ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

SISTEMA DE LICITACIONES ELECTRÓNICAS PARA ADMINISTRACIONES PÚBLICAS CON USO DE BLOCKCHAIN Y SMART CONTRACTS

ELECTRONIC TENDERS SYSTEM FOR PUBLIC ADMINISTRATIONS WITH USE OF BLOCKCHAIN AND SMART CONTRACTS

Realizado por
PABLO HIJANO CALDERÓN

Tutorizado por
ISAAC AGUDO RUIZ
JOSÉ MARÍA ÁLVAREZ PALOMO
JUAN ANTONIO RÍOS PELÁEZ

Departamento
LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

UNIVERSIDAD DE MÁLAGA
MÁLAGA, FEBRERO 2019

Fecha defensa:
El Secretario del Tribunal

Resumen

Se ha desarrollado una aplicación web que se conecta a una red blockchain de Ethereum para manejar licitaciones y permitir a las personas interesadas participar en ellas. Un gestor es el encargado de crear una licitación con unos parámetros concretos.

Cualquier persona, ya sea la representante de una empresa o un autónomo, puede registrarse en la aplicación introduciendo unos datos básicos. Una vez registrada, puede acceder a las licitaciones activas que haya en ese momento y presentar una oferta a la licitación que vea conveniente. En base a los parámetros seleccionados y a las fechas estipuladas, la red elige de entre todos los participantes a un ganador. Cuando se haya seleccionado el ganador, se publicará en la misma licitación de forma pública.

Durante todo este proceso, cualquier ciudadano puede acceder a las licitaciones de forma pública y ver, una vez transcurrido el periodo de presentaciones, qué empresas participan en una licitación concreta. Cuando se decida el ganador, el ciudadano podrá verlo al mismo tiempo que las empresas.

Palabras clave

Licitación, Blockchain, Ethereum, Smart Contract, Solidity

Abstract

It has been developed a web application that connects itself to a Etherum blockchain network to manage tenders and to allow interested people to participate in these auction processes. A manager is in charge of creating a tender with a specific parameters.

Any person, either a representative of a company or a freelance, can register in the application by entering some basic data. Once this person is registered, access to the active tenders which are avaliable at that moment and it can submit an offer to any tender. Based on the selected parameters and the dates chosen, the network selects a winner from all candidates. When the winner is chosen, it will publish in the tender publicy.

Throughout this process, any citizen can access to the tender publicy and see which companies participates in any tender when presentation period is over. When a winner is chosen, the citizen will be able to see it at the same time than companies.

Keywords

Tender, Blockchain, Ethereum, Smart Contract, Solidity

Índice

1. INTRODUCCIÓN

En este apartado se comentará el contenido de la memoria, así como antecedentes, motivación, resumen, conceptos clave, objetivos, herramientas y tecnologías utilizadas.

1.1 ANTECEDENTES Y MOTIVACIÓN

Actualmente, los sistemas de licitaciones que utilizan la mayoría de entidades públicas consisten en un mecanismo basado en sobres. En estos se encuentra la información de las ofertas presentadas. Desde la apertura de los sobres, que pueden ser 3, hasta la propuesta del ganador pueden pasar hasta 20 días sin mucha complicación. Pueden existir subsanaciones de documentación administrativa, justificación de ofertas anormalmente bajas, aclaraciones de documentación, etc. El procedimiento en cuestión en el que se va a centrar este proyecto es el Procedimiento abierto simplificado, donde únicamente hay criterios objetivos de valoración automática y se unifica en un mismo sobre la documentación administrada y los criterios de adjudicación.

El problema viene de que se ha sabido que en el uso de este sistema se reproducen varios defectos. La falta de transparencia, los cambios de presupuestos a posteriori cuando una empresa se ha conocido ganadora, la lentitud para elegir ganador, el desperdicio de recursos, etc. En algunas entidades empiezan a apostar por sistemas informáticos para realizar los procedimientos relacionados con las licitaciones. Pero, de nuevo, se encuentran algunos problemas similares al sistema basado en sobres, como la falta de transparencia.

La motivación para realizar este proyecto es crear un sistema informático para dar solución a los problemas anteriormente mencionados. Se desarrollará un sistema que haga uso de una red blockchain, en este caso la de Ethereum[1]. Este sistema será el encargado de manejar los procesos de licitaciones, comunicándose con la blockchain, que es donde estarán almacenadas las licitaciones en sí. En este caso, dado que se trata de un sistema informático, sólo se fijará en los criterios objetivos a la hora de seleccionar un ganador.

Este proyecto forma parte de Impulso TFE[2] de la UMA, un programa que busca la relación entre estudiantes y empresas o entidades para la realización de trabajos de fin de estudios. En este caso, la entidad es la Diputación de Málaga. En su representación, Juan Antonio Ríos Peláez, responsable del Departamento Informático, será el encargado de validar los requisitos, la implementación y todo lo relacionado con las características y funciones del proyecto.

En Aragón se está desarrollando un sistema con funciones similares al que se busca desarrollar en este caso. Algunas ideas, como la comprobación de que una oferta no será modificada una vez se haya enviado, han servido de inspiración para usarlas en este mismo proyecto. Se mencionará más adelante ciertas características obtenidas de su proyecto.

1.2 OBJETIVOS

Este proyecto busca desarrollar una aplicación web que sirva de intermediario entre la red blockchain, donde estarán almacenadas las licitaciones, y cualquier usuario que quiera participar en las licitaciones o ver desde fuera cómo se van desarrollando.

Este sistema informático dará solución a los problemas mencionados. Para homogeneizar el proceso de transparencia en la contratación pública, la tecnología basada en smart contracts almacenará las licitaciones con toda su información y elegirá a un ganador, siendo público en la red blockchain. Tanto la información como el código en sí, son públicos. Por lo que, aunque se diera el caso de que algún futuro desarrollador quisiera corromper el sistema en busca de seleccionar un ganador por intereses personales, el código estaría en la red y podría ser visualizado para comprobar su veracidad.

Otro objetivo de este proyecto es que un licitador no pueda modificar su oferta una vez la ha enviado. Así se evitaría la subida presupuestaria una vez se ha conocido ganadora para conseguir más beneficios, en detrimento de las arcas públicas. Dentro de este smart contract que representa a la licitación, se almacenará una cadena de texto que será un hash de los datos de la oferta. De modo que, si en algún momento un licitador cambia un solo carácter de su oferta, el sistema lo sabrá al hacer el hash y compararlo con el almacenado en la red.

1.3 METODOLOGÍA

En este apartado se explicará cuál ha sido la metodología utilizada durante el desarrollo del proyecto.

La metodología llevada a cabo para el desarrollo ha sido una metodología iterativa. Empezará por una fase inicial en la que se decidirán los objetivos fundamentales en un DGR (Documento General de Requisitos). Este DGR deberá estar aprobado por ambas partes, tanto por el desarrollador como por el representante, y funcionará como un contrato para validar si el proyecto final es lo que se esperaba. Los aspectos principales encontrados en el DGR se comentarán en el apartado de análisis de requisitos

Tras haber realizado el DGR, se procederá a crear bocetos de interfaz de usuario (o mockups) que servirán de base para tener una idea de cómo será el producto final, sobre todo las vistas. En estos mismos bocetos se podrán ver las funcionalidades principales de la aplicación declaradas en el DGR.

Con los bocetos de interfaz de usuario hechos, se realizará una fase de diseño donde se estudiará cómo almacenar los datos y dónde, ya que algunos se almacenarán en la base de datos de MongoDB y otros en la red blockchain.

Tras la fase de diseño, se empezará un desarrollo basado en iteraciones en las que se irán implementando un conjunto de requisitos en cada iteración. Se realizarán sprints en los que al final de cada uno se harán varias pruebas a las partes ya desarrolladas y se pensará en el próximo conjunto de funcionalidades a desarrollar. Durante el desarrollo, se irán desarrollando a la vez tanto el frontend como el backend, ya que así el representante podrá ir viendo cómo va avanzando el proyecto de forma más intuitiva.

1.4 RESUMEN Y CONCEPTOS CLAVE

En este apartado se tratará de explicar cómo funciona un proceso de una licitación, qué es una licitación y varios conceptos clave que aparecerán mencionados durante toda la memoria.

Licitación: subasta que realiza una entidad pública en la que se propone un proyecto que se quiere llevar a cabo para que cualquier empresa o autónomo interesado pueda participar enviando una oferta para realizar dicho proyecto. Tras un proceso de selección se elige al ganador en base a diversos criterios especificados con anterioridad.

Licitador: empresa o autónomo que participa en una licitación. También puede referirse a la entidad que saca a subasta la licitación, pero en el ámbito de esta memoria se utilizará su primera acepción.

Blockchain: tecnología que usa una red de nodos, ordenadores, para verificar toda la información que se almacena en ella. Se guarda una cadena de bloques que es verificada por toda la red. Todos los nodos contienen la misma información y cada vez que se añade un bloque nuevo es validado por todos los nodos.

Ethereum: plataforma blockchain que permite desplegar aplicaciones distribuidas a través de smart contracts.

Smart contract: o contrato inteligente. Se trata de un programa que se ejecuta en la red de Ethereum.

Mencionados los conceptos clave, se explicará cómo es un proceso de licitación.

En los procesos de licitaciones que se encargará de manejar el sistema fruto de este proyecto existen 2 fechas y períodos importantes. Fecha de fin de presentaciones y fecha de evaluación. Y también se hablará de período de presentaciones y período de evaluación. El período de presentaciones transcurre desde que se crea la licitación hasta la fecha de fin de presentaciones. Y el período de evaluación transcurre desde la fecha de fin de presentaciones hasta la fecha de evaluación. Una vez explicado esto, ya se puede desarrollar el ciclo de un proceso de licitación.

