GIF Manual Test Documentation

Release latest

Sep 03, 2019

Generic Insurance Framework User Manual

This manual explains the key ideas, terms, and principles of the Generic Insurance Framework (GIF) as a tool that enables to utilize both smart contracts and microservices to create specific insurance products. You will learn the best ways to interact with this framework, as well as employ the available functionality to the fullest.

The manual focuses on the needs of product builders and helps them to implement the existing solutions on top of the Generic Insurance Framework.

CHAPTER 1

User Manual for the Generic Insurance Framework

1.1 Terminology

Below, you will find a glossary of the technical terms used in this document.

Actor. Any participant of the DIP that uses the ecosystem to perform an activity on it (e.g., a product, a product owner, an oracle owner, the instance operator, etc.).

Application. Data applied by a customer requesting an insurance policy. An application is the predecessor of a policy. Not to be confused with software application; in our context we avoid the term "application" in this sense.

Claim. Data related to an insurance claim, which requires approval.

Decentralized Insurance Platform (DIP). An ecosystem supported by the DIP Foundation that unites product builders, risk pool keepers, resellers, oracle providers, claim adjusters, relayers, and underwriters.

Decentralized Insurance Protocol (DIP Protocol). A set of standards, rules, templates and definitions which define the interaction of participants in the ecosystem.

DIP Token. The Decentralized Insurance Protocol Token (DIP) is an ERC-20 token on the Ethereum Mainnet (token contract on etherscan). DIP tokens were issued by the DI Foundation during the DIP Token Generating Event in 2018, and will be used as a platform token in the DIP platform to incentivize different actors. The DIP token mechanics paper has more details on the token model."

Generic Insurance Framework (GIF). A combined codebase, which includes smart contracts and utility services (core smart contracts and microservices) provided by the DIP Foundation and partners. The codebase can be extended by product-specific smart contracts and microservices created by product builders. Using this framework, product builders can develop full-featured DApps.

GIF instance. A deployed set of core smart contracts, operated by an instance operator, in most cases together with an appropriate set of utility services. A GIF instance is essentially an "Insurance as a Service (IaaS)".

Instance operator. An Ethereum account which operates an instance of the GIF. An instance operator can be a decentralized organization (DAO) or a single account owned by some legal entity.

Metadata. A shared object between all the objects of a particular policy flow.

Oracle. A service used to provide information to smart contracts from external resources, confirm certain events, and deliver particular data to a product.

Oracle owner. An Ethereum account registered on the DIP with a set of permissions for creating oracles and oracle types and performing operations on them.

Oracle type. A type of request to an oracle containing attributes that describe a request and respond to it. Oracles join an oracle type.

Payout. Data related to the expected and actual payout for a claim.

Policy. Technical representation of the legal agreement between a policy buyer and a carrier.

Policy flow. A core smart contract that represents a workflow of insurance policy life cycle, involving such steps as application underwriting, risk assessment, claim review, and payouts.

Policy token. A ERC1521 token (extension of a ERC 721 NFT Token), which represents a policy as a set of particular fields.

Product. A registered smart contract with permissions to create and manage policy flows.

Product owner. An Ethereum account registered on the GIF with a set of permissions allowing to create and manage product contracts and oracle types.

CHAPTER 2

Generic Insurance Framework

The Generic Insurance Framework represents a combined codebase for the Decentralized Insurance Platform, a basic implementation that enables users to develop blockchain-based applications.

The basic idea behind the GIF is to abstract the generic parts shared across multiple different products and leave only product-specific parts, such as risk model, pricing, and payout configurations, to be adjusted. The goal is to enable quick and easy deployment of working products.

In its core, the GIF accumulates a number of componets: - core smart contracts - core microservices - product-specific smart contracts - product-specific microservices

Essentially, the GIF has two major layers — a smart contracts one and a utility one — with DIP Foundation and partners being able to contribute to both.

The **smart contracts layer** is designed in the way that any blockchain product built on top of the GIF can be easily implemented into any network supporting the Ethereum Virtual Machine. Any product owner is able to create a full-featured decentralized app by adding a couple of simple domain-specific contracts to a number of generic ones that the framework provides.

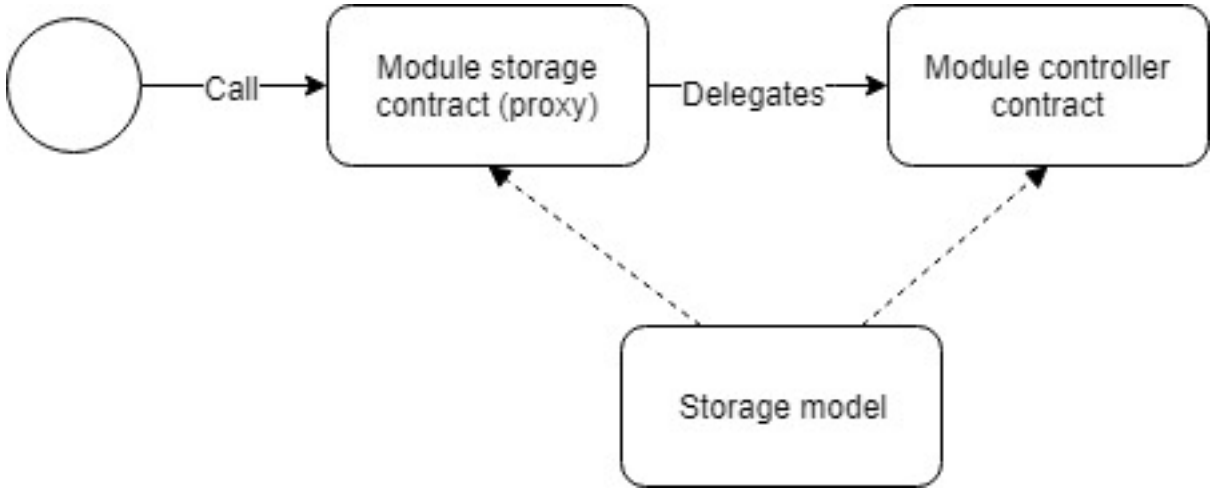
The core contracts are deployed on-chain and operate by an instance operator as a shared service for many different products. The instance operator can be a decentralized organization (DAO) or a more traditional legal entity. A product, working on top of the GIF, is a smart contract (or set of smart contracts) connected to the framework's core contracts through a unique entry point.

The DIP Protocol declares the underlying principles and requirements based on which the architecture of smart contracts is developed:

- Generic Insurance Framework provides a unified interface, which connects a product to data and decision providers (oracles).
- A product contract utilizes a simple and clear interface for integration with the GIF.
- Once the agreement is signed and a policy token is issued, parties cannot change the expected policy flow behavior. A policy life cycle should operate on the contracts, which this policy was issued by.
- Core contracts can be upgraded. This is needed to make bug fixes and add new features.

The smallest building blocks are called "modules." A "module" is a pair of a "storage" contract and a "controller." They share the same storage model and interface objects. The "storage" contract is a proxy, which delegates calls from

a “storage” to a “controller,” which implements basic logic (i.e., a method to change a state). This mechanism ensures that the module can be upgraded.



A “service” contract contains business logic details and defines rules (i.e., “Underwritten” is the next state after “Applied”). The “service” contract manages modules by calling controllers from storages. It is also an entry point for actors (products, oracles, product owners, etc.).

“Controllers” serve as entry points for “services.” It is important to differentiate “services” behavior from that of “controllers.”

The **utility layer** can contain any number of off-chain utility services supplementing the on-chain functionality. For example, statistical monitoring of events triggered by contracts, making e-mail or instant messenger notifications, accepting fiat payments for policies, as well as making fiat payouts. As a result, any product app can be fully functional on chain even without any support from the utility layer, as well as can provide a full spectrum of the required features.

The key feature to have in a framework is the ability to upgrade and replace individual elements of the system. For this purpose, the Generic Insurance Framework employs a microservices-based architecture approach for its utility layer. The GIF organizes off-chain operations as a collection of loosely coupled services, each implementing a single independent function — a state known as “decomposition by business capabilities.”

CHAPTER 3

Core Smart Contracts

Core smart contracts represent a number of key contracts and modules. The product service, policy flow, and modules are described below. Core smart contracts are deployed and operated by an instance operator - a DAO or some other (legal) entity. The instance operator publishes the entry points to its instance of the GIF (e.g. the address of the Product Service) and registers actors in the instance.

3.1 Product Service

The **product service** is an entry point for a product contract. During smart contract deployment, the address of the product service should be passed as one of the constructor arguments.

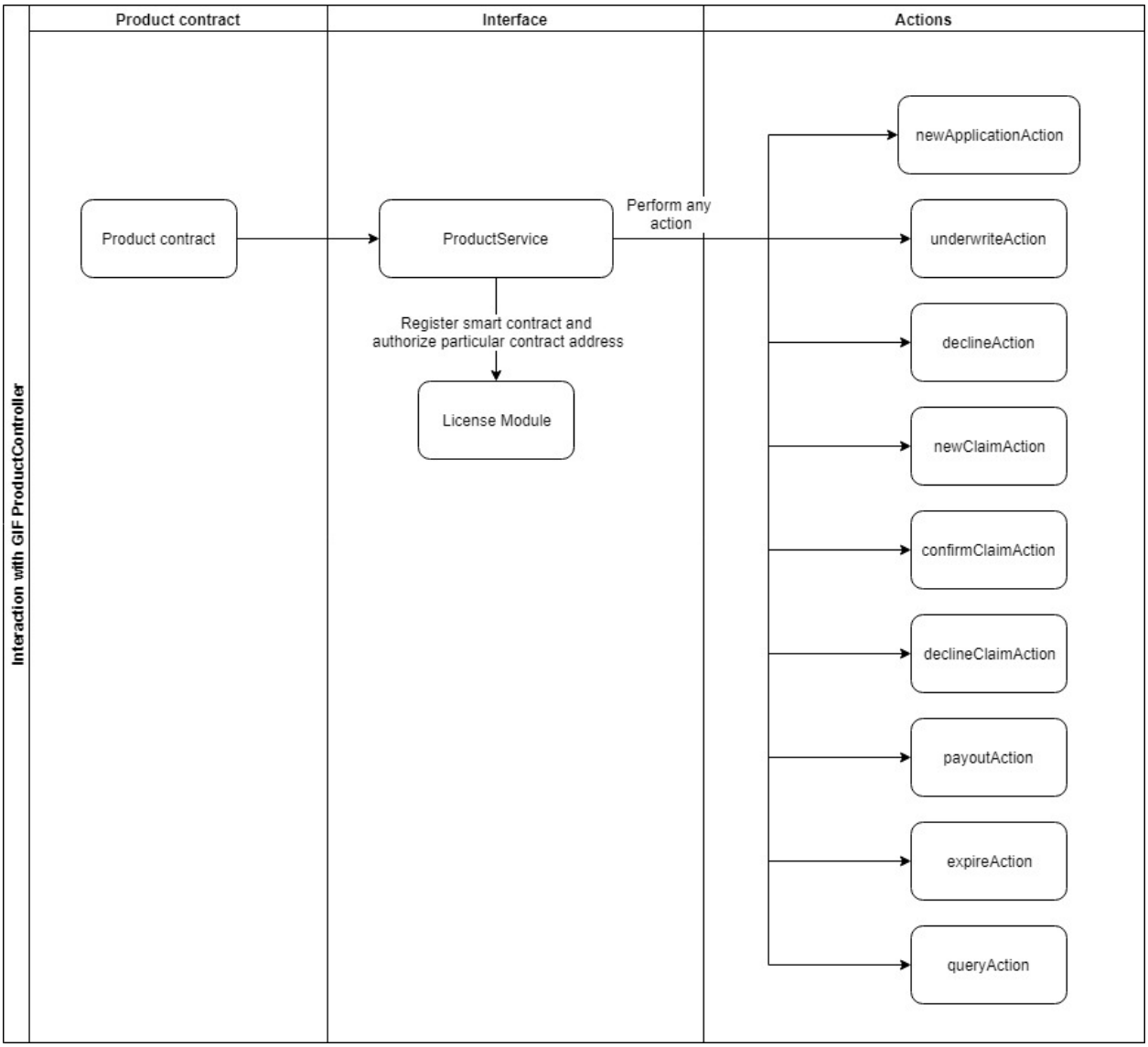
All product service methods are used by a product contract.

Below, you will find a list of the methods invoked by the product service:

- **register** is used to register new product contracts by providing a product name and specifying a policy flow. On approval, a product contract obtains access to call entry methods.
- **newApplication** is employed to store new application data, which contains such fields as premium amount, currency, payout options, risk definition, etc. A policy buyer signs a policy agreement using this method.
- **underwrite** is used to sign a policy agreement by an insurance company.
- **decline** declines an application.
- **newClaim** declares a new claim.
- **confirmClaim** confirms a claim. A new payout object is created after this.
- **declineClaim** declines a claim.
- **payout** declares payout that was handled off-chain or on-chain based on the policy currency.
- **expire** sets a policy expiration.
- **request** is used to communicate with oracles when a smart contract requires data or a decision by a particular actor.

- **getPayoutOptions** checks payout options data.
- **getPremium** checks a premium per application.

On the diagram below, you can see the actions a product service performs.



The code below illustrates how the above-mentioned methods can be invoked.

```
interface IProductService {
        function register(bytes32 _productName, bytes32 _policyFlow)
            external
            returns (uint256 _registrationId);

        function newApplication(
            bytes32 _bpExternalKey,
            uint256 _premium,
            bytes32 _currency,
            uint256[] calldata _payoutOptions
        ) external returns (uint256 _applicationId);
```

(continues on next page)

(continued from previous page)

```
12
13          function underwrite(uint256 applicationId)
14              external
15              returns (uint256 _policyId);
16
17          function decline(uint256 _applicationId) external;
18
19          function newClaim(uint256 _policyId) external returns (uint256 _claimId);
20
21          function confirmClaim(uint256 _claimId, uint256 _sum)
22              external
23              returns (uint256 _payoutId);
24
25          function declineClaim(uint256 _claimId) external;
26
27          function expire(uint256 _policyId) external;
28
29          function payout(uint256 _payoutId, uint256 _sum)
30              external
31              returns (uint256 _remainder);
32
33          function getPayoutOptions(uint256 _applicationId)
34              external
35              returns (uint256[] memory _payoutOptions);
36
37          function getPremium(uint256 _applicationId)
38              external
39              returns (uint256 _premium);
40
41          function request(
42              bytes calldata _input,
43              string calldata _callbackMethodName,
44              address _callabackContractAddress,
45              bytes32 _oracleTypeName,
46              uint256 _responsibleOracleId
47              ) external returns (uint256 _requestId);
48      }
```

3.2 Policy Flow

The **policy flow contract** implements business logic for a policy life cycle. A product contract should specify a desired policy flow contract during registration. The policy flow contract has permissions to manage modules.

