plugindocs Documentation

Ethereum

May 25, 2022

ABOUT

CHAPTER

ONE

REMIX PLUGIN

Remix plugin is a universal plugin system written in Typescript.

It provides an extendable engine that simplifies communication between multiple internal or external sources.

This repository manages multiple projects related to remix plugins. It's divided into two main categories :

- Engine: A library to manage communication between plugins.
- Plugin: A library to create an external plugin.

1.1 Engine

The core component of the engine is the `@remixproject/engine` library. It can be extended to run in different environments.

| Name | Latest Version | Next Version | ———————————————————————— | :————————: | :————————: | @remixproject/engine | badge | badge | @remixproject/engine-vscode | badge | badge | @remixproject/engine-web | badge | badge | @remixproject/engine-node | badge | badge

> To create a new environment connector, check out @remixproject/engine.

1.2 Plugin

The core component of the plugin is the `@remixproject/plugin` library. It can be extended to run in different environments.

| Name | Latest Version | Next Version | ———————————————————————— | :————————: | :————————: | @remixproject/plugin | badge | badge | @remixproject/plugin-vscode | badge | badge | @remixproject/plugin-iframe | badge | badge | @remixproject/plugin-webview | badge | badge | @remixproject/plugin-child-process | badge | badge

> To create a new environment connector, check out @remixproject/plugin.

1.3 API

Remix plugin offers a set of common APIs for plugins to implement. This set of APIs is used in remix-ide, therefore every plugin running inside remix-ide should be able to run in an engine that implements these APIs.

| Name | Latest Version | Next Version | ——————————- | :————–: | :————–: | @remixproject/plugin-api | badge | badge

> The first goal of **remix plugin** is to enable a plugin to work in the envrionments of multiple engines. If a plugin has dependancies on other plugins, each engine must implement these dependancies.

CHAPTER

TWO

CONTRIBUTE

2.1 Setup

```
git clone https://github.com/ethereum/remix-plugin.git
cd remix-plugin
npm install
```

2.2 See dependancy graph

To better understand the project structure, you can display a dependancy graph with:

```
npm run dep-graph
```

Open your browser on `http://localhost:4211/`.

2.3 Build

This uses nx's affected:build to only update what has been changes since last build.

```
npm run build
```

2.4 Build a specific project

```
npx nx build ${projectName} --with-deps
```

Example for engine-vscode :

```
npx nx build engine-vscode --with-deps
```

2.5 Test

This uses nx's affected:test to only update what has been changes since last test.

```
npm test
```

2.6 Publish

This uses lerna to deploy all the packages with a new version:

```
npm run deploy:latest
```

OR

```
npm run deploy:next
```

CHAPTER

THREE

PLUGIN-API

This library host all the API of the common plugins.

Here is the list of native plugins exposed by Remix IDE

Click on the name of the api to get the full documentation.

|API |Name |Description | |—————|———————————-|————| |File System |fileManager |Manages the File System |Compiler |solidity |The solidity Compiler |Editor |editor |Enables highlighting in the code Editor |Network |network |Defines the network (mainnet, ropsten, ...) and provider (web3, vm, injected) used |Udapp |udapp |Transaction listener |Unit Testing |solidityUnitTesting |Unit testing library in solidity |Settings |settings |Global settings of the IDE |Content Import |contentImport |Import files from github, swarm, ipfs, http or https.

CHAPTER

FOUR

CONTENT IMPORT

- Name in Remix: `contentImport`
- kind: `contentImport`

|Type |Name |Description | |———|———————|———–| |*method* |`resolve` |Resolve a file from github, ipfs, swarm, http or https

4.1 Examples

4.1.1 Methods

`resolve`: Resolve a file from github, ipfs, swarm, http or https

```
const link = "https://github.com/GrandSchtroumpf/solidity-school/blob/master/std-0/1_
↪HelloWorld/HelloWorld.sol"

const { content } = await client.call('contentImport', 'resolve', link)
// OR
const { content } = await client.contentImport.resolve(link)
```

4.2 Types

`ContentImport`: An object that describes the returned file

```
export interface ContentImport {
  content: string
  cleanUrl: string
  type: 'github' | 'http' | 'https' | 'swarm' | 'ipfs'
  url: string
}
```

Type Definitions can be found here

CHAPTER

FIVE

EDITOR

- Name in Remix: `editor`
- kind: `editor`

|Type |Name |Description | |———|———————|————| |*method* |`highlight` |Highlight a piece of code in the editor. |*method* |`discardHighlight` |Remove the highlight triggered by this plugin.

5.1 Examples

5.1.1 Methods

`highlight`: Highlight a piece of code in the editor.

```
const position = {                  // Range of code to highlight
  start: { line: 1, column: 1 },
  end: { line: 1, column: 42 }
}
const file = 'browser/ballot.sol'   // File to highlight
const color = '#e6e6e6'             // Color of the highlight

await client.call('editor', 'highlight', position, file, color)
// OR
await client.editor.highlight(position, file, color)
```

`discardHighlight`: Remove the highlight triggered by this plugin.

```
await client.call('editor', 'discardHighlight')
// OR
await client.editor('discardHighlight')
```

5.2 Types

`HighlightPosition`: The positions where the highlight starts and ends.

```
interface HighlightPosition {
  start: {
    line: number
    column: number
  }
  end: {
    line: number
    column: number
  }
}
```

Type Definitions can be found here

CHAPTER

SIX

FILE SYSTEM

- Name in Remix: `fileManager`
- kind: `fs`

|Type |Name |Description | |———|———————|————| |*event* |`currentFileChanged` |Triggered when a file changes. |*method* |`getCurrentFile` |Get the name of the current file selected. |*method* |`open` |Open the content of the file in the context (eg: Editor). |*method* |`writeFile` |Set the content of a specific file. |*method* |`readFile` |Return the content of a specific file. |*method* |`rename` |Change the path of a file. |*method* |`copyFile` |Upsert a file with the content of the source file. |*method* |`mkdir` |Create a directory. |*method* |`readdir` |Get the list of files in the directory.

6.1 Examples

6.1.1 Events

`currentFileChanged`: Triggered when a file changes.

```
client.solidity.on('currentFileChanged', (fileName: string) => {
  // Do something
})
// OR
client.on('fileManager', 'currentFileChanged', (fileName: string) => {
  // Do something
})
```

6.1.2 Methods

`getCurrentFile`: Get the name of the current file selected.

```
const fileName = await client.fileManager.getCurrentFile()
// OR
const fileName = await client.call('fileManager', 'getCurrentFile')
```

`open`:Open the content of the file in the context (eg: Editor).

```
await client.fileManager.open('browser/ballot.sol')
// OR
await client.call('fileManager', 'open', 'browser/ballot.sol')
```

`readFile`: Get the content of a file.

```
const ballot = await client.fileManager.getFile('browser/ballot.sol')
// OR
const ballot = await client.call('fileManager', 'readFile', 'browser/ballot.sol')
```

`writeFile`: Set the content of a file.

```
await client.fileManager.writeFile('browser/ballot.sol', 'pragma ....')
// OR
await client.call('fileManager', 'writeFile', 'browser/ballot.sol', 'pragma ....')
```

`rename`: Change the path of a file.

```
await client.fileManager.rename('browser/ballot.sol', 'browser/ERC20.sol')
// OR
await client.call('fileManager', 'rename', 'browser/ballot.sol', 'browser/ERC20.sol')
```

`copyFile`: Insert a file with the content of the source file.

```
await client.fileManager.copyFile('browser/ballot.sol', 'browser/NewBallot.sol')
// OR
await client.call('fileManager', 'copyFile', 'browser/ballot.sol', 'browser/NewBallot.sol
↪')
```

`mkdir`: Create a directory.

```
await client.fileManager.mkdir('browser/ERC')
// OR
await client.call('fileManager', 'mkdir', 'browser/ERC')
```

`readdir`: Create a directory.

```
const files = await client.fileManager.readdir('browser/ERC')
// OR
const files = await client.call('fileManager', 'readdir', 'browser/ERC')
```

Type Definitions can be found here

CHAPTER

SEVEN

NETWORK

- Name in Remix: `network`
- kind: `network`

The network exposes methods and events about :

- The provider: `web3`, `vm`, `injected`.
- The Ethereum Network: `mainnet`, `ropsten`, `rinkeby`, `kovan`, `Custom`

|Type |Name |Description |———|——————|– |*event* |`providerChanged` |Triggered when the provider changes. |*method* |`getNetworkProvider` |Get the current provider. |*method* |`getEndpoint` |Get the URL of the provider if `web3`. |*method* |`detectNetwork` |Get the current network used. |*method* |`addNetwork` |Add a custom network. |*method* |`removeNetwork` |Remove a custom network.

