defined, client apps should `call uri(_id)` on the base token id to retrieve the Default URI for the entire set of Non-Fungible tokens.

### Default URI

A Non-Fungible token that defines a Default URI in its base token has the option of using an {id} placeholder in the URI itself. This will get replaced with the distinct ID when accessing NFTs.

**Example:**

> yoursite.com/{id}.json ->
> yoursite.com/bd4818c04f57a2ebc473d74ee06d6e0600000000000000000000000000000001.json

### Images

If the Default URI contains an image property that in turn contains the {id} placeholder, the image URL will be used as the default image for all tokens of this type.

> yoursite.com/images/{id}.jpg ->
> yoursite.com/images/bd4818c04f57a2ebc473d74ee06d6e0600000000000000000000000000000001.jpg

The **image** property can also be a static URI without the placeholder, as desired.

### Using Enjin's API

If you want to set your metadata programmatically, you can do so using the following query:

```
mutation {
```

```
  SetUri(
    assetId: "70800000000029b7"
    uri:"your uri url here"
    wallet: "your wallet address"
  ) {
    transactionId
    id
    state
    value
    asset {
      name
      id
    }
    user {
      name
    }
  }
}
```

The ERC-1155 token standard includes optional formatting to allow for ID substitution by clients. If the string {id} exists in any JSON value, it MUST be replaced with the actual token ID, by all client software that follows this standard.

- The string format of the substituted hexadecimal ID MUST be lowercase alphanumeric: [0-9a-f] with no 0x prefix.
- The string format of the substituted hexadecimal ID MUST be leading zero-padded to 64 hex characters length if necessary.

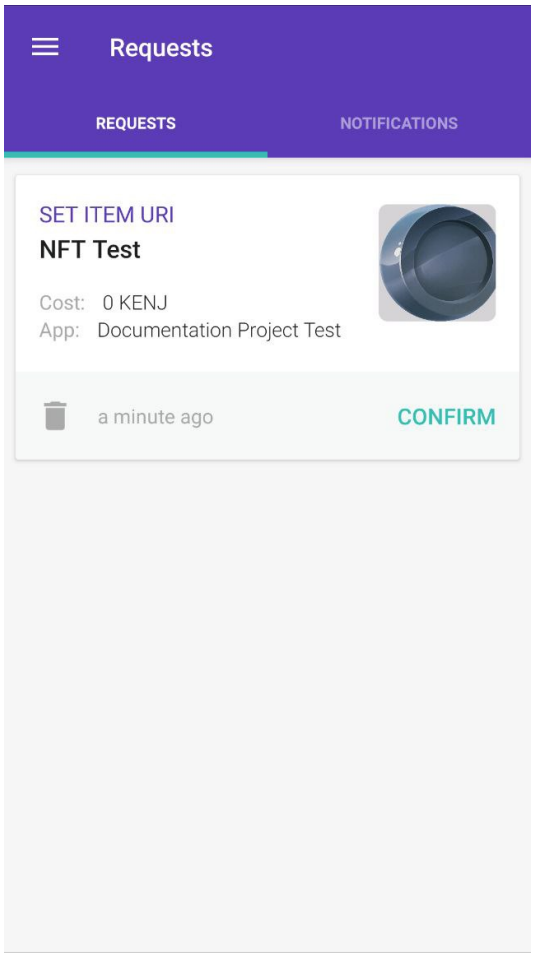
In this situation, the following address: *https://token-cdn-domain/{id}.json*

Would be replaced with: `https://token-cdn-domain/780000000000001e0000000000000000000000000000000000000000000000.json`

*If the request sent by you has all parameters correct, the response from the platform should be -*

```
{
  "data": {
    "SetUri": {
      "transactionId": null,
      "id": 98422,
      "state": "PENDING",
      "value": "0",
      "asset": {
        "name": "NFT Test",
        "id": "70800000000029b7"
      },
      "user": null
    }
  }
}
```

The request should be approved in your wallet under the requests tab, after that is done, we are ready to mint your first token!




## Minting your first Enjin Token

Now that you already have your token template and set valid metadata to it, it's time for you to mint your asset to your wallet.

The request for minting Fungible Tokens (FTs) vs Non-Fungible Tokens (NFTs) varies slightly. You can mint batches of any token type to multiple addresses or mint them to a single address.

The supply of Fungible Tokens is essentially represented by a quantity field within the token data, as opposed to Non-Fungible Tokens whose supply is represented by the quantity of separate token identities.

If you need to mint multiple NFTs in a single transaction, you will need to specify the receiving Ethereum address for each individual item.

> It's important to keep in mind that although there's no limit on the number of mint objects you can send through the mutation, there's a limit on the number of mint events that can be added into a

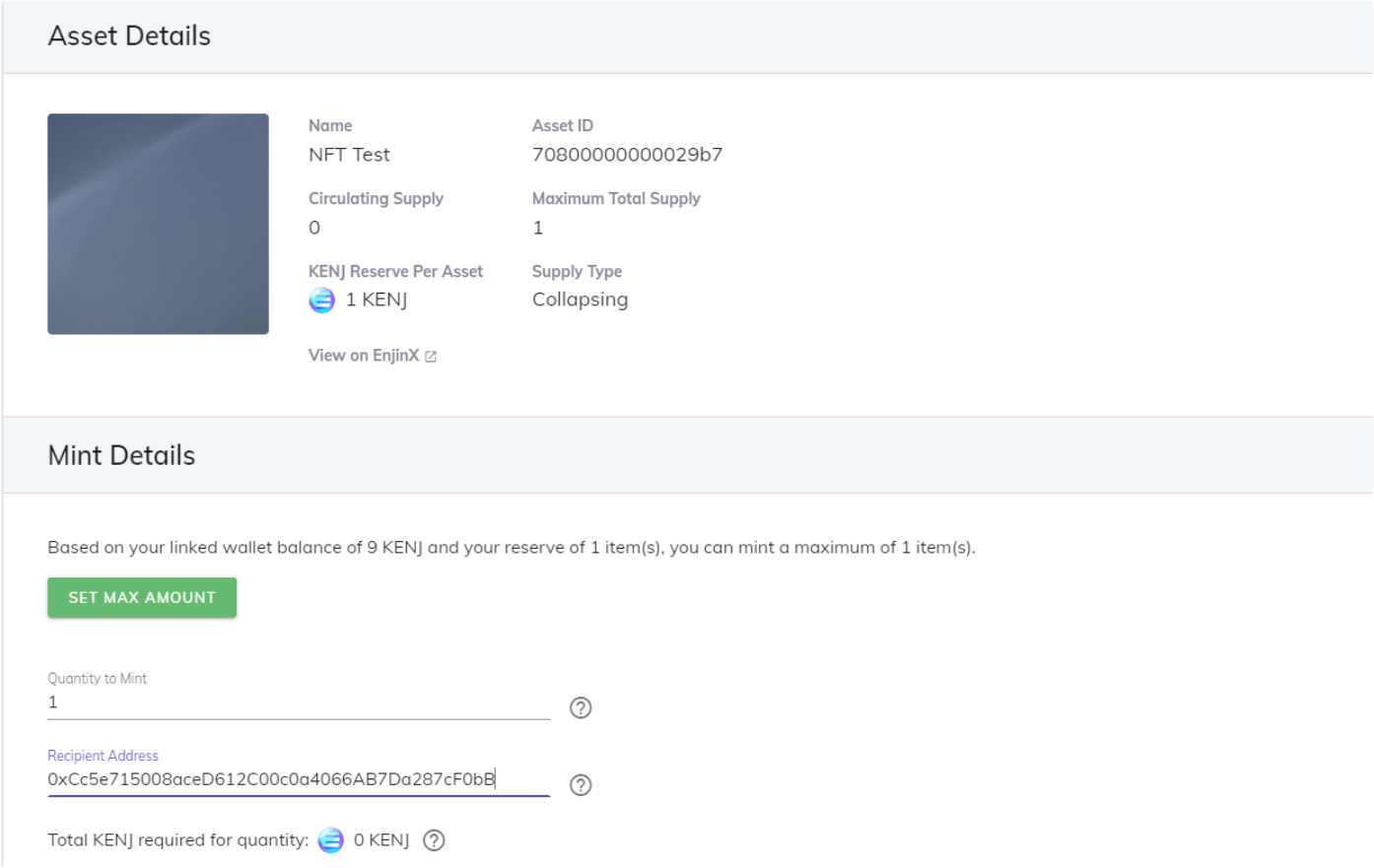
single transaction. In this case, if you specify to try and mint more than 150 assets from one request, it will be split into multiple transactions in the wallet to sign.

FTs do not have the same restriction, you can mint unlimited Fungible Tokens to an Ethereum Address. However, it is advisable not to mint any amount of Fungible Tokens into over 100 different Ethereum Addresses in one transaction.

### Minting via Platform

You can also mint via the Enjin Platform on Mainnet or on JumpNet. Minting via the Enjin Platform is simple.

1. First, locate the asset you want to mint, and click the `Mint` button on the far-right.
2. Select the quantity to mint.
3. Input an address that you'd like the asset(s) minted to.




> Minting via the Enjin Platform does have some limitations, such as you can only mint to 1 single address, rather than minting to multiple unique addresses.

### Minting using GraphiQL mutation

The following mutation allows you to mint assets over to a wallet address -

```
mutation {
  MintAsset(
```

```
    assetId: "70800000000029b7"
    mints: { to: "0xCc5e715008aceD612C00c0a4066AB7Da287cF0bB", value: 1 }
    wallet: "0xCc5e715008aceD612C00c0a4066AB7Da287cF0bB"
    send: true
  ) {
    asset {
      name
      id
    }
    transactionId
    id
    value
    wallet {
      ethAddress
    }
  }
}
```

If all the parameters of the request are correct, this is the response that you should get from the platform -

```
{
  "data": {
    "MintAsset": {
      "asset": {
        "name": "NFT Test",
        "id": "70800000000029b7"
      },
      "transactionId": null,
      "id": 98423,
      "value": "1",
      "wallet": {
        "ethAddress": "0xCc5e715008aceD612C00c0a4066AB7Da287cF0bB"
      }
    }

  }
}
```

The request should be approved in your wallet under the requests tab, after that is done, the asset should appear under the assets tab in your Enjin Wallet in a few minutes. You can also check https://goerli.enjinx.io/eth/assets with your Enjin Wallet address to have a look at your asset!

# Advanced Enjin Platform API Usage

## Enjin Platform API Requests

In the context of the Enjin Platform, by using GraphQL, developers are able to create, mint, transfer and fetch information on FTs and NFTs created using the platform without needing to interact with any back-end information from the blockchain.

As an example, we are going to display one Query and one Mutation that is used by the Enjin Platform

**Query -** with this specific query, we are able to perform a READ operation on the Enjin Database in order to retrieve information pertaining to a specific project, all queries **must** contain an object, fields, and arguments.

```
query{
  GetProject {
    id
    name
    description
    image
    createdAt
    updatedAt
  }
}
```

**Response** - In the response field, we can see that the `GetProject` query returned the Id of the project, the name, description, and the image attached to the project itself.

```
{
  "data": {

    "GetProject": {
      "id": 2912,
      "name": "Esther Project 1",
      "description": "QA Testing Project for Esther",
      "image": "<project-image>"
    }
  }
}
```

**Mutation -** Just like in queries, if the mutation field returns an object type, you can ask for nested fields. This can be useful for fetching the new state of an object after an update. Let's look at a simple example of a mutation.

```
mutation {
  Message(
    message: "This is a Test"
    wallet: "0x7AEAB7C231c88611a2ad434d3A2715Ab3C91F406"
  ) {
    type
    value
    state
```

```
  }
}
```

**Response -** with this mutation, we are sending a request to the platform to sign a message through the blockchain, which still needs to get approved by the wallet in order to change the information requested by us.

```
   "Message": {
      "type": "MESSAGE",
      "value": "0",
      "state": "PENDING"
    }
  }
}
```

## Managing Players

Learn how to manage your players with a few requests available in the Enjin Platform API

The following pages are going to go through important queries and mutations related to players in your project that should assist while integrating your project with the Enjin API.

## Linking a Player Wallet

Learn how to create a player identifier for your users and how to generate a linking code on their behalf.

#### CreatePlayer

Create a Player identity for one of the users by running this specific mutation.

```
mutation {
  CreatePlayer(id: "playername") {
    accessToken
  }
```

```
}
```

> *We recommend setting the username from your app as a static identifier from your existing database, so you always know how to reference the user's Enjin identity.*

The `getPlayer` query will return a linking code that you can display to the user. The user will scan the linking code with the Enjin Wallet and link their wallet to your project.

```
{
  GetPlayer(id: "playername") {
    id
    wallet {
      ethAddress
    }
    linkingInfo {
      code
      qr
    }
    createdAt
    updatedAt
  }
}
```

After the user links their wallet, you can run that same exact query to return the user-linked wallet address, you can save the address to query the user wallet anytime you may need to.

## Unlinking a Wallet

Learn how to unlink a player wallet by running a simple mutation.

#### UnlinkWallet

Run the `UnlinkWallet` mutation on GraphiQL with the user address as one of the parameters to unlink the user wallet.

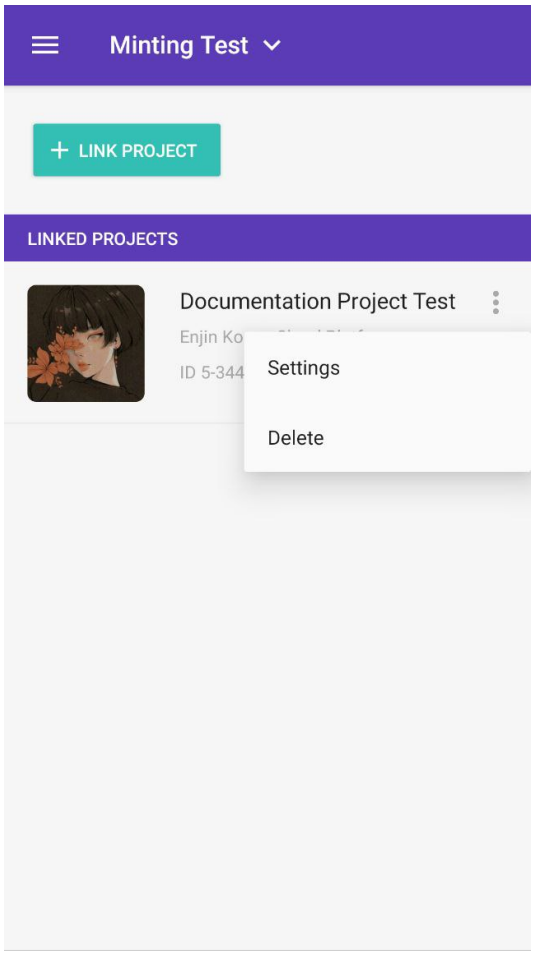
```
mutation {
  UnlinkWallet(address: "<address here>")
```

If the mutation has a valid wallet address, the platform will return a `true` response as shown below.

```
{
  "data": {
    "UnlinkWallet": true
  }
}
```

With that, you should be able to link the user's wallet again in case the user needs to switch their wallet or link it again through the LinkWallet mutation.

> Users are also able to unlink their wallets manually via the Enjin Wallet by going to the "Linked Projects" section, tapping on the project, tapping on the 3-dot menu, and selecting "Delete".



# Viewing Player Tokens

Learn how to view the contents of a user's verified wallet, which will then allow you to provide benefits and content based on which NFTs they own.

This specific query is used to view the contents of a user's verified wallet, with that, you should be able to interact with the user through your project based on how your project interacts with balances.

#### GetWallet

To check tokens owned by your users, you can run this specific query.

```
query {
  GetWallet(ethAddress: "<address here>") {
    ethAddress
    enjAllowance
    enjBalance
    ethBalance
    balances {
      asset {
        name
      }
      id
      index
      value
      project {
        name
      }
    }
  }
}
```

The expected response from this query should be

```
{
  "data": {
    "GetWallet": {
      "ethAddress": "0x7AEAB7C231c88611a2ad434d3A2715Ab3C91F406",
      "enjAllowance": 1.157920892373162e+59,
      "enjBalance": 23264.24307202013,
      "ethBalance": 1.2266632788,
      "balances": [
        {
          "asset": {
            "name": "Gate Key"
          },
          "id": "7000000000000b1d",
```

```
          "index": "0000000000000000",
          "value": 7,
          "project": null
        },
        {
          "asset": {
            "name": "King's Crown"
          },
          "id": "7000000000000b1e",
          "index": "0000000000000000",
          "value": 3,
          "project": null
        }
```


With that, you are able to view the ETH, and ENJ balance of the users, their wallet allowance, and token balances created by your project.

## Viewing Assets

Learn how to view assets from your project with the GetAssets query.

With the `GetAsset` query, you can get information regarding assets linked to your project/collection.

#### GetAsset

This query allows you to retrieve specific information from assets linked to your project.

```
query {
  GetAsset(id: "<token id here>") {
    id
    name
    createdAt
    updatedAt
    wallet {
      ethAddress
    }
  }
}
```

The expected output of this query should look like this.


```
{
  "data": {
    "GetAsset": {
      "id": "708000000000088e",
      "name": "Tassio Test 2",
      "createdAt": "2021-06-16T21:18:59+00:00",
      "updatedAt": "2021-06-16T21:39:39+00:00",
```

```
      "wallet": {
        "ethAddress": "0x7AEAB7C231c88611a2ad434d3A2715Ab3C91F406"
      }
    }
  }
}
```

## Managing Assets

Learn how to manage your assets with a few requests available in the Enjin Platform API

The Enjin API has been built to provide all the functionality you need to manage a robust blockchain-based gaming economy.

The following pages are going to go through important queries and mutations related to assets in your project that should assist while integrating your project with the Enjin API.

You will likely use the following queries and mutations quite often when it comes to managing your tokenized assets in your project or in your game.

Some of these requests involves managing your assets in different ways to customise the experience how you want your NFTs to work in your integration.

## Enjin Assets Bridge

The Enjin Bridge allows you to transfer tokens between Ethereum Mainnet and Enjin JumpNet.

Using the Bridge, both Enjin Coin (ENJ) and ERC-1155 assets created using the Enjin Platform can be transferred between the Enjin JumpNet and Ethereum Mainnet networks.

The Enjin Bridge smart contract receives an asset on one network, and once claimed, replicates the asset on the corresponding bridged network.

> ⓘ Currently, the asset bridging functionality is only accessible through the Enjin Platform API. The user interface component, accessible within the Enjin Wallet, will be released at a later stage.

**Mutations:**

The Enjin Bridge has 3 mutations:

| Mutation | Usage |
| --- | --- |
| BridgeAsset | Used to transfer a single asset. |
| BridgeAssets | Used to transfer multiple assets. |
| BridgeClaimAsset | Used to claim assets on the opposite network. |

**BridgeAsset**

The `BridgeAsset` mutation allows you to send one single asset/NFT to the bridge contract. You can run the following mutation to get started:

```
mutation {
  BridgeAsset(
    assetId: "tokenId"
    assetIndex: "0000000000000001"
    value: "1"
    wallet: "walletaddress"
  ) {
    id

    blockchainData {
      encodedData
    }
  }
}
```

**Steps & Process:**

1. Call the `BridgeAsset` mutation.
2. Confirm the request in your wallet.
3. The asset will moved from your address to the bridge, ready to be claimed from the other side of the bridge.

**Arguments:**

- `assetId` (type: `String!`, example: `"7000000000000002"`)
- `assetIndex` (optional, type: `String`, example: `"0"`)
- `value` (optional, type: `BigInt`, example: `"5"`, default: `"1"`)
  - Must be omitted, or `"1"`, when asset is non-fungible.
- `wallet` (optional, type: `EthAddress`, project-schema only)

### BridgeAssets

The `BridgeAssets` mutation allows you to send multiple digital assets/non-fungible tokens (NFTs).

You can send multiple different indices using the `BridgeAssets` function.

> ⓘ The maximum number of indices that can be bridged at any given time is 100 per transaction.

```
mutation {
  BridgeAssets(
    assetId: "tokenId"
    assetIndices: "0000000000000001"
    value: "1"
    wallet: "walletaddress"
  ) {
    id
    blockchainData {
      encodedData
    }
  }
}
```

**Steps & Process:**

1. Call the `BridgeAssets` mutation.
2. Confirm the request in the wallet.
3. The assets will be moved from your address to the bridge, ready to be claimed from the other side of the bridge.

**Arguments:**

- `assetId` (type: `String!`, example: `"7000000000000002"`)
- `assetIndices` (optional, type: `[String!]!`, example: `[1, 2, 3]`)
- `wallet` (optional, type: `EthAddress`, project-schema only)

### BridgeClaimAsset

The `BridgeClaimAsset` mutation allows you to claim your asset/NFT on the Enjin Bridge.

> ⓘ This is a **temporary mutation** that will be natively integrated with the upcoming Enjin Wallet 2.0 release. This mutation will therefore be deprecated in the future after support has been implemented into the updated wallet.