A policy life cycle could be defined as a "state machine." By this definition, a policy flow contract specifies transition rules between states of core objects (applications, policies, claims, and payouts) and a sequence of actions that manage the "state machine."

A policy flow contract contains the logic of how to handle the GIF contract modules and operate application, policy, claim and payout entities.

CHAPTER 4

Modules

A module represents a group of smart contracts, with each module containing at least one storage and one controller contract.

A storage contract acts as a database for the core objects. A controller contract includes an implementation that helps to manage core objects in a storage contract. In its turn, a storage contract delegates methods and makes calls to a controller contract, which modifies the state of a storage contract.

Here is the list of the modules behind the Generic Insurance Framework:

- a **policy module** (manages applications, policies, claims, payouts, and metadata objects)
- a **registry module** (registers sets of the core contracts used in a policy flow lifecycle in release groups)
- a **license module** (manages products)
- a **query module** (manages queries made to oracles and delivers responses from them).

CHAPTER 5

The license module

The **license module** stores registration data and the data related to the registered products. The module is responsible for authorization of a particular contract address and rejects calls from unauthorized senders.

The approval or disapproval of calls is managed by the responsible methods invoked by the instance operator. A product contract can be managed by any Ethereum account, be it a single account, a multisig, or a DAO.

Product contracts are registered in a smart contract, and its registration proposal is on review for the instance operator, which can then perform certain actions related to the registration of product contracts.

All **license controller methods** are used by the instance operator, except for the register method, which can be called by product owners only.

The methods invoked by the license controller include:

- **register** is used to register a proposal by a product contract.
- **declineRegistration** is called by the instance operator to decline registration.
- **approveRegistration** is called by the instance operator to approve registration.
- **disapproveProduct** is called when the instance operator wants to decline the registration, which was previously approved by it.
- **reapproveProduct** is used to approve the registration after it was declined by the instance operator.
- **pauseProduct** is employed by the instance operator to pause a product contract.
- **unpauseProduct** is used by the instance operator to unpause a product contract.
- **isApprovedProduct** is used by the instance operator to check if a product contract is approved.
- **isPausedProduct** is used by the instance operator to check if a product contract is paused.
- **isValidCall** is used by the instance operator to check if a product contract call is valid.
- **authorize** is used by the instance operator to check if a product contract address is authorized and what policy flow it uses.
- **getProductId** is used by the instance operator to check a product contract ID.

Below, you can see how to invoke all the above-mentioned methods available through the license controller.

```
interface ILicenseController {
function register(bytes32 _name, address _addr, bytes32 _policyFlow)
    external
    returns (uint256 _id);

function approveProduct(uint256 _id) external;

function disapproveProduct(uint256 _id) external;

function pauseProduct(uint256 _id) external;

function unpauseProduct(uint256 _id) external;

function isApprovedProduct(address _addr)
    external
    view
    returns (bool _approved);

function isPausedProduct(address _addr)
    external
    view
    returns (bool _paused);

function isValidCall(address _addr) external view returns (bool _valid);

function authorize(address _sender)
    external
    view
    returns (bool _authorized, address _policyFlow);

function getProductId(address _addr)
    external
    view
    returns (uint256 _productId);
    }
```

CHAPTER 6

The policy module

The **policy module** is responsible for managing applications, policies, claims, payouts, and metadata objects. The policy module is managed by a policy flow contract.

The methods invoked by the policy controller are as follows:

- **createPolicyFlow** is called to create a new policy flow.
- **setPolicyFlowState** is employed to set a policy flow state.
- **createApplication** is used to create a new application.
- **setApplicationState** sets an application state.
- **getApplicationData** helps to view application data per application ID.
- **getPayoutOptions** is called to view payout options per application ID.
- **getPremium** is invoked to view a premium amount per application ID.
- **createPolicy** creates a new policy.
- **setPolicyState** automatically sets a policy state.
- **createClaim** creates a new claim.
- **setClaimState** automatically sets a claim state.
- **createPayout** creates a new payout.
- **payOut** is called to get data on a payout remainder.
- **setPayoutState** automatically sets a payout state.

The code below illustrates how to invoke the above-mentioned methods of the policy module.

```
1  interface IPolicyController {
2  function createPolicyFlow(uint256 _productId, bytes32 _bpExternalKey)
3      external
4      returns (uint256 _metadataId);
5
```

(continues on next page)

(continued from previous page)

```
    function setPolicyFlowState(
        uint256 _productId,
        uint256 _metadataId,
        IPolicy.PolicyFlowState _state
    ) external;

    function createApplication(
        uint256 _productId,
        uint256 _metadataId,
        uint256 _premium,
        bytes32 _currency,
        uint256[] calldata _payoutOptions
    ) external returns (uint256 _applicationId);

    function setApplicationState(
        uint256 _productId,
        uint256 _applicationId,
        IPolicy.ApplicationState _state
    ) external;

    function createPolicy(uint256 _productId, uint256 _metadataId)
        external
        returns (uint256 _policyId);

    function setPolicyState(
        uint256 _productId,
        uint256 _policyId,
        IPolicy.PolicyState _state
    ) external;

    function createClaim(uint256 _productId, uint256 _policyId, bytes32 _data)
        external
        returns (uint256 _claimId);

    function setClaimState(
        uint256 _productId,
        uint256 _claimId,
        IPolicy.ClaimState _state
    ) external;

    function createPayout(uint256 _productId, uint256 _claimId, uint256 _amount)
        external
        returns (uint256 _payoutId);

    function payOut(uint256 _productId, uint256 _payoutId, uint256 _amount)
        external
        returns (uint256 _remainder);

    function setPayoutState(
        uint256 _productId,
        uint256 _payoutId,
        IPolicy.PayoutState _state
    ) external;

    function getApplicationData(uint256 _productId, uint256 _applicationId)
        external
        view
```

(continues on next page)

(continued from previous page)

```
    returns (
    uint256 _metadataId,
    uint256 _premium,
    bytes32 _currency,
    IPolicy.ApplicationState _state
);

function getPayoutOptions(uint256 _productId, uint256 _applicationId)
    external
    view
    returns (uint256[] memory _payoutOptions);

function getPremium(uint256 _productId, uint256 _applicationId)
    external
    view
    returns (uint256 _premium);

function getApplicationState(uint256 _productId, uint256 _applicationId)
    external
    view
    returns (IPolicy.ApplicationState _state);

function getPolicyState(uint256 _productId, uint256 _policyId)
    external
    view
    returns (IPolicy.PolicyState _state);

function getClaimState(uint256 _productId, uint256 _claimId)
    external
    view
    returns (IPolicy.ClaimState _state);

function getPayoutState(uint256 _productId, uint256 _payoutId)
    external
    view
    returns (IPolicy.PayoutState _state);
    }
```

CHAPTER 7

The query module

The **query module** allows any product contract to use oracles and access risk model data or get a confirmation about a particular real-world event off-chain.

The methods invoked by the query module include:

- **proposeOracleType** is called by oracle owners or product owners to submit a data input, a callback format, and definitions for a particular oracle type.
- **activateOracleType** is used by the instance operator to activate an oracle type.
- **deactivateOracleType** is employed by the instance operator to deactivate an oracle type.
- **removeOracleType** is used by the instance operator to remove an oracle type.
- **proposeOracle** is called by oracle owners or product owners to propose a particular oracle.
- **updateOracleContract** is called by oracle owners or product owners to update an oracle contract for a particular oracle.
- **activateOracle** is used by the instance operator to activate an oracle.
- **deactivateOracle** is used by the instance operator to deactivate an oracle.
- **proposeOracleToType** is called by oracle or product owners to propose a particular oracle to a specific oracle type.
- **revokeOracleToTypeProposal** is called by oracle owners or product owners to remove a proposal before it is approved.
- **assignOracleToOracleType** is used by the instance operator to assign an oracle to an oracle type.
- **removeOracleFromOracleType** is used by the instance operator to remove an oracle from an oracle type.
- **request** is called by a product to request data from an oracle by an oracle type.
- **respond** is called by the Oracle Service after an oracle response to respond to the request of a product.

Below, you can see how the above-mentioned methods can be invoked.

```
interface IQueryController {
function proposeOracleType(
    bytes32 _oracleTypeName,
    string calldata _inputFormat,
    string calldata _callbackFormat,
    string calldata _description
) external;

function activateOracleType(bytes32 _oracleTypeName) external;

function deactivateOracleType(bytes32 _oracleTypeName) external;

function removeOracleType(bytes32 _oracleTypeName) external;

function proposeOracle(
    address _sender,
    address _oracleContract,
    string calldata _description
) external returns (uint256 _oracleId);

function updateOracleContract(
    address _sender,
    address _newOracleContract,
    uint256 _oracleId
) external;

function activateOracle(uint256 _oracleId) external;

function deactivateOracle(uint256 _oracleId) external;

function removeOracle(uint256 _oracleId) external;

function proposeOracleToType(
    address _sender,
    bytes32 _oracleTypeName,
    uint256 _oracleId
) external returns (uint256 _proposalId);

function revokeOracleToTypeProposal(
    address _sender,
    bytes32 _oracleTypeName,
    uint256 _proposalId
) external;

function assignOracleToOracleType(
    bytes32 _oracleTypeName,
    uint256 _proposalId
) external;

function removeOracleFromOracleType(
    bytes32 _oracleTypeName,
    uint256 _oracleId
) external;

function request(
    bytes calldata _input,
    string calldata _callbackMethodName,
```

(continues on next page)

(continued from previous page)

```
58      address _callabackContractAddress,
59      bytes32 _oracleTypeName,
60      uint256 _responsibleOracleId
61  ) external returns (uint256 _requestId);
62
63  function respond(
64      uint256 _requestId,
65      address _responder,
66      bytes calldata _data
67  ) external returns (uint256 _responseId);
68      }
```

CHAPTER 8

The registry module

The **registry module** is responsible for registering sets of core contracts, which are used in a policy flow life cycle in release groups. The registry module is managed by the instance operator.

The functions available through this module are the following:

- **registerInRelease** is used to register new policies in a new release version.
- **register** is used to register a contract in the last release.
- **deregisterInRelease** is used to delete a contract from a release.
- **deregister** is used to delete a contract in the last release.
- **prepareRelease** is called to create a new release, move contracts from the last release to a new one, and update a release version.
- **getInContractRelease** is used to get a contract address depending on a release version.
- **getContract** is used to get a contract address in the last release.
- **getRelease** is used to get the last release's number.
- **registerService** is used to register a new service.
- **getService** is used to view a new service.

The code below illustrates how to invoke the functions of the registry module listed above.

```
interface IRegistryController {
function registerInRelease(
uint256 _release,
bytes32 _contractName,
address _contractAddress
) external;

function register(
bytes32 _contractName,
address _contractAddress
```

(continues on next page)

(continued from previous page)

```
) external;

function registerService(
bytes32 _name,
address _addr
) external;

function deregisterInRelease(
uint256 _release,
bytes32 _contractName
) external;

function deregister(
bytes32 _contractName
) external;

function prepareRelease(
) external returns (uint256 _release);

function getContractInRelease(
uint256 _release,
bytes32 _contractName
) external
view
returns (address _contractAddress);

function getContract(bytes32 _contractName
) external
view
returns (address _contractAddress);

function getService(bytes32 _contractName
) external
view
returns (address _contractAddress);

function getRelease(
) external view returns (uint256 _release);
}
```

CHAPTER 9

Use Cases for Product Owners

9.1 Register a product

For any product that expects to perform certain actions, it is crucial to register its product contract within the GIF instance.

The registration of a product contract takes place on a smart contract level.

A product creates a contract and inherits one of the GIF contracts — a product contract. After inheriting, a product contract is able to use the GIF functions to describe its business process.

The **register** function (see the code below) is used to register a new product contract. After approval, a contract obtains access to call entry methods.

```
function _register(
bytes32 _productName,
bytes32 _policyFlow
) internal
```

9.2 Role assignment by a product

To assign roles to specific contracts or people, role-based access control (RBAC) is used. A product contract should set up roles and specify what method can be called and by which role.

A lot depends on a business process, but there are two possible cases.

In the **first scenario**, actions can be called by people. It means a product contract may create a role, assign it to particular person, and this person will call a function (i.e., underwrite an application).

In the **second scenario**, actions perform a product contract. An oracle may respond with certain data, then a product contract will need to create a function of an oracle response handler, write certain logic, and automatically call the underwrite function.

A product owner defines necessary roles for its product contract and those who will be appointed to the created roles.

The data related to the roles is kept by a product contract. It inherits the "Product" contract and, after that, gets access to the RBAC methods. So, the roles belonging to the accounts and account data are stored in the product's account (without storage on the GIF).

The code below illustrates the contract details and function calls available.

```
contract RBAC {
        mapping(bytes32 => uint256) public roles;
        bytes32[] public rolesKeys;
        mapping(address => uint256) public permissions;
        modifier onlyWithRole(bytes32 _role) {
            require(hasRole(msg.sender, _role));
            _;
        }

        function createRole(bytes32 _role) public {
            require(roles[_role] == 0);
            // todo: check overflow
            roles[_role] = 1 << rolesKeys.length;
            rolesKeys.push(_role);
        }

        function addRoleToAccount(address _address, bytes32 _role) public {
            require(roles[_role] != 0);
            permissions[_address] = permissions[_address] | roles[_role];
        }

        function cleanRolesForAccount(address _address) public {
            delete permissions[_address];
        }

        function hasRole(address _address, bytes32 _role)
            public
            view
            returns (bool _hasRole)
        {
            _hasRole = (permissions[_address] & roles[_role]) > 0;
        }
    }
```

CHAPTER 10

Implementing a product policy workflow

10.1 Using a generic policy workflow

The GIF provides a list of entities to manage insurance business processes:

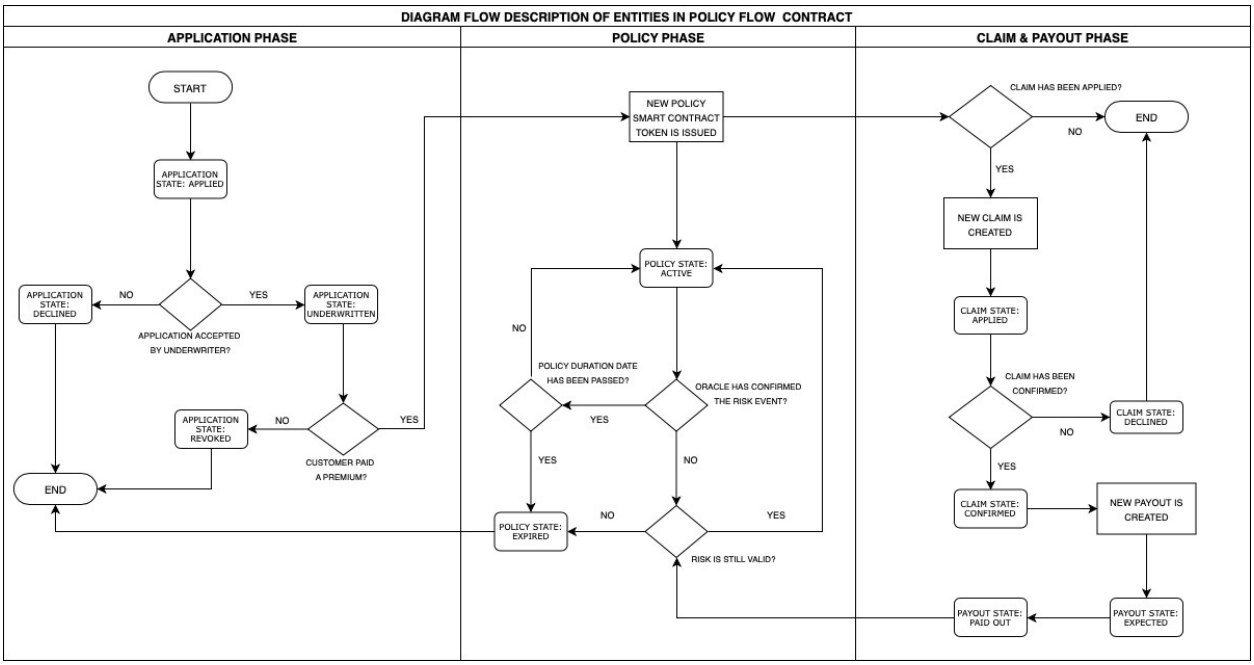
- an application
- a policy
- a claim
- a payout

These entities represent a generic policy workflow. In the course of a workflow, the state of entities will be changed visualizing insurance business process.

