TECHNICAL REPORT
IIT TR-05/2021

# Designing and implementation of a Model Driven Smart Contracts platform for Collaborative Processes

F. Donini, M. Martinelli

ISTITUTO DI INFORMATICA E TELEMATICA
Consiglio Nazionale delle Ricerche

# Designing and implementation of a Model Driven Smart Contracts platform for Collaborative Processes

Francesco Donini, Maurizio Martinelli

Digital Innovation Technological Unit

Institute of Informatics and Telematics

National Research Council - Pisa

## Table of Contents

# 1 Abstract

Model-Driven ChoreChain 2.0 platform responds to the need of distributed systems by offering a tool capable of building blockchain applications to enforce the correct execution of collaborative business processes, starting from a BPMN process model. Blockchain is a technology that offers basic building blocks to support the execution of collaborative business processes involving mutually untrusted parties in a decentralized environment. Several studies demonstrated the feasibility of designing blockchain-based collaborative business processes by means of a high-level notation. For example, the Business Process Model and Notation (BPMN) can be used to automatically generate the code artefacts required to execute these processes on a blockchain platform. In this technical report, we describe the principles and rationale of the model-driven approach to collaborative process automation deployed on the blockchain. To achieve these principles, we have created a platform capable of translating a BPMN process model into a set of smart contracts that can enforce the business process by triggering blockchain transactions on these contracts. We allow code artefacts to be deployed on the blockchain by encoding all the execution logic captured in the process model through ad-hoc tools. Business processes are modelled in BPMN 2.0 through a modeller UI component presented as a web application, which offers mechanisms to inject business logic inside. The whole platform is deployed as multi-container Docker applications and includes Ganache as local blockchain, while providing a REST API back-end within a microservice-based architecture.

# 2 Introduction

*Model-Driven* software development (MDD) consists of a model-based approach [1] [2]. Instead of requiring developers to spell out every detail of a system implementation using a programming language, it allows them to model only the functionality and the overall architecture that the system should have. This approach allows the developer to break away from the specific programming language and focus on the more important, often neglected aspects, such as design and interaction, further increasing the levels of abstraction.

Indeed, MDD aims at automating many of the complex (but routinely) programming tasks - such as providing support for systems persistence, interoperability, and distribution - which still need to be done manually today. Model-driven techniques involved in software development and testing methodologies are now quite widespread. Nowadays, they are applied successfully in different fields of software engineering, being very promising in terms of development time, cost reduction and quality of software product maintainability.

*Blockchain technology* adoption has grown in the last years because some of its features are very innovative, compared with the previous technologies [3]. In fact, blockchains are registries that can operate in untrusted environments - where there are not trusted parties - providing transparent, permanent and immutable transactions. Technically the blockchain technology is a growing list of records, which are called blocks. Each block is linked to the other using cryptography and contains a cryptographic hash of the previous block, a timestamp, and transaction data. The sequence of hashes linking each block to its parent, creates a chain going back all the way to the first block ever created. Blockchain technology has several features: it is *consensus based*, that is, a new block is appended to the chain only when a consensus is reached (according to the chosen consensus algorithm). It is *decentralized,* because replicated by a huge number of users and offers availability and protection from data loss. Taking advantage of cryptography, used to preserve block integrity and to link blocks in the chain, it guarantees *immutability* and tamper detection. Because users control their own data, the blockchain does not need a *central authority* and since all blocks are always visible, it guarantees *transparency*.

Taking advantages of both technologies, MDD [4] and blockchain, we addressed a study applied to distributed collaborative systems. Distributed systems are characterized by the complexity of coordinating the interaction between multiple components, possibly controlled by different organizations.

The *Business Process Model and Notation (BPMN)* [5] is what interested us most, among the tools used to model collaborative processes. BPMN is a graphical notation introduced to describe business processes but also widely used in the design of distributed system models nowadays. It is based on a flow-charting technique very similar to activity diagrams in Unified Modelling Language (UML). BPMN supports three main categories of processes: Simple Process, Process Collaboration, and Choreography Process. A Simple Process typically models a single coordinating point of view, while the Collaboration Process shows the participants and their interactions. Choreography Process [6] instead is a new model type (in BPMN 2.0). Its purpose is to show the interaction between participants in a different format focusing on the message flow instead of the individual detailed tasks of a process. For our purposes, the latter allows us to better describe system interactions in terms of message exchange from a global perspective. In fact, widely used standards like Business Process Model and Notation (BPMN) 2.0 collaboration diagrams are not well equipped to express choreographies succinctly. A more promising solution is provided by BPMN 2.0 choreography diagrams [7]. Indeed, the choice is justified because the scenario here considered refers to the possible integration and interaction of different organizations willing to cooperate in order to fulfil a shared goal. In such a scenario it is not relevant how a given organization behaves internally in order to enable the cooperation, instead the focus should just refer to the external observable interacting behaviour [8].

This work aims at modelling successful collaborative processes in distributed environments using a tool such as BPMN choreography model. According to the model driven approach, these models will be used for the generation of smart contracts [9] namely, an agreement between parties written in a particular language and executable on the blockchain. Leveraging the blockchain technology aims at offering a trusted environment between the parties [10]. In this way, thanks to Smart Contracts, it is possible to offer to participants a mechanism which should guarantee decentralised and reliable communications between them, while respecting data integrity without relying on a central authority. This work is the result of a collaboration carried out with the University of Camerino[1] in order to improve and continue the findings of this research [11].

The basic idea was to develop a solid product, easy to extend over time and which could be the starting point for future experiments. It would also be desirable for it to be a prototype easy-to-use and portable.
This document is organized in 8 chapters, each one is further divided into sections and sub-sections with the purpose of better illustrate the topics covered.

The document begins with an introduction showing an overview of the topics addressed and the motivations. The following chapters (Chapter 3,4) deal with the background of the main topics by providing a brief update of Model Driven Development, Blockchain Technology, BPMN and then an in-depth analysis with related works. Chapter 5 delves into the involved technologies trying to give an exhaustive overview. Chapter 6 explains to the reader the methodology applied, focusing in particular on the life cycle of the BPMN choreography used within the platform presented. Chapter 7 deals with technical aspects related to the prototype in order to explain the adopted solutions. Starting from the overview of the architecture and then moving on the details of the front-end, back-end, databases and generation of smart contract, explaining mechanisms and reasons related to the solutions found. In the final chapter (8), conclusions are presented by analysing the results obtained and describing the problems that are still open.

[1] http://computerscience.unicam.it/

# 3 Background

The purpose of this chapter is to provide an overview of the main topics covered in order to give the necessary context. In the first section the Model Driven Development approach is discussed, then, the BPMN world and the blockchain technology details described with attention to the Ethereum blockchain.

## 3.1 Model Driven Development

Model-driven development (MDD) is an approach used to write and implement software quickly, effectively and at minimum cost. MDD aims at the construction of a software model which must represent how the software system should work before the code is generated. Once the software is created, it should be tested using model-based testing (MBT) and then deployed.

The MDD methodology also groups other software development design approach such as model-driven architecture (MDA), model-driven software development (MDSD) and model-driven engineering (MDE).

MDD provides advantages in productivity compared to other development methodologies due to the simplifications introduced through the *model* during the engineering process. The model, in fact, represents the intended behaviours or actions of a software product before coding begins. The model is created by individuals and teams who collaborate in software development. Communication between product managers and developers, for example, provides clear definitions of what software is and how it works. Software management and test can be faster with MDD than with traditional development when developing more applications. In the Model-driven development there are two key concepts, abstraction and automation. The meaning of abstraction is the capacity of organising complex software systems. MDD is technically different from model-based development. The first is more in-depth than just having a model of the software in development. In MDD, complex software gets abstracted, which then extracts easy-to-define code. Once developers transform the abstraction, a working version of the software model gets automated.

## 3.2 Business Processes Management and Notation

For *business processes* we consider a set of tasks to be carried out in a certain sequential order and which aim at a business goal. Therefore, a business process can be represented by a model that coordinates its various activities according to the sequence of the workflow. To formally express this process, we will use the Business Process Model Notation (BPMN). BPMM is a graphic modelling language used for business analysis applications and specific workflows of business processes. It represents an open standard notation for graphical flowcharts usually applied to define business process workflows. It is based on popular and intuitive graphics in order to be easily understood by all interested parties, business users, business analysts, software developers and data architects.

BPMN supports three main categories of processes: simple process, choreography and collaboration.

- *Simple process*. It is a standard process, we most commonly come across in BPMN. It typically models a single coordinating point of view. A Simple Process describes a process within a single business entity that is contained in a Pool and normally has a well-formed context.
- *Collaboration process*. It shows the participants and their interactions. In BPMN, a collaboration only shows Pools and the message flow among them. To be more specific, a collaboration is any BPMN diagram that contains two or more participants, as shown by Pools which have a message flow among them.

- *Choreography process*. It is a new model diagram type in BPMN 2.0. Its purpose is to show the interaction between participants in a different format by focusing on the flow of messages rather than the individual tasks of a process. There are new object types, which include Sender and Receiver (instead of linking roles to a task or using pools) and connect sent/received messages. A choreography is a definition of expected behaviour, a contract between interacting participants. It shows the messages exchanged and their logical relationships. This allows business partners to plan their business processes for interaction without introducing conflicts.

## 3.3 Blockchain technologies

The blockchain is a distributed data structure replicated and shared between members of a network. It acts as a *distributed ledger* to keep track of the communications that exist between the participants of that network for the exchange of resources. Each communication is recorded in transactions, grouped in blocks with connected timestamps and constitutes the so-called *chain of blocks*.

The block chain is built assuming that each block follows this requirement:

1. It is identified by its hash value (value returned by a cryptography function applied on the contents of the block);
2. It contains the hash value of the block that precedes it in the chain.

Blockchain transactions are placed into the block, in bulk, if they are considered valid. They are defined *valid* when, through a distributed consent protocol, the entities constituting the network consider the occurred exchange to be secure since most of the participants in the network are "honest".

There are several distributed consent protocols. Thanks to Bitcoin, the most famous is *Proof of Work* (PoW), in which the network nodes must solve a computationally intensive activity to be selected for the insertion of a new block in the chain.

Another important protocol, thanks to the diffusion of Ethereum, is *Proof of Stake* (PoS). Here the selection of the new block creator is based on the principle that each user is required to demonstrate possession of a certain amount of cryptocurrency.

Other distributed consensus protocols have also been studied, the best known being that of the *Byzantine Full Tolerance* (BFT) algorithm.

In general, transactions must be validated according to previously defined rules; these rules are contained within what is called *Smart Contract* (SC).

SC is a predefined program written in some language (in our case *Solidity*) which codes the computation of transaction validation.

Blockchains using SC can be considered general purpose application platforms and the most popular and supported of them, at the time of writing, is *Ethereum*. Furthermore, it must be taken into account the domain of the application for which we can have different blockchain implementations:

- **Public blockchains** anyone can join the network anonymously. According to the constraints introduced in the consensus algorithm, this type of blockchain can be further classified into two groups:
  - *Permissionless blockchain*, as **Bitcoin** blockchain, any node on the network can participate in the consensus algorithm by validating transactions;

  - *Permissioned blockchain*, as **Ripple4** or **Stellar5** blockchain implementations, where only the nodes that respect certain rules can participate in the transaction validation algorithm and therefore be part of the consensus.

- **Private blockchains** only a selection of network nodes is enabled to join the network. As in public blockchains, they can be classified in *permissionless*, where any of the nodes can participate in the consensus algorithm (for example *Ethereum*), and blockchain *permissioned*, where only a set of the nodes are authorised to validate the transactions. In this case, the exploitation of the *Proof of Authority* by the distributed consensus protocol gives the authorisation to the nodes to create new blocks.

Executing a Smart Contract in the blockchain guarantees:

- *atomicity*, each operation is performed entirely or fails without ever affecting the status of the contract;
- *synchronicity*, the  transaction code is executed synchronously;
- *origin*, the code can only be executed by tracing external calls;
- *availability*, the code and associated data are always available because the contract is always reachable;
- *immutability*, the code cannot be changed and cannot be tampered with once it has been deployed;
- *persistence*, the code and data can only be removed through commits of self-destruct operation.

## 3.4 Ethereum

It is a project born and developed as a public Blockchain. It is an open source distributed computing platform conceived to make available creating, publishing and managing SC in a peer-to-peer mode. In a nutshell and perhaps with excessive simplification, it can be said that while the Blockchain is a platform for ”*Distributed Databases*”, Ethereum is a platform for ”*Distributed Computing*”, which has one of its main components in the Ethereum Virtual Machine (EVM). Ethereum is a computational platform *remunerated* through exchanges on a cryptocurrency calculated in **Ether**. It is a platform that can be adopted by all those who wish to join the Network and who, in this way, have a solution allowing all the participants to have an immutable and shared archive of all the operations implemented during his life. At the same time, it is designed not to be stopped, blocked or censored. Ethereum is a **programmable Blockchain** allowing users to create their “own operations” or different types of decentralised Blockchain applications (Dapps). The Ethereum engine is represented by the EVM, which actually represents the runtime environment for the development and management of Smart Contracts in Ethereum. EVM operates in a protected way, being completely separate from the Network. The code managed by the Virtual Machine does not have access to the Network and the same Smart Contracts generated are independent and separate from other Smart Contracts. Smart Contracts are therefore available on the Blockchain in *EVM bytecode* (an Ethereum-specific binary format), are written in Ethereum high level language, transformed into byte codes with an EVM compiler and uploaded to the Blockchain with an Ethereum client. Ethereum is a *Turing complete* system, that allows developers to create applications running on the EVM using programming languages that, in turn, refer to traditional platforms such as JavaScript and Python. Ethereum was the first blockchain with Smart Contract functionality, making it free for everyone to program them using the **Solidity** language. The implementation and execution of each transaction have a **GAS** cost, which can vary depending on the overload of the network. **GAS** is a virtual fuel used to execute SC. The EVM uses an accounting mechanism to measure the consumption of GAS and limit the consumption of computing resources.

Below, some definitions of elements making up the blockchain Ethereum environment are reported and used into the rest of the document. Definitions are taken from [12].

**Account:**

An object containing an address, balance, nonce, and optional storage and code. An account can be a contract account or an externally owned account (EOA).

**Address:**

Most generally, this represents an EOA or contract that can receive (destination address) or send (source address) transactions on the blockchain. More specifically, it is the rightmost 160 bits of a Keccak hash of an ECDSA public key.

**DApp:**

Decentralized application. At a minimum, it is a smart contract and a web user interface. More broadly, a DApp is a web application that is built on top of open, decentralized, peer-to-peer infrastructure services. In addition, many DApps include decentralized storage and/or a message protocol and platform.

**GAS:**

A virtual fuel used in Ethereum to execute smart contracts. The EVM uses an accounting mechanism to measure the consumption of GAS and limit the consumption of computing resources.

**EOA:**

Externally Owned Account. An account created by or for human users of the Ethereum network.

**Ether:**

The native cryptocurrency used by the Ethereum ecosystem, which covers gas costs when executing smart contracts. Its symbol is the Greek uppercase Xi character.

**Event**

Allows the use of EVM logging facilities. DApps can listen for events and use them to trigger JavaScript callbacks in the user interface.

**Receipt:**

Data returned by an Ethereum client to represent the result of a particular transaction, including a hash of the transaction, its block number, the amount of gas used, and, in case of deployment of a smart contract, the address of the contract.

**Transaction:**

Data committed to the Ethereum Blockchain signed by an originating account, targeting a specific address. The transaction contains metadata such as the GAS limit for that transaction.

**Wei:**

The smallest denomination of ether. 1018 Wei = 1 ether.

## 3.5 Solidity

Solidity is a programming language created for writing the SCs. It is an object-oriented and high-level language compiled in bytecode and executed on EVM. With Solidity, self-enforcing business logic embodied in smart contracts is implemented, thus leaving a non-repudiable and authoritative record of transactions on permisionless blockchain. Using Solidity is easy enough, from a developer point of view (apparently for those who already have programming skills). In Solidity a smart contract is formed by a collection of codes (functions) and data (its internal status) which has its specific address on the Ethereum blockchain. It has been designed to maintain the ECMAScript syntax and to make it familiar for Web developers, but unlike ECMAScript, it has variable static and return character types. Indeed, complex member variables are supported for contracts that include arbitrarily hierarchical mappings and structures. Inheritance, including multiple inheritances, is also supported. An *Application Binary Interface* (**ABI**) has also been introduced, in order to facilitate multiple secure functions within a single contract and to better describe the functions signatures. This is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction. Data are encoded according to its type, as described in this specification. Since data encoding is not self-describing a decoding scheme is required.

We assume the interface functions of a contract are strongly typed, known at compilation time, and static.

## 4 Related works

Nowadays the adoption of blockchain technology has grown due to some of its features which are very innovative [3]. Consequently, the experimentation of the BPMN model applied to blockchain technology is also spreading a lot. In fact, this is perceptible by the growing interest found in literature where scientific articles increasingly concern these topics.

The topic of Model-driven engineering methodologies, if applied to distributed systems, shows the same trend when connected with the blockchain environment. In fact, as reported by the authors [8] "In the context of blockchain-based applications, model-driven development is of particular relevance"; they introduced two different approaches used for model-based code generation. The first approach uses collaborative process models that cross organisational boundaries, while the second one is aimed at asset registers, such as title deeds, cars or digital assets. Interesting aspects can also be found in [4], where the goal is to introduce automation to software development processes on a large scale, or in [1] [2] where authors illustrate techniques and concepts related to the model-driven engineering. Thanks to the work done by the authors [8], the choreography diagrams are compared with those of collaboration in favour of the BPMN representation taken from different perspectives.

Moving a little bit more towards the topic of the blockchain, a relevant book [8] offers an architectural view of software systems that make beneficial use of blockchains and describing everything that software architects and developers need to know in order to build applications based on blockchain technology. The book focuses on the bigger picture for blockchain, covering the concepts and technical considerations in the design of blockchain-based applications. Furthermore, of particular interest are also the studies illustrated in [13] where the authors explain the possibility of developing blockchain-oriented software (BOS) that implements part of the business logic in the blockchain by using smart contracts. In this article they show three complementary modelling approaches (Entity Relationship Model, Unified Modelling Language and the Business Process Model and Notation) to realize a particular use case. In this proposal [21], instead, the authors describe a method based on model-driven engineering (MDE) of collaborative business processes over blockchain technology. They propose that the collaborative business process is modelled using a BPMN choreographic diagram. A choreographic diagram is a model of a collaborative process that offers a series of activities for the exchange of messages (choreographic activities) and routing constructs of the control flow (in particular XOR gateway and AND called parallel). In the approach of Weber et al. the parts in a choreography interact through message exchanges, which are sent as transactions on the blockchain. Prybila et al. [14] describe a different approach to monitor business processes compiled on the Bitcoin blockchain through specialized tokens. Similar to the previous proposal, they demonstrate that collaborative process can be modelled as a choreography. Both approaches assume that the parties in a collaborative business process interact via message exchanges and through the blockchain, to record message interactions and to verify or apply those exchanges in a given order. This effectively means that the blockchain platform acts as an execution component of a collaborative process. Relevant also this paper regarding security issues in inter-organisational processed over blockchain [19]. The authors discuss how blockchain can be used in support of secure inter-organizational processes pointing out which additional security issues the use of blockchain can bring such as data integrity and data confidentiality. Moreover it is underline how blockchain relies on the presence of a "trustable" mediator, called Oracle, that retrieves data from an external source and directly delivers them to smart contracts. Meddling et al. in [17] underline that the choreography diagrams of BPMN 2.0 have not found widespread adoption in the sector but are still consider them one of the promising approaches for the design of inter-organizational business processes. They state that the “whole area of choreographies may be revitalized by blockchain technology”.  However, the point out that for collaboration diagrams the expressive limitations in ownership and observability still represent a major obstacle while choreography fits better with the concept of smart contract. In their discussion the smart contract may store and manipulate data, own decisions and their logic and keep track of the overall execution state. Interactions within the choreography become transactions. Observability broadens considerably as each participant has access to the whole distributed ledger of transactions attached to the choreography.

Even more complex tools and similar to ours proposal, have already been released and presented. Starting from [22] it is introduced Caterpillar [15, 18], an open-source Business Process Management System (BPMS) that runs on top of the Ethereum blockchain. The main feature is that ”the state of each process instance is maintained on the Ethereum blockchain, and the workflow routing is performed by smart contracts generated by a BPMN-to-Solidity compiler”. Caterpillar has an engine exposed via a REST API and all core modules are implemented in Node.js. Its aim is to enable its users to build native blockchain applications to enforce the correct execution of collaborative business processes starting from a BPMN process model. In this context, the meaning of “native” is that code artefacts deployed on the blockchain encode all the execution logic captured in the process model. Specifically, Caterpillar aims at fulfilling the following three design principles. First, the collaborative process is modelled as if all the parties shared the same process execution infrastructure (the blockchain). Second, the full state of the process instance and of its subprocess instances is recorded on the blockchain. Third, the execution of a process instance proceed. To achieve these principles, Caterpillar translates a BPMN process model into a set of smart contracts.

Lorikeet [9] is another model-driven engineering (MDE) tool for the development of blockchain applications in the space of business processes. This tool consists of a modeler user interface and back-end components including the BPMN translator, Registry generator and Blockchain trigger. It can automatically produce blockchain smart contracts from business process models and asset data schemata. Lorikeet is a well-evaluated tool that is used for creating blockchain smart contracts in industry and academia and in commercial use by Data61 and has been applied in numerous industry projects. For further implementation regarding both platforms’ details, it recommends reading the relative papers or this interesting article [8] which provides a comparison between the two tools.

Several products related to Collaborations processes are already easy to find in internet such as Camunda[2], Activiti[3], Bonitasoft[4] or others.

As already pointed out in the introduction this work is inspired by [1], in order to enrich and improve the sketched prototype and their findings. The authors propose an interesting methodology to perform a Bpmn-to-Solidity translation and the whole life-cycle of choreographies approach.

We summarise that there are several existing works proposing approaches for the execution of collaborative business processes on blockchain technology used to record message exchanges or transactions. All the solutions, by means of smart contracts, control and (or) impose that messages are exchanged in a compatible way, with a collaborative process model, BPMN based. In practice,

[2] https://camunda.com

[3] https://www.activti.org

[4] https://www.bonitasoft.com

the blockchain is used to record, monitor, and occasionally control the interactions among the processes performed by each party.

# 5 Technologies

This Chapter presents a brief summary of concepts and technologies involved in the project. It starts with Section 5.1 where it is introduced the "full stack" concept, then, in Section 5.2 it is presented the ReactJS technology used to realise the front end together with `chor-js,` and `bpmn-js` described in the Sub-section 5.2.1. Next Section 5.3 and Section 5.4 describe the back-end side and indicate the database chosen and the SpringBoot framework. In conclusion sections 5.6 and 5.7 introduce the blockchain world.

## 5.1 Full Stack Architecture

Nowadays it is very common to hear about Full Stack applications. Actually, it is a modern way of referring to an entire computer system or application. From the front end (customer or end user) to the back end ("behind the scenes" technology such as database and internal architecture) to the software code that connects the two.

Front end, is where the full stack web developer uses a combination of web-oriented technologies such as HTML, CSS and JavaScript to create everything a user sees or uses to interact on a website.

Back end, is where the developer design in which way provides and retrieves data from the database server, how to process information and in which format provides results.

The necessary skills needed to set up a similar system are focused on stack of solutions such as a LAMP (Linux, Apache, MySQL, PHP) or MEAN (MongoDB, Express.js, AngularJS, Node.js) or other mixed technologies as our case Java, ReactJs and Mysql.