7.1 Examples

7.1.1 Events

`providerChanged`: Triggered when the provider changes.

```
client.solidity.on('providerChanged', (provider: NetworkProvider) => {
  // Do something
})
// OR
client.on('fileManager', 'currentFileChanged', (provider: NetworkProvider) => {
  // Do something
})
```

7.1.2 Methods

`getNetworkProvider`: Get the current provider.

```
const provider = await client.network.getNetworkProvider()
// OR
const provider = await client.call('network', 'getNetworkProvider')
```

`getEndpoint`: Get the URL of the provider if `web3`.

```
const endpoint = await client.network.getEndpoint()
// OR
const endpoint = await client.call('network', 'getEndpoint')
```

`detectNetwork`: Get the current network being used.

```
const network = await client.network.detectNetwork()
// OR
const network = await client.call('network', 'detectNetwork')
```

`addNetwork`: Add a custom network.

```
await client.network.addNetwork({ name: 'local', url: 'http://localhost:8586' })
// OR
await client.call('network', 'addNetwork', { name: 'local', url: 'http://localhost:8586'␣
↪})
```

`removeNetwork`: Remove a custom network.

```
await client.network.removeNetwork({ name: 'local', url: 'http://localhost:8586' })
// OR
await client.call('network', 'removeNetwork', 'local')
```

7.2 Types

`NetworkProvider`: A string literal : `vm`, `injected` or `web3`. `Network`: A simple object with the `name` and `id` of the network. `CustomNetwork`: A simple object with a `name` and `url`.

Type Definitions can be found here

CHAPTER

EIGHT

SETTINGS

- Name in Remix: `settings`
- kind: `settings`

|Type |Name |Description | |———|———————|————| |*method* |`getGithubAccessToken` |Returns the current Github Access Token provided in settings

8.1 Examples

8.1.1 Methods

`getGithubAccessToken`: Returns the current Github Access Token provided in settings

```
const token = await client.call('settings', 'getGithubAccessToken')
// OR
const token = await client.settings.getGithubAccessToken()
```

CHAPTER

NINE

SOLIDITY COMPILER

- Name in Remix: `solidity`
- kind: `compiler`

|Type |Name |Description | |———|——————|————| |*event* |`compilationFinished` |Triggered when a compilation finishes. |*method* |`getCompilationResult` |Get the current result of the compilation. |*method* |`compile` |Run solidity compiler with a file.

9.1 Examples

9.1.1 Events

`compilationFinished`:

```
client.solidity.on('compilationFinished', (fileName: string, source:␣
↪CompilationFileSources, languageVersion: string, data: CompilationResult) => {
  // Do something
})
// OR
client.on('solidity', 'compilationFinished', (fileName: string, source:␣
↪CompilationFileSources, languageVersion: string, data: CompilationResult) => {
  // Do something
})
```

9.1.2 Methods

`getCompilationResult`:

```
const result = await client.solidity.getCompilationResult()
// OR
const result = await client.call('solidity', 'getCompilationResult')
```

`compile`:

```
const fileName = 'browser/ballot.sol'
await client.solidity.compile(fileName)
// OR
await client.call('solidity', 'compile', 'fileName')
```

9.2 Types

`CompilationFileSources`: A map with the file name as the key and the content as the value.

`CompilationResult`: The result of the compilation matches the Solidity Compiler Output documentation.

Type Definitions can be found here

CHAPTER

TEN

UDAPP

- Name in Remix: udapp
- kind: udapp

The udapp exposes an interface for interacting with the account and transaction.

|Type |Name |Description |———|—————–|– |*event* |`newTransaction` |Triggered when a new transaction has been sent. |*method* |`sendTransaction` |Send a transaction **only for testing networks**. |*method* |`getAccounts` |Get an array with the accounts exposed. |*method* |`createVMAccount` |Add an account if using the VM provider.

10.1 Examples

10.1.1 Events

`newTransaction`: Triggered when a new transaction has been sent.

```
client.udapp.on('newTransaction', (tx: RemixTx) => {
  // Do something
})
// OR
client.on('udapp', 'newTransaction', (tx: RemixTx) => {
  // Do something
})
```

10.1.2 Methods

`sendTransaction`: Send a transaction **only for testing networks**.

```
const transaction: RemixTx = {
  gasLimit: '0x2710',
  from: '0xca35b7d915458ef540ade6068dfe2f44e8fa733c',
  to: '0xca35b7d915458ef540ade6068dfe2f44e8fa733c'
  data: '0x...',
  value: '0x00',
  useCall: false
}
const receipt = await client.udapp.sendTransaction(transaction)
// OR
const receipt = await client.call('udapp', 'sendTransaction', transaction)
```

`getAccounts`: Get an array with the accounts exposed.

```
const accounts = await client.udapp.getAccounts()
// OR
const accounts = await client.call('udapp', 'getAccounts')
```

`createVMAccount`: Add an account if using the VM provider.

```
const privateKey = "71975fbf7fe448e004ac7ae54cad0a383c3906055a75468714156a07385e96ce"
const balance = "0x56BC75E2D63100000"
const address = await client.udapp.createVMAccount({ privateKey, balance })
// OR
const address = await client.call('udapp', 'createVMAccount', { privateKey, balance })
```

10.2 Types

`RemixTx`: A modified version of the transaction for Remix. `RemixTxReceipt`: A modified version of the transaction receipt for Remix.

Type Definitions can be found here

CHAPTER

ELEVEN

CONTENT IMPORT

- Name in Remix: `solidityUnitTesting`
- kind: `unitTesting`

|Type |Name |Description | |———|———————|————| |*method* |`testFromPath` |Run a solidity test that is inside the file system |*method* |`testFromSource` |Run a solidity test file from the source

11.1 Examples

11.1.1 Methods

`testFromPath`: Run a solidity test that is inside the file system

```
const path = "browser/ballot_test.sol"

const result = await client.call('solidityUnitTesting', 'testFromPath', path)
// OR
const result = await client.solidityUnitTesting.testFromPath(path)
```

`testFromSource`: Run a solidity test file from the source

```
const testFile = `
pragma solidity >=0.5.0 <0.6.0;
import "remix_tests.sol";
import "./HelloWorld.sol";  // HelloWorl.sol must be in "browser"

contract HelloWorldTest {
  HelloWorld helloWorld;
  function beforeEach() public {
    helloWorld = new HelloWorld();
  }

  function checkPrint () public {
    string memory result = helloWorld.print();
    Assert.equal(result, string('Hello World!'), "Method 'print' should return 'Hello␣
↪World!'");
  }
}
`
```

(continues on next page)

(continued from previous page)

```
const result = await client.call('solidityUnitTesting', 'testFromSource', testFile)
// OR
const result = await client.solidityUnitTesting.testFromSource(testFile)
```

11.2 Types

`ContentImport`: An object that describes the returned file

```
export interface UnitTestResult {
  totalFailing: number
  totalPassing: number
  totalTime: number
  errors: UnitTestError[]
}
```

Type Definitions can be found here

CHAPTER

TWELVE

ENGINE CORE

This is the core library used to create a new plugin engine.

| Name | Latest Version | | ——————————————————| :——————: | | @remixproject/engine | badge |

Use this library if you want to create an engine **for a new environment**.

