GalaChain SDK Docs

Gala Games

© 2024 Gala Games

Table of contents

1. GalaChain SDK 4
  1.1 Features 4
  1.2 Tutorials 4
  1.3 Working with GalaChain 4
  1.4 Reference documentation 4
  1.5 Documentation in PDF format 4
2. GalaChain 5
  2.1 What is GalaChain? 5
  2.2 Why is GalaChain? 5
  2.3 What kind of technology is GalaChain? 5
  2.4 When is GalaChain? 5
  2.5 Where do nodes come in? 5
  2.6 How fast is GalaChain? 5
  2.7 GalaChain for Publishers 5
  2.8 GalaChain for Developers 6
3. Getting started 7
  3.1 Option 1: Local Environment (Linux, MacOS, or Windows with WSL) 7
  3.2 Option 2: Use Docker image (Linux, MacOS or Windows) 8
  3.3 Option 3: Using Dev Containers (Linux or MacOS) 9
4. Troubleshooting 10
  4.1 Docker Desktop on Windows 10
  4.2 Docker 10
  4.3 WSL 11
5. Chaincode development 12
  5.1 Contract classes 12
  5.2 Transaction decorators 13
  5.3 Transaction context 13
  5.4 Authentication and authorization 14
  5.5 DTO types 15
  5.6 Objects saved on chain 15
  5.7 Error handling 16
  5.8 State cache 17
  5.9 Prevent duplicate calls 17
6. Chaincode Client 18
  6.1 Hyperledger Fabric Client 18

© 2024 Gala Games

6.2 REST API Client 19
6.3 Builder and actual client 20
6.4 Extending the client API 20
7. Testing your chaincode 21
7.1 Unit testing 21
7.2 Integration testing 23
8. Chaincode deployment 27
8.1 The process 27
8.2 Reference 28
9. Authorization 30
10. From zero to deployment with GalaChain SDK 31
10.1 1. Install the GalaChain CLI 31
10.2 2. Initialize the project from template 31
10.3 3. Update the contract (optional) 31
10.4 4. Prepare and publish chaincode docker image 32
10.5 5. Connect your chaincode with GalaChain network 32
10.6 6. Deploy the chaincode 32
10.7 7. Call REST API 32
11. Using Windows with WSL 34
11.1 How to use GalaChain with Windows Subsystem for Linux (WSL) 34

© 2024 Gala Games

1. GalaChain SDK

Welcome to developing with GalaChain! GalaChain is a layer 1 blockchain designed to be the foundation of Web3 Gaming, Entertainment and more.

1.1 Features

- Utility libraries to allow seamless development of chaincodes
- Local development environment with hot code reload and local block browser
- Easy start with chaincode template
- Integration with GalaChain

Read more about GalaChain.

1.2 Tutorials

- **From zero to deployment**
- Getting started guide

1.3 Working with GalaChain

- Chaincode development
- Chaincode testing
- Chaincode deployment
- Authorization
- Chaincode client

1.4 Reference documentation

- `chain-api` - Common types, DTOs (Data Transfer Objects), APIs, signatures, and utils for GalaChain
- `chain-client` - GalaChain client library
- `chain-test` - Unit testing and integration testing for GalaChain
- `chaincode` framework - Framework for building chaincodes on GalaChain

1.5 Documentation in PDF format

- PDF file

© 2024 Gala Games

2. GalaChain

2.1 What is GalaChain?

GalaChain is a layer 1 blockchain designed to be the foundation of Web3 Gaming, Entertainment and more.

2.2 Why is GalaChain?

When Gala Games began integrating blockchain technology with games, we quickly realized that existing blockchain technology was not built to support the kinds of functionality that players and developers desired in Web3 gaming. We set out on a quest to build something different. As Gala has evolved, we recognize even more Web3 use cases that GalaChain is perfect for. We're building a foundation for Web3 Gaming, Entertainment and beyond.

2.3 What kind of technology is GalaChain?

The core technology that GalaChain is built on is Hyperledger Fabric. We have built infrastructure and code to add capabilities to easily onboard games and users. Now it's very straightforward to write contracts using typescript. We've also created a Token contract that can be implemented in any channel. This immediately gives the channel access to native Token operations such as transfer, mint, allowances, swapping, lending, and more.

2.4 When is GalaChain?

Now! GalaChain has already been integrated into live products including SpiderTanks, Music, PokerGO, and Champions Arena. Many more are being onboarded in preparation for launch. Additionally, we're working incrementally toward broader public access and participation.

2.5 Where do nodes come in?

We've got a lot of ideas about that! The first GalaChain workload that will be on nodes is likely to be one that helps perform bridge transaction verifications. More info coming soon!

2.6 How fast is GalaChain?

Super-fast (or slow). One of the cool features of GalaChain is that each channel can be configured for different performance options. Maybe your product wants to drop big chunks of data in each block and there's time between transactions, or you could have small data, moving really fast. It's configurable! We typically don't talk about tx/s simply because it's relative to the use case and so one data point may not be useful for everyone.

2.7 GalaChain for Publishers

- Managed end-to-end blockchain solution
- Managed infrastructure
- Framework to lower chaincode development costs
- Basic token features out of the box
- Monetization with fees
- Built in bridge

© 2024 Gala Games

2.8 GalaChain for Developers

- A framework and a set of tools for easy chaincode development
- Open Source SDK, battle tested at Gala
- Local development tools
- Standardized and documented REST API
- Testnet and Mainnet
- CLI to manage the whole development cycle

© 2024 Gala Games

3. Getting started

3.1 Option 1: Local Environment (Linux, MacOS, or Windows with WSL)

If you are using Windows with WSL don't forget to enable integration with WSL on Docker Desktop.

How to use Windows with WSL

3.1.1 Requirements

You need to have the following tools installed on your machine:

- Node.js 18+
- Docker and Docker Compose
- jq and yq

3.1.2 1. Install our CLI

```
npm i -g @gala-chain/cli
```

Check the CLI:

```
galachain --help
```

3.1.3 2. Initialize your project

```
galachain init <project-name>
```

It will create a sample project inside `<project-name>` directory.

Install all dependencies:

```
npm i
```

3.1.4 3. Start the network

```
npm run network:start
```

The network will start in hot-reload/watch mode, so leave the prompt with logs running and execute the following commands in a new prompt.

3.1.5 4. Run integration tests

Now you can run integration tests with:

```
npm run test:e2e
```

3.1.6 5. Verify changes in block browser and GraphQL

Navigate to http://localhost:3010/blocks to see our block browser which allows you to see what's saved on your local GalaChain network.

Navigate to http://localhost:3010/graphiql to interact with GraphQL and execute queries.

© 2024 Gala Games

3.1.7 6. Next steps

- Iterate on your chaincode
- Get familiar with GalaChain SDK
- Deploy chaincode to gc-testnet

3.2 Option 2: Use Docker image (Linux, MacOS or Windows)

3.2.1 Requirements

- Docker Desktop or Docker CLI.
- [Optional] VS Code with Dev Containers extension.

3.2.2 1. Run the Docker image

```
docker run --privileged -d -p 3010:3010 -it --name <container_name> ghcr.io/galachain/sdk:latest
```

Make sure the container is up and running. The Docker image initializes a new project with the name `chaincode-template` by default.

3.2.3 2. Open the running container

2.1 Open the container with bash

```
docker exec -ti <container_name> /bin/bash
```

2.2 Open the container with VSCode (Requires VSCode and Dev Containers Extension)

Open VSCode and press F1 to open the Command Palette and search for `Dev Containers: Attach to Running Container`

After attach the container you may have to open the project folder manually.

3.2.4 3. Start the network

Once the terminal is open, start the network with:

```
npm run network:start
```

The network is going to start in dev mode and the prompt will be left showing the logs, so don't close the prompt and open new ones to proceed with the following commands.

3.2.5 4. Run integration tests

Now you can run integration tests with:

```
npm run test:e2e
```

3.2.6 5. Verify changes in block browser and GraphQL

Navigate to http://localhost:3010/blocks to see our block browser which allows you to see what's saved on your local GalaChain network.

Navigate to http://localhost:3010/graphiql to interact with GraphQL and execute queries.

