Intro

Welcome to the hands-on guide for the ethers-rs library!

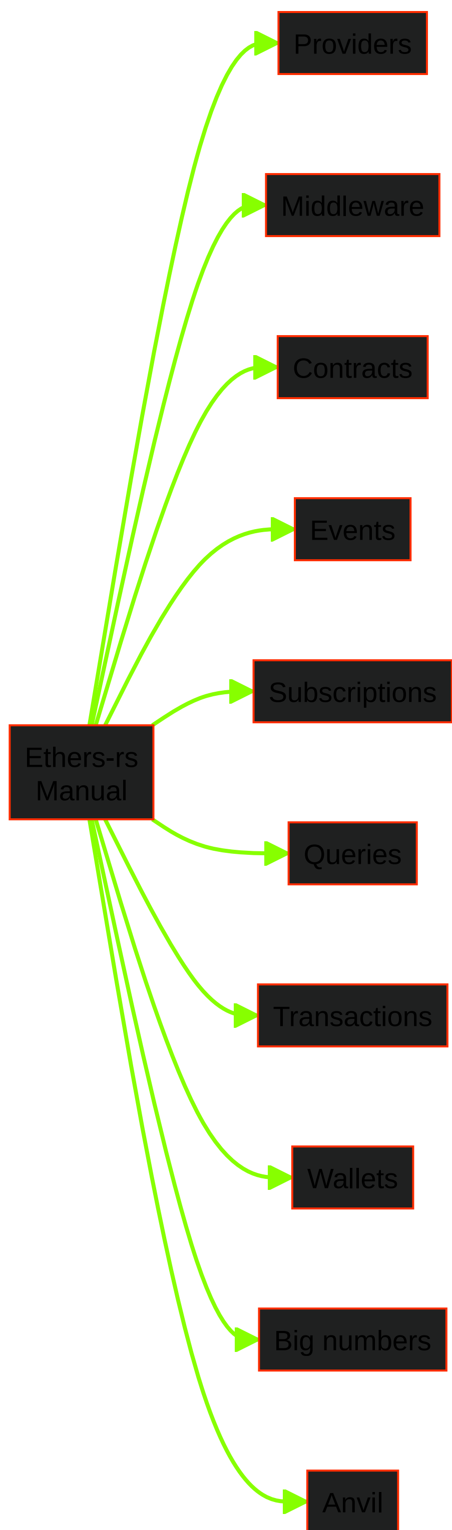
This documentation contains a collection of examples demonstrating how to use the library to build Ethereum-based applications in Rust. The examples cover a range of topics, from basic smart contract interactions to more advanced usage of ethers-rs.

> **Info**
>
> You can find the official ethers-rs documentation on docs.rs - here.

Each example includes a detailed description of the functionality being demonstrated, as well as complete code snippets that you can use as a starting point for your own projects.

We hope that these docs will help you get started with ethers-rs and give you a better understanding of how to use the library to build your own web3 applications in Rust. If you have any questions or need further assistance, please don't hesitate to reach out to the ethers-rs community.

The following is a brief overview diagram of the topics covered in this guide.



Bug

> This diagram is incomplete and will undergo continuous changes.

Start a new project

To set up a new project with ethers-rs, you will need to install the Rust programming language toolchain and the Cargo package manager on your system.

1. Install Rust by following the instructions at https://www.rust-lang.org/tools/install.
2. Once Rust is installed, create a new Rust project by running the following command:

```
cargo new my-project
```

   This will create a new directory called my-project with the necessary files for a new Rust project.

3. Navigate to the project directory and add ethers-rs as a dependency in your `Cargo.toml` file:

```
[dependencies]
ethers = "2.0"
# Ethers' async features rely upon the Tokio async runtime.
tokio = { version = "1", features = ["macros"] }
# Flexible concrete Error Reporting type built on std::error::Error with customizable Reports
eyre = "0.6"
```

   If you want to make experiments and/or play around with early ethers-rs features link our GitHub repo in the `Cargo.toml`.

```
[dependencies]
ethers = { git = "https://github.com/gakonst/ethers-rs" }

# Use the "branch" attribute to specify a branch other than master
[dependencies]
ethers = { git = "https://github.com/gakonst/ethers-rs", branch = "branch-name" }

# You can specify a tag or commit hash with the "rev" attribute
[dependencies]
ethers = { git = "https://github.com/gakonst/ethers-rs", rev = "84dda78" }
```

Note: using a Git repository as a dependency is generally not recommended

for production projects, as it can make it difficult to ensure that you are using a specific and stable version of the dependency. It is usually better to specify a version number or range to ensure that your project is reproducible.

Enable transports

Ethers-rs enables interactions with Ethereum nodes through different "transport" types, or communication protocols. The following transport types are currently supported by ethers.rs:

- **HTTP(S):** The HTTP(S) transport is used to communicate with Ethereum nodes over the HTTP or HTTPS protocols. This is the most common way to interact with Ethereum nodes. If you are looking to connect to a HTTPS endpoint, then you need to enable the `rustls` or `openssl` features:

```toml
[dependencies]
ethers = { version = "2.0", features = ["rustls"] }
```

- **WebSocket:** The WebSocket transport is used to communicate with Ethereum nodes over the WebSocket protocol, which is a widely-supported standard for establishing a bi-directional communication channel between a client and a server. This can be used for a variety of purposes, including receiving real-time updates from an Ethereum node, or submitting transactions to the Ethereum network. Websockets support is turned on via the feature-flag ws:

```toml
[dependencies]
ethers = { version = "2.0", features = ["ws"] }
```

- **IPC (Interprocess Communication):** The IPC transport is used to communicate with a local Ethereum node using the IPC protocol, which is a way for processes to communicate with each other on a single computer. This is commonly used in Ethereum development to allow applications to communicate with a local Ethereum node, such as geth or parity. IPC support is turned on via the feature-flag `ipc`:

```toml
[dependencies]
ethers = { version = "2.0", features = ["ipc"] }
```

Connect to an Ethereum node

Ethers-rs allows application to connect the blockchain using web3 providers. Providers act as an interface between applications and an Ethereum node, allowing you to send requests and receive responses via JSON-RPC messages.

Some common actions you can perform using a provider include:

- Getting the current block number
- Getting the balance of an Ethereum address
- Sending a transaction to the blockchain
- Calling a smart contract function
- Subscribe logs and smart contract events
- Getting the transaction history of an address

Providers are an important part of web3 libraries because they allow you to easily interact with the Ethereum blockchain without having to manage the underlying connection to the node yourself.

Code below shows a basic setup to connect a provider to a node:

```rust
// The `prelude` module provides a convenient way to import a number
// of common dependencies at once. This can be useful if you are working
// with multiple parts of the library and want to avoid having
// to import each dependency individually.
use ethers::prelude::*;

const RPC_URL: &str = "https://eth.llamarpc.com";

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let provider = Provider::<Http>::try_from(RPC_URL)?;
    let block_number: U64 = provider.get_block_number().await?;
    println!("{block_number}");

    Ok(())
}
```

Providers

A Provider is an abstraction of a connection to the Ethereum network, providing a concise, consistent interface to standard Ethereum node functionality.

This is achieved through the `Middleware` trait, which provides the interface for the Ethereum JSON-RPC API and other helpful methods, explained in more detail in the Middleware chapter, and the `Provider` struct, which implements `Middleware`.

Data transports

A `Provider` wraps a generic data transport `P`, through which all JSON-RPC API calls are routed.

Ethers provides concrete transport implementations for HTTP, WebSockets, and IPC, as well as higher level transports which wrap a single or multiple transports. Of course, it is also possible to define custom data transports.

Transports implement the `JsonRpcClient` trait, which defines a `request` method, used for sending data to the underlying Ethereum node using JSON-RPC.

Transports can optionally implement the `PubsubClient` trait, if they support the Publish-subscribe pattern, like `Websockets` and `IPC`. This is a supertrait of `JsonRpcClient`. It defines the `subscribe` and `unsubscribe` methods.

The Provider type

This is the definition of the `Provider` type:

```
#[derive(Clone, Debug)]
pub struct Provider<P> {
    inner: P,
    ens: Option<Address>,
    interval: Option<Duration>,
    from: Option<Address>,
    node_client: Arc<Mutex<Option<NodeClient>>>,
}
```

- `inner`: stores the generic data transport, which sends the requests;
- `ens`: optional override for the default ENS registry address;
- `interval`: optional value that defines the polling interval for `watch_*` streams;

- `from` : optional address that sets a default `from` address when constructing calls and transactions;
- `node_client` : the type of node the provider is connected to, like Geth, Erigon, etc.

Now that you have a basis for what the `Provider` type actually is, the next few sections will walk through each implementation of the `Provider` , starting with the HTTP provider.

Http

The `Http` provider establishes an HTTP connection with a node, allowing you to send RPC requests to the node to fetch data, simulate calls, send transactions and much more.

Initializing an Http Provider

Lets take a quick look at few ways to create a new `Http` provider. One of the easiest ways to initialize a new `Provider<Http>` is by using the `TryFrom` trait's `try_from` method.

```
use ethers::providers::{Http, Middleware, Provider};

#[tokio::main]
async fn main() -> eyre::Result<()> {
    // Initialize a new Http provider
    let rpc_url = "https://eth.llamarpc.com";
    let provider = Provider::try_from(rpc_url)?;

    Ok(())
}
```

The `Http` provider also supplies a way to initialize a new authorized connection.

```
use ethers::providers::{Authorization, Http};
use url::Url;

#[tokio::main]
async fn main() -> eyre::Result<()> {
    // Initialize a new HTTP Client with authentication
    let url = Url::parse("http://localhost:8545")?;
    let provider = Http::new_with_auth(url, Authorization::basic("admin",
"good_password"));

    Ok(())
}
```

Additionally, you can initialize a new provider with your own custom `reqwest::Client`.

```
use ethers::providers::Http;
use url::Url;

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let url = Url::parse("http://localhost:8545")?;
    let client = reqwest::Client::builder().build()?;
    let provider = Http::new_with_client(url, client);

    Ok(())
}
```

Basic Usage

Now that you have successfully established an Http connection with the node, you can use any of the methods provided by the `Middleware` trait. In the code snippet below, the provider is used to get the chain id, current block number and the content of the node's mempool.

```
use ethers::providers::{Http, Middleware, Provider};

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let rpc_url = "https://eth.llamarpc.com";
    let provider = Provider::try_from(rpc_url)?;

    let chain_id = provider.get_chainid().await?;
    let block_number = provider.get_block_number().await?;
    let tx_pool_content = provider.txpool_content().await?;

    Ok(())
}
```

You can also use the provider to interact with smart contracts. The snippet below uses the provider to establish a new instance of a UniswapV2Pool and uses the `get_reserves()` method from the smart contract to fetch the current state of the pool's reserves.

```rust
use ethers::{
    prelude::abigen,
    providers::{Http, Provider},
    types::Address,
};
use std::sync::Arc;

abigen!(
    IUniswapV2Pair,
    "[function getReserves() external view returns (uint112 reserve0, uint112
reserve1, uint32 blockTimestampLast)]"
);

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let rpc_url = "https://eth.llamarpc.com";
    let provider = Arc::new(Provider::try_from(rpc_url)?);

    // Initialize a new instance of the Weth/Dai Uniswap V2 pair contract
    let pair_address: Address =
"0xA478c2975Ab1Ea89e8196811F51A7B7Ade33eB11".parse()?;
    let uniswap_v2_pair = IUniswapV2Pair::new(pair_address, provider);

    // Use the get_reserves() function to fetch the pool reserves
    let (reserve_0, reserve_1, block_timestamp_last) =
        uniswap_v2_pair.get_reserves().call().await?;

    Ok(())
}
```

This example is a little more complicated, so let's walk through what is going on. The `IUniswapV2Pair` is a struct that is generated from the `abigen!()` macro. The `IUniswapV2Pair::new()` function is used to create a new instance of the contract, taking in an `Address` and an `Arc<M>` as arguments, where `M` is any type that implements the `Middleware` trait. Note that the provider is wrapped in an `Arc` when being passed into the `new()` function.