Developing both, front end and back end parts of an application, involves three levels. The *Presentation Layer* (the front-end part that deals with the user interface) the *Business Logic Level* (the ack end part that deals with the user interface) and the *Database Layer*. This architecture is also called the *three-tier web architecture* (TTWA).

This system allows the business logic separation from the UI, from data storage and from the database.

TTWA is designed to provide a high degree of flexibility and greater security that can be applied in different ways for each service at each level. TTWA also ensures improved performance because the activity is shared between servers. There are however also some significant downside of the TTWA. The complexity compared to having only 1 and 2 levels, the cost of maintenance and distribution of the network, which is greater than 1 and 2 levels and the absolute need to find a small team of trained developers or developers with extensive know-how.

## 5.2 Front-end (ReactJS)

ReactJS[5] is a free JavaScript library used for creating user interfaces, at the time of writing the current ReactJS release is 17. Created by Facebook[6], React was initially released in 2013, first with the BSD+Patents. In September 2017 the license had change with the MIT license, more acceptable due to potential problems regarding intellectual property for developers.

React is a declarative language that makes the creation of interactive UIs painless. By designing simple views for each state in the application, ReactJs will efficiently update and render just the right components when the data changes. It has some unique core concepts such as, virtual DOM,

[5] https://reactjs.org

[6] https://www.facebook.com

JSX components, input properties, props and the peculiarity the each React component has a proper state and a life cycle.

The **virtual DOM** is a node tree, just like the DOM. *"The virtual DOM (VDOM) is a programming concept where an ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM. This process is called reconciliation. This approach enables the declarative API of React: You tell React what state you want the UI to be in, and it makes sure the DOM matches that state. This abstracts out the attribute manipulation, event handling, and manual DOM updating that you would otherwise have to use to build your app.* " [16]

**JSX** is officially an XML-like syntax, similar to HTML but actually JavaScript. Is an extension of JavaScript to use with React to describe what the User Interface should look like. JSX may remember a model language, but it uses all the power of JavaScript. Below some examples:

```
react.createElement(component, props, ...children)
```

```
const element = <h1>Hello, world!</h1>;
```

All starts from the `react.createElement`. Instead of having to create an element by hand, we define a component. This component has several different attributes that we pass in to it. Then, it no more necessary to create the element, define the tag, and then pass in all the attributes etc.

**Components** are like JavaScript functions. The positive aspect of React, is that it splits the User Interface into independent reusable pieces. These pieces have an input of arbitrary size, a set of props, and then they return a React element. Each component is always returning a rendering function, composed by the elements that we want it to display. The rendering is a key point of this implementation.

**Props** are the overall attributes and properties of the component, in practice, the way in which the components pass the data. Below, it is shown how we deal with different attributes. As we see in the following example we can assign, as an attribute, the name of the author that built `MyClock` component. We just pass a name here and we will be able to use `this.props.name` when rendering this particular component. This is an easy way to pass data in and out.

```
class MyClock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world I am {this.props.authorName}!</h1>
      </div>
    );
  }
}
```