> ⓘ If the asset has never been transferred to the destination network before, this mutation will have to be run twice. The first instance will create the relevant asset definition on the network and after the transaction has been mined you can call this mutation a second time to complete the transfer of the asset to the network.

```
mutation {
  BridgeClaimAsset(assetId: "assetid", wallet: "walletaddress") {
    id
  }
}
```

**Steps & Process:**

1. After the bridging transaction has been mined successfully, wait for approximately 20 blocks. You'll then be able to use this method to claim it on the destination network.
2. You will need to specify the `assetId`, and you can optionally (on the project schema) override the `wallet` in question.
3. Confirm the request in your wallet.
4. The bridge will fulfill the claim by sending you the exact asset initiated from the other side of the bridge.

## Trade Requests

Learn more about how to set-up trading in your integration project.

Trading requests are possible with the Enjin API, with trading requests, you are able to interact with players through a `project > player` trade or `player > player` trade within your project, allowing you to set up an exchange environment within your own project.

Using trade requests, both Enjin Coin (ENJ) and ERC-1155 assets created using the Enjin Platform can be traded.

Mutations:

The Enjin Trade has 2 mutations:

| Mutation | Usage |
| --- | --- |
| CreateTrade | Used to initiate the trade request. |
| CompleteTrade | Used to complete the trade request based on parameters from the CreateTrade request. |

**CreateTrade**

The CreateTrade mutation allows you to trade assets (FTs or NFTs) and trade assets for ENJ as well.

```
mutation {
  CreateTrade(
    askingAssets: [{ assetId: "XXXXXXXXXX", value: 1 }],
    offeringAssets: [{ assetId: "XXXXXXXXXX", value: 1, assetIndex:"XXXXXXXXXX"}],
    secondParty: "WalletAddress"
    wallet: "WalletAddress"
  )
 {
    transactionId
    id
    state
    value
    asset{name id}
    user{name}
  }
}
```

Note that the `wallet` parameter is the initiator, while the `secondParty` is the person who's interacting with the trade.

After approving the request through the wallet, you may want to query the status of this specific transaction with the `GetTransaction` query as you will need to have the id for the transaction in order to complete the trade.

### CompleteTrade

Use this mutation to complete the trade request initiated by the previous mutation.

```
mutation {
  CompleteTrade(
    tradeId: "XXXXXX",
    wallet: "WalletAddress"
  )
 {
    transactionId
    id
    state
    value
    asset{name id}
    user{name}
  }
}
```

# Batch Sending Assets

Learn more about batch sending assets

Batch Sending Assets is a useful mutation if you are planning on sending assets to multiple users through a single transaction.

Using this request will allow you to send an asset to users of your project or request an asset to be transferred from `user A` to `user B`

The `AdvancedSendAsset` request has one single mutation.

| Mutation | Usage |
| --- | --- |
| AdvancedSendAsset | Send one or more assets in one transaction. |

> ⓘ The `value` property is only applicable in case you are minting FTs as NFTs are assumed to have a value of 1.

### AdvancedSendAsset

Use this mutation to send assets from one wallet to another wallet.

```
mutation {
  AdvancedSendAsset(
    transfers: [
      {
        from: "WalletAddress"
        to: "WalletAddress"
        assetId: "xxxxxxxxxxxxx"
        assetIndex: "xxxxxxxxxxxxx"
        value: "1"
      }
      {
        from: "WalletAddress"
        to: "WalletAddress"
        assetId: "xxxxxxxxxxxxx"
        assetIndex: "xxxxxxxxxxxxx"
        value: "1"
      }
    ]
  ) {
    transactionId
    id
    state
    value
    asset {
      name
```

```
      id
    }
    user {
      name
    }
  }
}
```

If all the request parameters are correct, this should be the expected response.

```
{
  "data": {
    "AdvancedSendAsset": {
      "transactionId": null,
      "id": 31369,
      "state": "PENDING",
      "value": "1",
      "asset": {
        "name": "New Test Asset",
        "id": "7880000000000089f"
      },
      "user": null
    }
  }
}
```

After approving the transaction, the tokens should be transferred over to the address once the transaction is completed.

## Batch Minting Assets

Batch Minting Assets is a useful mutation if you are planning on minting assets to multiple users through a single transaction.

Using this request will allow you to mint an asset to users of your project.

The `MintAsset` request has one single mutation.

| Mutation | Usage |
| --- | --- |
| MintAsset | Mint one or more assets into a single transaction. |

> It's important to keep in mind that although there's no limit on the number of mint objects you can send through the mutation, there's a limit on the number of mint events that can be added into a single transaction. In this case, if you specify to try and mint more than 150 assets from one request, it will be split into multiple transactions in the wallet to sign.

> ⓘ The `value` property is only applicable in case you are minting FTs as NFTs are assumed to have a value of 1.

**MintAsset**

```
mutation {
  MintAsset(
    assetId: "xxxxxxxxxxx"
    mints: [
      { to: "WalletAddress", value: 1 }
      { to: "WalletAddress", value: 1 }
      { to: "WalletAddress", value: 1 }
    ]
    wallet: "WalletAddress"
    send: true
  ) {

    asset {
      name
      id
    }
    transactionId
    id
    value
    wallet {
      ethAddress
    }
  }
}
```

If all the request parameters are correct, this should be the expected response.

```
{
  "data": {
    "MintAsset": {
      "asset": {
        "name": "Tassio Test 2",
        "id": "7080000000000088e"
      },
      "transactionId": null,
      "id": 31237,
      "value": "1",
      "wallet": {
        "ethAddress": "0x7AEAB7C231c88611a2ad434d3A2715Ab3C91F406"
      }
    }
  }
```

```
}
```

After approving the transaction, the tokens should be minted over to the address once the transaction is completed.

## Releasing Assets from Reserve

Learn more about the ReleaseReserve mutation and how to use it in your project.

The `releaseReserve` is a great mutation in case you need to release the ENJ(or JENJ) from assets that you didn't mint yet but still have a supply available to mint.

Using this request will allow you to release the reserve of assets that you didn't mint yet.

The `ReleaseReserve` request has one single mutation.

| Mutation | Usage |
| --- | --- |
| ReleaseReserve | Release the ENJ from an asset that hasn't been minted yet. |

#### ReleaseReserve

```
mutation {
 ReleaseReserve(
  assetId: "xxxxxxxxxx"
  value: "x"
  wallet: "xxxxxxxxx"
  ) {
    transactionId
    id
    state
    value
    asset {
      name
      id
    }
    user {
      name
    }
  }
}
```

If all the request parameters are correct, this should be the expected response.

```
{
  "data": {
```

```
    "ReleaseReserve": {
      "transactionId": null,
      "id": xxxxxx,
      "state": "PENDING",
      "value": "3",
      "asset": {
        "name": "tokenName",
        "id": "xxxxxxx"
      },
      "user": null
    }
  }
}
```

After approving the transaction, the infused ENJ should be released from reserve and returned to the token creator wallet address.

## Send Enjin Request

Learn more aboout the Send Enjin Request

The `sendEnj` is a mutation that allows you to request your users to send Enjin (or JENJ) from their wallet to another wallet address. The user involved in the request should be the one approving this specific transaction.

Using this request will allow you to request users to send ENJ from their wallet to another wallet address.

The `sendEnj` request has one single mutation.

| Mutation | Usage |
| --- | --- |
| sendEnj | Send Enjin or Jumpnet Enjin from wallet A to wall B. |

#### SendEnj

> ⓘ Multiply the value by 10^18 to include 18 decimals on the value field

```
mutation {
  SendEnj(
    to: "WalletAddress"
    value: "100000000000000000"
    wallet: "WalletAddress"
  ) {
    transactionId
    id
```

```
    state
    value
  }
}
```

If all the request parameters are correct, this should be the expected response.

```
{
  "data": {
    "SendEnj": {
      "transactionId": null,
      "id": xxxxx,
      "state": "PENDING",
      "value": "0.1"
    }
  }
}
```

After approving the transaction, the ENJ or JENJ should be transferred to the specified address.

## Advanced Metadata Guide

Learn Advanced Metadata Parameters and how to use it while interacting with your Enjin Tokens.

If you wish to create your own Metadata file, you will need to save it as a .json file.

Once you have your .json file uploaded with public read access, you can make the request to set the item URI (Uniform Resource Identifier).

Here is an example of a simple metadata schema:

```
{
 "name": "item_name",
 "description": "Description line 1.\nDescription line2.",
 "image": "/image.jpg"
}
```

For more information on how to create a more robust JSON file, visit the ERC-1155 Github.

#### Specific Metadata URI

Any token ID may have a metadata URI that can be retrieved by calling `uri(_id)` on the ERC-1155 contract.

If an individual Non-Fungible token ID has a metadata URI defined, client apps should use this URI. If not defined, client apps should `call uri(_id)` on the base token id to retrieve the Default URI for the entire set of Non-Fungible tokens.

### Default URI

A Non-Fungible token that defines a Default URI in its base token has the option of using an {id} placeholder in the URI itself. This will get replaced with the distinct ID when accessing NFTs.

**Example:**

> yoursite.com/{id}.json ->
> yoursite.com/bd4818c04f57a2ebc473d74ee06d6e0600000000000000000000000000000001.json

### Images

If the Default URI contains an image property that in turn contains the {id} placeholder, the image URL will be used as the default image for all tokens of this type.

> yoursite.com/images/{id}.jpg ->
> yoursite.com/images/bd4818c04f57a2ebc473d74ee06d6e0600000000000000000000000000000001.jpg

The **image** property can also be a static URI without the placeholder, as desired.

### Asset backgrounds

If you opt for generating your own metadata, in the JSON you can set various coloured backgrounds for your assets to display in the Enjin Wallet app.

To do this, you will need to set the `color`: attribute in your JSON file, such as the following example:

JSON

```
{
  "name": "John Wick",
  "description": "My name is John Wick",
  "image": "image url",
  "color": "teal"
}
```

You can set the following colors:

- `grey => #616266`
- `purple => #5b3cb6`
- `green => #477b19`
- `blue => #3761a8`

- `orange => #9b6132`
- `pink => #993b9a`
- `teal => #3b7e9a`
- `red => #9a3b3b`

### Asset properties

If you are interested to set unique properties to your assets, you can do this via your custom JSON file.

To do this, you will need to set the `properties` attribute in your JSON file, such as the following example:

JSON

```
{
        "name": "Asset Name",
        "description": "Lorem ipsum...",
        "image": "https:\/\/s3.amazonaws.com\/your-bucket\/images\/{id}.png",
        "properties": {
                "simple_property": "example value",
                "rich_property": {
                        "name": "Name",
                        "value": "123",
                        "display_value": "123 Example Value",
                        "class": "emphasis",
                        "css": {
                                "color": "#ffffff",
                                "font-weight": "bold",
                                "text-decoration": "underline"
                        }
                },
                "array_property": {
                        "name": "Name",
                        "value": [1,2,3,4],
                        "class": "emphasis"
                }
        }
}
```

You can set as many different properties as you'd like. Once set, these unique properties will display on your asset in the Enjin Wallet app, and on EnjinX.

### Localization

Metadata localization should be standardized to increase presentation uniformity across all languages. As such, a simple overlay method is proposed to enable localization. If the metadata JSON file contains a `localization` attribute, its content MAY be used to provide localized values for fields that need it. The

`localization` attribute should be a sub-object with three attributes: `uri`, `default` and `locales`. If the string `{locale}` exists in any URI, it MUST be replaced with the chosen locale by all client software.

BaseURI

```
{
  "name": "Advertising Space",
  "description": "Each token represents a unique Ad space in the city.",
  "localization": {
    "uri": "ipfs://QmWS1VAdMD353A6SDk9wNyvkT14kyCiZrNDYAad4w1tKqT/{locale}.json",
    "default": "en",
    "locales": ["en", "es", "fr"]
  }
}
```

es.json

```
{
  "name": "Espacio Publicitario",
  "description": "Cada token representa un espacio publicitario único en la ciudad."
}
```

fr.json

```
{
  "name": "Espace Publicitaire",
  "description": "Chaque jeton représente un espace publicitaire unique dans la ville."
}
```

**Setting Metadata based on the NFT index**

When setting individual images and descriptions for NFTs, we must take into consideration the fact that NFTs are unique and the whole purpose of having NFTs is the ability to set different parameters for tokens that share the same token ID with individual indexes.

**Note:** When doing this, you need your own internal CDN, somewhere to host the metadata URI JSON files.

**Example:**

Index 0: Token Group

- https://enjinx.io/eth/asset/56082269

Index +1: Individual Tokens

- https://enjinx.io/eth/asset/18800000000004c7/assets

This is the asset that we are going to use as an example of how to set individual metadata.

This asset has a total supply of 3 NFTs, so we will need to host 4 individual, different JSON files. One for the 0 index:

- `0x70800000000005c1000000000000000000000000000000000000000000000000`

Another 3 for the n+1 indexes:

- `0x70800000000005c1000000000000000000000000000000000000000000000001`
- `0x70800000000005c1000000000000000000000000000000000000000000000002`
- `0x70800000000005c1000000000000000000000000000000000000000000000003`

On your hosted CDN, you will need to host the files there accordingly, your image(s) of the NFTs and the JSON files. We typically name our hosted files like the following:

- `cdn.enjin.io/mint/image/70800000000005c1000000000000000000000000000000000000000000000000.jpg` for the hosted image, and
- `cdn.enjin.io/mint/image/70800000000005c1000000000000000000000000000000000000000000000000.json` for the hosted JSON file.
- Note the ending of the hosted files (.jpg for the image, and .json for the metadata file).

You will want to host them in order by the NFT index, so the token ID at the start, then the token Index at the end before the {.json}.

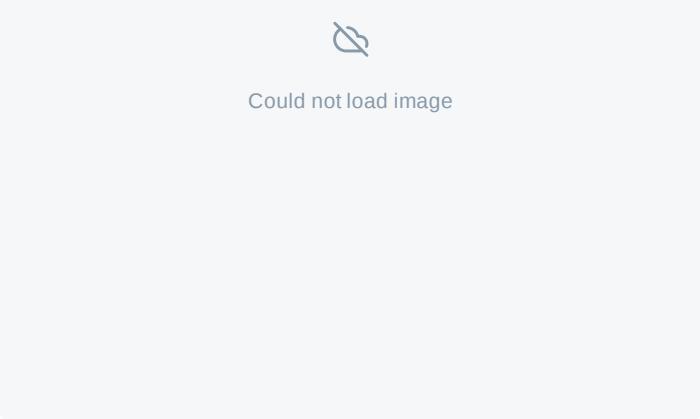
Example:

1. `cdn.enjin.io/mint/file/70800000000005c1000000000000000000000000000000000000000000000001.json`
2. `cdn.enjin.io/mint/file/70800000000005c1000000000000000000000000000000000000000000000002.json`

Step 2

Now that we have all of our files hosted based on the full token id (without the 0x) and index, it's time to set the metadata in the web panel. You will need to go to the `Advanced Editor` option.

https://cdn.enjin.io/mint/meta/{id}.json - This should be the path that we set in the Metadata URI field of the web panel, it means that EnjinX and our Platform/API, will query for the token id related to that specific token, and the index sequence.




Paste your own CDN path in the `Metadata URI` field and ensure you have titled {id}.json in the file as shown in our example screenshot above. After setting it up, accept the request in the wallet and you should have an NFT with different images based on the token index.

> ⓘ It's also possible to set the metadata using the GraphQL Playground with the `SetUri` mutation.

#### Invalidating Asset Metadata

If your asset metadata doesn't load or takes a while to load, we have implemented the `Invalidate Metadata` mutation to push the metadata through our servers so that your asset metadata can appear instantly.

```
mutation InvalidateTokenMetadata {
  InvalidateAssetMetadata(id: "<tokenId>")
}
```

`Id` is the token ID of the asset you want to invalidate.

This mutation will instruct the Platform to invalidate the metadata and thus fetch it again, directly from your server.

Please note the following when running this particular mutation:

1. It can take a few minutes, after invalidating your asset, for the new metadata to load.
2. This mutation can only be run once, per asset, every few minutes.
3. You can only run this mutation on assets that belong to a project, in which you have the minter role (or higher) on.

## Enjin Wallet Deep Linking

Learn more about the Enjin Wallet Deep Linking.

## Deep Links

Deep links are URIs of any scheme that takes users directly to a specific part of apps. In the context of the Enjin Wallet, Deep links can assist project developers while taking users directly into a few features available in the Enjin Wallet, such as Marketplace, Beam, and Wallet Linking to a specific Project or Collection. Android documentation reference.

| DeepLink | Function |
| --- | --- |
| enjinwallet://marketplace/code | Use this Deeplink to open the buy asset page fror the EnjinX Marketplace. |
| enjinwallet://marketplace/code?network= | Use this Deeplink to open the buy asset page fror the EnjinX Marketplace while specifying which network the asset is coming from. |
| enjinwallet://beam/code?network= | Use this Deeplink to open the beam QR claim pag while specifying which network the asset is comin from. |
| enjinwallet://link/code | Use this deep link to provide it over to users while linking their wallets to your project. |

Available Networks:

- Ethereum
- Jumpnet
- Goerli

**Examples**

`enjinwallet://marketplace/eb0ee74a-9e97-4db1-9535-0e1cd149023a`

`enjinwallet://marketplace/eb0ee74a-9e97-4db1-9535-0e1cd149023a?network=jumpnet`

`enjinwallet://beam/60f93c8e038fc32afa829f7d?network=jumpnet`

`enjinwallet://link/AF5434K`

# API Glossary

Learn more about all Queries and Mutations available in the Enjin Platform API.

The following pages are going to go over all the existing Queries and Mutations in the Enjin Platform with examples and responses.

> ⓘ Further explanation on each and every parameter available can be found here.

**Project & Player Queries**

**Project & Player Mutations**

## Project & Player Queries

### GetProject

`GetProject` Queries information about your project.

Project and Player*

```
query {
  GetProject {
    id
    name
    description
    image
    createdAt
    updatedAt
  }
}
```

**Expected output:**

```
{
  "data": {
    "GetProject": {
      "id": xxxx,
      "name": "ProjectName",
      "description": "Description Project",
      "image": "<project-image>"
    }
  }
```

```
}
```

### AuthProject

`AuthProject` queries the unique access token of your project by using the secret.

Project*

```
query {
  AuthProject(id: xxxx, secret:"appsecret"){
    accessToken
    expiresIn
  }
}
```

**Expected output:**

```
{
  "data": {
    "AuthProject": {
      "accessToken": "accesstoken",
      "expiresIn": 86400
    }
  }
}
```

### AuthPlayer

`AuthPlayer` use this query to get the AccessToken from a specific user.

Project*

```
query {
  AuthPlayer(id: "User"){
    accessToken
    expiresIn
  }
}
```

**Expected output:**

```
{
  "data": {
    "AuthPlayer": {
      "accessToken": "accesstoken",
      "expiresIn": 86399
    }
  }
```

```
}
```

### GetPlayers

`GetPlayers` queries for your created players on your project.

Project*

```
query {
  GetPlayers {
    id
    wallet {
      ethAddress
    }
    linkingInfo {
      code
      qr
    }
    createdAt
    updatedAt
  }
}
```

**Expected output:**

```
{
        "id": "BOB",
        "wallet": null,
        "linkingInfo": {
          "code": "xxxxxx",
          "qr": "https://chart.googleapis.com/chart?chs=512x512&cht=qr&chl=xxxxx"
        },
        "createdAt": "2021-06-08T10:31:28+00:00",
        "updatedAt": "2021-06-08T10:31:28+00:00"
      }
```

### GetPlayer

`GetPlayer` queries for a specific player that is part of your project.

Project & Player*

```
{
  GetPlayer(id: "yourUserName") {
    id
    wallet {
      ethAddress
    }
    linkingInfo {
```

```
      code
    }

    createdAt
    updatedAt
  }
}
```

**Expected output:**

```
{
  "data": {
    "GetPlayer": {
      "id": "User",
      "wallet": {
        "ethAddress": "walletaddress"
      },
      "linkingInfo": null,
      "createdAt": "2021-06-11T11:42:32+00:00",
      "updatedAt": "2021-06-11T12:29:20+00:00"
    }
  }
}
```

### GetBalances

`GetBalances` queries for all balances on a project.

Project & Player*

```
query {
  GetBalances {
    items {
      asset {
        name
        id
      }
      index
      value
    }
  }
}
```

**Expected output:**

```
{
  "data": {
    "GetBalances": {
```

```
      "items": [
          "asset": {

            "name": "Flower",
            "id": "7080000000000594"
          },
          "index": "0000000000000001",
          "value": 1
        }
```

### GetGasPrice

`GetGasPrice` queries for the latest gas prices.

Project & Player*

```
query {
  GetGasPrice {
    fast
    fastest
    safeLow
    average
  }
}
```

**Expected output:**

```
{
  "data": {
    "GetGasPrice": {
      "fast": 15,
      "fastest": 30,
      "safeLow": 3,
      "average": 8
    }
  }
}
```

### GetPlatform

`GetPlatform` queries information about the platform.