It is very common to wrap a provider in an `Arc` to share the provider across threads. Let's look at another example where the provider is used asynchronously across two tokio threads. In the next example, a new provider is initialized and used to asynchronously fetch the number of Ommer blocks from the most recent block, as well as the previous block.

```
use ethers::providers::{Http, Middleware, Provider};
use std::sync::Arc;

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let rpc_url = "https://eth.llamarpc.com";
    let provider = Arc::new(Provider::try_from(rpc_url)?);

    let current_block_number = provider.get_block_number().await?;
    let prev_block_number = current_block_number - 1;

    // Clone the Arc<Provider> and pass it into a new thread to get the uncle
count of the current block
    let provider_1 = provider.clone();
    let task_0 =
        tokio::spawn(async move {
provider_1.get_uncle_count(current_block_number).await });

    // Spin up a new thread to get the uncle count of the previous block
    let task_1 = tokio::spawn(async move {
provider.get_uncle_count(prev_block_number).await });

    // Wait for the tasks to finish
    for task in [task_0, task_1] {
        if let Ok(uncle_count) = task.await? {
            println!("Success!");
        }
    }

    Ok(())
}
```

Before heading to the next chapter, feel free to check out the docs for the `Http` provider. Keep in mind that we will cover advanced usage of providers at the end of this chapter. Now that we have the basics covered, let's move on to the next provider, Websockets!

WebSocket provider

The Ws provider allows you to send JSON-RPC requests and receive responses over WebSocket connections. The WS provider can be used with any Ethereum node that supports WebSocket connections. This allows programs interact with the network in real-time without the need for HTTP polling for things like new block headers and filter logs. Ethers-rs has support for WebSockets via Tokio. Make sure that you have the “ws” and “rustls” / “openssl” features enabled in your project's toml file if you wish to use WebSockets.

Initializing a WS Provider

Lets look at a few ways to create a new `WS` provider. Below is the most straightforward way to initialize a new `Ws` provider.

```
use ethers::providers::{Provider, Ws};

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let provider = Provider::<Ws>::connect("wss://...").await?;
    Ok(())
}
```

Similar to the other providers, you can also establish an authorized connection with a node via websockets.

```
use ethers::providers::{Authorization, Provider, Ws};

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let url = "wss://...";
    let auth = Authorization::basic("username", "password");
    let provider = Provider::<Ws>::connect_with_auth(url, auth).await?;
    Ok(())
}
```

Usage

The `Ws` provider allows a user to send requests to the node just like the other providers. In addition to these methods, the `Ws` provider can also subscribe to new logs and events, watch transactions in the mempool and other types of data streams from the node.

In the snippet below, a new `Ws` provider is used to subscribe to new pending transactions in the mempool as well as new block headers in two separate threads.

```
//! The Ws transport allows you to send JSON-RPC requests and receive
responses over
//! [WebSocket](https://en.wikipedia.org/wiki/WebSocket).
//!
//! This allows to interact with the network in real-time without the need
for HTTP
//! polling.

use ethers::prelude::*;

const WSS_URL: &str = "wss://mainnet.infura.io/ws/v3
/c60b0bb42f8a4c6481ecd229eddaca27";

#[tokio::main]
async fn main() -> eyre::Result<()> {
    // A Ws provider can be created from a ws(s) URI.
    // In case of wss you must add the "rustls" or "openssl" feature
    // to the ethers library dependency in `Cargo.toml`.
    let provider = Provider::<Ws>::connect(WSS_URL).await?;

    let mut stream = provider.subscribe_blocks().await?.take(1);
    while let Some(block) = stream.next().await {
        println!("{:?}", block.hash);
    }

    Ok(())
}
```

IPC provider

The IPC (Inter-Process Communication) transport allows our program to communicate with a node over a local Unix domain socket or Windows named pipe.

Using the IPC transport allows the ethers library to send JSON-RPC requests to the Ethereum client and receive responses, without the need for a network connection or HTTP server. This can be useful for interacting with a local Ethereum node that is running on the same network. Using IPC is faster than RPC, however you will need to have a local node that you can connect to.

Initializing an Ipc Provider

Below is an example of how to initialize a new Ipc provider.

```
use ethers::providers::Provider;

#[tokio::main]
async fn main() -> eyre::Result<()> {
    // Using a UNIX domain socket: `/path/to/ipc`
    #[cfg(unix)]
    let provider = Provider::connect_ipc("~/.ethereum/geth.ipc").await?;

    // Using a Windows named pipe: `\\<machine_address>\pipe\<pipe_name>`
    #[cfg(windows)]
    let provider = Provider::connect_ipc(r"\\.\pipe\geth").await?;

    Ok(())
}
```

Usage

The `Ipc` provider implements both `JsonRpcClient` and `PubsubClient` , just like `Ws` .

In this example, we monitor the `WETH/USDC` UniswapV2 pair reserves and print when they have changed.

```rust
//! The IPC (Inter-Process Communication) transport allows our program to
communicate
//! with a node over a local [Unix domain socket](https://en.wikipedia.org
/wiki/Unix_domain_socket)
//! or [Windows named pipe](https://learn.microsoft.com/en-us/windows/win32
/ipc/named-pipes).
//!
//! It functions much the same as a Ws connection.

use ethers::prelude::*;
use std::sync::Arc;

abigen!(
    IUniswapV2Pair,
    "[function getReserves() external view returns (uint112 reserve0, uint112
reserve1, uint32 blockTimestampLast)]"
);

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let provider = Provider::connect_ipc("~/.ethereum/geth.ipc").await?;
    let provider = Arc::new(provider);

    let pair_address: Address =
"0xb4e16d0168e52d35cacd2c6185b44281ec28c9dc".parse()?;
    let weth_usdc = IUniswapV2Pair::new(pair_address, provider.clone());

    let block = provider.get_block_number().await?;
    println!("Current block: {block}");

    let mut initial_reserves = weth_usdc.get_reserves().call().await?;
    println!("Initial reserves: {initial_reserves:?}");

    let mut stream = provider.subscribe_blocks().await?;
    while let Some(block) = stream.next().await {
        println!("New block: {:?}", block.number);

        let reserves = weth_usdc.get_reserves().call().await?;
        if reserves != initial_reserves {
            println!("Reserves changed: old {initial_reserves:?} - new
{reserves:?}");
            initial_reserves = reserves;
        }
    }

    Ok(())
}
```

Mock provider

```
//! `MockProvider` is a mock Ethereum provider that can be used for testing
purposes.
//! It allows to simulate Ethereum state and behavior, by explicitly
instructing
//! provider's responses on client requests.
//!
//! This can be useful for testing code that relies on providers without the
need to
//! connect to a real network or spend real Ether. It also allows to test
code in a
//! deterministic manner, as you can control the state and behavior of the
provider.
//!
//! In these examples we use the common Arrange, Act, Assert (AAA) test
approach.
//! It is a useful pattern for well-structured, understandable and
maintainable tests.

use ethers::prelude::*;

#[tokio::main]
async fn main() -> eyre::Result<()> {
    mocked_block_number().await?;
    mocked_provider_dependency().await?;
    Ok(())
}

async fn mocked_block_number() -> eyre::Result<()> {
    // Arrange
    let mock = MockProvider::new();
    let block_num_1 = U64::from(1);
    let block_num_2 = U64::from(2);
    let block_num_3 = U64::from(3);
    // Mock responses are organized in a stack (LIFO)
    mock.push(block_num_1)?;
    mock.push(block_num_2)?;
    mock.push(block_num_3)?;

    // Act
    let ret_block_3: U64 = JsonRpcClient::request(&mock, "eth_blockNumber",
()).await?;
    let ret_block_2: U64 = JsonRpcClient::request(&mock, "eth_blockNumber",
()).await?;
    let ret_block_1: U64 = JsonRpcClient::request(&mock, "eth_blockNumber",
()).await?;

    // Assert
    assert_eq!(block_num_1, ret_block_1);
    assert_eq!(block_num_2, ret_block_2);
    assert_eq!(block_num_3, ret_block_3);

    Ok(())
}

/// Here we test the `OddBlockOracle` struct (defined below) that relies
/// on a Provider to perform some logics.
```

```rust
/// The Provider reference is expressed with trait bounds, enforcing lose
coupling,
/// maintainability and testability.
async fn mocked_provider_dependency() -> eyre::Result<()> {
    // Arrange
    let (provider, mock) = crate::Provider::mocked();
    mock.push(U64::from(2))?;

    // Act
    // Let's mock the provider dependency (we ❤️ DI!) then ask for the answer
    let oracle = OddBlockOracle::new(provider);
    let answer: bool = oracle.is_odd_block().await?;

    // Assert
    assert!(answer);
    Ok(())
}

struct OddBlockOracle<P> {
    provider: Provider<P>,
}

impl<P> OddBlockOracle<P>
where
    P: JsonRpcClient,
{
    fn new(provider: Provider<P>) -> Self {
        Self { provider }
    }

    /// We want to test this!
    async fn is_odd_block(&self) -> eyre::Result<bool> {
        let block: U64 = self.provider.get_block_number().await?;
        Ok(block % 2 == U64::zero())
    }
}
```

Quorum provider

```
//! The `QuorumProvider` sends a request to multiple backends and only
returns a value
//! if the configured `Quorum` was reached.

use ethers::{
    core::utils::Anvil,
    providers::{Http, Middleware, Provider, Quorum, QuorumProvider,
WeightedProvider, Ws},
};
use eyre::Result;
use std::{str::FromStr, time::Duration};

#[tokio::main]
async fn main() -> Result<()> {
    let anvil = Anvil::new().spawn();

    // create a quorum provider with some providers
    let quorum = QuorumProvider::dyn_rpc()

.add_provider(WeightedProvider::new(Box::new(Http::from_str(&anvil.endpoint()
)?)))
        .add_provider(WeightedProvider::with_weight(
            Box::new(Ws::connect(anvil.ws_endpoint()).await?),
            2,
        ))
        .add_provider(WeightedProvider::with_weight(
            Box::new(Ws::connect(anvil.ws_endpoint()).await?),
            2,
        ))
        // the quorum provider will yield the response if >50% of the
weighted inner provider
        // returned the same value
        .quorum(Quorum::Majority)
        .build();

    let provider =
Provider::quorum(quorum).interval(Duration::from_millis(10u64));

    let _ = provider.get_accounts().await?;

    Ok(())
}
```

Retry client

```
//! The RetryClient is a type that wraps around a JsonRpcClient and
automatically retries failed
//! requests using an exponential backoff and filtering based on a
RetryPolicy. It presents as a
//! JsonRpcClient, but with additional functionality for retrying requests.
//!
//! The RetryPolicy can be customized for specific applications and
endpoints, mainly to handle
//! rate-limiting errors. In addition to the RetryPolicy, errors caused by
connectivity issues such
//! as timed out connections or responses in the 5xx range can also be
retried separately.

use ethers::prelude::*;
use reqwest::Url;
use std::time::Duration;

const RPC_URL: &str = "https://eth.llamarpc.com";

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let provider = Http::new(Url::parse(RPC_URL)?);

    let client = RetryClientBuilder::default()
        .rate_limit_retries(10)
        .timeout_retries(3)
        .initial_backoff(Duration::from_millis(500))
        .build(provider,
Box::<ethers::providers::HttpRateLimitRetryPolicy>::default());

    // Send a JSON-RPC request for the latest block
    let block_num = "latest".to_string();
    let txn_details = false;
    let params = (block_num, txn_details);

    let block: Block<H256> =
        JsonRpcClient::request(&client, "eth_getBlockByNumber",
params).await?;

    println!("{block:?}");

    Ok(())
}
```

RW provider

```
//! The RwClient wraps two data transports: the first is used for read
operations, and the second
//! one is used for write operations, that consume gas like sending
transactions.

use ethers::{prelude::*, utils::Anvil};
use url::Url;

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let anvil = Anvil::new().spawn();

    let http_url = Url::parse(&anvil.endpoint())?;
    let http = Http::new(http_url);

    let ws = Ws::connect(anvil.ws_endpoint()).await?;

    let _provider = Provider::rw(http, ws);

    Ok(())
}
```

Custom data transport

As we've previously seen, a transport must implement `JsonRpcClient`, and can also optionally implement `PubsubClient`.