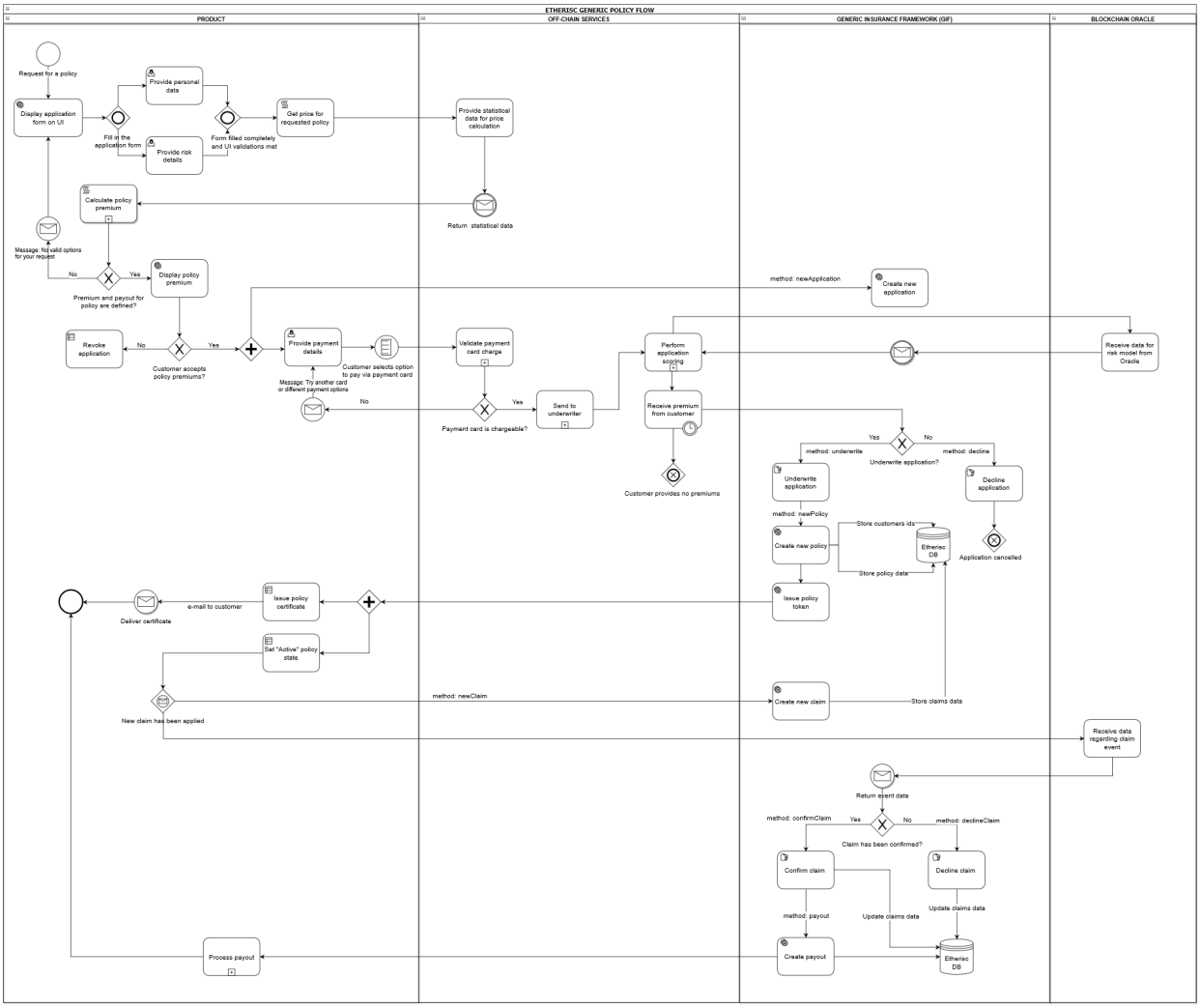
A product contract is able to use a workflow with both prepaid (before issuing a policy) and postpaid (after issuing a policy) premiums. On the diagram below, there are more details for a default scheme with prepaid premiums.

There are two possible ways of choosing premiums by a customer of a product: based on a fixed premium (a payout will correspond with a chosen premium) and based on a fixed payout (a premium will correspond with the desired amount of a payout).

A policy state flow diagram



The Business Process Model and Notation policy flow with prepaid premium diagram



Note: An example of a policy flow described above is one of the two possible flows (with a premium paid before a policy is issued). It is also possible to pay a premium after a policy issuance.

10.1.1 Managing an application

This insurance business process actually starts when any customer (or it might be an application from any organization that represents the interests of a group of customers) sends an application for an insurance policy via a user interface on a product contract level.

The DIP Protocol enables a product contract to perform the following actions:

- create an application
- underwrite an application
- decline an application
- revoke an application

Creating an application

To create an application for a policy, the **newApplication** function needs to be used. This function is invoked to store new application data, which contains such fields as a premium amount, currency, payout options, risk definition, etc. A policy buyer signs a policy agreement using this function.

The application state is "Applied." During scoring or underwriting processes, an application remains in the "Applied" status.

The code below demonstrates how the function is called.

```
function _newApplication(
    bytes32 _bpExternalKey,
    uint256 _premium,
    bytes32 _currency,
    uint256[] memory _payoutOptions
) internal returns (uint256 _applicationId) {
    _applicationId = productService.newApplication(
        _bpExternalKey,
        _premium,
        _currency,
        _payoutOptions
    );
}
```

Underwriting an application

To sign a policy agreement by a product contract, the **underwrite** function has to be used.

As soon as an application is accepted by an underwriter, its state is changed to "Underwritten." The application remains in the "Underwritten" state when a new policy is created.

The code below demonstrates how the function is invoked.

```
function _underwrite(uint256 _applicationId)
    internal
    returns (uint256 _policyId)
{
    _policyId = productService.underwrite(_applicationId);
}
```

Declining an application

This function is used simply to decline an application. The application state changes to "Declined."

The code below illustrates how the function performs.

```
function _decline(uint256 _applicationId) internal {
    productService.decline(_applicationId);
}
```

10.1.2 Managing a policy

By default, before issuing a policy, an underwriter must confirm that policy premiums are fully paid.

When a customer has an application underwritten and paid a premium for a product policy, the GIF methods allow to fulfill the following actions:

- create a policy
- expire a policy

Creating a policy

This function allows to create a new entity: issue a new policy token. A policy is created with the "Active" state.

A product contract sends a PDF policy certificate to a customer using the PDF Generator core microservice.

Expiring a policy

The function is used to set a policy expiration. The possible cases are the following:

- A policy duration date has expired.
- A risk for a policy has been confirmed and paid out (in case a risk is to be paid out once).
- The event has not been confirmed by an oracle in the course of a policy duration, which means no payout.

When the function is performed, a policy state is set as "Expired."

The code below demonstrates how to use the **expire** function.

```
function _expire(uint256 _policyId) internal {
    productService.expire(_policyId);
}
```

10.1.3 Managing a claim

The DIP allows products contracts to use the claim management methods. Specifically, the following actions can be performed:

- apply a claim
- confirm a claim
- decline a claim

Applying a claim

The function is used to declare a new claim. The claim state is set as "Applied."

Note: Claims can be applied when a policy has the "Active" or "Expired" status.

The code below demonstrates how the function is invoked.

```
function _newClaim(uint256 _policyId) internal returns (uint256 _claimId) {
   _claimId = productService.newClaim(_policyId);
}
```

Confirming a claim

The function is used to confirm a claim. A new payout object is created after performing this action. The claim state is set as "Confirmed."

The code below illustrates how the function performs.

```
function _confirmClaim(uint256 _claimId, uint256 _amount)
    internal
    returns (uint256 _payoutId)
{
    _payoutId = productService.confirmClaim(_claimId, _amount);
}
```

Declining a claim

This function is used to decline a claim. The claim state is set as "Declined."

The code below illustrates how the function is invoked.

```
function _decline(uint256 _applicationId) internal {
    productService.decline(_applicationId);
}
```

10.1.4 Managing a payout

Confirming a payout

The method is used to confirm the payout that has actually happened. The payout state changes to "PaidOut."

```
function _payout(uint256 _payoutId, uint256 _amount)
    internal
    returns (uint256 _remainder)
{
    _remainder = productService.payout(_payoutId, _amount);
}
```

10.2 Creating a new or update default policy workflow

Currently, the GIF offers a general purpose default policy workflow to products contracts. In case a product contract needs to update a default workflow or create a new one, there are three possible options to do this:

1. Pull a request from a product contract. This request will be reviewed by the Etherisc team and merged with the existing workflow. It may also be a new version of a policy workflow.
2. Create an issue on GitHub. A product contract can create an issue, and, after reviewal, the Etherisc team will plan the requested improvements on a policy workflow.
3. Direct a request via e-mail: contact@etherisc.com.

CHAPTER 11

On-chain and off-chain storage

Any Product on the DIP has a choice of:

- what type of data to store
- where to store data

The DIP storage model allows products contracts to store its data on:

- blockchain smart contracts
- a platform database
- a product database

Note: Payment card data should be stored on a payment provider level as it requires PCI compliance to store payment card data of customers in a database.

In many countries, a legal agreement is needed between a party that runs a storage service and a party that uses a storage service.

11.1 On-chain

As the DIP operates in the Ethereum environment, the term **on-chain** specifies smart contracts, where a product can store risk description and specific metadata per policy.

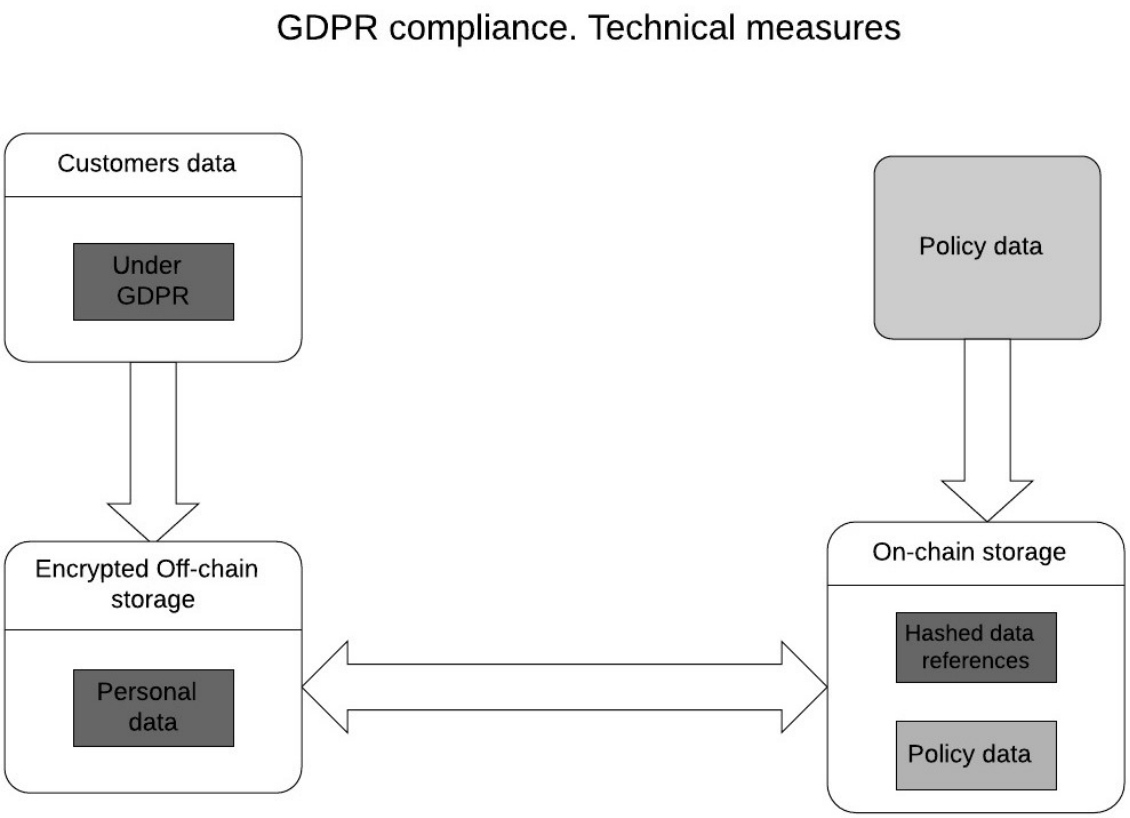
The key principle of how the DIP itself uses data is that no personal data is kept in smart contracts — only **unique hashed references**.

The DIP allows to store data for any product regarding its:

- customers (**Note:** *first_name, last_name,* and *e-mail* fields are required by the GIF.)
- policies
- claims

In case a product doesn't want to use a platform database, it is possible to use the product's database.

Attention: According to the EU General Data Protection Regulation requirements, we prevent you from storing personal data of customers on-chain. This data is to be stored **off-chain only**. There exist special identificators stored on-chain (hashed data references), which allow for retrieving data from an off-chain database. This prevents unauthorized access to sensitive data by an on-chain identificator. The diagram below illustrates the relations between on-chain and off-chain storage. This methodology implements the "Positionspapier des Bundesblock," the german association of blockchain companies which tries to implement this methodology in EU law.



11.2 Profiling

To avoid the possibility of the so-called customer "profiling," each newly issued policy gets a new unique customer ID (unique hashed reference).

CHAPTER 12

Make Payouts

In order to make payouts in fiat money, a product contract needs to use the GIF **Payout microservice**.