Project & Player*

```
query {
  GetPlatform {
    id
    name
    network
```

```
    contracts {
      enj
      cryptoItems
      platformRegistry
      supplyModels {
        fixed
        settable
        infinite
        collapsing
        annualValue
        annualPercentage
      }
    }
  }
}
```

**Expected output:**


```
{
  "data": {
    "GetPlatform": {
      "id": 7,
      "name": "Enjin Platform",
      "network": "Goerli",
      "contracts": {
        "enj": "0xf6fe970533fe5c63d196139b14522eb2956f8621",
        "cryptoItems": "0x44B8326616C2a7Ac37A2d2719E067d5aA8DaD0ba",
        "platformRegistry": "0x0000000000000000000000000000000000000000",
        "supplyModels": {
          "fixed": "0x0000000000000000000000000000000000000000",
          "settable": "0x4f98f38c6667fee10589c33d4280d821693e701e",
          "infinite": "0x46fc98b824072ed89b98c642e0d0232f0dd52806",
          "collapsing": "0xdc9fe2cdad407b60001e26455a0a4edbe1317f2a",
          "annualValue": "0x4f183ceee5f7567b70b3be737921d8d12e876397",
          "annualPercentage": "0x4aea9c3cc71e32c877ea4914f49e477cfe0cc2d8"
        }
      }
    }
  }
}
```


### GetTransaction

`getTransaction` finds further information regarding a transaction performed by you through GraphiQL, you can add further parameter results based on your needs.

Project & Player*

```
query {
  GetTransaction(id:98420) {
```

```
    id
    transactionId
    title

    type
    value
    state
    projectWallet
    asset{id, name}
  }
}
```

**Expected output:**

```
{
  "data": {
    "GetTransaction": {
      "id": 98420,
      "transactionId": "0x5e0790f6f253e11dae566a23760da161931ab6305b863289e3bbe2725536abfb",
      "title": "NFT Test",
      "type": "CREATE",
      "value": "1",
      "state": "EXECUTED",
      "projectWallet": true,
      "asset": {
        "id": "70800000000029b7",
        "name": "NFT Test"
      }
    }
  }
}
```

### GetWallet

`GetWallet` queries information for a specific wallet address.

Project & Player*

```
query {
  GetWallet(ethAddress: "0x7AEAB7C231c88611a2ad434d3A2715Ab3C91F406") {
    ethAddress
    enjAllowance
    enjBalance
    ethBalance
    balances {
      asset{name}
      id
      index
      value
      project {
        name
```

```
    } }
    assetsCreated {

      id
      items {
        name
        id
      }
    }
  }
}
```

**Expected Output:**


```
{
  "data": {
    "GetWallet": {
      "ethAddress": "0x7AEAB7C231c88611a2ad434d3A2715Ab3C91F406",
      "enjAllowance": 1.157920892373162e+59,
      "enjBalance": 23264.24307202013,
      "ethBalance": 1.2266632788,
      "balances": [
        {
          "asset": {
            "name": "Gate Key"
          },
          "id": "7000000000000b1d",
          "index": "0000000000000000",
          "value": 7,
          "project": null
        },
        {
          "asset": {
            "name": "King's Crown"
          },
          "id": "7000000000000b1e",
          "index": "0000000000000000",
          "value": 3,
          "project": null
        }
```


### GetWallets

`GetWallets` queries multiple wallets at once linked to your project.

Project*

```
query {
  GetWallets(
    ethAddresses: [
```

```
      "0x7AEAB7C231c88611a2ad434d3A2715Ab3C91F406"
    ]

  ) {
    ethAddress
    enjAllowance
    enjBalance
    ethBalance
    balances {
      asset {
        name
      }
      id
      index
      value
      project {
        name
      }
    }
    assetsCreated {
      id
      items {
        name
        id
      }
    }
  }
}
```

### Expected Output


```
{
  "data": {
    "GetWallets": [
      {
        "ethAddress": "0x7AEAB7C231c88611a2ad434d3A2715Ab3C91F406",
        "enjAllowance": 1.157920892373162e+59,
        "enjBalance": 23264.24307202013,
        "ethBalance": 1.2266632788,
        "balances": [
          {
            "asset": {
              "name": "Gate Key"
            },
            "id": "7000000000000b1d",
            "index": "0000000000000000",
            "value": 7,
            "project": n
            ull
          }
```

### GetAsset

`GetAsset` queries specific information from assets linked to your project.

Project & Player*

```
query {
  GetAsset(id: "708000000000088e") {
    id
    name
    createdAt
    updatedAt
    wallet {
      ethAddress
    }
  }
}
```

**Expected output:**

```
{
  "data": {
    "GetAsset": {
      "id": "708000000000088e",
      "name": "Tassio Test 2",
      "createdAt": "2021-06-16T21:18:59+00:00",
      "updatedAt": "2021-06-16T21:39:39+00:00",
      "wallet": {
        "ethAddress": "0x7AEAB7C231c88611a2ad434d3A2715Ab3C91F406"
      }
    }
  }
}
```

## Project & Player Mutations

### Unlink Wallet

`Unlink Wallet` is used to unlink a wallet from a specific project.

Project & Player*

```
mutation{
  UnlinkWallet(address: "<address here>")
}
```

**Expected output:**

```
{
  "data": {
    "UnlinkWallet": true
  }
}
```

### Create Asset

`CreateAsset` is used to create assets (your token blueprints) on the blockchain.

Project*

```
mutation {
  CreateAsset(
    name: "AssetName"
    totalSupply: "1"
    initialReserve: "1"
    supplyModel: COLLAPSING
    meltValue: "1000000000000000000"
    meltFeeRatio: 0
    transferable: PERMANENT
    transferFeeSettings: {type:PER_TRANSFER, assetId: "0", value: "0" }
    nonFungible: true
    wallet: "<address here>"
  ) {
    value
    id
    transactionId
    title
    state
    project{name}
  }
}
```

**Expected Output:**

```
{
  "data": {
    "CreateAsset": {
      "value": "1",
      "id": 31179,
      "transactionId": null,
      "title": "AssetNamegra",
      "state": "PENDING",
      "project": {
        "name": "V2 Testing Project"
      }
    }
  }
```

```
}
```

### MintAsset

`MintAsset` is used to mint assets with the parameters from your previously created asset template.

Project*

```
mutation {
  MintAsset(
    assetId: "assetId"
    mints: { to: "<address here>", value: 1 }
    wallet: "<address here>"
    send: true
  ) {
    asset {
      name
      id
    }
    transactionId
    id
    value
    wallet {
      ethAddress
    }
  }
}
```

**Expected Output:**

```
{
  "data": {
    "MintAsset": {
      "asset": {
        "name": "AssetName",
        "id": "TokenId"
      },
      "transactionId": null,
      "id": 31237,
      "value": "1",
      "wallet": {
        "ethAddress": "<address here>"
      }
    }
  }
}
```

### CreatePlayer

`CreatePlayer` mutation allows you to create your own players for your game.

Project*

```
mutation {
  CreatePlayer(id: "BOB") {
    accessToken
  }
}
```

**Expected Output:**

```
{
  "data": {
    "CreatePlayer": {
      "accessToken": "accessToken"
    }
  }
}
```

### InvalidateAssetMetadata

`InvalidateAssetMetadata` mutation allows you to invalidate metadata on EnjinX/Wallet.

Project*

```
mutation {
  InvalidateAssetMetadata(id: "<tokenId>")
}
```

**Expected Output:**

```
{
  "data": {
    "InvalidateAssetMetadata": true
  }
}
```

### Unlink Wallet

`UnlinkWallet` mutation is used to unlink a wallet from a specific project.

```
mutation {
  UnlinkWallet(address: "<walletaddress>")
}
```

**Expected output:**

```
{
  "data": {
    "UnlinkWallet": true
  }
}
```

### Create Asset

`CreateAsset` This mutation is used to create assets on the blockchain.

```
mutation {
  CreateAsset(
    name: "Test"
    totalSupply: "1"
    initialReserve: "1"
    supplyModel: COLLAPSING
    meltValue: "1000000000000000000"
    meltFeeRatio: 0
    transferable: PERMANENT
    transferFeeSettings: {type:PER_TRANSFER, assetId: "0", value: "0" }
    nonFungible: true
    wallet: "WalletAddress"
  ) {
    value
    id
    transactionId
    title
    state
    project{name}
  }
}
```

**Expected Output:**

```
{
  "data": {
    "CreateAsset": {
      "value": "1",
      "id": 31179,
      "transactionId": null,
      "title": "Test",
      "state": "PENDING",
      "project": {
        "name": "Testing Project"
      }
    }
  }
}
```

### MintAsset

`MintAsset` Use this mutation to mint assets with the parameters from your previously created asset template.

```
mutation {
  MintAsset(
    assetId: "assetId"
    mints: { to: "WalletAddress", value: 1 }
    wallet: "WalletAddress"
    send: true
  ) {
    asset {
      name
      id
    }
    transactionId
    id
    value
    wallet {
      ethAddress
    }
  }
}
```

**Expected Output:**

```
{
  "data": {
    "MintAsset": {
      "asset": {
        "name": "Test 2",
        "id": "7080000000000088e"
      },
      "transactionId": null,
      "id": 31237,
      "value": "1",
      "wallet": {
        "ethAddress": "WalletAddress"
      }
    }
  }
}
```

### AdvancedSendAsset

`AdvancedSendAsset` Use this mutation to send assets from one wallet to another wallet.

```
```

```
mutation {
  AdvancedSendAsset(
    transfers: {
      from: "WalletAddress"
      to: "WalletAddress"
      assetId: "AssetId"
      assetIndex: "AssetIndex"
      value: "1"
    }
  ) {
    transactionId
    id
    state
    value
    asset{name id}
    user{name}
  }
}
```

**Expected Output:**


```
{
  "data": {
    "AdvancedSendAsset": {
      "transactionId": null,
      "id": 31369,
      "state": "PENDING",
      "value": "1",
      "asset": {
        "name": "New Test Asset",
        "id": "788000000000089f"
      },
      "user": null
    }
  }
}
```


### ApproveEnj

`ApproveEnj` use this mutation to approve a specific amount of ENJ to be spent.

```
mutation {
  ApproveEnj(
    value: "0"
    wallet: "WalletAddress"
  ) {
    type
    value
    state
  }
}
```

**Expected Output:**

```
{
  "data": {
    "ApproveEnj": {
      "type": "APPROVE",
      "value": "0"
    }
  }
}
```

### ApproveMaxEnj

`ApproveMaxEnj` use this mutation to set the spending allowance to the max value possible (infinite).

```
mutation {
  ApproveMaxEnj(
    wallet: "WalletAddress"
  ) {
    type
    value
    state
  }
}
```

**Expected Output:**

```
{
  "data": {
    "ApproveMaxEnj": {
      "type": "APPROVE",
      "value": "-1",
      "state": "PENDING"
    }
  }
}
```

### Message

`Message` Sign a message to prove wallet ownership.

```
mutation {
  Message(
    message:"This is a Test"
    wallet: "0x7AEAB7C231c88611a2ad434d3A2715Ab3C91F406"
  ) {
```

```
    type
    value
    state
  }
}
```

**Expected Output:**

```
{
  "data": {
    "Message": {
      "type": "MESSAGE",
      "value": "0",
      "state": "PENDING"
    }
  }
}
```

### Create Trade

`CreateTrade` use this mutation to trade assets (FTs or NFTs) and trade assets for ENJ as well.

```
mutation {
  CreateTrade(
    askingAssets: [{ assetId: "assetId", value: 1 }],
    offeringAssets: [{ assetId: "assetId", value: 1, assetIndex:"assetIndex"}],
    secondParty: "walletAddress"
    wallet: "walletAddress"
  )
 {
    transactionId
    id
    state
    value
    asset{name id}
    user{name}
  }
}
```

**Expected Output:**

```
{
  "data": {
    "CreateTrade": {
      "transactionId": null,
      "id": 160502,
      "state": "PENDING",
      "value": "0",
      "asset": null,
      "user": null
```

```
    }
  }
}
```

### Complete Trade

`CompleteTrade` use this mutation to complete the trade request initiated by the previous mutation, note that the tradeID is the transactionID from the previous request.

```
mutation {
  CompleteTrade(
    tradeId: "160502",
    wallet: "walletaddressgrta"
  )
 {
    transactionId
    id
    state
    value
    asset{name id}
    user{name}
  }
}
```

**Expected Output:**

```
{
  "data": {
    "CompleteTrade": {
      "transactionId": null,
      "id": 160503,
      "state": "PENDING",
      "value": "0",
      "asset": null,
      "user": null
    }
  }
}
```

### Decrease Max Melt Fee

`DecreaseMaxMeltFee` use this mutation to decrease the Max Melt Fee of an asset.

```
  mutation {
    DecreaseMaxMeltFee(
      assetId: "assetId"
      maxMeltFee: 2500,
      wallet: "walletAddress")
```

```
      {
      transactionId
      id
      state
      value
      asset{name id}
      user{name}
      }
  }
```

**Expected Output:**

```
{
  "data": {
    "DecreaseMaxMeltFee": {
      "transactionId": null,
      "id": 160508,
      "state": "PENDING",
      "value": "0",
      "asset": {
        "name": "Melt Fee Asset",
        "id": "assetId"
      },
      "user": null
    }
  }
}
```

### Decrease Max Transfer Fee

`DecreaseMaxTransferFee` use this mutation to decrease the Max Transfer Fee of an asset.

```
mutation {
  DecreaseMaxTransferFee(
    assetId: "assetId"
    maxTransferFee: "0"
    wallet: "walletAddress"
  ) {
    transactionId
    id
    state
    value
    asset {
      name
      id
    }
    user {
      name
    }
  }
}
```

**Expected Output:**


```
{
  "data": {
    "DecreaseMaxTransferFee": {
      "transactionId": null,
      "id": 160515,
      "state": "PENDING",
      "value": "0",
      "asset": {
        "name": "Transfer Fee Asset Test",
        "id": "assetId"
      },
      "user": null
    }
  }
}
```


### Melt Asset

`MeltAsset` use this mutation to melt an FT or NFT.

```
mutation {
  MeltAsset(
  melts:[{assetId:"assetId",value:"1"}]
    wallet:"walletAddress"
  ) {
    transactionId
    id
    state
    value
    asset {
      name
      id
    }
    user {
      name
    }
  }
}
```

**Expected Output:**


```
{
  "data": {
    "MeltAsset": {
      "transactionId": null,
      "id": 160522,
```

```
      "state": "PENDING",
      "value": "1",
      "asset": {
        "name": "Transfer Fee Asset Test",
        "id": "assetIdgra"
      },
      "user": null
    }
  }
}
```

### Release Reserve

`ReleaseReserve` use the mutation to release the reserve of assets that you didn't mint yet.

```
mutation {
 ReleaseReserve(
  assetId: "assetId"
  value: "3"
  wallet: "walletAddress"
  ) {
    transactionId
    id
    state
    value
    asset {
      name
      id
    }
    user {
      name
    }
  }
}
```

### Expected Output:

```
{
  "data": {
    "ReleaseReserve": {
      "transactionId": null,
      "id": 160532,
      "state": "PENDING",
      "value": "3",
      "asset": {
        "name": "new corecore",
        "id": "assetId"
      },
      "user": null
    }
```

```
  }
}
```

### Cancel Transaction

`CancelTransaction` use this mutation to cancel a transaction that was not approved yet. It's important to know that once a transaction has been broadcast to the blockchain, it wouldn't be possible to cancel the request.a

```
mutation {
CancelTransaction(id:requestId)}
```

**Expected Output:**

```
{
  "data": {
    "CancelTransaction": true
  }
}
```

### Send Enjin or JEnjin

`SendEnj` - This mutation can be used to send ENJ from address A to address B, the user involved in the request should be the one approving the send mutation.

```
mutation {
  SendEnj(
    to: "walletAddress"
    value: "100000000000000000"
    wallet: "walletAddressgra"
  ) {
    transactionId
    id
    state
    value
  }
}
```

**Expected Output:**

```
{
  "data": {
    "SendEnj": {
      "transactionId": null,
      "id": 164719,
      "state": "PENDING",
      "value": "0.1"
```

```
  } }
}
```

### Send Asset

`SendAsset` use this mutation to send FTs or NFTs.

```
mutation {
  SendAsset(
    assetId: "assetId"
    assetIndex: "assetIndex"
    to: "walletAddress"
    value: "1"
    wallet: "walletAddress"
  ) {
    transactionId
    id
    state
    value
    asset {
      name
      id
    }
    user {
      name
    }
  }
}
```

**Expected Output:**

```
{
  "data": {
    "SendAsset": {
      "transactionId": null,
      "id": 164727,
      "state": "PENDING",
      "value": "1",
      "asset": {
        "name": "NFT Test",
        "id": "78c00000000004b1"
      },
      "user": null
    }
  }
}
```

### SetApprovalForAll

`SetApprovalForAll` use this mutation to allow an operator complete control on all assets owned by the caller.

```
mutation {
  SetApprovalForAll(
    operator: "walletAddress"
    approved: true
    wallet: "walletAddress"
  ) {
    transactionId
    id
    state
    value
    asset {
      name
      id
    }
    user {
      name
    }
  }
}
```

**Expected Output:**

```
{
  "data": {
    "SetApprovalForAll": {
      "transactionId": null,
      "id": 164731,
      "state": "PENDING",
      "value": "0",
      "asset": null,
      "user": null
    }
  }
}
```

**SetWhitelisted**

`SetWhiteListed` use this mutation to set whitelisting parameters to a specific asset.

```
mutation {
  SetWhitelisted(
    assetId:"assetId"
    account:"walletAddres"
    whitelisted:SEND_AND_RECEIVE
    on:true
    wallet: "walletAddresg"
```

```
  ) {
    transactionId
    id
    state
    value
    asset {
      name
      id
    }
    user {
      name
    }
  }
}
```

**Expected output:**

```
{
  "data": {
    "SetWhitelisted": {
      "transactionId": null,
      "id": 164732,
      "state": "PENDING",
      "value": "0",
      "asset": {
        "name": "NFT Test",
        "id": "78c00000000004b1"
      },
      "user": null
    }
  }
}
```

### SetUri

`SetUri` use this mutation to set the metadata of an asset

```
mutation {
  SetUri(
    assetId: "assetId"
    uri:"your uri url here"
    wallet: "your wallet address"
  ) {
    transactionId
    id
    state
    value
    asset {
      name
      id
```

```
    }
    user {
      name
    }
  }
}
```

**Expected Output:**

```
{
  "data": {
    "SetUri": {
      "transactionId": null,
      "id": 98422,
      "state": "PENDING",
      "value": "0",
      "asset": {
        "name": "NFT Test",
        "id": "assetId"
      },
      "user": null
    }
  }
}
```

### SetMeltFee

`SetMeltFee` use this mutation to set the Melt Fee of an asset.

```
mutation {
  SetMeltFee(
    assetId: "assetId"
    meltFee: 500
    wallet: "walletAddress"
  ) {
    transactionId
    id
    state
    value
    asset {
      name
      id
    }
    user {
      name
    }
  }
}
```

**Expected output:**


```
{
  "data": {
    "SetMeltFee": {
      "transactionId": null,

      "id": 181704,
      "state": "PENDING",
      "value": "0",
      "asset": {
        "name": "Melt Fee NFT",
        "id": "38c0000000000516"
      },
      "user": null
    }
  }
}
```


### SetTransferFee

`SetTransferFee` use this mutation to set a transfer fee with an asset.

```
mutation {
  SetTransferFee(
    assetId: "assetId"
    transferFee: 1000000000000000
    wallet: "walletAddress"
  ) {
    transactionId
    id
    state
    value
    asset {
      name
      id
    }
    user {
      name
    }
  }
}
```

**Expected Output:**


```
{
  "data": {
    "SetTransferFee": {
      "transactionId": null,
      "id": 181705,
      "state": "PENDING",
```

```
      "value": "0",
      "asset": {
        "name": "Transfer Fee NFT",
        "id": "assetId"
      },
      "user": null

    }
  }
}
```

### SetTransferable

`SetTransferable` Use this mutation to change the transferable status of an asset.

```
mutation {
  SetTransferable(
    assetId: "assetIdgra"
    transferable: BOUND
    wallet: "0x7AEAB7C231c88611a2ad434d3A2715Ab3C91F406"
  ) {
    transactionId
    id
    state
    value
    asset {
      name
      id
    }
    user {
      name
    }
  }
}
```

### Expected Output:

```
{
  "data": {
    "SetTransferable": {
      "transactionId": null,
      "id": 181709,
      "state": "PENDING",
      "value": "0",
      "asset": {
        "name": "Test",
        "id": "assetId"
      },
      "user": null
    }
  }
```

```
}
```

# EnjinX API

Learn more about EnjinX

## What is the EnjinX API?

The EnjinX API is a scalable blockchain development and data infrastructure service for companies building blockchain products, such as apps, games, wallets, marketplaces, and exchanges.

With the EnjinX API, users are able to tap into a wealth of data available on the Ethereum & Jumpnet chains, such as:

- Transaction history
- Validate transactions
- Balances (for ETH and ERC-20 tokens)
- ERC-721 & ERC-1155 assets

All of this data can be paginated and sorted.

> ⓘ GraphiQL EnjinX API link - https://api.enjinx.io/graphql

> ⓘ For further reference on the EnjinX API, please refer to the docs here - https://api-docs.enjinx.io/#intro

---

## Queries

### Fetch Listing

`MarketplaceListing` This query can be used to access the data of a marketplace listing.