Each component has a **state**, and it actually manages its own state. The state can be extracted and used or set in our code like a `prop`. It is the internal state of the component and, as developers, we're actually responsible for updating and dealing with `state`. In the example below, we see that when we create this clock component in the constructor, we have `this.state`. We pass in a new date, and then we can actually output that, in the render function. We can use easily states to perform common tasks like setting and extracting the state.

```
class MyClock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  render() {
    return (
      <div>
        <h1>Hello, world I am {this.props.authorName}!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Each component has a specific **lifecycle** that we can control. We have *mounting*, *updating*, and *unmounting* functions. The constructor, for example, can help us to set up the initial state. From the initial state  we have other events that we can hook into.

Comparing ReactJs with Angular or other MVC frameworks makes no sense since React is just a representation. React is a template-based language combined with several functions that support output to HTML, e.g. the result of React's operation is HTML code.

### 5.2.1 Chor-Js and Bpmn-Js

`Chor-js` is a JavaScript based library born as support tool for BPMN Choreography diagrams that provide a web-based, open-source choreography modelling framework [6, 10] based on `bpmn-js`.

`chor-js`[7] it is a full editor aims to be an extensible, intuitive and easy to integrate choreography diagram modeler, compliant with the BPMN 2.0 standard addressed at researchers and users alike.

`bpmn-js`[8] is a BPMN 2.0 huge library oriented to rendering toolkit and web modeler.

It is entirely written in JavaScript and embeds BPMN 2.0 diagrams. Is completely independently and it's required only a modern browser to be used with no needs of server backends. This is an important key point demonstrating that makes it easy to embed it into any web application. The library offers both a BPMN Choreography diagrams viewer and web modeler.

`chor-js` is a web-based framework adapted to recent web browsers. Figure 1 shows the graphical user interface of the `chor-js` application taken from the prototype. There are four principal components provided by the core `chor-js` library: (*i*) a left-hand side palette containing modeling tools and elements; (*ii*) a top palette implements several extensions (e.g. exposing features like importing, saving, exporting, etc.); (*iii*) another palette providing a switching and renaming mechanism for managing multiple diagrams in one model; (*iv*) a context menu providing actions on the currently selected elements.

[7] https://github.com/bpmn-io/chor-js

[8] https://github.com/bpmn-io/bpmn-js

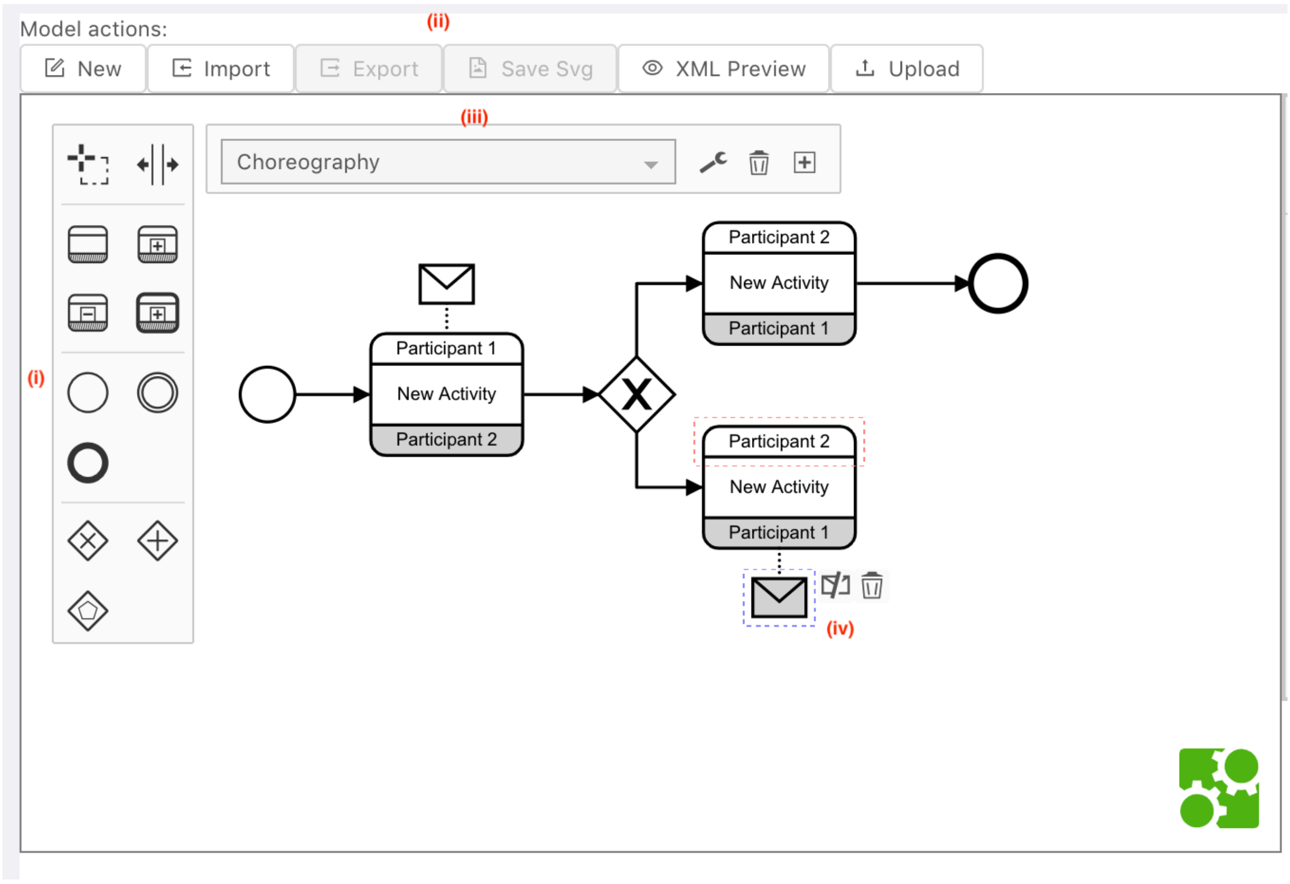


*Figure 1 - chor-js web-based application*

## 5.3 Database (Mysql)

Rather than talking about the chosen technology, which however concerns the adoption of **Mysql**[9] (or `SQLite`)[10], we would make a small digression regarding the type of database chosen. In fact, it was not difficult to choose which product to use but rather to understand which technology best suited the project between SQL and NoSQL.

We will motivate the decision thanks to a comparison between relational and non-relational databases proposed below.

Deciding whether to use SQL vs. NoSQL only depends on the type of information we are storing and the which is the best way to store it. Both technologies store data, but do it in a different way. So, the real answer is that it depends on what you are building, on the constraints dictated by who you are building and on the final state you are trying to achieve.

It must be said that while NoSQL is trending, and the adoption rate is increasing, it is not a substitute for SQL. It's just another option. Today it happens that you have to choose one over the other, but perhaps the best approach is to use them both.

Going back to us, the best option is probably to adopt SQL technology for some of these reasons:

[9] https://www.mysql.com/it/

[10] https://www.sqlite.org/index.html

- we may find ourselves working in the future with rather complex queries and NoSQL does not support relationships between data types. NoSQL queries exist but are much slower;
- SQL databases are better suited to heavy or complex transactions because they are more stable and guarantee data integrity;
- ACID compliance (Atomicity, Consistency, Insulation, Durability) must be guaranteed;
- if drastic changes due to the growth of the project are not foreseen and if we do not work with large volumes of data or many types of data that are probably unrelated to each other, NoSQL would be excessive;
- MySQL but I would say SQL technologies works perfectly with JPA via Spring Boot Framework;

## 5.4 Back end – *Spring Boot*

Spring Boot, built on top of the Spring framework, introduce a completely new development model that makes Java development very simple, avoiding some tedious development steps, code and boilerplate configuration that significantly increase the speed of development.

At the time of writing the document, Spring Boot is in the stable release version `2.0`. For the first time Released in mid 2014 in its history Spring Boot has had many developments and improvements.

It provides an easier and faster way to set up, configure and run both simple and web-based applications (the latter is not used in this project).

In the main Spring framework, you can configure everything yourself. Hence, it is possible to have many configuration files, such as XML descriptors. This is one of the main problems that Spring Boot solves for you.

Intelligently select dependencies, automatically configure all the features we would like to use and we can start our application with a click or simply launch an execution script. In addition, it also simplifies the distribution process of our application. In fact it has an integrate Tomcat server in which automatically it deploys the applications.

It can be a little scary, because there seem to be a lot of "magical" things going on in the background, but it really makes our life easier.

It has several features that help developers:

- *Dependency management*. Through starters and some package manager additions. For instance, combining Spring Boot with *Spring Data* and *Spring Security* we can have something up and running in no time. And it is not just "something", it is a solid base to build upon.
- *Automatic configuration*. Reduction of configuration times by trying to decrease the amount of configurations that a complex Spring application requires.
- *Production-ready features*. For example, some are Actuator, logging tool, monitoring, metrics or various PAAS integrations, etc.
- *Improved development experience*. Many test suites built on top, well integrated and available, or a better feedback loop using `spring-boot-devtools` package.

In a nutshell, the main goal of the Spring Boot Framework is to reduce development, unit test and integration times and facilitate the development of production-ready Web applications compared to the existing Spring Framework, which really takes longer.

## 5.5 REST

REST stands for Representational State Transfer. There is a set of protocols and standards that describe how communication between computers and other applications should take place, through the network, for the exchange of resources.

*"The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g., a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time."* [20]

Actually, REST is an architectural model and design for server network applications. Uses simple HTTP methods (*verbs*) to communicate between client and server. In fact, the main feature is that REST API uses the `GET`, `POST`, `PUT`, `DELETE` methods to communicate.

It should be noted though that HTTP and REST are not the same thing. REST API, as is known is a *set of rules* that developers follow when creating their API.

One of these rules states that we should be able to get a resource when we link to a specific URL. This URL represent a resource locator and the browser it is the means to retrieve it.

Now a list of some important REST properties:

- a REST service exposes resources, not methods, as occurs in a SOAP-based service;
- JSON is the format used. With JSON usually the content of the resource is presented, but nothing prevents the use of other types of media such as XML;
- the REST model is generally implemented through the HTTP protocol; HTTP methods / verbs are assigned to define actions to be used on resources;
- a REST web service is intrinsically stateless, so forget the concept of session; the server has run out of memory and each request will be handled independently.

A resource can be a singleton or a collection.

For example, "*employers*" is a collection resource and "employer" is a singleton resource. We can identify "employers" collection resource using the URI "`/employers`". We can identify a single "employ" resource using the URI

"`/employer/{employerId}`".

A resource may contain sub-collection resources also. For example, sub-collection resource "accounts" of a particular "employer" can be identified using the URI

`/employer/{employerId}/accounts`.

Similarly, a singleton resource "account" inside the sub-collection resource "accounts" can be identified as follows:

"`/employers/{employerId}/accounts/{accountId}`".

## 5.6 Ganache-cli

Ganache[11] is a virtual blockchain, previously known as `TestRPC`. Ganache CLI, part of the `Truffle`[12] suite of Ethereum development tools, is the command line version of Ganache, your personal blockchain for Ethereum development. It uses to make developing Ethereum applications faster, easier, and safer. It also includes all popular RPC functions and features (like events) and can be run deterministically to make development a good experience.

`ganache-cli`[13] is written in JavaScript and distributed as a `Node.js` package via `npm` or `yarn`. It is a blockchain emulator, fast to use and very versatile for the developer because it can be configured in many ways using several options. It mainly allows working in a blockchain environment, however without the need to face the overhead of running a real Ethereum node.

There is no "*mining*" operation by default with Ganache, in fact every transaction is immediately confirmed and is making its way at no transaction costs. This is very useful feature because make it possible iterative development: writing unit tests for the code that are performed on a simulated block, distributing smart contracts, calling functions and then reducing everything for further simulation or new tests, shutting down and resetting with the possibility of losing all the stored data.

It provides by default 10 predefined Etheruem addresses, with all private keys and preloads wallets with 100 simulated Ether each.

Ganache is available in two versions: CLI and UI. The recommendation is to use the `ganache-cli` version for its simplicity, speed and reliability. When it is turn it on, it will automatically be set to a specific port and IP address. The default address for the UI version it is `localhost:7545` or `127.0.0.1:7545`, while the CLI version tends to go with port `8545`. The GUI version gives a different overview of the *testchain* events.

## 5.7 MetaMask

MetaMask[14] is a free extension for web browsers that allows easy communication between web applications and the Ethereum blockchain. MetaMask acts like a wallet simplifying interaction hence token exchange with the Ethereum blockchain. In other words, MetaMask is a wallet for your browser.

MetaMask can be used:

- as a common wallet to send and receive `Ether`, the native currency of Ethereum;
- to manage ERC20 tokens;
- to connect directly to decentralized exchanges (DEX) without having to create new addresses;
- to interact with Ethereum dApps

[11] https://www.trufflesuite.com/ganache

[12] https://www.trufflesuite.com/

[13] https://github.com/trufflesuite/ganache-cli

[14] https://metamask.io/

# 6 Methodology

This chapter begins with an overview of the application and subsequent sections explain in depth the methodology developed and adopted.

## 6.1 Platform Overview

The Model Driven ChorChain 2.0 platform responds to the need to offer a tool capable of creating BPMN choreographic models (3.2) describing collaborative processes - more simply, the interactions among the participants involved in the model. The goal is to translate the model in Smart Contracts using Solidity as language. The Smart Contract's code is the result of the BPMN translation performed on a collection data which are collected by an approach Model Driven Development based (3.1). The translation takes place by means of the BPMN elements that make up the models. During the translation process Elements are used as *vectors* in order to carry important meta data. The Smart Contracts, deployed on the blockchain, guarantee immutability of the data involved in the message exchanges among the choreography participants, certifying the complete history of all the occurred operations connected to each transaction.

Through a dedicated UI the interaction mechanisms between all the actors are allowed, according to the rules described in the SCS created by the BPMN choreography model.

The MDD approach helps users to design collaborative process choreographies in an expressive way in order to make them operate on blockchain technology (3.3) as a Smart Contract and relieving the user from having to write code.

The platform intrinsically defines each choreographies life cycle. A life cycle of a choreography can be described according to this modelling. It is first **designed** as a BMPN *model* then created and stored. Secondly, the BPMN model is **translated** into a SC (via Solidity code) (3.5) constituting a *model instance*. Thirdly, the model instance generated is **deployed** as a "skeleton" smart contract on the blockchain - it is called skeleton Smart Contract because it contains a structure of stub functions inside defining something similar to a class interface – making a *distributed model instance*. Then, starting from the distributed model instance are generated one or more *model instance implementations* used as Collaborative Process among participants. In conclusion, after all the model instance implementations have been created, they become consumable between Participant through SC **interactions** governed by means of the dApp Interface.

The process described can be summarised in the four phases listed below which also represent the study addressed and the methodology adopted:

- Choreography Design
- Choreography Translation
- Choreography Deploy
- Choreography Interaction

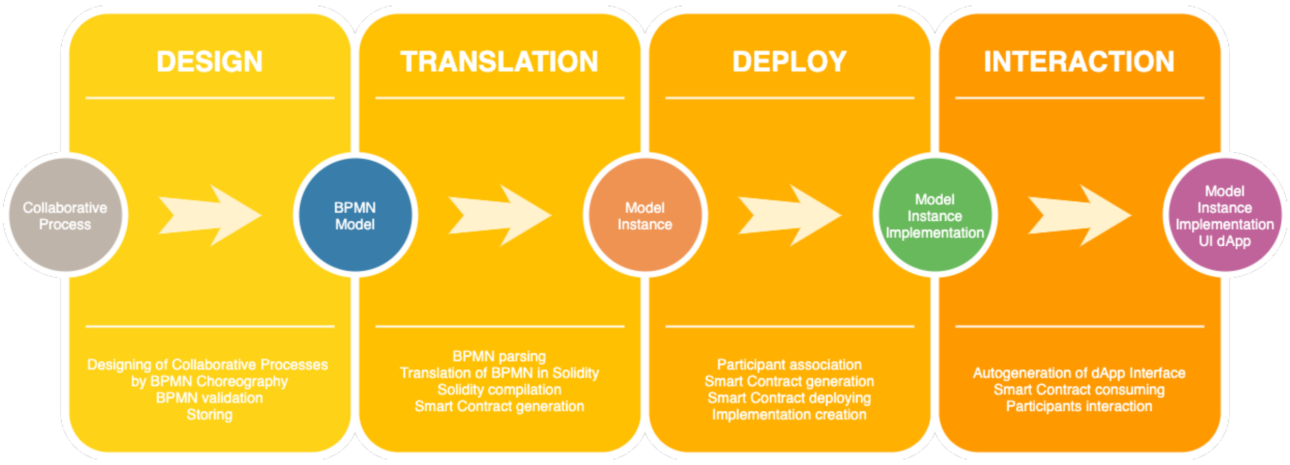


*Figure 2 - Choreography phases and life-cycle*

We point out that the MDD approach is horizontally permeated in the first three phases as we discuss later.

## 6.2 Choreography Design

The *first phase* involves the collection of user data. It gives the opportunity to design and model business processes at will, indicating the interaction with users, through ad hoc BPMN choreography editor. Here it will be possible to define which other participants interact with the model, defining their specific roles in the Choreography.

Choreographies are stored within the platform with associated information such as name, author, description, creation date etc. Basically, the idea is to store models - collaborative processes - to make them still available and reusable over time. Archiving is however only possible if the BPMN (XML based) document produced is successful validated by formal test.

In the second phase, as we will see shortly, each designed and saved choreography will be subsequently transformed into a real and proper Smart Contract.

The ad hoc editor has various features in addition to those necessary for BPM design (5.2.1). It is actually equipped with several custom functions aimed at extending the generated BPMN model. Practically speaking, the already present BPMN elements model are enriched with further information which are then processed in the following step. Hence, at the end of the Choreography Design phase, a choreography *model* is generated and stored.

## 6.3 Choreography Translation

The *second phase* concerns the translation of the BPMN choreography model into a programming language - Solidity in our specific case. The choreography, written essentially in XML, is sent to a platform tool - actually an API - which analyses, validate and interprets it. Once the choreography has been parsed the tool takes care of carrying out the translation process and generating the equivalent Solidity code. The methodology adopted granting that every XML block - corresponding to a specific BPMN element - is translated into Solidity according to (previously) predefined constructs [11].

Given that the BPMN consists of a Sequence Flow of elements, following the connections of the elements between them is enough to reconstruct the logic dependency tree of the whole elements. Subsequently, by coding each element of the tree encountered with the corresponding Solidity (predetermined) construct, it becomes relatively easy generating code.

During code generation, additional information, previously "injected" by the designer is considered. Thanks to the coding of the standard BPMN elements plus the additional information supplied with the BPMN elements, it is possible to proceed with the **skeleton smart contract** generation.

## 6.4 Choreography Deploy

The *third phase* is incapsulating the latter. Here we deal with collecting the operational data of the choreography in terms of who is interacting with whom and with which roles are involved in the collaborative process. Indeed, before proceeding to the code generation it is required specifying the association between all the actor's roles with real participants. For each choreography, there are two role types, **mandatory roles**, where the role association is mandatory in order to proceed with the contract's compilation, and **optional roles,** where the role association is not strictly required at this stage. Once the required role associations are completed, it will be possible to proceed with what we have called **deploy**.

In this context *deploy* means translating (6.3) the BPMN choreography model into Solidity language, compiling the Solidity code translated into Smart Contract and then, if all the step succeeds, generating a new Choreography *model instance*.

Therefore, Smart Contracts contains the business logic of our collaborative process described by the BPMN choreography model. SCs can be summarised as the protocol that defines how transactions of that collaborative process take place over the blockchain. The general objective of the SC is to satisfy common contractual conditions. For instance, if we want to design an auctioning system on Ethereum, we need to develop specific smart contracts according to the rules of the auctioning system chosen.

Thus, each **model instance** describes the "starting" business logic of the Choreography model associated. We said "*starting business logic*" because, with this specific Smart Contract, we want to represent the initial point where to build, later on, the Business Logic. We call it *skeleton Smart Contract* which is composed of three contracts (Figure 3) which follow the Template Method design pattern approach (6.4.1):

1. contract containing the **business workflow** of the BPMN choreography model invoking **stub methods** *(a)* which corresponds with the real user operation;
2. contract containing the **interface** (actually a *solidity abstract contract*) of the implementation contract *(b)*, stub methods interface.
3. contract containing the **implementation** of the stubs method *(c)*.

### 6.4.1 Template Method Design Pattern approach

Used to define the behaviour of a "superclass" by delegating some detail steps to subclasses. This pattern satisfies the need to specify the order of operations to be performed, delegating the implementation of certain operations to the subclasses. Therefore, the method that defines the algorithm, is implemented in the superclass while, the methods that define the detailed behaviours are declared abstract in the superclass and implemented in the subclasses.

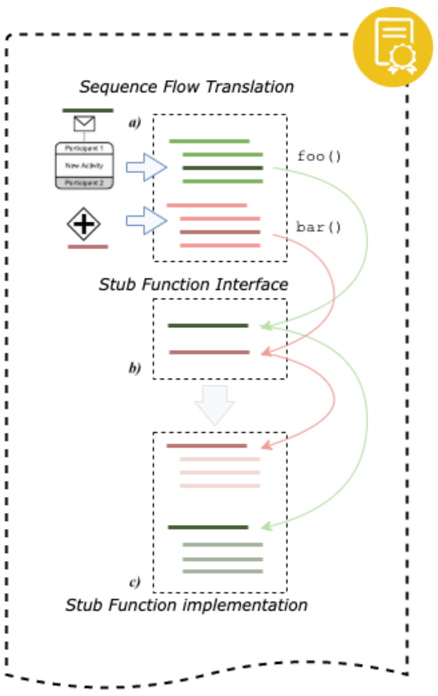


*Figure 3 – Generated smart contract architecture*

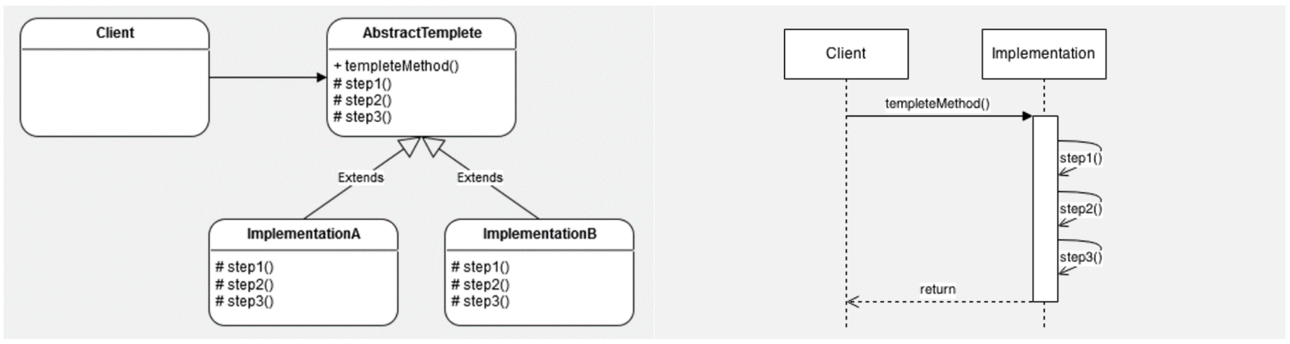


*Figure 4 - Template method class and sequence diagram*

The components included in the Template Method pattern are as follows:

- **Client**: It's the component which triggers the execution of the template (Figure 4 *a)*
- **AbstractTemplate**: It's an abstract class including a series of operations which define the necessary steps for carrying out the execution of the algorithm. This class has a templateMethod method for executing step1, step2 and step3 in order (Figure 4 *b)*
- **Implementation**: This class represents a concrete template which inherits from AbstractTemplate and implements its methods (Figure 4 *c)*

Then the interactions between components:

1. The client creates and gets an instance of the template implementation.
2. The client executes the templateMethod.
3. The default implementation of templateMethod executes the implementation returns a result.

In software development a **stub method** or simply *stub* is a piece of code used to stand in for some other programming functionality. A stub may simulate the behaviour of existing code (such as a procedure on a remote machine, such methods are often called mocks) or be a temporary substitute for "*yet-to-be-developed*" code as in our case.

We are trying to exploit some important functionalities of Solidity. One of these is **extensibility**, it is key when it comes to building larger, more complex distributed applications (Dapps). Solidity offers two ways to solve this problem within dapps: *abstract contracts* and *interfaces*.

*Abstract Contract* in Solidity are similar to classes in object-oriented languages. They include state variables that contain persistent data as well as functions that can manipulate the data in the state variables. Contracts are identified as abstract contracts when at least one of their functions lacks an implementation. As a result, they cannot be compiled. They can however be used as base contracts from which other contracts can inherit from.

On the other hand, there are Interfaces. *Interfaces* are similar to abstract contracts, but they are limited to what the contract's ABI can represent. In other words, you could convert an ABI into an interface, or vice-versa, and no information would be lost. According to the Solidity docs they have a few additional restrictions.

Along with improved extensibility, abstract contracts provide better self-documentation, instil patterns (like the Template method), and eliminate code duplication, so for our purpose we opted for abstract contract.

## 6.5 Choreography Interaction

The *fourth and final phase* deals with the self-generation of a JavaScript-based web interface. The web interface is modelled according to the public methods offered by the Smart Contracts and included on the first contract (Figure 4 a). The User Interface follows the progress of the flow expressed by the BPMN Sequence Flow presenting for each participant (respecting the played role) the right (the assignee one) method at the right time.

Each public method of the Smart Contract is invoked by means of the *Web3.js* library through the web app considering the flow described by Smart Contracts (Figure 4 a). The SC, depending on the active BPMN component, in turn, will invoke one of the methods of the Smart Contract Implementation side (Figure 4 b) which is described by its interface (Figure 4 c). Obviously, the web interface takes into account the participant who is about to consume the service enabling the proper methods according to the rules dictated by the Smart Contract.

# 7 Platform Implementation

This Chapter explains what Model Driven ChorChain 2.0 platform is and what is hidden underneath. The explanation is focused on the implementation side. In the first section, the platform architecture is introduced, while in the following sections, all the parts making up the architecture are discussed in detail.

## 7.1 Understanding the architecture

As previously introduced (5.1), the presented platform is a full stack-based type. This means, as shown in the Figure 5, that the platform is composed by three architectural layers: the *presentation layer* called **front-end**, the *business logic layer* called **back-end** and the **storage layer** called *database*. Furthermore, the architecture also provides another additional layer represented by the *blockchain layer*, namely **blockchain**. This layer has been added because the idea was to have a standalone application, as independent as possible, from an existing blockchain implementation (at least in the development phase).

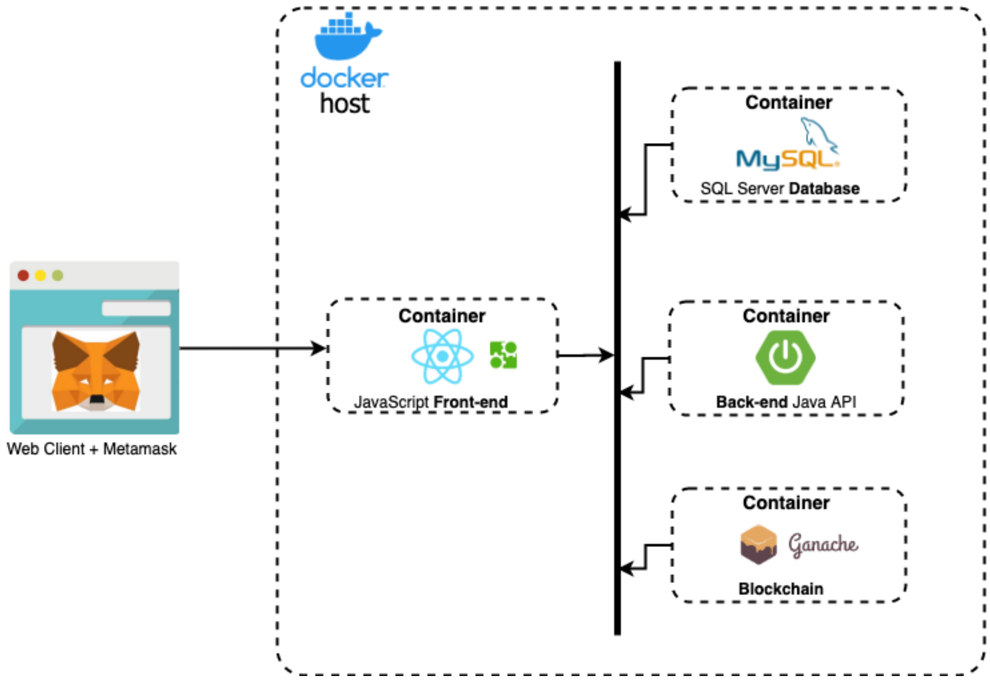


*Figure 5 - Platform Full Stack Architecture*

Based on the need to create a project easy to use, test and develop, a Docker[15] architecture has been adopted. The platform architecture is made up of containers following the logic "a container for each layer". In this way, each container can be easily configured or customized for its own purpose avoiding the need to tune the entire environment each time it is used or developed (which is a considerable effort). Hence, for convenience, a Docker image has been built providing a preconfigured version of ChorChain 2.0 Model Driven ready to be used.

Figure 6 shows the platform architecture, while Figure 5 shows the logical architecture implemented through Docker.

[15] https://www.docker.com

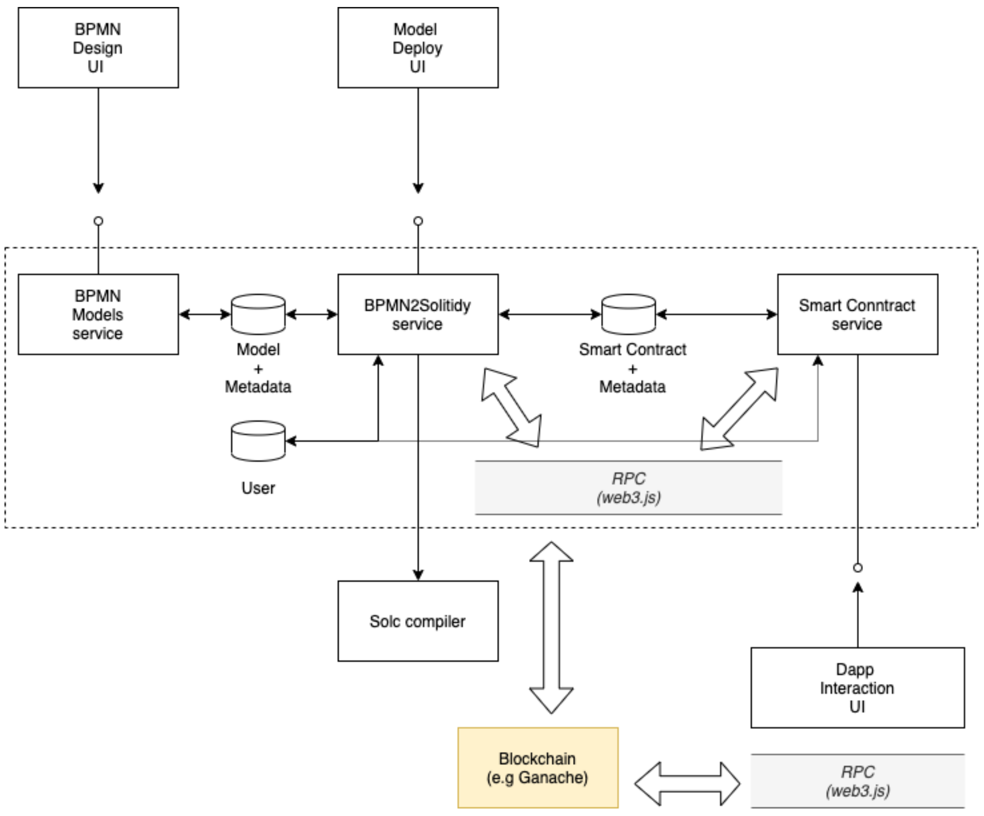


*Figure 6 - Model Driven ChorChain Architecture*

In Figure 6 the dashed rectangle encloses the core services, which are implemented as REST APIs, and represents the whole back end. The main platform services are respectively represented by the *BPMN Model service*, the *BPMN2Solidity service*, the *Smart Contract service* and one repository containing several entities such as *models*, *metadata*, *users*, *smart contracts* and *runtime data*. All those services are illustrated by means of API endpoints, indicated with empty dots in the above figure.

The *BPMN Models service* provides all CRUD operations and allows to manage the designed models. The BPMN model can be designed thanks to the *BPMN Design* user interface, which provides all the necessary tools. Once that the model is created, it can be stored into the database (in the current platform version the model is also stored on the file system for debugging and development reasons).

The *BPMN2Solidity service* is the key tool. It represents the mechanism responsible for the comprehensive mapping from BPMN to Solidity. In fact, given as input a BPMN model (XML format based), it generates a smart contract as output (Solidity based), which encapsulates the whole workflow related to the BPMN Sequence Flow logic of the processed model.

The BPMN2Solidty service also include (internally) a compiler tool, which is connected with an external compiler (`solc`), and a deploy tool, which is responsible for compiling the generated artefact. All of these actions, compilation and deploy, are made possible by the RPC protocol sublayer, that is implemented by means of the `web3.js` library. This library literally represents the bridge towards the blockchain. The compilation service is responsible for compiling the generated Smart Contract, while the deploy service is responsible for storing the successfully compiled Smart Contract and its metadata into the database.

The last module is the *Smart Contract service*. It is directly connected with the repository (database) and provides all the necessary operations to manage, control, and execute a processed model.

The model execution process, namely the Interaction, has its own dedicated user interface called *Dapp Interaction UI*. This UI provides all the features to allow users to interact among them by means of Smart Contracts. Actually, it is a JavaScript client connected to the blockchain via the web3 library. As highlighted, the blockchain instance is implemented by **ganache**, but it can be changed at will with other types, such as a public one based on Ethereum.

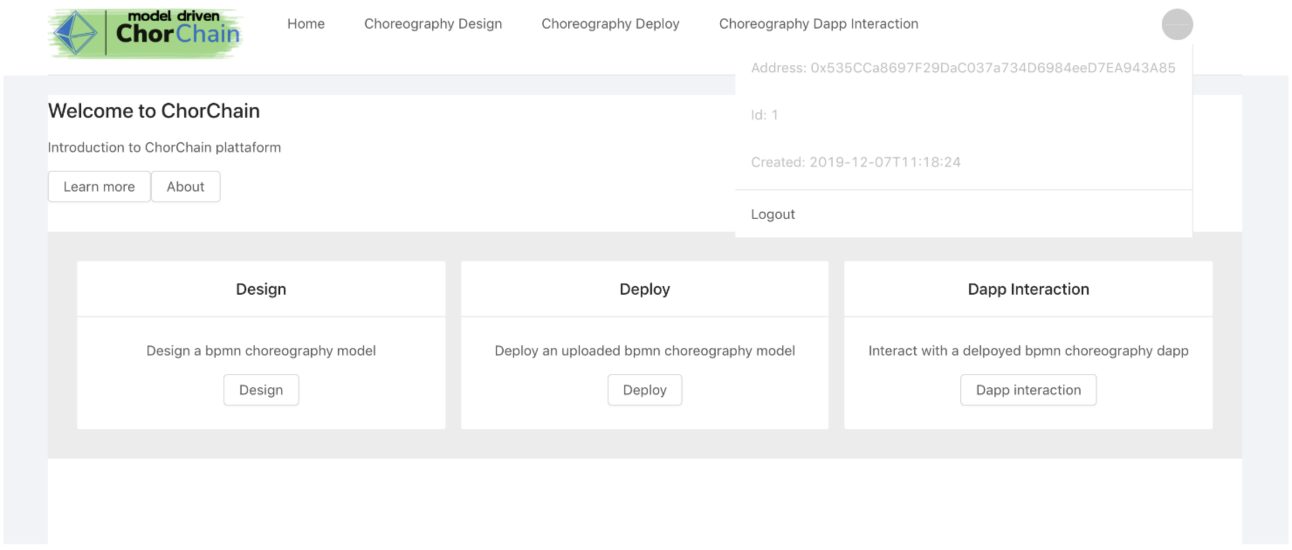
## 7.2 Front-end

The front-end (5.2) is the component that provides a user-friendly interface. It is mainly constituted of a JavaScript web application that runs on the client side and permits to deal with the platform services (API) after a prior authentication.

Together with the back-end, the front-end is the most complex component of the platform.

It is composed by many parts and provides several services. For instance, it allows users to access, design, deploy, and interact via Dapp in an authenticated way.

Let's introduce now the user workflow. After the authentication, the user is redirected to the platform Home Page (Figure 7), where all the important links to the respective sections are grouped (Choreography design, Choreography deploy and Choreography Dapp interaction).



Address: 0x535CCa8697F29DaC037a734D6984eeD7EA943A85

Id: 1

Created: 2019-12-07T11:18:24

Welcome to ChorChain

Introduction to ChorChain plattaform

Design

Design a bpmn choreography model

Deploy

Deploy an uploaded bpmn choreography model

Dapp Interaction

Interact with a delpoyed bpmn choreography dapp

*Figure 7 - Platform Home Page*

### 7.2.1 Technical overview

The front-end it is entirely developed with **ReactJs** technology that works sending HTTP requests to RESTful **endpoint** exposed through the back-end API (5.4). The user interface is entirely designed using `antd`[16], namely a React UI library containing a set of high-quality components building rich, interactive user interfaces.

The front-end is a standard ReactJs application, made by the `create-react-app` tool including the following libraries:

[16] https://ant.design

- `react-router`: popular library, is a collection of navigational components that compose declaratively with the application. Very useful whether the need is to have bookmarkable URLs for the web app;
- `axios`: a promise based HTTP client for the browser, used to make API endpoints calls;
- `react-refetch`: library that in simple, declarative, and composable way fetches data for React components;
- `chor-js, bpmn-js, diagram-js`: a complete suite library to enter into the BPMN world;
- `antd`: UI library, it allows to create rich, interactive user interfaces without having to write a lot of CSS;
- `web3.js`: used for communicating with the Ethereum blockchain. It effectively turns a React application into a blockchain-enabled application. In our case, we use Web3 to interact with the Smart Contracts.

### 7.2.2 Authentication

Unauthenticated users are not allowed in the platform prototype. In fact, unauthenticated users attempting to access the web application are redirected to a standard login form (Figure 8)requesting the user's credentials.

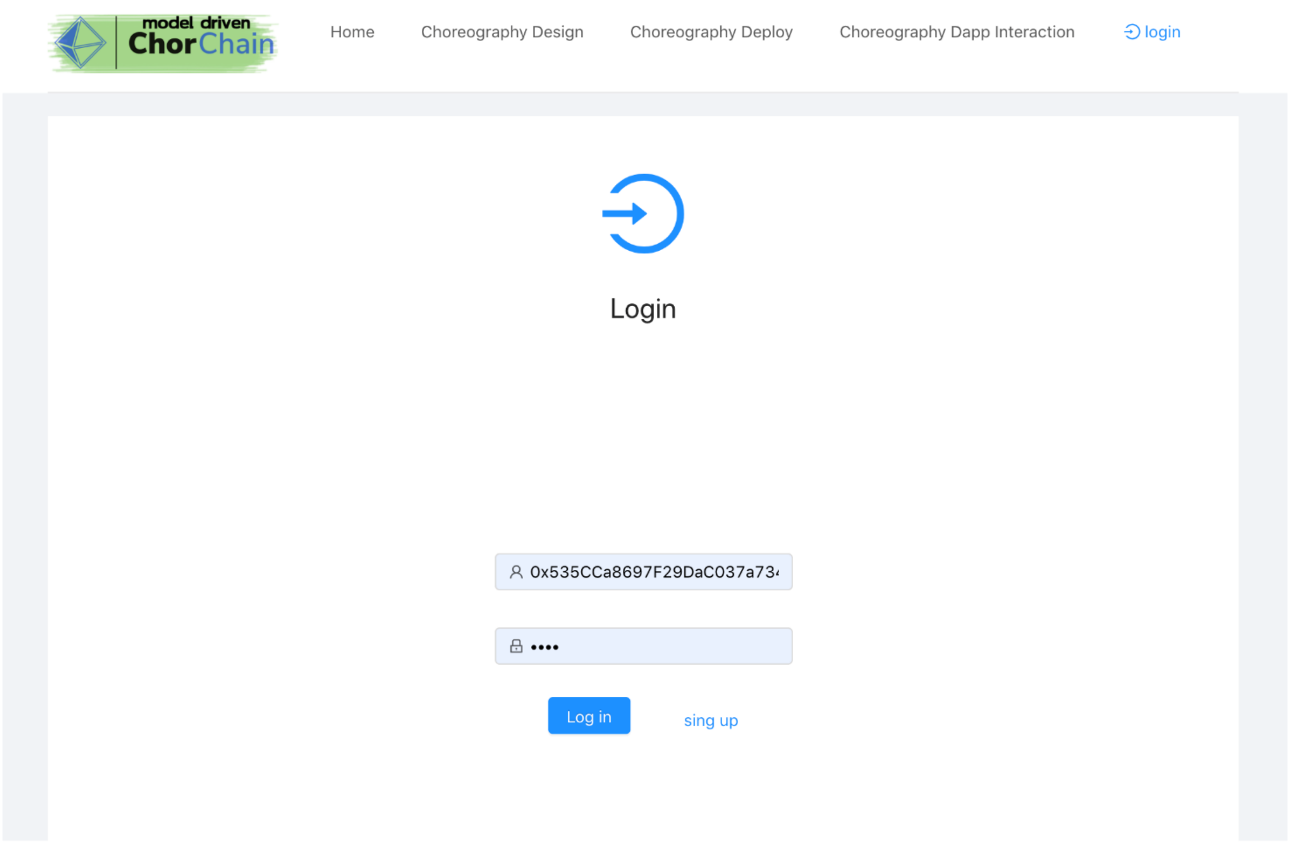


*Figure 8 - Platform user login*

Actually, during this first release, not much importance was given to security aspects, but they will be addressed later. However, there are two user categories:

- **local users**: registered users (stored in the database). This kind of users can only interact with choreographies in their entire lifecycles (4.1), except when a blockchain wallet interaction is required. They will be able for instance to design, create, store, search and deploy BPMN model, but not to make transactions with the blockchain.

- **blockchain users**: related to wallet blockchain users. These are the users that have tokens in the wallet and that can interact with the Dapp UI once the implemented smart contracts are released.

Normally blockchain users and local users should coincide, but currently this occurs only if they both register with the same username, otherwise they are actually two different users.

The platform development road map plans to unify these two user profiles and create a single user with both behaviours. In the actual release the front-end permits to deal with "local user" type only, exploiting the Spring Boot security integration module through the back-end API. Thanks to this module it should be easier profiling users and providing grained permission controls in the future, focusing on the user’s authorizations. Instead, with regards to "blockchain users", access and platform interactions are granted by MetaMask extension (5.7, 7.2.3).

In addition to these security aspects, the platform provides other common operations, such as *signup* which, for instance, allows a new user to be registered in the system and to make *logout*.

### 7.2.3 MetaMask

MetaMask (5.7) is a necessary browser extension. All users using this platform, must install the Chrome plugin, or the recommended one, in order to have their secure blockchain accounts directly on their browsers.

MetaMask allows the user to create a new wallet or restore it to an old one just from a seed phrase when it is opened for the first time. Both MetaMask and `ganache-cli` use the `bip44` standard for the wallet generation. Because of it, it is possible to restore Existing Vault by entering the mnemonic phrase that `ganache-cli` outputs when it starts it or if users have decided it in advance. Mnemonic phrase has to be 12 words long. For instance, our development mnemonic phrase used is the following:

```
include poem goose genuine baby flat mom token drama harsh sadness fit
```

To use MetaMask with the prototype ganache-cli accounts, it is first necessary to configure it. The ganache-cli address must be specified as that of the RPC provider, which by default is localhost: 8545. MetaMask provides a menu item to quickly switch to this local port. After setting up MetaMask as indicated above, it is possible to access the Dapps platform via MetaMask. After that, it is possible to see the MetaCoin balance of the first configured user account (Figure 9).

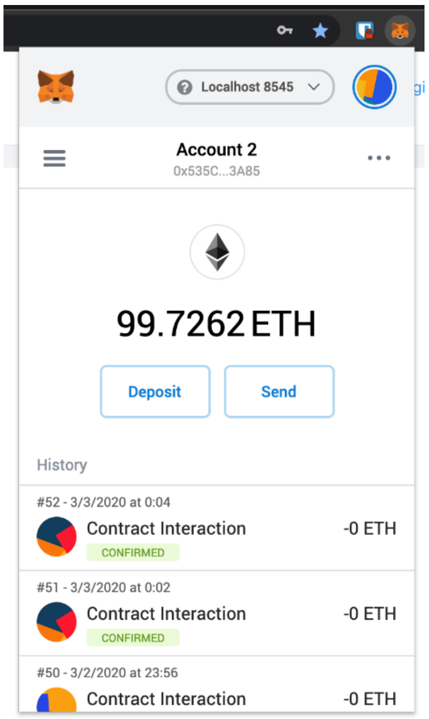


*Figure 9 - MetaMask account login example*

### 7.2.4 Choreography design editor

This is where users can design and transform a collaborative process into a BPMN model. Designing is made possible by a modeller, an editor included in the `chor-js` library (5.2.1). The `chor-js` library is like a big suite which makes easy the addition of (some) *basic functionalities* such as:

- *new*: creation of an empty diagram;
- *import*: importation of an existent diagram;
- *export*: exportation of the current diagram;
- *save svg*: saving in `.svg` format the current diagram;

In addition, we foresaw the addition of other important actions such as:

- *XML preview*: visualisation of the current XML generated by a modal window;
- *upload*: storage of the current BPMN model on the platform.

A strong work has been done to build an **enriched editor** which integrates the library functionalities with the custom features (Figure 10).

After a deep study of the Camunda `chor-js` and `bpmn-js` libraries, we experimented some solutions aiming at extending the XML based BPMN model and making it more detailed.

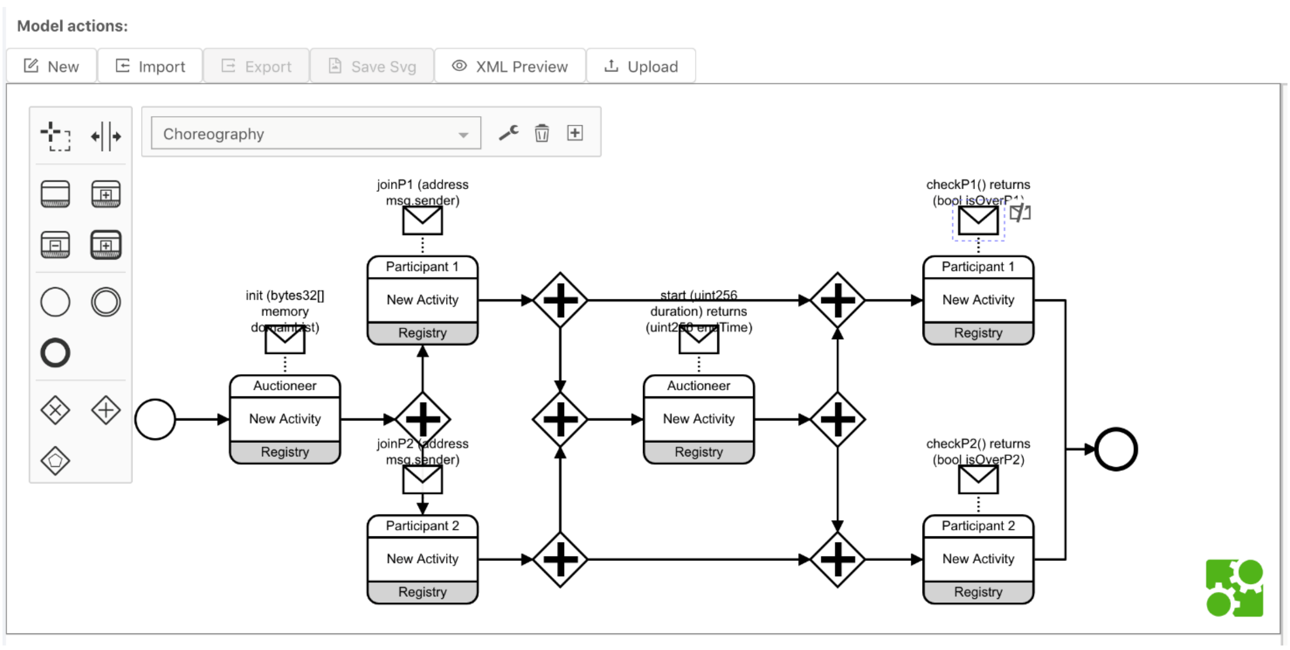

Model actions:
New
Import
Export
Save Svg
XML Preview
Upload
Choreography
joinP1 (address msg.sender)
checkP1() returns (bool isOverP1)
init (bytes32[] memory domainList)
start (uint256 duration) returns (uint256 endTime)
joinP2 (address msg.sender)
checkP2() returns (bool isOverP2)
Participant 1
Participant 2
Auctioneer
New Activity
Registry

*Figure 10 - BPMN model editor*

It's worth to remember that the platform's main goal is to design a BPMN choreography model able to be transformed in Smart Contracts based on Solidity. Each BPMN component, during the translation step, must be translated into a predefined Solidity sequence of code. Given that, it is very important transferring as much information as possible, collected from the design phase, into the XML document[17] before that the real translation take place. The information collection aims to better describe the collaborative process.

Therefore, in order to be as clear as possible, a **dedicated panel** has been created[18].

The panel allows the editing of the BPMN components properties.

To enrich all BPMN model components with meta-information, we exploited the *Camunda BPMN Extension Elements properties*[19],which is one of the many features offered by the library.

Thanks to this functionality we operated using two different approaches:

3. *Prototype approach*: the approach currently in use in the prototype. It considers as information vector only the message envelopes directly connected to the choreography task by mapping them as smart contract functions.

4. *Experimental approach*: it considers as information vector the whole BPMN model which carries information used during the generation of the smart contract business logic.

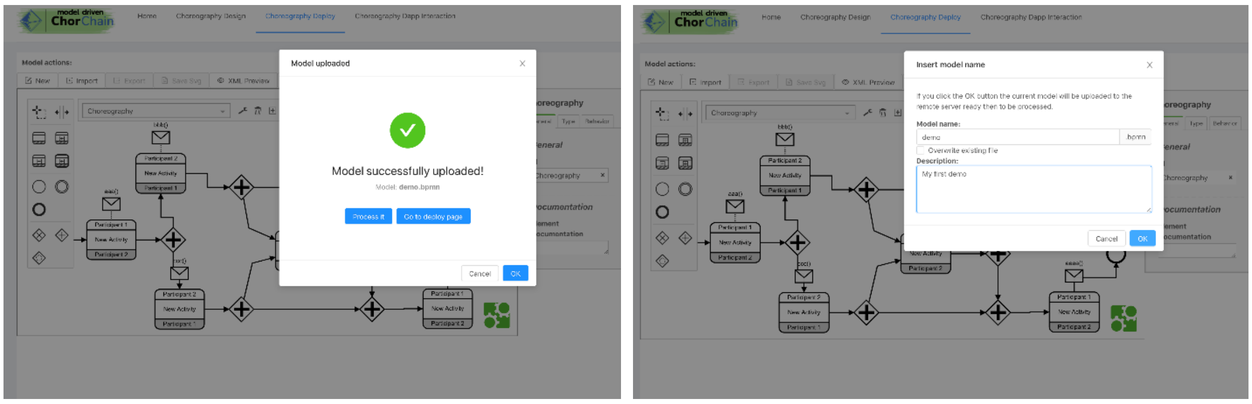
In addition to these functionalities, related to the design phase, we also added other important functions such as the model upload. **Uploading** the model means (Figure 11) storing the model in the platform (into the database) and starting its *life-cycle* (6.1). This is done only by specifying its

[17] Camunda BPMN is expressed by XML document

[18] https://github.com/bpmn-io/bpmn-js-example-react-properties-panel

[19] https://github.com/bpmn-io/bpmn-js-examples/tree/master/custom-elements

name and description and then uploading the model through the dedicated back-end API call. During the upload process the BPMN model is formally validated showing a success or failure feedback message to the user.


Model uploaded

Model successfully uploaded!

Model: demo.bpmn

Insert model name

If you click the OK button the current model will be uploaded to the remote server ready then to be processed.

Model name: demo .bpmn

Overwrite existing file

Description: My first demo

*Figure 11 - Uploading a model*

### 7.2.4.1 Prototype approach

This approach takes into account only the information related to the choreography task messages objects. Thanks to the creating *custom elements* feature provided by `bpmn-js` we are able to define *custom elements* into the BPMN (XML) model. Custom elements are ordinary BPMN 2.0 elements with domain-specific data, look, and feel. As described on `bpmn-js` documentation [10], the use cases for such elements includes:

- show certain elements in a distinct way
- restrict rules where a user can place elements on the diagram
- add data related to performance analytics such as KPI targets
- attach technical information related to model execution.

At the moment of writing, we have only defined a **new** single **element descriptor** (namely `ChorChain`) using the *attached "technical information related to model execution"* technique, as shown below.

Creating a new element descriptor means defining a new **XLS specification** and adding it to the definition of the object's XML namespaces. The new added objects declared in the descriptor can be recognized by the model itself.

```
{
  "name": "ChorChain",
  "prefix": "cc",
  "uri": "http://chorchain.com/schema/bpmn/cc",
  "associations": [],
  "types": [
    {
      "name": "signature",
      "superClass": [
        "Element"
      ],
      "properties": [
        {
          "name": "paramsType",
          "isAttr": true,
          "type": "String"
        },
        {
          "name": "paramsName",
          "isAttr": true,
          "type": "String"
        },
        {
          "name": "returnsType",
          "isAttr": true,
          "type": "String"
        },
        {
          "name": "returnsName",
          "isAttr": true,
          "type": "String"
        },
        {
          "name": "interfaceMethod",
          "isAttr": true,
          "type": "Boolean"
        }
      ]
    }
  ]
}
```

*Listing 1 – Definition of a new Element Descriptor*

In this case, the new descriptor `cc` describes all the properties and attributes of the (new) object type `signature`. The `cc` shows how this new element is represented and translated in a XML document. Thanks to the properties panel (Figure 12) and the descriptor itself, during the design process, it is possible generating BPMN models containing additional meta-data, as shown in the following Listing 2.

```
...
<bpmn2:message id="Message_037dl3h" name="offer1 (bytes32 oDomP1,
uint256 oAmountP1,address msg.sender) returns (string memory
oMsgP1,bool oFailP1,uint256 oCodeP1)">
    <bpmn2:extensionElements>
      <cc:signature
          paramsType="bytes32,uint256,address"
          paramsName="oDomP1,oAmountP1,msg.sender"
          returnsType="string memory,bool,uint256"
          returnsName="oMsgP1,oFailP1,oCodeP1"
          interfaceMethod="true"
          name="offer1"
          interfaceName="IRegistry"/>
    </bpmn2:extensionElements>
  </bpmn2:message>
...
```

*Listing 2 - Partial XML view representing the Signature tag properties contained in a BPMN model l*

In short, the `cc:signature` tag, child of the `bpmn2:extensionElements`, is used as vector of metadata aiming at enriching the Choreography Task with external information not strictly related to the BPMN model.

Hence the `Signature` tag in Listing 2 has been previously declared in the descriptor, the needed parameters useful to define a (future) **solidity signature method** are:

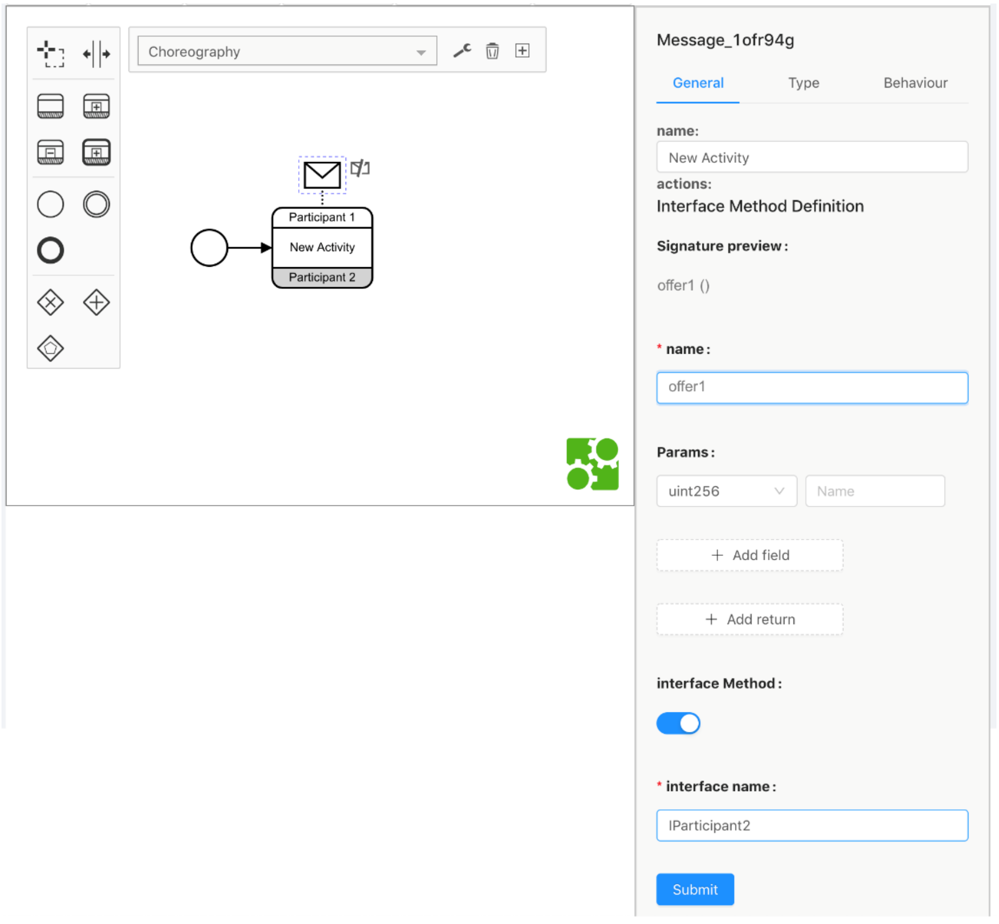
- *method name*: signature method name
- *parameter types*: array list containing the parameter types
- *parameter names*: array list containing the name of the previous parameter types
- *return types*: array list containing the return types
- *return names*: array list containing the name of the previous return types
- (possible) *interface name*: possible interface name (`true` value).

Obviously, the above property list can be easily extended at any time just adding new properties in in advance to the proper descriptor.

The translation of the `Signature` element into Solidity will result as follow:

```
offer1 (bytes32 oDomP1,uint256 oAmountP1,address msg.sender)
      returns (string memory oMsgP1,bool oFailP1,uint256 oCodeP1)}
```

*Listing 3 - Transformation of the Signature element offer1 into Signature Solidity element offer1*



Message_1ofr94g

name: New Activity

actions: Interface Method Definition

Signature preview: offer1 ()

* name: offer1

Params: uint256 Name

interface Method: ●

* interface name: IParticipant2

*Figure 12 - Custom properties panel used on the prototype approach*

In the right lower part of the property panel in Figure 12, the “interface” identifier is visible. At this point, a brief introduction on the **interface** (concept) property is necessary. Inside the properties panel and the descriptor, the option to consider the current method as an interface or not is given: the *offer1* function, previously defined, corresponds to the message envelope name linked to designed choreography task. This means that if the *offer1* method is considered as an **interface** (if interface method == `true`), then it is tagged and treated as stub method (4.4.1). Otherwise (if interface method == `false`), it is managed as a simple function[11].

To make this mechanism even more smarter we have also introduced additional *smart conditions*, such as the interpretation of the `msg.sender` parameter.

```
<bpmn2:message
  id="Message_0cmtbuo"
  name="offer2(uint256 oP2Amount, bytes32 oP2Dom, address msg.sender)
returns (string memory oP2Msg, bool oP2Fail, uint256 oP2Code)">
      <bpmn2:extensionElements>
            <cc:signature
            paramsType="uint256,bytes32,address"
            paramsName="oP2Amount,oP2Dom,msg.sender"
            returnsType="string memory, bool, uint256"
            returnsName="oP2Msg,oP2Fail, oP2Code"
            interfaceMethod="true"
            name="offer2"
            interfaceName="IRegistry"/>
      </bpmn2:extensionElements>
</bpmn2:message>
```

*Listing 4 - The msg.sender parameter*

In fact, we defined an additional parameter, called "`msg.sender`", of type address. If a parameter called "msg.sender" is defined in the properties panel, the translator is going to treat it as a reserved word, by transforming (later) the parameter as follows:

```
//Task(Bid): ChoreographyTask_06649j1 - TYPE: ONEWAY - offer2
function Message_0cmtbuo(uint256 oP2Amount, bytes32 oP2Dom) public
    checkOpt(optionalList[1])  {
            require(elements[position["Message_0cmtbuo"]].status == State.ENABLED);
            done("Message_0cmtbuo");
            currentMemory.oP2Amount = oP2Amount;
            currentMemory.oP2Dom = oP2Dom;
            (currentMemory.oP2Code,currentMemory.oP2Fail,currentMemory.oP2Msg) =
            iregistry.offer2(msg.sender,oP2Amount,oP2Dom);
            enable("ExclusiveGateway_0cw6nha");
            ExclusiveGateway_0cw6nha();
      }
```

*Listing 5 - msg.sender parameter behaviour during the Solidity transformation*

At the beginning, the `msg.sender` parameter is ignored by the translator which will not include it between the Solidity function parameters (line 2), but later it will associate it in the body function among the parameters of the **stub method** (line 9).

#### 7.2.4.2 Experimental approach

During the platform development we also tried out other solutions aiming at enriching as much as possible the XML based BPMN model. The experimental one, discussed in this chapter, is focused on **adding business logic through the editor** directly into the model. The strategy used is similar to the previous one, focused on the extension of the BPMN elements, though more complex. Because of this complexity we are going to take advantage and inspiration of the **Camunda Form mechanism**[20] already supported by the library, but actually used for other purposes.

The basic idea is creating a dedicated set of tags and properties useful for injecting technical information and parts of business logic into each BPMN elements in the whole choreography.

Given that we are able to inject *global variables*, *structures* and others constructs through the *Choreography Element*, we want to leverage the same mechanism to add *business logic events* (to emit) and other generic elements to represent Solidity constructs. To achieve this, a possible solution could be the adoption of "predefined code snippets". These code snippets, previously written and uploaded on the client side, should be then selectable by users via the editor during the model design process and transported within the BPMN model.

Below some real use cases are reported.

**Structs definition through Choreography Element**: through the model editor and the properties panel, after selecting the Choreography element by clicking on the model background, it is possible to choose between three tabs - General, Types and Behaviours (Figure 13). Choosing the second tab, *Type*, the user can proceed with the generation of global declarations such as *struct*, *global*, *mappings*, *custom type* into the future smart contract that is going to be created.

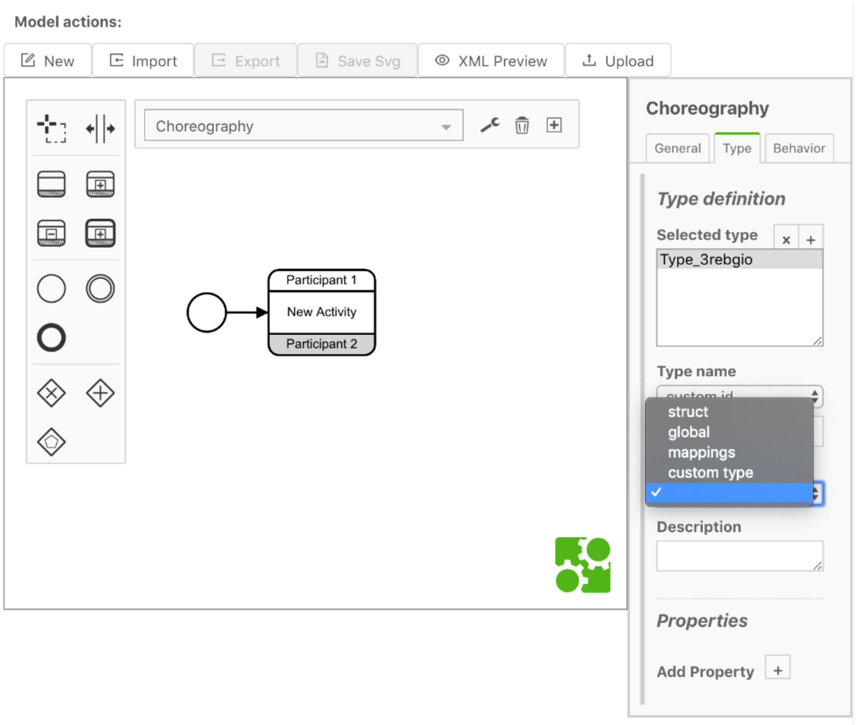
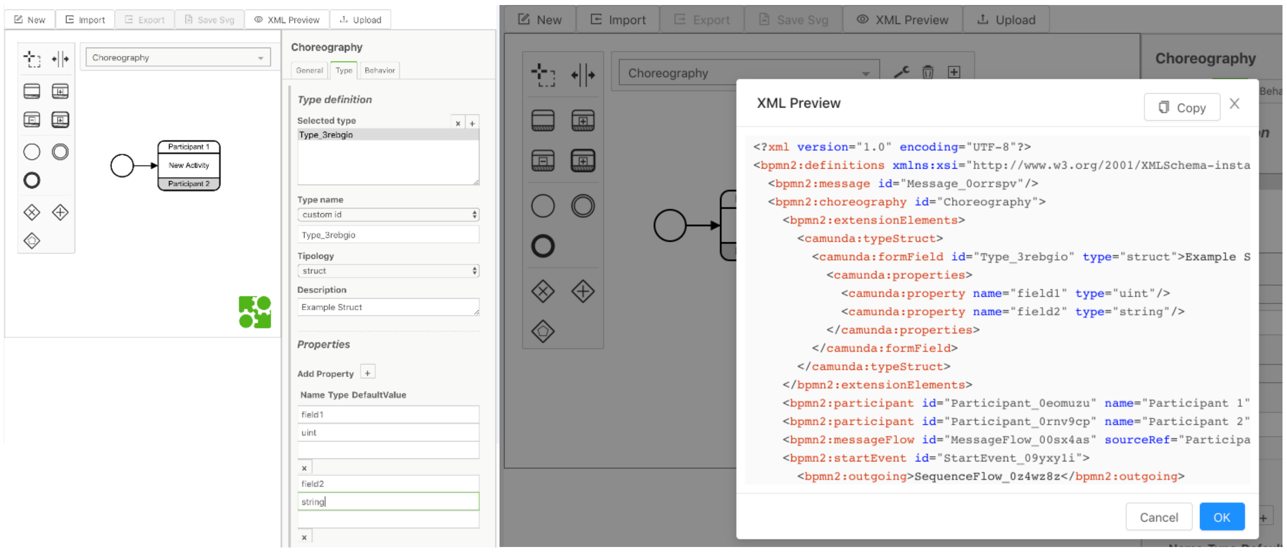


*Figure 13 - Type definition by Choreography Element*

A concrete example is shown below. It is about the definition of a new `struct`. In order to do it, the user must only choose the *struct* option in the Typology selector. Afterwards, the user must give some additional information such as the structure name, the description and parameters (Properties) by adding them through the + (Add property) button. The result is presented in the XML

[20] https://github.com/camunda-consulting/code/tree/master/snippets/camunda-modeler-plugins/camunda-modeler-plugin-usertask-generatedform-preview

Preview box (Figure 14). It shows how the injected information via the properties panel is added into the model thanks to the Camunda Form mechanism.



Choreography

*Type definition*

Selected type: Type_3rebgio

Type name: custom id

Type_3rebgio

Tipology: struct

Description: Example Struct

*Properties*

Add Property

Name Type DefaultValue

field1
uint

field2
string

XML Preview

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn2:definitions xmlns:xsi="http://www.w3.org/2001/XMLSchema-insta
  <bpmn2:message id="Message_0orrspv"/>
  <bpmn2:choreography id="Choreography">
    <bpmn2:extensionElements>
      <camunda:typeStruct>
        <camunda:formField id="Type_3rebgio" type="struct">Example S
          <camunda:properties>
            <camunda:property name="field1" type="uint"/>
            <camunda:property name="field2" type="string"/>
          </camunda:properties>
        </camunda:formField>
      </camunda:typeStruct>
    </bpmn2:extensionElements>
    <bpmn2:participant id="Participant_0eomuzu" name="Participant 1"
    <bpmn2:participant id="Participant_0rnv9cp" name="Participant 2"
    <bpmn2:messageFlow id="MessageFlow_00sx4as" sourceRef="Participa
    <bpmn2:startEvent id="StartEvent_09yxy1i">
      <bpmn2:outgoing>SequenceFlow_0z4wz8z</bpmn2:outgoing>
```