Your product **App** (here we mean a server App body of your product that connects and coordinates its on-chain and off-chain parts) needs to subscribe to the **Event Listener** microservice to get notifications about the off-chain events related to your product. This way, the product **App** knows that a new entity **"Payout"** appears with the **"expected"** state. To make a payout, a product contract should send a message to the framework with the following structure:

```
{
  id: 'payout',
  type: 'object',
  properties: {
    policyId: { type: 'string' },
    payoutAmount: { type: 'number' },
    currency: { type: 'string' },
    provider: { type: 'string' },
    contractPayoutId: { type: 'string' },
  },
```

Where the *'policyId'*, *'payoutAmount'*, *'currency'*, *'provider'*, and *'contractPayoutId'* attributes are required to be defined.

The product contract sends the above-mentioned message to the **Payout** microservice. The information should contain the address where payout funds are to be transferred and the transfer method (e.g., transfer to a bank account, payment card, a transferwise, PayPal account, coin wallet, post transfer, etc.).

Below, you can find an example of a payout message (referred to a particular policy, with ID equal to 1) made by the Payout microservice ("Transferwise"). The payout is made in fiat money (100 EUR) and a product contract is to be notified about this.

```
1 {
2 policyId: 1,
3 payoutAmount: 100,
4 currency: 'EUR',
5 provider: 'transferwise',
6 }
```

To describe the process in more detail, we provide the following clarification of the interaction between the GIF and product components during the payout process.

There are a few entities involved in the process of payout: the *product smart contract*, the *product App*, the *ProductService contract*, the *Policy module* and some microservices (such as *Event Listener*, *Payout* microservice and *Ethereum signer*).

The process starts with confirming a claim by calling the **_confirmClaim** function in the product contract. This function is addressed to the **ProductService** contract, which delegates it to the **Policy module**. The function is performed here and an entity **"Claim"** changes its state to **"Confirmed"**.

At the same time, a new entity **"Payout"** is created at the **Policy** module with the **"expected"** state. When the states are changed, the **LogPayoutStateChanged** event takes place. This event, like many other events, is "listened" by a particular microservice — **Event Listener** subscribed to the events of the core contract.

Then, Event Listener notifies the product **App** (with a business logic implemented) about this event. This **App** exchanges messages with other microservices involved (the **Payout** microservice and the **Ethereum signer** microservice) in order to make a payout. The **Ethereum signer** microservice can make payouts on the Ethereum blockchain. It also notifies the **product** contract, which calls the **_confirmPayout** function and addresses it to the **ProductService** contract.

This results to a similar flow (*ProductService — Policy module* — the payout changes state to *"Paid out"* — the *LogPayoutStateChanged* event occurs — *Event Listener* notifies the product *App* — the product *App* orders the *Notification* microservice to notify the customer about the payout — the *Notification* microservice sends a message to the customer and reports about it to the product *App*).

CHAPTER 13

Managing oracles

13.1 Actors

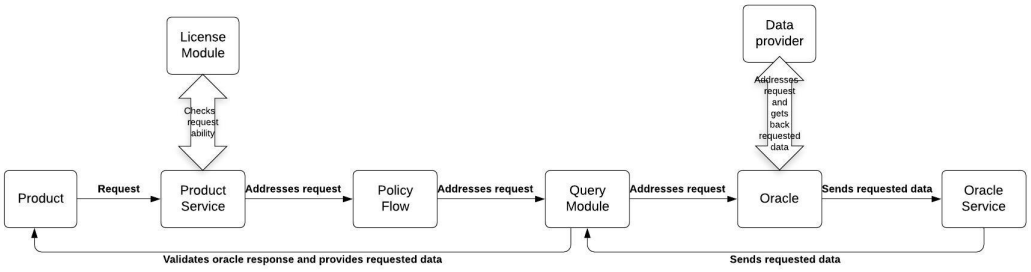
- A **Product** is a contract that provides a specific service to customers.
- The **Query module** is a service that forwards requests to different oracle providers.
- An **Oracle** is a service used to provide specific data to products.

13.2 Description

Oracle services provide huge leverage to the Generic Insurance Framework.

Product contracts use an oracle to obtain off-chain data and confirm or decline real-world events vital for an insurance process. The scheme below illustrates the request flow from the beginning (a product sends a request) till the end (a product receives the requested data).

The GIF accomplished a strategy for product contracts to get data from a specific oracle, which a product is particularly interested in.



13.3 A workflow

13.3.1 Registering an oracle

- Any product owner, oracle owner, or the instance operator is able to register its oracle type, where they specify criteria for the oracles that provide data back to the requesting parties. For this purpose, the **proposeOracleType** function is used. The parameters, such as *oracleTypeName*, *inputFormat*, *inputDefinitions*, *callbackFormat*, *callbackDefinitions*, and currency are defined here. Then, the instance operator activates an oracle type. It can also deactivate an oracle type.
- **deactivateOracle** is used by the instance operator to deactivate an oracle.

13.3.2 Registering an oracle to type

- An oracle owner, a product owner, or the instance operator can propose a particular oracle. For this purpose, the **proposeOracleToType** method is called to propose a particular oracle to a particular oracle type. The necessary parameters of the method are: *oracleTypeName*, *inputFormat*, *callbackFormat*, and a description.
- **revokeOracleToTypeProposal** is called by oracle owners or product owners to remove a proposal, before it is approved.
- **assignOracleToOracleType** is called by the instance operator to assign an oracle to an oracle type.
- **removeOracleFromOracleType** is called by the instance operator to remove an oracle from an oracle type.

13.3.3 Updating an oracle contract

- Oracle owners or product owners can update an oracle contract for a particular oracle.
- **updateOracleContract** is called to update an oracle contract.

13.3.4 Creating a request

- Product calls **request** a function.
- **ProductService** receives a request and verifies the correctness of the request according to the permissions of the License Module.
- **ProductSservice** addresses a request to a policy flow.
- A policy flow addresses a request to the query module to connect an oracle.
- The query module executes a request to a particular oracle that requests data from the data provider.
- An oracle calls to the requested data provider (i.e., Oraclize) to obtain the necessary data.
- **request** is called by a product contract to request data from an oracle by an oracle type. The request function uses the following arguments: *callbackMethodName*, *callbackContractAddress*, *oracleTypeName*, and *responsibleOracleID*.

13.3.5 Receiving a callback

- A particular data provider performs a callback to an oracle.
- An oracle sends a response to an oracle service with received data as an answer for the request.

- An oracle service addresses the received data to the query module, where the sender and the addressee are being verified. The query module specifies the response (which product contract made a query, what an oracle type is, and which Oracle is to respond to the query).
- Then, the response is to be checked. An oracle is confirmed to be registered to the system and to be assigned to an oracle type, which corresponds to that of the query. If everything matches, then an oracle provides a product contract with the requested data.

13.3.6 Making Respond

- An oracle contract makes a response using the **respond** method and sends the requested data in the **respond**.
- **respond** is called by an oracle service after an oracle responses to the request of a product contract.
- The methods of the query module are used to communicate with oracles when an insurance application requires data or a decision of a particular actor.

The code below illustrates the functions that can be called by the **OracleQueryController**.

```
interface IQueryController {

function proposeOracleType(
    bytes32 _oracleTypeName,
    string calldata _inputFormat,
    string calldata _callbackFormat,
    string calldata _description
) external;

function activateOracleType(bytes32 _oracleTypeName) external;

function deactivateOracleType(bytes32 _oracleTypeName) external;

function removeOracleType(bytes32 _oracleTypeName) external;

function proposeOracle(
    address _sender,
    address _oracleContract,
    string calldata _description
) external returns (uint256 _oracleId);

function updateOracleContract(
    address _sender,
    address _newOracleContract,
    uint256 _oracleId
) external;

function activateOracle(uint256 _oracleId) external;

function deactivateOracle(uint256 _oracleId) external;

function removeOracle(uint256 _oracleId) external;

function proposeOracleToType(
    address _sender,
    bytes32 _oracleTypeName,
    uint256 _oracleId
) external returns (uint256 _proposalId);

```

(continues on next page)

(continued from previous page)

```
function revokeOracleToTypeProposal(
    address _sender,
    bytes32 _oracleTypeName,
    uint256 _proposalId
) external;

function assignOracleToOracleType(
    bytes32 _oracleTypeName,
    uint256 _proposalId
) external;

function removeOracleFromOracleType(
    bytes32 _oracleTypeName,
    uint256 _oracleId
) external;

function request(
    bytes calldata _input,
    string calldata _callbackMethodName,
    address _callabackContractAddress,
    bytes32 _oracleTypeName,
    uint256 _responsibleOracleId
) external returns (uint256 _requestId);

function respond(
    uint256 _requestId,
    address _responder,
    bytes calldata _data
) external returns (uint256 _responseId);
}
```

CHAPTER 14

Upgrading policies

Once an insurance agreement is signed and a policy is issued, parties cannot unilaterally change the specified behavior. It means, the framework must ensure that all the parties involved can always exactly predict which set of smart contracts will execute the policy.

There are two ways to provide upgradability of contracts within the system and two different situations that could trigger an update of a smart contract:

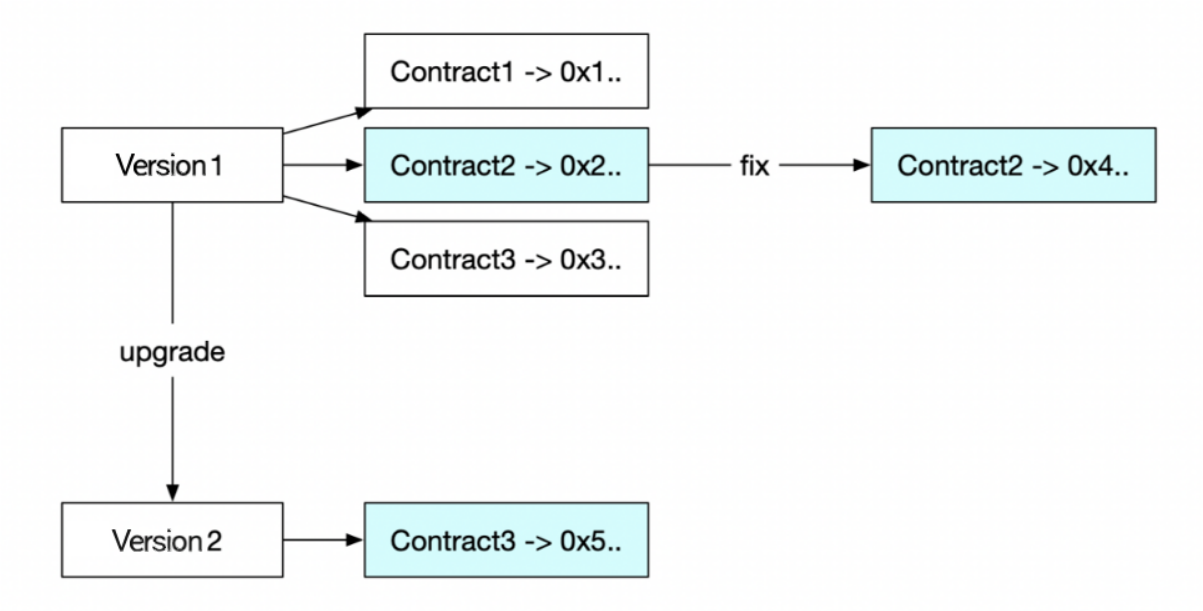
- **Fixes**. In case a bug has been detected. A bug is a deviation of expected behaviour to actual behaviour—a difference between a specification and an implementation. Bugs can be technical (a flawed implementation of a certain calculation leading to wrong results), but they can also occur in the translation process from legal prose to code. In this case, the specification would be flawed, and correct implementation of a flawed specification still leads to wrong results.
- **Upgrades**. New features need to be implemented for various reasons: modifications in pricing, a risk model, etc. In this case, the specification changes.

Both cases can occur on the core contract or product-specific contract levels.

We handle the two cases differently:

1. A "bug fix" upgrade will affect all policies, both the existing and new ones.
2. A "new feature" upgrade will affect only new policies. The existing policies will be executed with the original set of smart contracts (modulo bug fixes).

The following diagram illustrates the above-mentioned ways to change contracts and the result of such changes (when you get a new contract address). Then you can replace the existing contract by the new one if there was a "bug fixing," or deploy a new "additional" version of the contract if there was an upgrade of the existing contract.


Contract1 -> 0x1..
Version 1
Contract2 -> 0x2..
fix
Contract2 -> 0x4..
Contract3 -> 0x3..
upgrade
Version 2
Contract3 -> 0x5..

CHAPTER 15

Notify Clients

To provide notifications to its customers Product's contract can use Notification microservice, which provides notifications based on triggering events.

This microservice informs Product's contract and its customers about the processing and the state of his application for policy, policy itself, payment details, his claim for policy and payout details via e-mail as well as notifies Product builders via Telegram chat specifically.

There are two main transports available for Product contractss on Etherisc platform:

- SMTP transport
- Telegram transport

Notification microservice receives an event message about the need for delivery from other microservices and sends it for delivery to the appropriate transports.

Every message for Notification MS contains what, where and with what credentials to send.

The event message should include the following data:

1. name of the template;
2. data to fill in the template;
3. properties (i.e. recipient e-mail)

Product may create branded templates to use instead of default templates. Notification microservice simply takes a template, substitutes the data and sends it where it is requested. A Product's contract can have templates in any languages, saving them to platform templates.

Product application provides notificationSettingsUpdate event with branded templates and description what notification are to be used.

See code details:

```
transports: [
        {
          name: 'smtp',
```

(continues on next page)

(continued from previous page)

```
          props: {
            from: 'noreply@etherisc.com',
        },
          events: ['application_declined', 'application_error'],
        },
        {
          name: 'telegram',
          props: {
            chatId:'54321',
        },
          events: ['application_declined', 'application_error'],
        },
      ],

templates: [
        {
          name: 'application_error',
          transport: 'smtp',
          template: '<h1>Application Error {{policy.id}}</h1>',
        },
        {
          name: 'application_error',
          transport: 'telegram',
          template: 'Application Error {{policy.id}}',
        },
      ];

      type: 'application_error',
      data: { policy: { id: 1514 } },
      props: {
        recipient: 'foo@email.com',
        subject: 'Etherisc application error',
      }
```

See events triggers here:

Message Name	Description	Triggered
quote_successful	Your quote is created successfully. We are contacting you regarding payment details. Please provide a necessary payment for premium	Application state is changed to "Underwritten"
charge_cancelled	We're unable to process your payment card. Please try another card	Payment request to provider was not successful
application_declined	We are sorry to inform that the application for Policy has been declined by underwriter	Application state is changed to "Declined"
application_error	Technical error: the requested application has not been found	Application has not been found
premium_paid	We've received your premium payment	Customer has paid a premium
application_revoked	We confirm that the application for Policy has been revoked. Your premium minus cancellation fee will be sent back to you	Application state is changed to "Revoked"
policy_issued	We confirm that Policy has been issued. Details below...	Token has been issued. Policy is created
policy_error	Technical error: the requested policy has not been found	Policy has not been found
policy_expired	We inform you that Policy has been expired	Policy state is changed to "Expired"
claim_confirmed	Your claim regarding Policy has been confirmed. The payout will be transferred to your payment card	Claim state is changed to "Confirmed"
claim_declined	We inform you that your claim regarding Policy has been declined	Claim state is changed to "Declined"
claim_error	Technical error: the requested claim has not been found	Claim has not been found
claim_paid_out	Your claim for Policy has been paid out	Payout state is changed to "PaidOut"

CHAPTER 16

View Ledger of Funds on Different Accounts

There are several accounts we distribute money to:

- Balance - wallet is used to transfer funds;
- Premium - amount of money client pays for the policy;
- Risk Fund - fund which covers risks in case of payout;
- Operations Fund - fund which covers all operational costs (pricing, underwriting, oracle costs);
- Oracle Costs - amount of money we pay for pricing, underwriting, events confirmation;
- Payout - amount of money client gets when policy risk actually happens;
- Reward - payment to Etherisc platform from premiums (might not need it).