Primeramente, se crea una licitación por parte de un gestor autorizado. En ese momento la licitación se encuentra en período de presentaciones. Es aquí cuando cualquier licitador que quiera participar en la licitación debe enviar una oferta en un documento XML que podrá descargar desde la propia aplicación. Debe rellenar el documento exponiendo las características de su oferta. Es importante que el licitador guarde este documento porque tendrá que enviarlo de nuevo más adelante. Al enviar la oferta, se almacena el hash de esta en la blockchain.

Llegada la fecha de fin de presentaciones, acaba el período de presentaciones y comienza el de evaluación. En este nuevo período, será necesario que el licitador envíe de nuevo la oferta que envió inicialmente en el período de presentaciones. Al enviar de nuevo la oferta, se realizará el hash de esta y se comprobará con el almacenado en la blockchain. Si no coincide, se avisará al licitador de que el documento ha sido modificado y hasta que no envíe el original no puede seguir participando en la licitación. Si finaliza el período de evaluación y no lo ha enviado, no podrá resultar ganador de la licitación. En caso de que haya enviado la misma oferta que envió originalmente en el período de presentaciones, se le comunicará que la oferta es la misma y que no es necesario que haga nada más. En este momento, el licitador únicamente tendrá que esperar a la fecha de evaluación.

Una vez llegada la fecha de evaluación, el período de evaluación ha acabado. En este momento, la red blockchain, mediante una fórmula y teniendo en cuenta los criterios que eligió el gestor en el momento de crear la licitación, elige un ganador. El ganador solo podrá ser alguno de los que envió la misma oferta tanto en el período de presentaciones como en el de evaluación. En cuanto se elige al ganador, este es publicado en la licitación que se muestra en la aplicación. Se puede ver una representación de este proceso en la figura 1.1.

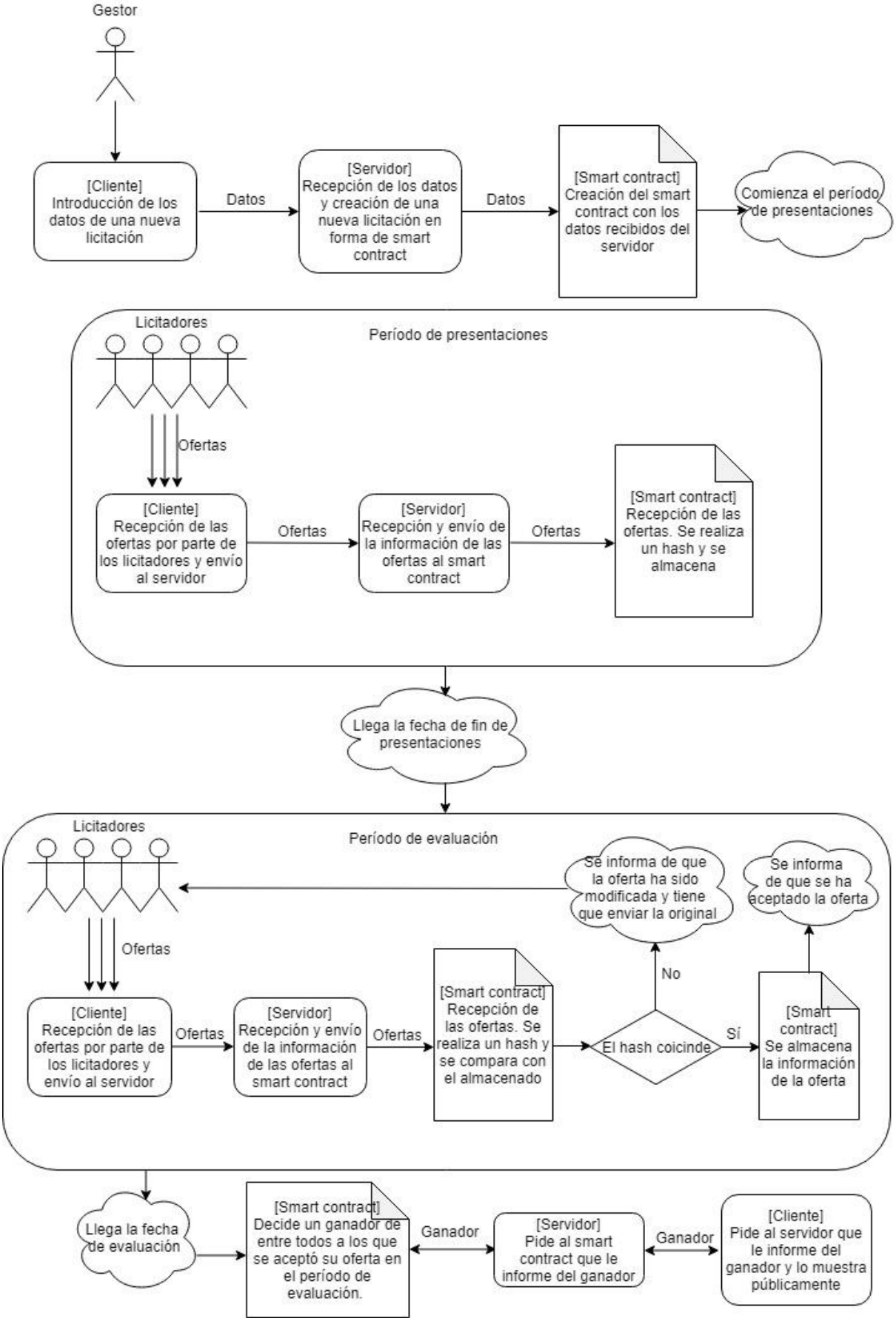


Figura 1.1: proceso de licitación en el sistema

1.5 TECNOLOGÍAS Y HERRAMIENTAS UTILIZADAS

En el apartado actual se mencionarán las tecnologías y herramientas utilizadas durante todo el proceso de desarrollo, así como la versión utilizada en los casos requeridos. Es importante el tema de tener las mismas versiones que las aquí mencionadas, puesto que podría ser que versiones anteriores o posteriores eliminen o no tengan implementadas aún características que han sido usadas para el desarrollo del proyecto.

1.5.1 BITBUCKET

Es una alternativa al famoso GitHub, con la característica principal de que permite tener repositorios privados de forma gratuita[3].

Figura 1.2: logo de Bitbucket

Puede ser usado, como otros tantos, por la interfaz de la línea de comandos de Git. Por lo que hace muy sencillo su uso conociendo los comandos básicos. Se ha utilizado durante todo el proyecto para ir guardando los cambios realizados conforme avanzaba el desarrollo y tener una copia en todo momento en la red. Todo su uso se ha basado en la línea de comandos.

1.5.2 NODE.JS

Es un entorno de ejecución de JavaScript diseñado para la parte del servidor. Está desarrollado para soportar eventos asíncronos con muchas peticiones de forma concurrente. Toda su filosofía se basa en la asincronía, de modo que hay que estar muy atento a la hora de desarrollar para no dejar ningún hilo de ejecución que se quede vagando por la CPU. La versión de Node.js[4] utilizada ha sido la 8.11.3.

Figura 1.3: logo de Node.js

En este entorno se ha desarrollado y se ejecutará la API REST que aceptará peticiones del cliente y se comunicará con la red blockchain. Npm es el manejador de paquetes por defecto de Node.js. Nos permitirá instalar multitud de herramientas creadas para usarlas en el servidor. La versión de npm utilizada es la 5.6.0. Se comentarán varias de ellas más adelante, pero cabe recalcar aquí por su importancia, Express[5]. Se trata de un framework que entre sus muchas características, nos facilita mucho el desarrollo de la API REST en sí. Proporciona funciones, como la de *Router*, que nos permite crear rutas para las peticiones HTTP de forma sencilla. La versión de Express utilizada es la 4.16.4.

1.5.3 REACT

React[6] (o ReactJS) es una librería de JavaScript para crear interfaces de usuario. Entre sus muchas características, una muy interesante es que permite cambiar datos mostrados en pantalla al instante, sin tener que recargar toda la vista entera.

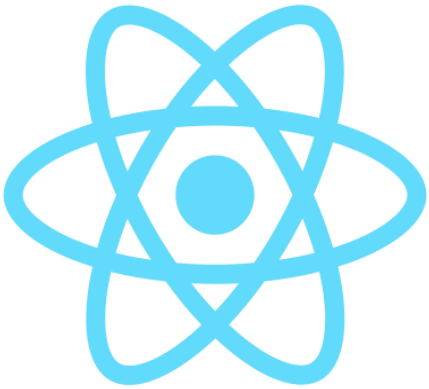


Figura 1.4: logo de React

Su uso de las vistas a modo de componentes, permite una interacción y reutilización del código muy eficiente. Cabe comentar que para realizar las peticiones al servidor se ha utilizado axios[7], una librería que incluye funciones para enviar peticiones HTTP al servidor y devolver el resultado de nuevo al cliente. La versión utilizada es la 16.6.0.

1.5.4 MONGODB

MongoDB[8] es un gestor de base de datos no relacional orientado a documentos. Almacena los datos en documentos JSON, de manera que hace bastante ameno el interactuar con ellos. Al tratarse de una base de datos NoSQL puede manejar grandes cantidades de datos de manera eficiente. La versión utilizada es la 4.0.3.

Figura 1.5: logo de MongoDB

En este caso, se ha usado una herramienta para facilitar el desarrollo. Se trata de Mongoose[9]. Este nos permite modelar los datos de MongoDB como si fueran objetos e interactuar con ellos como tal. La versión de Mongoose utilizada es la 5.2.8.

1.5.5 GANACHE

Se trata de una red blockchain Ethereum privada que se puede ejecutar en el propio ordenador. Simula una red blockchain, de forma que se puede utilizar para realizar pruebas de forma completamente gratuita.

Figura 1.6: logo de Ganache

Ganache[10] forma parte de un framework llamado Truffle (o Truffle Suite) que contiene dos componentes más, Truffle y Drizzle. El primero proporciona herramientas para lanzar, testar y comunicarnos con los smart contracts. El segundo es un conjunto de librerías para el frontend. En un principio, se iba a utilizar Truffle, el componente en sí, para este proyecto, pero tras varios problemas se descartó como opción. La versión de Ganache utilizada es la 1.2.1.