© 2024 Gala Games

3.3 Option 3: Using Dev Containers (Linux or MacOS)

3.3.1 Requirements

- VSCode
- Dev Containers Extension
- Node.js
- Docker

3.3.2 1. Install our CLI

```
npm i -g @gala-chain/cli
```

Check the CLI:

```
galachain --help
```

3.3.3 2. Initialize your project

```
galachain init <project-name>
```

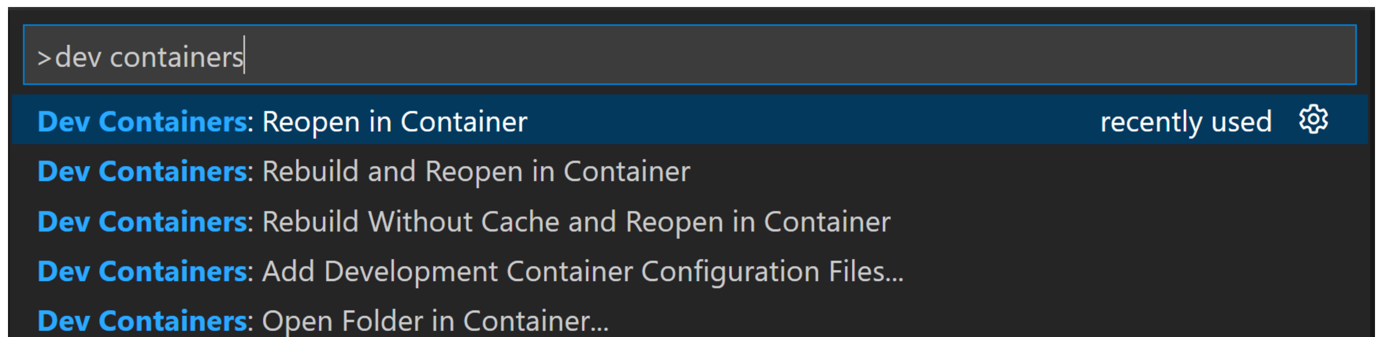
It will create a sample project inside `<project-name>` directory.

Open the directory on VSCode.

```
cd <project-name>
code .
```

3.3.4 3. Open in a Dev Container

While in VSCode, press F1 to open the Command Palette and search for `Dev Containers: Reopen in Container`



You can also click on the Remote Indicator in the status bar to get a list of the most common commands.

© 2024 Gala Games

3.3.5 4. Install dependencies and start network

Open a new prompt when in a Dev Conatiner and run the commands:

```
npm install
```

```
npm run network:start
```

The network will start in dev mode, so leave the prompt with logs running and execute the following commands in a new prompt.

3.3.6 5. Run integration tests

Now you can run integration tests with:

```
npm run test:e2e
```

3.3.7 6. Verify changes in block browser and GraphQL

Navigate to http://localhost:3010/blocks to see our block browser which allows you to see what's saved on your local GalaChain network.

Navigate to http://localhost:3010/graphiql to interact with GraphQL and execute queries.

---

4. Troubleshooting

4.1 Docker Desktop on Windows

If you are using Windows with WSL don't forget to enable integration with WSL on Docker Desktop.

```
Docker Desktop > Settins > Resources > WSL Integration
```

Docker: image operating system "linux" cannot be used on this platform: operating system is not supported.

Some versions of the Docker Desktop for Windows have a bug that prevents the use of Linux images. If you are facing this issue, you can use the WSL2 backend to run Docker. To do so, go to Docker Desktop > Settings > General and select WSL2 as the default backend.

Docker: "no matching manifest for windows/amd64 in the manifest list entries".

To bypass this issue you can run the Docker daemon in experimental mode:

```
Docker Desktop > Settins > Docker Engine > Edit the Docker daemon file > Set the "experimental": true > Apply & Restart
```

4.2 Docker

Docker: Error response from daemon: Conflict. The container name "/" is already in use by container "".

You have to remove (or rename) that container to be able to reuse that name.

© 2024 Gala Games

4.3 WSL

./fablo-target/fabric-config/configtx.yaml: no such file or directory

Make sure you are running it as a administrator of the cmd or powershell.

docker: Got permission denied

If you get a `docker: Got permission denied` error when running npm run network:start or npm run network:up, you may need to enable the configuration `Settings` > `Resources` > `Expose daemon on tcp://localhost:2375 without TLS` .

© 2024 Gala Games

5. Chaincode development

The GalaChain SDK allows you to write Hyperledger Fabric chaincodes in TypeScript in a more convenient way, while adjusting them to the GalaChain platform.

Key features: - Contract classes - Transaction decorators - Transaction context - Authentication and authorization - DTO types - Objects saved on chain - Error handling - State cache - Recommended project structure - Tracing support

All samples in this document come from the GalaChain chaincode template. You can find the template in our source code in `chain-cli/chaincode-template` directory, or initialize it with the `galachain init` command.

5.1 Contract classes

The GalaChain SDK allows developers to write chaincodes in an object-oriented way. It reuses the concept of contract classes and contract methods from the Hyperledger Fabric Contract API. Typically, a contract class is a TypeScript class that extends the `GalaContract` class from the `@gala-chain/chaincode` library. It is recommended to treat each contract class as a controller in the MVC pattern (Model, View, Controller) and minimize the logic within it.

Sample contract class:

```
import { Evaluate, GalaChainContext, GalaContract, Submit } from "@gala-chain/chaincode";
import { version } from "../../package.json";
import { AppleTreeDto, FetchTreesDto, PagedTreesDto, fetchTrees, plantTree } from "../apples";

export class AppleContract extends GalaContract {
  constructor() {
    super("AppleContract", version);
  }

  @Submit({
    in: AppleTreeDto
  })
  public async PlantTree(ctx: GalaChainContext, dto: AppleTreeDto): Promise<void> {
    await plantTree(ctx, dto);
  }

  @Evaluate({
    in: FetchTreesDto,
    out: PagedTreesDto
  })
  public async FetchTrees(ctx: GalaChainContext, dto: FetchTreesDto): Promise<PagedTreesDto> {
    return await fetchTrees(ctx, dto);
  }
}
```

`GalaContract` is a base class for all contract classes. It provides several features: - It ensures that all contract methods have access to the proper transaction context (`GalaChainContext` , see Transaction decorators). - It adds common methods for a contract: `GetContractVersion` , `GetContractAPI` , `GetObjectByKey` , and `GetObjectHistory` . - It saves all writes from the GalaChain state cache to the ledger at the end of a successful transaction (see State cache). - It enhances tracing (see Tracing support).

The constructor `GalaContract` class requires two parameters: `name` and `version` . `name` is a name of the contract, and `version` is a version of the contract. Typically, you can read the version from the `package.json` file and version numbers conventionally follow the npm / semver standards.

Each method of the contract class require two parameters: `ctx` and `dto` . `ctx` is a transaction context, an object that extends Hyperledger Fabric `Context` class. Aside from the standard Fabric context, it provides some additional methods and properties (see Transaction context).

The second parameter, `dto` , is an object that contains all parameters of the transaction (see DTO types).

Also, all contract methods are decorated with `@Submit` , `@Evaluate` , or `@GalaTransaction` decorators (see Transaction decorators). These decorators are required for various reasons. For instance, they allow you to properly expose the contract methods in GalaChain, deserialize and validate input parameters, normalize the response, handle authorization, etc.

© 2024 Gala Games

5.2 Transaction decorators

Transaction decorators enhance the contract methods with various features: - They allow to properly expose the contract methods API in GalaChain. - They deserialize and validate the DTO before method is called. - They normalize the chaincode method output from any type to `GalaChainResponse`. - They handle authorization. - They can be used to ensure uniqueness of the transaction in case of duplicate calls. - They can be used to define actions that should be executed before and after the transaction.

GalaChain defines three decorator types: `@Submit`, `@Evaluate`, and `@GalaTransaction`.

- `@Submit` decorator is used for contract methods that modify the ledger state.
- `@Evaluate` decorator is used for contract methods that only read the ledger state.
- `@GalaTransaction` decorator is used for both types of contract methods, but is more verbose. It is recommended to use `@Submit` and `@Evaluate` decorators instead.

All decorators support the following parameters: - `in` - input DTO class that extends `GalaChainDto` class from `@gala-chain/chaincode` library (default: `ChainCallDTO` class). This parameter is used to properly deserialize and validate the input parameters of the transaction, and to properly expose the contract method API in GalaChain. It is highly recommended to provide a custom dto class as a parameter, otherwise the validation won't work at all: There will be issues with deserialization of non-standard input parameters like nested classes, `BigDecimal` values etc. - `out` - output type of the chaincode method (default: `"null"`). It might be a string representing the type (`"number"`, `"string"`, `"boolean"`, `"null"`, `"object"`), or a custom class, or an object `{ "arrayOf": X }` where `X` is a string representing the type or a custom class. This parameter is used to properly expose the contract method API in GalaChain. - `description` - optional description of the contract method that is presented in GalaChain contract method API definition. - `allowedOrgs` - optional parameter to define which organizations are allowed to call the contract method. It is a string array with organization names. If not provided, all organizations are allowed to call the contract method. - `apiMethodName` - optional name of the contract method that should be used in the GalaChain REST API. If not provided, the name of the contract method is used. - `sequence` - optional parameter for advanced use cases. It means that the method call should actually be defined as a sequence of calls. It is useful when a GalaChain REST API call should consist of multiple calls, and each call should be executed in a separate transaction in a separate block. The sequence of calls is handled by GalaChain REST API. - `enforceUniqueKey` - ensures that DTO contain a `uniqueKey` property, which is required to prevent duplicate calls (see Prevent attacks or bad data state from duplicate calls). - `before` - optional parameter defining a function to be executed before the actual transaction (but after the authorization). - `after` - optional parameter defining a function to be executed after the actual transaction (but before the state cache is saved to the ledger).