The `uuid` variable is that unique identifier of the marketplace listing. You can retrieve this from the `uuid` property on the marketplace listing.

```
query MarketplaceListing($uuid: String!) {
  MarketplaceListing(uuid: $uuid) {
```

```
    asset {
      metadata {
        name
        image
        description
      }
      marketplaceData {
        lastSoldPrice
        lastSoldAt

        currentListings
        lowestPrice
        highestPrice
      }
    }
    remaining
    price
    seller {
      address
    }
    listedAt
  }
}
```

### Example Response


```
{
  "data": {
    "MarketplaceListing": {
      "asset": {
        "metadata": {
          "name": "HAPPY NEW YEAR 2020",
          "image": "https://cdn.enjinx.io/metadata/raw/d26b77650f01f685d05a81a26626c00e7d7e5b
          "description": "Lunar New Year 2020 Special Edition"
        },
        "marketplaceData": {
          "lastSoldPrice": null,
          "lastSoldAt": null,
          "currentListings": 1,
          "lowestPrice": null,
          "highestPrice": null
        }
      },
      "remaining": 1,
      "price": 0.005,
      "seller": {
        "address": "0xc42709c680799ca52851692037326ec8c0019da3"
      },
      "listedAt": "2020-05-16T23:54:46+00:00"
    }
  }
}
```

### Fetch All Listings

`MarketplaceListing` This query can be used to access all marketplace listings.

The `page` variable (*optional, default* = `1`) is the current cursor position of the query.

The `limit` variable (*optional, default* = `25`) is the maximum number of marketplace listings to return per page.

```
query MarketplaceListings($page: Int, $limit: Int) {
  MarketplaceListings(limit: $limit, page: $page) {
    data {
      uuid
      asset {
        metadata {
          name
          image
          description
        }
        marketplaceData {
          lastSoldPrice
          lastSoldAt
          currentListings
          lowestPrice
          highestPrice
        }
      }
      remaining
      price
      seller {
        address
      }
      listedAt
    }
    total
    currentPage
    lastPage
  }
}
```

### Example Response

```
{
  "data": {
    "MarketplaceListings": {
      "data": [
        {
          "uuid": "0f090ee7-1d69-4665-a127-65f0c637ad44",
          "asset": {
            "metadata": {
```

```
            "name": "HAPPY NEW YEAR 2020",
            "image": "https://cdn.enjinx.io/metadata/raw/d26b77650f01f685d05a81a26626c00e7d
            "description": "Lunar New Year 2020 Special Edition"
          },
          "marketplaceData": {
            "lastSoldPrice": null,
            "lastSoldAt": null,
            "currentListings": 1,
            "lowestPrice": null,
            "highestPrice": null
          }
        },
        "remaining": 1,
        "price": 0.005,
        "seller": {
          "address": "0xc42709c680799ca52851692037326ec8c0019da3"
        },
        "listedAt": "2020-05-16T23:54:46+00:00"
      }
```


### Fetch All Listings By Address

`AddressMarketPlaceListings` This query can be used to access all marketplace listings for a given seller.

The `address` variable is the wallet address of the seller (eg. `0x65ffe5a603b9dac9bca330bf387979701374c96c`).

The `page` variable (*optional, default* = `1`) is the current cursor position of the query.

The `limit` variable (*optional, default* = `25`) is the maximum number of marketplace listings to return per page.

```
query AddressMarketplaceListings($address: String!, $page: Int, $limit: Int) {
  Address(address: $address) {
    marketplaceListings(limit: $limit, page: $page) {
      data {
        uuid
        asset {
          metadata {
            name
            image
            description
          }
          marketplaceData {
            lastSoldPrice
            lastSoldAt
            currentListings
            lowestPrice
            highestPrice
          }
        }
```

```
        remaining
        price
        listedAt
      }
      total
      currentPage
      lastPage
    }
  }
}
```

### Example Response


```
{
  "data": {
    "Address": {
      "marketplaceListings": {
        "data": [
          {
            "uuid": "cade506d-3187-4065-aa4d-96d803974d69",
            "asset": {
              "metadata": {
                "name": "Major Tom",
                "image": "https://cdn.enjinx.io/metadata/raw/bad1bffe29f039d102382d02da6f0d0a
                "description": "The fastest astronaut in the crypto universe from the Changel
              },
              "marketplaceData": {
                "lastSoldPrice": 6,
                "lastSoldAt": "2020-07-09T10:31:56+00:00",
                "currentListings": 64,
                "lowestPrice": 2.5,
                "highestPrice": 25
              }
            },
            "remaining": 1,
            "price": 666,
            "listedAt": "2020-12-08T22:44:15+00:00"
          }
        ],
        "total": 1,
        "currentPage": 1,
        "lastPage": 1
      }
    }
  }
}
```


### Fetch All Listings By Asset

`AssetMarketPlaceListings` This query can be used to access all marketplace listings for a given

asset

The `id` variable is the on-chain unique identifier for the asset (eg. `5080000000000027`).

The `index` (*optional*) is the on-chain unique index for a specific (non-fungible) asset represented as a decimal (eg. `1`).

The `page` variable (*optional, default* = `1`) is the current cursor position of the query.

The `limit` variable (*optional, default* = `25`) is the maximum number of marketplace listings to return per page.

```
query AssetMarketplaceListings($id: String!, $index: Int, $page: Int, $limit: Int) {
  Asset(id: $id, index: $index) {
    metadata {
      name
      image
      description
    }
    marketplaceListings(limit: $limit, page: $page) {
      data {
        uuid
        remaining
        price
        seller {
          address
        }
        listedAt
      }
      total
      currentPage
      lastPage
    }
    marketplaceData {
      lastSoldPrice
      lastSoldAt
      currentListings
      lowestPrice
      highestPrice
    }
  }
}
```

### Example Response


```
{
  "data": {
    "Asset": {
      "metadata": {
        "name": "Oindrasdain",
        "image": "https://cdn.enjinx.io/metadata/raw/a41e1c9a773f11f078a2b479afda8216fb41e8b0/
        "description": "Oindrasdain is a fearsome and powerful weapon.\nAccording to the Saga
```

```
    },
    "marketplaceListings": {
      "data": [
        {
          "uuid": "0da80805-ff1e-4e37-99b2-aac784629594",
          "remaining": 1,
          "price": 27.5,
          "seller": {
            "address": "0x819d8b7b854ba4d877b9d2557f09dc1c18a3f6d1"
          },
          "listedAt": "2019-10-30T06:08:33+00:00"
        },
        {
          "uuid": "7a7eeb1d-281a-4072-a441-51e9cd470cab",
          "remaining": 1,
          "price": 28,
          "seller": {
            "address": "0x819d8b7b854ba4d877b9d2557f09dc1c18a3f6d1"
          },
          "listedAt": "2019-11-14T17:31:53+00:00"
        }
```


### Fetch All Listings By Project

`ProjectMarketplaceListings` **This query can be used to access all marketplace listings for a given project.**

The `uuid` variable is the unique identifier of the project you're querying for.

The `page` variable (*optional, default =* `1`) is the current cursor position of the query.

The `limit` variable (*optional, default =* `25`) is the maximum number of marketplace listings to return per page.

```
query ProjectMarketplaceListings($uuid: String!, $page: Int, $limit: Int) {
  Project(uuid: $uuid) {
    marketplaceListings(limit: $limit, page: $page) {
      data {
        uuid
        asset {
          metadata {
            name
            image
            description
          }
          marketplaceData {
            lastSoldPrice
            lastSoldAt
            currentListings
            lowestPrice
            highestPrice
          }
```

```
        }
        remaining
        price
        seller {
          address
        }
        listedAt
      }
      total
      currentPage
      lastPage
    }
  }
}
```

### Example Response


```
{
  "data": {
    "Project": {
      "marketplaceListings": {
        "data": [
          {
            "uuid": "c93c65c6-a12b-4f47-b920-30f33e8da5e7",
            "asset": {
              "metadata": {
                "name": "FIO Express",
                "image": "https://cdn.enjinx.io/metadata/raw/7192bda941a7bc6e2f3eac4c66285eef
                "description": "The FIO Express is an elite troop of servicemen whose mission
              },
              "marketplaceData": {
                "lastSoldPrice": 3.4,
                "lastSoldAt": "2020-05-14T18:08:15+00:00",
                "currentListings": 43,
                "lowestPrice": 1,
                "highestPrice": 9
              }
            },
            "remaining": 1,
            "price": 3.9,
            "seller": {
              "address": "0xecadbaf1fce4db184680169bfa184c59ecdbdf9b"
            },
            "listedAt": "2020-04-11T16:13:56+00:00"
          },
          {
            "uuid": "fbb73af8-33b0-4518-bfa6-ee5fb88fcfad",
            "asset": {
              "metadata": {
                "name": "FIO Express",
                "image": "https://cdn.enjinx.io/metadata/raw/7192bda941a7bc6e2f3eac4c66285eef
                "description": "The FIO Express is an elite troop of servicemen whose mission
```

```
      },
      "marketplaceData": {
        "lastSoldPrice": 3.4,
        "lastSoldAt": "2020-05-14T18:08:15+00:00",
        "currentListings": 43,
        "lowestPrice": 1,
        "highestPrice": 9
      }
    }
```

#### Fetch Project UUID By Asset

`ProjectUuidByAsset` This query can be used to retrieve the `uuid` of a project through an asset that belongs to the project.

The `id` variable is the on-chain unique identifier for the asset (eg. `5080000000000027`).

```
query ProjectUuidByAsset($id: String!) {
  Asset(id: $id) {
    project {
        uuid
    }
  }
}
```

#### Example Response

```
{
  "data": {
    "Asset": {
      "project": {
        "uuid": "3223b8fe-0693-4f5d-abed-8d73867ff824"
      }
    }
  }
}
```

---

## Mutations

#### Purchase Listing

`PurchaseListing` This mutation can be used to purchase an asset from the marketplace.

The `uuid` variable is that unique identifier of the marketplace listing. You can retrieve this from the `uuid` property on the marketplace listing.

The `quantity` variable (*optional, default* = `1`) is the number of assets to purchase from this listing. This number cannot exceed the `remaining` property of the marketplace listing.

```
mutation PurchaseListing($uuid: String!, $quantity: Int) {
  MarketplacePurchaseListing(uuid: $uuid, quantity: $quantity) {
    uuid
    link
    qr
  }
}
```

**Example Response**

```
{
  "data": {
    "MarketplacePurchaseListing": {
      "uuid": "944ffb26-d4a3-4e0c-956a-c90ef11945d0",
      "link": "enjinwallet://marketplace/944ffb26-d4a3-4e0c-956a-c90ef11945d0",
      "qr": "https://enjinx.io/qrcode/marketplace:944ffb26-d4a3-4e0c-956a-c90ef11945d0"
    }
  }
}
```

# Errors and Statuses

## Understanding Errors in the Enjin Platform API (Work In Progress)

Learn more about common errors in the Enjin API and how to handle it.

The purpose of this section is to give an example of some of the most common errors in the Enjin Platform API and how to handle it.

> In case the error you encounter is not listed below, we would recommend getting in touch with our support team at - enj.in/support.

# Wallet Daemon

# Enjin Wallet Daemon Overview

**Enjin Wallet Daemon Overview**

The Wallet Daemon is a tool that you can use to automate the authorization of transaction requests to and from the Enjin Platform.

Without the Wallet Daemon, you would need to sign (authorize) every in-game blockchain transaction via the Enjin Wallet (e.g., sending a sword to a player).

Wallet Daemon manages an Ethereum address linked to an Enjin Platform user. When a transaction is submitted on the Enjin Platform, the Wallet Daemon receives that specific request, signs it, and sends it back to the Enjin Platform.

Currently, only a console version of the Wallet Daemon is available. In the future, we will likely create a simple graphical user interface to make it easier for you to use.

**Resources**

Getting started with the Enjin Wallet Daemon

Enjin Wallet Daemon First Steps

Enjin Wallet Daemon Usage

> ⓘ In case you encounter any errors while setting the Wallet Daemon up, please contact our support team at enj.in/discord.

# Getting Started with the Enjin Wallet Daemon

Learn how to install the Wallet Daemon

The wallet daemon allows you to automate the transaction signing process, so all of your blockchain transactions are actioned instantly, creating a persistent bridge between your game and the blockchain, and ensuring your players can enjoy a seamless and fluid gaming experience.

**Wallet Daemon Installation**

Here are the instructions to install the Enjin Wallet Daemon application under various OSes. This document

Administrator access to the target computer is required.

> ⓘ Please be aware that using a wallet daemon will mean that ANY transaction that is generated via your App Secret will be signed automatically. Please ensure there is no way for any unauthorized party to access your App Secret and process unapproved transactions.

### Windows 10

#### Requirements Summary

- node.js
- Python 2
- Visual Studio Build Tools
- git

See https://github.com/nodejs/node-gyp/#on-windows

#### Install node.js

```
npm install --global --production windows-build-tools
```

Install git from the official website https://git-scm.com/download/win

#### Install Wallet Daemon

Get the zip for the wallet daemon HERE.

Right-click enjin-wallet-daemon-master.zip and select "*Extract All...*" Then, in a command line (replace <CODE_FOLDER> with whatever folder you extracted the archive to):

```
cd <CODE_FOLDER>\enjin-wallet-daemon-master
npm install
```

### MacOS

#### Requirements Summary

- macOS command line developer tools
- node.js

#### Install homebrew

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/insta
```

#### Install node.js

```
brew install node
```

#### Install Wallet Daemon

```
unzip enjin-wallet-daemon-master.zip
cd enjin-wallet-daemon-master
npm install
```

### Linux (Debian 10 *Buster*)

#### Requirements Summary

- build tools (gcc, make, etc.) build-essential
- git
- node.js
- unzip

#### Install node.js

```
su -
apt install build-essential curl git unzip
curl -sL https://deb.nodesource.com/setup_11.x | bash -
apt install nodejs
```

#### Install Wallet Daemon

```
unzip enjin-wallet-daemon-master.zip
cd enjin-wallet-daemon-master
npm install
```

### Linux (Ubuntu Server 18.04 LTS)

#### Requirements Summary

- build tools (gcc, make, etc.) build-essential
- node.js
- unzip

#### Install node.js

```
unzip enjin-wallet-daemon-master.zip
cd enjin-wallet-daemon-master
npm install
```

#### All Platforms

#### Run Wallet Daemon (New Wallet)

Open an account on the **Enjin Platform** and get a link code (e.g. XY1ABC). Replace the placeholder bellow with that code.

```
node src/main.js account new
node src/main.js link <LINK_CODE>
node src/main.js
```

## Enjin Wallet Daemon First Steps

Learn how to get started with the Enjin Wallet Daemon.

When a transaction is submitted on the Enjin Platform, the wallet daemon receives that transaction, signs it, and sends it back to the Trusted Cloud.

#### Enjin Wallet Daemon Configuration

Before initializing the Wallet Daemon, open your enjin-wallet-daemon.1.x.x-beta folder and search for the example-config.json file, the contents of this file features some basic configuration parameters.

Copy the file from example-config.json to config.json and then open it in a text editor.

```
{
   "salt": "193e9997-5a10-4d9e-a829-69ddcf6cbf70",
   "keyIterations": 1000,
   "chain": "goerli",
   "enjinxEndpoint": "https://daemon.api.enjinx.io/",
   "minGasPrice": 1000000000,
   "maxGasPrice": 21000000000
}
```

"`salt": "193e9997-5a10-4d9e-a829-69ddcf6cbf70`" - This paramater is used when encrypting/decrypting your daemon password. For improved security, you can set the salt to anything else (such as a different UUID4 string).

`"chain"`: `"goerli"`, `"mainnet"` , `"jumpnet"` - This parameter defines which network you are going to be running your wallet daemon on.

`"minGasPrice"` and `"maxGasPrice"` - Both parameters refer to ETH gas prices and can be used to constrain the minimum and maximum amount of gas to be used when signing a transaction

**New Wallet Creation**

This is the recommended way to initialize a Wallet Daemon.

Run node src/main.js account new
Notice the new wallet address is printed on the console.

**Keep a backup of your password somewhere safe! Otherwise, there is no way to recover your account.**

**Import Existing Private Key**

**From Enjin Wallet**

The Enjin Wallet uses the Ledger (ETH) *HD derivation path* (`m/44'/60'/0'`). You can use MyEtherWallet to rebuild your private key from the 12-word recovery phrase.

This method is *rather insecure* and should only be used knowing that it exposes your private key to a website that could have been compromised. To mitigate the risk, it's highly recommended to deploy a private copy of MyEtherWallet from https://github.com/kvhnuke/etherwallet/releases

If you decide to go this route, click on "View Wallet Info" and follow the onscreen instructions:

- Mnemonic Phrase
- Pasting your 12 words and keeping the password field empty
- Selecting the *Ledger (ETH)* derivation path
- Choosing the correct address from the suggestion list

**From MetaMask**

Go into account details and select "Export Private Key"

**From Parity/Geth**

Assuming your client installation uses the default data folders, the keys are stored there:

**Parity**

- Windows: `%HOMEPATH%/AppData/Roaming/Parity/Ethereum/keys`
- macOS: `~/Library/Application\ Support/io.parity.ethereum/keys`
- Linux: `$HOME/.local/share/io.parity.ethereum/keys`

**Geth**

- Windows: `%APPDATA%\Ethereum\keystore`
- macOS: `~/Library/Ethereum/keystore`
-

    Linux: `~/.ethereum/keystore`

Each key is stored in an extensionless json file. Here again, you can use "MyEtherWallet" to extract the private keys.

This method is *rather insecure* and should only be used knowing that it exposes your private key to a website that could have been compromised. To mitigate the risk, it's highly recommended to deploy a private copy of MyEtherWallet from https://github.com/kvhnuke/etherwallet/releases

If you decide to go this route, click on "View Wallet Info" and follow the onscreen instructions:

- Keystore / JSON File
- Select your file, enter your password

Link To the Enjin Platform

1. Create an account on the Enjin Platform.
    1. Mainnet: cloud.enjin.io
    2. JumpNet: jumpnet.cloud.enjin.io
    3. Goerli: goerli.cloud.enjin.io
2. Create an identity for the application you want to control with the wallet daemon.
3. Copy the linking code from the identity `<CODE>`.
4. Run node src/main.js link `<CODE>`

Run The Wallet Daemon

Run `node src/main.js`.

## Enjin Wallet Daemon Usage

#### Storage

The Wallet Daemon storage file is located in the following folder:

- Windows: `%LOCALAPPDATA%\enjin-wallet-daemon\storage.json`
- macOS: `$HOME\enjin-wallet-daemon\storage.json`
- Linux: `$HOME\enjin-wallet-daemon\storage.json`

#### Logs

All Wallet Daemon activity is logged in the following files:

- Windows: `%LOCALAPPDATA%\enjin-wallet-daemon\enjin-wallet-daemon-`

`<DATE>.json`

- macOS: `$HOME\enjin-wallet-daemon\enjin-wallet-daemon-<DATE>.json`
- Linux: `$HOME\enjin-wallet-daemon\enjin-wallet-daemon-<DATE>.json`

#### Command-List

- `account new`
- `account import <PRIVATE KEY>`
- `link <CODE>`
- `run`
- `backup <FILENAME>`
- `decrypt`
- `decrypt <FILENAME>`

#### New Account

Create a new account for this wallet. The wallet *must* not already have an address. If it does, this command will do nothing and you will get an error.

If you want to create a new wallet address, you have to move or rename the storage file first.

**Example:**

```
node src/main.js account new
```

#### Import Wallet

This command takes a single argument: the private key to use with the wallet. The format of the private key must be a 160-bit hexadecimal string prefixed with 0x.

**Example:**

```
node src/main.js account import 0x37986485ee024917a6cb7748c60d5d58214c7ca6e9a1d5d3880e2d94983
```

#### Link

Connect the wallet daemon with your Enjin Platform identity. You can link with many different identities.

**Example:**

```
node src/main.js link B3DJF4
```

**Run**

Start the wallet daemon. This is the default command if none is supplied. The application will keep running until it's stopped (Ctrl-C) or killed.

**Example:**

```
node src/main.js run
```

**Backup**

Copy the current storage file to the specified location. The file remains safely encrypted.

**Example:**

```
node src/main.js backup C:\Users\User\enjin-wallet-daemon-backup.json
```

**Decrypt**

Read an encrypted storage file and output its content to the console. If no storage file is specified, the default storage is read.

**Example(s):**

```
node src/main.js decrypt C:\Users\User\enjin-wallet-daemon-backup.json
node src/main.js decrypt
```

# Enjin Platform SDKs

## Introduction to Enjin Platform SDKs

Learn how to get started with the Enjin SDKs

The Enjin Platform SDKs provide a framework that developers can use on their desired programming language as it allows games and apps to interact with the Enjin API without the need of building a bridge

between your application and our API requests, the Enjin SDKs facilitate the usage of listening and processing of events emitted by the Enjin Cloud, which helps us deliver a better user-experience in our application and eliminate unnecessary API requests we would otherwise make to the Enjin Platform.

### V.2 Schemas Documentation

The documentation for the Enjin SDKs pertains to the Project and Player schemas which are part of our V.2 API GraphQL Schemas. To enable The Project and Player schema, please refer to this section in our documentation - Enabling V.2 Schemas.

### SDK Libraries

| Library | Link |
| --- | --- |
| Java | enjin/enjin-java-sdk |
| C# | enjin/enjin-csharp-sdk |
| C++ | enjin/enjin-cpp-sdk |
| Unity | enjin/enjin-unity-sdk |
| Unreal Engine | enjin/enjin-unreal-sdk |

## Getting Started with the Enjin Platform SDKs

## Authenticating a Project Client

### Creating the Client

The first step we take in setting up our `ProjectClient` is to instantiate it. To do so we must utilize a builder class nested within the client.

Java