1.5.6 WEB3

Web3[11] (o web3.js) es un conjunto de librerías que permiten conectarnos a una red blockchain de Ethereum y comunicarnos con ella. Funciona tanto con redes privadas, como Ganache, como con cualquier red pública de Ethereum.

Figura 1.7: logo de Web3

Ofrece muchas funciones y todas ellas vienen bien detalladas en su documentación, que incluye muchos ejemplos, de forma que facilita bastante el aprendizaje. La versión de web3 utilizada es la 0.20.0.

1.5.7 SUBLIME TEXT

Editor de texto enfocado al desarrollo de código que funciona con multitud de tipos de archivo. Para el caso, se ha utilizado la versión 2, ya que es la que estaba instalada en el ordenador.

Figura 1.8: logo de Sublime Text

En un principio, se pensaba utilizar WebStorm, un IDE para código JavaSript, tanto en el servidor como en el cliente, que ofrece muchas utilidades para facilitar el desarrollo. El problema es que el ordenador en el que se ha desarrollado el proyecto tiene unos componentes muy poco potentes, de forma que no funcionaba

correctamente. Ese es el motivo principal de que se haya utilizado Sublime Text[12], ya que este editor es muy ligero y no ralentiza la implementación.

El desarrollo de los smart contracts en la blockchain se realiza con el lenguaje de programación Solidity[13]. De este lenguaje se hablará más en detalle en el apartado de implementación.

Durante el desarrollo del proyecto se han utilizado más herramientas que se comentarán más adelante, cuando surjan características o resoluciones de conflictos relacionados con estas. No se han incluido en esta parte, ya que no se han usado tanto como las aquí mencionadas. Se han usado más en ciertos momentos puntuales o durante etapas muy concretas del desarrollo.

2. ANÁLISIS DE REQUISITOS

En este apartado vamos a declarar qué se espera que haga el sistema una vez finalizada su implementación. Forma parte de un DGR que funcionará a modo de contrato con el representante. El DGR se puede ver en el Anexo I.

2.1 CONSIDERACIONES PREVIAS

Se busca realizar una aplicación web que se encargue de enviar y recibir datos con una red blockchain. Se pensó en Ethereum, puesto que incorpora los smart contracts. Estos son contratos inteligentes que permiten ejecutar código en la red blockchain, una red descentralizada. El papel de la red será el de almacenar licitaciones y las ofertas presentadas a estas. Llegada una fecha, decidirá un ganador siguiendo una fórmula de puntuación.

Habrá 2 tipos de usuarios principales. Aquellos que podrán participar en las licitaciones, una vez se hayan registrado en la web, y aquellos que observarán desde fuera cómo se van desarrollando los procesos de las licitaciones, pero sin participar. Por comodidad, a los primeros los llamaremos *licitadores* y a los segundos simplemente *usuarios*. Además, existirá un rol superior, que es el de *gestor*. Él será el encargado de crear las licitaciones con los parámetros convenientes.

Cabe comentar también que hay funciones o características que no aparecerán en el DGR ni en estos requisitos, pero que ha sido necesario implementar también. Se comentarán a posteriori en esta memoria.

2.2 REQUISITOS FUNCIONALES

RF1. Registrar un nuevo licitador.

El sistema permitirá registrar nuevos licitadores a partir de unos datos pedidos.

RF2. Acceso por parte de un licitador.

El sistema permitirá acceder, mediante un sistema de acceso, a un licitador para para que pueda hacer uso del sistema.

RF3. Configurar una nueva licitación.

El sistema permitirá la creación de licitaciones por parte de un gestor registrado.

RF4. Presentar propuesta a una licitación.

El sistema permitirá la recepción de archivos por parte de un licitador, de los cuales únicamente almacenará su hash y algunos datos para identificarlo y así poder incluirlo en la red blockchain. Estos datos no tendrán información de la oferta en sí.

RF5. Enviar propuesta para su valoración.

El sistema permitirá enviar la oferta por parte de un licitador. Si el hash de esta concuerda con el almacenado en la red blockchain, se procederá a su evaluación.

RF6. Ver ofertas presentadas a una licitación.

El sistema permitirá a cualquier usuario ver las ofertas presentadas a una licitación una vez finalizado el periodo de presentación de ofertas.

RF7. Seleccionar oferta ganadora.

El sistema, mediante la blockchain, seleccionará la oferta ganadora en base a la fórmula utilizada para la valoración.

2.3 REQUISITOS NO FUNCIONALES

RNF1. Función hash para comparación de documentos.

Se utilizará una función hash para comparar los documentos enviados por parte del cliente durante los procesos de presentación y evaluación.

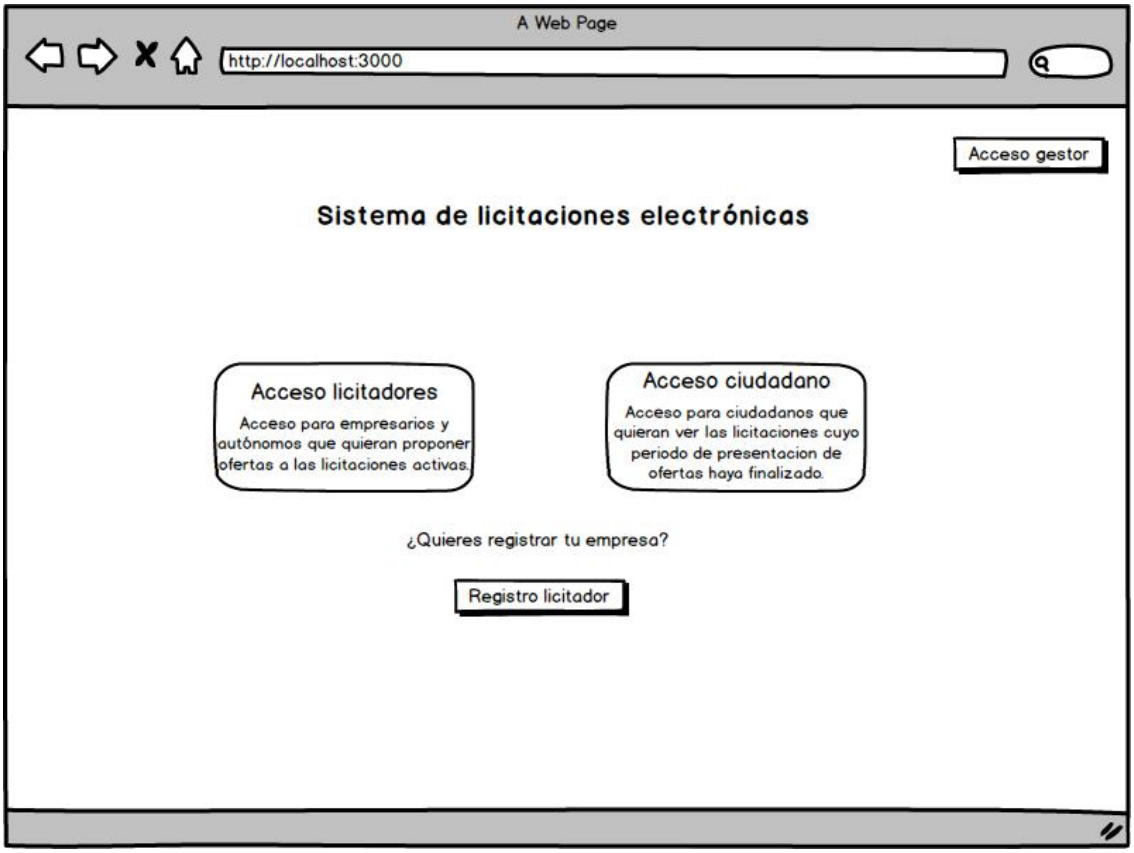
RNF2. Uso de la red Ethereum.

Se utilizará la red de blockchain Ethereum y los smart contracts para algunos requisitos del proyecto.

3. BOCETOS DE INTERFAZ DE USUARIO

En esta sección mostraremos los bocetos de interfaz de usuario creados para tener una versión inicial de la apariencia de la aplicación. En ellos también se dejan ver las funcionalidades de la aplicación. Para la realización de estos bocetos se ha utilizado la herramienta Balsamiq Mockups 3[14].

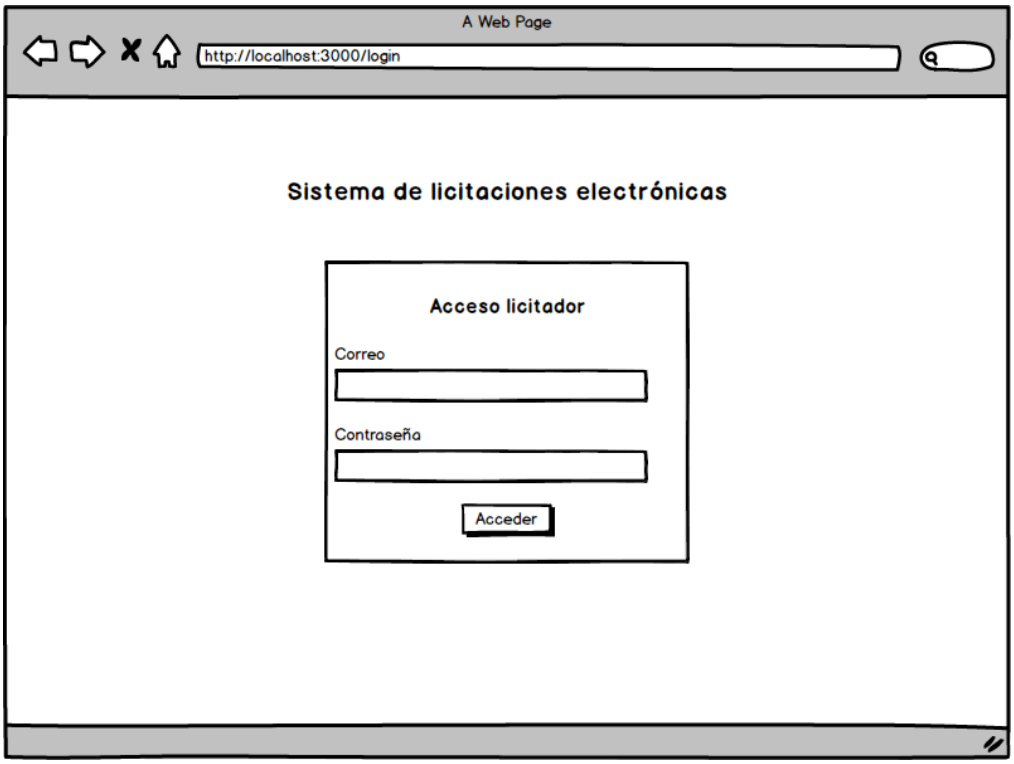
Boceto 1 – Pantalla de inicio



Boceto 3.1: pantalla de inicio

Esta será la vista principal que tendrá cualquier usuario que acceda a la aplicación.