Additionally, `@GalaTransaction` decorator supports `type` and `verifySignature` parameters. `type` can be `GalaTransactionType.SUBMIT` or `GalaTransactionType.EVALUATE` and means whether the transaction is a submit or evaluate transaction. `verifySignature` can be `true` or `false` and means whether the transaction should be verified against the signature. It is NOT recommended to use `verifySignature` as `false`, because it disables authorization for the transaction.

`@Submit` decorator is a shortcut for `@GalaTransaction({ type: GalaTransactionType.SUBMIT, verifySignature: true })`. `@Evaluate` decorator is a shortcut for `@GalaTransaction({ type: GalaTransactionType.EVALUATE, verifySignature: true })`.

5.3 Transaction context

`GalaChainContext` is an object that extends Hyperledger Fabric `Context` class. Asides from standard Fabric context, it provides some additional methods and properties:

- `callingUser` - returns standardized user id with prefix and actual name (note calling user is something different, than user in Fabric CA; see Authentication and authorization).
- `callingUserEthAddress` - returns eth address that is derived from calling user public key (see Authentication and authorization).
- `txUnixTime` - returns unix time of the transaction.
- `span` - returns tracing span of the transaction (see Tracing support).

`GalaChainContext` also changes the behavior of the `stub` property. In a standard Fabric context, the `stub` property returns a `ChaincodeStub` object. In a GalaChain context, the `stub` property returns a proxy object that wraps `ChaincodeStub` in a way to support caching (see State cache).

© 2024 Gala Games

Finally, it adds some customization to the `logger` property.

5.4 Authentication and authorization

A method call requires authorization when it is marked with `@Submit` or `@Evaluate` decorator, or with `@GalaTransaction` with `verifySignature: true`.

Authorization is handled chaincode-side, on the basis of secp256k1 signature of the transaction. GalaChain recovers the public key from the signature, and derives the corresponding eth address from the public key. Then, GalaChain checks whether the eth address is registered in GalaChain as a user.

If the user is registered, `ctx.callingUser` and `ctx.callingUserEthAddress` properties are set in the transaction context. `ctx.callingUser` is a standardized user id with prefix and actual name. It may be `eth|<user-eth-address>` or `client|<user-alias>`, depending on the way the user was registered (`RegisterUser` and `RegisterEthUser` respectively).

If the user is not registered, or the signature is missing or invalid, then the transaction is rejected.

Additional notes: * If a method is exposed but (1) does not require authorization (marked with `@GalaTransaction` with `verifySignature: false`), and (2) its DTO does not have a signature, then `ctx.callingUser` contains the Fabric CA username (`client|<ca-username>`) and executing `ctx.callingUserEthAddress` throws an exception. * If a method is exposed, does not require authorization, and the DTO has a signature, then the regular authorization flow is performed.

5.4.1 Additional notes about signatures

A JSON payload to be signed is created from a DTO object without `signature` and `trace` properties, with its keys sorted alphabetically, and no end of line character(s) at the end. (Further reading as to why the must be the case can be found in the official Hyperledger Fabric documentation). Sample `jq` command to produce valid data to sign: `jq -cSj "." dto-file.json`.

Also, all `BigNumber` data should be provided as strings (not numbers or directly serialized BigNumber objects) with fixed decimal point notation.

The EC secp256k1 signature should be created for keccak256 hash of the data. The recommended format of the signature is a HEX encoded string, including r + s + v values. Signature in this format is supported by `ethers` library.

Sample signature:

```
b7244d62671319583ea8f30c8ef3b343cf28e7b7bd56e32b21a5920752dc95b94a9d202b2919581bcf776f0637462cb67170828ddbcc1ea63505f6a211f9ac5b1b
```

GalaChain also supports DER encoded signatures for authorization, but since the DER signature does not contain `v` value (the recovery part), you need to additionally provide `signerPublicKey` parameter to the transaction DTO.

Sample DER signature (first line), and the corresponding `signerPublicKey` (second line):

```
3045022100b7244d62671319583ea8f30c8ef3b343cf28e7b7bd56e32b21a5920752dc95b902204a9d202b2919581bcf776f0637462cb67170828ddbcc1ea63505f6a211f9ac5b
04fa7d9e30902207fd821a1518ce777e1935a45e52180d6a6339f37c3e3f759d1a64e33ed1e334070d37731f6ce3f4a5daa6ee4c9884f21860601fed892d40b2a9
```

5.4.2 Restricting access by organization name

You can restrict access to a contract method by organization name using `allowedOrgs` parameter of the transaction decorators. It is a string array with organization MSP names.

For example, if you want to allow only `CuratorOrg` and `FarmerOrg` to call a contract method, you can use:

GalaChain authorization will check whether the Fabric CA user which called the transaction belongs to one of the allowed organizations. Thus, the check is not related with user profile saved on chain, but related with the CA user which called the transaction.

```
@Submit({
  in: AppleTreeDto,
  allowedOrgs: ["CuratorOrg", "FarmerOrg"]
})
```


© 2024 Gala Games

Additionally, if you don't want to hardcode the organization names in the contract code, you can use `AUTHORITY_ORG_NAME` const from `@gala-chain/chaincode` library. It takes the organization name from `AUTHORITY_ORG_NAME` environment variable (which defaults to `CuratorOrg`).

5.5 DTO types

We consider DTO as an object that contains all parameters of the transaction (transaction input parameters). It is passed as a second parameter to the contract method and deserialized with the use of transaction decorator `in` parameter.

Each DTO class should extend `ChainCallDTO` class from `@gala-chain/chaincode` library. It defines some additional fields that are required for GalaChain to properly handle the transaction: - `signature` - optional signature of the transaction. It is required for authorization (see Authentication and authorization). - `signerPublicKey` - optional signer public key of the transaction. It is required for authorization when the transaction is signed with DER signature (see Authentication and authorization). - `uniqueKey` - optional unique key of the transaction. It is required to prevent duplicate calls (see Prevent duplicate calls). - `trace` - optional tracing span of the transaction (see Tracing support).

Sample DTO class:

```
import { ChainCallDTO, StringEnumProperty } from "@gala-chain/api";
import { Type } from "class-transformer";
import { ArrayNotEmpty, ValidateNested } from "class-validator";

export enum Variety {
  GALA = "GALA",
  GOLDEN_DELICIOUS = "GOLDEN_DELICIOUS"
}

export class AppleTreeDto extends ChainCallDTO {
  @StringEnumProperty(Variety)
  public readonly variety: Variety;

  public readonly index: number;
}

export class AppleTreesDto extends ChainCallDTO {
  @ValidateNested({ each: true })
  @Type(() => AppleTreeDto)
  @ArrayNotEmpty()
  public readonly trees: AppleTreeDto[];
}
```

GalaChain uses `class-transformer` and `class-validator` libraries for DTO serialization and validation. It also provides some additional decorators for DTO properties, like `@StringEnumProperty`, or `@BigNumberProperty`. You should consult the documentation of these libraries, especially for more complex use cases (including but not limited to): Nested objects, arrays of objects, etc. (note decorators for `trees` property in the sample above).

Optionally, you can provide a `@JSONSchema` decorator from the `class-validator-jsonschema` library, either for whole DTO class, or for each property. It is used to generate a JSON schema for the DTO, which is used in GalaChain REST API.

5.6 Objects saved on chain

GalaChain uses the same validation and serialization libraries for objects saved on chain as for DTOs (`class-transformer` and `class-validator`). Accordingly, you can use the same decorators for objects saved on chain as for DTOs.

```
import { BigNumberProperty, ChainKey, ChainObjectBase, StringEnumProperty } from "@gala-chain/api";
import BigNumber from "bignumber.js";
import { IsString } from "class-validator";
import { Variety } from "./types";

export class AppleTree extends ChainObject {
  static INDEX_KEY = "GCAPPL";

  @ChainKey({ position: 0 })
  @IsString()
  public readonly plantedBy: string;

  @ChainKey({ position: 1 })
  @StringEnumProperty(Variety)
  public readonly variety: Variety;

  @ChainKey({ position: 2 })
  public readonly index: number;
```


© 2024 Gala Games

```
  public readonly plantedAt: number;

  @BigNumberProperty()
  public applesPicked: BigNumber;
}
```

Aside from standard validation and serialization, GalaChain provides a `@ChainKey` decorator. It is used to define parts of the key of the object saved on chain. For instance in the sample above, the key of the object saved on chain consists of `INDEX_KEY`, `plantedBy`, `variety`, and `index` properties. Since it is build from multiple properties, it is called a composite key.