If you want to create an engine for an existing envrionment, use the specific library. For example :

- Engine on the web : @remixproject/engine-web
- Engine on node : @remixproject/engine-node
- Engine on vscode : @remixproject/engine-vscode

12.1 Tutorial

1. Getting Started
2. Plugin Communication
3. Host a Plugin with UI
4. External Plugins
5. Plugin Service

12.2 API

| API | Description | | ——————————-| :——————————–: | | Engine | Register plugins & redirect messages | | Manager | Activate & Deactive plugins |

12.3 Connector

The plugin connector is the main component of `@remixproject/engine`, it defines how an external plugin can connect to the engine. Checkout the documentation.

12.4 Getting started

```
npm install @remixproject/engine
```

The engine works a with two classes :

- `PluginManager`: manage activation/deactivation
- `Engine`: manage registration & communication

```
import { PluginManager, Engine, Plugin } from '@remixproject/engine'

const manager = new PluginManager()
const engine = new Engine()
const plugin = new Plugin({ name: 'plugin-name' })

// Wait for the manager to be loaded
await engine.onload()

// Register plugins
engine.register([manager, plugin])

// Activate plugins
manager.activatePlugin('plugin-name')
```

12.4.1 Registration

The registration makes the plugin available for activation in the engine.

To register a plugin you need:

- `Profile`: The ID card of your plugin.
- `Plugin`: A class that expose the logic of the plugin.

```
const profile = {
  name: 'console',
  methods: ['print']
}

class Console extends Plugin {
  constructor() {
    super(profile)
  }
  print(msg: string) {
    console.log(msg)
  }
}
const consolePlugin = new Console()

// Register plugins
engine.register(consolePlugin)
```

> In the future, this part will be manage by a `Marketplace` plugin.

12.4.2 Activation

The activation process is managed by the `PluginManager`.

Activating a plugin makes it visible to other plugins. Now they can communicate.

```
manager.activatePlugin('console')
```

> The `PluginManager` is a plugin itself.

12.4.3 Communication

`Plugin` exposes a simple interface for communicate between plugins :

- `call`: Call a method exposed by another plugin (This returns always a Promise).
- `on`: Listen on event emitted by another plugin.
- `emit`: Emit an event broadcasted to all listeners.

This code will call the method `print` from the plugin `console` with the parameter `'My message'`.

```
plugin.call('console', 'print', 'My message')
```

12.4.4 Full code example

```
import { PluginManager, Engine, Plugin } from '@remixproject/engine'
const profile = {
  name: 'console',
  methods: ['print']
}

class Console extends Plugin {
  constructor() {
    super(profile)
  }
  print(msg: string) {
    console.log(msg)
  }
}

const manager = new PluginManager()
const engine = new Engine()
const emptyPlugin = new Plugin({ name: 'empty' })
const consolePlugin = new Console()

// Register plugins
engine.register([manager, plugin, consolePlugin])

// Activate plugins
manager.activatePlugin(['empty', 'console'])

// Plugin communication
emptyPlugin.call('console', 'print', 'My message')
```

CHAPTER

THIRTEEN

ENGINE

The `Engine` deals with the registration of the plugins, to make sure they are available for activation.

It manages the interaction between plugins (calls & events).

13.1 Constructor

The `Engine` depends on the plugin manager for the permission system.

```
const manager = new PluginManager()
const engine = new Engine()
engine.register(manager)
```

13.1.1 register

```
register(plugins: Plugin | Plugin[]): string | string[]
```

Register one or several plugins into the engine and return their names.

A plugin **must be register before being activated**.

13.1.2 isRegistered

```
isRegistered(name: string): boolean
```

Checks if a plugin with this name has already been registered by the engine.

13.2 Hooks

13.2.1 onRegistration

```
onRegistration(plugin: Plugin) {}
```

This method triggered when a plugin is registered.

CHAPTER

FOURTEEN

PLUGIN MANAGER

The `PluginManager` deals with activation and deactivation of other plugins. It also manages the permission layer between two plugins.

You can use it with a very loose permissions, or inherit from it to create a custom set of permission rules.

```
class RemixManager extends PluginManager {
  remixPlugins = ['manager', 'solidity', 'fileManager', 'udapp']

  // Create custom method
  isFromRemix(name: string) {
    return this.remixPlugins.includes(name)
  }

  canActivate(from: Profile, to: Profile) {
    return this.isFromRemix(from.name)
  }
}
```

14.1 Events

14.1.1 profileAdded

```
this.on('manager', 'profileAdded', (profile: Profile) => { ... })
```

Emitted when a plugin has been registered by the `Engine`.

14.1.2 profileUpdated

```
this.on('manager', 'profileUpdated', (profile: Profile) => { ... })
```

Emitted when a plugin updates its profile through the `updateProfile` method.

14.1.3 pluginActivated

```
this.on('manager', 'pluginActivated', (profile: Profile) => { ... })
```

Emitted when a plugin has been activated, either with `activatePlugin` or `toggleActive`.

If the plugin was already active, the event won't be triggered.

14.1.4 pluginDeactivated

```
this.on('manager', 'pluginDeactivated', (profile: Profile) => { ... })
```

Emitted when a plugin has been deactivated, either with `deactivatePlugin` or `toggleActive`.

If the plugin was already deactivated, the event won't be triggered.

14.2 Constructor

Used to create a new instance of `PluginManager`. You can specify the profile of the manager to extend the default one.

The property `name` of the profile must be `manager`.

```
const profile = { name: 'manager', methods: [...managerMethods, 'isFromRemiw' ] }
const manager = new RemixManager(profile)
```

14.3 Properties

14.3.1 requestFrom

Return the name of the caller. If no request was provided, it means that the method has been called from the IDE - so we use "manager".

Use this method when you expose custom methods from the Plugin Manager.

14.4 Methods

14.4.1 getProfile

Get the profile if its registered.

```
const profile = manager.getProfile('solidity')
```

14.4.2 updateProfile

Update the profile of the plugin. This method is used to lazy load services in plugins.

Only the caller plugn can update its profile.

The properties "name" and "url" cannot be updated.

```
const methods = [ ...solidity.methods, 'serviceMethod' ]
await solidity.call('manager', 'updateProfile', { methods })
```

14.4.3 isActive

Verify if a plugin is currently active.

```
const isActive = await manager.isActive('solidity')
```

14.4.4 activatePlugin

Check if caller can activate a plugin and activate it if authorized.

This method call `canActivate` *under the hood.*

It can be called directly on the `PluginManager`:

```
manager.activatePlugin('solidity')
```

Or from another plugin (for dependancy for example) :

```
class EthDoc extends Plugin {
  onActivation() {
    return this.call('manager', 'activatePlugin', ['solidity', 'remix-tests'])
  }
}
```

14.4.5 deactivatePlugin

Check if caller can deactivate plugin and deactivate it if authorized.

This method call `canDeactivate` *under the hood.*

It can be called directly on the `PluginManager`:

```
manager.deactivatePlugin('solidity')
```

Or from another plugin :

```
class EthDoc extends Plugin {
  onDeactivation() {
    return this.call('manager', 'deactivatePlugin', ['solidity', 'remix-tests'])
  }
}
```

Deactivating a plugin can have side effect on other plugins that depend on it. We recommend limiting the access to this method to a small set of plugins -if any (see `canDeactivate`).

14.4.6 toggleActive

Activate or deactivate by bypassing permission.

This method should ONLY be used by the IDE. Do not expose this method to other plugins.

```
manager.toggleActive('solidity') // Toggle One
manager.toggleActive(['solidity', 'remix-tests'])  // Toggle Many
```

14.5 Permission

By extending the `PluginManager` you can override the permission methods to create your own rules.

14.5.1 canActivate

Check if a plugin can activate another.

Params

- `from`: The profile of the caller plugin.
- `to`: The profile of the target plugin.

```
class RemixManager extends PluginManager {
  // Ask permission to user if it's not a plugin from Remix
  async canActivate(from: Profile, to: Profile) {
    if (this.isFromRemix(from.name)) {
      return true
    } else {
      return confirm(`Can ${from.name} activates ${to.name}`)
    }
  }
}
```

Don't forget to let 'manager' activate plugins if you're not using `toggleActivate`.

14.5.2 canDeactivate

Check if a plugin can deactivate another.