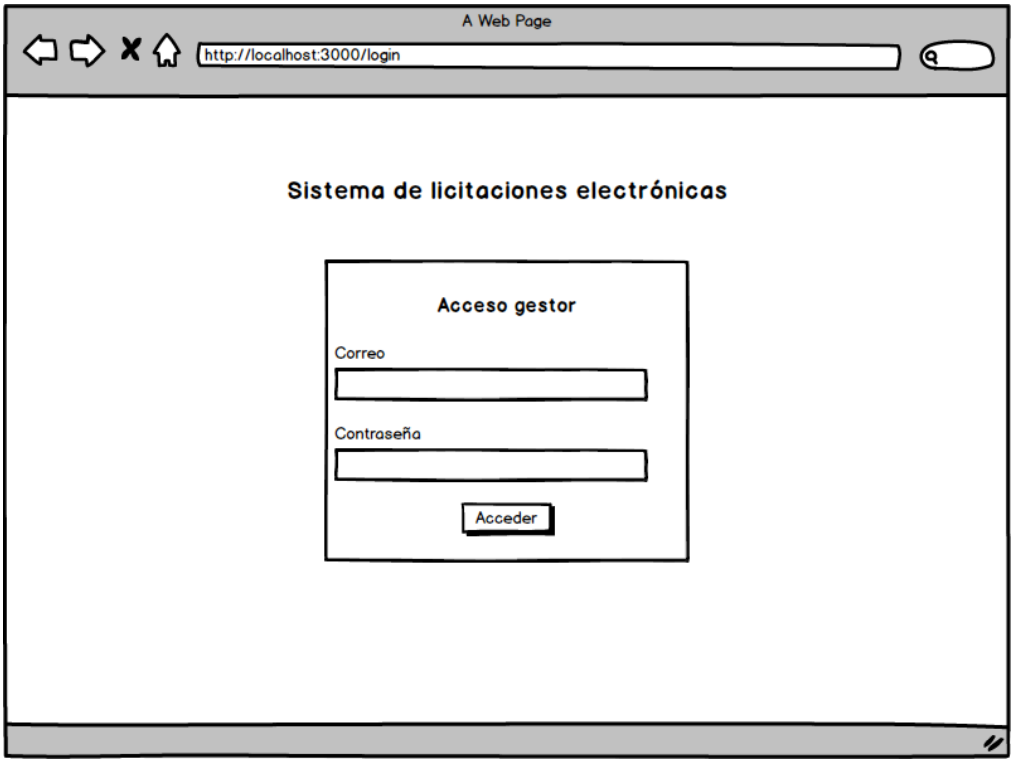
Boceto 2 – Acceso de licitador



Boceto 3.2: acceso de licitador

Vista de acceso para los licitadores.

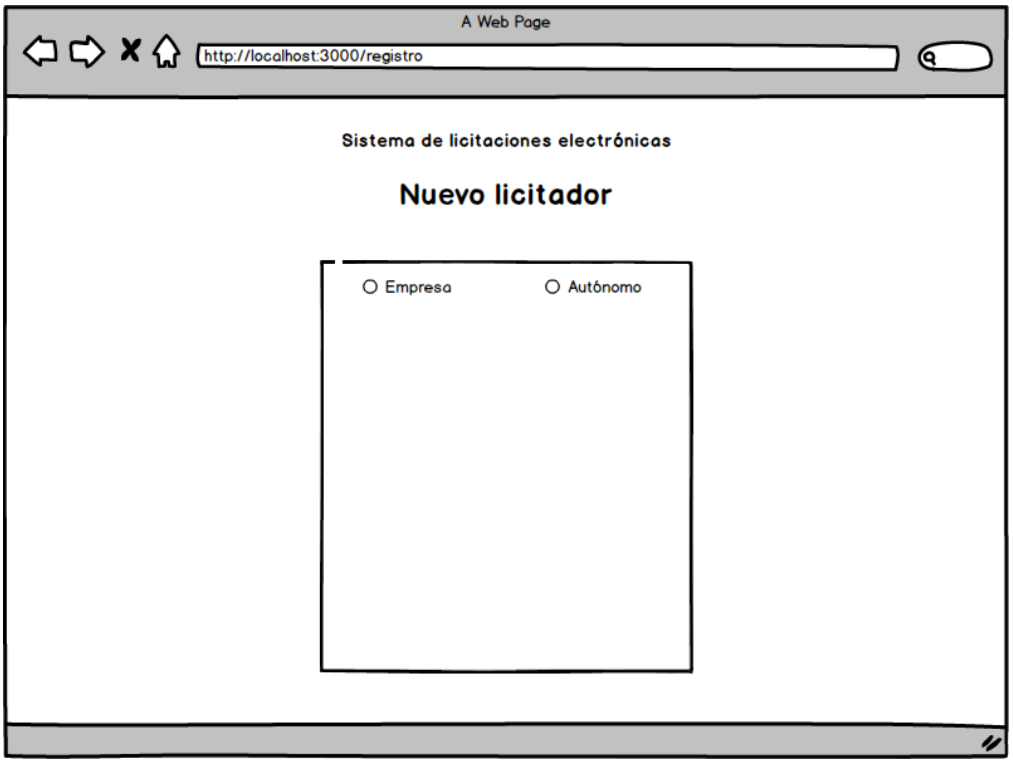
Boceto 3 – Acceso de gestor



Boceto 3.3: acceso de gestor

Vista de acceso para un gestor autorizado.

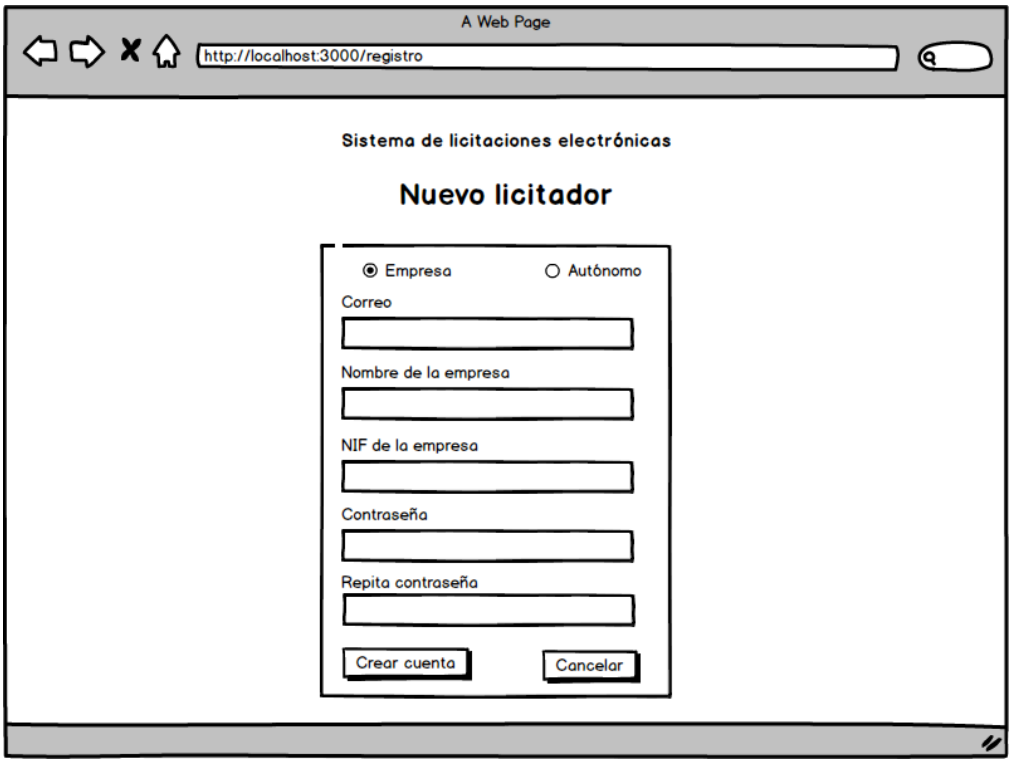
Boceto 4 – Registro de licitador



Boceto 3.4: registro de licitador

Vista del registro de licitador cuando aún no hay selección por el usuario.