CHAPTER 17

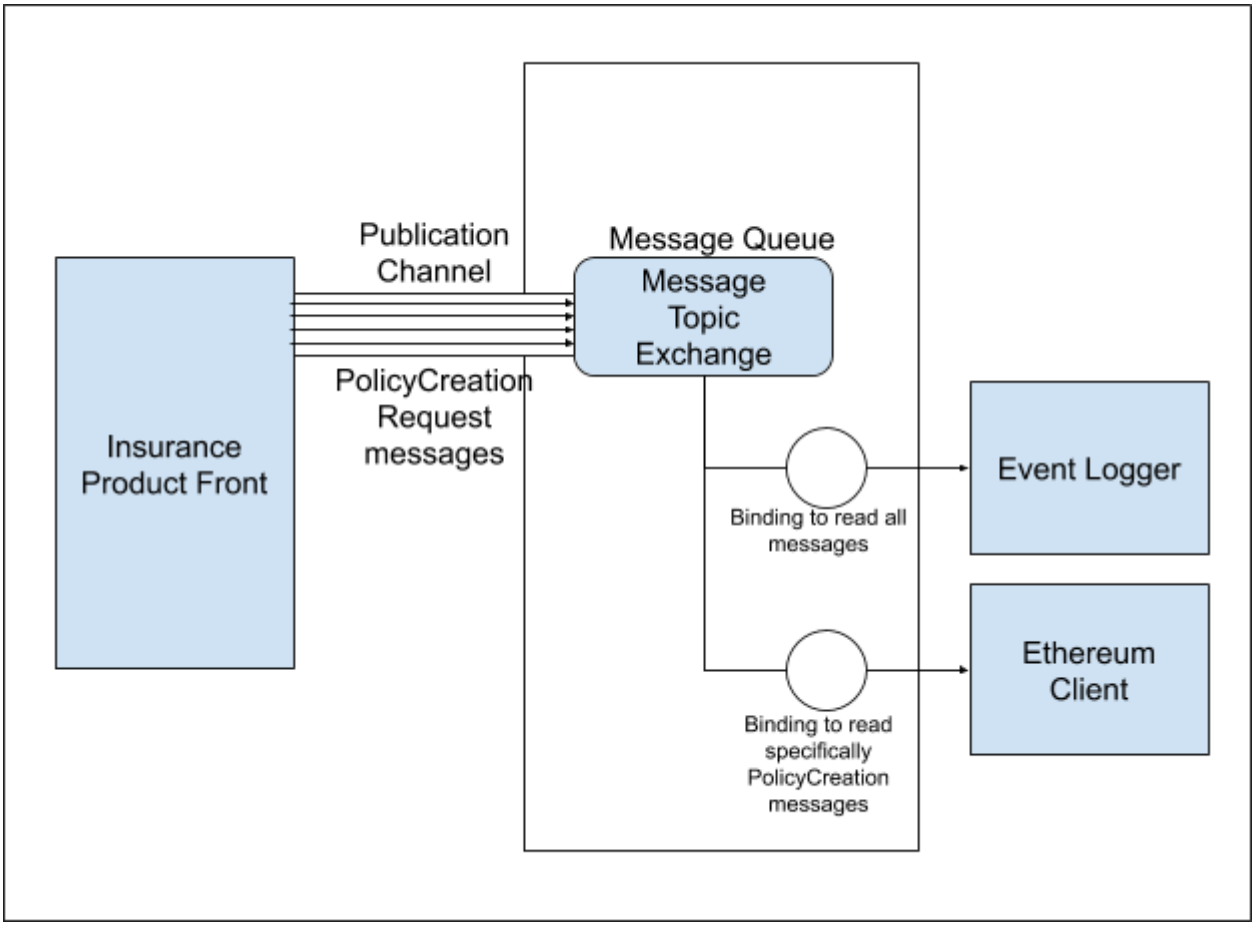
Message Queue

The concept used for connecting numerous platform microservices is the following: to use a central message queue supporting the AMQP protocol, currently implemented in RabbitMQ, to deliver signals between different actors. As one of the possible patterns for organization of microservices, the message queue helps to solve some inherent microservices issues, such as:

Discoverability. Each microservice does not need to find all other microservices it interacts with. The only thing it should know is where the queue is.

Extensibility. New types of microservices can be introduced to process the old types of messages without any changes to existing ones.

Resiliency. The message queue serves as a storage for undelivered messages until it needs to deliver them to get the microservice back to the operational state.


Publication
Channel
Message Queue
Message
Topic
Exchange
Insurance
Product Front
PolicyCreation
Request
messages
Event Logger
Binding to read all
messages
Ethereum
Client
Binding to read
specifically
PolicyCreation
messages

CHAPTER 18

The @etherisc/microservice npm package

For convenience, the logic of the message queue interaction for all the node-based microservices (along with some other potentially common adapters, like a relational database or a file storage) was packaged into a npm package called @etherisc/microservice.

Product's contract developers should include @etherisc/microservice as an 'npm' dependency for any of its Product specific microservices.

To get it working properly, you'd need to provide a list of environmental variables:

AMQP_MODE, AMQP_HOST, AMQP_PORT, AMQP_USERNAME, AMQP_PASSWORD - your product credentials for message queue access, which should be provided to you when you register your product to operate on the DIP platform.

The main thing you get for including this npm package is the ability to start your node.js microservice by simply invoking the 'bootstrap' method in your main 'node' executable. That will allow you to pick which general areas you'd like your microservice to work with (amqp, database, storage) or, for the product microservices, which insurance process template you'd like your product to follow.

Let's assume that you want to create a message logging microservice for your product.

```
'bash >
    npm install --save @etherisc/microservice
```

```
const { bootstrap } = require('@etherisc/microservice');

class MessageLogger {
    constructor({ amqp }) {
        this.amqp = amqp;
    }
    async bootstrap() {
        // The code for your particular amqp subscriptions and
        // other asynchronous behaviours to be initialized
    }
}
bootstrap(MessageLogger, {
```

(continues on next page)

(continued from previous page)

```
    amqp: true,
});
```

CHAPTER 19

Message versioning. Publish and Subscribe functions

We'd like to point out that for a microservice architecture controlled via a message queue, two types of objects need to be strictly versioned (following the Semantic Versioning 2.0 specification when possible):

Message types. For all message types on the platform, we're planning to have an up-to-date entry in the '@etherisc/microservice/io/amqp/messages/types' folder that will keep the information on what data is expected to be included in that message, as well as if attributes are required or optional. It was decided to tag each message type with a pair of numbers 'X.Y' in a way that X is the MAJOR version and Y is MINOR version. Any change to the schema of a message should result in aversion change. If the changes are 'breaking' (such as the removal of required attributes), the MAJOR version needs to be incremented, otherwise - the MINOR one.

Microservice repositories. Each individual core microservice application will have a version label consisting of three integers 'X.Y.Z', as well as a reference list of message types it creates and consumes (with their appropriate version tags). The subscription logic inside the service needs to be aware of the version it accepts, while the publishing logic needs to clearly indicate what version of a particular message is produced.

So for the example implementation of MessageLogger, the subscription command would look like this:

```
const { bootstrap } = require('@etherisc/microservice');

class MessageLogger {
    constructor({ amqp }) {
       this.amqp = amqp;
    }
    async bootstrap() {
       this.amqp.consume({
            messageType: '*',
            // A name for the message type you subscribe for, there is a default␣
↪value of '*'
            // that means all types will be handled messagetypeVersion: '#',
            // You can specify the particular version,
            // all the versions for the fixed major ( for example '1.*' ),
            // accept all versions with '#' or '*.*',
            //, or use 'latest' as the value to accept the most recent version only,␣
↪ignoring all others.
            handler: async ({ content, fields, properties }) => {
```

(continues on next page)

(continued from previous page)

```
            // the contents of the handling function
            // In case of our logging service that could be any persistence logic
            // saveToDatabase(content);
            },
        });
    }
}
```

Conversely, the publishing logic, which would most likely be a part of the handler for some external event or another message, may look like this:

```
await this.amqp.publish({
    messageType: 'eventNotification',
    // there is no default here, a particular message type needs to be invoked␣
↪explicitly
    messageVersion: '1.*',
    // The version of the schema to use, accepts some of the masks used for␣
↪subscriptions
    content: { key1: value1, key2: value2, arrayofdata: [data1, data2], nestedobject:
↪{key3: value3} },
    // The content that will be passed to all the consumers, the structure will be␣
↪validated
    // to conform to the structure described in the message type description
correlationId: properties.correlationId,
    // An optional trace attribute employed by RabbitMQ, used for debugging when the
    // handling of one message fires an event that publishes another message
});
```

CHAPTER 20

User Manual for the GIF sandbox Command Line Interface

Once developed by the Etherisc team, the Generic Insurance Framework (GIF) is released as an environment where product builders can create their own products. To access the GIF and to make user experience more enjoyable, the Etherisc team has also created a so-called "sandbox" — a particular working environment where product builders are able to experiment with their products in a test mode, not worrying about possible "unexpected" results.

The next step is to equip product builders with a plain and powerful solution, enabling them to operate the GIF, as well as a sandbox for it. Such a solution is now available for product builders through the GIF Command Line Interface (CLI). Here, we present a quick reference on how to use the Command Line Interface while working with the GIF.

For better understanding of this document, you can refer to the User manual for the GIF to refresh the content and the features of the framework. There you can also find basic methods necessary for creating your own product's contract.

20.1 Prerequisites

To make interactions with the GIF efficient and pleasant, the very first thing to do is to setup the required working environment. Such environment already exists and runs in the sandbox with all the necessary core smart contracts deployed, as well as microservices activated and ready to use. In addition, you should install NodeJS (version 8.12.0 or later, npm version 6.4.1 or later) and Python.

After that, to install the CLI you just need to insert the `npm install -g @etherisc/gifcli` command in the command line of your computer. You can also check the version of the CLI by running the `gifcli version` command, as in the example below.

```
$ npm install -g @etherisc/gifcli

$ gifcli (-v|--version|version)
@etherisc/gifcli/1.1.2 win32-x64 node-v10.15.3
```

20.2 General description

The basic principle of the CLI is the same as of any other command line: to manage a programming working environment with commands. Actually, you run the CLI within the operating system console of your computer (e.g., a command line of Microsoft Windows, Ubuntu, etc.).

Here, we list commands available for product builders and will further describe each command in more detail. There are a few basic commands in the **CLI mode** available right after installation of the **gifcli** necessary for making the very first steps: `gifcli console` , `gifcli exec` , `gifcli help [COMMAND]` (or any other gifcli command with a space after "help" and without an additional "gifcli", e.g. gifcli help user:register), `gifcli user:register` , `gifcli product:create` , `gifcli product:select` , `gifcli artifact:send` and `gifcli user:logout` .

The other commands are used when you have already created a product. These commands (beginning with "gif") are available in the **console mode** , which is run by the `gifcli console` command.

In the table below, you can see a list of commands, available on the GIF CLI.

Group of commands/ methods	CLI mode (gifcli [COMMAND] — commands)	Console + Execute mode (gif.[METHOD] — methods)
Common service commands/ methods	gifcli console gifcli exec gifcli help [type here the name of the necessary command, i.e. product:create]	gif.help() gif.help('type a command here, i.e. product.get')
Commands related to users (product builders)	gifcli user:register gifcli user:logout	—
Commands to manage products	gifcli product:create gifcli product:select gifcli artifact:send	gif.product.get
Methods to set and oversee the business processes of a product	—	gif.bp.create gif.bp.list gif.bp.getById gif.bp.getByKey
Methods to interact with product contracts	—	gif.contract.call gif.contract.send
Methods related to customers (end-users of a product, developed by product builders)	—	gif.customer.create gif.customer.getById gif.customer.list
Methods to manage applications	—	gif.application.list gif.application.getById
Methods to interact with a policy	—	gif.policy.list gif.policy.getById
Methods to manage claims	—	gif.claim.list gif.claim.getById
Methods to work with payouts	—	gif.payout.list gif.payout.getById

When using the CLI for the first time, you need to register a user. Then, you can create your product, add customers, etc.

There are three modes of working with the CLI: a basic **CLI mode** (you can use it by inputing `gifcli [COMMAND]` in your system's command line), a **console mode** (using the `gifcli console command`) and an **execute mode** (by the `gifcli exec` command). As we've mentioned, the first one becomes available right after the installation of the GIF CLI. The other commands would be ready for use just after you have created a user and a product. The **console mode** enables you to input methods one by one directly into the command line, and the **execute mode** allows to write sequence of commands in a particular file and then execute this file in the CLI.

In the console mode, as well as in the execute mode, you interact with your product directly — the CLI executes commands on behalf of the current product (you can see the name of your current product in such a line: `GIF :: ` **your product name** ` $ [COMMAND]`). As soon as you have several products, you can switch between them getting to the necessary product by the **gifcli product:select** command. To execute the gifcli commands, you should first exit from the console (or execute) mode by running `Ctrl+C twice`.