*Figure 14 - Struct definition by Choreography Element, XML preview*

The final mapping in Solidity will be:

```
//Example struct
struct Type_3rebgio {
    uint   field1,
    string field2
}
```

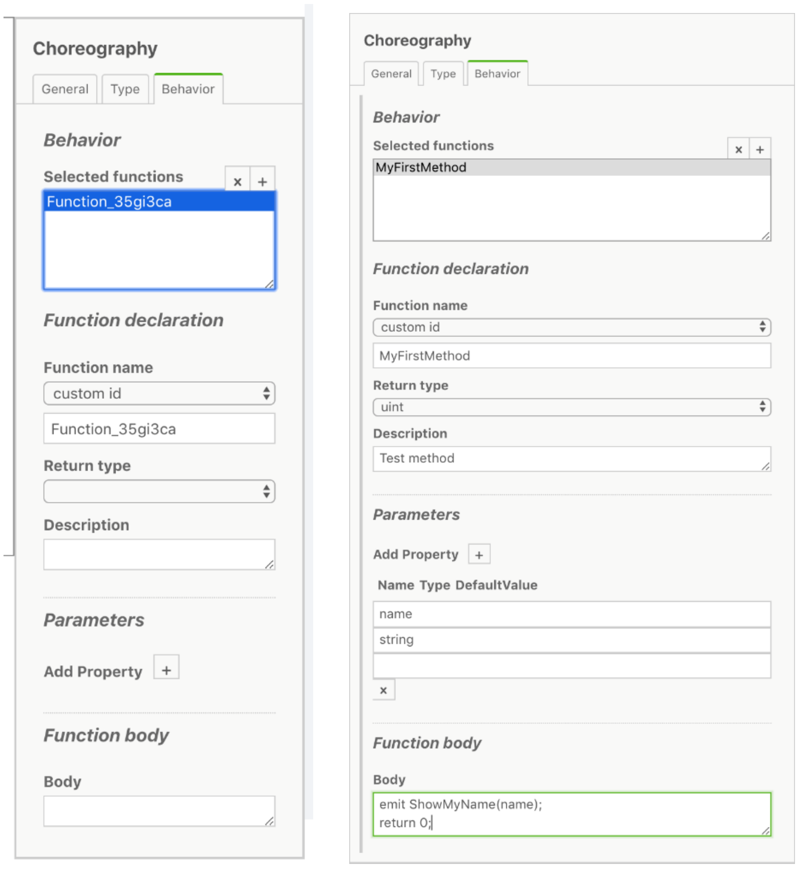
*Listing 6 - Solidity transformation of the struct*

**Behaviour definition through Choreography Element**: similar as before, after selecting the Choreography element by clicking on the model background, the user can choose the *Behaviour* tab. As shown in Figure 15, the user can add a new behaviour by clicking on the + button close to the "selected functions" row. Afterwards, the user can proceed with the definition of the parameters such as name, description and body.

For instance, in the following examples, the body definition is:

```
emit ShowMyName(name);
return 0;
```

This Solidity code snippet will be included roughly in the SC instance resulting from the transformation. In Figure 16, it is shown how all the metadata injected to the BPMN model are expressed into the XML document.


Choreography

General | Type | Behavior

*Behavior*

Selected functions × +

Function_35gi3ca

*Function declaration*

Function name

custom id

Function_35gi3ca

Return type

Description

*Parameters*

Add Property +

*Function body*

Body

Choreography

General | Type | Behavior

*Behavior*

Selected functions × +

MyFirstMethod

*Function declaration*

Function name

custom id

MyFirstMethod

Return type

uint

Description

Test method

*Parameters*

Add Property +

Name Type DefaultValue

name

string

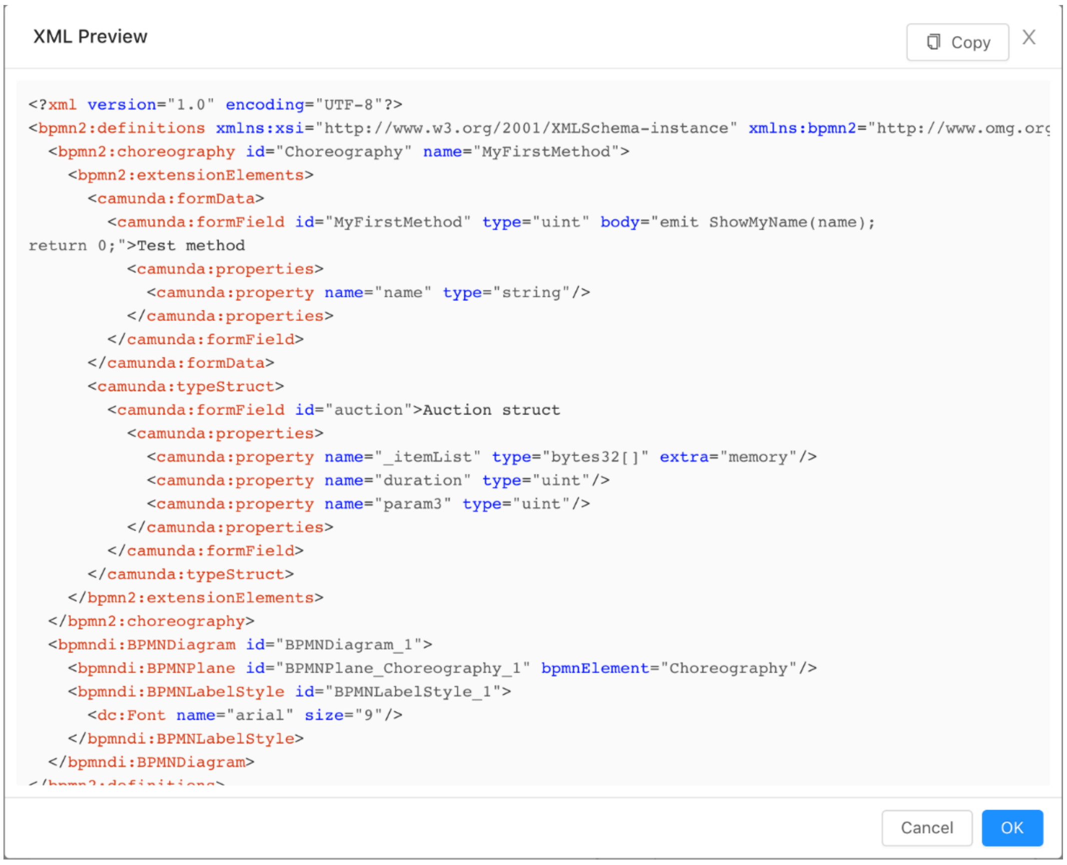
×

*Function body*

Body

```
emit ShowMyName(name);
return 0;
```

*Figure 15 - Behaviour definition by Choreography Element*


XML Preview

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn2:definitions xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:bpmn2="http://www.omg.or
  <bpmn2:choreography id="Choreography" name="MyFirstMethod">
    <bpmn2:extensionElements>
      <camunda:formData>
        <camunda:formField id="MyFirstMethod" type="uint" body="emit ShowMyName(name);
return 0;">Test method
          <camunda:properties>
            <camunda:property name="name" type="string"/>
          </camunda:properties>
        </camunda:formField>
      </camunda:formData>
      <camunda:typeStruct>
        <camunda:formField id="auction">Auction struct
          <camunda:properties>
            <camunda:property name="_itemList" type="bytes32[]" extra="memory"/>
            <camunda:property name="duration" type="uint"/>
            <camunda:property name="param3" type="uint"/>
          </camunda:properties>
        </camunda:formField>
      </camunda:typeStruct>
    </bpmn2:extensionElements>
  </bpmn2:choreography>
  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
    <bpmndi:BPMNPlane id="BPMNPlane_Choreography_1" bpmnElement="Choreography"/>
    <bpmndi:BPMNLabelStyle id="BPMNLabelStyle_1">
      <dc:Font name="arial" size="9"/>
    </bpmndi:BPMNLabelStyle>
  </bpmndi:BPMNDiagram>
</bpmn2:definitions>
```

*Figure 16 - Behaviour definition by Choreography Element XML preview*

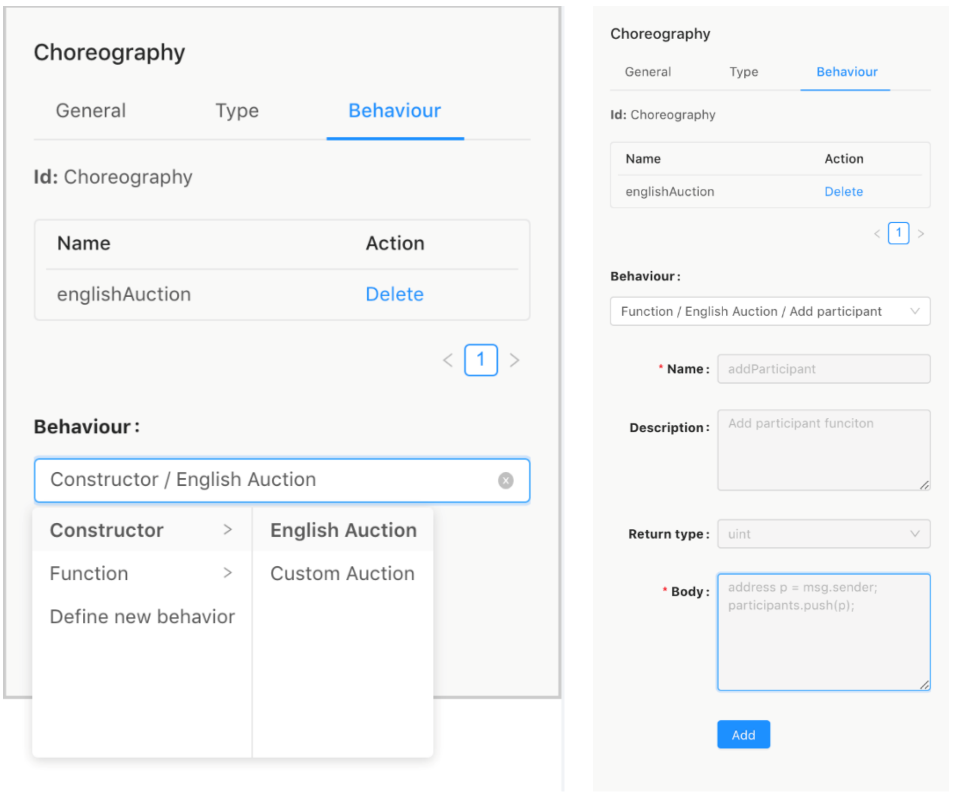
The final complete mapping in Solidity is:

```
//Example struct
function MyFirstMethod(string memory name) return (uint){
    emit ShowMyName(name);
    return 0;
}
```

**Defining behaviour with a predefined function via Choreography Element**: this is another good example showing the use of predefined behaviours.

In this case, the basic idea is providing a set of predefined functions via panel properties to the users. These functions should guide and simplify the user construction of the BPMN model from a Smart Contract perspective. In fact, the user can decide which business logic to include in the future SC for a specific method only choosing from a set of atomic functions related to a particular domain. To achieve this, the administrator must declare in the first instance a function descriptor containing all the necessary specifications. Then, after having uploaded these files containing the descriptors on the server,and the (pre-defined) functions will magically appear on the property panel. In the example below (Figure 17) a panel showing all the behaviour possibilities already loaded in the server is shown. As we can see, the user has different options, such as *constructor*, *functions* or *custom functions* (in this case it is possible inject code into a specific function through the "Define new behaviour" voice. In the figure, two different constructors are already defined (right side of the figure) corresponding to auctions constructors. One constructor represents the English type auction and the other one can is for a Custom auction. The "*englishAuction*" constructor (right side of the figure) was previously added, as shown, on the method list in figure (left side of the figure).

In the right side of the Figure 17, all the details of the chosen function, called *Add Participant*, are shown.



Id: Choreography

| Name | Action |
|---|---|
| englishAuction | Delete |

Behaviour: Constructor / English Auction

Id: Choreography

| Name | Action |
|---|---|
| englishAuction | Delete |

Behaviour: Function / English Auction / Add participant

* Name: addParticipant

Description: Add participant funciton

Return type: uint

* Body:

```
address p = msg.sender;
participants.push(p);
```

*Figure 17 - Preloaded functions example*

The XML result of the *Add Participant* function added using the editor is:

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn2:definitions xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:bpmn2="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" x
mlns:xs="http://www.w3.org/2001/XMLSchema"
id="_tTv5YOycEeiHGOQ2NkJZNQ" targetNamespace="http://www.signavio.com">
  <bpmn2:choreography id="Choreography">
    <bpmn2:extensionElements>
      <camunda:formData>
        <camunda:formField id="function_134df" defaultValue="false"
        body="addresd p =msg.sender;participants.push(p);return 0;"
name="addParticipant"
        returnType="uint">
        Add participant function
       </camunda:formField>
      </camunda:formData>
    </bpmn2:extensionElements>
  </bpmn2:choreography>
  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
    <bpmndi:BPMNPlane id="BPMNPlane_Choreography_1" bpmnElement="Choreography" />
    <bpmndi:BPMNLabelStyle id="BPMNLabelStyle_1">
      <dc:Font name="arial" size="9" />
    </bpmndi:BPMNLabelStyle>
  </bpmndi:BPMNDiagram>
</bpmn2:definitions>
```

Listing 7 - Example of using Camunda Form to transport predefined functions

The final result in Solidity is:

```
//Add participant function
function addParticipant(){
      address p = msg.sender;
      participants.push(p);
      return 0;
}
```

Listing 8 - Transformation in Solidity of a predefined function - AddParticipant

Now let’s give a brief look at the example descriptor of the predefined function *Add Participant uploaded to the server*.

```
[
  {
    "name": "addParticipant",
    "value": "addParticipant",
    "description": "Add a participant function",
    "returns": [
      {
        "name": "",
        "type": "uint",
        "extra": ""
      }
    ],
    "dependencies": {
      "vars": [
        "participants"
      ],
      "functions": [
        "createLotAuction"
      ]
    },
    "body": "address p = msg.sender();participants.push(p);return 0;"

  },
]
```

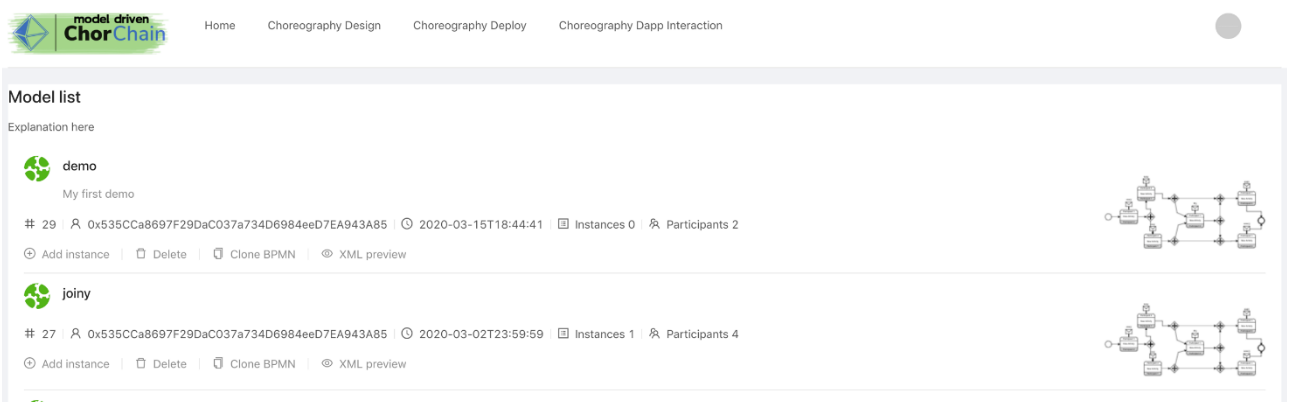
*Listing 9 - Experimental descriptor of a predefined function - AddParticipant*

It is important to point out that this is an *experimental descriptor*. These aspects are still under study and therefore the examples are incomplete experimental drafts: they are only used to test and understand if the way chosen is the right one.

The descriptor above is part of the predefined functions set and specifically defines how the AddParticipant function is composed. On line 12 of the Listing 9, there is an important property called “dependencies". The purpose of this property is to create a dependency list for each function, and it is useful both for validating the model and understanding which functions, types, structs, etc. are necessary. The dependency list can have many uses, for instance can help in case of validation check before storing the model. The tool can, infact, calculate if all the dependencies are satisfied or not and, in case, can warn the user.

### 7.2.5 Choreography deploy page

In this section all the actions possible with stored BPMN models are presented. Thanks to a dedicated web page, called “Choreography Deploy”, users can handle all the stored BPMN models. Users can view paginated list of models, add a new model instance, clone models or preview their XML content and, finally, delete BPMN models. There are still other features, such as models searching using search criteria, which are not yet implemented but planned for the next future.



Model list

Explanation here

demo

My first demo

# 29 | 0x535CCa8697F29DaC037a734D6984eeD7EA943A85 | 2020-03-15T18:44:41 | Instances 0 | Participants 2

Add instance | Delete | Clone BPMN | XML preview

joiny

# 27 | 0x535CCa8697F29DaC037a734D6984eeD7EA943A85 | 2020-03-02T23:59:59 | Instances 1 | Participants 4

Add instance | Delete | Clone BPMN | XML preview

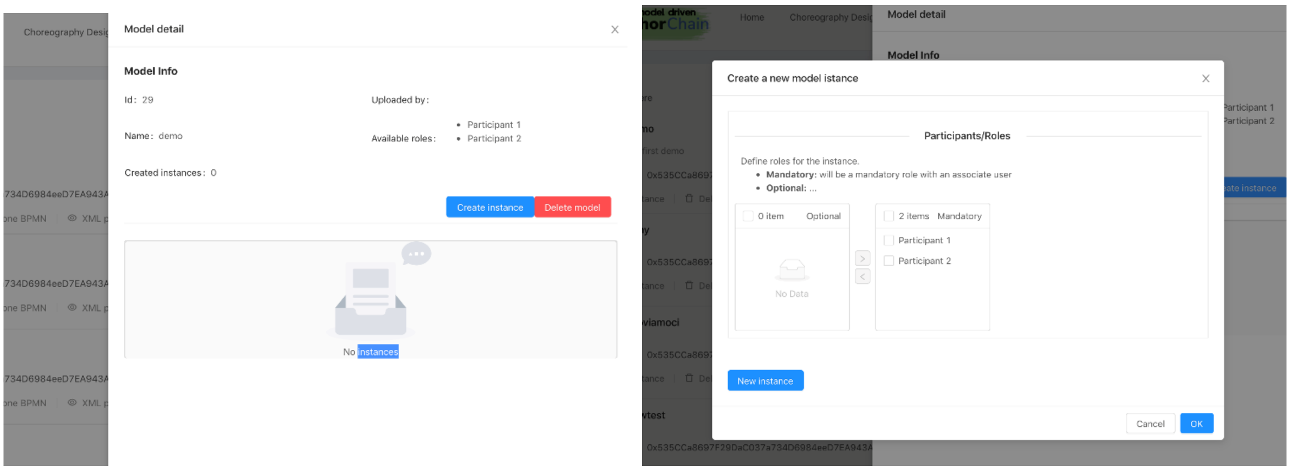
*Figure 18 - Choreography deploy page*

The Choreography deploy the UI, which shows, as a default view, a list of stored models. Each row in the list links some actions related to the corresponding model, besides summarising important model information (Figure 18) such as:

- *name*: the choreography model name;
- *description*: the choreography model description;
- *model* id: the entity id provided by the database;
- *creation user*: the user who created and stored the model identified by his blockchain user address;
- *creation date*: date of creation;
- *instances number*: number of model instances created by users;
- *participants number*: number of defined choreography participants;
- *model thumbnail*: a small box showing the model thumbnail. By clicking on that will be open a modal window displaying a bigger model preview.

#### 7.2.5.1 Model Instance creation

After a BMPN model has been created (6.1), it is possible to generate a new Model Instance representing the related Smart Contract. This means that each Model Instance has its own linked Smart Contract. To proceed with the Instance creation the user must click on the choreography model name and access to the section of the model detail. Clicking on the "create instance" button, a modal window is opened and the user can then associate roles (mandatory/optional) and participants by guiding users interaction. These roles will be a future requirement for the model instance deployment.


Model detail

Model Info

Id: 29

Uploaded by:

Name: demo

Available roles:
- Participant 1
- Participant 2

Created instances: 0

No instances

Create a new model istance

Participants/Roles

Define roles for the instance.
- **Mandatory:** will be a mandatory role with an associate user
- **Optional:** ...

0 item Optional

No Data

2 items Mandatory

Participant 1

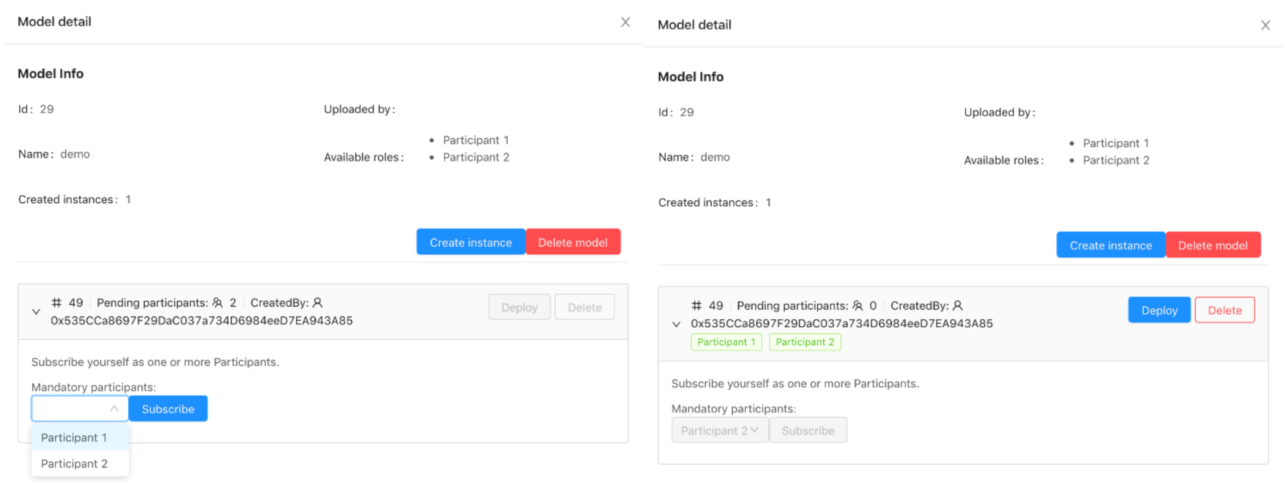
Participant 2

*Figure 19 - Model Instance creation detail*

The "new instance" button triggers the next stage of the choreography life-cycle (6.3). In fact, if all requirements are ok, a new Model Instance is created, otherwise an error message will appear reporting the cause of the denial. Once the creation is successfully, the Model Instance is added to the empty list displayed in Figure 15. For each new choreography instance, a row is shown (Figure 20) providing the following features:

- *instance id*: the instance id provided by the database;
- *total participants*: the total numbers of choreography participants;
- *creation user*: the user who has created the choreography instance;
- *contract blockchain address*: the contract blockchain address resulted from the contract compilation;
- *deploy button action*: the button enabling the deploy of the contract on the blockchain;
- *delete button action*: the button deleting the current choreography instance;

- *association roles/participants form*: the form aiming at permitting the association between participants and roles. The association considers only logged users, and this means that each chosen role will be associated with the current user.