Params

- `from`: The profile of the caller plugin.
- `to`: The profile of the target plugin.

```
class RemixManager extends PluginManager {
  // Only "manager" can deactivate plugins
  async canDeactivate(from: Profile, to: Profile) {
```

(continues on next page)

(continued from previous page)

```
    return from.name === 'manager'
  }
}
```

Don't forget to let 'manager' deactivate plugins if you're not using `toggleActivate`.

14.5.3 canCall

Check if a plugin can call a method of another plugin.

Params

- `from`: Name of the caller plugin
- `to`: Name of the target plugin
- `method`: Method targeted by the caller
- `message`: Optional Message to display to the user

This method can be called from a plugin to protect the access to one of its methods. Every plugin implements a helper function that takes care of `from` & `to`

```
class SensitivePlugin extends Plugin {
  async sensitiveMethod() {
    const canCall = await this.askUserPermission('sensitiveMethod', 'This method give␣
↪access to sensitvie information')
    if (canCall) {
      // continue sensitive method
    }
  }
}
```

Then the IDE defines how to handle this call :

```
class RemixManager extends PluginManager {
  // Keep track of the permissions
  permissions: {
    [name: string]: {
      [methods: name]: string[]
    }
  } = {}
  // Remember user preference
  async canCall(from: Profile, to: Profile, method: string) {
    // Make sure the caller of this methods is the target plugin
    if (to.name !== this.currentRequest) {
      return false
    }
    // Check if preference exist, else ask the user
    if (!this.permissions[from.name]) {
      this.permissions[from.name] = {}
    }
    if (!this.permissions[from.name][method]) {
      this.permissions[from.name][method] = []
```

(continues on next page)

(continued from previous page)

```
    }
    if (this.permissions[from.name][method].includes(to.name)) {
      return true
    } else {
      confirm(`Can ${from.to} call method ${method} of ${to.name} ?`)
        ? !!this.permissions[from.name][method].push(to.name)
        : false
    }
  }
}
```

Consider keeping the preferences in the localstorage for a better user experience.

14.6 Activation Hooks

`PluginManager` provides an interface to react to changes of its state.

protected onPluginActivated?(profile: Profile): any protected onPluginDeactivated?(profile: Profile): any protected onProfileAdded?(profile: Profile): any

14.6.1 onPluginActivated

Triggered whenever a plugin has been activated.

```
class RemixManager extends PluginManager {
  onPluginActivated(profile: Profile) {
    updateMyUI('activate', profile.name)
  }
}
```

14.6.2 onPluginDeactivated

Triggered whenever a plugin has been deactivated.

```
class RemixManager extends PluginManager {
  onPluginDeactivated(profile: Profile) {
    updateMyUI('deactivate', profile.name)
  }
}
```

14.6.3 onProfileAdded

Triggered whenever a plugin has been registered (profile is added to the manager).

```
class RemixManager extends PluginManager {
  onPluginDeactivated(profile: Profile) {
    updateMyUI('add', profile)
  }
}
```

CHAPTER

FIFTEEN

DEVELOP & PUBLISH A CLIENT CONNECTOR

15.1 Install

```
npm install @remixproject/plugin@next
```

15.2 Create your connector

Create a file `index.ts`

```
import { ClientConnector, createConnectorClient, PluginClient, Message } from
↪'@remixproject/plugin'

export class SocketIOConnector implements ClientConnector {

  constructor(private socket) {}
  send(message: Partial<Message>) {
    this.socket.emit('message', message)
  }
  on(cb: (message: Partial<Message>) => void) {
    this.socket.on('message', (msg) => cb(msg))
  }
}

// A simple wrapper function for the plugin developer
export function createSocketIOClient(socket, client?: PluginClient) {
  const connector = new SocketIOConnector(socket)
  return createConnectorClient(connector, client)
}
```

15.3 Build

```
npx tsc index --declaration
```

15.4 Package.json

```
{
  "name": "client-connector-socket.io",
  "main": "index.js",
  "types": "index.d.ts",
  "dependencies": {
    "@remixproject/plugin": "next"
  },
  "peerDependencies": {
    "socket.io": "^2.3.0"
  }
}
```

Some notes here :

- We use `dependancies` for `@remixproject/plugin` as this is the base code for your connector.
- We use `peerDependencies` for the library we wrap (here `socket.io`), as we want to let the user choose his version of it.

15.5 Publish

```
npm publish
```

CHAPTER

SIXTEEN

USE A CLIENT CONNECTOR

Here is how to use your client connector in a plugin :

16.1 Install

```
npm install client-connector-socket.io socket.io
```

16.2 Create a client

This example is an implementation of the Server documentation from socket.io.

```
const { createSocketIOClient } = require('client-connector-socket.io')
const http = require('http').createServer();
const io = require('socket.io')(http);

io.on('connection', async (socket) => {
  const client = createSocketIOClient(socket)
  await client.onload()
  const code = await client.call('fileManager', 'read', 'Ballot.sol')
});

http.listen(3000);
```

CHAPTER

SEVENTEEN

DEVELOP & PUBLISH A PLUGIN CONNECTOR

17.1 Install

```
npm install @remixproject/engine@next
```

17.2 Create your connector

Create a file `index.ts`

```
import { PluginConnector, Profile, ExternalProfile, Message } from '@remixproject/engine'
import io from 'socket.io-client';

export class SocketIOPlugin extends PluginConnector {
  socket: SocketIOClient.Socket

  constructor(profile: Profile & ExternalProfile) {
    super(profile)
  }

  protected connect(url: string): void {
    this.socket = io(url)
    this.socket.on('connect', () => {
      this.socket.on('message', (msg: Message) => this.getMessage(msg))
    })
  }

  protected disconnect(): void {
    this.socket.close()
  }

  protected send(message: Partial<Message>): void {
    this.socket.send(message)
  }

}
```

17.3 Build

```
npx tsc index --declaration
```

17.4 Package.json

```
{
  "name": "plugin-connector-socket.io",
  "main": "index.js",
  "types": "index.d.ts",
  "peerDependencies": {
    "@remixproject/engine": "next",
    "socket.io-client": "^2.3.0"
  }
}
```

17.5 Publish

```
npm publish
```

CHAPTER

EIGHTEEN

USE A PLUGIN CONNECTOR

Here is how to use your plugin connector in an engine :

18.1 Install

```
npm install @remixproject/engine@next plugin-connector-socket.io socket.io-client
```

18.2 Create a client

```
import { PluginManager, Engine, Plugin } from '@remixproject/engine'
import { SocketIOPlugin } from 'plugin-connector-socket.io'

const manager = new PluginManager()
const engine = new Engine()
const plugin = new SocketIOPlugin({ name: 'socket', url: 'http://localhost:3000' })

// Register plugins
engine.register([manager, plugin])

// Activate plugins
manager.activatePlugin('socket')
```

CHAPTER

NINETEEN

CONNECTOR

The engine exposes the **connector** to manage communications with plugins that are not running in the engine's main process.

The choice of connectors depends upon the platform that the engine is operating on.

For example an engine running on the web can have connectors with :

- Iframes
- Webworkers
- …

On the other hand an engine running in a node environment will have :

- Child Process
- Worker Threads
- …

19.1 Create a Connector

A connector is a simple wrapper on both sides of a communication layer. It should implement :

- `ClientConnector`: Connector used by the plugin (client).
- `PluginConnector`: Connector used by the engine.

  From a user point of view, the plugin is the "client" even if it's running in a server.

Let's create a connector for socket.io where :

- `ClientConnector`: Plugin code that runs the server.
- `PluginConnector`: Engine recipient that runs in a browser

19.1.1 ClientConnector

The **connector**'s connection on the plugin side implements the `ClientConnector` interface:

```
export interface ClientConnector {
  /** Send a message to the engine */
  send(message: Partial<Message>): void
  /** Get message from the engine */
  on(cb: (message: Partial<Message>) => void): void
}
```

```
import { ClientConnector, createConnectorClient, PluginClient, Message } from
↪'@remixproject/plugin'

export class SocketIOConnector implements ClientConnector {

  constructor(private socket) {}
  send(message: Partial<Message>) {
    this.socket.emit('message', message)
  }
  on(cb: (message: Partial<Message>) => void) {
    this.socket.on('message', (msg) => cb(msg))
  }
}

// A simple wrapper function for the plugin developer
export function createSocketIOClient(socket, client?: PluginClient) {
  const connector = new SocketIOConnector(socket)
  return createConnectorClient(connector, client)
}
```

Checkout how to publish your client connector on npm.