Consider you have `appleTree` which is an instance of `AppleTree` class, and you want to save it on chain. You can use `putChainObject` method from `@gala-chain/chaincode` library:

```
await putChainObject(ctx, appleTree);
```

If you want to delete it from chain, you can use `deleteChainObject` method:

```
await deleteChainObject(ctx, appleTree);
```

If you have a key of the object saved on chain (`key`), you can use `getChainObject` method to get the object from chain (`AppleTree` class is required to properly deserialize the object):

```
await getObjectByKey(ctx, AppleTree, key);
```

You can get object history with `getObjectHistory` method:

```
await getObjectHistory(ctx, key);
```

And you can check if the object exists on chain with `objectExists` method:

```
await objectExists(ctx, key);
```

Additionally, since GalaChain uses composite keys, you can get all objects with the same prefix using the `getObjectByPartialCompositeKey` method. For instance, if you want to get all gala apple trees planted by a farmer, you can use:

```
await getObjectByPartialCompositeKey(ctx, AppleTree.INDEX_KEY, ["farmer1", Variety.GALA], AppleTree);
```

There is also a relevant method that uses pagination (can be used only read-only transactions): `getObjectByPartialCompositeKeyWithPagination`.

5.6.1 Ranged objects

GalaChain also supports ranged objects. Ranged objects do not use composite keys, so they can be used in Hyperledger Fabric range queries.

Instead of `ChainObject` class, you should use `RangedChainObject` class. Then, you can use `@ChainKey` decorator to define the key parts of the object saved on chain the same way as for `ChainObject` class. In order to put ranged object on chain, you should use `putRangedChainObject` method.

5.7 Error handling

We recommend handling errors with exceptions. The GalaChain SDK provides a `ChainError` class that extends the Node.js `Error` class, which additionally contains: * `code` property mapped to a corresponding HTTP code in GalaChain REST API. * `key` property which is an autogenerated string key from the error class name (for easier debugging). * `payload` property which is an optional object with additional information about the error.

The GalaChain SDK provides also several predefined error classes which contain proper `code` values: `ValidationError`, `UnauthorizedError`, `PaymentRequiredError`, `ForbiddenError`, `NotFoundError`, `ConflictError`, `NoLongerAvailableError`, `DefaultError`, `RuntimeError`, `NotImplementedError`. You may use them in your code or (preferably) create your own error classes that extend one of the predefined error classes.

© 2024 Gala Games

When a contract method throws an error: * no state changes are saved to the ledger; * the error is logged; * the error is automatically handled, so the response is always a `GalaChainResponse` object (in case of error the response object contains error properites `Status`, `Message`, `ErrorCode`, `ErrorKey`, and `ErrorPayload`); * the transaction is saved on the ledger in transaction history.

5.8 State cache

When you get state in Hyperledger Fabric, it always returns the latest value from the ledger. When you update the state in a method, and get it again in the same method, it returns the same value as before the update. To avoid this behavior, the GalaChain SDK has a built-in state cache.

This way, when you get state in a transaction method, and update it in the same method, the second get returns the updated value.

The state cache also prevents inconsistent state in case of exceptions. Since all state changes are flushed to the ledger only at the end of a successful transaction, if an exception is thrown, the state is not updated.

5.9 Prevent duplicate calls

Accidental (or maliciously intentional) duplicate calls of some transactions could potentially lead to bad data states, spend of additional token quantities, application layer vulnerabilities, or other ill effects. To prevent this class of problems, DTOs can contain a `uniqueKey` property. It is an optional string, provided client-side, that is used to prevent duplicate calls. If the same `uniqueKey` is provided in two different transactions, the second transaction is rejected with `UniqueTransactionConflictError` error.

© 2024 Gala Games

6. Chaincode Client

The `@gala-chain/client` package provides a client for interacting with the chaincode. Currently, it supports the following client types: * client for interacting directly with the Hyperledger Fabric network, built on top of the `fabric-network` and `fabric-ca-client` packages; * client for interacting with the chaincode via REST API that meets the GalaChain REST API specification, used internally at GalaGames, and is also compatible with the slightly different REST API exposed by Fablo REST.

All client types share the same API, so it is easy to switch between them, depending on your needs.

Also, `@gala-chain/client` package is designed to be lightweight. This is why `fabric-network` and `fabric-ca-client` dependencies are marked as optional `peerDependencies` and should be installed separately.

6.1 Hyperledger Fabric Client

In order to connect to the Hyperledger Fabric network, you need to provide the following configuration: 1. `HFClientParams` - information containing basic information about network topology and credentials for connecting to the network; 2. `ContractConfig` - information about the chaincode that will be used to interact with the network. 3. Optionally, a custom API specification to make the client type-safe.

6.1.1 `HFClientConfig`

The `HFClientConfig` interface defines parameters that are required to connect to the Hyperledger Fabric network.

```
```typescript
const params: HFClientConfig = {
  orgMsp: "PartnerOrg1",
  userId: "admin",
  userSecret: "adminpw",
  connectionProfilePath: path.resolve(networkRoot, "connection-profiles/cpp-partner.json")
};
```

- `orgMsp` - Hyperledger Fabric MSP name of the organization that the client will connect to;
- `userId` - id of the user in Fabric CA that will be used to connect to the network;
- `userSecret` - password/secret of the user in CA;
- `connectionProfilePath` - path to the connection profile file that describes the network topology.

Both `adminId` and `adminPass` are required to authorize the client with the network. If they are not provided, the client will try to get them from the environment variables `ADMIN_ID` and `ADMIN_PASS` respectively.

The `connectionProfilePath` should refer to a valid connection profile JSON file. For local development, you can use the connection profile provided in the `<network-root>/connection-profiles` directory of the network generated by GalaChain CLI.

6.1.2 `ContractConfig`

The `ContractConfig` interface defines parameters that are required to interact with the chaincode.

```
const contract: ContractConfig = {
  channelName: "product-channel",
  chaincodeName: "basic-product",
  contractName: "PublicKeyContract"
};
```

- `channelName` - name of the channel that the client will connect to;
- `chaincodeName` - name of the chaincode that the client will use to interact with the network;
- `contractName` - name of the contract that the client will use to interact with the chaincode.

© 2024 Gala Games
```

6.1.3 Creating the client

```
const client: ChainClient = gcclient
  .forConnectionProfile(params)
  .forContract(contract);
```

The client creation is a two-step process. First, you need to create a client builder instance using the `forConnectionProfile` method. Then the `forContract` method returns the actual client instance.

As a result, you get a `ChainClient` instance that can be used to interact with the chaincode. It supports `evaluateTransaction` and `submitTransaction` methods that are used to invoke chaincode functions.

After you end interacting with the chaincode, you should disconnect the client:

```
await client.disconnect();
```

Otherwise, the client will keep the GRPC connection to the network open.

6.2 REST API Client

The REST API client is used to interact with the chaincode via REST API, that matches the specification of managed infrastructure of GalaChain.

In order to connect to the REST API, you need to provide the following configuration: 1. `RestApiClientConfig` - information containing basic information about path mapping and credentials for connecting to the network; 2. `ContractConfig` - information about the chaincode that will be used to interact with the network. 3. Optionally, a custom API specification to make the client type-safe.

6.2.1 `RestApiClientConfig`

The `RestApiClientConfig` interface defines parameters that are required to connect to the REST API.

```
const params: RestApiClientConfig = {
  apiUrl: "http://localhost:3000/api",
  configPath: path.resolve(__dirname, "api-config.json")
};
```

- `orgMsp` - Hyperledger Fabric MSP name of the organization that the client will connect to;
- `apiUrl` - URL of the REST API;
- `configPath` - path to the configuration file that describes path mapping for channels, chaincodes, and contracts. Sample configuration file can be found in the `e2e` directory of the chaincode generated from template by GalaChain CLI.

6.2.2 `ContractConfig`

The `ContractConfig` interface defines parameters that are required to interact with the chaincode.

```
const contract: ContractConfig = {
  channelName: "product-channel",
  chaincodeName: "basic-product",
  contractName: "PublicKeyContract"
};
```

- `channelName` - name of the channel that the client will connect to;
- `chaincodeName` - name of the chaincode that the client will use to interact with the network;
- `contractName` - name of the contract that the client will use to interact with the chaincode.

6.2.3 Creating the client

```
const client: ChainClient = gcclient
  .forApiConfig(params)
  .forContract(contract);
```


© 2024 Gala Games

The client creation is a two-step process. First, you need to create a client builder instance using the `forConnectionProfile` method. Then the `forContract` method returns the actual client instance.

As a result, you get a `ChainClient` instance that can be used to interact with the chaincode. It supports `evaluateTransaction` and `submitTransaction` methods that are used to invoke chaincode functions.

After you end interacting with the chaincode, you should disconnect the client:

```
await client.disconnect();
```

6.3 Builder and actual client

For all high-level operations, the client uses the `Builder` pattern: 1. first, you create a builder instance using the `forConnectionProfile` or `forApiConfig` method; 2. then you configure the builder instance using the `forContract` method.