Let's see how we can create a custom data transport by implementing one that stores either a `Ws` or an `Ipc` transport:

```rust
//! Create a custom data transport to use with a Provider.

use async_trait::async_trait;
use ethers::{core::utils::Anvil, prelude::*};
use serde::{de::DeserializeOwned, Serialize};
use std::fmt::Debug;
use thiserror::Error;
use url::Url;

/// First we must create an error type, and implement [`From`] for
/// [`ProviderError`].
///
/// Here we are using [`thiserror`](https://docs.rs/thiserror) to wrap
/// [`WsClientError`] and [`IpcError`].
///
/// This also provides a conversion implementation ([`From`]) for both, so we
/// can use the [question mark operator](https://doc.rust-lang.org/rust-by-
example/std/result/question_mark.html)
/// later on in our implementations.
#[derive(Debug, Error)]
pub enum WsOrIpcError {
    #[error(transparent)]
    Ws(#[from] WsClientError),

    #[error(transparent)]
    Ipc(#[from] IpcError),
}

/// In order to use our `WsOrIpcError` in the RPC client, we have to
implement
/// this trait.
///
/// [`RpcError`] helps other parts off the stack get access to common
provider
/// error cases. For example, any RPC connection may have a `serde_json`
error,
/// so we want to make those easily accessible, so we implement
/// `as_serde_error()`
///
/// In addition, RPC requests may return JSON errors from the node,
describing
/// why the request failed. In order to make these accessible, we implement
/// `as_error_response()`.
impl RpcError for WsOrIpcError {
    fn as_error_response(&self) -> Option<&ethers::providers::JsonRpcError> {
        match self {
            WsOrIpcError::Ws(e) => e.as_error_response(),
            WsOrIpcError::Ipc(e) => e.as_error_response(),
        }
    }

    fn as_serde_error(&self) -> Option<&serde_json::Error> {
        match self {
            WsOrIpcError::Ws(WsClientError::JsonError(e)) => Some(e),
            WsOrIpcError::Ipc(IpcError::JsonError(e)) => Some(e),
            _ => None,
```

```
            }
        }
    }

    /// This implementation helps us convert our Error to the library's
    /// [`ProviderError`] so that we can use the `?` operator
    impl From<WsOrIpcError> for ProviderError {
        fn from(value: WsOrIpcError) -> Self {
            Self::JsonRpcClientError(Box::new(value))
        }
    }

    /// Next, we create our transport type, which in this case will be an enum
    that contains
    /// either [`Ws`] or [`Ipc`].
    #[derive(Clone, Debug)]
    enum WsOrIpc {
        Ws(Ws),
        Ipc(Ipc),
    }

    // We implement a convenience "constructor" method, to easily initialize the
    transport.
    // This will connect to [`Ws`] if it's a valid [URL](url::Url), otherwise
    it'll
    // default to [`Ipc`].
    impl WsOrIpc {
        pub async fn connect(s: &str) -> Result<Self, WsOrIpcError> {
            let this = match Url::parse(s) {
                Ok(url) => Self::Ws(Ws::connect(url).await?),
                Err(_) => Self::Ipc(Ipc::connect(s).await?),
            };
            Ok(this)
        }
    }

    // Next, the most important step: implement [`JsonRpcClient`].
    //
    // For this implementation, we simply delegate to the wrapped transport and
    return the
    // result.
    //
    // Note that we are using [`async-trait`](https://docs.rs/async-trait) for
    asynchronous
    // functions in traits, as this is not yet supported in stable Rust; see:
    // <https://blog.rust-lang.org/inside-rust/2022/11/17/async-fn-in-trait-
    nightly.html>
    #[async_trait]
    impl JsonRpcClient for WsOrIpc {
        type Error = WsOrIpcError;

        async fn request<T, R>(&self, method: &str, params: T) -> Result<R,
    Self::Error>
        where
            T: Debug + Serialize + Send + Sync,
            R: DeserializeOwned + Send,
        {
```

```
        let res = match self {
            Self::Ws(ws) => JsonRpcClient::request(ws, method,
params).await?,
            Self::Ipc(ipc) => JsonRpcClient::request(ipc, method,
params).await?,
        };
        Ok(res)
    }
}

// We can also implement [`PubsubClient`], since both `Ws` and `Ipc`
implement it, by
// doing the same as in the `JsonRpcClient` implementation above.
impl PubsubClient for WsOrIpc {
    // Since both `Ws` and `Ipc`'s `NotificationStream` associated type is
the same,
    // we can simply return one of them.
    // In case they differed, we would have to create a
`WsOrIpcNotificationStream`,
    // similar to the error type.
    type NotificationStream = <Ws as PubsubClient>::NotificationStream;

    fn subscribe<T: Into<U256>>(&self, id: T) ->
Result<Self::NotificationStream, Self::Error> {
        let stream = match self {
            Self::Ws(ws) => PubsubClient::subscribe(ws, id)?,
            Self::Ipc(ipc) => PubsubClient::subscribe(ipc, id)?,
        };
        Ok(stream)
    }

    fn unsubscribe<T: Into<U256>>(&self, id: T) -> Result<(), Self::Error> {
        match self {
            Self::Ws(ws) => PubsubClient::unsubscribe(ws, id)?,
            Self::Ipc(ipc) => PubsubClient::unsubscribe(ipc, id)?,
        };
        Ok(())
    }
}

#[tokio::main]
async fn main() -> eyre::Result<()> {
    // Spawn Anvil
    let anvil = Anvil::new().block_time(1u64).spawn();

    // Connect to our transport
    let transport = WsOrIpc::connect(&anvil.ws_endpoint()).await?;

    // Wrap the transport in a provider
    let provider = Provider::new(transport);

    // Now we can use our custom transport provider like normal
    let block_number = provider.get_block_number().await?;
    println!("Current block: {block_number}");

    let mut subscription = provider.subscribe_blocks().await?.take(3);
    while let Some(block) = subscription.next().await {
```

```
            println!("New block: {:?}", block.number);
        }

        Ok(())
    }
```

Advanced Usage

`CallBuilder`

The `CallBuilder` is an enum to help create complex calls. `CallBuilder` implements `RawCall` methods for overriding parameters to the `eth_call` rpc method.

Lets take a quick look at how to use the `CallBuilder`.

```
use ethers::{
    providers::{Http, Provider},
    types::{TransactionRequest, H160},
    utils::parse_ether,
};
use std::sync::Arc;

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let rpc_url = "https://eth.llamarpc.com";
    let provider: Arc<Provider<Http>> = Arc::new(Provider::
<Http>::try_from(rpc_url)?);

    let from_adr: H160 =
"0x6fC21092DA55B392b045eD78F4732bff3C580e2c".parse()?;
    let to_adr: H160 = "0x000000000000000000000000000000000000dead".parse()?;
    let val = parse_ether(1u64)?;

    let tx = TransactionRequest::default()
        .from(from_adr)
        .to(to_adr)
        .value(val)
        .into();

    let result = provider.call_raw(&tx).await?;

    Ok(())
}
```

First, we initialize a new provider and create a transaction that sends `1 ETH` from one address to another. Then we use `provider.call_raw()`, which returns a `CallBuilder`. From here, we can use `await` to send the call to the node with exactly the same behavior as simply using `provider.call()`. We can also override the parameters sent to the node by using the methods provided by the `RawCall` trait. These methods allow you to set the block number that the call should execute on as well as give you access to the state override set.

Here is an example with the exact same raw call, but executed on the previous block.

```
use ethers::{
    providers::{call_raw::RawCall, Http, Middleware, Provider},
    types::{BlockId, TransactionRequest, H160},
    utils::parse_ether,
};
use std::sync::Arc;

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let rpc_url = "https://eth.llamarpc.com";
    let provider: Arc<Provider<Http>> = Arc::new(Provider::
<Http>::try_from(rpc_url)?);

    let from_adr: H160 =
"0x6fC21092DA55B392b045eD78F4732bff3C580e2c".parse()?;
    let to_adr: H160 = "0x000000000000000000000000000000000000dead".parse()?;
    let val = parse_ether(1u64)?;

    let tx = TransactionRequest::default()
        .from(from_adr)
        .to(to_adr)
        .value(val)
        .into();

    let previous_block_number: BlockId = (provider.get_block_number().await?
- 1).into();
    let result = provider.call_raw(&tx).block(previous_block_number).await?;

    Ok(())
}
```

Let's look at how to use the state override set. In short, the state override set is an optional address-to-state mapping, where each entry specifies some state to be ephemerally overridden prior to executing the call. The state override set allows you to override an account's balance, an account's nonce, the code at a given address, the entire state of an account's storage or an individual slot in an account's storage. Note that the state override set is not a default feature and is not available on every node.

```rust
use ethers::{
    providers::{
        call_raw::RawCall,
        Http, Provider,
    },
    types::{TransactionRequest, H160, U256, U64},
    utils::parse_ether,
};
use std::sync::Arc;

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let rpc_url = "https://eth.llamarpc.com";
    let provider: Arc<Provider<Http>> = Arc::new(Provider::
<Http>::try_from(rpc_url)?);

    let from_adr: H160 =
"0x6fC21092DA55B392b045eD78F4732bff3C580e2c".parse()?;
    let to_adr: H160 = "0x000000000000000000000000000000000000dead".parse()?;
    let val = parse_ether(1u64)?;

    let tx = TransactionRequest::default()
        .from(from_adr)
        .to(to_adr)
        .value(val)
        .into();

    let mut state = spoof::State::default();

    // Set the account balance to max u256
    state.account(from_adr).balance(U256::MAX);
    // Set the nonce to 0
    state.account(from_adr).nonce(U64::zero());

    let result = provider.call_raw(&tx).state(&state).await?;

    Ok(())
}
```

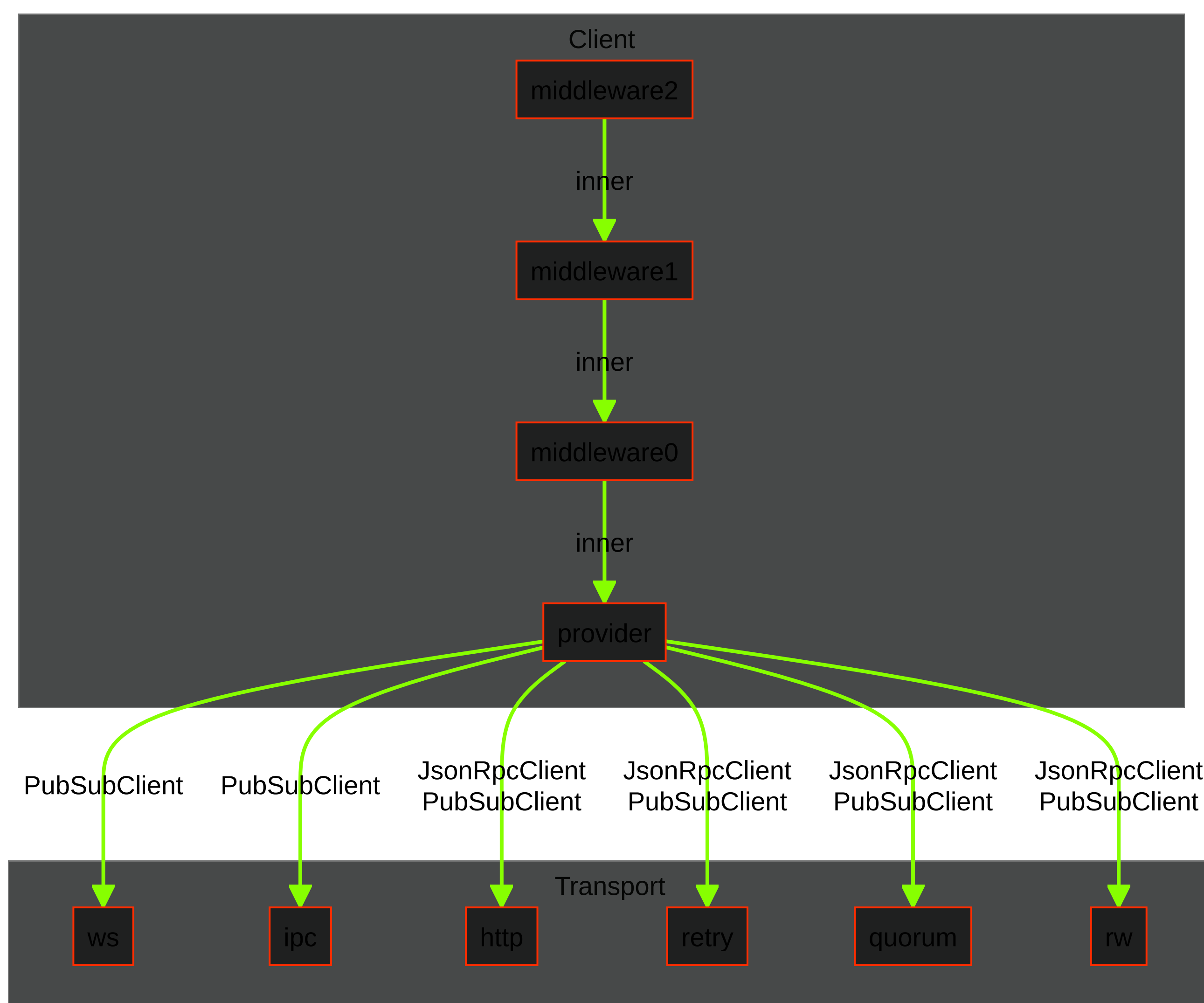
In this example, the account balance and nonce for the `from_adr` is overridden. The state override set is a very powerful tool that you can use to simulate complicated transactions without undergoing any actual state changes.