19.1.2 PluginConnector

The `PluginConnector` is an abstract class to be extended:

```
import { PluginConnector, Profile, ExternalProfile, Message } from '@remixproject/engine'
import io from 'socket.io-client';

export class SocketIOPlugin extends PluginConnector {
  socket: SocketIOClient.Socket

  constructor(profile: Profile & ExternalProfile) {
    super(profile)
  }

  protected connect(url: string): void {
    this.socket = io(url)
    this.socket.on('connect', () => {
      this.socket.on('message', (msg: Message) => this.getMessage(msg))
    })
```

(continues on next page)

(continued from previous page)

```
  }

  protected disconnect(): void {
    this.socket.close()
  }

  protected send(message: Partial<Message>): void {
    this.socket.send(message)
  }

}
```

Let's take a look :

- `connect` will be called when the plugin is activated.
- `disconnect` will be called when the plugin is deactivated.
- `send` will be called when another plugin calls the plugin's methods (on the server).
- `getMessage` should be called whenever a message arrives.

Checkout how to publish your plugin connector on npm.

CHAPTER

TWENTY

CREATE THE ENGINE

1. Create the Plugin Manager

The plugin manager can activate/deactivate plugins, and manages permissions between plugins.

```
import { PluginManager } from '@remixproject/engine';

const manager = new PluginManager()
```

1. Create the Engine

The engine manages the communication between plugins. It requires a `PluginManager`.

```
import { PluginManager, Engine } from '@remixproject/engine';

const manager = new PluginManager()
const engine = new Engine()
```

1. Register a plugin

We need to register a plugin before activating it. This is done by the `Engine`.

> **IMPORTANT** You need to register the "manager" before beeing able to activate a plugin

```
import { PluginManager, Engine, Plugin } from '@remixproject/engine';

const manager = new PluginManager()
const engine = new Engine()
const plugin = new Plugin({ name: 'plugin-name' })

// Register plugin
engine.register([manager, plugin])
```

1. Activate a plugin

Once your plugin is registered you can activate it. This is done by the `PluginManager`

```
const manager = new PluginManager()
const engine = new Engine()
const plugin = new Plugin({ name: 'plugin-name' })

// Register plugins
engine.register([manager, plugin])
```

(continues on next page)

(continued from previous page)

```
// Activate plugins
manager.activatePlugin('plugin-name')
```

Tested code available here

CHAPTER

TWENTYONE

PLUGINS COMMUNICATION

21.1 Methods

Each plugin can call methods exposed by other plugin. Let's see how to expose a method from one plugin and call it from another.

1. Create Plugin that exposes a method You can extend the `Plugin` class to create your own plugin. The list of exposed methods are defined in the field `methods` of the profile:

```
class FirstPlugin extends Plugin {
  constructor() {
    // Expose method "getVersion" to other plugins
    super({ name: 'first', methods: ['getVersion']})
  }
  // Implementation of the exposed method
  getVersion() {
    return 0
  }
}
```

1. Create a Plugin that calls the `getVersion` The `Plugin` class provides a `call` method to make a request to another plugin's methods

   The `call` method is available only when the plugin is activated by the plugin manager

```
class SecondPlugin extends Plugin {
  constructor() {
    super({ name: 'second' })
  }

  getFirstPluginVersion(): Promise<number> {
    // Call the methode "getVersion" of plugin "first"
    return this.call('first', 'getVersion')
  }
}
```

1. Register & activate plugins `Engine` & `PluginManager` can register & activate a list of plugins at once.

```
const manager = new PluginManager()
const engine = new Engine()
const first = new FirstPlugin()
const second = new SecondPlugin()
```

(continues on next page)

(continued from previous page)

```
//  Don't forget to wait for the manager to be loaded
await engine.onload()

// Register both plugins
engine.register([manager, first, second])

// Activate both plugins
await manager.activatePlugin(['first', 'second'])

// Call method "getVersion" of first plugin from second plugin
const firstVersion = await second.getFirstPluginVersion()
```

21.2 Events

Every plugin can emit and listen to events with :

- `emit`: Broadcast an event to all plugins listening.
- `on`: Listen to an event from another plugin.
- `once`: Listen once to one event of another plugin.
- `off`: Stop listening on an event that the plugin was listening to.

```
// Listen and broadcast "count" event
let value = 0
second.on('first', 'count', (count: number) => value = count)
first.emit('count', 1)
first.emit('count', 2)

// Stop listening on event
second.off('first', 'count')
```

CHAPTER

TWENTYTWO

FULL EXAMPLE

```
class FirstPlugin extends Plugin {
  constructor() {
    // Expose method "getVersion" to other plugins
    super({ name: 'first', methods: ['getVersion']})
  }
  // Implementation of the exposed method
  getVersion() {
    return 0
  }
}

class SecondPlugin extends Plugin {
  constructor() {
    super({ name: 'second' })
  }

  getFirstPluginVersion(): Promise<number> {
    // Call the methode "getVersion" of plugin "first"
    return this.call('first', 'getVersion')
  }
}


const manager = new PluginManager()
const engine = new Engine()
const first = new FirstPlugin()
const second = new SecondPlugin()

// Register both plugins
engine.register([manager, first, second])

// Activate both plugins
await manager.activatePlugin(['first', 'second'])

// Call method "getVersion" of first plugin from second plugin
const firstVersion = await second.getFirstPluginVersion()

// Listen and broadcast "count" event
let value = 0
second.on('first', 'count', (count: number) => value = count)
```

(continues on next page)

(continued from previous page)

```
first.emit('count', 1)
first.emit('count', 2)

// Stop listening on event
second.off('first', 'count')
```

Tested code available here

CHAPTER

TWENTYTHREE

HOSTED PLUGIN

If your plugin has a UI, you can specify where to host it. For that you need:

- A `HostPlugin` that manages the view.
- A `ViewPlugin` that displays the UI of your plugin.

23.1 Host Plugin

The Host plugin defines a zone on your IDE where a plugin can be displayed. It must exposes 3 methods:

- `addView`: Add a new view plugin in the zone.
- `removeView`: Remove an existing view plugin from that zone.
- `focus`: Focus the UI of the view on the zone.

  Adding a view doesn't focus the UI automatically, you need to trigger the `focus` method for that.

The way to add/draw element on the screen is different depending on your framework (LitElement, Vue, React, Angular, Svelte, ...). In this example we are going to use directly the standard Web API. Note that there is no support for WebGL yet, consider opening an issue if you're in this situation.

1. Create a `HostPlugin`

Let's extend the `HostPlugin` to create a zone on the side part of the screen:

```
// Host plugin display
class SidePanel extends HostPlugin {
  plugins: Record<string, HTMLElement> = {}
  focused: string
  root: Element
  constructor() {
    // HostPlugin automatically expose the 4 abstract methods 'focus', 'isFocus', 'addView',
↪'removeView'
    super({ name: 'sidePanel' })
  }
  currentFocus(): string {}
  addView(profile: Profile, view: HTMLElement) {}
  removeView(profile: Profile) {}
  focus(name: string) {}
}
```

Remix IDE defines two zone "sidePanel" & "mainPanel". We recommend using those two names as plugins working on Remix IDE will work on your IDE as well.