In case of doubt, you can always refer to the `gifcli help [COMMAND]` command in the **CLI mode**. There, you can find a list of currently available commands. The execution of this command looks like that.

```
gifcli help
$ gifcli help
gifcli ======

VERSION
@etherisc/gifcli/1.0.5 win32-x64 node-v10.15.3

USAGE
  $ gifcli [COMMAND]

COMMANDS
  artifact  manage artifacts
  console   run console mode
  exec      execute file
  help      display help for gifcli
  product   manage products
  update    update the gifcli CLI
  user      manage user
```

In the **console mode** (appears by the `gifcli console` command), you can input the `gif.help()` method into the command line. This will show you methods available for the user. Here is an example.

```
$ gifcli console
GIF :: `your product name` > gif.help()
gif.info                Information about the product
gif.help                Get information about the command
gif.artifact.get        Get artifact for contract
gif.contract.send       Send transaction to contract
gif.contract.call       Call contract
gif.customer.create     Create customer
gif.customer.getById    Get customer by id
gif.customer.list       Get all customers
gif.bp.create           Create new business process
gif.bp.getByKey         Get business process by key identifier
gif.bp.getById          Get business process by id identifier
gif.bp.list             Get all business processes
gif.application.getById Get application by id
gif.application.list    Get all applications
gif.policy.getById      Get policy by id
gif.policy.list         Get all policies
gif.claim.getById       Get claim by id
gif.claim.list          Get all claims
```

(continues on next page)

(continued from previous page)

```
gif.payout.getById      Get payout by id
gif.payout.list         Get all payouts
gif.product.get         Get product instance
```

To learn more about each of the above-mentioned methods use the `gif.help('...')` method. For instance, `gif.help('product.get')`.

20.3 A step-by-step guide

Here, we present basic steps that demonstrate you how to start working with the GIF and its command line interface — from registering a user to making a payout by your product. In addition, you will find other available extension commands in the General description section. This will help you to execute all the necessary processes.

We will go through all the steps necessary to interact with the GIF CLI on the basis of our default sample contracts. You can create your own products (contracts) using whether required basic methods or other methods and business logic developed and implemented by yourself.

Start working with the GIF CLI directly from running command line on your computer:

1. First, you need to input the `gifcli user:register` command in the CLI. After that, fill in the fields with your first name, last name, and e-mail address, as well as create a password.

```
$ gifcli user:register

Firstname: John
Lastname: Johnson
Email: john.johnson@mail.com
Password: *******
Repeat password: *******

User registered
```

After this, a user will be created.

Attention: Be careful with the `gifcli user:logout` command. You need to use it only in case you want to make a new user instead of the previous one. This command clears up the **.gifconfig.json** file in your home directory. After executing the command, you will not be able to access your previously created products and customers. The password, as well as email address, first and last names for a new user should be different to that of the previous one.

In case you would need to exploit your previous user, you should backup the **.gifconfig.json** file with the required credentials and then use it instead of the .gifconfig.json file with the data of your current one.

2. Then, obviously, you would like to start dealing with your products. If you want to create a product and become a product owner, use the `gifcli product:create` command. There, you can specify a product name. This name at the same time is registered at the RabbitMQ message broker.

```
$ gifcli product:create

Product name: one

Product created
```

3. After that, you should create a directory by the `mkdir` command (`mkdir my-first-product` in our example) for your product (the "one" for our case), and go to it (using the `cd ./my-first-product` command).

```
$ mkdir my-first-product

Directory: /Users/username

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         3/26/2019  16:30 PM                my-first-product


PS ./Users/username> cd my-first-product
PS ./Users/username/my-first-product>
```

4. Then, run the `npm init -y` command.

```
$ npm init -y

Wrote to ./my-first-product/package.json:

{
  "name": "my-first-product",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo /"Error: no test specified/" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

5. After that, you should use the `npm install truffle openzeppelin-solidity truffle-hdwallet-provider @etherisc/gif` command. A successful execution should end up with the following lines.

```
$ npm install truffle openzeppelin-solidity truffle-hdwallet-provider @etherisc/gif

...
+ truffle@5.0.10
+ truffle-hdwallet-provider@1.0.6
+ openzeppelin-solidity@2.2.0
+ @etherisc/gif@1.0.0
added 892 packages from 1374 contributors and audited 3757 packages in 79.988s
found 0 vulnerabilities
```

6. The next step is to execute the `./node_modules/.bin/truffle init` command:

```
$ ./node_modules/.bin/truffle init

> Preparing to download
> Downloading
> Cleaning up temporary files
> Setting up box
```

(continues on next page)

(continued from previous page)

```
Unbox successful. Sweet!


Commands:

  Compile:        truffle compile
  Migrate:        truffle migrate
  Test contracts: truffle test
```

7. Now you need to create your product's smart contract and deploy it. In our example, we need to take the following steps:

7a. First, we should replace the content of the **truffle-config.js** file in the "my-first-product" directory on our computer with the following one:

```
const HDWalletProvider = require('truffle-hdwallet-provider');


module.exports = {
  migrations_directory: process.env.MIGRATIONS_DIRECTORY || './migrations',
  contracts_build_directory: process.env.CONTRACTS_BUILD_DIRECTORY || './build',

  networks: {
    development: {
      host: 'localhost',
      port: 8545,
      network_id: 7777,
      gas: 6600000,
      gasPrice: 10 * (10 ** 9),
      websockets: true,
    },

    coverage: {
      host: 'localhost',
      network_id: '*',
      port: 8555, // the same port as in .solcover.js.
      gas: 0xfffffffffff,
      gasPrice: 0x01,
    },

    kovan: {
      // MNEMONIC: BIP39 mnemonic, e.g. https://iancoleman.io/bip39/#english
      // HTTP_PRODIVER: e.g. https://kovan.infura.io/<your-token>
      provider: () => new HDWalletProvider(process.env.MNEMONIC, process.env.HTTP_
↪PROVIDER),
      network_id: 42,
      confirmation: 2,
      timeoutBlocks: 200,
      skipDryRun: true,
      gas: 6600000,
      gasPrice: 10 * (10 ** 9),
    },

    rinkeby: {
      // MNEMONIC: BIP39 mnemonic, e.g. https://iancoleman.io/bip39/#english
      // HTTP_PRODIVER: e.g. https://rinkeby.infura.io/<your-token>
      provider: () => new HDWalletProvider(process.env.MNEMONIC, process.env.HTTP_
↪PROVIDER),
```

(continues on next page)

(continued from previous page)

```
      network_id: 4,
      confirmation: 2,
      timeoutBlocks: 200,
      skipDryRun: true,
      gas: 6600000,
      gasPrice: 10 * (10 ** 9),
    },
  },

  mocha: {
    timeout: 20000,
    useColors: true,
  },

  compilers: {
    solc: {
      version: '0.5.2',
      settings: {
        optimizer: {
          enabled: true,
          runs: 200,
        },
        evmVersion: 'byzantium', // -> constantinople
      },
    },
  },
};
```

7b. Then, we can create our product contract taking the following one as an example. We create a **SimpleProduct.sol** file in the "contracts" folder in our "my-first-product" directory with the content below.

```
pragma solidity 0.5.2;

import "@etherisc/gif/contracts/Product.sol";


contract SimpleProduct is Product {

    event NewApplication(uint256 applicationId);
    event NewPolicy(uint256 policyId);
    event ApplicationDeclined(uint256 applicationId);
    event NewClaim(uint256 policyId, uint256 claimId);
    event NewPayout(uint256 claimId, uint256 payoutId, uint256 payoutAmount);
    event PolicyExpired(uint256 policyId);
    event PayoutConfirmation(uint256 payoutId, uint256 amount);

    bytes32 public constant NAME = "SimpleProduct";
    bytes32 public constant POLICY_FLOW = "PolicyFlowDefault";

    constructor(address _productController)
        public
        Product(_productController, NAME, POLICY_FLOW)
    {}

    function applyForPolicy(
        bytes32 _bpExternalKey,
```

(continues on next page)

(continued from previous page)

```
        uint256 _premium,
        bytes32 _currency,
        uint256[] calldata _payoutOptions
    ) external onlySandbox {
        uint256 applicationId = _newApplication(
            _bpExternalKey,
            _premium,
            _currency,
            _payoutOptions
        );
        emit NewApplication(applicationId);
    }

    function underwriteApplication(uint256 _applicationId) external onlySandbox {
        uint256 policyId = _underwrite(_applicationId);
        emit NewPolicy(policyId);
    }

    function declineApplication(uint256 _applicationId) external onlySandbox {
        _decline(_applicationId);
        emit ApplicationDeclined(_applicationId);
    }

    function newClaim(uint256 _policyId) external onlySandbox {
        uint256 claimId = _newClaim(_policyId);
        emit NewClaim(_policyId, claimId);
    }

    function confirmClaim(uint256 _claimId, uint256 _payoutAmount) external␣
↪onlySandbox {
        uint256 payoutId = _confirmClaim(_claimId, _payoutAmount);
        emit NewPayout(_claimId, payoutId, _payoutAmount);
    }

    function expire(uint256 _policyId) external onlySandbox {
        _expire(_policyId);
        emit PolicyExpired(_policyId);
    }

    function confirmPayout(uint256 _payoutId, uint256 _amount) external onlySandbox {
        _payout(_payoutId, _amount);
        emit PayoutConfirmation(_payoutId, _amount);
    }

    function getQuote(uint256 _sum) external view returns (uint256 _premium) {
        require(_sum > 0);
        _premium = _sum.div(20);
    }
}
```

7c. Now we can proceed with making a deployment migration. Like in the previous step, we use the following sample for migration. We create a **2_deploy_SimpleProduct.js** file in the "migrations" folder in our "my-first-product" directory and paste the text of the sample contract here.

```
const SimpleProduct = artifacts.require('SimpleProduct');

```

(continues on next page)

(continued from previous page)

```
const GIF_PRODUCT_SERVICE_CONTRACT = '0x0';

module.exports = deployer => deployer.deploy(SimpleProduct, GIF_PRODUCT_SERVICE_
↪CONTRACT);
```

7d. After that, we need to set the value of the constant GIF_PRODUCT_SERVICE_CONTRACT to **0x6520354fa128cc6483B9662548A597f7FcB7a687** — the address of the deployed smart contract. It should be placed in the **GIF_PRODUCT_SERVICE_CONTRACT** line of the **2_deploy_SimpleProduct.js** file. For your convenience we list addresses of the core smart contracts at the end of this manual.

7e. To finish with this step, we need to add the `"compile": "truffle compile", "migrate": "truffle migrate"`, commands to the "scripts" section of the **package.json** file in the my-first-product directory.

8. Then, you should execute the `npm run compile` command.

```
$ npm run compile

> my-first-product@1.0.0 compile ./my-first-product
> truffle compile

Compiling your contracts...
===========================
> Compiling @etherisc/gif/contracts/Product.sol
> Compiling @etherisc/gif/contracts/services/IProductService.sol
> Compiling @etherisc/gif/contracts/shared/RBAC.sol
> Compiling ./contracts/Migrations.sol
> Compiling ./contracts/SimpleProduct.sol
> Compiling openzeppelin-solidity/contracts/math/SafeMath.sol
> Compiling openzeppelin-solidity/contracts/ownership/Ownable.sol

    ...

> Artifacts written to ./my-first-product/build
> Compiled successfully using:
    -solc: 0.5.2+commit.1df8f40c.Emscripten.clang
```

Note: Before running the next command, you should create a mnemonic here.

It is also required to fund your account with some test ETH on Rinkeby test network.

9. After that, you can continue with the migration using the `HTTP_PROVIDER="https://rinkeby.infura.io/v3/KEY" MNEMONIC="mnemonic" npm run migrate -- --network rinkeby` command. In the command text, instead of the word "KEY" paste your infura key and, instead of the word "mnemonic", input here the mnemonic, created in the previous step. To execute the command, you need to create an account at Infura (if you haven't yet) and paste the key from your account into the mentioned space in the command.

Note: Operating on the Ethereum environment, all the transactions consume "gas". You can face a warning message like this: *"Error: ** Deployment Failed *** "Migrations" – The contract code couldn't be stored, please check your gas limit."* In this case, you need to top up your account with some ETH and execute the command again.

```
$ HTTP_PROVIDER="https://rinkeby.infura.io/v3/paste your infura key here" MNEMONIC=".
↪." npm run migrate -- --network rinkeby
```

(continues on next page)

(continued from previous page)

```
> my-first-product-2@1.0.0 migrate ./my-first-product-2
> truffle migrate "--network" "rinkeby"

Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.

Starting migrations...
======================
> Network name:    'rinkeby'
> Network id:      4
> Block gas limit: 0x6acec5

1_initial_migration.js
======================
    Deploying 'Migrations'
    ----------------------
    > transaction hash:    ␣
↪0x9313aeb218ae3b1174fd365c1ae921cc978e961d36b5616558a1003032d661ea
    > Blocks: 0            Seconds: 8
    > contract address:    0xACE701BfFd5c14EEFA565D1651f83D9ED9bd5e48
    > account:             0x1DdCFb13eb5109E53763677E04BC9FB8fAb40D4b
    > balance:             xx.xxxxxxxx
    > gas used:            221171
    > gas price:           10 gwei
    > value sent:          0 ETH
    > total cost:          0.00xxxxxx ETH

    > Saving migration to chain.
    > Saving artifacts
    -------------------------------------
    > Total cost:          0.00xxxxxx ETH

2_deploy_SimpleProduct.js
=========================
    Deploying 'SimpleProduct'
    -------------------------
    > transaction hash:␣
↪0xcd7bfec51303bb66639bd90cf6db2c40f2e875d744e97b35c41102c3e5a03170
...
    > Saving migration to chain.
    > Saving artifacts
    -------------------------------------
    > Total cost:       0.0xxxxxxx ETH

Summary
=======
> Total deployments: 2
> Final cost:        0.0xxxxxxx ETH
```

10. Now you should input the `gifcli artifact:send --file {PATH_TO_CONTRACT_ARTIFACT} --network rinkeby` command, where PATH_TO_CONTRACT_ARTIFACT stands for a path to the **.json** file with artifacts for the contract. In our example, this part of the command looks like that: gifcli artifact:send –file **./my-first-product/build/SimpleProduct.json** –network rinkeby. You can find the SimpleProduct.json file (from our example) in the "build" folder of the "my-first-product" directory. It will appear on your computer after you execute the npm run compile command. The response for the successful execution of the command

will be the following:

```
$ gifcli artifact:send --file ./my-first-product/build/SimpleProduct.json --network␣
↪rinkeby


{ result: 'Artifact saved',
  product: 'one',
  contractName: 'SimpleProduct',
  address: '0xF8450d6b6be91C861d7ef2a91B5e2695aeAf335a',
  network: 'rinkeby',
  version: '1.0.5' }
```

Now we've successfully created a product smart contract.

11. As we are already in the "my-first-product" directory, we can run the console mode to proceed interacting with our product "one". We execute the `gifcli console` command.

```
$ gifcli console


GIF :: one >
```

12. By executing the `gif.product.get()` method, the CLI demonstrates the artifacts of the current product as they are registered on the GIF (compare the "name" of the product "SimpleProduct" instead of "one" at RabbitMQ).