Middlewares

In ethers-rs, middleware is a way to customize the behavior of certain aspects of the library by injecting custom logic into the process of interacting with the Ethereum JSON-RPC API. Middlewares act in a chain of responsibility and can be combined to achieve the desired behavior.

The library allows developers to create custom middlewares, going beyond standard operations and unlocking powerful use cases.

A JSON-RPC client instance can be constructed as a stack of middlewares, backed by a common instance of `Provider` of one specific type among `JsonRpcClient` and `PubSubClient`.



The following middlewares are currently supported:

- Gas Escalator: Avoids transactions being stucked in the mempool, by bumping the gas price in background.
- Gas Oracle: Allows retrieving the current gas price from common RESTful APIs,

instead of retrieving it from blocks.

- Nonce Manager: Manages nonces of transactions locally, without waiting for them to hit the mempool.
- Policy: Allows to define rules or policies that should be followed when submitting JSON-RPC API calls.
- Signer: Signs transactions locally, with a private key or a hardware wallet.
- Time lag: Poses a block delay on JSON-RPC interactions, allowing to shift the block number back of a predefined offset.
- Transformer: Allows intercepting and transforming a transaction to be broadcasted via a proxy wallet.

Middleware builder

```rust
use ethers::{
    core::types::BlockNumber,
    middleware::{
        gas_escalator::{Frequency, GasEscalatorMiddleware,
GeometricGasPrice},
        gas_oracle::{GasNow, GasOracleMiddleware},
        MiddlewareBuilder, NonceManagerMiddleware, SignerMiddleware,
    },
    providers::{Http, Middleware, Provider},
    signers::{LocalWallet, Signer},
};
use std::convert::TryFrom;

const RPC_URL: &str = "https://eth.llamarpc.com";
const SIGNING_KEY: &str =
"fdb33e2105f08abe41a8ee3b758726a31abdd57b7a443f470f23efce853af169";

/// In ethers-rs, middleware is a way to customize the behavior of certain
aspects of the library by
/// injecting custom logic into the process of sending transactions and
interacting with contracts
/// on the Ethereum blockchain. The MiddlewareBuilder trait provides a way to
define a chain of
/// middleware that will be called at different points in this process,
allowing you to customize
/// the behavior of the Provider based on your needs.
#[tokio::main]
async fn main() {
    builder_example().await;
    builder_example_raw_wrap().await;
}

async fn builder_example() {
    let signer = SIGNING_KEY.parse::<LocalWallet>().unwrap();
    let address = signer.address();
    let escalator = GeometricGasPrice::new(1.125, 60_u64, None::<u64>);
    let gas_oracle = GasNow::new();

    let provider = Provider::<Http>::try_from(RPC_URL)
        .unwrap()
        .wrap_into(|p| GasEscalatorMiddleware::new(p, escalator,
Frequency::PerBlock))
        .gas_oracle(gas_oracle)
        .with_signer(signer)
        .nonce_manager(address); // Outermost layer

    match provider.get_block(BlockNumber::Latest).await {
        Ok(Some(block)) => println!("{:?}", block.number),
        _ => println!("Unable to get latest block"),
    }
}

async fn builder_example_raw_wrap() {
    let signer = SIGNING_KEY.parse::<LocalWallet>().unwrap();
    let address = signer.address();
    let escalator = GeometricGasPrice::new(1.125, 60_u64, None::<u64>);
```

```
    let provider = Provider::<Http>::try_from(RPC_URL)
        .unwrap()
        .wrap_into(|p| GasEscalatorMiddleware::new(p, escalator,
Frequency::PerBlock))
        .wrap_into(|p| SignerMiddleware::new(p, signer))
        .wrap_into(|p| GasOracleMiddleware::new(p, GasNow::new()))
        .wrap_into(|p| NonceManagerMiddleware::new(p, address)); // Outermost
layer

    match provider.get_block(BlockNumber::Latest).await {
        Ok(Some(block)) => println!("{:?}", block.number),
        _ => println!("Unable to get latest block"),
    }
}
```

Create custom middleware

```
use async_trait::async_trait;
use ethers::{
    core::{
        types::{transaction::eip2718::TypedTransaction, BlockId,
TransactionRequest, U256},
        utils::{parse_units, Anvil},
    },
    middleware::MiddlewareBuilder,
    providers::{Http, Middleware, MiddlewareError, PendingTransaction,
Provider},
    signers::{LocalWallet, Signer},
};
use thiserror::Error;

/// This example demonstrates the mechanisms for creating custom middlewares
in ethers-rs.
/// The example includes explanations of the process and code snippets to
illustrate the
/// concepts. It is intended for developers who want to learn how to
customize the behavior of
/// ethers-rs providers by creating and using custom middlewares.
///
/// This custom middleware increases the gas value of transactions sent
through an ethers-rs
/// provider by a specified percentage and will be called for each
transaction before it is sent.
/// This can be useful if you want to ensure that transactions have a higher
gas value than the
/// estimated, in order to improve the chances of them not to run out of gas
when landing on-chain.
#[derive(Debug)]
struct GasMiddleware<M> {
    inner: M,
    /// This value is used to raise the gas value before sending transactions
    contingency: U256,
}

/// Contingency is expressed with 4 units
/// e.g.
/// 50% => 1 + 0.5  => 15000
/// 20% => 1 + 0.2  => 12000
/// 1%  => 1 + 0.01 => 10100
const CONTINGENCY_UNITS: usize = 4;

impl<M> GasMiddleware<M>
where
    M: Middleware,
{
    /// Creates an instance of GasMiddleware
    /// `ìnner` the inner Middleware
    /// `perc` This is an unsigned integer representing the percentage
increase in the amount of gas
    /// to be used for the transaction. The percentage is relative to the gas
value specified in the
    /// transaction. Valid contingency values are in range 1..=50. Otherwise
a custom middleware
```

```
    /// error is raised.
    pub fn new(inner: M, perc: u32) -> Result<Self, GasMiddlewareError<M>> {
        let contingency = match perc {
            0 => Err(GasMiddlewareError::TooLowContingency(perc))?,
            51.. => Err(GasMiddlewareError::TooHighContingency(perc))?,
            1..=50 => {
                let decimals = 2;
                let perc = U256::from(perc) * U256::exp10(decimals); // e.g.
50 => 5000
                let one = parse_units(1, CONTINGENCY_UNITS).unwrap();
                let one = U256::from(one);
                one + perc // e.g. 50% => 1 + 0.5 => 10000 + 5000 => 15000
            }
        };

        Ok(Self { inner, contingency })
    }
}

/// Let's implement the `Middleware` trait for our custom middleware.
/// All trait functions are derived automatically, so we just need to
/// override the needed functions.
#[async_trait]
impl<M> Middleware for GasMiddleware<M>
where
    M: Middleware,
{
    type Error = GasMiddlewareError<M>;
    type Provider = M::Provider;
    type Inner = M;

    fn inner(&self) -> &M {
        &self.inner
    }

    /// In this function we bump the transaction gas value by the specified
percentage
    /// This can raise a custom middleware error if a gas amount was not set
for
    /// the transaction.
    async fn send_transaction<T: Into<TypedTransaction> + Send + Sync>(
        &self,
        tx: T,
        block: Option<BlockId>,
    ) -> Result<PendingTransaction<'_, Self::Provider>, Self::Error> {
        let mut tx: TypedTransaction = tx.into();

        let curr_gas: U256 = match tx.gas() {
            Some(gas) => gas.to_owned(),
            None => Err(GasMiddlewareError::NoGasSetForTransaction)?,
        };

        println!("Original transaction gas: {curr_gas:?} wei");
        let units: U256 = U256::exp10(CONTINGENCY_UNITS);
        let raised_gas: U256 = (curr_gas * self.contingency) / units;
        tx.set_gas(raised_gas);
        println!("Raised transaction gas: {raised_gas:?} wei");
```

```
        // Dispatch the call to the inner layer
        self.inner().send_transaction(tx,
block).await.map_err(MiddlewareError::from_err)
    }
}

/// This example demonstrates how to handle errors in custom middlewares. It
shows how to define
/// custom error types, use them in middleware implementations, and how to
propagate the errors
/// through the middleware chain. This is intended for developers who want to
create custom
/// middlewares that can handle and propagate errors in a consistent and
robust way.
#[derive(Error, Debug)]
pub enum GasMiddlewareError<M: Middleware> {
    /// Thrown when the internal middleware errors
    #[error("{0}")]
    MiddlewareError(M::Error),
    /// Specific errors of this GasMiddleware.
    /// Please refer to the `thiserror` crate for
    /// further docs.
    #[error("{0}")]
    TooHighContingency(u32),
    #[error("{0}")]
    TooLowContingency(u32),
    #[error("Cannot raise gas! Gas value not provided for this
transaction.")]
    NoGasSetForTransaction,
}

impl<M: Middleware> MiddlewareError for GasMiddlewareError<M> {
    type Inner = M::Error;

    fn from_err(src: M::Error) -> Self {
        GasMiddlewareError::MiddlewareError(src)
    }

    fn as_inner(&self) -> Option<&Self::Inner> {
        match self {
            GasMiddlewareError::MiddlewareError(e) => Some(e),
            _ => None,
        }
    }
}

#[tokio::main]
async fn main() -> eyre::Result<()> {
    let anvil = Anvil::new().spawn();

    let wallet: LocalWallet = anvil.keys()[0].clone().into();
    let wallet2: LocalWallet = anvil.keys()[1].clone().into();
    let signer = wallet.with_chain_id(anvil.chain_id());

    let gas_raise_perc = 50; // 50%;
    let provider = Provider::<Http>::try_from(anvil.endpoint())?
```

```
        .with_signer(signer)
        .wrap_into(|s| GasMiddleware::new(s, gas_raise_perc).unwrap());

    let gas = 15000;
    let tx =
TransactionRequest::new().to(wallet2.address()).value(10000).gas(gas);

    let pending_tx = provider.send_transaction(tx, None).await?;

    let receipt = pending_tx.await?.ok_or_else(|| eyre::format_err!("tx
dropped from mempool"))?;
    let tx = provider.get_transaction(receipt.transaction_hash).await?;

    println!("Sent tx: {}\n", serde_json::to_string(&tx)?);
    println!("Tx receipt: {}", serde_json::to_string(&receipt)?);

    Ok(())
}
```

Gas escalator

```
use ethers::{
    core::{types::TransactionRequest, utils::Anvil},
    middleware::gas_escalator::*,
    providers::{Http, Middleware, Provider},
};
use eyre::Result;

/// The gas escalator middleware in ethers-rs is designed to automatically
increase the gas cost of
/// transactions if they get stuck in the mempool. This can be useful if you
want to
/// ensure that transactions are processed in a timely manner without having
to manually adjust the
/// gas cost yourself.
#[tokio::main]
async fn main() -> Result<()> {
    let every_secs: u64 = 60;
    let max_price: Option<i32> = None;

    // Linearly increase gas price:
    // Start with `initial_price`, then increase it by fixed amount
`increase_by` every `every_secs`
    // seconds until the transaction gets confirmed. There is an optional
upper limit.
    let increase_by: i32 = 100;
    let linear_escalator = LinearGasPrice::new(increase_by, every_secs,
max_price);
    send_escalating_transaction(linear_escalator).await?;

    // Geometrically increase gas price:
    // Start with `initial_price`, then increase it every 'every_secs'
seconds by a fixed
    // coefficient. Coefficient defaults to 1.125 (12.5%), the minimum
increase for Parity to
    // replace a transaction. Coefficient can be adjusted, and there is an
optional upper limit.
    let coefficient: f64 = 1.125;
    let geometric_escalator = GeometricGasPrice::new(coefficient, every_secs,
max_price);
    send_escalating_transaction(geometric_escalator).await?;

    Ok(())
}

async fn send_escalating_transaction<E>(escalator: E) -> Result<()>
where
    E: GasEscalator + Clone + 'static,
{
    // Spawn local node
    let anvil = Anvil::new().spawn();
    let endpoint = anvil.endpoint();

    // Connect to the node
    let provider = Provider::<Http>::try_from(endpoint)?;
    let provider = GasEscalatorMiddleware::new(provider, escalator,
Frequency::PerBlock);
```

```
    let accounts = provider.get_accounts().await?;
    let from = accounts[0];
    let to = accounts[1];
    let tx = TransactionRequest::new().from(from).to(to).value(1000);

    // Bumps the gas price until transaction gets mined
    let pending_tx = provider.send_transaction(tx, None).await?;
    let receipt = pending_tx.await?;

    println!("{receipt:?}");

    Ok(())
}
```