1. Define the root element of the `SidePanel`

The `root` element of a `HostPlugin` is the container node. Let's default it to the body element here.

```
constructor(root = document.body) {
  super({ name: 'sidePanel' })
  this.root = root
}
```

1. Implements `addView`

When a view plugin is added, the reference of the view plugin's HTMLElement is passed to the method.

```
addView(profile: Profile, view: HTMLElement) {
  this.plugins[profile.name] = view
  view.style.display = 'none'   // view is added but not displayed
  this.root.appendChild(view)
}
```

1. Implements `focus`

Here we want to display one specific view amongst all the views of the panel.

```
focus(name: string) {
  if (this.plugins[name]) {
    // Remove focus on previous view if any
    if (this.plugins[this.focused]) this.plugins[this.focused].style.display = 'none'
    this.plugins[name].style.display = 'block'
    this.focused = name
  }
}
```

1. Implements `currentFocus`

Return the name of the current focussed plugin in the Host Plugin.

```
currentFocus() {
  return this.focused
}
```

1. Implements `removeView`

We remove the view from the list, and remove the focus if it had it.

```
removeView(profile: Profile) {
  if (this.plugins[name]) {
    this.root.removeChild(this.plugins[profile.name])
    if (this.focused === profile.name) delete this.focused
    delete this.plugins[profile.name]
  }
}
```

23.2 ViewPlugin

Ok, now that we have our `HostPlugin` we can write a simple `ViewPlugin` to inject into.

A `ViewPlugin` must:

- have a `location` key in its profile, with the name of the `HostPlugin`.
- implement the `render` method that returns its root element.

```
class HostedPlugin extends ViewPlugin {
  root: HTMLElement
  constructor() {
    // Specific the location where this plugin is hosted
    super({ name: 'hosted', location: 'sidePanel' })
  }
  // Render the element into the host plugin
  render(): Element {
    if (!this.root) {
      this.root = document.createElement('div')
    }
    return this.root
  }
}
```

23.3 Instantiate them in the Engine

The `ViewPlugin` will add itself into its `HostPlugin` once activated.

Important: When activating a `HostPlugin` and a `ViewPlugin` with one call, the order is important (see comment in the code below).

```
const manager = new PluginManager()
const engine = new Engine()
const sidePanel = new SidePanel()
const hosted = new Host

// Register both plugins
engine.register([manager, sidePanel, hosted])

// Activate both plugins: ViewPlugin will automatically be added to the view
// The order here is important
await manager.activatePlugin(['sidePanel', 'hosted'])

// Focus on
sidePanel.focus('hosted')

// Deactivate 'hosted' will remove its view from the sidepanel
await manager.deactivatePlugin(['hosted'])
```

Do not deactive a `HostPlugin` that still manage activated `ViewPlugin`.

Tested code available here

CHAPTER

TWENTYFOUR

EXTERNAL PLUGINS

The superpower of the Engine is the ability to interact safely with external plugins.

Currently the Engine supports 2 communication channels:

- Iframe
- Websocket

The interface of these plugins is exacly the same as the other plugins.

24.1 Iframe

For the `IframePlugin` we need to specify the `location` where the plugin will be displayed, and the `url` which will be used as the source of the iframe.

The Engine can fetch the content of plugin hosted on IPFS or any other server accessible through HTTPS

```
const ethdoc = new IframePlugin({
  name: 'ethdoc',
  location: 'sidePanel',
  url: 'ipfs://QmQmK435v4io3cp6N9aWQHYmgLxpUejjC1RmZCbqL7MJaM'
})
```

24.2 Websocket

For the `WebsocketPlugin` you just need to specify the `url` as there is no UI to display.

This plugin is very useful for connecting to a local server like remixD, and an external API

```
const remixd = new WebsocketPlugin({
  name: 'remixd',
  url: 'wss://localhost:8000'
})
```

In the future, we'll implement more communication connection like REST, GraphQL, JSON RPC, gRPC, ...

CHAPTER

TWENTYFIVE

PLUGIN SERVICE

Each plugin can be broken down into small lazy-loaded services. This is a great way to provide a modular API to your plugin.

Let's look at a "Command Line Interface" plugin that would expose a "git" service.

```
const manager = new PluginManager()
const engine = new Engine()
const cmd = new Plugin({ name: 'cmd' })
const plugin = new Plugin({ name: 'caller' })

engine.register([manager, cmd, plugin])
await manager.activatePlugin(['cmd', 'caller'])

// Create a service inside cmd
// IMPORTANT: Your plugin needs to be activated before creating a service
await cmd.createService('git', {
  methods: ['fetch'],
  fetch: () => true,    // exposed
  commit: () => false   // not exposed
})

// Call a service
const fetched = await plugin.call('cmd.git', 'fetch')
```

IMPORTANT: Services are lazy-loaded. They can be created only *after activation*.

25.1 API

1. `createService`

A plugin can use `createService` to extends it's API.

```
const git = await cmd.createService('git', {
  methods: ['fetch'],
  fetch: () => true,    // exposed
  commit: () => false   // not exposed
})
```

A service can also use `createService` to create a deeper service.

```
await git.createService('deepGit', {
  methods: ['deepMethod'],
  deepMethod: () => console.log('Message from cmd.git.deepGit')
})
```

1. `call('name.service', 'method')`

To access a method from a plugin's service, you should use the name of the plugin and the name of the service separated by ".": `pluginName.serviceName`.

```
// Call a service
await plugin.call('cmd.git', 'fetch')
// Call a deep nested service
await plugin.call('cmd.git.deepGit', 'deepMethod')
```

Only the methods defined inside the `methods` key of the services are exposed. **If not defined, all methods are exposed**.

1. `on('name', 'event')`

The event listener does **not** require the name of the service because the event is actually emitted at the plugin level.

```
// Start lisening on event emitted by cmd plugin
plugin.on('cmd', 'committed', () => console.log('Committed!'))
const git = await cmd.createService('git', {})
// Service "git" from "cmd" emit event "committed"
git.emit('committed')
```

25.2 PluginService

For a larger service, you might want to use a class based interface. For that, your must extend the abstract `PluginService` class.

You need to specify at least the :

- `path`: name of the service.
- `plugin`: the reference to the root plugin the service is attached to.

```
// class based version
class GitService extends PluginService {
  path = 'git' // Name of the service
  methods = ['fetch']

  // Requires a reference to the plugin
  constructor(protected plugin: Plugin) {
    super()
  }

  fetch() {
    return true
  }

  commit() {
    return false
```

(continues on next page)

(continued from previous page)

```
  }
}

// Class based plugin
class CmdPlugin extends Plugin {
  git: GitService

  constructor() {
    super({ name: 'cmd' })
  }

  // On Activation if git service is not defined, creates it
  async onActivation() {
    if (!this.git) {
      this.git = await this.createService('git', new GitService(this))
    }
  }
}
```

In this example, we activate the service on activation, but **only the first time**.

Now let's register the plugin :

```
const manager = new PluginManager()
const engine = new Engine()
const plugin = new Plugin({ name: 'caller' })
const cmd = new CmdPlugin()

engine.register([manager, cmd, plugin])
await manager.activatePlugin(['cmd', 'caller'])

// Service is already created by the `onActivation` hook.
const fetched = await plugin.call('cmd.git', 'fetch')
```

CHAPTER

TWENTYSIX

ENGINE-NODE

This library was generated with Nx.

26.1 Running unit tests

Run `ng test engine-node` to execute the unit tests via Jest.

CHAPTER

TWENTYSEVEN

ENGINE-THEIA

This library was generated with Nx.

27.1 Running unit tests

Run `nx test engine-theia` to execute the unit tests via Jest.

CHAPTER

TWENTYEIGHT

ENGINE VSCODE

The **vscode engine** provides a list of connectors & plugins for a **plugin engine** that is built inside vscode.

```
npm install @remixproject/engine-vscode
```

28.1 Setup

You can use the remixproject engine to create a plugin system on top of a vscode extension. For that you need to create an engine and start registering your plugins.

> checkout @remixproject/engine documentation for more details.

```
import { Engine, Manager } from '@remixproject/engine';

export async function activate(context: ExtensionContext) {
  const manager = new Manager();
  const engine = new Engine();
}
```

28.2 Build-in plugins

`@remixproject/engine-vscode` comes with build-in plugins for `vscode`.