Model detail

**Model Info**

Id: 29
Uploaded by:
Name: demo
Available roles:
- Participant 1
- Participant 2

Created instances: 1

Create instance | Delete model

# 49 | Pending participants: 2 | CreatedBy:
0x535CCa8697F29DaC037a734D6984eeD7EA943A85
Deploy | Delete

Subscribe yourself as one or more Participants.
Mandatory participants:
Subscribe
Participant 1
Participant 2

Model detail

**Model Info**

Id: 29
Uploaded by:
Name: demo
Available roles:
- Participant 1
- Participant 2

Created instances: 1

Create instance | Delete model

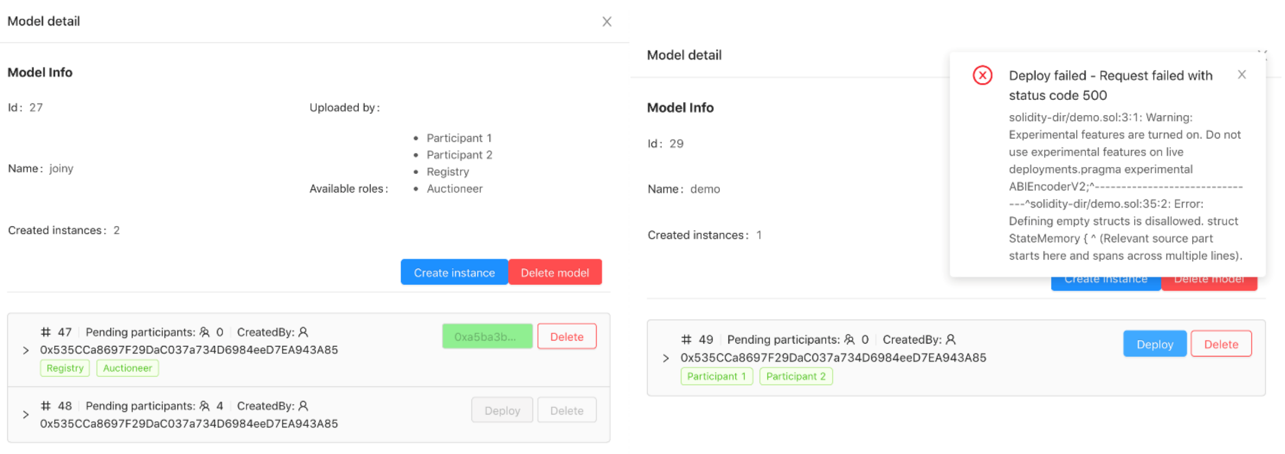
# 49 | Pending participants: 0 | CreatedBy:
0x535CCa8697F29DaC037a734D6984eeD7EA943A85
Participant 1 | Participant 2
Deploy | Delete

Subscribe yourself as one or more Participants.
Mandatory participants:
Participant 2 | Subscribe

*Figure 20 - Instance row detail*

### 7.2.5.2 *Model instance deploy*

At this point, the user must decide which are the roles he wants to subscribe to; for each associated role a green tag is shown. To proceed with the deploy process all the mandatory roles must first be subscribed by the participants (even by different users). When all mandatory roles have been associated, the "deploy" button will be automatically enabled (right side of the Figure 20). Once the deploy button has been clicked, the back-end API is invoked, and the deploy process starts. If the deploy operation is successful, a confirmation message is displayed, while the button turns green displaying the contract address (left side Figure 21). Otherwise, if the deploy is not successful, an error notification appears summarising the causes given by the compiler (right side Figure 21).


Model detail

**Model Info**

Id: 27
Uploaded by:
Name: joiny
Available roles:
- Participant 1
- Participant 2
- Registry
- Auctioneer

Created instances: 2

Create instance | Delete model

# 47 | Pending participants: 0 | CreatedBy:
0x535CCa8697F29DaC037a734D6984eeD7EA943A85
Registry | Auctioneer
0xa5ba3b... | Delete

# 48 | Pending participants: 4 | CreatedBy:
0x535CCa8697F29DaC037a734D6984eeD7EA943A85
Deploy | Delete

Model detail

**Model Info**

Id: 29
Name: demo
Created instances: 1

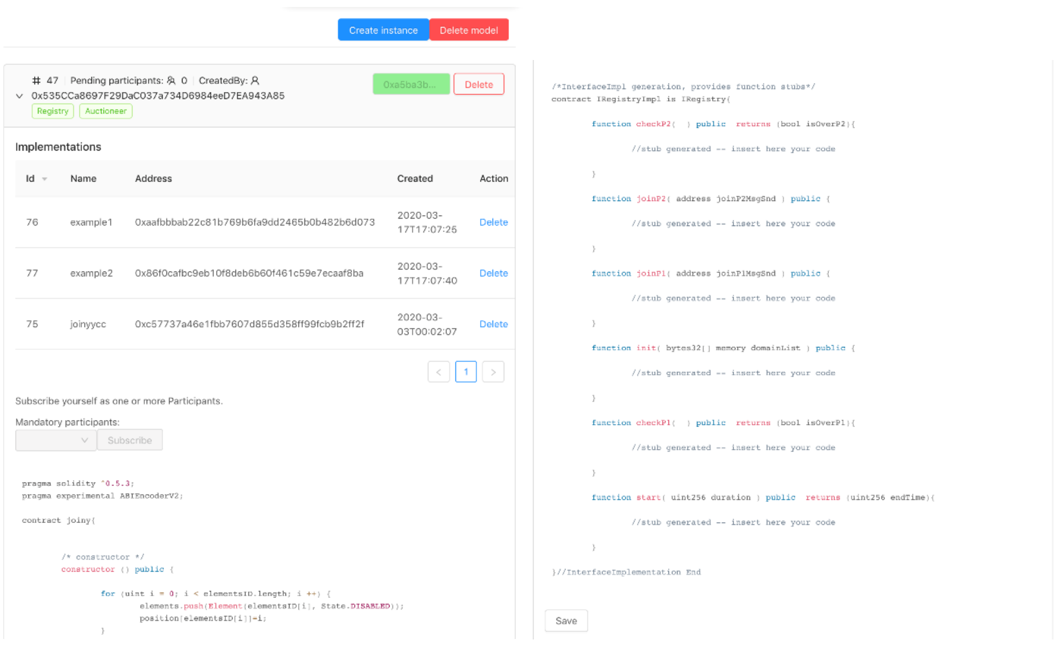
```
Deploy failed - Request failed with status code 500
solidity-dir/demo.sol:3:1: Warning: Experimental features are turned on. Do not use experimental features on live deployments.pragma experimental ABIEncoderV2;^------------------------------^solidity-dir/demo.sol:35:2: Error: Defining empty structs is disallowed. struct StateMemory { ^ (Relevant source part starts here and spans across multiple lines).
```

Create instance | Delete model

# 49 | Pending participants: 0 | CreatedBy:
0x535CCa8697F29DaC037a734D6984eeD7EA943A85
Participant 1 | Participant 2
Deploy | Delete

*Figure 21 - Examples of deploy result, successful and failure*

A successful deployed instance, corresponding to skeleton Smart Contract (6.4), will be the starting point for the creation of next Implementations. Users can edit the generated skeleton Instance and submit their customization aiming at distributing a personal "model instance implementation" through the blockchain. Each new implementation is added to the implementation list (Implementations) contained in the instance row details. Thanks to the editor, users can directly edit the skeleton SC (Figure 22). All the customisations made by adding the business logic to the stub methods early generated are then stored by clicking the Save button.



# 47 | Pending participants: 0 | CreatedBy:
0x535CCa8697F29DaC037a734D6984eeD7EA943A85

Implementations

| Id | Name | Address | Created | Action |
|---|---|---|---|---|
| 76 | example1 | 0xaafbbbab22c81b769b6fa9dd2465b0b482b6d073 | 2020-03-17T17:07:25 | Delete |
| 77 | example2 | 0x86f0cafbc9eb10f8deb6b60f461c59e7ecaaf8ba | 2020-03-17T17:07:40 | Delete |
| 75 | joinyycc | 0xc57737a46e1fbb7607d855d358ff99fcb9b2ff2f | 2020-03-03T00:02:07 | Delete |

Subscribe yourself as one or more Participants.

Mandatory participants:

```
pragma solidity ^0.5.3;
pragma experimental ABIEncoderV2;

contract joiny{

        /* constructor */
        constructor () public {

                for (uint i = 0; i < elementsID.length; i ++) {
                        elements.push(Element(elementsID[i], State.DISABLED));
                        position[elementsID[i]]=i;
                }
```

```
/*InterfaceImpl generation, provides function stubs*/
contract IRegistryImpl is IRegistry{

        function checkP2(  ) public  returns (bool isOverP2){

                //stub generated -- insert here your code

        }

        function joinP2( address joinP2MsgSnd ) public {

                //stub generated -- insert here your code

        }

        function joinP1( address joinP1MsgSnd ) public {

                //stub generated -- insert here your code

        }

        function init( bytes32[] memory domainList ) public {

                //stub generated -- insert here your code

        }

        function checkP1(  ) public  returns (bool isOverP1){

                //stub generated -- insert here your code

        }

        function start( uint256 duration ) public  returns (uint256 endTime){

                //stub generated -- insert here your code

        }

}//InterfaceImplementation End
```

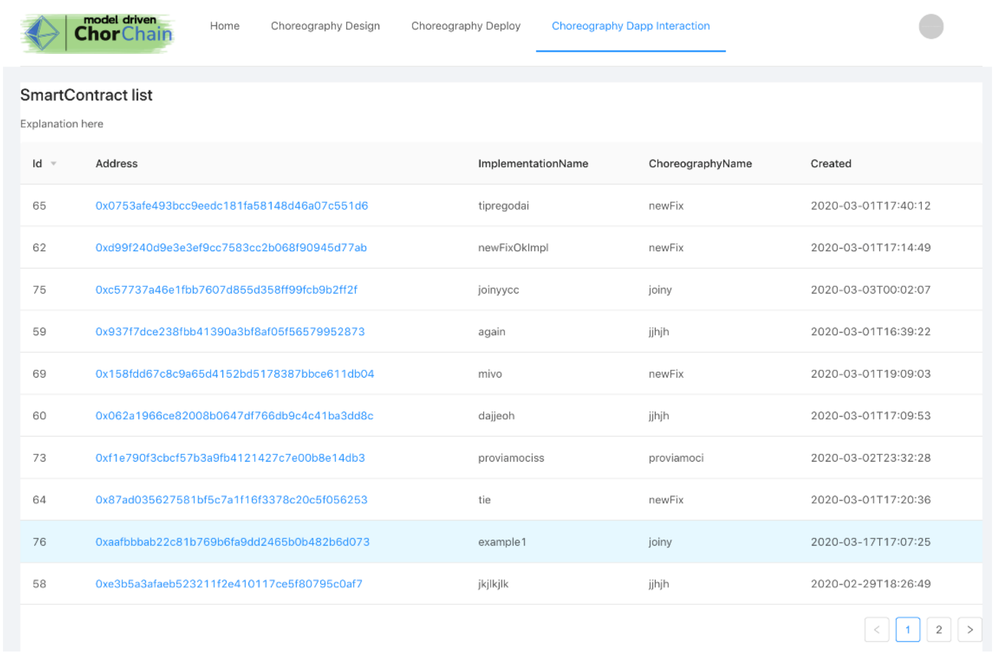
*Figure 22 – Example of creating Model instance implementation*

### 7.2.6 Choreography interaction

Coreography interaction represents the last UI section dedicated to choreography lifecycle. From here users can access the list of model instance implementations previously generated and already deployed on the blockchain. Access to this page is allowed only to a blockchain user (7.2.2, 7.2.3), hence the installation of MetaMask is mandatory. At the time of writing, the development of this section is not very advanced and therefore it lacks a bit in graphical details and functionalities (Figure 23 and Figure 24). The goal of the Interaction page was to offer a UI allowing a simplified interaction to participants through a collaborative process described by a graphical choreography model where users have their assigned role.

Once a participant selects the address corresponding to the SC of the Model Implementation chosen (Figure 23), the interaction page is opened. The interaction page shown in Figure 24 is composed of:

- *User account*: the blockchain user address;
- *Optional Role*: the free optional roles to be associated with;
- *Contract*: the contract information name and address (extendible with much more information, for instance the model name, the author, the instance name, etc.);
- *Viewer*: a BPMN viewer showing the progress of the process;
- *Current operation panel*: the current user interaction that is performed;
- *Current state*: the smart contract state by displaying the global variables values;
- *Smart contract viewer code*: the formatted Solidity code representing the smart contract.



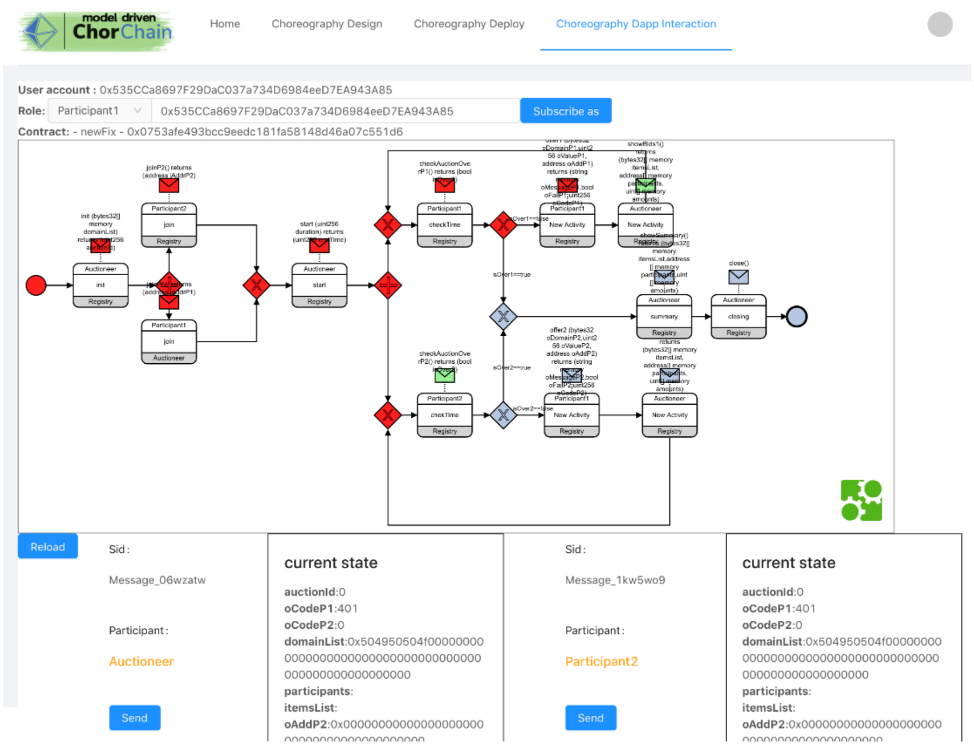
SmartContract list

Explanation here

| Id | Address | ImplementationName | ChoreographyName | Created |
|---|---|---|---|---|
| 65 | 0x0753afe493bcc9eedc181fa58148d46a07c551d6 | tipregodai | newFix | 2020-03-01T17:40:12 |
| 62 | 0xd99f240d9e3e3ef9cc7583cc2b068f90945d77ab | newFixOkImpl | newFix | 2020-03-01T17:14:49 |
| 75 | 0xc57737a46e1fbb7607d855d358ff99fcb9b2ff2f | joinyycc | joiny | 2020-03-03T00:02:07 |
| 59 | 0x937f7dce238fbb41390a3bf8af05f56579952873 | again | jjhjh | 2020-03-01T16:39:22 |
| 69 | 0x158fdd67c8c9a65d4152bd5178387bbce611db04 | mivo | newFix | 2020-03-01T19:09:03 |
| 60 | 0x062a1966ce82008b0647df766db9c4c41ba3dd8c | dajjeoh | jjhjh | 2020-03-01T17:09:53 |
| 73 | 0xf1e790f3cbcf57b3a9fb4121427c7e00b8e14db3 | proviamociss | proviamoci | 2020-03-02T23:32:28 |
| 64 | 0x87ad035627581bf5c7a1f16f3378c20c5f056253 | tie | newFix | 2020-03-01T17:20:36 |
| 76 | 0xaafbbbab22c81b769b6fa9dd2465b0b482b6d073 | example1 | joiny | 2020-03-17T17:07:25 |
| 58 | 0xe3b5a3afaeb523211f2e410117ce5f80795c0af7 | jkjlkjlk | jjhjh | 2020-02-29T18:26:49 |

< 1 2 >

*Figure 23 - Model Instance implementations list in the Choreography interaction page*



**User account :** 0x535CCa8697F29DaC037a734D6984eeD7EA943A85

**Role:** Participant1 0x535CCa8697F29DaC037a734D6984eeD7EA943A85

**Contract:** - newFix - 0x0753afe493bcc9eedc181fa58148d46a07c551d6

Sid :

Message_06wzatw

Participant :

Auctioneer

current state

**auctionId**:0
**oCodeP1**:401
**oCodeP2**:0
**domainList**:0x504950504f00000000000000000000000000000000000000000000000000000
**participants**:
**itemsList**:
**oAddP2**:0x0000000000000000000000000000000000000000

Sid :

Message_1kw5wo9

Participant :

Participant2

current state

**auctionId**:0
**oCodeP1**:401
**oCodeP2**:0
**domainList**:0x504950504f00000000000000000000000000000000000000000000000000000
**participants**:
**itemsList**:
**oAddP2**:0x0000000000000000000000000000000000000000

*Figure 24 - Choreography Interaction page*

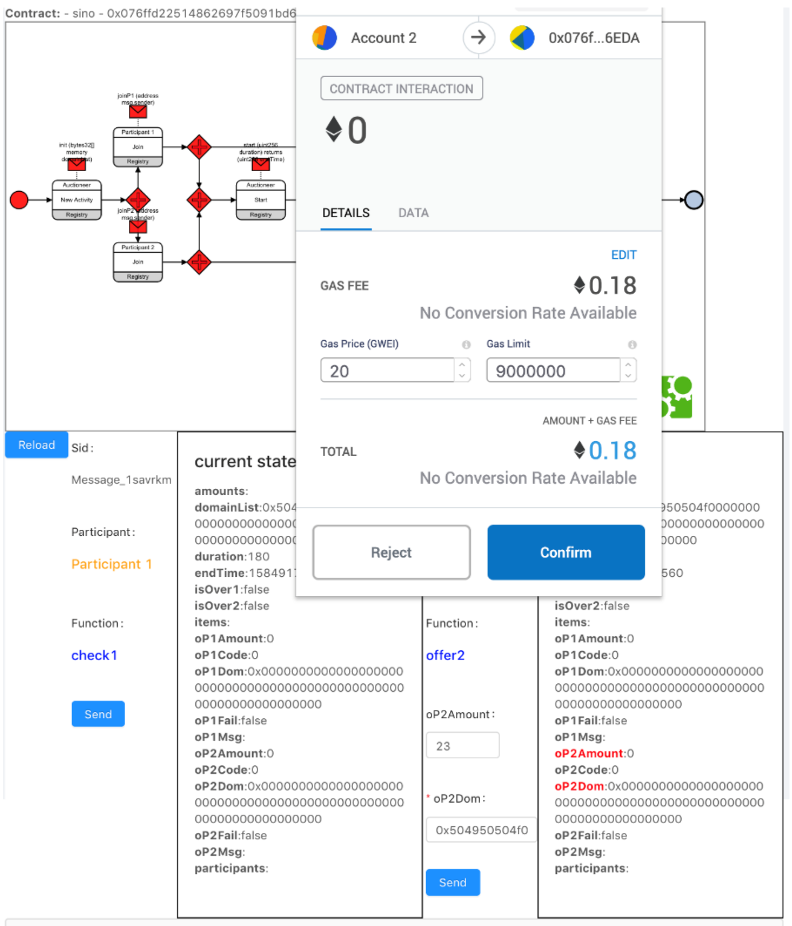
The BPMN viewer is a custom version (at a very early stage) of the BPMN viewer present in the `chor-js` library (5.2.1). It shows the entire BPMN model involved in the Choreography. The behaviour of this component is to indicate the progress of the collaborative process by differentiating the states of the BPMN model components through different colours. Allowed component states of the BPMN model are ENABLED, DONE, DISABLED. For each of these states, a specific colour has been adopted, respectively light blue, green and red. By colouring the model in this way, users have immediately the perception of what is happening, what still can happen and where they are positioned within the collaborative process.

Each Choreography Task message corresponds to a smart contract stub function and its name is extracted from the function signature.

Two important concepts appear below the box:

1. *current operations value*: identified by a box containing all the Message elements, actually the function parameters mapped as form inputs and corresponding to the selected green Message in the Choreography Task;
2. *current state*: identified by a box displaying all the states of the SC global variables at that time. The box updates the modified values only at the next SC execution, which therefore corresponds to the next change of state.

Current operations (Choreography Task Message) are represented by a form built dynamically. The form displays the function name and its parameters at each SC change of state. After filling all the required function parameters, the user must submit the operation by clicking the "send" button. This event will trigger a blockchain transaction that needs to be confirmed via a transaction confirmation pop-up message from MetaMask.



GAS FEE 0.18
No Conversion Rate Available
Gas Price (GWEI) 20
Gas Limit 9000000
AMOUNT + GAS FEE
TOTAL 0.18
No Conversion Rate Available

Sid:
Message_1savrkm
Participant:
Participant 1
Function:
check1

current state
amounts:
domainList:0x504
000000000000
000000000000
duration:180
endTime:158491
isOver1:false
isOver2:false
items:
oP1Amount:0
oP1Code:0
oP1Dom:0x000000000000000000
00000000000000000000000000
0000000000000000
oP1Fail:false
oP1Msg:
oP2Amount:0
oP2Code:0
oP2Dom:0x000000000000000000
00000000000000000000000000
0000000000000000
oP2Fail:false
oP2Msg:
participants:

Function:
offer2
oP2Amount:
23
* oP2Dom:
0x504950504f0

950504f0000000
00000000000000
00000
560
isOver2:false
items:
oP1Amount:0
oP1Code:0
oP1Dom:0x000000000000000000
00000000000000000000000000
0000000000000000
oP1Fail:false
oP1Msg:
oP2Amount:0
oP2Code:0
oP2Dom:0x000000000000000000
00000000000000000000000000
0000000000000000
oP2Fail:false
oP2Msg:
participants:

*Figure 25 - Interaction example during a transaction confirmation*

A dedicated ReactJS component has been created to auto-generate forms dynamically. The component leverages the JSON ABI contract interface (via the json Interface `Web3.js` property) to recognize every single element present in the related SC operation signature. Therefore, for each scenario a proper form box is built according to the information retrieved by the `jsonInterface`. For each SC function parameter found, hence for each input form, a value validation, according to its data type and participant associated, is performed before the submission. Following the model progress, according to the model sequence flow, the UI automatically proposes the operations allowed to that particular user at that particular time. The internal change of state of the Smart Contract marks the flow progress. Changes are driven by the event notifications stateChanged, called following the execution of each SC operation. The notified event will report the value of all the global variables defined in the Smart Contract showing its current state.