Since all `ChainClient` builders share the same interface, you can just parametrize the builder type and use the same code for all client types, for instance:

```
const builder: ChainClientBuilder = process.env.USE_REST_API === "true"
  ? gcclient.forApiConfig(...)
  : gcclient.forConnectionProfile(...);

const client: ChainClient = builder.forContract(...);
```

6.4 Extending the client API

The `@gala-chain/client` package provides a default API definition that is used to make the client type-safe. By default `ChainClient` defines `evaluateTransaction` and `submitTransaction` methods that are used to interact with the chaincode. However, you can provide your own API definition, if you want to extend the client API or use a different API. The API definition is a function that accepts a `ChainClient` instance and returns an object with methods that will be added to the client.

```
function customAPI(client: ChainClient) {
  return {
    async GetProfile(privateKey: string) {
      const dto = new GetMyProfileDto().signed(privateKey, false);
      const response = await client.evaluateTransaction("GetMyProfile", dto, UserProfile);
      if (GalaChainResponse.isError(response)) {
        throw new Error(`Cannot get profile: ${response.Message} (${response.ErrorKey})`);
      } else {
        return response.Data as UserProfile;
      }
    }
  };
}
```

Now, when you enhance the client with your custom API, you can use not only default methods but also the ones that you defined:

```
const client: ChainClient = ...;
client.evaluateTransaction(...);  // available
client.submitTransaction(...);    // available
client.GetProfile(...);           // compilation error

const extendedClient = client.extend(customAPI);
client.evaluateTransaction(...);  // available
client.submitTransaction(...);    // available
client.GetProfile(...);           // available
```


© 2024 Gala Games

7. Testing your chaincode

The GalaChain SDK includes a comprehensive set of tools in the `@gala-chain/test` package to facilitate the testing of your chaincode. This package supports both unit testing for individual contracts and integration/end-to-end testing for running networks.

7.1 Unit testing

The `@gala-chain/test` package offers utilities designed for straightforward unit testing of your chaincode. The recommended library for tests is Jest.

7.1.1 Writing unit tests

Consider a contract `AppleContract` with the following methods:

```
export class AppleContract extends GalaContract {
  public async PlantTree(ctx: GalaChainContext, dto: AppleTreeDto): Promise<void> { ... }
  public async PickApple(ctx: GalaChainContext, dto: PickAppleDto): Promise<void> { ... }
}
```

Let's create tests for the following scenarios: 1. `AppleContract` should allow to plant a tree. 2. `AppleContract` should fail to plant a tree if tree already exists. 3. `AppleContract` should allow to pick an apple.

Note: `AppleContract` with the referenced implementation and all relevant tests are available in chaincode template. You can follow the instructions in Getting started to create a new chaincode project with `AppleContract` included.

Test 1. `AppleContract` should allow to plant a tree

This test ensures that the `AppleContract` allows users to successfully plant a new apple tree. It validates the contract's behavior during the tree planting process.

```
import { fixture, transactionSuccess, writesMap } from "@gala-chain/test";
import { AppleTree, AppleTreeDto, Variety } from "../apples";
import { AppleContract } from "./AppleContract";

it("should allow to plant a tree", async () => {
  // Given
  const {contract, ctx, writes} = fixture(AppleContract);
  const dto = new AppleTreeDto(Variety.GALA, 1);
  const expectedTree = new AppleTree(ctx.callingUser, dto.variety, dto.index, ctx.txUnixTime);

  // When
  const response = await contract.PlantTree(ctx, dto);

  // Then
  expect(response).toEqual(transactionSuccess());
  expect(writes).toEqual(writesMap(expectedTree));
});
```

In this test, we set up the initial environment using the `fixture` utility from `@gala-chain/test`. The `fixture` contains: - `contract` -- instance of the `AppleContract` class, - `ctx` -- test chaincode context, - `writes` -- object capturing changes to the blockchain state.

Also, we define the `AppleTreeDto` instance containing details about the apple tree to be planted, and the `expectedTree` instance containing the expected object to be written to the blockchain state.

The primary action involves invoking the `PlantTree` method on the `contract` instance.

Then, we assert that the response from planting the tree aligns with the expected success result with `transactionSuccess` matcher from `@gala-chain/test`. Furthermore, we verify that the changes to the blockchain state (`writes`) match the expected modifications. Since `writes` is a map of key-value pairs, we use the `writesMap` utility from `@gala-chain/test` to get a key-value representation of the `expectedTree` instance.

© 2024 Gala Games

Test 2. `AppleContract` should fail to plant a tree if tree already exists

In this test case, we aim to verify the behavior of the AppleContract when attempting to plant a new apple tree that already exists. In our case a tree is considered to exist if it has the same `variety` and `index` as is planted by the same user.

```
import { fixture, transactionErrorMessageContains } from "@gala-chain/test";
import { ChainUser } from "@gala-chain/client";
import { AppleTree, AppleTreeDto, Variety } from "../apples";
import { AppleContract } from "./AppleContract";

it("should fail to plant a tree if tree already exists", async () => {
  // Given
  const user = ChainUser.withRandomKeys();

  const {contract, ctx, writes} = fixture(AppleContract)
    .callingUser(user)
    .savedState(new AppleTree(user.identityKey, Variety.GOLDEN_DELICIOUS, 1, 0));

  // When
  const response = await contract.PlantTree(ctx, new AppleTreeDto(Variety.GOLDEN_DELICIOUS, 1));

  // Then
  expect(response).toEqual(transactionErrorMessageContains("Tree already exists"));
  expect(writes).toEqual({});
});
```

In this test case, we also use the `fixture` utility from `@gala-chain/test` to set up the initial environment. However, we use the `callingUser` method to specify the user who will invoke the contract method, and we use the `savedState` method to specify the initial state of the blockchain. Also, the saved `AppleTree` instance is marked to be planted by the user who invokes the contract method (`user.identityKey` is the same as `ctx.callingUser` in this setup).

This way calling `PlantTree` method with the same `variety` and `index` will result in an error.

During validation, we assert that the response from planting the tree contains the expected error message, and no changes to the blockchain state (`writes`) are made. To assert the error we use the `transactionErrorMessageContains` matcher from `@gala-chain/test`. Other useful matchers include `transactionError` (for providing an exact error object) and `transactionErrorKey` (for providing the error key).

Test 3. `AppleContract` should allow to pick an apple

In this test case, we aim to verify the behavior of the `AppleContract` when attempting to pick an apple from an existing apple tree.

```
import { fixture, transactionSuccess, writesMap } from "@gala-chain/test";
import { plainToInstance } from "class-transformer";
import { AppleTree, PickAppleDto, Variety } from "../apples";
import { AppleContract } from "./AppleContract";

it("should allow to pick apples", async () => {
  // Given
  const twoYearsAgo = new Date(new Date().getTime() - 1000 * 60 * 60 * 24 * 365 * 2).getTime();
  const existingTree = new AppleTree("client|some-user", Variety.GALA, 1, twoYearsAgo);
  const {contract, ctx, writes} = fixture(AppleContract).savedState(existingTree);

  const dto = new PickAppleDto(existingTree.plantedBy, existingTree.variety, existingTree.index);

  // When
  const response = await contract.PickApple(ctx, dto);

  // Then
  expect(response).toEqual(transactionSuccess());
  expect(writes).toEqual(writesMap(plainToInstance(AppleTree, {
    ...existingTree,
    applesPicked: existingTree.applesPicked.plus(1)
  })));
});
```

In our case a tree has apples if a given time passes. That's why we start from a tree that was planted two years ago.

During validation, we assert that the response from picking an apple is successful, and the change to the blockchain state (`writes`) is overriding current apple tree with the updated picked apples count.

© 2024 Gala Games

7.1.2 Using `fixture` for regular functions

`fixture` can be used for regular functions as well, without the need to call contract methods directly. However, the `ctx` parameter is tied to the contract, and you must provide any contract class, such as `AppleContract` or any class that extends `GalaContract` from the `@gala-chain/chaincode` package.

```
import { ChainUser } from "@gala-chain/client";
import { fixture, writesMap } from "@gala-chain/test";
import { GalaContract } from "@gala-chain/chaincode";
import { AppleTreeDto, AppleTreesDto } from "./dtos";
import { Variety } from "./types";
import { AppleTree } from "./AppleTree";
import { plantTrees } from "./plantTrees";

class TestContract extends GalaContract {
  constructor() {
    super("TestContract", "0.0.1");
  }
}

it("should allow to plant trees", async () => {
  // Given
  const user = ChainUser.withRandomKeys();

  const {ctx, writes} = fixture(TestContract).callingUser(user);

  const dto = new AppleTreesDto([
    new AppleTreeDto(Variety.GALA, 1),
    new AppleTreeDto(Variety.MCINTOSH, 2),
  ]);

  const expectedTrees = dto.trees.map(t => new AppleTree(user.identityKey, t.variety, t.index, ctx.txUnixTime));

  // When
  const response = await plantTrees(ctx, dto);

  // Then
  expect(response).toEqual(expectedTrees);

  await ctx.stub.flushWrites();
  expect(writes).toEqual(writesMap(...expectedTrees));
});
```

Using `fixture` for regular functions is useful when you want to test the behavior of the function without the need to call the contract method. However, if you want to verify writes, you need to explicitly call `contract.afterTransaction` or `ctx.stub.flushWrites` method. This is required, because all writes actually are added to internal cache, and are executed after the contract method is successfully executed.