28.2.1 webview

The webview plugin opens a webview in the workspace and connects to it. The plugin must use `@remixproject/plugin-webview` to be able to establish connection.

```
import { WebviewPlugin } from '@remixproject/engine-vscode'
import { Engine, Manager } from '@remixproject/engine';

export async function activate(context: ExtensionContext) {
  const manager = new Manager();
  const engine = new Engine();
  const webview = new WebviewPlugin({
    name: 'webview-plugin',
    url: 'https://my-plugin-path.com',
```

(continues on next page)

(continued from previous page)

```
    methods: ['getData']
  }, { context }) // We need to pass the context as scond parameter
  engine.register([manager, webview]);
  // This will create the webview and inject the code inside
  await manager.activatePlugin('webview-plugin');
  const data = manager.call('webview-plugin', 'getData');
}
```

The url can be :

- remote
- absolute
- relative to the **extension file** (option.relativeTo === 'extension')
- relative to the **open workspace** (option.relativeTo === 'workspace')

  The url can also be local. In this case you must provide an **absolute path**.

Options

- `context`: The context of the vscode extension.
- `column`: The `ViewColumn` in which run the webview.
- `relativeTo`: If url is relative, is it relative to 'workspace' or 'extension' (default to 'extension')

28.2.2 terminal

The terminal plugin gives access to the current terminal in vscode.

```
import { TerminalPlugin } from '@remixproject/engine-vscode'
import { Engine, Manager } from '@remixproject/engine';

export async function activate(context: ExtensionContext) {
  const manager = new Manager();
  const engine = new Engine();
  const terminal = new TerminalPlugin()

  engine.register([manager, terminal]);
  await manager.activatePlugin('terminal');
  // Execute "npm run build" in the terminal
  manager.call('terminal', 'exec', 'npm run build');
}
```

28.2.3 Window

Provides access to the native window of vscode.

```
import { WindowPlugin } from '@remixproject/engine-vscode'
import { Engine, Manager } from '@remixproject/engine';

export async function activate(context: ExtensionContext) {
  const manager = new Manager();
  const engine = new Engine();
  const window = new WindowPlugin()

  engine.register([manager, window]);
  await manager.activatePlugin('window');
  // Open a prompt to the user
  const fortyTwo = await manager.call('window', 'prompt', 'What is The Answer to the␣
↪Ultimate Question of Life, the Universe, and Everything');
}
```

28.2.4 File Manager

Provides access to the file system through vscode api.

```
import { FileManagerPlugin } from '@remixproject/engine-vscode'
import { Engine, Manager } from '@remixproject/engine';

export async function activate(context: ExtensionContext) {
  const manager = new Manager();
  const engine = new Engine();
  const fs = new FileManagerPlugin()

  engine.register([manager, fs]);
  await manager.activatePlugin('filemanager');
  // Open a file into vscode
  // If path is relative it will look at the root of the open folder in vscode
  await manager.call('filemanager', 'open', 'package.json');
}
```

28.2.5 Theme

Remix's standard theme wrapper for vscode. Use this plugin to take advantage of the Remix's standard themes for your plugins. Otherwise, consider using vscode's color api directly in your webview.

```
import { ThemePlugin } from '@remixproject/engine-vscode'
import { Engine, Manager } from '@remixproject/engine';

export async function activate(context: ExtensionContext) {
  const manager = new Manager();
  const engine = new Engine();
  const theme = new ThemePlugin();
```

(continues on next page)

(continued from previous page)

```
  engine.register([manager, fs]);
  await manager.activatePlugin('theme');
  // Now your webview can listen on themeChanged event from the theme plugin
}
```

CHAPTER

TWENTYNINE

ENGINE WEB

The web engine provides a connector for Iframe & Websocket. `npm install @remixproject/engine-web`

29.1 Iframe

The iframe connector is used to load & connect a plugin inside an iframe. Iframe based plugin are webview using an `index.html` as entry point & need to use `@remixproject/plugin-iframe`.

```
const myPlugin = new IframePlugin({
  name: 'my-plugin',
  url: 'https://my-plugin-path.com',
  methods: ['getData']
})
engine.register(myPlugin);
// This will create the iframe with src="https://my-plugin-path.com"
await manager.activatePlugin('my-plugin');
const data = manager.call('my-plugin', 'getData');
```

Communication between the plugin & the engine uses the `window.postMessage()` API.

29.2 Websocket

The websocket connector wraps the native Websocket object from the Web API. Websocket based plugin are usually server with a Websocket connection open. Any library can be used, remixproject provide a wrapper around the `ws` library : `@remixproject/plugin-ws`.

```
const myPlugin = new WebsocketOptions({
  name: 'my-plugin',
  url: 'https://my-server.com',
  methods: ['getData']
}, {
  reconnectDelay: 5000 // Time in ms to wait to reconnect after a disconnection
});
engine.register(myPlugin);
// This will open a connection with the server. The server must be running first.
await manager.activatePlugin('my-plugin');
const data = manager.call('my-plugin', 'getData');
```

CHAPTER

THIRTY

PLUGIN-CHILD-PROCESS

This library was generated with Nx.

30.1 Running unit tests

Run `ng test plugin-child-process` to execute the unit tests via Jest.

CHAPTER

THIRTYONE

PLUGIN CORE

This is the core library used to create a new external plugin.

| Name | Latest Version | | ——————————| :——————: | | @remixproject/plugin | badge |

Use this library if you want to create a plugin **for a new environment**.

If you want to create a plugin in an existing envrionment, use the specific library. For example :

- Plugin on an iframe: @remixproject/plugin-iframe
- Plugin on a node child-process: @remixproject/plugin-child-process
- Plugin on an vscode extension or webview : @remixproject/plugin-vscode

31.1 API

| API | Description | | ——————————| :————————————————-: | | PluginClient | Entry point to communicate with other plugins |

31.2 Getting Started

This getting started is for building **iframe based plugin** (only supported by remix-ide for now).

Installation :

```
npm install @remixproject/plugin-iframe
```

or with a unpkg :

```
<script src="https://unpkg.com/@remixproject/plugin"></script>
```

31.2.1 Plugin Client

The plugin client is how you connect your plugin to remix.

To import (the ES6 way) with NPM use:

```
import { createClient } from '@remixproject/plugin'
const client = createClient()
```

Or if you are using unpkg use:

```
const { createClient } = remixPlugin
const client = createClient()
```

31.3 Test inside Remix IDE

To test your plugin with remix:

1. Go to http://remix-alpha.ethereum.org. (if your localhost is over HTTP, you need to use http for Remix IDE).
2. Click on the plugin manager (Plug icon on the left).
3. Click on "Connect to a Local Plugin".
4. Fill the profile info of you plugin ().
5. Click on "ok".
6. A new icon should appear on the left, this is where you can find you plugin.

31.3.1 Testing your plugin

You can test your plugin direcly on the alpha version of Remix-IDE. Go to the `pluginManager` (plug icon in the sidebar), and click "Connect to a Local Plugin".

Here you can add :

- A name : this is the name used by other plugin to listen to your events.
- A displayName : Used by the IDE.
- The url : May be a localhost for testing.

  Note: No need to do anything if you localhost auto-reload, a new `handshake` will be send by the IDE.

31.4 Status

Every plugin has a status object that can display notifications on the IDE. You can listen on a change of status from any plugin using `statusChanged` event :

```
client.on('fileManager', 'statusChanged', (status: Status) => {
  // Do Something
})
```

The status object is used for displaying a notification. It looks like that :

```
interface Status {
  key: number | 'edited' | 'succeed' | 'loading' | 'failed' | 'none'  // Display an icon␣
↪or number
  type?: 'success' | 'info' | 'warning' | 'error'  // Bootstrap css color
  title?: string  // Describe the status on mouseover
}
```

- If you want to remove a status use the `'none'` value for `key`.
- If you don't define type, it would be the default value ('info' for Remix IDE).

You can also change the status of your own plugin by emitting the same event :

```
client.emit('statusChanged', { key: 'succeed', type: 'success', title: 'Documentation␣
↪ready !' })
```

The IDE can use this status to display a notification to the user.

31.4.1 Client Options

CSS Theme

Remix is using Bootstrap. For better User Experience it's **highly recommanded** to use the same theme as Remix in your plugin. For that you *just* have to use standard bootstrap classes.