## 7.3 Back-end – Chorchain API

The backend hides the processing of the user requests. Furthermore, one of the reasons why we chose a back-end solution was to rely on a central processing point, saving users to install any additional application aside from the browser and MetaMask (as in our case).

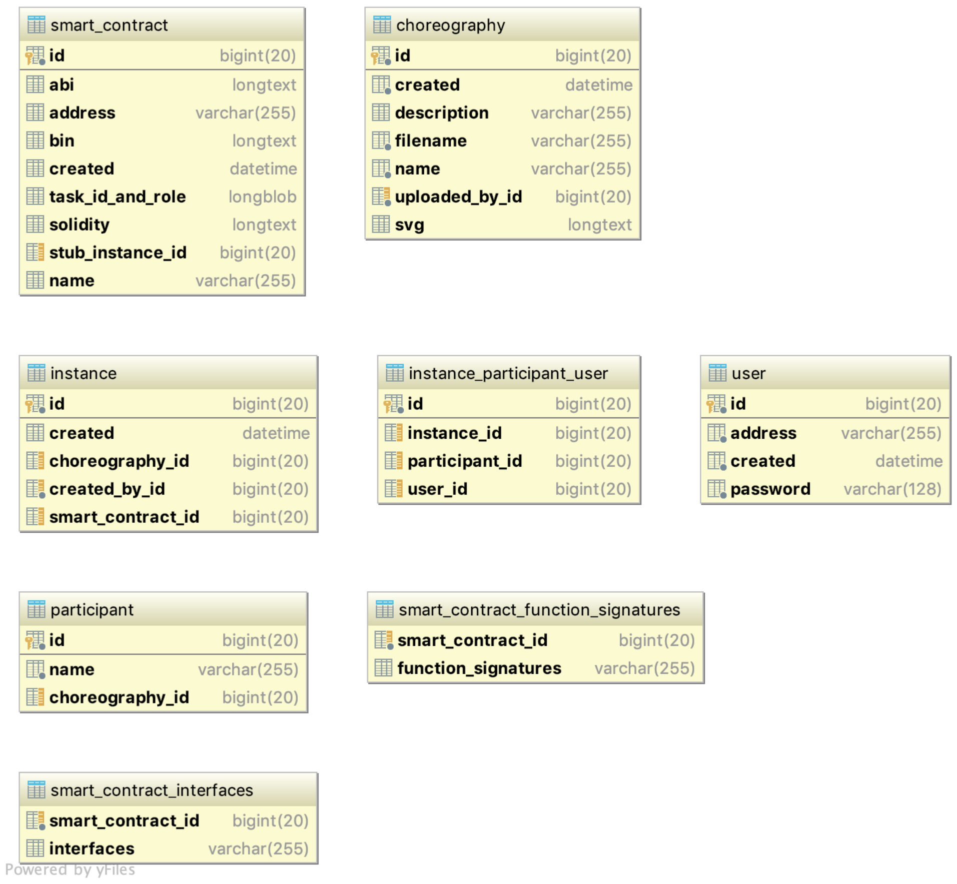
The backend is entirely based on Spring Boot technology (5.4), which uses a Maven approach to handle the dependencies. The main libraries used have been:

- `springdoc-openapi-ui`: library that helps to automate the generation of the API documentation using Spring Boot projects (Figure 27). It works by examining an application at runtime to infer API semantics based on Spring configurations, class structure and various annotations. The library automatically generates documentation in `JSON/YAML` and `HTML` formatted pages. The generated documentation can be complemented using `swagger-api` annotations;
- `org.hibernate`: an object-relational mapping library for the Java programming language. It provides a framework for mapping an object-oriented domain model to a relational database. Hibernate handles object-relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions;
- `spring-boot-starter-data-jpa`: makes it easy implement `JPA` based repositories. This module deals with enhanced support for `JPA` based data access layers. It simplifies building Spring-powered applications that use data access technologies;
- `org.web3`: lightweight, highly modular, reactive, type safe Java and Android library for working with Smart Contracts and integrating with clients (nodes) on the Ethereum network;
- `org.camunda.bpm.model`: a BPMN model API that enables easy extraction of information from an existing process definition, editing an existing process definition or creating a complete new one without any manual XML parsing.

The platform is a web service based on RESTful (5.5). In REST web services, resources are identified by URI. Spring Boot allows to manage requests sent to the application server through Controller objects. In fact, Spring Boot provides and uses many annotations to simplify as much as possible the developer work in handling resource requests.

Let's discover and explain the main Spring Boot annotations.

Everything starts from an `@Entity`. The `@Entity` annotation specifies that the annotated class is an entity and for this reason is mapped in the database as table. Then a `@Repository` component comes. `@Repository` is a Spring annotation indicating that the decorated class is a repository.



| smart_contract | |
|---|---|
| id | bigint(20) |
| abi | longtext |
| address | varchar(255) |
| bin | longtext |
| created | datetime |
| task_id_and_role | longblob |
| solidity | longtext |
| stub_instance_id | bigint(20) |
| name | varchar(255) |

| choreography | |
|---|---|
| id | bigint(20) |
| created | datetime |
| description | varchar(255) |
| filename | varchar(255) |
| name | varchar(255) |
| uploaded_by_id | bigint(20) |
| svg | longtext |

| instance | |
|---|---|
| id | bigint(20) |
| created | datetime |
| choreography_id | bigint(20) |
| created_by_id | bigint(20) |
| smart_contract_id | bigint(20) |

| instance_participant_user | |
|---|---|
| id | bigint(20) |
| instance_id | bigint(20) |
| participant_id | bigint(20) |
| user_id | bigint(20) |

| user | |
|---|---|
| id | bigint(20) |
| address | varchar(255) |
| created | datetime |
| password | varchar(128) |

| participant | |
|---|---|
| id | bigint(20) |
| name | varchar(255) |
| choreography_id | bigint(20) |

| smart_contract_function_signatures | |
|---|---|
| smart_contract_id | bigint(20) |
| function_signatures | varchar(255) |

| smart_contract_interfaces | |
|---|---|
| smart_contract_id | bigint(20) |
| interfaces | varchar(255) |

•

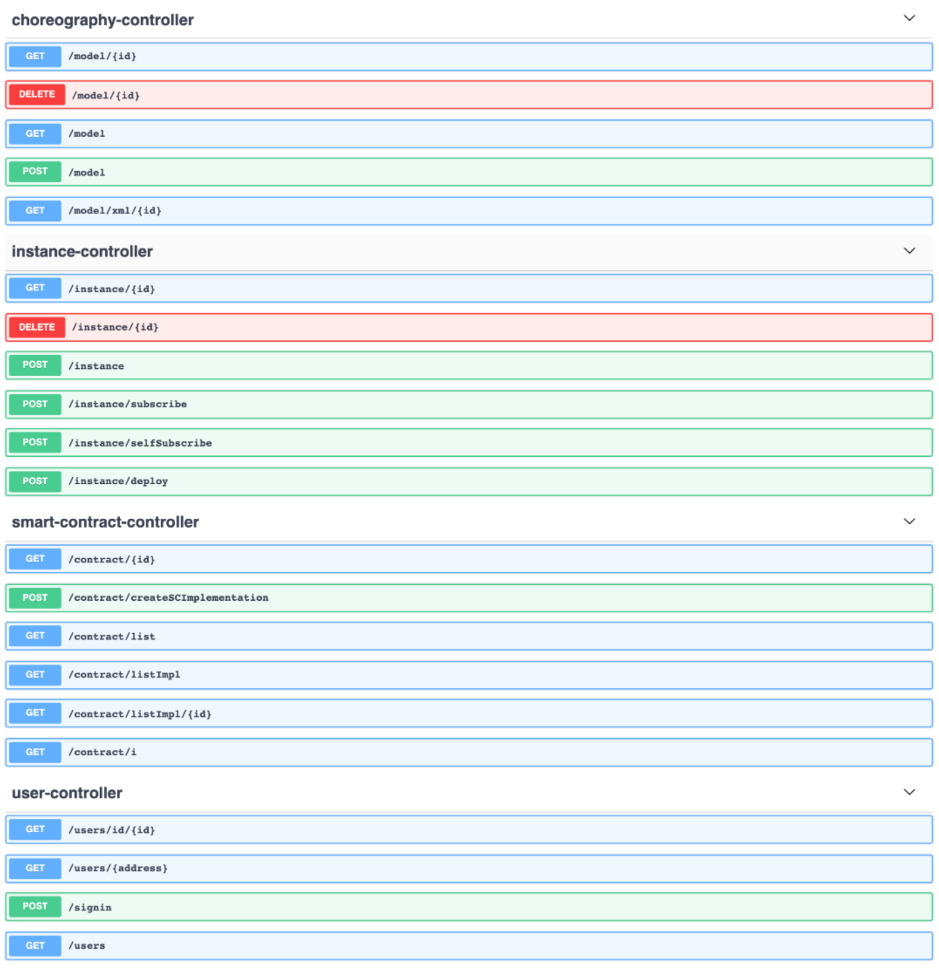
• *Figure 26 - ChorChain database schema*

A repository is a mechanism for encapsulating storage, retrieval, and search behaviour, emulating a collection of objects. It is a subclass specialisation of the `@Component` annotation allowing for implementation classes to be autodetected through classpath scanning. Next two important annotations are `@Service` and `@Controller`. Service it uses to hold the business logic of a certain domain. Developers use `@Service` as bean to put all methods related to the Entity domain. To make these services and their public methods available through endpoints a `@Controller` or `@RestController` annotations are used. `@RestController` annotation marks classes as controllers and this means that every method returned from that class contains a domain object retrieved directly from the Service method linked. In short `@RestController` is the annotation including both `@Controller` and `@ResponseBody`.

We introduce now the main Model Driven ChorChain architecture components, explaining what they do and what is their domain.

In **Errore. L'origine riferimento non è stata trovata.** all the Entity components translated as database tables by Spring Boot are represented.

The figure shows the database tables keys, the attributes and the relationships. While Figure 27 shows all the real API endpoints exposed by the Controllers within the application.



choreography-controller

GET /model/{id}
DELETE /model/{id}
GET /model
POST /model
GET /model/xml/{id}

instance-controller

GET /instance/{id}
DELETE /instance/{id}
POST /instance
POST /instance/subscribe
POST /instance/selfSubscribe
POST /instance/deploy

smart-contract-controller

GET /contract/{id}
POST /contract/createSCImplementation
GET /contract/list
GET /contract/listImpl
GET /contract/listImpl/{id}
GET /contract/i

user-controller

GET /users/id/{id}
GET /users/{address}
POST /signin
GET /users

*Figure 27 - ChorChain endpoints by OpenAPI definitions*

### 7.3.1 Choreography

The Choreography entity is representing the BPMN choreography model.

Entity attributes are:

- *created*: creation date;
- *description*: choreography description entered by the user;
- *filename*: the uploaded filename;
- *name*: the identified choreography name;
- *uploadedByID*: the user id who uploaded;
- *svg*: svg content representing the model graphically;

As we know from the frontend description, users can design their own BPMN model and store it in the platform. Actually, the designed BPMN model, purely XML content, is uploaded on the filesystem server and then all the related information memorised in the database (filename, name, description, svg content, etc.). Specific endpoints are provided to handle these operations such as:

- `GET /model/`: retrieves a model list
- `GET /model/{id}`: retrieves a specific model by id
- `DELETE /model/{id}`: delete a specific model by the id
- `POST /model/{id}`: creates a new model in the database after it has been uploaded by clicking the upload button on the front-end Choreography Deploy page.

- `GET /model/xml/{id}`: provides the model XML content (XML preview functionality)

### 7.3.2 Instance

The Instance entity is representing the BPMN choreography model translation triggered by the user. The entity attributes are:

- *created*: creation date;
- *choreographyId*: the parent Choreography;
- *createdById*: the user providing the instance creation;
- *smartContractId*: the related skeleton smart contract eventually produced;
- *participants*: a `OneToMany` relationship; the choreography participants list related to the user table realised by the `instance_participant_user` table;

An instance is created from a BPMN model (its parent). After the definition of all the possibly participant roles it is possible to create the instance. Specific endpoints are provided to handle these operations such as:

- `GET /instance/`: retrieves an instance list
- `GET /instance/{id}`: retrieves a specific instance by id
- `DELETE /instance/{id}`: deletes a specific instance by the id
- `POST /instance/subscribe`: allows users to subscribe to the instance as participant
- `POST /instance/deploy`: provides the skeleton smart contract generation related to this instance, the translation phase.

### 7.3.3 Smart Contract

The Smart Contract entity contains all the information associated to the Smart Contract deployed on the Blockchain. The information is useful mainly for the front-end Smart Contract interaction. The Smart Contract attributes list is:

- *created*: creation date;
- *name*: Smart Contract name;
- *bin*: solidity binary file containing the hex-encoded binary to provide the transaction request;
- *abi*: solidity Application Binary Interface file which details all of the publicly accessible contract methods and their associated parameters. These details along with the contract address are crucial for interacting with smart contracts;
- *solidity*: the entire generated Solidity file;
- *address*: Ethureum address of the deployed Smart Contract;
- *stubInstanceId*: `ManyToOne` relationship with itself. Identifies the Implementation Smart contract derived from the parent skeleton Smart Contract;

Now we take a look the endpoints provided by the API to manage and interact with Smart Contracts.

- `GET /contract/{id}`: retrieves a specific contract by id
- `GET /contract/listImpl`: retrieves a contract list of Instance Implementations
- `POST /contract/createSCImplementation`: creates a Smart Contract implementation

### 7.3.4 Internal API detail

In this section a detail overview of internal API mechanism is given. A schematic view of the API operations is shown in Figure 28. Through a simplified working scheme, we try to summarise the API, leaving out the user management mechanism. Conceptually, each *blue* block represents a set of logic components constituted by the three Entity, Service and RestController components. The *green* block is showing an artefact while the orange block indicating procedures. All internal platform actions or user invoked are represented by labels.

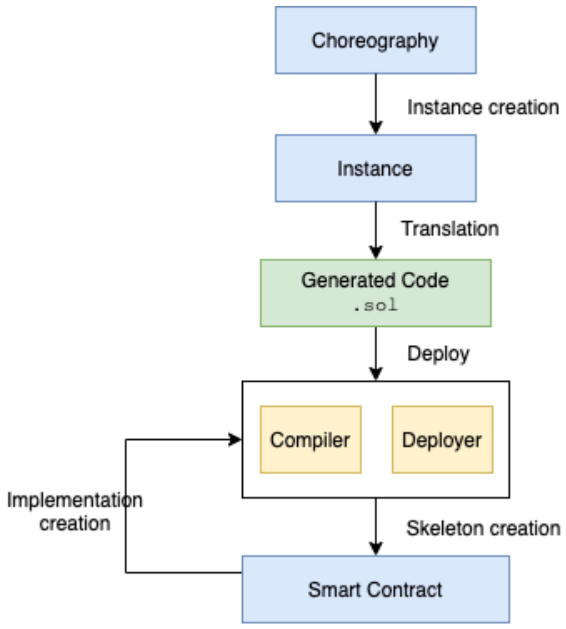


*Figure 28 - Internal API schema*

#### *7.3.4.1 Translation or code generation*

This feature accepts as input an XML based BPMN model and provides as result the translation into Solidity code (`.sol`). Technically this is a complex business logic part leveraging the Factory Method design pattern to perform a proper translation.

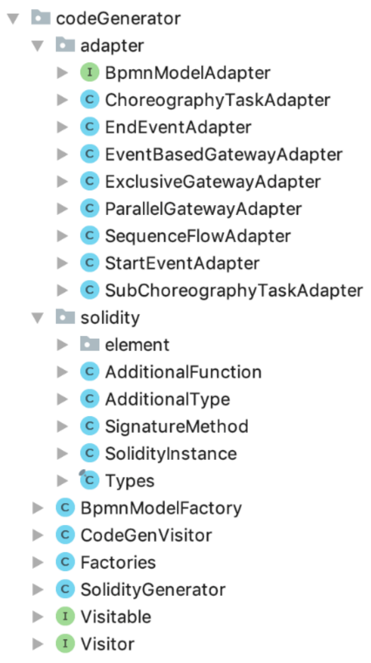
**Factory Method**: A normal factory produces goods while a software factory produces objects. This is done without specifying the exact class of the objects to be created. To accomplish this, objects are created by calling a factory method instead of calling a constructor.

Usually, object creation in Java occurs in the following way:

```
SomeClass someClassObject = new SomeClass();
```

The problem with the above approach is that the code using the `SomeClass’s` object becomes dependent on the concrete implementation of SomeClass. There is nothing wrong with using `new` to create objects but it tightly couples our code to the concrete implementation class, which can occasionally be problematic.

In order to adapt the BPMN Camunda elements into our platform and provide the Solidity transformation, we created a dedicated `codeGeneretor` package (Figure 29). The `codeGeneretor` creates an internal BPMN element from each Camunda BPMN element processed by applying the Factory Method Pattern. The result is a structure of BPMN elements loosely coupled with Camunda BPMN elements but with additional behaviours and data. Having BPMN elements loosely coupled with Camunda promotes future package extensibility. The relevant definitions of all BPMN elements (of interest) useful for the translation process have been placed in the `adapter` package (Figure 29).



```
▼ codeGenerator
  ▼ adapter
    ▶ BpmnModelAdapter
    ▶ ChoreographyTaskAdapter
    ▶ EndEventAdapter
    ▶ EventBasedGatewayAdapter
    ▶ ExclusiveGatewayAdapter
    ▶ ParallelGatewayAdapter
    ▶ SequenceFlowAdapter
    ▶ StartEventAdapter
    ▶ SubChoreographyTaskAdapter
  ▼ solidity
    ▶ element
    ▶ AdditionalFunction
    ▶ AdditionalType
    ▶ SignatureMethod
    ▶ SolidityInstance
    ▶ Types
  ▶ BpmnModelFactory
  ▶ CodeGenVisitor
  ▶ Factories
  ▶ SolidityGenerator
  ▶ Visitable
  ▶ Visitor
```

*Figure 29 - CodeGenerator package*

An interface `BpmnModelAdapter` defines our base BPMN element, then the `BpmnModelFactory` recognise the Camunda BPMN element processed and creates a new custom BPMN element using the proper adapter.

```
public interface BpmnModelAdapter extends Visitable {
      String getId();
      String getOrigId();
      String getName();
      DomElement getDomElement();
      ModelInstance getModelInstance();
      String getClassSimpleName();
      List<BpmnModelAdapter> getIncoming();
      List<BpmnModelAdapter> getOutgoing();
}
```

*Listing 10 - BpmnModelAdapter interface declaration*

The prototype interface is now containing only the BPMN *name*, the BPMN *id*, the list of *outgoing elements* and of *incoming elements*, the original *className*, and the *modelInstance* pointer, a "magic" tool provided by Camunda library which allows to query BPMN elements.

```
public class BpmnModelFactory {

    public BpmnModelAdapter create(StartEvent value) {
        return new StartEventAdapter(value);
    }

    //ModelElementInstance {Start,End, Task , ..}
    public BpmnModelAdapter create(ModelElementInstance value) {
        /* Add here more instance type if needed */
        if (EndEventImpl.class.equals(value.getClass())) {
            return new EndEventAdapter((EndEvent) value);
        } else if (ParallelGatewayImpl.class.equals(value.getClass())) {
            return new ParallelGatewayAdapter((ParallelGateway) value);
        } else if (EventBasedGatewayImpl.class.equals(value.getClass())) {
            return new EventBasedGatewayAdapter((EventBasedGateway) value);
        } else if (ExclusiveGatewayImpl.class.equals(value.getClass())) {
            return new ExclusiveGatewayAdapter((ExclusiveGateway) value);
        } else {
            if (((ModelElementInstanceImpl) value).getElementType()
            .getTypeName()
            .equals("subChoreography")) {
                return new SubChoreographyTaskAdapter(value);
            }
            return new ChoreographyTaskAdapter(value);
        }
    }

    //SequenceFLow element - used only to build the follow the tree
    public BpmnModelAdapter create(FlowElement value) {
        return new SequenceFlowAdapter(value);
    }
}
```

*Listing 11 - BpmnModelFactory implementation*

To simplify the code generation, we also leverage the Visitor design pattern applied to the data structure used to identify our BPMN sequenceFlow.

**Visitor**: The Visitor pattern suggests putting new behaviours into a separate class called visitor, rather than trying to integrate it into existing classes. The original object to which a behaviour is to be applied is now passed to one of the visitor's methods as an argument, giving the method access to all necessary data contained within the object.

To implement the Visitor pattern, we defined a `Visitable` interface applied to `BpmnModelAdapter`.

```
public interface Visitable {
   void accept(Visitor visitor);
}
public interface Visitor {
   void visit(BpmnModelAdapter node);
   void visitStartEvent(StartEventAdapter node);
   void visitEndEvent(EndEventAdapter node);
   void visitParallelGateway(ParallelGatewayAdapter node);
   void visitExclusiveGateway(ExclusiveGatewayAdapter node);
   void visitEventBasedGateway(EventBasedGatewayAdapter node);
   void visitChoreographyTask(ChoreographyTaskAdapter node);
   void visitSubChoreographyTask(SubChoreographyTaskAdapter
                                    subChoreographyTaskAdapter);
}
```

*Listing 12 - Visitor and Visitable interface applied to BpmnModelAdapter*

Therefore, taken a BPMN model and parsed all its structure by the recursive `traverse` method - following nodes of the SequenceFlow - we populate the `bmpnTree` list of `BpmnModelAdapter` type. Then, looping over this list we execute the `accept` method by forcing each processed element to be interpreted by the `CodeGenVistor` class. This class contains all the specific translation business logic for each element processed.

```
public void traverse(BpmnModelAdapter node) {
    if (!visited.contains(node.getId())) {
       bpmnTree.add(node);
       visited.add(node.getId());
    }

     node.getOutgoing().forEach(this::traverse);
}
```

*Listing 13 - Method to traverse the BMPN SequenceFlow*

The specific translation, carried out by the `CodeGenVistor` class, makes use of particular Solidity *builders* created ad hoc and inserted in the *Solidity* package. Once that the CodeGenVistor class runs, a Solidity Instance class it is built. `SolidityInstance` is the class containing the entire instance structures used to generate the `.sol` file using the solidity builders.

### 7.3.4.2 Compiler

This is an internal function included in `/instance/deploy` and `/contract/createSCImplementation` endpoints. It is used to compile the Solidity code generated by the platform during the transformation process. It makes use of `solc` compiler, an external tool namely Solidity Compiler[21]. Its role somehow validates the generated code, thus telling us if it is correct or not, compiling it. Furthermore, it is also responsible for generating Smart Contracts together with its accompanying information such as ABI, BIN and address.