Gas oracle

```
use ethers::{
    core::types::Chain,
    etherscan::Client,
    middleware::gas_oracle::{
        BlockNative, Etherscan, GasCategory, GasNow, GasOracle, Polygon,
ProviderOracle,
    },
    providers::{Http, Provider},
};

/// In Ethereum, the "gas" of a transaction refers to the amount of
computation required to execute
/// the transaction on the blockchain. Gas is typically measured in units of
"gas," and the cost of
/// a transaction is determined by the amount of gas it consumes.
///
/// A "gas oracle" is a tool or service that provides information about the
current price of gas on
/// the Ethereum network. Gas oracles are often used to help determine the
appropriate amount of gas
/// to include in a transaction, in order to ensure that it will be processed
in a timely manner
/// without running out of gas.
///
/// Ethers-rs includes a feature called "gas oracle middleware" that allows
you to customize the
/// behavior of the library when it comes to determining the gas cost of
transactions.
#[tokio::main]
async fn main() {
    blocknative().await;
    etherscan().await;
    gas_now().await;
    polygon().await;
    provider_oracle().await;
    //etherchain().await; // FIXME: Etherchain URL is broken (Http 404)
}

async fn blocknative() {
    let api_key: Option<String> = std::env::var("BLOCK_NATIVE_API_KEY").ok();
    let oracle = BlockNative::new(api_key).category(GasCategory::Fastest);
    match oracle.fetch().await {
        Ok(gas_price) => println!("[Blocknative]: Gas price is
{gas_price:?}"),
        Err(e) => panic!("[Blocknative]: Cannot estimate gas: {e:?}"),
    }
}

async fn etherscan() {
    let client = Client::new_from_opt_env(Chain::Mainnet).unwrap();
    let oracle = Etherscan::new(client).category(GasCategory::Fast);
    match oracle.fetch().await {
        Ok(gas_price) => println!("[Etherscan]: Gas price is {gas_price:?}"),
        Err(e) => panic!("[Etherscan]: Cannot estimate gas: {e:?}"),
    }
}
```

```rust
async fn gas_now() {
    let oracle = GasNow::new().category(GasCategory::Fast);
    match oracle.fetch().await {
        Ok(gas_price) => println!("[GasNow]: Gas price is {gas_price:?}"),
        Err(e) => panic!("[GasNow]: Cannot estimate gas: {e:?}"),
    }
}

async fn polygon() {
    let chain = Chain::Polygon;
    if let Ok(oracle) = Polygon::new(chain) {
        match oracle.category(GasCategory::SafeLow).fetch().await {
            Ok(gas_price) => println!("[Polygon]: Gas price is
{gas_price:?}"),
            Err(e) => panic!("[Polygon]: Cannot estimate gas: {e:?}"),
        }
    }
}

async fn provider_oracle() {
    const RPC_URL: &str = "https://eth.llamarpc.com";
    let provider = Provider::<Http>::try_from(RPC_URL).unwrap();
    let oracle = ProviderOracle::new(provider);
    match oracle.fetch().await {
        Ok(gas_price) => println!("[Provider oracle]: Gas price is
{gas_price:?}"),
        Err(e) => panic!("[Provider oracle]: Cannot estimate gas: {e:?}"),
    }
}

/*
// FIXME: Etherchain URL is broken (Http 404)
async fn etherchain() {
    let oracle = Etherchain::new().category(GasCategory::Standard);
    match oracle.fetch().await {
        Ok(gas_price) => println!("[Etherchain]: Gas price is
{gas_price:?}"),
        Err(e) => panic!("[Etherchain]: Cannot estimate gas: {e:?}"),
    }
}*/
```

Nonce manager

```rust
use ethers::{
    core::{
        types::{BlockNumber, TransactionRequest},
        utils::Anvil,
    },
    middleware::MiddlewareBuilder,
    providers::{Http, Middleware, Provider},
};
use eyre::Result;

/// In Ethereum, the nonce of a transaction is a number that represents the
number of transactions
/// that have been sent from a particular account. The nonce is used to
ensure that transactions are
/// processed in the order they are intended, and to prevent the same
transaction from being
/// processed multiple times.
///
/// The nonce manager in ethers-rs is a middleware that helps you manage the
nonce
/// of transactions by keeping track of the current nonce for a given account
and automatically
/// incrementing it as needed. This can be useful if you want to ensure that
transactions are sent
/// in the correct order, or if you want to avoid having to manually manage
the nonce yourself.
#[tokio::main]
async fn main() -> Result<()> {
    let anvil = Anvil::new().spawn();
    let endpoint = anvil.endpoint();

    let provider = Provider::<Http>::try_from(endpoint)?;
    let accounts = provider.get_accounts().await?;
    let account = accounts[0];
    let to = accounts[1];
    let tx = TransactionRequest::new().from(account).to(to).value(1000);

    let nonce_manager = provider.nonce_manager(account);

    let curr_nonce = nonce_manager
        .get_transaction_count(account, Some(BlockNumber::Pending.into()))
        .await?
        .as_u64();

    assert_eq!(curr_nonce, 0);

    nonce_manager.send_transaction(tx, None).await?.await?.unwrap();
    let next_nonce = nonce_manager.next().as_u64();

    assert_eq!(next_nonce, 1);

    Ok(())
}
```

Policy middleware

```
use ethers::{
    core::{types::TransactionRequest, utils::Anvil},
    middleware::{
        policy::{PolicyMiddlewareError, RejectEverything},
        MiddlewareBuilder, PolicyMiddleware,
    },
    providers::{Http, Middleware, Provider},
};
use eyre::Result;

/// Policy middleware is a way to inject custom logic into the process of
sending transactions and
/// interacting with contracts on the Ethereum blockchain. It allows you to
define rules or policies
/// that should be followed when performing these actions, and to customize
the behavior of the
/// library based on these policies.
#[tokio::main]
async fn main() -> Result<()> {
    let anvil = Anvil::new().spawn();
    let endpoint = anvil.endpoint();

    let provider = Provider::<Http>::try_from(endpoint)?;

    let accounts = provider.get_accounts().await?;
    let account = accounts[0];
    let to = accounts[1];
    let tx = TransactionRequest::new().from(account).to(to).value(1000);

    let policy = RejectEverything;
    let policy_middleware = provider.wrap_into(|p| PolicyMiddleware::new(p,
policy));

    match policy_middleware.send_transaction(tx, None).await {
        Err(e) => {
            // Given the RejectEverything policy, we expect to execute this
branch
            assert!(matches!(e, PolicyMiddlewareError::PolicyError(())))
        }
        _ => panic!("We don't expect this to happen!"),
    }

    Ok(())
}
```

Signer

```
use ethers::{
    core::{types::TransactionRequest, utils::Anvil},
    middleware::SignerMiddleware,
    providers::{Http, Middleware, Provider},
    signers::{LocalWallet, Signer},
};
use eyre::Result;
use std::convert::TryFrom;

/// In Ethereum, transactions must be signed with a private key before they
can be broadcast to the
/// network. Ethers-rs provides a way to customize this process by allowing
/// you to define a signer, called to sign transactions before they are sent.
#[tokio::main]
async fn main() -> Result<()> {
    let anvil = Anvil::new().spawn();

    let wallet: LocalWallet = anvil.keys()[0].clone().into();
    let wallet2: LocalWallet = anvil.keys()[1].clone().into();

    // connect to the network
    let provider = Provider::<Http>::try_from(anvil.endpoint())?;

    // connect the wallet to the provider
    let client = SignerMiddleware::new(provider,
wallet.with_chain_id(anvil.chain_id()));

    // craft the transaction
    let tx = TransactionRequest::new().to(wallet2.address()).value(10000);

    // send it!
    let pending_tx = client.send_transaction(tx, None).await?;

    // get the mined tx
    let receipt = pending_tx.await?.ok_or_else(|| eyre::format_err!("tx
dropped from mempool"))?;
    let tx = client.get_transaction(receipt.transaction_hash).await?;

    println!("Sent tx: {}\n", serde_json::to_string(&tx)?);
    println!("Tx receipt: {}", serde_json::to_string(&receipt)?);

    Ok(())
}
```

Contracts

In ethers-rs, contracts are a way to interact with smart contracts on the Ethereum blockchain through rust bindings, which serve as a robust rust API to these objects.

The ethers-contracts module includes the following features:

- Abigen: A module for generating Rust code from Solidity contracts.
- Compile: A module for compiling Solidity contracts into bytecode and ABI files.
- Creating Instances: A module for creating instances of smart contracts.
- Deploy Anvil: A module for deploying smart contracts on the Anvil network.
- Deploy from ABI and bytecode: A module for deploying smart contracts from their ABI and bytecode files.
- Deploy Moonbeam: A module for deploying smart contracts on the Moonbeam network.
- Events: A module for listening to smart contract events.
- Events with Meta: A module for listening to smart contract events with metadata.
- Methods: A module for calling smart contract methods.

The ethers-contracts module provides a convenient way to work with Ethereum smart contracts in Rust. With this module, you can easily create instances of smart contracts, deploy them to the network, and interact with their methods and events.

The Abigen module allows you to generate Rust code from Solidity contracts, which can save you a lot of time and effort when writing Rust code for Ethereum smart contracts.

The Compile module makes it easy to compile Solidity contracts into bytecode and ABI files, which are required for deploying smart contracts.

The Deploy Anvil and Deploy Moonbeam modules allow you to deploy smart contracts to specific networks, making it easy to test and deploy your smart contracts on the desired network.

The Events and Events with Meta modules allow you to listen to smart contract events and retrieve event data, which is essential for building applications that interact with Ethereum smart contracts.

Finally, the Methods module provides a simple way to call smart contract methods from Rust code, allowing you to interact with smart contracts in a programmatic way.

Overall, the ethers-contracts module provides a comprehensive set of tools for working with Ethereum smart contracts in Rust, making it an essential tool for Rust developers building decentralized applications on the Ethereum network.

Abigen

```
use ethers::{
    prelude::{abigen, Abigen},
    providers::{Http, Provider},
    types::Address,
};
use eyre::Result;
use std::sync::Arc;

/// Abigen is used to generate Rust code to interact with smart contracts on
the blockchain.
/// It provides a way to encode and decode data that is passed to and from
smart contracts.
/// The output of abigen is Rust code, that is bound to the contract's
interface, allowing
/// developers to call its methods to read/write on-chain state and subscribe
to realtime events.
///
/// The abigen tool can be used in two ways, addressing different use-cases
scenarios and developer
/// taste:
///
/// 1. **Rust file generation:** takes a smart contract's Application Binary
Interface (ABI)
/// file and generates a Rust file to interact with it. This is useful if the
smart contract is
/// referenced in different places in a project. File generation from ABI can
also be easily
/// included as a build step of your application.
/// 2. **Rust inline generation:** takes a smart contract's solidity
definition and generates inline
/// Rust code to interact with it. This is useful for fast prototyping and
for tight scoped
/// use-cases of your contracts.
/// 3. **Rust inline generation from ABI:** similar to the previous point but
instead of Solidity
/// code takes in input a smart contract's Application Binary Interface (ABI)
file.
#[tokio::main]
async fn main() -> Result<()> {
    rust_file_generation()?;
    rust_inline_generation().await?;
    rust_inline_generation_from_abi();
    Ok(())
}

fn rust_file_generation() -> Result<()> {
    let abi_source = "./examples/contracts/examples/abi/IERC20.json";
    let out_file = std::env::temp_dir().join("ierc20.rs");
    if out_file.exists() {
        std::fs::remove_file(&out_file)?;
    }
    Abigen::new("IERC20", abi_source)?.generate()?.write_to_file(out_file)?;
    Ok(())
}

fn rust_inline_generation_from_abi() {
```

```
    abigen!(IERC20, "./examples/contracts/examples/abi/IERC20.json");
}

async fn rust_inline_generation() -> Result<()> {
    // The abigen! macro expands the contract's code in the current scope
    // so that you can interface your Rust program with the blockchain
    // counterpart of the contract.
    abigen!(
        IERC20,
        r#"[
            function totalSupply() external view returns (uint256)
            function balanceOf(address account) external view returns
(uint256)
            function transfer(address recipient, uint256 amount) external
returns (bool)
            function allowance(address owner, address spender) external view
returns (uint256)
            function approve(address spender, uint256 amount) external
returns (bool)
            function transferFrom( address sender, address recipient, uint256
amount) external returns (bool)
            event Transfer(address indexed from, address indexed to, uint256
value)
            event Approval(address indexed owner, address indexed spender,
uint256 value)
        ]"#,
    );

    const RPC_URL: &str = "https://eth.llamarpc.com";
    const WETH_ADDRESS: &str = "0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2";

    let provider = Provider::<Http>::try_from(RPC_URL)?;
    let client = Arc::new(provider);
    let address: Address = WETH_ADDRESS.parse()?;
    let contract = IERC20::new(address, client);

    if let Ok(total_supply) = contract.total_supply().call().await {
        println!("WETH total supply is {total_supply:?}");
    }

    Ok(())
}
```