Remix will automatically create a `<link/>` tag in the header of your plugin with the current theme used. And it'll update the link each time the user change the theme.

If you really want to use your own theme, you can use the `customTheme` flag in the option :

```
const client = createClient({ customTheme: true })
```

Custom Api

By default `@remixproject/plugin` will use remix IDE api. If you want to extends the API you can specify it in the `customApi` option.

A good use case is when you want to use an external plugin not maintained by Remix team (3box plugin for example):

```
import { remixProfiles, IRemixApi } from '@remixproject/plugin'
interface ICustomApi extends IRemixApi {
  box: IBox;
}

export type CustomApi = Readonly<ICustomApi>;

export type RemixClient = PluginClient<any, CustomApi> & PluginApi<CustomApi>;

const customApi: ProfileMap<RemixIDE> = Object.freeze({
  ...remixProfiles,
  box: boxProfile
});
const client = createClient({ customApi })
```

You'll need Typescript > 3.4 to leverage the types.

DevMode

Plugins communicate with the IDE through the `postMessage` API. It means that `PluginClient` needs to know the origin of your IDE.

If you're developing a plugin with your IDE running on `localhost` you'll need to specify the port on which your IDE runs. By default the port used is *8080*. To change it you can do:

```
const devMode = { port: 3000 }
const client = createClient({ devMode })
```

CHAPTER

THIRTYTWO

CLIENT API

32.1 Loaded

`PluginClient` listen on a first handshake from the IDE before beeing able to communicate back. For that you need to wait for the Promise / callback `onload` to be called.

```
client.onload(() => /* Do something */)
client.onload().then(_ => /* Do Something now */)
await client.onload()
```

32.2 Events

To listen to an event you need to provide the name of the plugin you're listening on, and the name of the event :

```
client.on(/* pluginName */, /* eventName */, ...arguments)
```

For exemple if you want to listen to Solidity compilation :

```
client.on('solidity', 'compilationFinished', (target, source, version, data) => {
    /* Do Something on Compilation */
  }
)
```

Be sure that your plugin is loaded before listening on an event.

See all available event *below*.

32.3 Call

You can call some methods exposed by the IDE with with the method `call`. You need to provide the name of the plugin, the name of the method, and the arguments of the methods :

```
await client.call(/* pluginName */, /* methodName */, ...arguments)
```

Note: `call` is alway Promise

For example if you want to upsert the current file :

```
async function upsertCurrentFile(content: string) {
  const path = await client.call('fileManager', 'getCurrentFile')
  await client.call('fileManager', 'setFile', path, content)
}
```

Be sure that your plugin is loaded before making any call.

32.4 Emit

Your plugin can emit events that other plugins can listen on.

```
client.emit(/* eventName */, ...arguments)
```

Let's say your plugin build deploys a Readme for your contract on IPFS :

```
async function deployReadme(content) {
  const [ result ] = await ipfs.files.add(content);
  client.emit('readmeDeployed', result.hash)
}
```

> Note: Be sure that your plugin is loaded before making any call.

32.5 Expose methods

Your plugin can also exposed methods to other plugins. For that you need to extends the `PluginClient` class, and override the `methods` property :

```
class MyPlugin extends PluginClient {
  methods: ['sayHello'];

  sayHello(name: string) {
    return `Hello ${name} !`;
  }
}
const client = buildIframeClient(new MyPlugin())
```

> When extending the `PluginClient` you need to connect your client to the iframe with `buildIframeClient`.

You can find an exemple here.

CHAPTER

THIRTYTHREE

PLUGIN FRAME

Except if you want your plugin to ONLY work on the web, prefer @remixproject/plugin-webview

This library provides connectors to connect a plugin to an engine running in a web environment.

```
npm install @remixproject/plugin-iframe
```

If you do not expose any API you can create an instance like this :

```
import { createClient } from '@remixproject/plugin-iframe'

const client = createClient()
client.onload(async () => {
  const data = client.call('filemanager', 'readFile', 'ballot.sol')
})
```

If you need to expose an API to other plugin you need to extends the class:

```
import { createClient } from '@remixproject/plugin-iframe'
import { PluginClient } from '@rexmixproject/plugin'

class MyPlugin extends PluginClient {
  methods = ['hello']
  hello() {
    console.log('Hello World')
  }
}
const client = createClient()
client.onload(async () => {
  const data = client.call('filemanager', 'readFile', 'ballot.sol')
})
```

CHAPTER

THIRTYFOUR

PLUGIN-THEIA

This library was generated with Nx.

34.1 Running unit tests

Run `nx test plugin-theia` to execute the unit tests via Jest.

CHAPTER

THIRTYFIVE

PLUGIN VSCODE

This library provides connectors to run plugin in a vscode environment. Use this connector if you have a web based plugin that needs to run inside vscode.

Except if you want your plugin to ONLY work on vscode, prefer @remixproject/plugin-webview

```
npm install @remixproject/plugin-vscode
```

35.1 Webview

Similar to `@remixproject/plugin-iframe`, the webview connector will connect to an engine running inside vscode.

If you do not expose any API you can create an instance like this :

```
<script>
  const client = createClient(ws)
  client.onload(async () => {
    const data = client.call('filemanager', 'readFile', 'ballot.sol')
  })
</script>
```

If you need to expose an API to other plugin you need to extends the class:

```
<script>
  class MyPlugin extends PluginClient {
    methods = ['hello']
    hello() {
      console.log('Hello World')
    }
  }
  const client = createClient(ws)
  client.onload(async () => {
    const data = client.call('filemanager', 'readFile', 'ballot.sol')
  })
</script>
```

CHAPTER

THIRTYSIX

PLUGIN WEBVIEW

This library provides connectors to connect a plugin to an engine that can load webview or iframes.

```
npm install @remixproject/plugin-webview
```

If you do not expose any API you can create an instance like this :

```
import { createClient } from '@remixproject/plugin-webview'

const client = createClient()
client.onload(async () => {
  const data = client.call('filemanager', 'readFile', 'ballot.sol')
})
```

If you need to expose an API to other plugin you need to extends the class:

```
import { createClient } from '@remixproject/plugin-webview'
import { PluginClient } from '@rexmixproject/plugin'

class MyPlugin extends PluginClient {
  methods = ['hello']
  hello() {
    console.log('Hello World')
  }
}
const client = createClient()
client.onload(async () => {
  const data = client.call('filemanager', 'readFile', 'ballot.sol')
})
```

CHAPTER

THIRTYSEVEN

PLUGIN WEBWORKER

This library provides connectors to connect a plugin to an engine that can load webworkers.

```
npm install @remixproject/plugin-webworker
```

If you do not expose any API you can create an instance like this :

```
import { createClient } from '@remixproject/plugin-webworker'

const client = createClient()
client.onload(async () => {
  const data = client.call('filemanager', 'readFile', 'ballot.sol')
})
```

If you need to expose an API to other plugin you need to extends the class:

```
import { createClient } from '@remixproject/plugin-webworker'
import { PluginClient } from '@rexmixproject/plugin'

class MyPlugin extends PluginClient {
  methods = ['hello']
  hello() {
    console.log('Hello World')
  }
}
const client = createClient()
client.onload(async () => {
  const data = client.call('filemanager', 'readFile', 'ballot.sol')
})
```

CHAPTER

THIRTYEIGHT

PLUGIN WS

This library is a connector that connects a node server to using the `ws` library to the engine.

If you do not expose any API you can create an instance like this :

```
const wss = new WebSocket.Server({ port: 8080 });
wss.on('connection', (ws) => {
  const client = createClient(ws)
})
```

If you need to expose an API to other plugin you need to extends the class:

```
class MyPlugin extends PluginClient {
 methods = ['hello']
 hello() {
  console.log('Hello World')
 }
}
const wss = new WebSocket.Server({ port: 8080 });
wss.on('connection', (ws) => {
 const client = createClient(ws, new MyPlugin())
})
```

CHAPTER

THIRTYNINE

PLUGIN-UTILS

A simple utils library used by `@remixproject/engine` & `@remixproject/plugin`.