```java
import com.enjin.sdk.ProjectClient;
import com.enjin.sdk.ProjectClient.ProjectClientBuilder;

ProjectClientBuilder builder = ProjectClient.builder();
```

C# | Unity

```
using static Enjin.SDK.ProjectClient;

ProjectClientBuilder builder = Builder();
```

C++

```
#include "enjinsdk/ProjectClient.hpp"

using namespace enjin::sdk;

ProjectClient::ProjectClientBuilder builder = ProjectClient::builder();
```

Unreal

```
#include "ProjectClient.h"

using namespace Enjin::Sdk;

FProjectClient::FProjectClientBuilder Builder = FProjectClient::Builder();
```

Once we have the builder we may call its methods to configure it for the client we would like to create. One configuration we must set is the network our project exists on. Our choices are Mainnet, JumpNet, and Goerli. URLs for these networks can be programmatically acquired from the `EnjinHosts` class and passed to the builder.

Java

```
import com.enjin.sdk.EnjinHosts;

String mainnet = EnjinHosts.MAIN_NET;
String jumpnet = EnjinHosts.JUMP_NET;
String goerli = EnjinHosts.GOERLI;

builder.baseUri(/* Enjin host here */);
```

C# | Unity

```
using Enjin.SDK;

System.Uri mainnet = EnjinHosts.MAIN_NET;
System.Uri jumpnet = EnjinHosts.JUMP_NET;
System.Uri goerli = EnjinHosts.GOERLI;

builder.BaseUri(/* Enjin host here */);
```

C++

```
#include "enjinsdk/EnjinHosts.hpp"
#include <string>

using namespace enjin::sdk;

std::string mainnet = EnjinHosts::MainNet;
std::string jumpnet = EnjinHosts::JumpNet;
std::string goerli = EnjinHosts::Goerli;

builder.base_uri(/* Enjin host here */);
```

Unreal

```
#include "EnjinHosts.h"

using namespace Enjin::Sdk;

FString MainNet = FEnjinHosts::MainNet();
FString JumpNet= FEnjinHosts::JumpNet();
FString Goerli= FEnjinHosts::Goerli();

Builder.BaseUrl(/* Enjin host here */);
```

After we finish configuring the builder, we then create the client by calling the builder's build method.

Java

```
import com.enjin.sdk.ProjectClient;

ProjectClient client = builder.build();
```

C# | Unity

```
using Enjin.SDK;

ProjectClient client = builder.Build();
```

C++

```
#include "enjinsdk/ProjectClient.hpp"
#include <memory>

using namespace enjin::sdk;

std::unique_ptr<ProjectClient> client = builder.build();
```

Unreal

```
#include "ProjectClient.h"

using namespace Enjin::Sdk;

TUniquePtr<FProjectClient> Client = Builder.Build();
```

The configuration methods return the builder itself to allow for chaining calls to other methods.

### Sending the Authentication Request

To authenticate our client, we will need our project's UUID and secret key. These may be found by navigating to the project page on the Enjin platform and found under `Settings` -> `API Credentials`.

You should make sure that you're only authenticating as a project in a **secure environment**. The project's secret enables a **large degree of access** over the project and is intended solely to be ran in an environment managed by the developer. It must not be exposed to others.

Once we have retrieved the UUID and secret, we may go about authenticating our client. For a project client we may either create the request ourselves to send to the platform or we may use the built-in convenience method to authenticate as shown directly below in the example.

Java

```java
String uuid = "<the-project's-uuid>";
String secret = "<the-project's-secret>";

client.authClient(uuid, secret).get();
```

C# | Unity

```csharp
string uuid = "<the-project's-uuid>";
string secret = "<the-project's-secret>";

client.AuthClient(uuid, secret).Wait();
```

C++

```cpp
#include <string>

std::string uuid = "<the-project's-uuid>";
std::string secret = "<the-project's-secret>";

client->auth_client(uuid, secret).wait();
```

Unreal

```
FString Uuid = TEXT("<the-project's-uuid>");
FString Secret = TEXT("<the-project's-secret>");

Client->AuthClient(Uuid, Secret);
```

### Creating the Authentication Request

Before creating the `AuthProject` request, we must first gather our project's UUID and secret key. To do so, we may navigate to our project page on the Enjin platform and find these items under `Settings` -> `API Credentials`.

Once the UUID and secret have been acquired, we can pass them to the `AuthProject` request we have created.

Java

```java
import com.enjin.sdk.schemas.project.queries.AuthProject;

AuthProject req = new AuthProject()
    .uuid("<the-project's-uuid>")
    .secret("<the-project's-secret>");
```

C# | Unity

```csharp
using Enjin.SDK.ProjectSchema;

AuthProject req = new AuthProject()
    .Uuid("<the-project's-uuid>")
    .Secret("<the-project's-secret>");
```

C++

```cpp
#include "enjinsdk/project/AuthProject.hpp"

using namespace enjin::sdk::project;

AuthProject req = AuthProject()
    .set_uuid("<the-project's-uuid>")
    .set_secret("<the-project's-secret>");
```

Unreal

```unreal
#include "Project/AuthProject.h"

using namespace Enjin::Sdk::Project;

FAuthProject Req = FAuthProject()
    .SetUuid(TEXT("<the-project's-uuid>"))
    .SetSecret(TEXT("<the-project's-secret>"));
```

### Sending the Authentication Request

To send any request to the platform, we must call the method in our client whose name matches the request. The result of these methods in a synchronous operation will be a GraphQL response that wraps the data we want. An example of what this looks like for the `AuthProject` request is shown below:

Java

```java
import com.enjin.sdk.graphql.GraphQLResponse;
import com.enjin.sdk.models.AccessToken;

GraphQLResponse<AccessToken> res = client.authProject(req).get();
```

C# | Unity

```csharp
using Enjin.SDK.Graphql;
using Enjin.SDK.Models;

GraphqlResponse<AccessToken> res = client.AuthProject(req).Result;
```

C++

```cpp
#include "enjinsdk/GraphqlResponse.hpp"
#include "enjinsdk/Models/AccessToken.hpp"

using namespace enjin::sdk::graphql;

GraphqlResponse<AccessToken> res = client->auth_project(req).get();
```

Unreal

```cpp
#include "GraphQlResponse.h"
#include "Model/AccessToken.h"

using namespace Enjin::Sdk::GraphQl;
using namespace Enjin::Sdk::Model;

Client->AuthProject(Req)
```

```
{   .Next([this](TGraphQlResponseForOnePtr<FAccessToken> Res)
    // Validate and authenticate
});
```

With the GraphQL response from the platform we can get the data contained within it. For authentication requests this will be a `AccessToken` class representing the platform's `Auth` type. Before retrieving the data however, we may want to consider checking if the response was successful. To do so, we can use the method shown below and get its result:

Java

```
boolean result = res.isSuccess();
```

C# | Unity

```
bool result = res.IsSuccess;
```

C++

```
bool result = res.is_successful();
```

Unreal

```
bool bResult = Res.IsValid() && Res->IsSuccessful();
```

Now that we have determined that the response was successful we can retrieve the `AccessToken` model from the response. This may be done with the code shown below:

Java

```
import com.enjin.sdk.models.AccessToken;

AccessToken accessToken = res.getData();
```

C# | Unity

```
using Enjin.SDK.Models;

AccessToken accessToken = res.Result;
```

C++

```
#include "enjinsdk/models/AccessToken.hpp"

using namespace enjin::sdk::models;

AccessToken access_token = res.get_result().value();
```

Unreal

```
#include "Model/AccessToken.h"

using namespace Enjin::Sdk::Model;

FAccessToken AccessToken = Res->GetResult().GetValue();
```

### Authenticating

With the `AccessToken` we can now pass the string token contained within to the client's authentication method.

Java

```
client.auth(accessToken.getToken());
```

C# | Unity

```
client.Auth(accessToken.Token);
```

C++

```
client->auth(access_token.get_token().value());
```

Unreal

```
Client->Auth(AccessToken.GetToken().GetValue());
```

After authenticating, we may check to see if the authentication was successful. We can do so with the method shown below and getting the Boolean value returned.

Java

```
boolean result = client.isAuthenticated();
```

C# | Unity

```
bool result = client.IsAuthenticated;
```

C++

```
bool result = client->is_authenticated();
```

Unreal

```
bool bResult = Client->IsAuthenticated();
```

With our client successfully authenticated we can now use it to make further requests to the platform. One point of note though is that our client's authentication with the platform will eventually expire. The time in

seconds until the authentication expires can retrieved from the `AccessToken` as shown below:

Java

```java
long time = accessToken.getExpiresIn();
```

C# | Unity

```csharp
long time = accessToken.ExpiresIn.Value;
```

C++

```cpp
long time = access_token.get_expires_in().value();
```

Unreal

```
int64 Time = AccessToken.GetExpiresIn().GetValue();
```

### Automatic Reauthentication

The `ProjectClient` class may be setup to automatically reauthenticate itself before its access token expires. To do so we must first return to our client builder and make a call to its method which enables automatic reauthentication.

Java

```java
builder.enableAutomaticReauthentication();
```

C# | Unity

```csharp
builder.EnableAutomaticReauthentication();
```

C++

```
builder.enable_automatic_reauthentication();
```

Unreal

```
Builder.EnableAutomaticReauthentication();
```

After doing so we will then make a call to the client's authentication method using our project's UUID and secret key. This will allow our client to make the initial authentication request as well as cache the UUID and secret key that it will need to perform reauthentication on its own.

Java

```
String uuid = "<the-project's-uuid>";
String secret = "<the-project's-secret>";

client.authClient(uuid, secret).get();
```

C# | Unity

```
string uuid = "<the-project's-uuid>";
string secret = "<the-project's-secret>";

client.AuthClient(uuid, secret).Wait();
```

C++

```
#include <string>

std::string uuid = "<the-project's-uuid>";
std::string secret = "<the-project's-secret>";

client->auth_client(uuid, secret).wait();
```

Unreal

```
FString Uuid = TEXT("<the-project's-uuid>");
FString Secret = TEXT("<the-project's-secret>");

Client->AuthClient(Uuid, Secret);
```

In cases where automatic reauthentication is interrupted we may interface with our client to receive an event warning us that the process handling reauthentication has stopped. Examples for how to do so for each SDK are shown below:

Java

```
import com.enjin.sdk.IAuthenticationEventListener;

// As a configuration on the client builder
builder.authenticationListener(new IAuthenticationEventListener() {
    @Override
    public void onAutomaticReauthenticationStopped() {
        // Handle the event
    }
});
```

C# | Unity

```
// As a event on the client
client.OnAutomaticReauthenticationStopped += (sender, args) =>
{
    // Handle the event
};
```

C++

```
// As a configuration on the client builder
builder.reauthentication_stopped_handler([]() {
    // Handle the event
});
```

Unreal

```
// As a configuration on the client builder
Builder.ReauthenticationStoppedHandler([]()
{
    // Handle the event
});
```

---

## Authenticating a Player Client

To authenticate a player client we must first instantiate and authenticate a project client, which has access to the schemas we need. Read over the Authenticate a Project Client section for steps on setting up a project client ready to send requests to the platform.

### Getting the Access Token

Currently, there are two ways to get the access token for authenticating a player client. One such way is through the `CreatePlayer` request, which is for new players that do not yet exist for our project. The other way to acquire a player's access token is via the `AuthPlayer` request, which is used for existing players.

#### Creating a New Player

When using the `CreatePlayer` request, we provide the ID of the new player we wish to add to our project as shown below:

Java

```
import com.enjin.sdk.schemas.project.mutations.CreatePlayer;

CreatePlayer req = new CreatePlayer()
    .id("<the-player's-id>");
```

C# | Unity

```
using Enjin.SDK.ProjectSchema;

CreatePlayer req = new CreatePlayer()
    .Id("<the-player's-id>");
```

```
// Using a authenticated ProjectClient
GraphqlResponse<AccessToken> res = client.CreatePlayer(req).Result;
```

C++

```
#include "enjinsdk/project/CreatePlayer.hpp"

using namespace enjin::sdk::project;

CreatePlayer req = CreatePlayer()
    .set_id("<the-player's-id>");
```

Unreal

```
#include "Project/CreatePlayer"

using namespace Enjin::Sdk::Project;

FCreatePlayer Req = FCreatePlayer()
    .SetId(TEXT("<the-player's-id>"));
```

With the request in hand, we can now send the `CreatePlayer` request to the platform and if successful we will get the player's access token in the response as shown below:

Java

```
import com.enjin.sdk.graphql.GraphQLResponse;
import com.enjin.sdk.models.AccessToken;

// Using a authenticated ProjectClient
GraphQLResponse<AccessToken> res = client.createPlayer(req).get();

AccessToken accessToken = res.getData();
```

C# | Unity

```
using Enjin.SDK.Graphql;
```

```
using Enjin.SDK.Models;
// Using a authenticated ProjectClient
GraphqlResponse<AccessToken> res = client.CreatePlayer(req).Result;

AccessToken accessToken = res.Result;
```

C++

```
#include "enjinsdk/GraphqlResponse.hpp"
#include "enjinsdk/models/AccessToken.hpp"

using namespace enjin::sdk::graphql;
using namespace enjin::sdk::models;

// Using a authenticated ProjectClient
GraphqlResponse<AccessToken> res = client->create_player(req).get();

AccessToken access_token = res.get_result().value();
```

Unreal

```
#include "GraphQlResponse.h"
#include "Model/AccessToken.h"

using namespace Enjin::Sdk::GraphQl;
using namespace Enjin::Sdk::Model;

// Using a authenticated FProjectClient
Client->CreatePlayer(Req)
    .Next([this](TGraphQlResponseForOnePtr<FAccessToken> Res)
{
    FAccessToken AccessToken = Res->GetResult().GetValue();
});
```

### Authenticating an Existing Player

When using the `AuthPlayer` request, we must specify the ID of the player whom we wish to authenticate for as shown below:

Java

```java
import com.enjin.sdk.schemas.project.queries.AuthPlayer;

AuthPlayer req = new AuthPlayer()
    .id("<the-player's-id>");
```

C# | Unity

```csharp
using Enjin.SDK.ProjectSchema;

AuthPlayer req = new AuthPlayer()
    .Id("<the-player's-id>");
```

C++

```cpp
#include "enjinsdk/project/AuthPlayer.hpp"

using namespace enjin::sdk::project;

AuthPlayer req = AuthPlayer()
    .set_id("<the-player's-id>");
```

Unreal

```
#include "Project/AuthPlayer.h"

using namespace Enjin::Sdk::Project;

FAuthPlayer Req = FAuthPlayer()
    .SetId(TEXT("<the-player's-id>"));
```

With the request in hand, we can now send the `AuthPlayer` request to the platform and get the player's access token.

Java

```java
import com.enjin.sdk.graphql.GraphQLResponse;
import com.enjin.sdk.models.AccessToken;
```

```
// Using a authenticated ProjectClient
GraphQlResponse<AccessToken> res = client.authPlayer(req).get();

AccessToken accessToken = res.getData();
```

C# | Unity

```
using Enjin.SDK.Graphql;
using Enjin.SDK.Models;

// Using a authenticated ProjectClient
GraphqlResponse<AccessToken> res = client.AuthPlayer(req).Result;

AccessToken accessToken = res.Result;
```

C++

```
#include "enjinsdk/GraphqlResponse.hpp"
#include "enjinsdk/models/AccessToken.hpp"

using namespace enjin::sdk::graphql;
using namespace enjin::sdk::models;

// Using a authenticated ProjectClient
GraphqlResponse<AccessToken> res = client->auth_player(req).get();

AccessToken access_token = res.get_result().value();
```

Unreal

```
#include "GraphQlResponse.h"
#include "Model/AccessToken.h"

using namespace Enjin::Sdk::GraphQl;
using namespace Enjin::Sdk::Model;

// Using a authenticated FProjectClient
Client->AuthPlayer(Req)
    .Next([this](TGraphQlResponseForOnePtr<FAccessToken> Res)
{
    FAccessToken AccessToken = Res->GetResult().GetValue();
});
```

### Authenticating the Client

Step 1: Create the Client

Creating a player client is similar to creating a project client, but instead of using the `ProjectClient` and its related classes we will be using the `PlayerClient` class.

Java

```java
import com.enjin.sdk.EnjinHosts;
import com.enjin.sdk.PlayerClient;

String mainnet = EnjinHosts.MAIN_NET;
String jumpnet = EnjinHosts.JUMP_NET;
String goerli = EnjinHosts.GOERLI;

PlayerClient playerClient = PlayerClient
    .builder()
    .baseUri(/* Enjin host here */)
    .build();
```

C# | Unity

```csharp
using Enjin.SDK;

System.Uri mainnet = EnjinHosts.MAIN_NET;
System.Uri jumpnet = EnjinHosts.JUMP_NET;
System.Uri goerli = EnjinHosts.GOERLI;

PlayerClient playerClient = PlayerClient
    .Builder()
    .BaseUri(/* Enjin host here */)
    .Build();
```

C++

```cpp
#include "enjinsdk/PlayerClient.hpp"
#include <memory>
#include <string>

using namespace enjin::sdk;
```

```
std::string mainnet = EnjinHosts::MainNet;
std::string jumpnet = EnjinHosts::JumpNet;
std::string goerli = EnjinHosts::Goerli;

std::unique_ptr<PlayerClient> player_client = PlayerClient::builder()
    .base_uri(/* Enjin host here */)
    .build();
```

Unreal

```
#include "PlayerClient.h"

using namespace Enjin::Sdk;

FString MainNet = FEnjinHosts::MainNet();
FString JumpNet= FEnjinHosts::JumpNet();
FString Goerli= FEnjinHosts::Goerli();

TUniquePtr<FPlayerClient> PlayerClient = FPlayerClient::Builder()
    .BaseUrl(/* Enjin host here */)
    .Build();
```

Step 2: Pass the String Token

Once the player client has been created, it may be authenticated with the access token we received from the platform using either the `CreatePlayer` or `AuthPlayer` requests. To do so, we call the respective authentication method and provide the string token contained within.

Java

```
playerClient.auth(accessToken.getToken());
```

C# | Unity

```
playerClient.Auth(accessToken.Token);
```

C++

```
player_client->auth(access_token.get_token().value());
```

Unreal

```
PlayerClient->Auth(AccessToken.GetToken().GetValue());
```

As with the project client, we may check if the token was valid by using the method seen below and getting its Boolean value.

Java

```
boolean result = playerClient.isAuthenticated();
```

C# | Unity

```
bool result = playerClient.IsAuthenticated;
```

C++

```
bool result = player_client->is_authenticated();
```

Unreal

```
bool bResult = PlayerClient->IsAuthenticated();
```

## Managing Players with the Enjin SDKs

## Creating a New Player

The first step in player management is to create a player for our project. By creating a player for our project we will be able to link their Enjin Wallet to the project enabling us to create interactions for melting, sending, and trading assets within our application for users.

Step 1: Create the Request

To create a player we will be using the `CreatePlayer` request. For this request we will need to set the ID we want for our player as shown below:

Java

```java
import com.enjin.sdk.schemas.project.mutations.CreatePlayer;

CreatePlayer req = new CreatePlayer()
    .id("<the-player's-id>");
```

C# | Unity

```csharp
using Enjin.SDK.ProjectSchema;

CreatePlayer req = new CreatePlayer()
    .Id("<the-player's-id>");
```

C++

```cpp
#include "enjinsdk/project/CreatePlayer.hpp"


using namespace enjin::sdk::project;

CreatePlayer req = CreatePlayer()
    .set_id("<the-player's-id>");
```

Unreal

```unreal
#include "Project/CreatePlayer.h"

using namespace Enjin::Sdk::Project;

FCreatePlayer Req = FCreatePlayer()
    .SetId(TEXT("<the-player's-id>"));
```

Step 2: Send the Request

With the `CreatePlayer` request we have created and our `ProjectClient` we can now send the request to the platform as shown below:

Java

```java
import com.enjin.sdk.graphql.GraphQLResponse;
import com.enjin.sdk.models.AccessToken;

// Using a authenticated ProjectClient
GraphQLResponse<AccessToken> res = client.createPlayer(req).get();
```

C# | Unity

```csharp
using Enjin.SDK.Graphql;
using Enjin.SDK.Models;

// Using a authenticated ProjectClient
GraphqlResponse<AccessToken> res = client.CreatePlayer(req).Result;
```

C++

```cpp
#include "enjinsdk/GraphqlResponse.hpp"
#include "enjinsdk/models/AccessToken.hpp"

using namespace enjin::sdk::graphql;
using namespace enjin::sdk::models;

// Using a authenticated ProjectClient
GraphqlResponse<AccessToken> res = client->create_player(req).get();
```

Unreal

```cpp
#include "GraphQlResponse.h"
#include "Model/AccessToken.h"
```

```
using namespace Enjin::Sdk::GraphQl;