```
$ gif.product.get()

{ key: 18,
    created: '2019-03-26T16:47:07.176Z',
    updated: '2019-03-26T16:49:21.580Z',
    productId: 21,
    name: 'SimpleProduct',
    addr: '0xf8450d6b6be91c861d7ef2a91b5e2695aeaf335a',
    policyFlow: 'PolicyFlowDefault',
    release: 0,
    policyToken: '0x0000000000000000000000000000000000000000',
    approved: true,
    paused: false,
  productOwner: '0x0000000000000000000000000000000000000000' }
```

13. Now, you can proceed with creating a customer. Here, the `gif.customer.create({ firstname: '...', lastname: '...', email: '...@....com'})` method will help:

```
$ gif.customer.create({firstname:'Dear',lastname:'Customer',email:'dear.customer@mail.
↪com', age: 33})

{ customerId:
    '5efaf976b1fb4fe0be9b0d68e833c469757c2749863c33b77ce907e6f3bc8cee'
}
```

You can add other necessary arguments about your customers, e.g., the age (as in our example), etc. in the text of the method.

14. Then, using the `gif.customer.getById("insert customer ID here")` method, you can receive specific data related to a certain customer by a customer ID. From the previous step, you will receive the output with the customer's first name, last name, e-mail address, and age.

```
$ gif.customer.getById(
↪"5efaf976b1fb4fe0be9b0d68e833c469757c2749863c33b77ce907e6f3bc8cee")


{ id:
   '5efaf976b1fb4fe0be9b0d68e833c469757c2749863c33b77ce907e6f3bc8cee',
  firstname: 'Dear',
  lastname: 'Customer',
  email: 'dear.customer@mail.com',
  created: '2019-03-26T16:49:59.059Z',
  updated: '2019-03-26T16:49:59.059Z',
  age: '33' }
```

15. You can also input the `gif.customer.list()` method. Like other methods related to the "lists" of particular issues, this method results in the list of customers of your current productt. In our example, we have only one customer.

```
$ gif.customer.list()

[ { id:
     '5efaf976b1fb4fe0be9b0d68e833c469757c2749863c33b77ce907e6f3bc8cee',
    firstname: 'Dear',
    lastname: 'Customer',
    email: 'dear.customer@mail.com',
    created: '2019-03-26T16:50:20.059Z',
    age: '33' } ]
```

16. The (bp - business process) `gif.bp.create({ manager: 'customer_name' or customerId: '...' or both as well})` method returns **bpExternalKey** required for **applyForPolicy** in a contract to link policy flow objects with an external database. This very method is used to connect a customer (a customer name or an ID is required) and all his/her data (optional inputs are provided in the {} brackets) important for the business process. The method can also look like that: `gif.bp.create({ manager: 'Dear', customer: { firstname: 'Dear', lastname: 'Customer', email: 'dear.customer@mail.com' } })`.

```
$ gif.bp.create({manager: 'Dear', customerId:
↪'5efaf976b1fb4fe0be9b0d68e833c469757c2749863c33b77ce907e6f3bc8cee'})

{ bpExternalKey: 'b5aaa0546e264f39a92baea697f53be5',
    customerId:
    '5efaf976b1fb4fe0be9b0d68e833c469757c2749863c33b77ce907e6f3bc8cee' }
```

17. You can also make a list of your business processes by using the `gif.bp.list()` method:

```
$ gif.bp.list()

[ { key: 'b5aaa0546e264f39a92baea697f53be5',
    created: '2019-03-26T16:50:53.855Z',
    customerId: '5efaf976b1fb4fe0be9b0d68e833c469757c2749863c33b77ce907e6f3bc8cee',
    contractKey: null,
    productId: 1,
    id: 1,
    applicationId: 1,
    policyId: 0,
    hasPolicy: false,
    hasApplication: true,
    tokenContract: '0x0000000000000000000000000000000000000000',
```

(continues on next page)

(continued from previous page)

```
    tokenId: -1,
    registryContract: '0x0000000000000000000000000000000000000000',
    release: 0,
    state: 0,
    stateMessage: '',
    bpExternalKey: 'b5aaa0546e264f39a92baea697f53be5',
    createdAt: 1553619141,
    updatedAt: 1553619141,
    manager: 'Dear' } ]
```

You can use the `gif.bp.getById()` method as well as the `gif.bp.getByKey()` method to read a part of commonly shared data (metadata) of a particular business process. Metadata is contained both in the product's contract and in the product's database. The `gif.bp.getById()` method uses the ID of a business process in the product's contract (as you see the "id" line from above). The `gif.bp.getByKey()` method, that requires to input a unique key of the business process — an identifier in your product database (the "key" line in the example above). The same key is used when you apply for a policy (the 20th step in our example).

18. One more step is to execute the `gif.contract.call("ProductName", "getQuote", [e.g. sum of payout by the contract])` method. In our case, this method calls the method "getQuote", which sets the premium for our contract. As you can see from the sample, the premium is about 5% of the payout. By the gif.contract.call method, you can read any data of your product's contract or get a result of an executed function. This method does not change the state of the contract and does not make a transaction on the blockchain.

Here is the data from our sample:

```
$ gif.contract.call('SimpleProduct','getQuote',[200])

{ _premium: '10' }
```

19. The `gif.contract.send("ProductName", "applyForPolicy", ['ExternalKey', sum of payout, 'currency', [sum of premium]])` method can be used for different purposes. In place of the "ExternalKey" text in the method you need to input the key given at the 18th step. As you can see from our example, it helps to apply for a policy but it is also used for underwriting applications, as well as creating and confirming claims. We will do this in a few steps. By this method, you can send transactions to the contract's method. As a result, the state of the contract is changed and a transaction on the blockchain is made.

```
$ gif.contract.send('SimpleProduct', 'applyForPolicy', [
↪'b5aaa0546e264f39a92baea697f53be5', 200,'EUR',[10]])

{ blockHash:
    '0xd21fc587a9dfa50b65e08267b6d4f43d1b68fe7a1dc5a3330c0d0e9bcaae9773',
    blockNumber: 4139120,
    contractAddress: null,
    cumulativeGasUsed: 437007,
    from: '0x0e48196f6e7c8df0006bb7e7122e1e9f5ef46d6a',
    gasUsed: 351892,
    logsBloom:
...
    returnValues: [Object],
    event: 'NewApplication',
    signature:
    '0x0ff47c4a3dc48719ecfd1876116e80d7d76ec7cb67248ae49449f9104747af29',
    raw: [Object] } } }
```

20. To look through applications of your product, you can execute the `gif.application.list()` method.

```
$ gif.application.list()

{ key: 'e0937732cb1749c7aa81795393c7d3d2',
    created: '2019-03-26T16:52:22.019Z',
    contractKey: null,
    productId: 21,
    id: 13,
    metadataId: 13,
    premium: 200,
    currency: 'EUR',
    payoutOptions: '["10"]',
    state: 0,
    stateMessage: '',
    createdAt: 1553619141,
    updatedAt: 1553619141 }
```

21. After creating applications, you can get data of a particular application by its ID using the `gif.application.getById(ID number of an application)` method. In our example, we got the ID number of the application (see the previous step). Its ID = 13. Then, we place it in brackets.

```
$ gif.application.getById(13)
{ key: 'e0937732cb1749c7aa81795393c7d3d2',
    created: '2019-03-26T16:52:22.019Z',
    updated: '2019-03-26T16:52:22.019Z',
    contractKey: null,
    productId: 21,
    id: 13,
    metadataId: 13,
    premium: 200,
    currency: 'EUR',
    payoutOptions: '["10"]',
    state: 0,
    stateMessage: '',
    createdAt: 1553619141,
    updatedAt: 1553619141 }
```

22. With the `gif.contract.send("ProductName", "underwriteApplication", [application ID])` method, you can underwrite a certain application.

```
$ gif.contract.send('SimpleProduct','underwriteApplication',[13])

{ blockHash:
    '0x1d580e979734106c2b46eccb8f9b2522e342e58b6666104bbcbcd697fceb9152',
    blockNumber: 4139193,
    contractAddress: null,
    cumulativeGasUsed: 1884903,
    from: '0x0e48196f6e7c8df0006bb7e7122e1e9f5ef46d6a',
    gasUsed: 235013,
    logsBloom:
...
    returnValues: [Object],
    event: 'NewPolicy',
    signature:
     '0x174c94eb4ef02e690e5bd01790c284af662a414381f1c631bf388a8850a5db13',
    raw: [Object] } } }
```

23. The `gif.policy.list()` method enables you to get a list of policies:

```
$ gif.policy.list()

[ { key: '30762af6af2d4267afc72f1714b1eb52',
    created: '2019-03-26T16:56:06.630Z',
    contractKey: null,
    productId: 21,
    id: 3,
    metadataId: 13,
    state: 0,
    stateMessage: '',
    createdAt: 1553619366,
    updatedAt: 1553619366 } ]
```

24. You can also receive specific data related to a certain policy by a policy ID using the `gif.policy.getById(ID number of a policy)` method. As you can see from the previous step, the ID number of the policy is 3:

```
$ gif.policy.getById(3)

{ key: '30762af6af2d4267afc72f1714b1eb52',
    created: '2019-03-26T16:56:06.630Z',
    updated: '2019-03-26T16:56:06.630Z',
    contractKey: null,
    productId: 21,
    id: 3,
    metadataId: 13,
    state: 0,
    stateMessage: '',
    createdAt: 1553619366,
    updatedAt: 1553619366 }
```

25. To create a claim use the `gif.contract.send("ProductName", "newClaim", [ID number of a policy])` method:

```
$ gif.contract.send('SimpleProduct','newClaim',[3])

{ blockHash:
    '0x30da89398de8083a250f031af72fbfc27fa64cfd2bb1a88d3963e5e151fc9582',
    blockNumber: 4139333,
    contractAddress: null,
    cumulativeGasUsed: 1017872,
    from: '0x0e48196f6e7c8df0006bb7e7122e1e9f5ef46d6a',
    gasUsed: 185825,
    logsBloom:
...
    returnValues: [Object],
    event: 'NewClaim',
    signature: '0xcb97bbaee7e6aa4ae5d3a69e8a66d1f15b6d4ebb585e5f8f26eaab86c49ae665',
    raw: [Object] } } }
```

26. To list claims, you can use the `gif.claim.list()` method.

```
$ gif.claim.list()

[ { key: '651328ab2b764b52b4ba696a2f791ab9',
    created: '2019-03-26T16:58:21.538Z',
    contractKey: null,
```

(continues on next page)

(continued from previous page)

```
    productId: 21,
    id: 3,
    metadataId: 13,
    data: '',
    state: 0,
    stateMessage: '',
    createdAt: 1553619501,
    updatedAt: 1553619501 } ]
```

27. As you have already seen earlier, the same behavior, can be achieved by the `gif.claim.getById(ID number of a policy)` method:

```
$ gif.claim.getById(3)

{ key: '651328ab2b764b52b4ba696a2f791ab9',
    created: '2019-03-26T16:58:21.538Z',
    updated: '2019-03-26T16:58:21.538Z',
    contractKey: null,
    productId: 21,
    id: 3,
    metadataId: 13,
    data: '',
    state: 0,
    stateMessage: '',
    createdAt: 1553619501,
    updatedAt: 1553619501 }
```

28. You can provide a confirmation of a claim by the `gif.contract.send("ProductName", "confirmClaim", [ ID number of a claim, sum of payout])` method:

```
$ gif.contract.send('SimpleProduct','confirmClaim',[3,100])

{ blockHash:
    '0x129315bc294f7444c90e84c73ef81e2629c5939dd62bac1d23d15b4538ee809b',
    blockNumber: 4139427,
    contractAddress: null,
    cumulativeGasUsed: 1932170,
    from: '0x0e48196f6e7c8df0006bb7e7122e1e9f5ef46d6a',
    gasUsed: 283098,
    logsBloom:
...
    returnValues: [Object],
    event: 'NewPayout',
    signature:
     '0xf2891b2b2049ac20caebda64567475aab2ad4d50f1faa089cda0d70aaa1fb3f2',
    raw: [Object] } } }
```

29. To make a payout, you need to confirm it using the `gif.contract.send("ProductName", "confirmPayout", [3, 100])` method:

```
$ gif.contract.send('SimpleProduct','confirmPayout',[3,100])

{ blockHash:
        '0x80c925e2f6e4eea469d5c6ab33f70e8291c1a25c3e56478155423e15bf917ae8',
    blockNumber: 4139446,
    contractAddress: null,
```

(continues on next page)

(continued from previous page)

```
    cumulativeGasUsed: 110977,
    from: '0x0e48196f6e7c8df0006bb7e7122e1e9f5ef46d6a',
    gasUsed: 110977,
    logsBloom:
...
    returnValues: [Object],
    event: 'PayoutConfirmation',
    signature:
     '0x0ad736fbe1571767f34d1bfa0cebbaf3c0424d30452fdc42167509bb5060ad82',
    raw: [Object] } } }
```

30. Finally, you can see a list of payouts of your product by executing the `gif.payout.list()` method:

```
$ gif.payout.list()