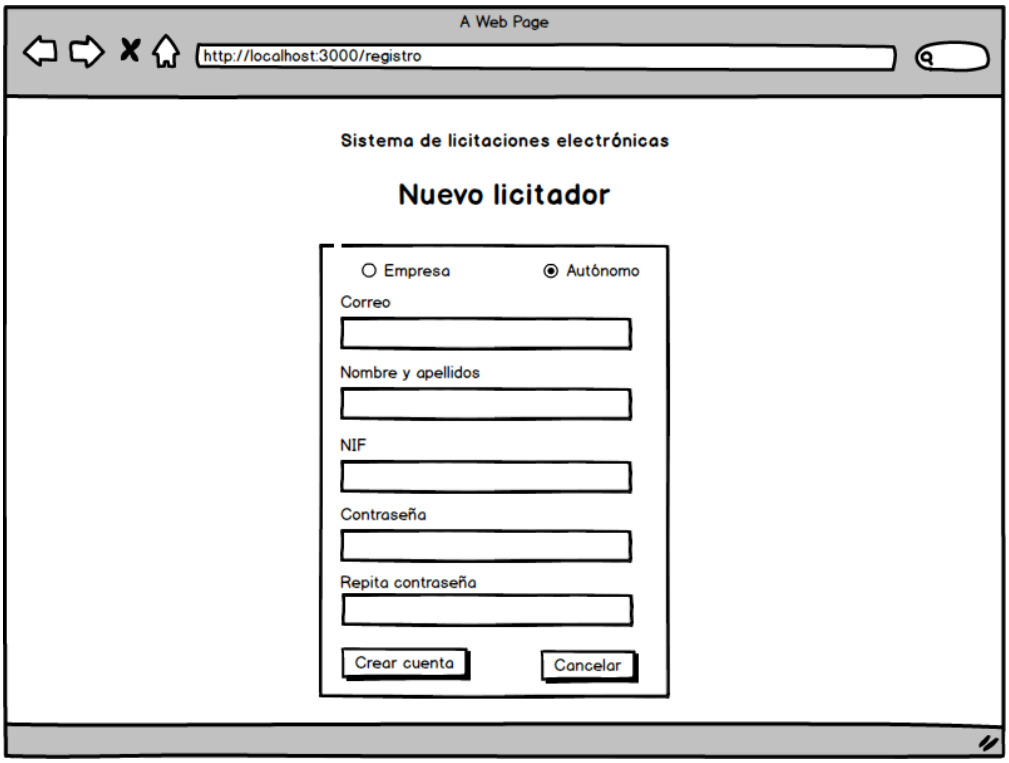
Boceto 5 – Registro de licitador de empresa



Boceto 3.5: registro de licitador de empresa

Vista del registro de licitador cuando se selecciona el tipo Empresa.

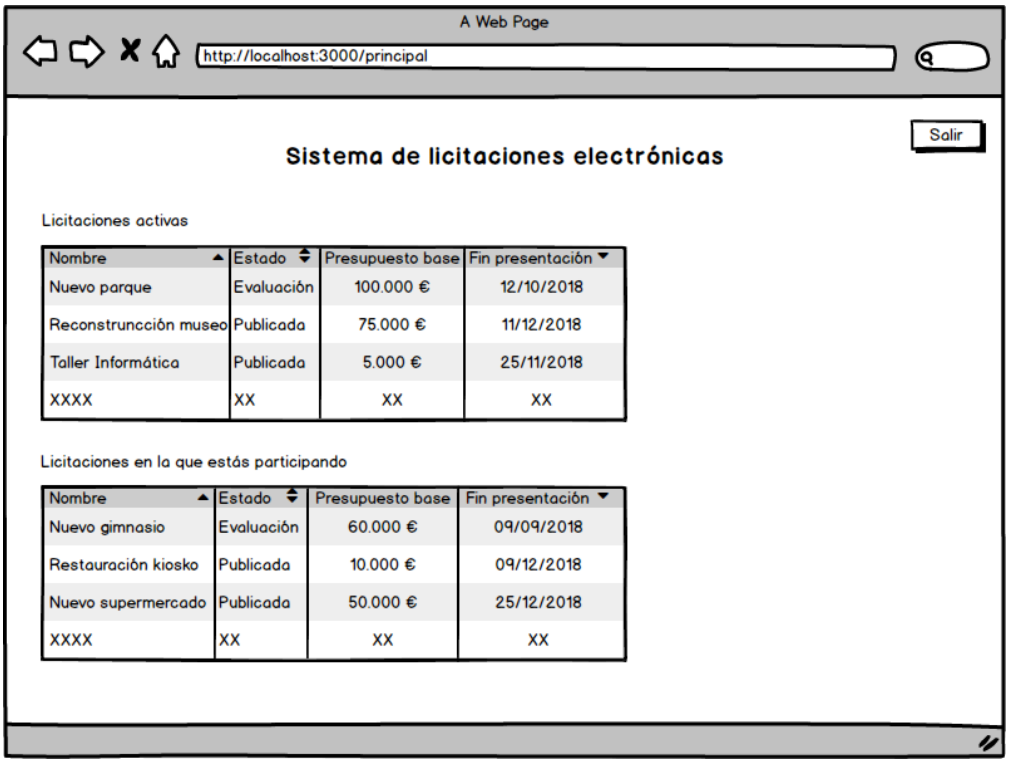
Boceto 6 – Registro de licitador de autónomo



Boceto 3.6: registro de licitador de autónomo

Vista del registro de licitador cuando se selecciona el tipo Autónomo.

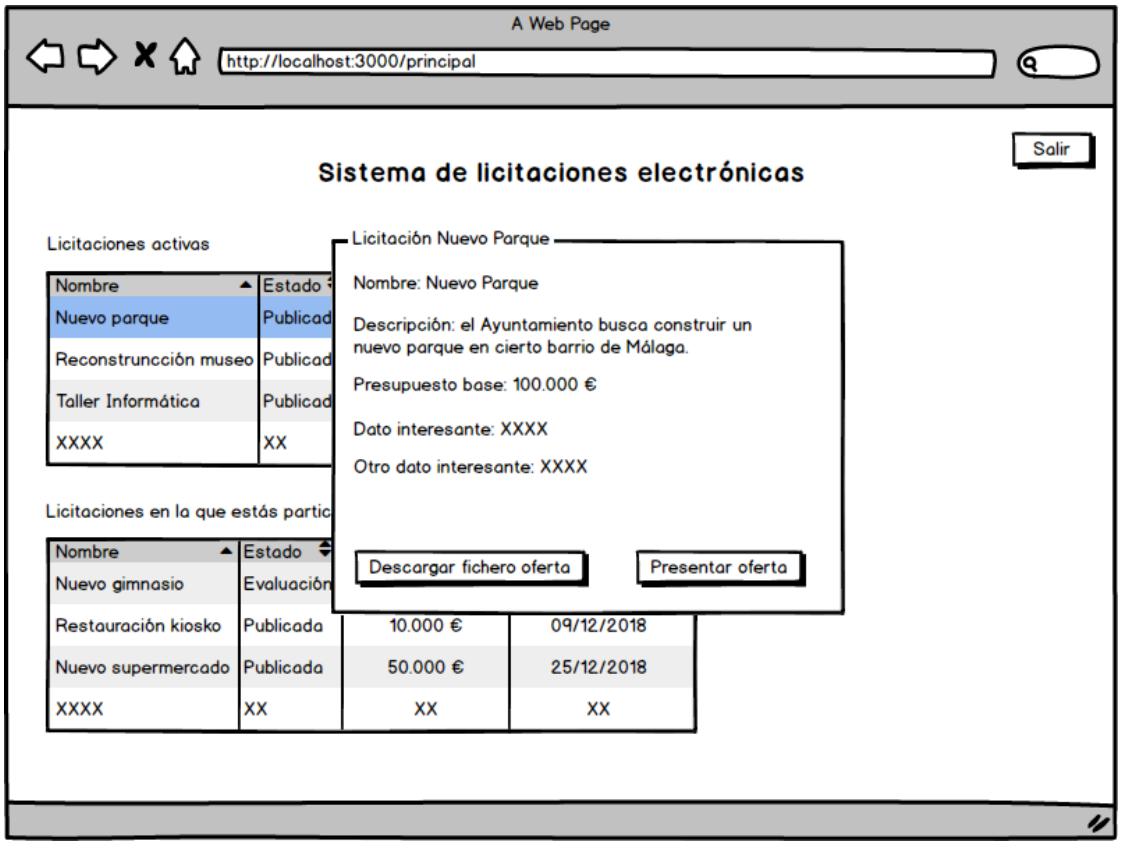
Boceto 7 – Vista principal del licitador



Boceto 3.7: vista principal del licitador

Vista principal del licitador una vez ha accedido a la aplicación.

Boceto 8 – Licitación activa seleccionada

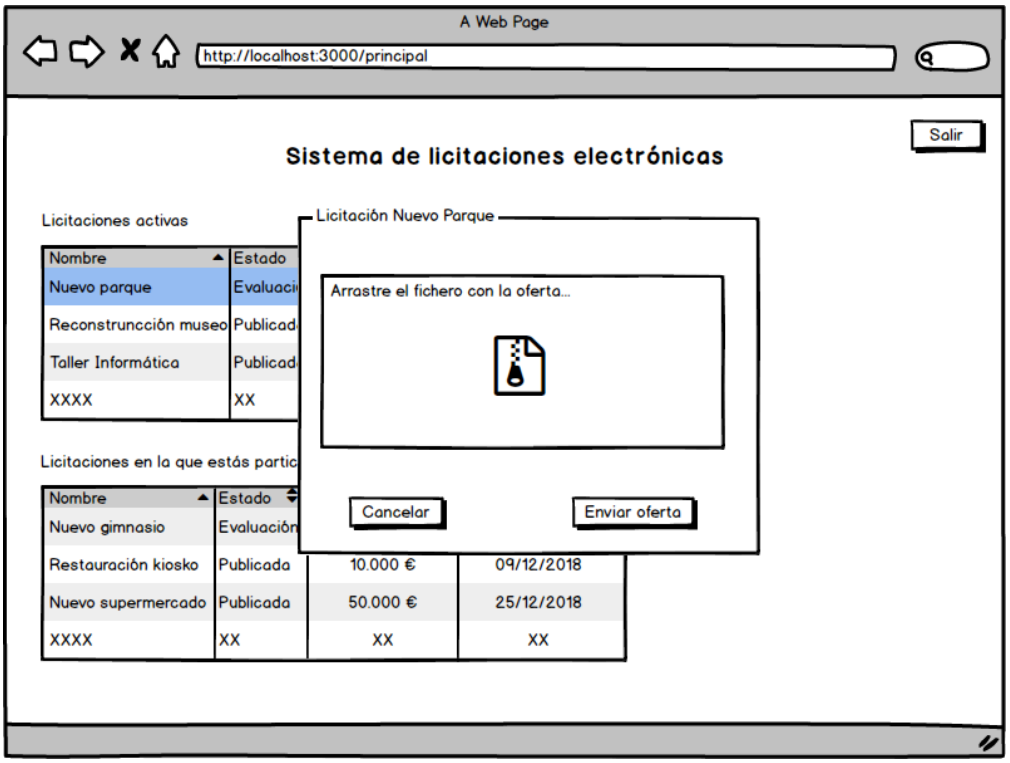


Boceto 3.8: licitación activa seleccionada

Vista de la aplicación cuando se ha seleccionado una licitación activa. Al ser una licitación activa en período de presentaciones de ofertas, no aparecen aún las empresas que están participando. Se expone así por petición del colaborador externo, ya que nos informa de que en esta fase del proceso pueden existir malas prácticas por parte de algunas empresas al conocer al resto que están participando.