// Using a authenticated FProjectClient
Client->CreatePlayer(Req)
    .Next([](TGraphQlResponseForOnePtr<FAccessToken> Res)
{
    // Handle response
});
```

If the request was successful then we will receive the `AccessToken` for the player in the response. With the `AccessToken` we may also now authenticate a platform client for our new player if we choose to do so.

---

## Getting an Existing Player

Once there are players created for our project we may want to get the data associated with them. We will go over how may request this data for an individual player in this section.

Step 1: Create the Request

When requesting player data, how we create our request depends on the schema we are using.

To create the `GetPlayer` request in the project schema we need to specify the ID of the player whose data we are requesting. For this we use a chaining method for setting the ID as shown below:

Java

```
import com.enjin.sdk.schemas.project.queries.GetPlayer;

GetPlayer req = new GetPlayer()
    .id("<the-player's-id>");
```

C# | Unity

```
using Enjin.SDK.ProjectSchema;

GetPlayer req = new GetPlayer()
    .Id("<the-player's-id>");
```

C++

```cpp
#include "enjinsdk/project/GetPlayer.hpp"

using namespace enjin::sdk::project;

GetPlayer req = GetPlayer()
    .set_id("<the-player's-id>");
```

Unreal

```cpp
#include "Project/GetPlayer.h"

using namespace Enjin::Sdk::Project;

FGetPlayer Req = FGetPlayer()
    .SetId(TEXT("<the-player's-id>"));
```

When creating the `GetPlayer` in the player schema we do not specify the player's ID, since the platform infers this from our player client's credentials. This means we need only to instantiate the request as shown below:

Java

```java
import com.enjin.sdk.schemas.player.queries.GetPlayer;

GetPlayer req = new GetPlayer();
```

C# | Unity

```csharp
using Enjin.SDK.PlayerSchema;

GetPlayer req = new GetPlayer();
```

C++

```cpp
#include "enjinsdk/player/GetPlayer.hpp"
```

```
using namespace enjin::sdk::player;

GetPlayer req = GetPlayer();
```

Unreal

```
#include "Player/GetPlayer.h"

using namespace Enjin::Sdk::Player;

FGetPlayer Req = FGetPlayer();
```

Step 2: Send the Request

After we have created our request we send it to the platform using our client's corresponding get player method. In the platform's response, we expect a `Player` model representing the data we requested, which we may retrieve if the request was successful.

Java

```
import com.enjin.sdk.graphql.GraphQLResponse;
import com.enjin.sdk.models.Player;

// Using an authenticated client
GraphQLResponse<Player> res = client.getPlayer(req).get();

Player player = res.getData();
```

C# | Unity

```
using Enjin.SDK.Graphql;
using Enjin.SDK.Models;

// Using an authenticated client
GraphqlResponse<Player> res = client.GetPlayer(req).Result;

Player player = res.Result;
```

C++

```cpp
#include "enjinsdk/GraphqlResponse.hpp"
#include "enjinsdk/models/Player.hpp"

using namespace enjin::sdk::graphql;
using namespace enjin::sdk::models;

// Using an authenticated client
GraphqlResponse<Player> res = client->get_player(req).get();

Player player = res.get_result().value();
```

Unreal

```cpp
#include "GraphQlResponse.h"
#include "Model/Player.h"

using namespace Enjin::Sdk::GraphQl;
using namespace Enjin::Sdk::Model;

// Using an authenticated client
Client->GetPlayer(Req)
    .Next([](TGraphQlResponseForOnePtr<FPlayer> Res)
{
    FPlayer Player = Res->GetResult().GetVaule();
});
```

---

## Linking a Player

### Linking Information

In the `Player` model there's a `LinkingInfo` field that contains information we may use to allow our users to link their Enjin Wallet to their player account in our project. When we get a populated `LinkingInfo` model from the platform it will have two fields we are interested in, the code field which contains the linking code users may enter in their Enjin Wallet to link to the project and the QR field which contains the URL to the QR image we can display and have our users scan in place of using the linking code.

Java

```java
import com.enjin.sdk.models.LinkingInfo;

LinkingInfo linkingInfo = /* linking info source */;

String code = linkingInfo.getCode();
String qr = linkingInfo.getQr();
```

C# | Unity

```csharp
using Enjin.SDK.Models;

LinkingInfo linkingInfo = /* linking info source */;

string code = linkingInfo.Code;
string qr = linkingInfo.Qr;
```

C++

```cpp
#include "enjinsdk/models/LinkingInfo.hpp"
#include <string>

using namespace enjin::sdk::models;

LinkingInfo linking_info = /* linking info source */;

std::string code = linking_info.get_code().value();
std::string qr = linking_info.get_qr().value();
```

Unreal

```cpp
#include "Model/LinkingInfo.h"

using namespace Enjin::Sdk::Model;

FLinkingInfo LinkingInfo = /* linking info source */;

FString Code = LinkingInfo.GetCode().GetValue();
FString Qr = LinkingInfo.GetQr().GetValue();
```

### Getting the Linking Information

There is a minor variation between the project and player schemas when creating the request for getting the player information. For this reason we will cover the creation of the request separately for each client.

### Creating the Request for a Project Client

We may get the player's linking information using the `GetPlayer` request from the project schema and specify that we want the player data to be returned **with** the player's linking information set. To do so we use two chaining methods. One to set the player's ID and another to set the request to retrieve the linking information as seen below:

Java

```java
import com.enjin.sdk.schemas.project.queries.GetPlayer;

GetPlayer req = new GetPlayer()
    .id("<the-player's-id>")
    .withLinkingInfo();
```

C# | Unity

```csharp
using Enjin.SDK.ProjectSchema;
using Enjin.SDK.Shared;

GetPlayer req = new GetPlayer()
    .Id("<the-player's-id>")
    .WithLinkingInfo();
```

C++

```cpp
#include "enjinsdk/project/GetPlayer.hpp"

using namespace enjin::sdk::project;

GetPlayer req = GetPlayer()
    .set_id("<the-player's-id>")
    .set_with_linking_info();
```

Unreal

```
#include "Project/GetPlayer.h"
```

```
using namespace Enjin::Sdk::Project;

FGetPlayer Req = FGetPlayer()
    .SetId(TEXT("<the-player's-id>"))
    .SetWithLinkingInfo();
```

### Creating the Request for a Player Client

In the player schema we use the `GetPlayer` request, however unlike in the project schema this `GetPlayer` request does not need the player's ID set, since the platform infers it from our player client's credentials. This means we only need to specify that we want to request the player data **with** the linking information set for our request as shown below:

Java

```
import com.enjin.sdk.schemas.player.queries.GetPlayer;

GetPlayer req = new GetPlayer()
    .withLinkingInfo();
```

C# | Unity

```
using Enjin.SDK.PlayerSchema;
using Enjin.SDK.Shared;

GetPlayer req = new GetPlayer()
    .WithLinkingInfo();
```

C++

```
#include "enjinsdk/player/GetPlayer.hpp"

using namespace enjin::sdk::player;

GetPlayer req = GetPlayer()
    .set_with_linking_info();
```

Unreal

```
#include "Player/GetPlayer.h"

using namespace Enjin::Sdk::Player;

FGetPlayer Req = FGetPlayer()
    .SetWithLinkingInfo();
```

### Sending the Request

After creating our request we now send it to the platform by using the method in our client of the same name as our request. Once we have the response when can retrieve the data from it.

Java

```
import com.enjin.sdk.graphql.GraphQLResponse;
import com.enjin.sdk.models.LinkingInfo;
import com.enjin.sdk.models.Player;

// Using an authenticated client
GraphQLResponse<Player> res = client.getPlayer(req).get();

Player player = res.getData();

LinkingInfo linkingInfo = player.getLinkingInfo();
```

C# | Unity

```
using Enjin.SDK.Graphql;
using Enjin.SDK.Models;

// Using an authenticated client
GraphqlResponse<Player> res = client.GetPlayer(req).Result;

Player player = res.Result;

LinkingInfo linkingInfo = player.LinkingInfo;
```

C++

```cpp
#include "enjinsdk/GraphqlResponse.hpp"
#include "enjinsdk/models/LinkingInfo.hpp"
#include "enjinsdk/models/Player.hpp"

using namespace enjin::sdk::graphql;
using namespace enjin::sdk::models;

// Using an authenticated client
GraphqlResponse<Player> res = client->get_player(req).get();

Player player = res.get_result().value();

LinkingInfo linking_info = player.get_linking_info().value();
```

Unreal

```cpp
#include "GraphQlResponse.h"
#include "Model/LinkingInfo.h"
#include "Model/Player.h"

using namespace Enjin::Sdk::GraphQl;
using namespace Enjin::Sdk::Model;

// Using an authenticated client
Client->GetPlayer(Req)
    .Next([](TGraphQlResponseForOnePtr<FPlayer> Res)
{
    FPlayer Player = Res->GetResult().GetValue();

    FLinkingInfo LinkingInfo = Player.GetLinkingInfo().GetValue();
});
```

> In the event that the player is already linked to a wallet, the `Player` model returned by the platform may **not** have its `LinkingInfo` set.

---

## Unlinking a Player

### Using the Project Client

Step 1: Get the Wallet Address

Before creating our request to unlink a player's wallet we must first know what their wallet's address is. If we do not already have this value cached in our application we need not worry as we may retrieve it from the platform with the `GetPlayer` request.

To use the `GetPlayer` request to get the wallet address we must specify that we want the player data to be returned **with** the player's wallet data set. For this we use a chaining method for including wallet data on the request as shown below:

Java

```java
import com.enjin.sdk.schemas.project.queries.GetPlayer;

GetPlayer req = new GetPlayer()
    .id("<the-player's-id>")
    .withWallet();
```

C# | Unity

```csharp
using Enjin.SDK.ProjectSchema;
using Enjin.SDK.Shared;

GetPlayer req = new GetPlayer()
    .Id("<the-player's-id>")
    .WithWallet();
```

C++

```cpp
#include "enjinsdk/project/GetPlayer.hpp"

using namespace enjin::sdk::project;

GetPlayer req = GetPlayer()
    .set_id("<the-player's-id>")
    .set_with_wallet();
```

Unreal

```unreal
#include "Project/GetPlayer.h"

using namespace Enjin::Sdk::Project;
```

```
FGetPlayer Req = FGetPlayer()
    .SetId(TEXT("<the-player's-id>"))
    .SetWithPlayerWallet();
```

After creating the request we can now send it to the platform and get the player data returned in the response.

Java

```java
import com.enjin.sdk.graphql.GraphQLResponse;
import com.enjin.sdk.models.Player;

// Using an authenticated ProjectClient
GraphQLResponse<Player> res = client.getPlayer(req).get();

Player player = res.getData();
```

C# | Unity

```csharp
using Enjin.SDK.Graphql;
using Enjin.SDK.Models;


// Using an authenticated ProjectClient
GraphqlResponse<Player> res = client.GetPlayer(req).Result;

Player player = res.Result;
```

C++

```cpp
#include "enjinsdk/GraphqlResponse.hpp"
#include "enjinsdk/models/Player.hpp"

using namespace enjin::sdk::graphql;
using namespace enjin::sdk::models;

// Using an authenticated ProjectClient
GraphqlResponse<Player> res = client->get_player(req).get();

Player player = res.get_result().value();
```

Unreal

```
#include "GraphQlResponse.h"
#include "Model/Player.h"

using namespace Enjin::Sdk::GraphQl;
using namespace Enjin::Sdk::Model;

// Using an authenticated FProjectClient
Client->GetPlayer(Req)
    .Next([](TGraphQlResponseForOnePtr<FPlayer> Res)
{
    FPlayer Player = Res->GetResult().GetValue();
});
```

The `Player` model contains a `Wallet` model as one of its fields. We can get this field and in turn get the Ethereum address of the wallet.

Java

```
import com.enjin.sdk.models.Wallet;

Wallet wallet = player.getWallet();

String ethAddress = wallet.getEthAddress();
```

C# | Unity

```
using Enjin.SDK.Models;

Wallet wallet = player.Wallet;

string ethAddress = wallet.EthAddress;
```

C++

```
#include "enjinsdk/models/Wallet.hpp"
#include <string>
```

```cpp
using namespace enjin::sdk::models;

Wallet wallet = player.get_wallet().value();

std::string eth_address = wallet.get_eth_address().value();
```

Unreal

```cpp
#include "Model/Wallet.h"

using namespace Enjin::Sdk::Model;

FWallet Wallet = Player.GetWallet().GetValue();

FString EthAddress = Wallet.GetEthAddress().GetValue();
```

> ⓘ If the player does not have a wallet linked to them then the `Wallet` field will **not** be set.

Step 2: Create the Request

Once we know the player's wallet address we can create the `UnlinkWallet` request and pass it the wallet address as shown below:

Java

```java
import com.enjin.sdk.schemas.project.mutations.UnlinkWallet;

UnlinkWallet req = new UnlinkWallet()
    .ethAddress("<the-player's-eth-address>");
```

C# | Unity

```cpp
using Enjin.SDK.ProjectSchema;

UnlinkWallet req = new UnlinkWallet()
    .EthAddress("<the-player's-eth-address>");
```

C++

```cpp
#include "enjinsdk/project/UnlinkWallet.hpp"

using namespace enjin::sdk::project;

UnlinkWallet req = UnlinkWallet()
    .set_eth_address("<the-player's-eth-address>");
```

Unreal

```cpp
#include "Project/UnlinkWallet.h"

using namespace Enjin::Sdk::Project;

FUnlinkWallet Req = FUnlinkWallet()
    .SetEthAddress(TEXT("<the-player's-eth-address>"));
```

Step 3: Send the Request

With the created request we now pass it to the client's method with the same name as our request. For `UnlinkWallet` request we expect the returned response to wrap a Boolean value.

Java

```java
import com.enjin.sdk.graphql.GraphQLResponse;

// Using an authenticated ProjectClient
GraphQLResponse<Boolean> res = client.unlinkWallet(req).get();

Boolean result = res.getData();
```

C# | Unity

```cs
using Enjin.SDK.Graphql;

// Using an authenticated ProjectClient
GraphqlResponse<bool?> res = client.UnlinkWallet(req).Result;
```

```
bool? result = res.Result;
```

C++

```
#include "enjinsdk/GraphqlResponse.hpp"

using namespace enjin::sdk::graphql;

// Using an authenticated ProjectClient
GraphqlResponse<bool> res = client->unlink_wallet(req).get();

bool result = res.get_result().value();
```

Unreal

```
#include "GraphQlResponse.h"

using namespace Enjin::Sdk::GraphQl;

// Using an authenticated FProjectClient
Client->UnlinkWallet(Req)
    .Next([](TGraphQlResponseForOnePtr<bool> Res)
{
    bool bResult = Res->GetResult().GetValue();
});
```

The Boolean value contained within the response indicates if the unlink operation was successful. A value of `true` tells us that we successful unlinked our player, whereas a value of `false` may imply that the wallet was already unlinked when we sent our request.

### Using the Player Client

Step 1: Create the Request

Unlike with the project schema when we use the `UnlinkWallet` request from the player schema the platform determines the player we intend to unlink using the credentials of our `PlayerClient`. This means that we do not specify our player's wallet address and instead create the `UnlinkWallet` request with no additional setup as shown below:

Java

```java
import com.enjin.sdk.schemas.player.mutations.UnlinkWallet;

UnlinkWallet req = new UnlinkWallet();
```

C# | Unity

```csharp
using Enjin.SDK.PlayerSchema;

UnlinkWallet req = new UnlinkWallet();
```

C++

```cpp
#include "enjinsdk/player/UnlinkWallet.hpp"

using namespace enjin::sdk::player;

UnlinkWallet req = UnlinkWallet();
```

Unreal

```unrealscript
#include "Player/UnlinkWallet.h"

using namespace Enjin::Sdk::Player;

FUnlinkWallet Req = FUnlinkWallet();
```

Step 2: Send the Request

We now pass our request to the client using the method of the same name. Our expected response will wrap a Boolean value which we may retrieve.

Java

```java
import com.enjin.sdk.graphql.GraphQLResponse;

// Using an authenticated PlayerClient
GraphQLResponse<Boolean> res = client.unlinkWallet(req).get();
```

```
Boolean result = res.getData();
```

C# | Unity

```
using Enjin.SDK.Graphql;

// Using an authenticated PlayerClient
GraphqlResponse<bool?> res = client.UnlinkWallet(req).Result;

bool? result = res.Result;
```

C++

```
#include "enjinsdk/GraphqlResponse.hpp"

using namespace enjin::sdk::graphql;

// Using an authenticated PlayerClient
GraphqlResponse<bool> res = client->unlink_wallet(req).get();

bool result = res.get_result().value();
```

Unreal

```
#include "GraphQlResponse.h"

using namespace Enjin::Sdk::GraphQl;

// Using an authenticated FPlayerClient
Client->UnlinkWallet(Req)
.Next([](TGraphQlResponseForOnePtr<bool> Res)
{
    bool bResult = Res->GetResult().GetValue();
});
```

The Boolean value contained within the response indicates if the unlink operation was successful. A value of `true` tells us that we successful unlinked our player, whereas a value of `false` may imply that our player was not linked when we sent our request.

# Managing Assets with the Enjin SDKs

## Minting Assets

To mint assets to a wallet, we may use the `MintAsset` request. This request is only available for the project schema, therefore we will need an authenticated project client.

Step 1: Create the Mint Data

Before creating the request, we will need to create the `MintInput` that we will be supplying to the request. In order to create the input data we need to specify the wallet address we would like the minted assets to go to as well as the number assets we wish to mint. An example of creating the `MintInput` data can be seen in the code block below:

Java

```java
import com.enjin.sdk.models.MintInput;

MintInput input = new MintInput()
    .to("<the-recipient's-address>")
    .value("<the-number-of-assets>");
```

C# | Unity

```csharp
using Enjin.SDK.Models;

MintInput input = new MintInput()
    .To("<the-recipient's-address>")
    .Value("<the-number-of-assets>");
```

C++

```cpp
#include "enjinsdk/models/MintInput.hpp"

using namespace enjin::sdk::models;

MintInput input = MintInput()
    .set_to("<the-recipient's-address>")
    .set_value("<the-number-of-assets>");
```

Unreal

```
#include "Model/MintInput.h"

using namespace Enjin::Sdk::Model;

FMintInput Input = FMintInput()
    .SetTo(TEXT("<the-recipient's-address>"))
    .SetValue(TEXT("<the-number-of-assets>"));
```

Step 2: Create the Request

Once we have created all the inputs we desire we can start taking a look at creating the request itself. For the request we will use the `MintAsset` request. This request requires us to provide the address of a project wallet, the ID of the asset we are minting, and the input data for the mint(s) we are performing as shown below:

Java

```
import com.enjin.sdk.schemas.project.mutations.MintAsset;

MintAsset req = new MintAsset()
    .ethAddress("<the-wallet's-address>")
    .assetId("<the-asset's-id>")
    .mints(input /* append any additional inputs */));
```

C# | Unity

```
using Enjin.SDK.ProjectSchema;

MintAsset req = new MintAsset()
    .EthAddress("<the-wallet's-address>")
    .AssetId("<the-asset's-id>")
    .Mints(input /* append any additional inputs */);
```

C++

```
#include "enjinsdk/project/MintAsset.hpp"
```

```
using namespace enjin::sdk::project;

MintAsset req = MintAsset()
    .set_eth_address("<the-wallet's-address>")
    .set_asset_id("<the-asset's-id>")
    .set_mints({ input /* append any additional inputs */ });
```

Unreal

```
#include "Project/MintAsset.h"

using namespace Enjin::Sdk::Project;

FMintAsset Req = FMintAsset()
    .SetEthAddress(TEXT("<the-wallet's-address>"))
    .SetAssetId(TEXT("<the-asset's-id>"))
    .SetMints({ Input /* append any additional inputs */ });
```

Step 3: Send the Request

Now that we have created our request we need to send it to the platform using our client's method for minting assets. For this method, we expect to receive a GraphQL response wrapping a `Transaction` model with details for the transaction.

Java

```
import com.enjin.sdk.graphql.GraphQLResponse;
import com.enjin.sdk.models.Transaction;

// Using an authenticated ProjectClient
GraphQLResponse<Transaction> res = client.mintAsset(req).get();
```

C# | Unity

```
using Enjin.SDK.Graphql;
using Enjin.SDK.Models;

// Using an authenticated ProjectClient
GraphqlResponse<Transaction> res = client.MintAsset(req).Result;
```

C++

```cpp
#include "enjinsdk/GraphqlResponse.hpp"
#include "enjinsdk/models/Transaction.hpp"

using namespace enjin::sdk::graphql;
using namespace enjin::sdk::models;

// Using an authenticated ProjectClient
GraphqlResponse<Transaction> res = client->mint_asset(req).get();
```

Unreal

```cpp
#include "GraphQlResponse.h"
#include "Model/Transaction.h"

using namespace Enjin::Sdk::GraphQl;
using namespace Enjin::Sdk::Model;

// Using an authenticated FProjectClient
Client->MintAsset(Req)
    .Next([](TGraphQlResponseForOnePtr<FTransaction> Res)
{
    // Handle response
});
```

---

## Melting Assets

Step 1: Create the Melt Data

Before creating the request, we will need to create the `MeltInput` that we will be supplying to the request. The data we need to set for this input is the asset ID and depending on if the asset being melted is an NFT, then we will need to specify the index and if the asset is not an NFT, then we will specify the amount of the asset being melted instead.

Java