7.1.3 Additional notes

Signatures

In most transactions, DTOs require a secp256k1 signature to verify the identity of the user invoking the contract method. When using `fixture`, there's no need to provide a signature as it's handled automatically.

`beforeTransaction` and `afterTransaction`

In the context of testing contract methods with `fixture`, you don't need to manually call `contract.beforeTransaction` and `contract.afterTransaction` methods; they are invoked automatically.

7.2 Integration testing

The `@gala-chain/test` package, combined with the `@gala-chain/client` package, provides utilities for integration testing your chaincode. The primary objective of integration or end-to-end tests is to call transactions on the actual Hyperledger Fabric network and verify the results.

The recommended library for tests is Jest.

© 2024 Gala Games

7.2.1 Writing integration tests

Assume you have a contract `AppleContract` with the following methods:

```
export class AppleContract extends GalaContract {
  public async PlantTrees(ctx: GalaChainContext, dto: AppleTreesDto): Promise<void> { ... }
  public async FetchTrees(ctx: GalaChainContext, dto: FetchTreesDto): Promise<PagedTreesDto> { ... }
  public async PickApple(ctx: GalaChainContext, dto: PickAppleDto): Promise<void> { ... }
}
```

Let's write tests for the following scenarios: 1. Plant a bunch of trees 2. Fetch GALA trees planted by a user 3. Fail to pick a GOLDEN_DELICIOUS apple because tree is too young

Note: `AppleContract` with the referenced implementation and all relevant tests (file: `e2e/apples.spec.ts`) are available in chaincode template.

Setup

Before writing integration tests, ensure you have a running GalaChain network. You can use the `npm run network:start` command provided by the chaincode template to start a local network in dev mode with hot-reload enabled.

Integration tests are executed against the running network, which is not recreated after each test. To make tests independent, you may need to randomize test data or clean up the data on the chain after tests.

In our case for apples, we use random users defined at the test suite level to create different users for each run, ensuring test suite independence. However, each test in the suite uses the same user and is not independent. Thus, we use scenario-like testing in the apples test suite, and each test is dependent on the previous one.

Also, since we are using the running network, we need a client to interact with the network. It needs to be connected to the network, anf it needs to be disconnected after the tests are finished.

Here is an example of the test setup:

```
import { AdminChainClients, TestClients, transactionErrorKey, transactionSuccess, } from "@gala-chain/test";
import { GalaChainResponse } from "@gala-chain/api";
import { ChainClient, ChainUser } from "@gala-chain/client";
import { AppleTreeDto, AppleTreesDto, FetchTreesDto, PagedTreesDto, PickAppleDto, Variety } from "../src/apples";

describe("Apple trees", () => {
  const appleContractConfig = {apples: {name: "AppleContract", api: appleContractAPI}};
  let client: AdminChainClients<typeof appleContractConfig>;
  let user: ChainUser;

  beforeAll(async () => {
    client = await TestClients.createForAdmin(appleContractConfig);
    user = await client.createRegisteredUser();
  });

  afterAll(async () => {
    await client.disconnect();
  });
  ...
```

Optional setup -- custom API

By default `client` is not aware of your chaincode and types, providing only generic methods for submitting or evaluating transactions:

```
submitTransaction(method: string): Promise<GalaChainResponse<unknown>>;
submitTransaction(method: string, dto: ChainCallDTO): Promise<GalaChainResponse<unknown>>;
submitTransaction<T>(method: string, resp: ClassType<Inferred<T>>): Promise<GalaChainResponse<T>>;
submitTransaction<T>(method: string, dto: ChainCallDTO, resp: ClassType<Inferred<T>>): Promise<GalaChainResponse<T>>;

evaluateTransaction(method: string): Promise<GalaChainResponse<unknown>>;
evaluateTransaction(method: string, dto: ChainCallDTO): Promise<GalaChainResponse<unknown>>;
evaluateTransaction<T>(method: string, resp: ClassType<Inferred<T>>): Promise<GalaChainResponse<T>>;
evaluateTransaction<T>(method: string, dto: ChainCallDTO, resp: ClassType<Inferred<T>>): Promise<GalaChainResponse<T>>;
```

They are generic, and you need to provide a method name, and optionally a DTO and response type to deserialize the response to a proper type. But you can define a custom API, that will be aware of your chaincode and types.

© 2024 Gala Games

If you choose not to use a custom API, you can create a test client as follows:

```
const appleContractConfig = {apples: "AppleContract"};
let client: AdminChainClients<typeof appleContractConfig>;
...
beforeAll(async () => {
  client = await TestClients.createForAdmin(appleContractConfig);
  ...
```

This way you will be able to use only the generic methods to call chaincodes: `client.apples.evaluateTransaction(...)` or `client.apples.submitTransaction(...)`.

However, defining a custom API offers type-safe calls, as demonstrated in the apples test suite. You can define it as follows:

```
interface AppleContractAPI {
  PlantTrees(dto: AppleTreesDto): Promise<GalaChainResponse<void>>;
  FetchTrees(dto: FetchTreesDto): Promise<GalaChainResponse<PagedTreesDto>>;
}

function appleContractAPI(client: ChainClient): AppleContractAPI {
  return {
    PlantTrees(dto: AppleTreesDto) {
      return client.submitTransaction("PlantTrees", dto) as Promise<GalaChainResponse<void>>;
    },
    FetchTrees(dto: FetchTreesDto) {
      return client.evaluateTransaction("FetchTrees", dto, PagedTreesDto);
    }
  };
}
```

And provide it for client creation:

```
const appleContractConfig = {apples: {name: "AppleContract", api: appleContractAPI}};
let client: AdminChainClients<typeof appleContractConfig>;
...
beforeAll(async () => {
  client = await TestClients.createForAdmin(appleContractConfig);
  ...
```

And it allows you to use type-safe calls, defined in the API, like `client.apples.PlantTrees(...)` or `client.apples.FetchTrees(...)`.

Test 1. Plant a bunch of trees

```
test("Plant a bunch of trees", async () => {
  // Given
  const dto = new AppleTreesDto([
    new AppleTreeDto(Variety.GALA, 1),
    new AppleTreeDto(Variety.GOLDEN_DELICIOUS, 2),
    new AppleTreeDto(Variety.GALA, 3),
  ])
    .signed(user.privateKey, false);

  // When
  const response = await client.apples.PlantTrees(dto);

  // Then
  expect(response).toEqual(transactionSuccess());
});
```

In this test case, we create a DTO with three trees to plant. We sign the DTO with the user's private key to prove the identity of the user. This is required, in contrast to unit tests.

Then we call `PlantTrees` method, defined in our custom API, and we assert that the response is successful.

As a result the test writes three trees to the blockchain, planted by the user. We will use them in the next test.

Test 2. Fetch GALA trees planted by a user

```
test("Fetch GALA trees planted by a user", async () => {
  // Given
  const dto = new FetchTreesDto(user.identityKey, Variety.GALA)
    .signed(user.privateKey, false);

  // When
  const response = await client.apples.FetchTrees(dto);

  // Then
  expect(response).toEqual(transactionSuccess({
```


© 2024 Gala Games

```
      trees: [
        expect.objectContaining({plantedBy: user.identityKey, variety: Variety.GALA, index: 1}),
        expect.objectContaining({plantedBy: user.identityKey, variety: Variety.GALA, index: 3})
      ],
      bookmark: ""
    }))
  })
```

In the previous test, we planted three trees, two of them are GALA. In this test, we fetch all GALA trees planted by the user.

The response contains two trees, planted by the user, and the bookmark for fetching next page (though in this case, it's empty).

Test 3. Fail to pick a GOLDEN_DELICIOUS apple because tree is too young

```
  test("Fail to pick a GOLDEN_DELICIOUS apple because tree is too young", async () => {
    // Given
    const dto = new PickAppleDto(user.identityKey, Variety.GOLDEN_DELICIOUS, 2)
      .signed(user.privateKey, false);

    // When
    const response = await client.apples.PickApple(dto);

    // Then
    expect(response).toEqual(transactionErrorKey("NO_APPLES_LEFT"));
  });
```

In this test case, we try to pick an apple from the tree that was planted in the first test. However, the tree is too young, so we expect an error.

These examples provide a comprehensive guide for unit and integration testing of GalaChain smart contracts using the `@gala-chain/test` package. Adjust and expand the provided code snippets based on your specific contract implementations and testing requirements.

© 2024 Gala Games

8. Chaincode deployment

Chaincode is published as a Docker image to GalaChain repository. Once the image is published, it can be deployed to GalaChain testnet or sandbox. In order to publish and deploy chaincode, you need to contact GalaChain support and add provide your secp256k1 public key.