Compile

```
use ethers::{prelude::Abigen, solc::Solc};
use eyre::Result;

fn main() -> Result<()> {
    let mut args = std::env::args();
    args.next().unwrap(); // skip program name

    let contract_name = args.next().unwrap_or_else(||
"SimpleStorage".to_owned());
    let contract: String = args
        .next()
        .unwrap_or_else(|| "examples/contracts/examples/contracts
/contract.sol".to_owned());

    println!("Generating bindings for {contract}\n");

    // compile it
    let abi = if contract.ends_with(".sol") {
        let contracts = Solc::default().compile_source(&contract)?;
        let abi = contracts.get(&contract,
&contract_name).unwrap().abi.unwrap();
        serde_json::to_string(abi).unwrap()
    } else {
        contract
    };

    let bindings = Abigen::new(&contract_name, abi)?.generate()?;

    // print to stdout if no output arg is given
    if let Some(output_path) = args.next() {
        bindings.write_to_file(output_path)?;
    } else {
        bindings.write(&mut std::io::stdout())?;
    }

    Ok(())
}
```

Creating Instances

```
#[tokio::main]
async fn main() {}
```

Deploy Anvil

```
use ethers::{
    contract::{abigen, ContractFactory},
    core::utils::Anvil,
    middleware::SignerMiddleware,
    providers::{Http, Provider},
    signers::{LocalWallet, Signer},
    solc::{Artifact, Project, ProjectPathsConfig},
};
use eyre::Result;
use std::{path::PathBuf, sync::Arc, time::Duration};

// Generate the type-safe contract bindings by providing the ABI
// definition in human readable format
abigen!(
    SimpleContract,
    r#"[
        function setValue(string)
        function getValue() external view returns (string)
        event ValueChanged(address indexed author, string oldValue, string
newValue)
    ]"#,
    event_derives(serde::Deserialize, serde::Serialize)
);

#[tokio::main]
async fn main() -> Result<()> {
    // the directory we use is root-dir/examples
    let root = PathBuf::from(env!("CARGO_MANIFEST_DIR")).join("examples");
    // we use `root` for both the project root and for where to search for
contracts since
    // everything is in the same directory
    let paths =
ProjectPathsConfig::builder().root(&root).sources(&root).build().unwrap();

    // get the solc project instance using the paths above
    let project =
Project::builder().paths(paths).ephemeral().no_artifacts().build().unwrap();
    // compile the project and get the artifacts
    let output = project.compile().unwrap();
    let contract = output.find_first("SimpleStorage").expect("could not find
contract").clone();
    let (abi, bytecode, _) = contract.into_parts();

    // 2. instantiate our wallet & anvil
    let anvil = Anvil::new().spawn();
    let wallet: LocalWallet = anvil.keys()[0].clone().into();

    // 3. connect to the network
    let provider =
        Provider::
<Http>::try_from(anvil.endpoint())?.interval(Duration::from_millis(10u64));

    // 4. instantiate the client with the wallet
    let client = SignerMiddleware::new(provider,
wallet.with_chain_id(anvil.chain_id()));
    let client = Arc::new(client);
```

```
    // 5. create a factory which will be used to deploy instances of the
contract
    let factory = ContractFactory::new(abi.unwrap(), bytecode.unwrap(),
client.clone());

    // 6. deploy it with the constructor arguments
    let contract = factory.deploy("initial
value".to_string())?.send().await?;

    // 7. get the contract's address
    let addr = contract.address();

    // 8. instantiate the contract
    let contract = SimpleContract::new(addr, client.clone());

    // 9. call the `setValue` method
    // (first `await` returns a PendingTransaction, second one waits for it
to be mined)
    let _receipt = contract.set_value("hi".to_owned()).send().await?.await?;

    // 10. get all events
    let logs =
contract.value_changed_filter().from_block(0u64).query().await?;

    // 11. get the new value
    let value = contract.get_value().call().await?;

    println!("Value: {value}. Logs: {}", serde_json::to_string(&logs)?);

    Ok(())
}
```

Deploying a Contract from ABI and Bytecode

```
use ethers::{
    contract::abigen,
    core::utils::Anvil,
    middleware::SignerMiddleware,
    providers::{Http, Provider},
    signers::{LocalWallet, Signer},
};
use eyre::Result;
use std::{convert::TryFrom, sync::Arc, time::Duration};

// Generate the type-safe contract bindings by providing the json artifact
// *Note*: this requires a `bytecode` and `abi` object in the `greeter.json`
artifact:
// `{"abi": [..], "bin": "..."}` , `{"abi": [..], "bytecode": {"object":
"..."}}` or
// `{"abi": [..], "bytecode": "..."}` this will embedd the bytecode in a
variable `GREETER_BYTECODE`
abigen!(Greeter, "ethers-contract/tests/solidity-contracts/greeter.json",);

#[tokio::main]
async fn main() -> Result<()> {
    // 1. compile the contract (note this requires that you are inside the
`examples` directory) and
    // launch anvil
    let anvil = Anvil::new().spawn();

    // 2. instantiate our wallet
    let wallet: LocalWallet = anvil.keys()[0].clone().into();

    // 3. connect to the network
    let provider =
        Provider::
<Http>::try_from(anvil.endpoint())?.interval(Duration::from_millis(10u64));

    // 4. instantiate the client with the wallet
    let client = Arc::new(SignerMiddleware::new(provider,
wallet.with_chain_id(anvil.chain_id())));

    // 5. deploy contract
    let greeter_contract =
        Greeter::deploy(client, "Hello
World!".to_string()).unwrap().send().await.unwrap();

    // 6. call contract function
    let greeting = greeter_contract.greet().call().await.unwrap();
    assert_eq!("Hello World!", greeting);

    Ok(())
}
```

Deploy Moonbeam

```
use ethers::contract::abigen;

abigen!(
    SimpleContract,
    "./examples/contracts/examples/abi/contract_abi.json",
    event_derives(serde::Deserialize, serde::Serialize)
);

/// This requires a running moonbeam dev instance on `localhost:9933`
/// See `https://docs.moonbeam.network/builders/get-started/moonbeam-dev/`
for reference
///
/// This has been tested against:
///
/// ```bash
///  docker run --rm --name moonbeam_development -p 9944:9944 -p 9933:9933
purestake/moonbeam:v0.14.2 --dev --ws-external --rpc-external
/// ```
#[tokio::main]
async fn main() -> eyre::Result<()> {
    use ethers::prelude::*;
    use std::{convert::TryFrom, path::Path, sync::Arc, time::Duration};

    const MOONBEAM_DEV_ENDPOINT: &str = "http://localhost:9933";

    // set the path to the contract, `CARGO_MANIFEST_DIR` points to the
directory containing the
    // manifest of `ethers`. which will be `../` relative to this file
    let source = Path::new(&env!
("CARGO_MANIFEST_DIR")).join("examples/contract.sol");
    let compiled = Solc::default().compile_source(source).expect("Could not
compile contracts");
    let (abi, bytecode, _runtime_bytecode) =
        compiled.find("SimpleStorage").expect("could not find
contract").into_parts_or_default();

    // 1. get a moonbeam dev key
    let key = ethers::core::utils::moonbeam::dev_keys()[0].clone();

    // 2. instantiate our wallet with chain id
    let wallet: LocalWallet =
LocalWallet::from(key).with_chain_id(Chain::MoonbeamDev);

    // 3. connect to the network
    let provider =
        Provider::
<Http>::try_from(MOONBEAM_DEV_ENDPOINT)?.interval(Duration::from_millis(10u64
));

    // 4. instantiate the client with the wallet
    let client = SignerMiddleware::new(provider, wallet);
    let client = Arc::new(client);

    // 5. create a factory which will be used to deploy instances of the
contract
    let factory = ContractFactory::new(abi, bytecode, client.clone());
```

```
    // 6. deploy it with the constructor arguments, note the `legacy` call
    let contract = factory.deploy("initial
value".to_string())?.legacy().send().await?;

    // 7. get the contract's address
    let addr = contract.address();

    // 8. instantiate the contract
    let contract = SimpleContract::new(addr, client.clone());

    // 9. call the `setValue` method
    // (first `await` returns a PendingTransaction, second one waits for it
to be mined)
    let _receipt =
contract.set_value("hi".to_owned()).legacy().send().await?.await?;

    // 10. get all events
    let logs =
contract.value_changed_filter().from_block(0u64).query().await?;

    // 11. get the new value
    let value = contract.get_value().call().await?;

    println!("Value: {value}. Logs: {}", serde_json::to_string(&logs)?);

    Ok(())
}
```

Events

```rust
use ethers::{
    contract::abigen,
    core::types::Address,
    providers::{Provider, StreamExt, Ws},
};
use eyre::Result;
use std::sync::Arc;

abigen!(
    IERC20,
    r#"[
        event Transfer(address indexed from, address indexed to, uint256 
value)
        event Approval(address indexed owner, address indexed spender, 
uint256 value)
    ]"#,
);

const WSS_URL: &str = "wss://mainnet.infura.io/ws/v3
/c60b0bb42f8a4c6481ecd229eddaca27";
const WETH_ADDRESS: &str = "0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2";

#[tokio::main]
async fn main() -> Result<()> {
    let provider = Provider::<Ws>::connect(WSS_URL).await?;
    let client = Arc::new(provider);
    let address: Address = WETH_ADDRESS.parse()?;
    let contract = IERC20::new(address, client);

    listen_all_events(&contract).await?;
    listen_specific_events(&contract).await?;

    Ok(())
}

/// Given a contract instance, subscribe to all possible events.
/// This allows to centralize the event handling logic and dispatch
/// proper actions.
///
/// Note that all event bindings have been generated
/// by abigen. Feel free to investigate the abigen expanded code to
/// better understand types and functionalities.
async fn listen_all_events(contract: &IERC20<Provider<Ws>>) -> Result<()> {
    let events = contract.events().from_block(16232696);
    let mut stream = events.stream().await?.take(1);

    while let Some(Ok(evt)) = stream.next().await {
        match evt {
            IERC20Events::ApprovalFilter(f) => println!("{f:?}"),
            IERC20Events::TransferFilter(f) => println!("{f:?}"),
        }
    }

    Ok(())
}
```

```
/// Given a contract instance subscribe to a single type of event.
///
/// Note that all event bindings have been generated
/// by abigen. Feel free to investigate the abigen expanded code to
/// better understand types and functionalities.
async fn listen_specific_events(contract: &IERC20<Provider<Ws>>) ->
Result<()> {
    let events = contract.event::<ApprovalFilter>().from_block(16232696);
    let mut stream = events.stream().await?.take(1);

    while let Some(Ok(f)) = stream.next().await {
        println!("ApprovalFilter event: {f:?}");
    }

    Ok(())
}
```

Events with meta

```rust
use ethers::{
    contract::abigen,
    core::types::Address,
    providers::{Provider, StreamExt, Ws},
};
use eyre::Result;
use std::sync::Arc;

// Generate the type-safe contract bindings by providing the ABI
// definition in human readable format
abigen!(
    ERC20,
    r#"[
        event  Transfer(address indexed src, address indexed dst, uint wad)
    ]"#,
);

#[tokio::main]
async fn main() -> Result<()> {
    let client =
        Provider::<Ws>::connect("wss://mainnet.infura.io/ws/v3
/c60b0bb42f8a4c6481ecd229eddaca27")
            .await?;

    let client = Arc::new(client);

    // WETH Token
    let address = "0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2".parse::
<Address>()?;
    let weth = ERC20::new(address, Arc::clone(&client));

    // Subscribe Transfer events
    let events = weth.events().from_block(16232698);
    let mut stream = events.stream().await?.with_meta().take(1);
    while let Some(Ok((event, meta))) = stream.next().await {
        println!("src: {:?}, dst: {:?}, wad: {:?}", event.src, event.dst,
event.wad);

        println!(
            r#"address: {:?},
               block_number: {:?},
               block_hash: {:?},
               transaction_hash: {:?},
               transaction_index: {:?},
               log_index: {:?}
            "#,
            meta.address,
            meta.block_number,
            meta.block_hash,
            meta.transaction_hash,
            meta.transaction_index,
            meta.log_index
        );
    }

    Ok(())
```

```
}
```

Methods

```rust
#[tokio::main]
async fn main() {}
```

Ethers-rs: Working with Events

In this section we will discuss how to monitor, subscribe, and listen to events using the ethers-rs library. Events are an essential part of smart contract development, as they allow you to track specific occurrences on the blockchain, such as transactions, state changes, or function calls.