```java
import com.enjin.sdk.models.MeltInput;
```

```
MeltInput input = new MeltInput()
    .assetId("<the-asset's-id>")
    .assetIndex("<the-asset's-index>") // Set this value for NFTs
    .value("<the-number-of-assets>");  // Set this value for FTs
```

C# | Unity

```
using Enjin.SDK.Models;

MeltInput input = new MeltInput()

    .AssetId("<the-asset's-id>")
    .AssetIndex("<the-asset's-index>") // Set this value for NFTs
    .Value("<the-number-of-assets>");  // Set this value for FTs
```

C++

```
#include "enjinsdk/models/MeltInput.hpp"

using namespace enjin::sdk::models;

MeltInput input = MeltInput()
    .set_asset_id("<the-asset's-id>")
    .set_asset_index("<the-asset's-index>") // Set this value for NFTs
    .set_value("<the-number-of-assets>");   // Set this value for FTs
```

Unreal

```
#include "Model/MeltInput.h"

using namespace Enjin::Sdk::Model;

FMeltInput Input = FMeltInput()
    .SetAssetId(TEXT("<the-asset's-id>"))
    .SetAssetIndex(TEXT("<the-asset's-index>")) // Set this value for NFTs
    .SetValue(TEXT("<the-number-of-assets>"));  // Set this value for FTs
```

### Step 2: Create the Request

Next we will create the `MeltAsset` request we will be sending. This request accepts the melt data we created in the previous step as well as any additional melt data we may have created since. However,

depending on the schema we are using we may need to provide additional data.

In the project schema we must specify a project wallet that the request is operating on. An example of this is shown below:

Java

```java
import com.enjin.sdk.schemas.project.mutations.MeltAsset;

MeltAsset req = new MeltAsset()
    .ethAddress("<the-wallet's-address>")
    .melts(input /* append any additional melts */);
```

C# | Unity

```csharp
using Enjin.SDK.ProjectSchema;

MeltAsset req = new MeltAsset()
    .EthAddress("<the-wallet's-address>")
    .Melts(input /* append any additional melts */);
```

C++

```cpp
#include "enjinsdk/project/MeltAsset.hpp"

using namespace enjin::sdk::project;

MeltAsset req = MeltAsset()
    .set_eth_address("<the-wallet's-address>")
    .set_melts({ input /* append any additional melts */ });
```

Unreal

```unreal
#include "Project/MeltAsset.h"

using namespace Enjin::Sdk::Project;

FMeltAsset Req = FMeltAsset()
    .SetEthAddress(TEXT("<the-wallet's-address>"))
    .SetMelts({ Input /* append any additional melts */ });
```

In the player schema we need not provide any additional data aside from the melt data, as the platform infers the which wallet the request operates on by using our client's credentials.

Java

```java
import com.enjin.sdk.schemas.player.mutations.MeltAsset;

MeltAsset req = new MeltAsset()

    .melts(input /* append any additional melts */);
```

C# | Unity

```csharp
using Enjin.SDK.PlayerSchema;

MeltAsset req = new MeltAsset()
    .Melts(input /* append any additional melts */);
```

C++

```cpp
#include "enjinsdk/player/MeltAsset.hpp"

using namespace enjin::sdk::player;

MeltAsset req = MeltAsset()
    .set_melts({ input /* append any additional melts */ });
```

Unreal

```unreal
#include "Player/MeltAsset.h"

using namespace Enjin::Sdk::Player;

FMeltAsset Req = FMeltAsset()
    .SetMelts({ Input /* append any additional melts */ });
```

Step 3: Send the Request

With our created request we may now send it to the platform using our client's method for melting assets. The response we expect to receive is a GraphQL response wrapping a `Transaction` model containing details for the transaction.

Java

```java
import com.enjin.sdk.graphql.GraphQLResponse;
import com.enjin.sdk.models.Transaction;

// Using an authenticated client
GraphQLResponse<Transaction> res = client.meltAsset(req).get();
```

C# | Unity

```csharp
using Enjin.SDK.Graphql;
using Enjin.SDK.Models;

// Using an authenticated client
GraphqlResponse<Transaction> res = client.MeltAsset(req).Result;
```

C++

```cpp
#include "enjinsdk/GraphqlResponse.hpp"
#include "enjinsdk/models/Transaction.hpp"

using namespace enjin::sdk::graphql;
using namespace enjin::sdk::models;

// Using an authenticated client
GraphqlResponse<Transaction> res = client->melt_asset(req).get();
```

Unreal

```cpp
#include "GraphQlResponse.h"
#include "Model/Transaction.h"

using namespace Enjin::Sdk::GraphQl;
using namespace Enjin::Sdk::Model;
```

```
// Using an authenticated client
Client->MeltAsset(Req)
    .Next([](TGraphQlResponseForOnePtr<FTransaction> Res)
{
    // Handle response
});
```

---

## Sending Assets

### Sending One Asset

Step 1: Create the Request

The request we will be using to send a single asset is the `SendAsset` request. To create this request we must know the wallet address of the recipient we are sending the asset to, the ID of the asset, and depending on if the asset is an NFT or not we will need to either know its index if it is an NFT or the amount we wish to send if it is not an NFT. In addition to these arguments, the schema we are using may necessitate setting special arguments.

When using the project schema to send the request we must specify a project wallet that the transaction will operate on. An example of the created request can be seen below:

Java

```java
import com.enjin.sdk.schemas.project.mutations.SendAsset;

SendAsset req = new SendAsset()
    .ethAddress("<the-wallet's-address>")
    .recipientAddress("<the-recipient's-address>")
    .assetId("<the-asset's-id>")
    .assetIndex("<the-asset's-index>") // Set this value for NFTs
    .value("<the-number-of-assets>");  // Set this value for FTs
```

C# | Unity

```
using Enjin.SDK.ProjectSchema;

SendAsset req = new SendAsset()
    .EthAddress("<the-wallet's-address>")
    .RecipientAddress("<the-recipient's-address>")
```

```
    .AssetId("<the-asset's-id>")
    .AssetIndex("<the-asset's-index>") // Set this value for NFTs
    .Value("<the-number-of-assets>");  // Set this value for FTs
```

C++

```cpp
#include "enjinsdk/project/SendAsset.hpp"

using namespace enjin::sdk::project;


SendAsset req = SendAsset()
    .set_eth_address("<the-wallet's-address>")
    .set_recipient_address("<the-recipient's-address>")
    .set_asset_id("<the-asset's-id>")
    .set_asset_index("<the-asset's-index>") // Set this value for NFTs
    .set_value("<the-number-of-assets>");   // Set this value for FTs
```

Unreal

```
#include "Project/SendAsset.h"

using namespace Enjin::Sdk::Project;

FSendAsset Req = FSendAsset()
    .SetEthAddress(TEXT("<the-wallet's-address>"))
    .SetRecipientAddress(TEXT("<the-recipient's-address>"))
    .SetAssetId(TEXT("<the-asset's-id>"))
    .SetAssetIndex(TEXT("<the-asset's-index>")) // Set this value for NFTs
    .SetValue(TEXT("<the-number-of-assets>"));  // Set this value for FTs
```

When using the player schema to send the request we do not specify a wallet for the transaction as this is inferred from our player client's credentials.

Java

```
import com.enjin.sdk.schemas.player.mutations.SendAsset;

SendAsset req = new SendAsset()
    .recipientAddress("<the-recipient's-address>")
    .assetId("<the-asset's-id>")
    .assetIndex("<the-asset's-index>") // Set this value for NFTs
    .value("<the-number-of-assets>");  // Set this value for FTs
```

C# | Unity

```
using Enjin.SDK.PlayerSchema;

SendAsset req = new SendAsset()
    .RecipientAddress("<the-recipient's-address>")
    .AssetId("<the-asset's-id>")
    .AssetIndex("<the-asset's-index>") // Set this value for NFTs
    .Value("<the-number-of-assets>");  // Set this value for FTs
```

C++

```
#include "enjinsdk/player/SendAsset.hpp"

using namespace enjin::sdk::player;

SendAsset req = SendAsset()
    .set_recipient_address("<the-recipient's-address>")
    .set_asset_id("<the-asset's-id>")
    .set_asset_index("<the-asset's-index>") // Set this value for NFTs
    .set_value("<the-number-of-assets>");   // Set this value for FTs
```

Unreal

```
#include "Player/SendAsset.h"

using namespace Enjin::Sdk::Player;

FSendAsset Req = FSendAsset()
    .SetRecipientAddress(TEXT("<the-recipient's-address>"))
    .SetAssetId(TEXT("<the-asset's-id>"))
    .SetAssetIndex(TEXT("<the-asset's-index>")) // Set this value for NFTs
    .SetValue(TEXT("<the-number-of-assets>"));  // Set this value for FTs
```

### Step 2: Send the Request

With the request created we can now send it to the platform for processing by using the method in our client that has the same name as the request itself. In response we will receive the `Transaction` model with details for the transaction.

Java

```java
import com.enjin.sdk.graphql.GraphQLResponse;
import com.enjin.sdk.models.Transaction;

// Using an authenticated client
GraphQLResponse<Transaction> res = client.sendAsset(req).get();
```

C# | Unity

```csharp
using Enjin.SDK.Graphql;
using Enjin.SDK.Models;

// Using an authenticated client
GraphqlResponse<Transaction> res = client.SendAsset(req).Result;
```

C++

```cpp
#include "enjinsdk/GraphqlResponse.hpp"
#include "enjinsdk/models/Transaction.hpp"

using namespace enjin::sdk::graphql;
using namespace enjin::sdk::models;

// Using an authenticated client
GraphqlResponse<Transaction> res = client->send_asset(req).get();
```

Unreal

```cpp
#include "GraphQlResponse.h"
#include "Model/Transaction.h"

using namespace Enjin::Sdk::GraphQl;
using namespace Enjin::Sdk::Model;

// Using an authenticated client
Client->SendAsset(Req)
    .Next([](TGraphQlResponseForOnePtr<FTransaction> Res)
{
    // Handle response
```

```
});
```

### Sending Multiple Assets

Step 1: Create the Transfer Data

To send multiple assets with a request we will need to set up the transfer data which details each send in the batch request. For this data we specify the wallet address the fund is originating from, the wallet address it is heading to, and the asset data such as the ID, the index if it is an NFT, or the amount if not.

Java

```java
import com.enjin.sdk.models.TransferInput;

TransferInput input = new TransferInput()
    .from("<the-source-address>")
    .to("<the-destination-address>")
    .assetId("<the-asset's-id>")
    .assetIndex("<the-asset's-index>") // Set this value for NFTs
    .value("<the-number-of-assets>");  // Set this value for FTs
```

C# | Unity

```csharp
using Enjin.SDK.Models;

TransferInput input = new TransferInput()
    .From("<the-source-address>")
    .To("<the-destination-address>")
    .AssetId("<the-asset's-id>")
    .AssetIndex("<the-asset's-index>") // Set this value for NFTs
    .Value("<the-number-of-assets>");  // Set this value for FTs
```

C++

```cpp
#include "enjinsdk/models/TransferInput.hpp"

using namespace enjin::sdk::models;

TransferInput input = TransferInput()
    .set_from("<the-source-address>")
    .set_to("<the-destination-address>")
```

```
    .set_asset_id("<the-asset's-id>")
    .set_asset_index("<the-asset's-index>") // Set this value for NFTs
    .set_value("<the-number-of-assets>");   // Set this value for FTs
```

Unreal

```
#include "Model/TransferInput.h"

using namespace Enjin::Sdk::Model;


FTransferInput Input = FTransferInput()
    .SetFrom(TEXT("<the-source-address>"))
    .SetTo(TEXT("<the-destination-address>"))
    .SetAssetId(TEXT("<the-asset's-id>"))
    .SetAssetIndex(TEXT("<the-asset's-index>")) // Set this value for NFTs
    .SetValue(TEXT("<the-number-of-assets>"));  // Set this value for FTs
```

> ⓘ If the asset ID is omitted, then the transfer data will send **ENJ** instead.

Step 2: Create the Request

Next we create the `AdvancedSendAsset` request that we will be sending to the platform. For this request we will be using a chaining method to pass the transfer data we created as well as any additional transfers we might add. However, what additional argument(s) we set for our request is determined by the schema we are using.

When using the project schema to send the request we must specify a project wallet that the transaction will operate on. An example of the created request can be seen below:

Java

```
import com.enjin.sdk.schemas.project.mutations.AdvancedSendAsset;

AdvancedSendAsset req = new AdvancedSendAsset()
    .ethAddress("<the-wallet's-address>")
    .transfers(input /* append any additional transfers */);
```

C# | Unity

```
using Enjin.SDK.ProjectSchema;

AdvancedSendAsset req = new AdvancedSendAsset()
    .EthAddress("<the-wallet's-address>")
    .Transfers(input /* append any additional transfers */);
```

C++

```
#include "enjinsdk/project/AdvancedSendAsset.hpp"


using namespace enjin::sdk::project;

AdvancedSendAsset req = AdvancedSendAsset()
    .set_eth_address("<the-wallet's-address>")
    .set_transfers({ input /* append any additional transfers */ });
```

Unreal

```
#include "Project/AdvancedSendAsset.h"

using namespace Enjin::Sdk::Project;

FAdvancedSendAsset Req = FAdvancedSendAsset()
    .SetEthAddress(TEXT("<the-wallet's-address>"))
    .SetTransfers({ Input /* append any additional transfers */ });
```

When using the player schema to send the request we do not specify a wallet for the transaction as this is inferred from our player client's credentials. Therefore we need only pass the transfer(s) data as shown:

Java

```
import com.enjin.sdk.schemas.player.mutations.AdvancedSendAsset;

AdvancedSendAsset req = new AdvancedSendAsset()
    .transfers(input /* append any additional transfers */);
```

C# | Unity

```
using Enjin.SDK.PlayerSchema;
AdvancedSendAsset req = new AdvancedSendAsset()
    .Transfers(input /* append any additional transfers */);
```

C++

```
#include "enjinsdk/player/AdvancedSendAsset.hpp"

using namespace enjin::sdk::player;


AdvancedSendAsset req = AdvancedSendAsset()
    .set_transfers({ input /* append any additional transfers */ });
```

Unreal

```
#include "Player/AdvancedSendAsset.h"

using namespace Enjin::Sdk::Player;

FAdvancedSendAsset Req = FAdvancedSendAsset()
    .SetTransfers({ Input /* append any additional transfers */ });
```

### Step 3: Send the Request

With the request created we can now send it to the platform for processing by using the method in our client that has the same name as the request itself. In response, we will receive the `Transaction` model with details for the transaction.

Java

```
import com.enjin.sdk.graphql.GraphQLResponse;
import com.enjin.sdk.models.Transaction;

// Using an authenticated client
GraphQLResponse<Transaction> res = client.advancedSendAsset(req).get();
```

C# | Unity

```
using Enjin.SDK.Graphql;
using Enjin.SDK.Models;

// Using an authenticated client
GraphqlResponse<Transaction> res = client.AdvancedSendAsset(req).Result;
```

C++

```
#include "enjinsdk/GraphqlResponse.hpp"
#include "enjinsdk/models/Transaction.hpp"

using namespace enjin::sdk::graphql;
using namespace enjin::sdk::models;

// Using an authenticated client
GraphqlResponse<Transaction> res = client->advanced_send_asset(req).get();
```

Unreal

```
#include "GraphQlResponse.h"
#include "Model/Transaction.h"

using namespace Enjin::Sdk::GraphQl;
using namespace Enjin::Sdk::Model;

// Using an authenticated client
Client->AdvancedSendAsset(Req)
    .Next([](TGraphQlResponseForOnePtr<FTransaction> Res)
{
    // Handle response
});
```

## Setting up an Event Service

## Setup & Start

Step 1: Getting the Platform Data

Before starting the `PusherEventService` we first need to request information from the platform related

to its notification settings. For the Pusher service, this will be the cluster our service needs to connect to, the key it needs to connect to, as well as any other options the platform has specified.

To get the platform information we use the `GetPlatform` request that is shared between and usable by both the `ProjectClient` and the `PlayerClient`, allowing us to use either one for this step. When creating the request we need to specify that we want our response **with** the platform's notification settings. This can be done so with a chaining method as shown in the example below:

Java

```java
import com.enjin.sdk.schemas.shared.queries.GetPlatform;


GetPlatform req = new GetPlatform()
    .withNotificationDrivers();
```

C# | Unity

```csharp
using Enjin.SDK.Shared;

GetPlatform req = new GetPlatform()
    .WithNotificationDrivers();
```

C++

```cpp
#include "enjinsdk/shared/GetPlatform.hpp"

using namespace enjin::sdk::shared;

GetPlatform req = GetPlatform()
    .set_with_notifications();
```

Unreal

```unreal
#include "Shared/GetPlatform.h"

using namespace Enjin::Sdk::Shared;

FGetPlatform Req = FGetPlatform()
    .SetWithNotifications();
```

Now that we have created the request we can send it to the platform. We will need to use the method with the matching name as our request and the result we are expecting is a GraphQL response wrapping the `Platform` model, which represents the type of the same name in the platform's API.

Java

```java
import com.enjin.sdk.graphql.GraphQLResponse;
import com.enjin.sdk.models.Platform;

// Using an authenticated client

GraphQLResponse<Platform> res = client.getPlatform(req).get();

Platform platform = res.getData();
```

C# | Unity

```csharp
using Enjin.SDK.Graphql;
using Enjin.SDK.Models;

// Using an authenticated client
GraphqlResponse<Platform> res = client.GetPlatform(req).Result;

Platform platform = res.Result;
```

C++

```cpp
#include "enjinsdk/GraphqlResponse.hpp"
#include "enjinsdk/models/Platform.hpp"

using namespace enjin::sdk::graphql;
using namespace enjin::sdk::models;

// Using an authenticated client
GraphqlResponse<Platform> res = client->get_platform(req).get();

Platform platform = res.get_result().value();
```

Unreal

```cpp
#include "GraphQlResponse.h"
#include "Model/Platform.h"

using namespace Enjin::Sdk::GraphQl;
using namespace Enjin::Sdk::Model;

// Using an authenticated client
Client->GetPlatform()
    .Next([](TGraphQlResponseForOnePtr<FPlatform> Res)
{
    FPlatform Platform = Res->GetResult().GetValue();
});
```

### Step 2: Starting the Service

To start the Pusher event service we must use its builder class construct it and pass the `Platform` model instance from the previous step into its constructor.

Java

```java
import com.enjin.sdk.events.PusherEventService;

PusherEventService service = PusherEventService
    .builder()
    .platform(platform)
    .build();
```

C# | Unity

```cs
using Enjin.SDK.Events;

PusherEventService service = PusherEventService
    .Builder()
    .Platform(platform)
    .Build();
```

C++

```cpp
#include "enjinsdk/PusherEventService.hpp"
#include <memory>
```

```
using namespace enjin::sdk::events;

std::unique_ptr<PusherEventService> service = PusherEventService::builder()
    .platform(platform)
    .build();
```

Unreal

```
#include "PusherEventService.h"

using namespace Enjin::Sdk::Event;


TUniquePtr<FPusherEventService> Service = FPusherEventService::Builder()
    .Platform(Platform)
    .Build();
```

Now that we have instantiated our event service we may call its start method. Starting the service tends to involve asynchronous processes. For this reason, event services in the SDKs return an asynchronous type, which we may use to wait for the process to complete if we choose to do so.

Java

```
import java.util.concurrent.Future;

Future<Void> future = service.start();
```

C# | Unity

```
using System.Threading.Tasks;

Task task = service.Start();
```

C++

```
#include <future>

std::future<void> future = service->start();
```

Unreal

```
Service->Start().Next([](bool bResult)
{
    // Handle service start
});
```

Additionally the event service comes with a method which we may use at any time to check if the event service is connected to the server as shown below:

Java

```
boolean result = service.isConnected();
```

C# | Unity

```
bool result = service.IsConnected();
```

C++

```
bool result = service->is_connected();
```

Unreal

```
bool bResult = Service->IsConnected();
```

Step 3: Shutting Down the Service

Once we are done using the event service we may want to shut it down to free up any resources that may have been acquired during its lifetime. To do so we call the service's shutdown method, and as with the start method, we may use the asynchronous type returned to wait for this process to finish.

Java

```java
import java.util.concurrent.Future;

Future<Void> future = service.shutdown();
future.get();
```

C# | Unity

```csharp
using System.Threading.Tasks;

Task task = service.Shutdown();
task.Wait();
```

C++

```cpp
#include <future>

std::future<void> future = service->shutdown();
future.wait();
```

Unreal

```
Service->Shutdown().Next([](bool bResult)
{
    // Handle service shut down
});
```



In the C++ and Unreal SDKs the destructor for the Pusher event service will close the WebSocket connection if it has not already been closed.

---

## Service Notifications

The event services offer ways for us to listen for and handle when certain flags are raised within them, such

as when they establish a connection with the server, are disconnected from the server, or encounter an error. However, the API for doing so varies between the SDKs, so we will look at them individually throughout this section.

### Java SDK

For the Java SDK we provide the `IConnectionEventListener` interface which may be passed to the service through any of the `start()` methods which accept it as an argument. The example code block below shows how this listener may be set with an anonymous class.