8.1 The process

1. **Build and publish chaincode Docker image.**

   Sample for DockerHub (It uses the ttl.sh to make it available for 1 day):

   ```
   docker build --push -t ttl.sh/<IMAGE_NAME>:1d .
   ```

   Provide us the image name (everything before the `:` character of full docker tag). In the sample above it is the content of `ttl.sh/<IMAGE_NAME>`.

2. **Provide to GalaChain support chaincode information and public keys.**

   The keys are automatically generated when you initialize the project using `galachain init`, so you can find these keys in `keys/gc-admin-key.pub` and `keys/gc-dev-key.pub`.

   Note: The developer key should be shared with all team members who want to deploy the chaincode.

   If you can't find the keys, you can generate them using the following commands:

   ```
   galachain keygen gc-admin-key
   galachain keygen gc-dev-key
   ```

3. **Deploy the chaincode to testnet or sandbox:**

   ```
   galachain deploy <docker-image-tag> <path-to>/gc-dev-key
   ```

   Note: you need to provide docker image name and also the version part. If you used the `ttl.sh` example, the `docker-image-tag` should be something like `ttl.sh/<IMAGE_NAME>:1d`.

4. **Fetch information about the chaincode and deployments:**

   ```
   galachain info <path-to>/gc-dev-key
   ```

   Once the status is `CC_DEPLOYED` you can visit the Swagger webpage: https://gateway.stage.galachain.com/docs/. You can find your chaincode (`gc-<eth-addr>`). If the version is still unknown (and you see `v?.?.?`), it means you may need to wait a couple of minutes till the chaincode is ready.

   Once it is ready, you can use the webpage to call chaincodes. It's good to start `PublicKeyContract/GetPublicKey` with empty object as request body. It should return the admin public key you provided before.

5. **Call the deployed chaincode**

   You can use any REST API client (like `axios` to call your chaincodes). Remember in most cases you will need to sign the DTO with either the `gc-admin-key` or any key of registered user.

   We highly recommend to use the `@gala-chain/api` library for handling DTOs and signing. For instance, you can register a user by calling `/api/.../...-PublicKeyContract/RegisterEthUser` and providing the following `RegisterEthUser` as payload:

   ```
   const dto = new RegisterEthUser();
   dto.publicKey = <newUserPublicKey>;
   dto.sign(<gc-admin-key>);
   const payloadString = dto.serialize();
   ```

   In the current version of the library, local environment exposes slightly different endpoints than the production environment. `gcclient` and `@gala-chain/client` packages are compatible with the local environment only. For calling the production environment, you should consult the Swagger documentation at https://gateway.stage.galachain.com/docs/, and use generic REST API client.

© 2024 Gala Games

8.2 Reference

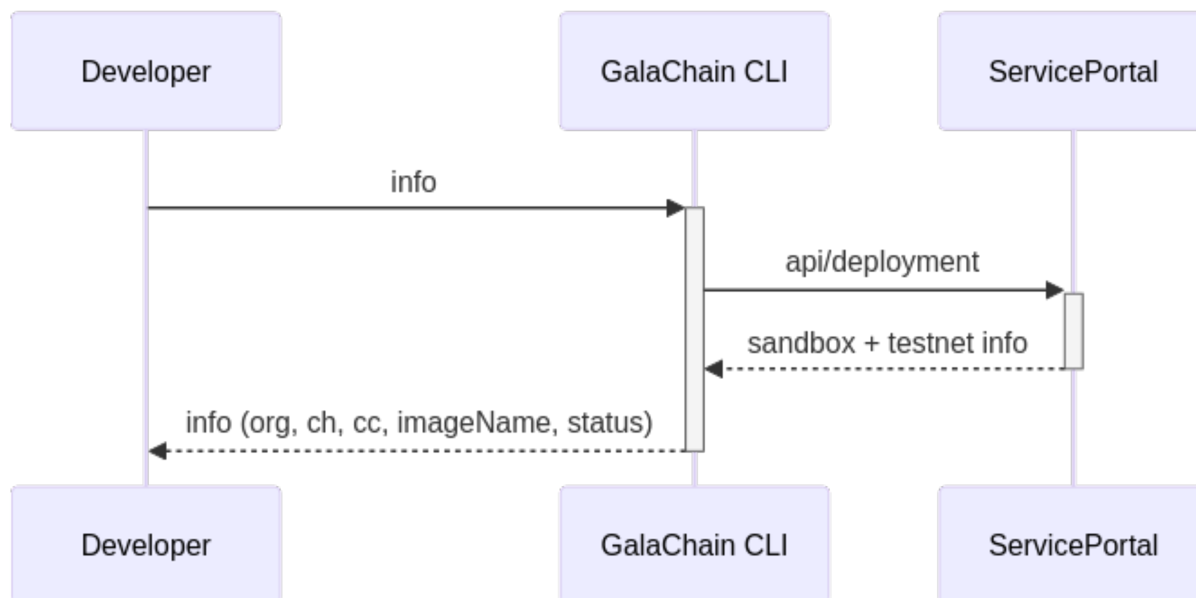
GalaChain CLI calls some local command and accesses ServicePortal REST API to accomplish certain tasks. Each REST request body to ServicePortal (1) is signed using our default GalaChain signature type (secp256k1, non-DER), and (2) contains unique request id. Both signing and creating the ID is managed by GalaChain CLI.

8.2.1 Fetching information about chaincode and deployments

```
galachain info
```

This command will display:

- Org, channel, chaincode names.
- Status of the chaincode deployment.



8.2.2 Deploying the chaincode

Deploying to GalaChain testnet:

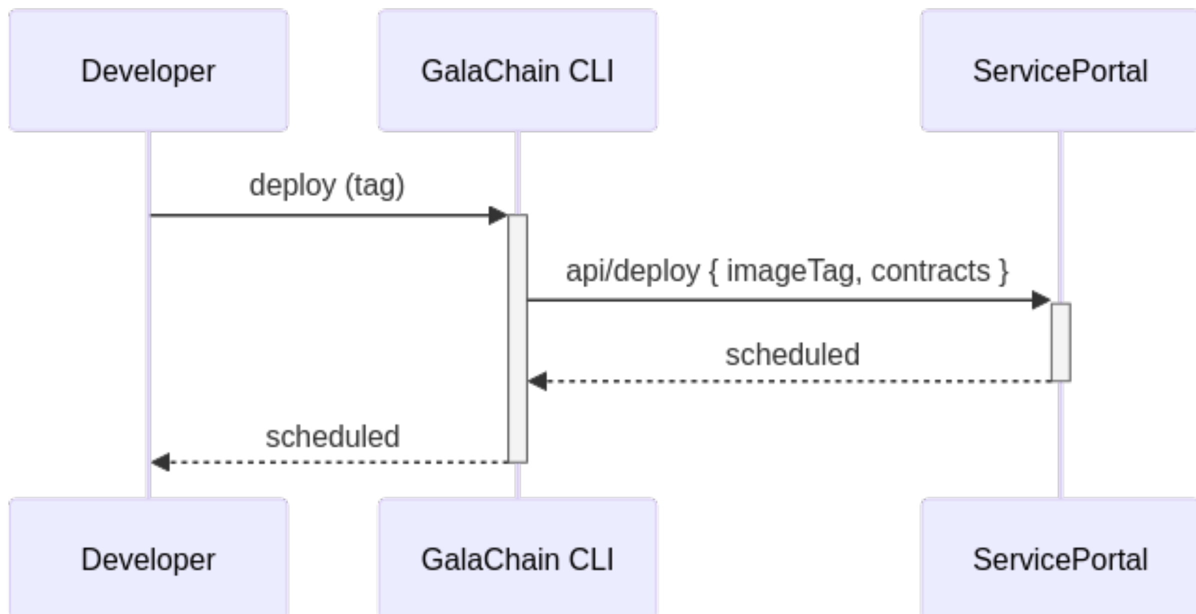
```
galachain test-deploy <docker-image-tag> <path-to>/gc-dev-key
```

Deploying to GalaChain sandbox:

```
galachain deploy <docker-image-tag> <path-to>/gc-dev-key
```

This command schedules deployment of published chaincode Docker image to GalaChain testnet or sandbox. In order to get the information about the current status of deployments, you need to use `galachain info` command.

© 2024 Gala Games



© 2024 Gala Games

9. Authorization

TBD https://github.com/GalaChain/sdk/issues/120

© 2024 Gala Games

10. From zero to deployment with GalaChain SDK

GalaChain SDK allows you to develop and deploy GalaChain chaincodes (contracts) in TypeScript. This tutorial will guide you through the process of creating a new GalaChain chaincode, connecting it with GalaChain network, deploying, and calling it.

10.1 1. Install the GalaChain CLI

GalaChain SDK provides a CLI to manage your chaincode. You can install it with:

```
npm i -g @gala-chain/cli
```

To verify it works you can use:

```
galachain --help
```

GalaChain CLI requires Node.js v18+. For running a local test network you also need Docker with Docker Compose, and `jq`. If you work on Windows, you either need to have WSL, or you can use our Dev Containers.