Como también podemos observar, desde esta vista, se nos permitirá descargar un fichero para presentar la oferta. Se explicará más sobre este fichero a posteriori. Una vez rellenado el fichero, se podrá presentar pinchando en Presentar oferta.

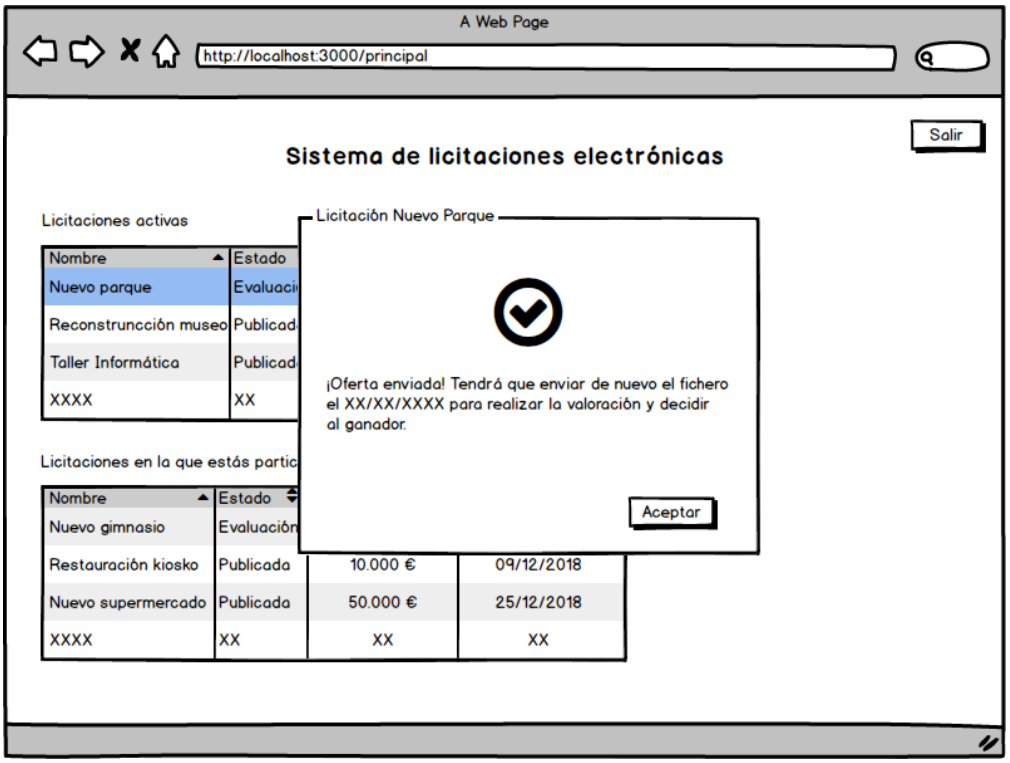
Boceto 9 – Licitación activa seleccionada presentando oferta



Boceto 3.9: licitación activa seleccionada presentando oferta

Vista de una licitación activa seleccionada cuando vamos a presentar la oferta.

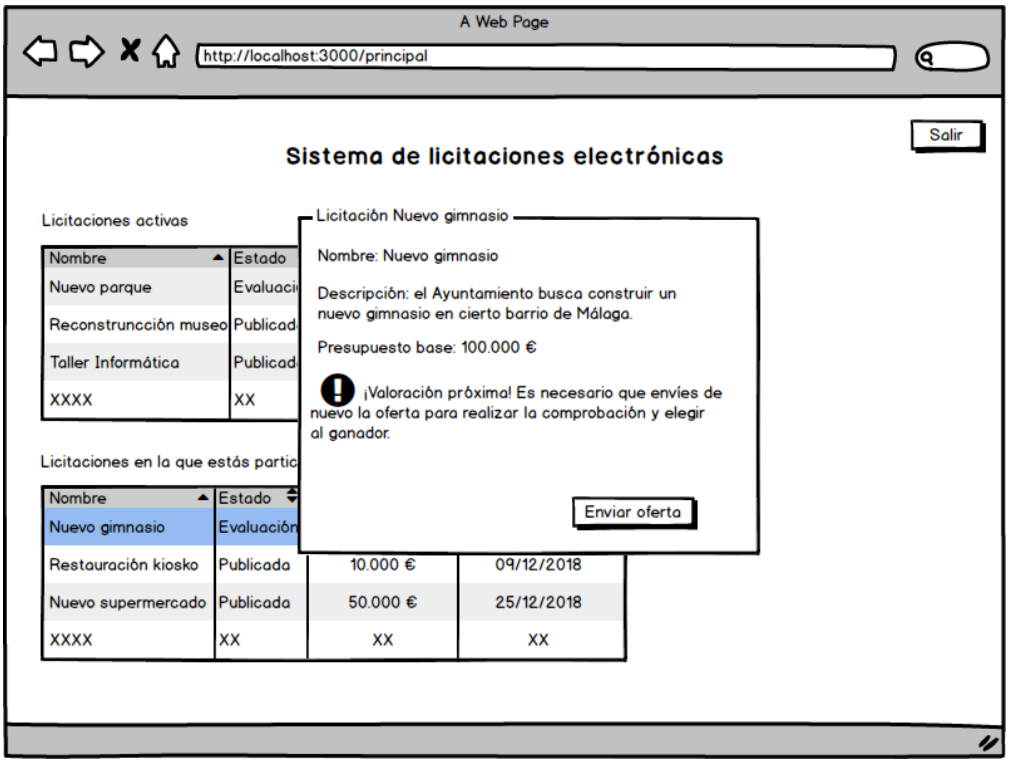
Boceto 10 – Licitación activa seleccionada con oferta enviada



Boceto 3.10: licitación activa seleccionada con oferta enviada

Vista de una licitación activa seleccionada cuando hemos enviado la oferta.

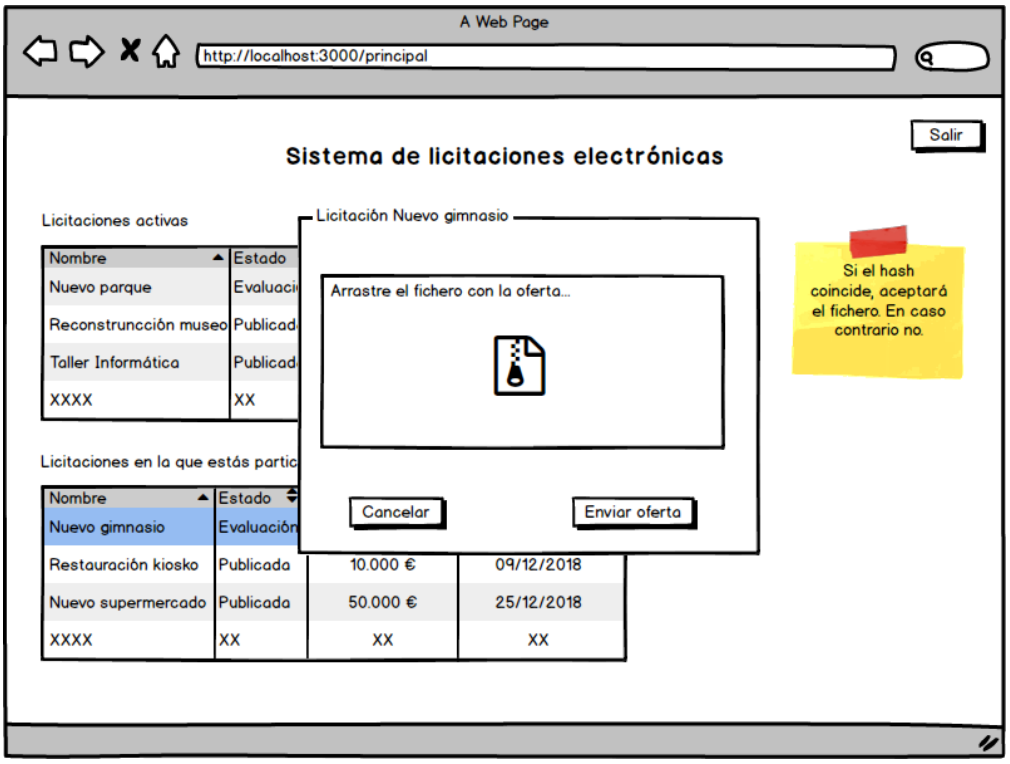
Boceto 11 – Licitación en período de evaluación



Boceto 3.11: licitación en período de evaluación

Vista de una licitación cuando se encuentra en el período de evaluación.