```
import com.enjin.sdk.events.IConnectionEventListener;

service.start(new IConnectionEventListener() {

    @Override
    public void onConnect() { /* Place code here */ }

    @Override
    public void onDisconnect() { /* Place code here */ }

    @Override
    public void onError(Exception e) { /* Place code here */ }
});
```

> ⓘ The `IConnectionEventListener` interface utilizes default methods, which enables us to override only the methods we wish to implement ourselves and ignore the rest.

### C# and Unity SDKs

For the C# and Unity SDKs we utilize .NET's field like events for raising events related to the service's internal state. The example code block below shows how these events may be used with a delegate.

```
service.Connected += (sender, args) => { /* Place code here */ };

service.Disconnected += (sender, args) => { /* Place code here */ };

service.Error += (sender, exception) => { /* Place code here */ };
```

### C++ SDK

For the C++ SDK we utilize the function wrapper `std::function` from the standard library as our handler for notifications raised by the service. The example code block below shows how these handlers may be used with lambda expressions.

```
service->set_connected_handler([]() { /* Place code here */ });
```

```
service->set_disconnected_handler([]() { /* Place code here */ });

service->set_error_handler([](const std::exception& e) { /* Place code here */ });
```

#### Unreal SDK

For the Unreal SDK we utilize Unreal's architecture for declaring, binding, and executing events. The example code block below shows how we may bind to these events.

```
Service->OnConnected().AddLambda([]() { /* Place code here */ });

Service->OnDisconnected().AddLambda([]() { /* Place code here */ });


Service->OnError().AddLambda([](const FString& Error) { /* Place code here */ });
```

Documentation for this architecture is available at Unreal Engine's documentation site:

> Events

## Interacting with Cloud Events

## Creating an Event Listener

For us to be notified when our event service receives an event from the cloud we must provide it with an event listener for it to relay to. The SDKs provide an interface that we will use to implement a listener the service will use. An example of how we may implement the `IEventListener` interface can be seen below:

Java

```
import com.enjin.sdk.events.IEventListener;
import com.enjin.sdk.models.NotificationEvent;

class Listener implements IEventListener {
    @Override
    public void notificationReceived(NotificationEvent event) {
        // Place code here
    }
}
```

C# | Unity

```
using Enjin.SDK.Events;
using Enjin.SDK.Models;

class Listener : IEventListener {
    public void NotificationReceived(NotificationEvent notificationEvent) {
        // Place code here
    }
}
```

C++

```
#include "enjinsdk/IEventListener.hpp"
#include "enjinsdk/models/NotificationEvent.hpp"

using namespace enjin::sdk::events;
using namespace enjin::sdk::models;

class Listener : public IEventListener {
public:
    void notification_received(const NotificationEvent& event) override {
        // Place code here
    }
};
```

Unreal

```
#include "IEventListener.h"
#include "Model/NotificationEvent.h"

using namespace Enjin::Sdk::Event;
using namespace Enjin::Sdk::Model;

class FEventListener : public IEventListener
{
public:
    void NotificationReceived(const FNotificationEvent& Event) override
    {
        // Place code here
    }
};
```

The `NotificationEvent` argument in the notification received method models an event that the event service has received and parsed. This argument will contain information telling us the type of the event that was received, the channel the event was broadcasted on, and the JSON message that is the data associated with the event, such as the asset ID for asset events or the wallet address for wallet events.

---

## Listener Registration

### Registering a Listener

Once we have defined an event listener we may register it with our event service. The event service has multiple registration methods with different functionality, but for now, we will use the most basic of these methods. The basic registration method will register our listener and for any event the service receives and processes from the cloud, our listener will receive the parsed event data. When registering our listener we will also receive the listener registration object containing data about our registration.

Java

```java
import com.enjin.sdk.events.EventListenerRegistration;

Listener listener = new Listener();

EventListenerRegistration reg = service.registerListener(listener);
```

C# | Unity

```csharp
using Enjin.SDK.Events;

Listener listener = new Listener();

EventListenerRegistration reg = service.RegisterListener(listener);
```

C++

```cpp
#include "enjinsdk/EventListenerRegistration.hpp"
#include <memory>

using namespace enjin::sdk::events;

std::shared_ptr<Listener> listener = std::make_shared<Listener>();
```

```
EventListenerRegistration reg = service->register_listener(listener);
```

Unreal

```
#include "EventListenerRegistration.h"

using namespace Enjin::Sdk::Event;


FEventListenerRef Listener =
    MakeShared<FEventListener, ESPMode::ThreadSafe>();

FEventListenerRegistrationRef Reg =
    EventService->RegisterListener(Listener);
```

### Unregistering a Listener

If we wish to do so we may also unregister our listener from the service. To do so we pass our listener as an argument to the service's unregister method. In the event that we did not create a variable for our listener before registering it, we may use the reference that is stored in the registration that we received upon registration as shown below:

Java

```
// Using a listener registration
service.unregisterListener(reg.getListener());
```

C# | Unity

```
// Using a listener registration
service.UnregisterListener(reg.Listener);
```

C++

```
// Using a listener registration
service->unregister_listener(reg.get_listener());
```

Unreal

```
// Using a listener registration
Service->UnregisterListener(Reg->GetListener().Get());
```

---

## Event Channel Subscription

### Subscribing to a Channel

To subscribe to one of the four-event channels used by the cloud we must provide the identifier for it. See the code block below for what these identifiers are:

Java

```
service.subscribeToProject("<the-project's-uuid>");

service.subscribeToPlayer("<the-project's-uuid>", "<the-player's-id>");

service.subscribeToAsset("<the-asset's-id>");

service.subscribeToWallet("<the-wallet's-address>");
```

C# | Unity

```
service.SubscribeToProject("<the-project's-uuid>");

service.SubscribeToPlayer("<the-project's-uuid>", "<the-player's-id>");

service.SubscribeToAsset("<the-asset's-id>");

service.SubscribeToWallet("<the-wallet's-address>");
```

C++

```
service->subscribe_to_project("<the-project's-uuid>");

service->subscribe_to_player("<the-project's-uuid>", "<the-player's-id>");
```

```
service->subscribe_to_asset("<the-asset's-id>");

service->subscribe_to_wallet("<the-wallet's-address>");
```

Unreal

```
Service->SubscribeToProject(TEXT("<the-project's-uuid>"));

Service->SubscribeToPlayer(TEXT("<the-project's-uuid>"), TEXT("<the-player's-id>"))

Service->SubscribeToAsset(TEXT("<the-asset's-id>"));

Service->SubscribeToWallet(TEXT("<the-wallet's-address>"));
```

After subscribing to a channel our event service will now receive events that are broadcasted on the specified event channel.

> When using the `PusherEventService`, subscribed channels may only be subscribed to after starting the service for the first time.

We may also check to see if our service is already subscribed to a particular event channel by using the appropriate method and passing the identifiers as shown below:

Java

```
service.isSubscribedToProject("<the-project's-uuid>");

service.isSubscribedToPlayer("<the-project's-uuid>", "<the-player's-id>");

service.isSubscribedToAsset("<the-asset's-id>");

service.isSubscribedToWallet("<the-wallet's-address>");
```

C# | Unity

```
service.IsSubscribedToProject("<the-project's-uuid>");

service.IsSubscribedToPlayer("<the-project's-uuid>", "<the-player's-id>");
```

```
service.IsSubscribedToAsset("<the-asset's-id>");

service.IsSubscribedToWallet("<the-wallet's-address>");
```

C++

```
service->is_subscribed_to_project("<the-project's-uuid>");


service->is_subscribed_to_player("<the-project's-uuid>", "<the-player's-id>");

service->is_subscribed_to_asset("<the-asset's-id>");

service->is_subscribed_to_wallet("<the-wallet's-address>");
```

Unreal

```
Service->IsSubscribedToProject(TEXT("<the-project's-uuid>"));

Service->IsSubscribedToPlayer(TEXT("<the-project's-uuid>"), TEXT("<the-player's-id>

Service->IsSubscribedToAsset(TEXT("<the-asset's-id>"));

Service->IsSubscribedToWallet(TEXT("<the-wallet's-address>"));
```

ⓘ Channel subscriptions on the `PusherEventService` persist after shutting down and restarting the service.

### Unsubscribing from a Channel

To unsubscribe our service from a channel we will need to recall the information we provided when we first subscribed and provide it to the service's unsubscribe method as well. After doing so our service will no longer receive events from the cloud for the specified channel.

Java

```
service.unsubscribeToProject("<the-project's-uuid>");

service.unsubscribeToPlayer("<the-project's-uuid>", "<the-player's-id>");
```

```
service.unsubscribeToAsset("<the-asset's-id>");

service.unsubscribeToWallet("<the-wallet's-address>");
```

C# | Unity

```
service.UnsubscribeToProject("<the-project's-uuid>");

service.UnsubscribeToPlayer("<the-project's-uuid>", "<the-player's-id>");

service.UnsubscribeToAsset("<the-asset's-id>");

service.UnsubscribeToWallet("<the-wallet's-address>");
```

C++

```
service->unsubscribe_to_project("<the-project's-uuid>");

service->unsubscribe_to_player("<the-project's-uuid>", "<the-player's-id>");

service->unsubscribe_to_asset("<the-asset's-id>");

service->unsubscribe_to_wallet("<the-wallet's-address>");
```

Unreal

```
Service->UnsubscribeToProject(TEXT("<the-project's-uuid>"));

Service->UnsubscribeToPlayer(TEXT("<the-project's-uuid>"), TEXT("<the-player's-id>"

Service->UnsubscribeToAsset(TEXT("<the-asset's-id>"));

Service->UnsubscribeToWallet(TEXT("<the-wallet's-address>"));
```

---

## Matching Listeners for Events

In the case where we would like to register a more specialized event listener that only receives particular

event types, we may use various means to specify which events we want to receive to offload the responsibility of filtering events from our listener to the event service instead.

### Functional Matcher

For registering our listener in the service to process events under tailored conditions we specify, we may use the service's method for registering with a paired matcher for our listener. If the matcher returns `true`, then the service will pass the `NotificationEvent` to our listener for processing, whereas if the matcher returns `false`, then our listener will not receive the event. As with standard registration, this method returns a listener registration object.

Java

```
service.registerListenerWithMatcher(listener, notificationEvent -> {
    // Place code here

    return true;
}); // Returns listener registration
```

C# | Unity

```
service.RegisterListenerWithMatcher(listener, eventType => {
    // Place code here

    return true;
}); // Returns listener registration
```

C++

```
#include "enjinsdk/models/EventType.hpp"

using namespace enjin::sdk::models;

service->register_listener_with_matcher(listener, [](EventType type) {
    // Place code here

    return true;
}); // Returns listener registration
```

Unreal

```cpp
#include "Model/EventType.h"

using namespace Enjin::Sdk::Model;

Service->RegisterListenerWithMatcher(Listener, [](const EEventType Type)
{
    // Place code here

    return true;
}); // Returns listener registration
```

### Including Events

For registering our listener in the service to process only a select set of events, we may use the service's method for registering with including event types. As with standard registration, this method returns a listener registration object.

Java

```java
import com.enjin.sdk.models.EventType;

service.registerListenerIncludingTypes(listener,
    EventType.ASSET_MELTED,
    EventType.ASSET_MINTED,
    EventType.ASSET_TRANSFERRED
); // Returns listener registration
```

C# | Unity

```cs
using Enjin.SDK.Events;

service.RegisterListenerIncludingTypes(listener,
    EventType.ASSET_MELTED,
    EventType.ASSET_MINTED,
    EventType.ASSET_TRANSFERRED
); // Returns listener registration
```

C++

```cpp
#include "enjinsdk/models/EventType.hpp"
```

```
using namespace enjin::sdk::models;

service->register_listener_including_types(listener, {
    EventType::AssetMelted,
    EventType::AssetMinted,
    EventType::AssetTransferred
}); // Returns listener registration
```

Unreal

```
#include "Model/EventType.h"

using namespace Enjin::Sdk::Model;

Service->RegisterListenerIncludingTypes(Listener,
{
    EEventType::AssetMelted,
    EEventType::AssetMinted,
    EEventType::AssetTransferred
}); // Returns listener registration
```

### Excluding Events

For registering our listener in the service to not process a select set of events, we may use the service's method for registering with excluding event types. As with standard registration, this method returns a listener registration object.

Java

```
import com.enjin.sdk.models.EventType;

service.registerListenerExcludingTypes(listener,
    EventType.ASSET_MELTED,
    EventType.ASSET_MINTED,
    EventType.ASSET_TRANSFERRED
); // Returns listener registration
```

C# | Unity

```
using Enjin.SDK.Events;

service.RegisterListenerExcludingTypes(listener,
```

```
    EventType.ASSET_MELTED,
    EventType.ASSET_MINTED,
    EventType.ASSET_TRANSFERRED
); // Returns listener registration
```

C++

```cpp
#include "enjinsdk/models/EventType.hpp"

using namespace enjin::sdk::models;

service->register_listener_excluding_types(listener, {
    EventType::AssetMelted,
    EventType::AssetMinted,
    EventType::AssetTransferred
}); // Returns listener registration
```

Unreal

```cpp
#include "Model/EventType.h"

using namespace Enjin::Sdk::Model;

Service->RegisterListenerExcludingTypes(Listener,
{
    EEventType::AssetMelted,
    EEventType::AssetMinted,
    EEventType::AssetTransferred
}); // Returns listener registration
```

### Filtered Listener

We may also attach attributes/annotations on listener classes to indicate to the event service which events we want our listener to receive or not receive. Generally, we can set which events we want them to filter for and also indicate if we want the filter to allow or disallow said events. Examples of what this may look like in the SDKs which support this feature can be seen below:

Java

```java
import com.enjin.sdk.events.EventFilter;
import com.enjin.sdk.events.IEventListener;
```

```
import com.enjin.sdk.models.EventType;

@EventFilter(allow = true, value = {EventType.ASSET_MELTED /* more events */})
class Listener implements IEventListener { /* ... */ }
```

C# | Unity

```
using Enjin.SDK.Events;
using Enjin.SDK.Models;

[EventFilter(allowed: true, EventType.ASSET_MELTED /* more events */)]
class Listener : IEventListener { /* ... */ }
```

# Message Logging

## Logger

### ILogger Interface

For logging, the SDKs define an `ILogger` interface that provides a common API for their logging purposes.

### Built-in Logger

Each SDK comes with a built-in `Logger` class that implements the `ILogger` interface. These loggers offer basic functionality that allow us to jump into using logging features without dedicating too much time to develop our own implementation first.

---

## Message Levels

For logging, the SDKs define six levels that messages may be logged on. These log levels are:

- `DEBUG`
- `TRACE`
- `INFO`
- `WARN`
- `ERROR`

- `SEVERE`

> Some SDKs may not support all message levels for their built-in `Logger` class. Check the class documentation to see which log levels the logger may utilize.

---

## Logger Provider

The `LoggerProvider` is the highest level class that we interact with when utilizing logging functions in the SDKs. The logger provider allows components within the SDKs to share log settings and resources and helps us save resources as well.

To use `LoggerProvider` we must pass an instance of a class implementing the `ILogger` interface, such as the `Logger` class built into the SDKs as shown in the example below:

Java

```java
import com.enjin.sdk.utils.Logger;
import com.enjin.sdk.utils.LoggerProvider;

Logger logger = new Logger();

LoggerProvider provider = new LoggerProvider(logger);
```

C# | Unity

```csharp
using Enjin.SDK.Utils;

Logger logger = new Logger();

LoggerProvider provider = new LoggerProvider(logger);
```

C++

```cpp
#include "enjinsdk/Logger.hpp"
#include "enjinsdk/LoggerProvider.hpp"
#include <memory>

using namespace enjin::sdk::utils;
```

```
std::shared_ptr<Logger> logger = std::make_shared<Logger>();

std::shared_ptr<LoggerProvider> provider = std::make_shared<LoggerProvider>(logger)
```

Unreal

```
#include "Logger.h"
#include "LoggerProvider.h"

using namespace Enjin::Sdk::Util;

TSharedRef<FLogger> Logger = MakeShared<FLogger>();

FLoggerProviderPtr Provider = MakeShared<FLoggerProvider>(Logger);
```

When creating the `LoggerProvider` we may also choose to define the default message level and the default debug level it calls the logger on as shown in the example below:

Java

```
import com.enjin.sdk.utils.LogLevel;
import com.enjin.sdk.utils.LoggerProvider;

LogLevel message = LogLevel.WARN;
LogLevel debug = LogLevel.ERROR;

// With a defined ILogger instance
new LoggerProvider(logger, message, debug);
```

C# | Unity

```
using Enjin.SDK.Utils;

LogLevel message = LogLevel.WARN;
LogLevel debug = LogLevel.ERROR;

// With a defined ILogger instance
new LoggerProvider(logger, message, debug);
```

C++

```cpp
#include "enjinsdk/LogLevel.hpp"
#include "enjinsdk/LoggerProvider.hpp"
#include <memory>

using namespace enjin::sdk::utils;

LogLevel message = LogLevel::Warn;
LogLevel debug = LogLevel::Error;

// With a defined ILogger instance
std::make_shared<LoggerProvider>(logger, message, debug);
```

Unreal

```cpp
#include "LogLevel.h"
#include "LoggerProvider.h"

using namespace Enjin::Sdk::Util;

ELogLevel Message = ELogLevel::Warn;
ELogLevel Debug = ELogLevel::Error;

// With a defined ILogger instance
MakeShared<FLoggerProvider>(Logger, Message, Debug);
```

By default, `LoggerProvider` sets the message level to `INFO` and the debug level to `DEBUG` if we do not define them ourselves.

---

## HTTP Traffic Logging

Platform clients may be configured to log the traffic of the requests and responses they send to and receive from the platform. We use an enum, `HttpLogLevel` to define the log levels we may use for HTTP traffic. These log levels are:

- `NONE`
- `BASIC`
- `HEADERS`
- `BODY`

The default log level for HTTP traffic is `NONE` making this an **opt-in** feature, which will require us to declare the log level we will want to use. `BASIC` enables logging of the request method (e.g. `GET`, `POST`, etc...), the URI, the content-length as well as the response's status, URI, and round-trip time. `HEADERS` is similar to `BASIC`, but also includes the request and response headers. And finally, `BODY` logs similar information as `HEADERS` but also includes the entire content body of out going requests and incoming responses.

> ⚠ CAUTION: When using the `BODY` log level, any `AuthProject` request will be logged as well as the project's secret key which is sent as a parameter of the request.

To configure the client, we call a method on its builder to set the HTTP log level and we must also provide an instance of `LoggerProvider` that log messages will be sent to as shown in the example below:

Java

```java
import com.enjin.sdk.http.HttpLogLevel;

HttpLogLevel basic = HttpLogLevel.BASIC;
HttpLogLevel headers = HttpLogLevel.HEADERS;
HttpLogLevel body = HttpLogLevel.BODY;

// Builder from either ProjectClient or PlayerClient
builder.httpLogLevel(/* HTTP log level here */)
       .loggerProvider(/* LoggerProvider here */);
```

C# | Unity

```csharp
using Enjin.SDK.Http;

HttpLogLevel basic = HttpLogLevel.BASIC;
HttpLogLevel headers = HttpLogLevel.HEADERS;
HttpLogLevel body = HttpLogLevel.BODY;

// Builder from either ProjectClient or PlayerClient
builder.HttpLogLevel(/* HTTP log level here */)
       .LoggerProvider(/* LoggerProvider here */);
```

C++

```cpp
#include "enjinsdk/HttpLogLevel.hpp"
```

```
using namespace enjin::sdk::http;

HttpLogLevel basic = HttpLogLevel::Basic;
HttpLogLevel headers = HttpLogLevel::Headers;
HttpLogLevel body = HttpLogLevel::Body;

// Builder from either ProjectClient or PlayerClient
builder.http_log_level(/* HTTP log level here */)
       .logger_provider(/* LoggerProvider shared-pointer here */);
```

Unreal

```
#include "HttpLogLevel.h"

using namespace Enjin::Sdk::Http;

EHttpLogLevel Basic = EHttpLogLevel::Basic;
EHttpLogLevel Headers = EHttpLogLevel::Headers;
EHttpLogLevel Body = EHttpLogLevel::Body;

// Builder from either FProjectClient or FPlayerClient
Builder.HttpLogLevel(/* HTTP log level here */)
       .LoggerProvider(/* FLoggerProviderPtr here */)
```

---

## Cloud Event Logging

The `PusherEventService` may be configured to log incoming cloud events or any errors it encounters by providing its builder with a logger provider as shown below:

Java

```
// Builder from PusherEventService
builder.loggerProvider(/* LoggerProvider here */);
```

C# | Unity

```
// Builder from PusherEventService
builder.LoggerProvider(/* LoggerProvider here */);
```

C++

```
// Builder from PusherEventService
builder.logger_provider(/* LoggerProvider shared-pointer here */);
```

Unreal

```
// Builder from FPusherEventService
Builder.LoggerProvider(/* FLoggerProviderPtr here */);
```

# RESOURCES

# DOWNLOADS