10.2 2. Initialize the project from template

GalaChain CLI can create a fully functional sample GalaChain chaincode with some features, tests, local env. setup and many others. Just type:

```
galachain init my-gc-chaincode
```

This will create a new directory `my-gc-chaincode` with the chaincode template. Change the directory to the newly created one and see what's inside:

```
cd my-gc-chaincode
ls
```

Among others, you will find the following directories: - `src` - the source code of your chaincode, - `e2e` - end-to-end tests for your chaincode, - `keys` - keys that are required for calling our managed infrastructure.

Additionally, init command creates private keys for the chaincode admin and developer in your home directory at `~/.gc-keys/<chaincode-name>`, where `<chaincode-name>` consists of `gc-` prefix and eth address calculated from chaincode admin public key.

10.3 3. Update the contract (optional)

The chaincode template comes with some sample contract code. It exposes three contract classes: - `PublicKeyContract` - makes the chaincode conform to the GalaChain authorization model (the only one you should not modify), - `GalaChainTokenContract` - contains features for managing tokens (feel free modify or remove it if you want), - `AppleContract` - a sample showcase contract, probably the easier to start with.

Feel free to modify the contract code to suit your needs.

If you want to verify that your contract works, you can start the local test network with:

```
npm run network:start
```

And then run the end-to-end tests with:

```
npm run test:e2e
```

See the Chaincode Development and Chaincode Testing reference for more details.

© 2024 Gala Games

10.4 4. Prepare and publish chaincode docker image

Before you can deploy your chaincode, you need to build a Docker image with it and publish it to a registry of your choice (e.g. Docker Hub).

Chaincode template comes with a `Dockerfile` that you should use to build a chaincode image. Also, it is recommended to use `buildx` to ensure that the image architecture is `linux/amd64` (required by GalaChain network).

Assuming you have Docker tag name in `$TAG` environment variable, you can build and publish the image with the following commands:

```
docker buildx build --platform linux/amd64 -t $TAG .
docker push $TAG
```

Docker image should be publicly accessible, as GalaChain network will download it during the deployment.

10.5 5. Connect your chaincode with GalaChain network

In order to deploy a chaincode GalaChain support needs to review and approve it. To do so, you need to provide us the following information: - Docker image tag (without the version, or `:latest` part), - Chaincode admin public key (from `keys/gc-admin-key.pub` file), - Developer public key (from `keys/gc-dev-key.pub` file).

After the approval, call the following command to verify you registration:

```
galachain info
```

You should get a JSON response with your chaincode information. Note the `chaincode` field, which is your chaincode name, and the `image` field, which is the Docker image tag you provided.

10.6 6. Deploy the chaincode

To deploy the chaincode, you need to call the following command:

```
galachain deploy <image-tag>
```

Replace `<image-tag>` with the Docker image tag you provided, plus the version (e.g. `my-registry/my-gc-chaincode:1.0.0`).

The command will deploy the chaincode to the GalaChain network. The deployment process may take a while, as the network needs to download the chaincode image and start it.

You can verify the deployment status with `galachain info` command.

See the Chaincode Deployment reference for more details.

10.7 7. Call REST API

GalaChain Gateway provides a REST API to interact with the chaincode. The simplest way to call it is to use `curl` (for convenience, you can use `galachain info` and `jq` to build chaincode url):

```
info=$(galachain info)
chaincode=$(jq -r '.chaincode' <<< $info)
channel=$(jq -r '.channel' <<< $info)
url=https://gateway.stage.galachain.com/api/$channel/$chaincode-AppleContract/GetChaincodeVersion

curl -X POST -d '{}' $url
```

You can also visit the GalaChain Gateway page at https://gateway.stage.galachain.com/docs to see the Swagger UI and explore the API.

© 2024 Gala Games

Additionally, which is most convenient, you can use the GalaChain client library to interact with the chaincode.

```
const params: RestApiClientConfig = {
  apiUrl: "https://gateway.stage.galachain.com/api",
  configPath: path.resolve(__dirname, "api-config.json")
};

const client: ChainClient = gcclient
  .forApiConfig(params)
  .forContract(contract);
```

Also remember to sign the payload with your private key before sending it to the network. The initial user on chain is the admin, so you can use the relevant `gc-admin-key` from the `~/.gc-keys/<chaincode-name>` directory. See the Chaincode Client and the Authorization reference for more details.

© 2024 Gala Games

11. Using Windows with WSL

11.1 How to use GalaChain with Windows Subsystem for Linux (WSL)

11.1.1 1. Install Docker Desktop

Download and install Docker Desktop from the official website: https://www.docker.com/products/docker-desktop

If you already have Docker Desktop installed, make sure to update it to the latest version.

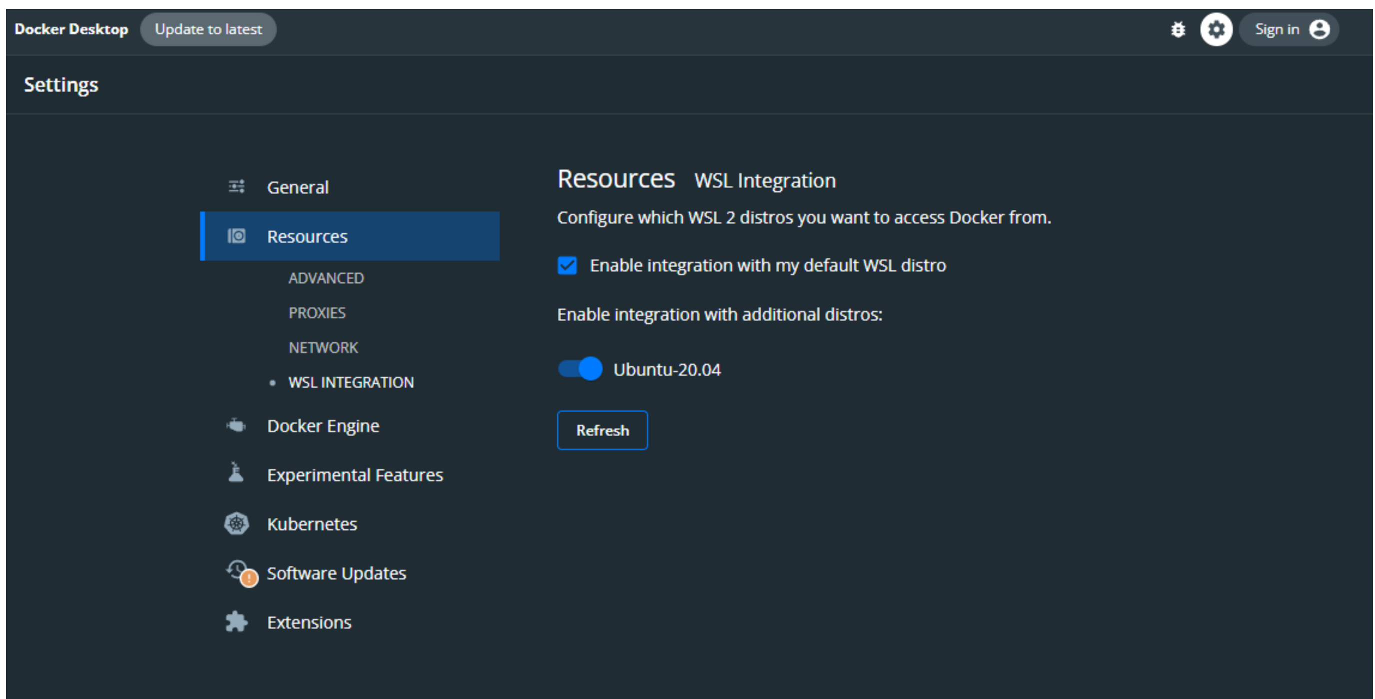
11.1.2 2. Install WSL 2 and a Ubuntu distribution

Follow the official guide to install WSL 2: https://docs.microsoft.com/en-us/windows/wsl/install

Here is a short video from Microsoft about how to install WSL 2 and how to prepare it to build Node.js applications:

11.1.3 3. Enable WSL integration on Docker Desktop

Open Docker Desktop and go to `Settings` > `Resources` > `WSL Integration` and enable the integration with your WSL distribution.



11.1.4 4. Install dependencies and start network

1. Use the WSL extension on VSCode to connect to your WSL distribution.
2. Install Node Version Manager (NVM) on your WSL distribution: https://learn.microsoft.com/en-us/windows/dev-environment/javascript/nodejs-on-wsl#install-nvm-nodejs-and-npm
3. Install `yq` and `jq` on your WSL distribution:

```
sudo snap install yq jq
```

4. At this point your WSL environment should be ready to use GalaChain. Follow the instructions on the Getting Started guide to install the CLI and initialize your project.

© 2024 Gala Games