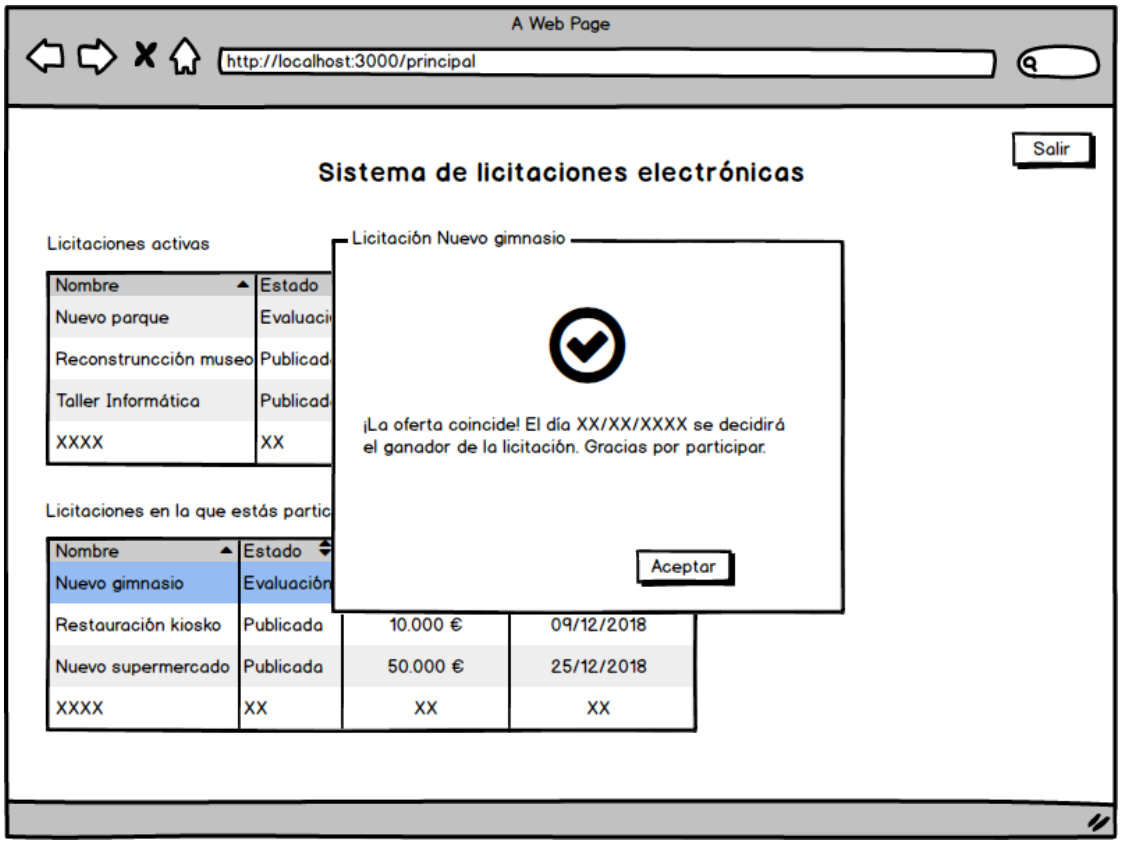
Boceto 12 – Licitación en período de evaluación enviando oferta



Boceto 3.12: licitación en evaluación enviando oferta

Vista de una licitación en período de evaluación cuando se está enviando una oferta. Como ya indica la nota del boceto, sólo se aceptará el fichero si su hash coincide con el almacenado en la red blockchain. En caso de que no coincida se le indicará al licitador que el archivo ha sido modificado y que envíe el original si quiere poder seguir participando en la licitación. En los dos siguientes bocetos veremos ambas posibilidades.

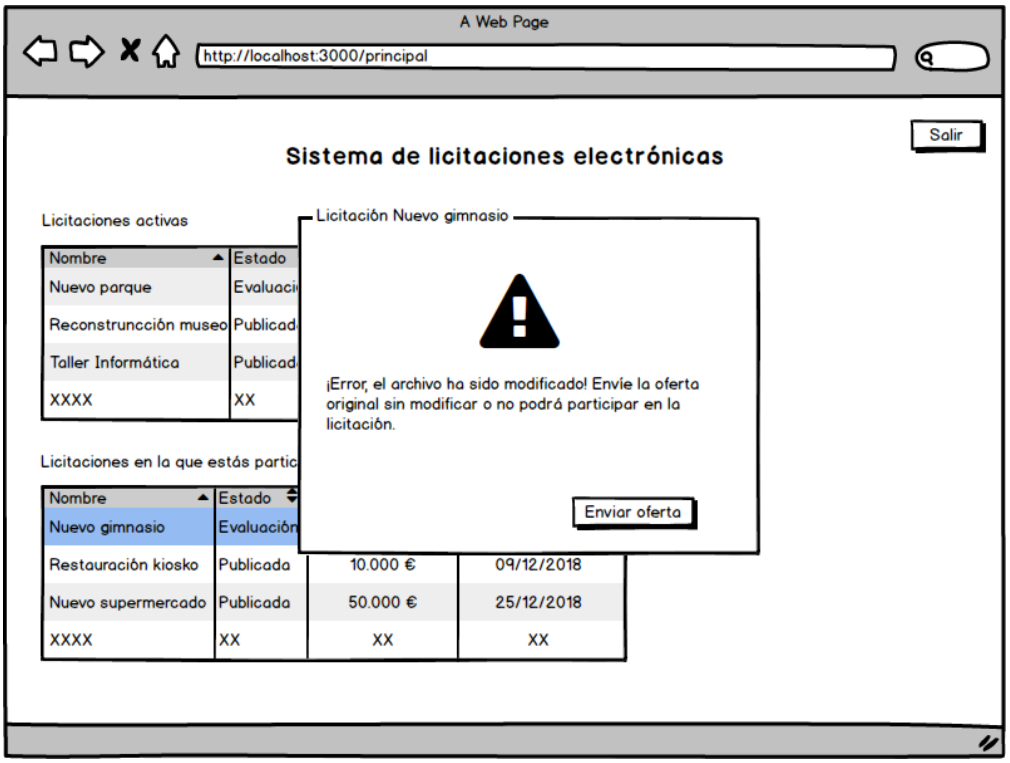
Boceto 13 – Oferta enviada en período de evaluación con archivo original



Boceto 3.13: oferta enviada en período de evaluación con archivo original

Vista de la licitación en período de evaluación cuando se ha enviado el mismo archivo que se envió en el período de presentaciones. Se ha comprobado el hash con el almacenado en la red blockchain y han coincidido.

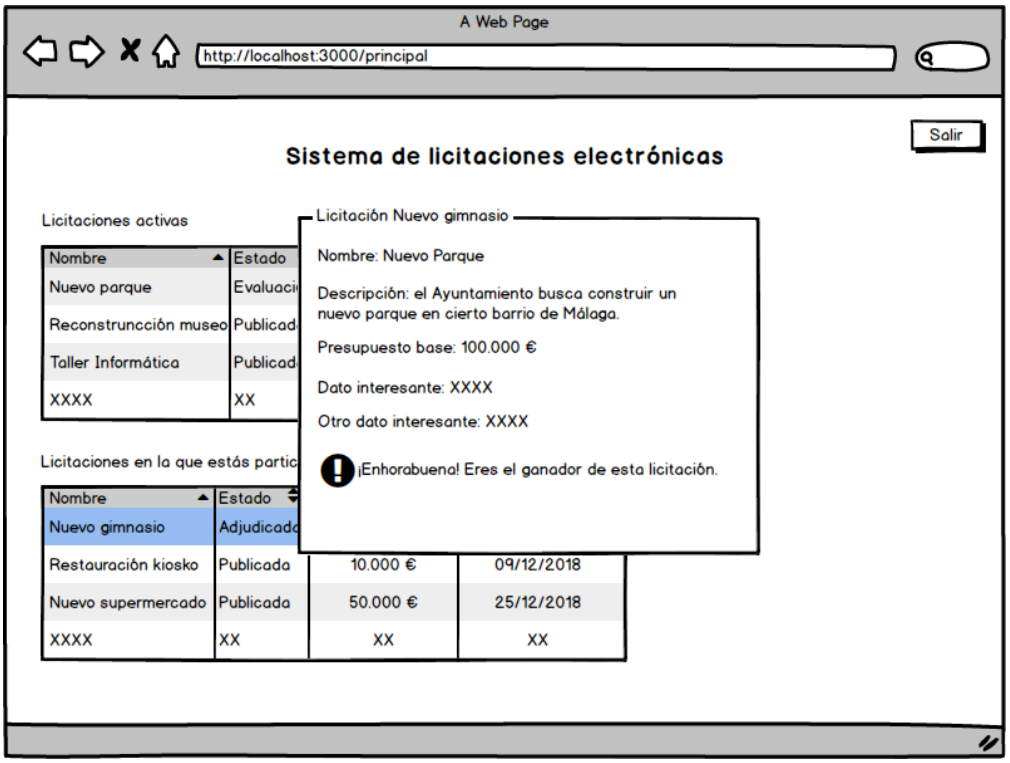
Boceto 14 – Oferta enviada en período de evaluación con archivo modificado



Boceto 3.14: oferta enviada en período de evaluación con archivo modificado

Vista de la oferta enviada en período de evaluación con el archivo modificado.