[ { key: 'de2c53312e72425ab913c2e760ec5efd',
    created: '2019-03-26T17:00:06.647Z',
    contractKey: null,
    productId: 21,
    id: 3,
    metadataId: 13,
    claimId: 3,
    expectedAmount: 0,
    actualAmount: 100,
    state: 1,
    stateMessage: '',
    createdAt: 1553619606,
    updatedAt: 1553619741 } ]
```

You can also use the `gif.payout.getById(ID number of a payout)` method when you want to receive specific data related to a certain payout by its ID.

With these basic steps, you can start using the Generic Insurance Framework.

Note: For your convenience, we also provide the addresses of the smart contracts, deployed in the blockchain test network Rinkeby. These contracts enable the necessary functionality for the GIF CLI. In particular, you should use the ProductService contract to deploy your own product's contract.

Network: rinkeby (id: 4)

InstanceOperatorService: 0x39F7826D3796BC4a2Eb2F0B8fF3799f30D02CBf5

License: 0x9Fb57F1C2291395a0F654A03C2053309a9928d39

LicenseController: 0xd5337b57c636EEF4Aa5C78625816715AE945f81A

Migrations: 0xa38910BB20F790aaC9F03C498b5bb61382a0dCF7

OracleOwnerService: 0xcD8438bA7580139e5df05067cd868ea31A7eb9E8

OracleService: 0x5F4a25c03054f8072Bd10C6afc515E5C4a146f27

Policy: 0x10154588296B531B880ca669E0807A3dA78F2Ae8

PolicyController: 0x1fCda1D5efBCC82d24e0438C618DDCe7383827AB

PolicyFlowDefault: 0x04EC0D88D70713ba304ad54c6f22200ea93dDd57

ProductService: 0x6520354fa128cc6483B9662548A597f7FcB7a687

Query: 0x2936555290B17062e3472CF3a5A3DE3B84A01515

QueryController: 0xAd517b5da0b62DfF56ac57d612f4bEf0eA1e1b78

Registry: 0x5E78A5a3ffd005761B501D6264cEcD87E2d331B0

RegistryController: 0x4Bf8b2622a1b5B6b2865087323E6C518a3946AbA

CHAPTER 21

GIF Tutorial: How to build an insurance product on GIF

This is the first part of the GIF Tutorial aimed at explaining how to build products on top of Etherisc's Generic Insurance Framework. In this article, we will go through all the steps of creating your very first product on top of GIF. So put your fingers on the keyboard and let's get moving.

First of all, what is "GIF"? The acronym stands for "Generic Insurance Framework". Its main goal is to provide a simple and clean interface to build decentralized insurance products. Core contracts are aimed to be deployed on-chain.

A product contract acts as a client and utilizes generic methods which are important parts of every policy lifecycle. From this point of view, a product's business logic could be defined in a single smart contract and all the hard work is delegated to the GIF.

21.1 Some insurance terminology

Before we start, let's define some insurance-specific terminology.

- An **insurance company** is a legal body that is qualified in accepting risks against a premium. In most countries, insurance companies need a license. The insurance company is not necessarily the entity which runs the technical process; it can also be outsourced to a service provider. Most of the other functions in the value chain can also be taken by other parties (e.g. distribution, claims management, etc.). The only function which cannot be delegated is the actual transfer of the risk.
- An **application** is a formal request of the customer for an insurance policy. It is a signal from the customer to the insurance company: "I ask for insurance cover".
- The application is then checked by the insurance company and either accepted or declined. The process of accepting an application is called **underwriting**. The person who performs this action is an **"underwriter"**. By underwriting, the insurance company commits itself to transfer the risk from the customer to the insurance company. This fact is documented in a contract. This contract is called a **policy**. A contract is binding for both parties: The customer is committed to paying the premium and the insurance company is committed to covering a loss in case an insured event occurs.
- If an insured event occurs, the customer typically files a **claim** — a step which can be omitted in case of parametric insurance, where a data-driven oracle takes the responsibility to file the claim automatically.

- The claim is then checked by the insurance company and if the insurance company decides that the claim is valid then a **payout** is triggered (e.g. the insurance company will deny the claim if the premium has not been paid). Again, in a parametric case, the insurance company can delegate the decision to a rule-based engine or to an external independent oracle.
- Payouts can be done in one or more parts.

GIF provides generic functions for this lifecycle and a generic workflow which controls the sequence of states. In the next section, we will describe these functions and how they work in detail. Every function can be assigned to a **role**, which can be defined by the product designer. Typical roles are e.g. **underwriter**, **claims manager**, **application manager**, **bookkeeper** but of course the product designer is not limited to these.

21.2 Generic Lifecycle Functions

Here is the list of magic methods available for every product:

- **_register** (each product needs to be registered in order to get access to the GIF functionality)
- **_newApplication** (to put application data into the storage)
- **_underwrite** (to underwrite an application and create a new policy)
- **_decline** (to decline an application)
- **_newClaim** (to create a new claim for policy)
- **_confirmClaim** (to confirm a claim and create a payout object)
- **_declineClaim** (to decline a claim)
- **_payout** (to confirm an off-chain payout)
- **_request** (to request data or action from the oracle)
- **_createRole** (to create roles for actors)
- **_addRoleToAccount** (to assign roles to actors)

The names of these methods start with an **underscore** to highlight the fact that these are internal methods and you can override them in your product. For example, you are free to have the **newApplication** method in your contract and use **_newApplication** in it as well.

CHAPTER 22

The idea for a new product

In this article, we will build a very simple product. The main goal is to provide you a path through the process of building your own product. In this case, we will build an insurance product for an electronic store, which sells screens for laptops.

Let's assume that a screen is unrepairable. If it's broken (and of course only if it is an insurance case), a payout should be equal to the price of the screen.

The pricing model is very simple. Let's assume that a policy premium should be 10% of the laptop screen price.

The policy expiration period is 1 year.

The following actors will take part in the process: an application manager, an underwriter, a claims manager and a bookkeeper.

Here are the steps of the product life cycle:

- The customer applies for a policy.
- The application manager creates an application for the policy on behalf of the customer.
- The underwriter underwrites or declines the application.
- If the application is underwritten, a new policy is issued.
- If something wrong happens with the laptop screen, the customer can create a claim.
- The claims manager confirms or declines this claim.
- If the claim is confirmed, a new payout is created.
- Then, the bookkeeper should pay to the customer and confirm that the payout has been executed.
- After the expiration period ends, the policy should be expired.

CHAPTER 23

Setting up a development environment

23.1 Prerequisites:

Node.js should be installed on your computer. If you already have it, then check its version. It shouldn't be too old. Use at least version 8.12.0 but below 12 (to see if Node.js is installed, open your terminal and type "node -v", this should show you a current version).

GIF consists of 3 parts:

- **Core smart contracts**
- **Microservices** represent a utility layer. They are available to provide useful functionality for product builders (off-chain data storage, watch Ethereum events, interact with smart contracts (call data and send transactions), send emails, telegram notifications, etc.). More detailed information about available microservices will be published in the next articles.
- **CLI tool (gifcli)**. A command-line interface to interact with the sandbox environment.

23.2 Registration

In order to create products, you have to be registered as a product owner. Use the "./bin/run user:register" command to register in the sandbox. It will require to insert some information: first name, last name, email, and password.

Prepare environment variables, like this and put them to your shell env file (e.g. .bashrc, .zshrc, etc.):

```
export GIF_API_HOST="http://localhost"
export GIF_AMQP_HOST="localhost"
cd ./cli
./bin/run user:register
```

After the registration, the product owner is permitted to create a product. Let's create the first one.

```
./bin/run product:create
```

That's it. Now we can start creating a smart contract for the product.

CHAPTER 24

Coding part

24.1 Configure a Truffle project

Now we are ready to build a product. Create a new directory for it.

```
mkdir estore-insurance
cd estore-insurance
```

Install required dependencies:

- Truffle — a development environment for smart contracts, truffle-hdwallet-provider — we use it to sign transactions during the deployment with a mnemonic account.
- @etherisc/gif — GIF core smart contracts — we need the Product.sol contract to inherit from it.
- openzeppelin-solidity — a library for smart contract development — we will use Ownerable.sol from it.

The last command here will bootstrap a typical Truffle project for us.

```
npm init -y
npm install truffle truffle-hdwallet-provider @etherisc/gif openzeppelin-solidity
./node_modules/.bin/truffle init
```

Edit truffle-config.json

```
    module.exports = {
 networks: {
    development: {
      host: 'localhost',
      port: 8545,
      network_id: 7777,
      gas: 6600000,
      gasPrice: 10 * (10 ** 9),
      websockets: true,
    },
```

(continues on next page)

(continued from previous page)

```
  },
  compilers: {
    solc: {
      version: "0.5.2",
      settings: {
        optimizer: {
          enabled: true,
          runs: 200,
        },
      },
    },
  },
};
```

Then add “compile” and “migrate” commands to the scripts section in the package.json file. The first one will be used to compile smart contracts. The second one is needed to deploy them to the relevant network (we will deploy to local blockchain).

```
...

"scripts": {
  "compile": "truffle compile",
  "migrate": "truffle migrate"
},

...
```

File: ./package.json

CHAPTER 25

Create a smart contract

Create a new file "EStoreInsurance.sol" in the contracts directory with the following content:

```
pragma solidity 0.5.2;
import "<PATH_TO_GIF>/core/gif-contracts/contracts/Product.sol";
contract EStoreInsurance is Product {
  bytes32 public constant NAME = "EStoreInsurance";
  bytes32 public constant POLICY_FLOW = "PolicyFlowDefault";
  constructor(address _productService)
  public
Product(_productService, NAME, POLICY_FLOW) {}
}
```

File: ./contracts/EStoreInsurance.sol

First of all, we imported the Product contract from the GIF package and inherited from it. From now, if we deploy this contract, our product will become a client for GIF contracts through the ProductService contract (we pass its address as a constructor argument). Look at these magic constants: NAME and POLICY_FLOW. NAME is a name for your product. The only restriction here is that it should fit the 32-bytes length. The POLICY_FLOW constant should be defined to choose the core contract, which will represent the policy lifecycle for this product. Right now, we have only this contract — "PolicyFlowDefault". And we have plans to create different versions of it for different policy lifecycles.

In the constructor, we call the Product's constructor and pass the address of ProductService to it (this address is published in GIF repository on Github). The Product's constructor will call the "register" function and your product contract will be proposed as a product (It should be approved by the administrator before this product starts to accept new applications. But don't worry, in the sandbox, the product will be approved automatically).

Then let's define risk. So far, as each product has its own set of fields we keep such data on the product contract side. In this particular case, risk contains brand, model, and year fields.

```
...

bytes32 public constant POLICY_FLOW = "PolicyFlowDefault";
struct Risk {
```

(continues on next page)

(continued from previous page)

```
  bytes32 brand;
  bytes32 model;
  uint256 year;
}
mapping(bytes32 => Risk) public risks;
constructor(address _productService)

...
```

File: ./contracts/EStoreInsurance.sol

Add a public function to calculate a premium based on a certain price. This function could be used to provide quotas to applicants.

```
...

constructor(address _productService) public Product(_productService, NAME, POLICY_
↪FLOW) {}
function getQuote(uint256 _price) public pure returns (uint256 _premium) {
  require(_price > 0, "ERROR::INVALID PRICE");
  _premium = _price.div(10);

}
```

File: ./contracts/EStoreInsurance.sol

Now we can define a function that will be used to apply for a policy.

```
  ...

  _premium = _price.div(10);
}
function applyForPolicy(
  bytes32 _brand,
  bytes32 _model,
  uint256 _year,
  uint256 _price,
  uint256 _premium,
  bytes32 _currency,
  bytes32 _bpExternalKey
)external onlySandbox {
  require(_premium > 0, "ERROR:INVALID_PREMIUM");
  require(getQuote(_price) == _premium, "ERROR::INVALID_PREMIUM");
  bytes32 riskId = keccak256(abi.encodePacked(_brand, _model, _year));
  risks[riskId] = Risk(_brand, _model, _year);
  uint256[] memory payoutOptions = new uint256[](1);
  payoutOptions[0] = _price;
  uint256 applicationId = _newApplication(_bpExternalKey, _premium, _currency,␣
↪payoutOptions);
  emit LogRequestUnderwriter(applicationId);
}
```

File: ./contracts/EStoreInsurance.sol

We use the modifier "onlySandbox" here. It restricts permissions for this method to the sandbox account. As you will see later, as a product builder you can utilize the sandbox microservice to send transactions to your contract. The applyForPolicy method also contains the _newApplication invocation. It will create a new application in GIF core contracts.

Create other required functions in the same manner.

```
 ...

emit LogRequestUnderwriter(applicationId);
}
function underwriteApplication(uint256 _applicationId) external onlySandbox {
  uint256 policyId = _underwrite(_applicationId);
  emit LogApplicationUnderwritten(_applicationId, policyId);
}
function declineApplication(uint256 _applicationId) external onlySandbox {
  _decline(_applicationId);
  emit LogApplicationDeclined(_applicationId);
}
function createClaim(uint256 _policyId) external onlySandbox {
  uint256 claimId = _newClaim(_policyId);
  emit LogRequestClaimsManager(_policyId, claimId);
}
function confirmClaim(uint256 _applicationId, uint256 _claimId) external onlySandbox {
  uint256[] memory payoutOptions = _getPayoutOptions(_applicationId);
  uint256 payoutId = _confirmClaim(_claimId, payoutOptions[0]);
  emit LogRequestPayout(payoutId);
}
function confirmPayout(uint256 _claimId, uint256 _amount) external onlySandbox {
  _payout(_claimId, _amount);
  emit LogPayout(_claimId, _amount);
}
```

File: ./contracts/EStoreInsurance.sol

Don't forget to define events for your product.

```
contract EStoreInsurance is Product {
  event LogRequestUnderwriter(uint256 applicationId);
  event LogApplicationUnderwritten(uint256 applicationId, uint256 policyId);
  event LogApplicationDeclined(uint256 applicationId);
  event LogRequestClaimsManager(uint256 policyId, uint256 claimId);
  event LogClaimDeclined(uint256 claimId);
  event LogRequestPayout(uint256 payoutId);
  event LogPayout(uint256 claimId, uint256 amount);
  bytes32 public constant NAME = "EStoreInsurance";
...
```

File: ./contracts/EStoreInsurance.sol

Now is the time to deploy this contract. In the migrations folder, create the "2_deploy_EStoreInsurance.js" file with the content:

```
const EStoreInsurance = artifacts.require("EStoreInsurance");
const GIF_PRODUCT_SERVICE_CONTRACT ="<!-- Insert address of productService contract --
↪>";
module.exports = deployer => deployer.deploy(EStoreInsurance, GIF_PRODUCT_SERVICE_
↪CONTRACT);
```

File: ./migrations/2_deploy_EStoreInsurance.js

Look how we use the address of the ProductService contract to define GIF core contract. This address is published in GIF repository on Github . The product will interact with it.

```
npm run compile
```

If everything is fine, the contract will be compiled without issues.

```
npm run migrate
```

The contract will be deployed on the local blockchain.

CHAPTER 26

Interact with the smart contract

Send artifacts of your deployment to GIF Sandbox:

```
cd ./cli
./bin/run artifact:send --file ./build/contracts/EStoreInsurance.json --network␣
↪development
```

After that, the product will be approved automatically and you can start interacting with it. Enter the console mode:

```
./bin/run console
```

Run these commands one by one to go through the whole policy lifecycle from application creation to policy payout.

First of all, execute this command. If your product is approved, you will get information about it.

```
gif.product.get()
```

Create a new customer. This data is private and available only to the product owner.

```
gif.customer.create({ firstname: "Jow", lastname: "Dow", email: "jow@dow.com" })
```

Now start a new business process.

```
gif.bp.create({ customerId: "GET-CUSTOMER-ID-FROM-PREV-COMMAND" })
```

Here is how you can call your contract data:

```
gif.contract.call("EStoreInsurance", "getQuote", [100])
```

Now let's apply for a policy:

```
gif.contract.send("EStoreInsurance", "applyForPolicy", [ "APPLE", "A1278", "2012",␣
↪1000, 100, "EUR", "PUT-BP-KEY-HERE"])
```

Check if it is created:

```
gif.application.getById(1)
gif.application.list()
```

Underwrite the application. A new policy should be issued.

```
gif.contract.send("EStoreInsurance", "underwriteApplication", [1])
```

Check the new policy:

```
gif.policy.getById(1)
gif.policy.list()
```

Create a claim for the policy:

```
gif.contract.send("EStoreInsurance", "createClaim", [1])
```

Check the claim:

```
gif.claim.getById(1)
gif.claim.list()
```

Confirm the claim. A new payout should be created.

```
gif.contract.send("EStoreInsurance", "confirmClaim", [ 1, 1 ])
```

Check the payout status:

```
gif.payout.getById(1)
gif.payout.list()
```

As soon as we use fiat payments here, the external payout should be confirmed. Let's do it.

```
gif.contract.send("EStoreInsurance", "confirmPayout", [ 1, 100 ])
```

And check the final payout status:

```
gif.payout.getById(1)
gif.payout.list()
```

Congratulations!) You have just built your first insurance product!