Overview

ethers-rs provides a simple and efficient way to interact with events emitted by smart contracts. You can listen to events, filter them based on certain conditions, and subscribe to event streams for real-time updates. The key components you will work with are:

1. Event: A struct representing an event emitted by a smart contract.
2. EventWatcher: A struct that allows you to monitor and filter events.
3. SubscriptionStream: A stream of events you can subscribe to for real-time updates.

Getting Started

Before diving into event handling, ensure you have ethers-rs added to your project's dependencies in Cargo.toml:

```
[dependencies]
ethers = { version = "2.0.0.", features = ["full"] }
```

Now, let's import the necessary components from the ethers-rs library:

```
use ethers::{
prelude::contract::{Contract, EthEvent},
};
```

Listening to Events

To listen to events, you'll need to instantiate a Contract object and use the event method to create an Event struct. You'll also need to define a struct that implements the EthEvent trait, representing the specific event you want to listen to.

Consider a simple smart contract that emits an event called ValueChanged:

```
pragma solidity ^0.8.0;

contract SimpleStorage {

    uint256 public value;
    event ValueChanged(uint256 newValue);

    function setValue(uint256 _value) public {
        value = _value;
        emit ValueChanged(_value);
    }

}
```

First, define a struct representing the ValueChanged event:

```
#[derive(Debug, Clone, EthEvent)]
pub struct ValueChanged {
    pub new_value: U256,
}
```

Then, create an instance of the Contract object and listen for the ValueChanged event:

```
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let provider = Provider::<Http>::try_from("http://localhost:8545")?;
    let contract_address = "0xcontract_address_here".parse()?;
    let contract = Contract::from_json(provider,
        contract_address,
        include_bytes!("../contracts/abis/SimpleStorage.json"))?;

    let event = contract.event::<ValueChanged>()?;

    // Your code to handle the event goes here.

    Ok(())

}
```

Filtering Events

You can filter events based on specific conditions using the EventWatcher struct. To create an EventWatcher, call the watcher method on your Event object:

```
let watcher = event.watcher().from_block(5).to_block(10);
```

In this example, the EventWatcher will only monitor events from block 5 to block 10.

Subscribing to Events

To receive real-time updates for an event, create a SubscriptionStream by calling the subscribe method on your EventWatcher:

```
let mut stream = watcher.subscribe().await?;
```

You can now listen to events as they are emitted by the smart contract:

```
while let Some(event) = stream.next().await {
    match event {
        Ok(log) => {println!("New event: {:?}", log)},
        Err(e) => {println!("Error: {:?}", e)},
```

Logs and filtering

```
use ethers::{
    core::types::{Address, Filter, H160, H256, U256},
    providers::{Http, Middleware, Provider},
};
use eyre::Result;
use std::sync::Arc;

const HTTP_URL: &str = "https://rpc.flashbots.net";
const V3FACTORY_ADDRESS: &str = "0x1F98431c8aD98523631AE4a59f267346ea31F984";
const DAI_ADDRESS: &str = "0x6B175474E89094C44Da98b954EedeAC495271d0F";
const USDC_ADDRESS: &str = "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48";
const USDT_ADDRESS: &str = "0xdAC17F958D2ee523a2206206994597C13D831ec7";

/// This example demonstrates filtering and parsing event logs by fetching
all Uniswap V3 pools
/// where both tokens are in the set [USDC, USDT, DAI].
///
/// V3 factory reference: https://github.com/Uniswap/v3-core/blob
/main/contracts/interfaces/IUniswapV3Factory.sol
#[tokio::main]
async fn main() -> Result<()> {
    let provider = Provider::<Http>::try_from(HTTP_URL)?;
    let client = Arc::new(provider);
    let token_topics = [
        H256::from(USDC_ADDRESS.parse::<H160>()?),
        H256::from(USDT_ADDRESS.parse::<H160>()?),
        H256::from(DAI_ADDRESS.parse::<H160>()?),
    ];
    let filter = Filter::new()
        .address(V3FACTORY_ADDRESS.parse::<Address>()?)
        .event("PoolCreated(address,address,uint24,int24,address)")
        .topic1(token_topics.to_vec())
        .topic2(token_topics.to_vec())
        .from_block(0);
    let logs = client.get_logs(&filter).await?;
    println!("{} pools found!", logs.iter().len());
    for log in logs.iter() {
        let token0 = Address::from(log.topics[1]);
        let token1 = Address::from(log.topics[2]);
        let fee_tier = U256::from_big_endian(&log.topics[3].as_bytes()
[29..32]);
        let tick_spacing = U256::from_big_endian(&log.data[29..32]);
        let pool = Address::from(&log.data[44..64].try_into()?);
        println!(
            "pool = {pool}, token0 = {token0}, token1 = {token1}, fee =
{fee_tier}, spacing = {tick_spacing}"
        );
    }
    Ok(())
}
use ethers::prelude::*;
use eyre::Result;
use serde::{Deserialize, Serialize};
use std::sync::Arc;

const WSS_URL: &str = "wss://mainnet.infura.io/ws/v3
```

```
/c60b0bb42f8a4c6481ecd229eddaca27";

#[derive(Clone, Debug, Serialize, Deserialize, EthEvent)]
pub struct Transfer {
    #[ethevent(indexed)]
    pub from: Address,
    #[ethevent(indexed)]
    pub to: Address,
    pub tokens: U256,
}

/// This example shows how to subscribe to events using the Ws transport for
a specific event
#[tokio::main]
async fn main() -> Result<()> {
    let provider = Provider::<Ws>::connect(WSS_URL).await?;
    let provider = Arc::new(provider);
    let event = Transfer::new::<_, Provider<Ws>>(Filter::new(),
Arc::clone(&provider));
    let mut transfers = event.subscribe().await?.take(5);
    while let Some(log) = transfers.next().await {
        println!("Transfer: {:?}", log);
    }

    Ok(())
}
```

Ethers-rs: Subscriptions

Here we will discuss how to use `ethers-rs` to subscribe and listen to blocks, events, and logs. Subscriptions provide a way to receive real-time updates on various activities on the Ethereum blockchain, allowing you to monitor the network and react to changes as they happen.

Overview

ethers-rs offers a convenient way to work with subscriptions, enabling you to listen to new blocks, transaction receipts, and logs. The main components you will work with are:

1. Provider: The main struct used to interact with the Ethereum network.
2. SubscriptionStream: A stream of updates you can subscribe to for real-time notifications.

Getting Started

Before working with subscriptions, make sure you have ethers-rs added to your project's dependencies in Cargo.toml:

```
[dependencies]
ethers = { version = "2.0.0", features = ["full"] }
```

Next, import the necessary components from the ethers-rs library:

```
use ethers::{prelude::\*,types::H256,};
```

Subscribing to Events

As we discussed in the previous section on events, you can subscribe to specific events emitted by smart contracts using the EventWatcher struct. To create a SubscriptionStream, call the subscribe method on your EventWatcher:

```
let mut stream = watcher.subscribe().await?;
```

Now, you can listen to events as they are emitted by the smart contract:

```
while let Some(event) = stream.next().await {
    match event {
        Ok(log) => {
            println!("New event: {:?}", log);
        }
        Err(e) => {
            eprintln!("Error: {:?}", e);
        }
    }
}
```

By using the subscription features provided by ethers-rs, you can efficiently monitor and react to various activities on the Ethereum network. Subscriptions are a powerful tool for building responsive and dynamic applications that can interact with smart contracts and stay up-to-date with the latest network events.

Unsubscribing from Subscriptions

In some cases, you may want to stop listening to a subscription. To do this, simply drop the SubscriptionStream:

```
drop(stream);
```

This will stop the stream from receiving any further updates.

Subscribing to New Blocks

To subscribe to new blocks, create a Provider instance and call the subscribe_blocks method:

```
async fn main() -> Result<(), Box<dyn std::error::Error>> {
let provider = Provider::<Http>::try_from("http://localhost:8545")?;

    let mut stream = provider.subscribe_blocks().await?;

    // Your code to handle new blocks goes here.

    Ok(())

}
```

You can now listen to new blocks as they are mined:

```
while let Some(block) = stream.next().await {
    match block {
        Ok(block) => {
            println!("New block: {:?}", block);
        }
        Err(e) => {
            eprintln!("Error: {:?}", e);
        }
    }
}
```

Here is another example of subscribing to new blocks:

```rust
use ethers::providers::{Middleware, Provider, StreamExt, Ws};
use eyre::Result;

#[tokio::main]
async fn main() -> Result<()> {
    let provider =
        Provider::<Ws>::connect("wss://mainnet.infura.io/ws/v3
/c60b0bb42f8a4c6481ecd229eddaca27")
            .await?;
    let mut stream = provider.subscribe_blocks().await?.take(1);
    while let Some(block) = stream.next().await {
        println!(
            "Ts: {:?}, block number: {} -> {:?}",
            block.timestamp,
            block.number.unwrap(),
            block.hash.unwrap()
        );
    }

    Ok(())
}
```

Subscribe events by type

```rust
use ethers::{
    contract::{abigen, Contract},
    core::types::ValueOrArray,
    providers::{Provider, StreamExt, Ws},
};
use std::{error::Error, sync::Arc};

abigen!(
    AggregatorInterface,
    r#"[
        event AnswerUpdated(int256 indexed current, uint256 indexed roundId,
uint256 updatedAt)
    ]"#,
);

const PRICE_FEED_1: &str = "0x7de93682b9b5d80d45cd371f7a14f74d49b0914c";
const PRICE_FEED_2: &str = "0x0f00392fcb466c0e4e4310d81b941e07b4d5a079";
const PRICE_FEED_3: &str = "0xebf67ab8cff336d3f609127e8bbf8bd6dd93cd81";

/// Subscribe to a typed event stream without requiring a `Contract`
instance.
/// In this example we subscribe Chainlink price feeds and filter out them
/// by address.
#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let client = get_client().await;
    let client = Arc::new(client);

    // Build an Event by type. We are not tied to a contract instance. We use
builder functions to
    // refine the event filter
    let event = Contract::event_of_type::<AnswerUpdatedFilter>(client)
        .from_block(16022082)
        .address(ValueOrArray::Array(vec![
            PRICE_FEED_1.parse()?,
            PRICE_FEED_2.parse()?,
            PRICE_FEED_3.parse()?,
        ]));

    let mut stream = event.subscribe_with_meta().await?.take(2);

    // Note that `log` has type AnswerUpdatedFilter
    while let Some(Ok((log, meta))) = stream.next().await {
        println!("{log:?}");
        println!("{meta:?}")
    }

    Ok(())
}

async fn get_client() -> Provider<Ws> {
    Provider::<Ws>::connect("wss://mainnet.infura.io/ws/v3
/c60b0bb42f8a4c6481ecd229eddaca27")
        .await
        .unwrap()
}
```

Subscribing to Logs

To subscribe to logs, create a Filter object that specifies the criteria for the logs you want to listen to. Then, pass the filter to the Provider's subscribe_logs method:

```
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let provider = Provider::<Http>::try_from("http://localhost:8545")?;

    let filter = Filter::new().address("0xcontract_address_here".parse()?);

    let mut stream = provider.subscribe_logs(filter).await?;

    // Your code to handle logs goes here.

    Ok(())

}
```

You can now listen to logs that match your filter criteria:

```
while let Some(log) = stream.next().await {
    match log {
        Ok(log) => {
            println!("New log: {:?}", log);
        }
        Err(e) => {
            eprintln!("Error: {:?}", e);
        }
    }
}
```