We are running the complier with these options:

- `bin`: generates the Smart Contract binary content
- `abi`: generates the Smart Contract ABI interface
- `overwrite`: overwrites existent .sol found on the directory • optimize: optimize the solidity bytecode
- `o`: the output directory

### 7.3.4.3 Deploy

If the compiling operation is successful then it is possible to move on to the next operation, the Smart Contract deploying. This is also a platform internal operation. The deploy operation is responsible of distributing the Smart Contract to the blockchain. To complete this job, it is needed a blockchain connector and some of the Smart Contract information. Below you can find our implementation.

We have used the `Web3.js` library, which allows communicating with the blockchain, as blockchain connector.

```
//Unlocking administration account
adm.personalUnlockAccount(adminAccount, passAdminAccount).send();



EthGetTransactionCount ethGetTransactionCount =
                   web3j.ethGetTransactionCount(
                 adminAccount,
                 DefaultBlockParameterName.LATEST).sendAsync().get();
BigInteger nonce = ethGetTransactionCount.getTransactionCount();
```

[21] https://docs.soliditylang.org/en/v0.4.24/using-the-compiler.html

```
BigInteger GAS_PRICE = BigInteger.valueOf(gasPrice);
BigInteger GAS_LIMIT = BigInteger.valueOf(gasLimit);

//compiled smart contract code
String compiledSCCode = new String(Files.readAllBytes(Paths.get(projectPath +
File.separator + parseName(name, ".bin"))));

Transaction transaction = Transaction.createContractTransaction(adminAccount,
               nonce, GAS_PRICE, GAS_LIMIT, BigInteger.ZERO, "0x" + compiledSCCode);
//send sync
EthSendTransaction transactionResponse = web3j.ethSendTransaction(transaction).send();
```

*Listing 14 - Part of the deploy operation*

As shown in Listing 14, we need to unlock the admin account, create a blockchain transaction and setting a nonce, gas price, gas limit and the `.bin` file of the compiled Smart Contract. Furthermore, the transaction will be submitted to the Blockchain waiting for response.

## 7.4 Smart Contract internal

As we have learned so far, the Smart Contract generation is an automatic operation made by the back-end and addressed by the user – the user chooses and designs what the SC should contain internally.

Regarding the internal structure of the SC, we took inspiration from the solution proposed in [11]. According to the proposed study, a contract is made up of two parts:

1. *static*, it is practically unchanged for all the contracts and it is used to manage the internal state of the contract;
2. *dynamic*, it is the result of the Bpmn-To-Solidity translation.

As we already know, our smart contract is composed by three contracts (6.1) One of this part is a static one (Listing 15) belonging to the first contract of the three. To better proceed with a more complete explanation, the auto-generated code is now partially shown.

```
pragma solidity ˆ0.5.3;
pragma experimental ABIEncoderV2;



contract ChoreographySmartContract{
   /* constructor */
   constructor () public {
     for (uint i = 0; i < elementsID.length; i ++) {
        elements.push(Element(elementsID[i], State.DISABLED));
        position[elementsID[i]]=i;
     }


     //roles definition
     //mettere address utenti in base ai ruoli
     roles["Participant1"] = 0x535CCa8697F29DaC037a734D6984eeD7EA943A85;
     roles["Participant2"] = 0x535CCa8697F29DaC037a734D6984eeD7EA943A85;
     optionalRoles["Participant3"] = 0x0000000000000000000000000000000000000000;
     optionalRoles["Participant4"] = 0x0000000000000000000000000000000000000000;
     //enable the start process
     _init();
   }
```

```
/* Mappings */
mapping(string => uint) position;


mapping(string => address) roles;
mapping(string => address) optionalRoles;



/* Structs */
struct Element {
   string ID;
   State status;
}

struct StateMemory {
   …
}

/* Enums */
enum State {DISABLED, ENABLED, DONE} State s;

/* Variables */
address payable public owner;
string [] elementsID = ["ExclusiveGateway_0cw6nha", "ParallelGateway_1b8idm5", .. ];
IRegistryImpl iregistry= new IRegistryImpl();
Element[] elements;
StateMemory currentMemory;


string [] roleList = ["Participant1","Participant2"];
string [] optionalList = ["Participant3","Participant4"];
 uint counter;


/* Events */
event stateChanged(uint);
event functionDone(string);

/* Modifiers */
modifier checkMand(string storage role){
   require(msg.sender == roles[role]);
   _;
}

modifier checkOpt(string storage role){
   require(msg.sender == optionalRoles[role]);
   _;
}

modifier Owner(string memory task){
   require(elements[position[task]].status == State.ENABLED);
   _;
}

/* Functions */

 .....  removed for length reason.......

/* Custom */
function subscribe_as_participant(string memory _role) public {
   if (optionalRoles[_role] == 0x0000000000000000000000000000000000000000) {
          optionalRoles[_role] = msg.sender; }
      }

   function() external payable {}
   function enable(string memory _taskID) internal {
      elements[position[_taskID]].status = State.ENABLED;
      emit stateChanged(counter++);
   }


   function disable(string memory _taskID) internal {
      elements[position[_taskID]].status = State.DISABLED;
   }

   function done(string memory _taskID) internal {
       elements[position[_taskID]].status = State.DONE;
       emit functionDone(_taskID);
   }


   function getCurrentState() public view
     returns (Element[] memory, StateMemory memory){
        // emit stateChanged(elements, currentMemory);
        return (elements, currentMemory);
   }

   function compareStrings(string memory a, string memory b) internal pure
     returns (bool) {
      return keccak256(abi.encode(a)) == keccak256(abi.encode(b));
   }

   function _init() internal {
     bool result = true;
     for (uint i = 0; i < roleList.length; i++) {
         if (roles[roleList[i]] == 0x0000000000000000000000000000000000000) {
             result = false;
             break;
         }
      }
      if (result) {
        //This is the start point
        enable("StartEvent_0l50fnp"); StartEvent_0l50fnp();
        emit functionDone("Contract creation");
      }
   }
}//Contract end
```

*Listing 15 -Static, unchanged Smart Contract part*

### Internal state and Sequence Flow crossing

The Smart Contract here proposed offers two data structures to provide its internal state to users. These are:

- *Element*: is composed by the element `id` and the `status` of the actual element (`ENABLED`, `DISABLED`, `DONE`). `DISABLED` means that the current BPMN element is waiting to be activated, the flow is still not arrived there, `ENABLED` means that the flow has reached the current element, `DONE` means that the current element will never be reached by the flow;

- *StateMemory*: memory area containing all the global variables. Here, all parameters and return variables included in the signature method produced by the translation Bpmn-To-Solidity process are declared.

Hence, by means of the array of `elements` and `currentMemory` it is possible through the emission of the event `stateChange` - which reports these two values - to know the current choreography internal state. This event is emitted every time there is a SC change of state which should correspond at each (tick) flow progress within the BPMN model. The ID of the traversed elements are gradually stored in the `elementsID` array. Whenever there is an *external interaction* - i.e. a SC method is invoked in some way by the Participant interaction through an external call - the BPMN flow makes a movement from one element to another, namely makes a *tick*. Flow navigation is allowed by means of the three functions `enable`, `disable` , `done` . Each of these functions perform an internal change of state of the elements enabling the next step and disabling the current one just performed. The SC `constructor` initialised all the elements with the `DISABLE` state.

**Role and participant management**

Participants roles are stored in two different arrays `roleList`, `optionalList`. During the translation process mandatory roles are mapped into the `roles` mapping with their known associated user addresses, while the `optionalRoles` are mapped with the default address 0 by the constructor. The last mapping will be populated if needed at runtime through the “subscribe as participant” (line 83) function invoked by Participants who desire to subscribe to the choreography process. Their addresses are automatically retrieved via the use of `msg.sender` which enables to get the address of the caller.

A security control is also expected and performed over the operation that each Participant during their interactions process invoke. This security control is done by two modifiers `checkMand` and `checkOpt` and they are used to enforce, from the contract side, the right identity of the ”sender”. They both check if the Participant role at that time corresponds to the role associated with the operation to invoke and if the participant account is in one of the two mappings `roles` and `optionalRole`.

**Coding BPMN Elements**

Because of its length, this part referred as “…….” is not reported in the listing. However, it represents the core of the Smart Contract generation. The generated code is the result of the Bpmn-To-Solidity translation process actuated for all the elements discovered in BPMN Sequence Flow.

The authors of [11] proposed a nice solution for BPMN-To-Solidty transformation but in that proposal not all the BPMN elements have been defined, so we provided some extra transformations.

In this section we will only deal with a single case of transformation. We will take the transformation of a One-Way Coreography Task Element as an example by explaining it. For this transformation we also have proposed a different translation solution while for all the other transformations, you can refer to the article [11].

In the BPMN model a One-Way Choreography Task is represented by a choreography task with an associated message to the upper Participant, while similarly, the Two-Way choreography task is represented by two messages respectively associated one to the superior participant and the other to the lower one. Thus, the choreography elements appearing in the contract can be divided in two main categories: messages - representing the interaction functions between participants - and control flow elements. During the translation process each task message (graphically the envelope message associated with a participant) is transformed in a public function. All parts of the object are broken down hence the message name and all the information injected during the design stage are now considered. Any metadata previously added to the BPMN model is used to properly regenerate the function signature.

In the proposed example the function associated with the Task Message (`Message_0l2eq5d`) it is defined like this during the design stage:

*start (uint256 duration) returns (uint256 endTime*)

We point out that our prototype solution allows to define functions with both input and output parameters. During the transformation process parameters are automatically added to the `stateMemory` global variable struct which is instantiated as `currentMemory` mapping. At the same time the function signature is calculated by the system and its call is filled in the Message function body (line 6). In the meantime, the SC contract containing the methods implementation (Listing 15) is also instantiated if not already present in the main SC. In the example the reference to SC implementation corresponds with the variable `iregistry` (Listing 15). This variable represents the link to another (external) contract dedicated to containing the business logic and also storing the one for the `start` function (Listing 16 line 6). The translator phase also provides other two definitions of the function. One to the `IRegisty` smart contract that acts as Interface (Listing 17 line 4) and the other to the `IRegistryImpl` (Listing 18 line 6) contract that extends the `IRegistry` interface providing the real implementation.

Two other important definitions relating to the function are also provided during the translation phase. One is related to the `IRegistry` SC that acts as an interface (Listing 17 line 4) and the other is related to the `IRegistryImpl` SC (Listing 18 line 6) used to extend the `IRegistry` interface providing the actual implementation.

```
/Task(Start): ChoreographyTask_0n0f3pe - TYPE: ONEWAY
function Message_0l2eq5d(uint256 duration) public checkMand(roleList[1]) {
   require(elements[position["Message_0l2eq5d"]].status == State.ENABLED);
   done("Message_0l2eq5d");
   currentMemory.duration = duration;
   (currentMemory.endTime) = iregistry.start(duration);
   enable("ParallelGateway_02fwm56");
}
```

*Listing 16 - Translation detail of a One-Way Choreography Task Message*

```
/*Interface generation*/
contract IRegistry{
    ...
    function start( uint256 duration ) public returns (uint256 endTime);
    ...
}//Interface End
```

*Listing 17 - Interface of the "Implementation contract functions"*

```
/*InterfaceImpl generation, provides function stubs*/
contract IRegistryImpl is IRegistry{
   ...
   function start( uint256 duration ) public returns (uint256 endTime){
      //stub generated -- insert here your code
   }
   ...
}//InterfaceImplementation End
```

*Listing 18 - Contract of the Implementation functions containing the business logic*

## 7.5 Blockchain environment

The prototype goal is to allow the interaction of participants through the blockchain. As previously discussed we have chosen Ganache as a blockchain solution to make the whole application autonomous and easy to be developed at least at an early stage. Ganache-cli (3.6) comes with the Truffle suite of Ethereum development tools. Ganache cli is a personal blockchain for Ethereum development, uses `ethereumjs` to simulate full client behaviour and make developing Ethereum applications faster, easier, and safer. It also includes all popular RPC functions and features (like events) and can be run deterministically to make development easy. We have created a blockchain start-up script (list shown below) that has used some interesting features enriching this tool.

```
#!/usr/bin/env bash

ganache-cli -d --db ganache-db --port 8545 -l 9000000000 -g 20000000000
--accounts 10 --mnemonic 'include poem goose genuine baby flat mom token drama harsh
sadness fit' --networkId 5777 --verbose
--allowUnlimitedContractSize
```

*Listing 19 -Ganache-cli startup script*

Below you can find the options:

- `nmemonic`: uses a bip39 mnemonic phrase for generating a PRNG seed, which is in turn used for hierarchical deterministic (HD) account generation
- `d`: generates deterministic addresses based on a pre-defined mnemonic.
- `db`: specifies a path to a directory to save the chain database. If a database already
- exists, ganache-cli will initialize that chain instead of creating a new one
- `ganache-db`: enables the database creation useful to store data for history purpose
- `port`: port number to listen on
- `l`: the block gas limit
- `g`: the price of gas in wei
- `network`: specifies the network id ganache-cli will use to identify itself
- `verbose`: logs all requests and responses to stdout
- `allowUnlimitedContractSize`: allows unlimited contract sizes while debug-ging. By enabling this flag, the check within the EVM for contract size limit of 24KB (see EIP-170) is bypassed. Enabling this flag will cause ganache-cli to behave differently than production environments

Using this configuration, we have unlimited contract size and also a huge amount of gas and limited block of gas limits. Many of these options are very useful for actual purpose due to the big dimension of our Smart Contract. Other additional benefits are given by the mnemonic options, which allow using all the time the same environment with the same user accounts. Moreover, there is the db option which continuity is guaranteed over time so it is possible to refer to addresses of contracts already distributed even after closing and opening the blockchain again. Obviously, the greatest advantage of our implementation is that we do not need a blockchain instance to configure, considering all the effort that required it, both in creation and in maintenance.

# 8 Conclusion

In this Technical Report a prototype named Model Driven ChorChain 2.0 platform has been proposed. The platform by means of a model driven approach generates smart contracts aimed at certifying the interaction between participants in collaborative processes previously designed by an editor of the platform. The applied methodology foresees the use of BPMN standards through the use of BPMN choreographic models which, thanks to the techniques proposed previously in the document, are transformed into Solidity code and distributed as smart contracts over the blockchain.

Mainly, the study proposed in this TR is still at an experimental stage. The experiments conducted were aimed more at a study of feasibility rather than at a formalisation of the method. The hope is that future investigations aim  to formalise the method so that future work can continue.

This work is already part of the continuation of the research conducted by the authors [11] which previously had defined a useful methodology for transforming the BPMN sequence flow in Solidity code. In our investigation, we focused more on making the BPMN choreographic model more exploitable through the model-driven approach. In fact, it has been demonstrated that through the use of the template method design pattern applied to the generation of smart contracts, a skeleton contract can be easily built. Taking advantage of loose coupling with the skeleton contract, introduced by the template method design pattern, it is possible to extend the model and subsequently adding business logic in any form.

An application scenario of the proposed platform could be, for example, the one where an architect designs its collaborative process through the choreography model and, a developer or an experienced Solidity programmer, implements just the business logic part knowing only the auto generated interface. An interesting future work, and probably the next step, would be to demonstrate the validity of the tool through a real example. For example, to provide a real case study in which to define SC and add business logic taken from an already existing contract in order to certify and guide interactions between participants through the platform and blockchain.

Another interesting but challenging task could be leveraging the skeleton contract to invoke rest stub methods to some REST service. The web service could be written in any other language and therefore use the collaboration process over the blockchain only to “certify” the communications that occurred between the parties. This scenario is still to be explored but a hypothetical interesting tool to test could be that one proposed by Provable[22] through oracles.

However, the addressed studies to understand how to enrich the BPMN model was very important. It has been noted that through Camunda Form it is very easy to transport additional metadata into the BPMN model and the idea of defining dedicated operation behaviours as data container in the UI properties. panel can be a winning strategy. Also the inclusion in the UI of libraries ”packages” containing predefined business logic and mechanisms the let define structs, events, functions and other more smart contracts constructs, have been convincing.

Other important aspects were on the development side. An attempt was made to develop a solid and real product with structured APIs rather than a single prototype. The idea, in fact, was to go in the direction of defining software that it is easy to release, expand and extend. Therefore, the architectural choices were made following this idea

[22] https://provable.xyz

Blockchain technologies represent a fundamentally new way to transact business. They represent a robust and smart next generation of applications for the registry and exchange of physical and virtual resources. Thanks to the key features such as cryptographic security, decentralized consensus, and a shared public ledger, blockchain, and a shared public ledger, blockchain technologies can profoundly change the way to interconnect distributed systems.

## 8.1 Future Works

**Increase the BPMN translator engine capacity**: it is essential to focus on the working of the BPMN translator. It represents the real heart of the project and its reliability and the ability to be developed further are very important. Nevertheless, during the platform development an attempt was made to define a fairly robust API that offers the basis for creating a structured translation engine with the hope of being able to easily expand its possibilities. Therefore, this can be a good starting point

**Smart Contracts optimizations and best practices**: plenty of optimization techniques or design patterns can be applied to our smart contracts. In the proposed study not much attention has been paid to this aspect. In fact, we also encountered some difficulties with regards to the costs incurred for compiling contracts and for transactions. However, this aspect it is always a big concern for developers. Focusing on reducing implementation and interaction costs could be another future work. At the same time, attention must be paid to the security aspects of the contract, avoiding bad programming techniques that can open security holes and make it unsafe.

**Formalisation and validation**: it would be interesting to carry out a study in the methodologies suitable for formalising the operations performed during the generation of the code for smart contracts. Both for the transformation of BPMN elements into solidity code and for the methods used to insert Solidity information with UI into (via XML) to the BPMN model. Inevitable functionality should be the ability to test the generated code and thus avoid incorrect code production for instance using model-based testing (2.1) approaches.

**Connect smart contracts with external APIs**: Another interesting feature already discussed should be exchanging information between smart contract and external environments by API connection such as REST endpoints. Data authenticity is the key to protecting the decentralised logic. This feature is integrated with most public blockchains and can work on any private one too. In this case the blockchain relies on the presence of a "trustable" mediator, called Oracle, that retrieves data from an external source and directly delivers them to smart contracts. A fairly widespread solution on the internet regarding this aspect is that provided by Provable[23] which offers a product called Provable™ oracle service. Oracles allows these kinds of communications thanks to their mediation system between contracts and the network. Thus, deepening the study concerning the aspect of the oracles aimed at increasing the versatility of the proposed platform it is a very interesting thing.

**User interface enhancement and improvement**: certainly, one of the most important components of the platform is the UI. It handles all the user requests helping interaction with the API and with the blockchain. For this reason, taking care of all the details needs a lot of effort, hence the development of the UI has been very demanding so far. Ensuring a good user experience though is mandatory for a platform where model driven development is the main aspect, therefore the part concerning the editor of the BPMN model must still be taken care of. Not less important is the development of the UI side relating to interaction with the blockchain. This is a very young area where there are useful and emerging technologies. Last but not least, we would recommend considering the adoption of tools belonging to the Truffle suite (the one from which the ganache-cli

[23] https://github.com/provable-things/ethereum-api

comes from), a collection of tools that make the dapp management easier, Truffle, Drizzle and Ganache. Truffle[24] takes care of managing the contract artefacts so we do not have to. It includes support for custom deployments, testing, library linking and complex Ethereum applications aiming at making life as a developer easier. Drizzle[25] is a collection of front-end libraries that make writing dapp front-ends easier and more predictable, especially with ReactJs.

[24] https://www.trufflesuite.com/truffle

[25] https://www.trufflesuite.com/drizzle

# 9 References

[1] D. C. Schmidt, «Guest Editor's Introduction: Model-Driven Engineering,» *Conmputer,* vol. 39, n. 2, pp. 25-31, 2006.

[2] M. M. Daniel F., "Model-Driven Software Development," in *Mashups. Data-Centric Systems and Applications.*, Berlin, Heidelberg: Springer, 2014, pp. 71-93.

[3] P. M. Damiano Di Francesco Maesa, «Blockchain 3.0 applications survey,» *Journal of Parallel and Distributed Computing,* vol. 138, pp. 99-114, 2020.

[4] T. B. F. N. A. S. C. H. a. T. R. Aitor Aldazabal, «Automated Model Driven Development Processes,» in *Tools & Process Integration Workshop*, Berlin, 2008.

[5] OMG, «Business Process Model and Notation (BPMN), Version 2.0.2,» 2013. [Online]. Available: https://www.omg.org/spec/BPMN/2.0.2/PDF.

[6] A. v. W. e. M. W. Jan Ladleif, «chor-js: A Modeling Framework for BPMN 2.0 Choreography Diagrams,» in *Proceedings of the ER Forum and Poster & Demos Session 2019 on Publishing Papers with CEUR- WS co-located with 38th International Conference on Conceptual Modeling*, Salvador, Brazil, 2019.

[7] F. &. M. A. &. P. A. &. R. B. &. T. F. Corradini, «Collaboration vs. Choreography Conformance in BPMN 2.0: From Theory to Practice,» in *IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, 2018.

[8] I. W. e. M. S. Xiwei Xu, «Model-Driven Engineering for Block- chain Applications,» in *Architecture for Blockchain Applications*, Springer, 2019, pp. 149-172.

[9] Q. L. e. I. W. An Binh Tran, «Lorikeet: A Model-Driven Engineering Tool for Blockchain-Based Business Process Execution and Asset Management,» in *BPM*, 2018.

[10] «Bpmn-js documentation,» [Online]. Available: https://github.com/bpmn-io/bpmn-js-examples/blob/master/custom-elements/README.md.

[11] A. M. A. M. A. P. B. R. a. F. T. F. Corradini, «Engineering Trustable Choreography-based Systems using Blockchain,» in *SAC 35th ACM/SIGAPP Symposium On Applied Computing*, Brno, 2020.

[12] D. G. W. A. M. Antonopouls, Mastering Ethereum, O'Reilly, 2019.

[13] S. D. Henrique Rocha, «Preliminary steps towards modeling blockchain oriented software,» in *WETSEB '18: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018.

[14] S. S. C. H. I. W. Christoph Prybila, «Runtime verification for business processes utilizing the Bitcoin blockchain,» *Future Generation Computer Systems,* vol. 107, pp. 816-831, 2020.

[15] O. &. G.-B. L. &. D. M. &. W. I. &. P. A. Pintado, «CATERPILLAR: A Business Process Execution Engine on the Ethereum Blockchain,» 2018.

[16] Facebook, «Virtual DOM and Internals,» [Online]. Available: https://reactjs.org/docs/faq-internals.html.

[17] I. W. W. V. d. A. o. Jan Mendling, «Blockchains for Business Process Management - Challenges and Opportunities,» *ACM Transactions on Management Information Systems,* vol. 9, n. 1, pp. 1-16, 02 2018.

[18] M. D. L. G.-B. I. W. Orlenys López-Pintado, «Dynamic Role Binding in Blockchain-Based Collaborative Business Processes,» in *Advanced Information Systems Engineering*, 2019.

[19] E. F. e. C. R. Barbara Carminati, «Blockchain as a Platform for Secure InterOrganizational Business Processes,» in *IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, 2018.

[20] R. T. Fielding., «Architectural Styles and the Design of Network-based Software Architectures,» [Online]. Available: https://www.ics.uci.edu/˜fielding/ pubs/dissertation/rest_arch_style.htm#sec_5_2_1_1.

[21] X. X. R. G. A. P. J. M. Ingo Weber, «Untrusted Business Process Monitoring and Execution Using Blockchain,» in *International Conference on Business Process Management*, 2016.

[22] O. Pintado, «Caterpillar: A Blockchain-Based Business Process Management System,» in *Proceedings of the Demo Track and Dissertation Award of the 15th International Conference on Business Process Modeling (BPM 2017)*, 2017.

[23] J. R. V. J. G. E. H. Richard, Design Patterns, Addison-Wesley, 2002.