Here is another example of subscribing to logs:

```rust
use ethers::{
    core::{
        abi::AbiDecode,
        types::{Address, BlockNumber, Filter, U256},
    },
    providers::{Middleware, Provider, StreamExt, Ws},
};
use eyre::Result;
use std::sync::Arc;

#[tokio::main]
async fn main() -> Result<()> {
    let client =
        Provider::<Ws>::connect("wss://mainnet.infura.io/ws/v3
/c60b0bb42f8a4c6481ecd229eddaca27")
            .await?;
    let client = Arc::new(client);

    let last_block =
client.get_block(BlockNumber::Latest).await?.unwrap().number.unwrap();
    println!("last_block: {last_block}");

    let erc20_transfer_filter =
        Filter::new().from_block(last_block -
25).event("Transfer(address,address,uint256)");

    let mut stream =
client.subscribe_logs(&erc20_transfer_filter).await?.take(2);

    while let Some(log) = stream.next().await {
        println!(
            "block: {:?}, tx: {:?}, token: {:?}, from: {:?}, to: {:?},
amount: {:?}",
            log.block_number,
            log.transaction_hash,
            log.address,
            Address::from(log.topics[1]),
            Address::from(log.topics[2]),
            U256::decode(log.data)
        );
    }

    Ok(())
}
```

Mulitple Multiple Subscriptions

You may need to handle multiple subscriptions simultaneously in your application. To manage multiple SubscriptionStreams, you can use the futures crate to efficiently process updates from all streams concurrently:

```
[dependencies]
futures = "0.3"
```

Then, import the necessary components:

```
use futures::{stream, StreamExt, TryStreamExt};
```

Create multiple subscription streams and merge them into a single stream using the stream::select_all function:

```
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Create multiple subscription streams.
    let mut block_stream = provider.subscribe_blocks().await?;
    let mut log_stream = provider.subscribe_logs(filter).await?;
    let mut event_stream = watcher.subscribe().await?;

    // Merge the streams into a single stream.
    let mut combined_stream = stream::select_all(vec![
        block_stream.map_ok(|block| EventType::Block(block)),
        log_stream.map_ok(|log| EventType::Log(log)),
        event_stream.map_ok(|event| EventType::Event(event)),
    ]);

    // Your code to handle the events goes here.

    Ok(())

}
```

Now, you can listen to updates from all the subscription streams concurrently:

```
while let Some(event) = combined_stream.next().await {
    match event {
        Ok(event) => match event {
            EventType::Block(block) => println!("New block: {:?}", block),
            EventType::Log(log) => println!("New log: {:?}", log),
            EventType::Event(event) => println!("New event: {:?}", event),
        },
        Err(e) => {
            eprintln!("Error: {:?}", e);
        }
    }
}
```

This approach allows you to efficiently handle multiple subscriptions in your application and react to various network activities in a unified manner.

By leveraging the powerful subscription capabilities of ethers-rs, you can create responsive and dynamic applications that stay up-to-date with the latest events on the Ethereum network. The library's flexibility and ease of use make it an ideal choice for developers looking to build robust and performant applications that interact with smart contracts and the Ethereum blockchain.

Big numbers

Ethereum uses big numbers (also known as "bignums" or "arbitrary-precision integers") to represent certain values in its codebase and in blockchain transactions. This is necessary because the EVM operates on a 256-bit word size, which is different from the usual 32-bit or 64-bit of modern machines. This was chosen for the ease of use with 256-bit cryptography (such as Keccak-256 hashes or secp256k1 signatures).

It is worth noting that Ethereum is not the only blockchain or cryptocurrency that uses big numbers. Many other blockchains and cryptocurrencies also use big numbers to represent values in their respective systems.

Utilities

In order to create an application, it is often necessary to convert between the representation of values that is easily understood by humans (such as ether) and the machine-readable form that is used by contracts and math functions (such as wei). This is particularly important when working with Ethereum, as certain values, such as balances and gas prices, must be expressed in wei when sending transactions, even if they are displayed to the user in a different format, such as ether or gwei. To help with this conversion, ethers-rs provides two functions, `parse_units` and `format_units`, which allow you to easily convert between human-readable and machine-readable forms of values. `parse_units` can be used to convert a string representing a value in ether, such as "1.1", into a big number in wei, which can be used in contracts and math functions. `format_units` can be used to convert a big number value into a human-readable string, which is useful for displaying values to users.

Comparison and equivalence

```rust
use ethers::types::U256;

/// `U256` implements traits in `std::cmp`, that means `U256` instances
/// can be easily compared using standard Rust operators.
fn main() {
    // a == b
    let a = U256::from(100_u32);
    let b = U256::from(100_u32);
    assert!(a == b);

    // a < b
    let a = U256::from(1_u32);
    let b = U256::from(100_u32);
    assert!(a < b);

    // a <= b
    let a = U256::from(100_u32);
    let b = U256::from(100_u32);
    assert!(a <= b);

    // a > b
    let a = U256::from(100_u32);
    let b = U256::from(1_u32);
    assert!(a > b);

    // a >= b
    let a = U256::from(100_u32);
    let b = U256::from(100_u32);
    assert!(a >= b);

    // a == 0
    let a = U256::zero();
    assert!(a.is_zero());
}
```

Conversion

```
use ethers::{types::U256, utils::format_units};

/// `U256` provides useful conversion functions to enable transformation into
native Rust types.
///
/// It is important to note that converting a big-number to a floating point
type (such as a `f32`
/// or `f64`) can result in a loss of precision, since you cannot fit 256
bits of information into
/// 64 bits.
///
/// However, there may be cases where you want to perform conversions for
presentation purposes.
/// For example, you may want to display a large number to the user in a more
readable format.
fn main() {
    let num = U256::from(42_u8);

    let a: u128 = num.as_u128();
    assert_eq!(a, 42);

    let b: u64 = num.as_u64();
    assert_eq!(b, 42);

    let c: u32 = num.as_u32();
    assert_eq!(c, 42);

    let d: usize = num.as_usize();
    assert_eq!(d, 42);

    let e: String = num.to_string();
    assert_eq!(e, "42");

    let f: String = format_units(num, 4).unwrap();
    assert_eq!(f, "0.0042");
}
```

Creating instances

```rust
use ethers::{
    types::{serde_helpers::Numeric, U256},
    utils::{parse_units, ParseUnits},
};

fn main() {
    // From strings
    let a = U256::from_dec_str("42").unwrap();
    assert_eq!(format!("{a:?}"), "42");

    let amount = "42";
    let units = 4;
    let pu: ParseUnits = parse_units(amount, units).unwrap();
    let b = U256::from(pu);
    assert_eq!(format!("{b:?}"), "420000");

    // From numbers
    let c = U256::from(42_u8);
    assert_eq!(format!("{c:?}"), "42");

    let d = U256::from(42_u16);
    assert_eq!(format!("{d:?}"), "42");

    let e = U256::from(42_u32);
    assert_eq!(format!("{e:?}"), "42");

    let f = U256::from(42_u64);
    assert_eq!(format!("{f:?}"), "42");

    let g = U256::from(42_u128);
    assert_eq!(format!("{g:?}"), "42");

    let h = U256::from(0x2a);
    assert_eq!(format!("{h:?}"), "42");

    let i: U256 = 42.into();
    assert_eq!(format!("{i:?}"), "42");

    // From `Numeric`
    let num: Numeric = Numeric::U256(U256::one());
    let l = U256::from(num);
    assert_eq!(format!("{l:?}"), "1");

    let num: Numeric = Numeric::Num(42);
    let m = U256::from(num);
    assert_eq!(format!("{m:?}"), "42");
}
```

Math operations

```
use ethers::{types::U256, utils::format_units};
use std::ops::{Div, Mul};

/// `U256` implements traits in `std::ops`, that means it supports arithmetic
operations
/// using standard Rust operators `+`, `-`. `*`, `/`, `%`, along with
additional utilities to
/// perform common mathematical tasks.
fn main() {
    let a = U256::from(10);
    let b = U256::from(2);

    // addition
    let sum = a + b;
    assert_eq!(sum, U256::from(12));

    // subtraction
    let difference = a - b;
    assert_eq!(difference, U256::from(8));

    // multiplication
    let product = a * b;
    assert_eq!(product, U256::from(20));

    // division
    let quotient = a / b;
    assert_eq!(quotient, U256::from(5));

    // modulo
    let remainder = a % b;
    assert_eq!(remainder, U256::zero()); // equivalent to `U256::from(0)`

    // exponentiation
    let power = a.pow(b);
    assert_eq!(power, U256::from(100));
    // powers of 10 can also be expressed like this:
    let power_of_10 = U256::exp10(2);
    assert_eq!(power_of_10, U256::from(100));

    // Multiply two 'ether' numbers:
    // Big numbers are integers, that can represent fixed point numbers.
    // For instance, 1 ether has 18 fixed
    // decimal places (1.000000000000000000), and its big number
    // representation is 10^18 = 1000000000000000000.
    // When we multiply such numbers we are summing up their exponents.
    // So if we multiply 10^18 * 10^18 we get 10^36, that is obviously
incorrect.
    // In order to get the correct result we need to divide by 10^18.
    let eth1 = U256::from(10_000000000000000000_u128); // 10 ether
    let eth2 = U256::from(20_000000000000000000_u128); // 20 ether
    let base = U256::from(10).pow(18.into());
    let mul = eth1.mul(eth2).div(base); // We also divide by 10^18
    let s: String = format_units(mul, "ether").unwrap();
    assert_eq!(s, "200.000000000000000000"); // 200
}
```

Utilities

```rust
use ethers::{
    types::U256,
    utils::{format_units, parse_units, ParseUnits},
};

fn main() {
    parse_units_example();
    format_units_example();
}

/// dApps business logics handle big numbers in 'wei' units (i.e. sending transactions, on-chain
/// math, etc.). We provide convenient methods to map user inputs (usually in 'ether' or 'gwei')
/// into 'wei' format.
fn parse_units_example() {
    let pu: ParseUnits = parse_units("1.0", "wei").unwrap();
    let num = U256::from(pu);
    assert_eq!(num, U256::one());

    let pu: ParseUnits = parse_units("1.0", "kwei").unwrap();
    let num = U256::from(pu);
    assert_eq!(num, U256::from(1000));

    let pu: ParseUnits = parse_units("1.0", "mwei").unwrap();
    let num = U256::from(pu);
    assert_eq!(num, U256::from(1000000));

    let pu: ParseUnits = parse_units("1.0", "gwei").unwrap();
    let num = U256::from(pu);
    assert_eq!(num, U256::from(1000000000));

    let pu: ParseUnits = parse_units("1.0", "szabo").unwrap();
    let num = U256::from(pu);
    assert_eq!(num, U256::from(1000000000000_u128));

    let pu: ParseUnits = parse_units("1.0", "finney").unwrap();
    let num = U256::from(pu);
    assert_eq!(num, U256::from(1000000000000000_u128));

    let pu: ParseUnits = parse_units("1.0", "ether").unwrap();
    let num = U256::from(pu);
    assert_eq!(num, U256::from(1000000000000000000_u128));

    let pu: ParseUnits = parse_units("1.0", 18).unwrap();
    let num = U256::from(pu);
    assert_eq!(num, U256::from(1000000000000000000_u128));
}

/// dApps business logics handle big numbers in 'wei' units (i.e. sending transactions, on-chain
/// math, etc.). On the other hand it is useful to convert big numbers into user readable formats
/// when displaying on a UI. Generally dApps display numbers in 'ether' and 'gwei' units,
/// respectively for displaying amounts and gas. The `format_units` function
```

```rust
will format a big
/// number into a user readable string.
fn format_units_example() {
    // 1 ETHER = 10^18 WEI
    let one_ether = U256::from(1000000000000000000_u128);

    let num: String = format_units(one_ether, "wei").unwrap();
    assert_eq!(num, "1000000000000000000.0");

    let num: String = format_units(one_ether, "gwei").unwrap();
    assert_eq!(num, "1000000000.000000000");

    let num: String = format_units(one_ether, "ether").unwrap();
    assert_eq!(num, "1.000000000000000000");

    // 1 GWEI = 10^9 WEI
    let one_gwei = U256::from(1000000000_u128);

    let num: String = format_units(one_gwei, 0).unwrap();
    assert_eq!(num, "1000000000.0");

    let num: String = format_units(one_gwei, "wei").unwrap();
    assert_eq!(num, "1000000000.0");

    let num: String = format_units(one_gwei, "kwei").unwrap();
    assert_eq!(num, "1000000.000");

    let num: String = format_units(one_gwei, "mwei").unwrap();
    assert_eq!(num, "1000.000000");

    let num: String = format_units(one_gwei, "gwei").unwrap();
    assert_eq!(num, "1.000000000");

    let num: String = format_units(one_gwei, "szabo").unwrap();
    assert_eq!(num, "0.001000000000");

    let num: String = format_units(one_gwei, "finney").unwrap();
    assert_eq!(num, "0.000001000000000");

    let num: String = format_units(one_gwei, "ether").unwrap();
    assert_eq!(num, "0.000000001000000000");
}
```