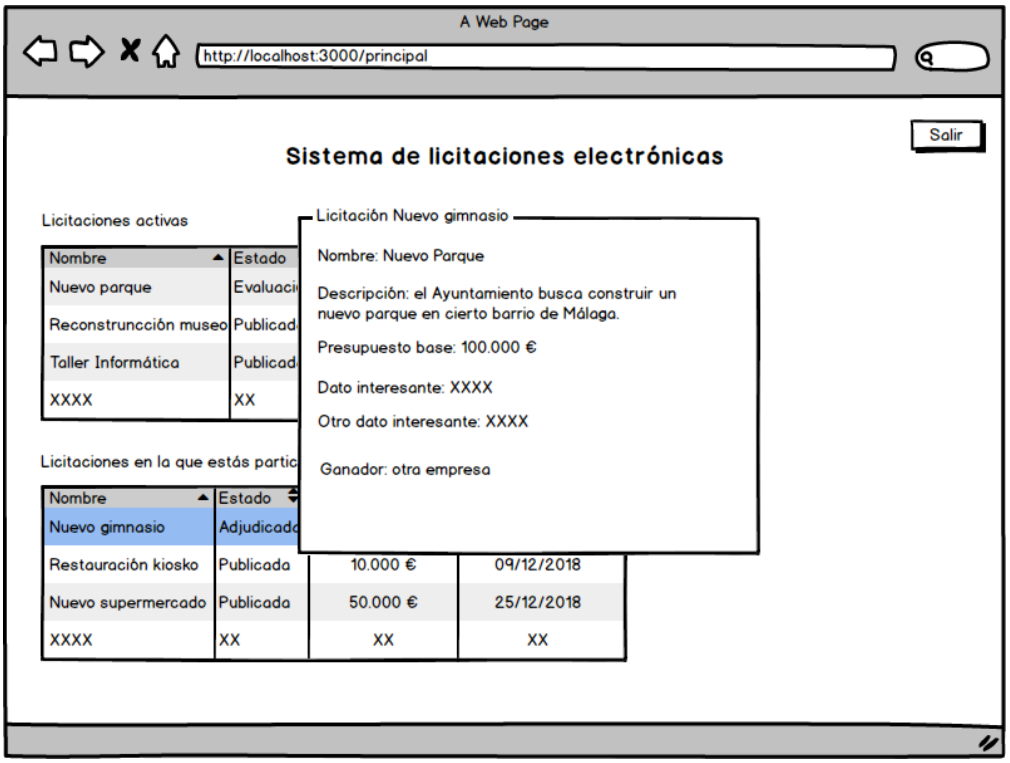
Boceto 15 – Licitación adjudicada con empresa ganadora



Boceto 3.15: licitación adjudicada con empresa ganadora

Vista de una licitación adjudicada cuando la empresa ha resultado ganadora.

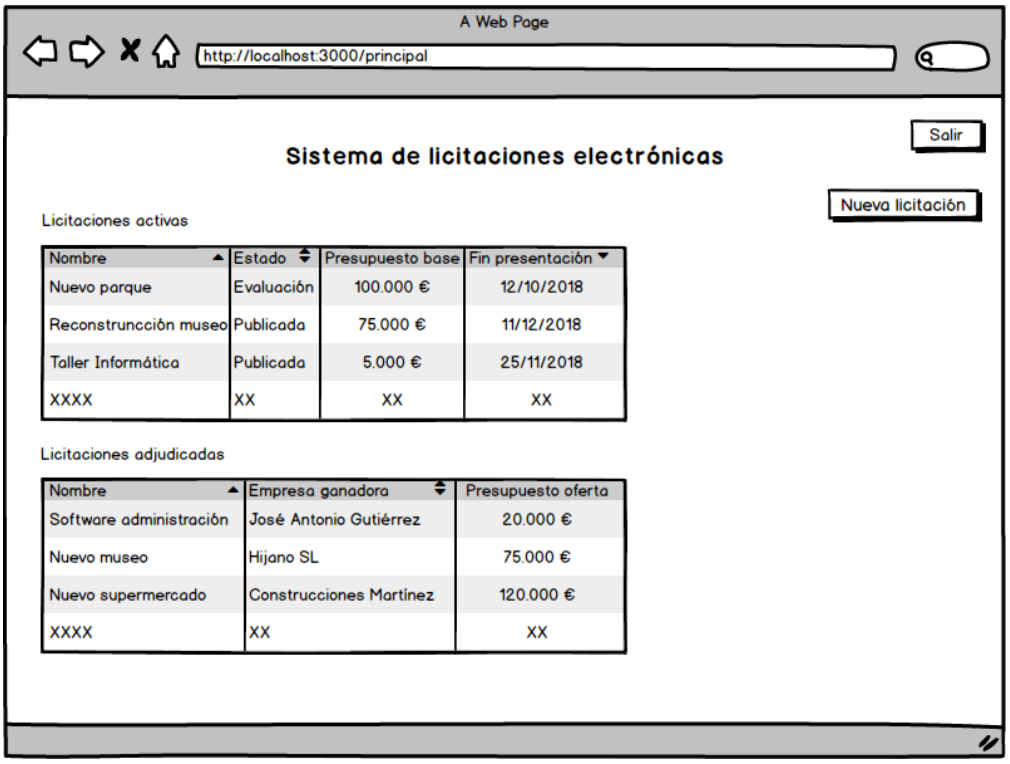
Boceto 16 – Licitación adjudicada con empresa no ganadora



Boceto 3.16: licitación adjudicada con empresa no ganadora

Vista de una licitación adjudicada cuando la empresa no ha sido la ganadora.

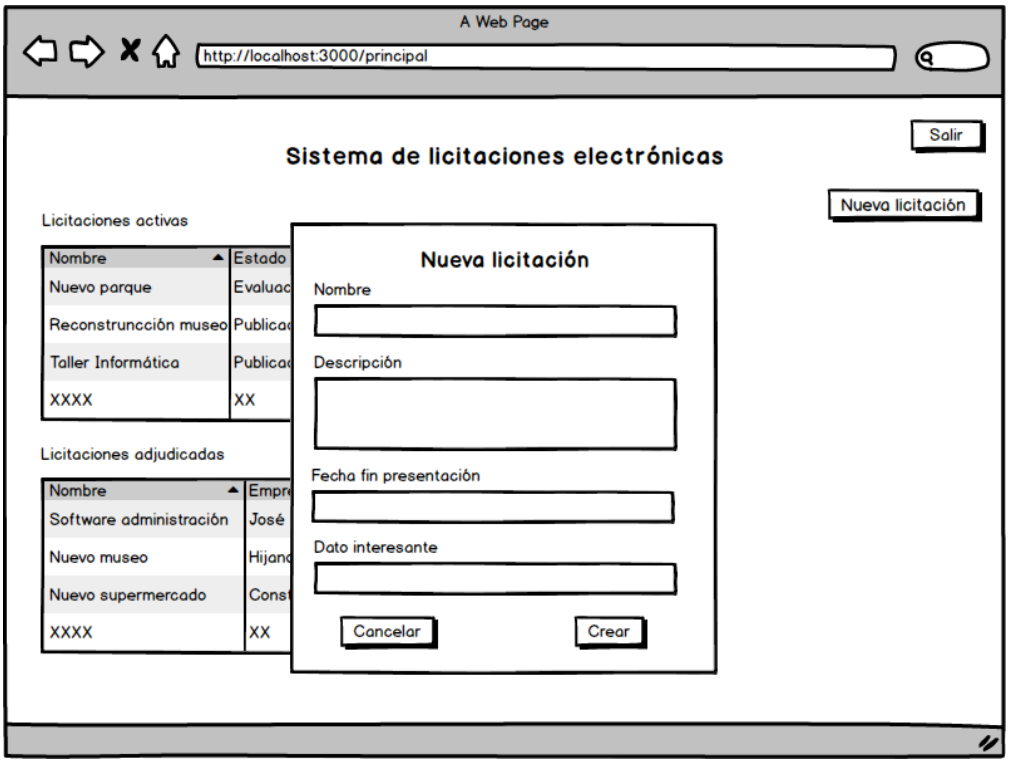
Boceto 17 – Vista principal del gestor



Boceto 3.17: vista principal del gestor

Vista principal del gestor una vez ha accedido a la aplicación.

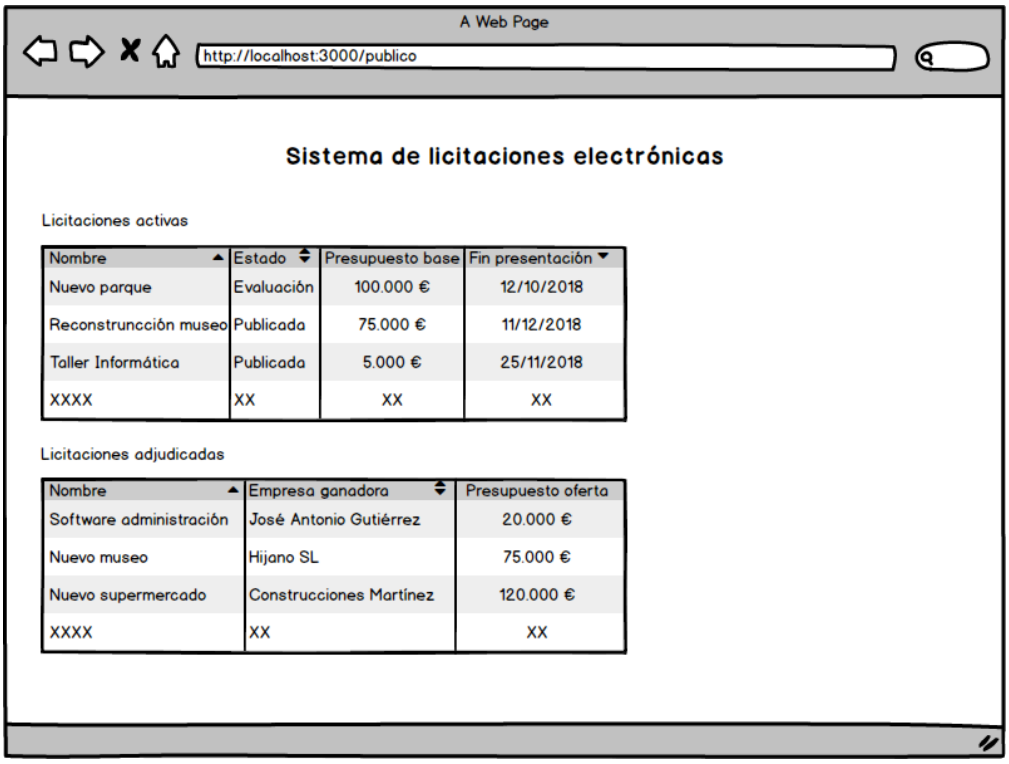
Boceto 18 – Nueva licitación



Boceto 3.18: nueva licitación

Vista de nueva licitación por parte de un gestor.

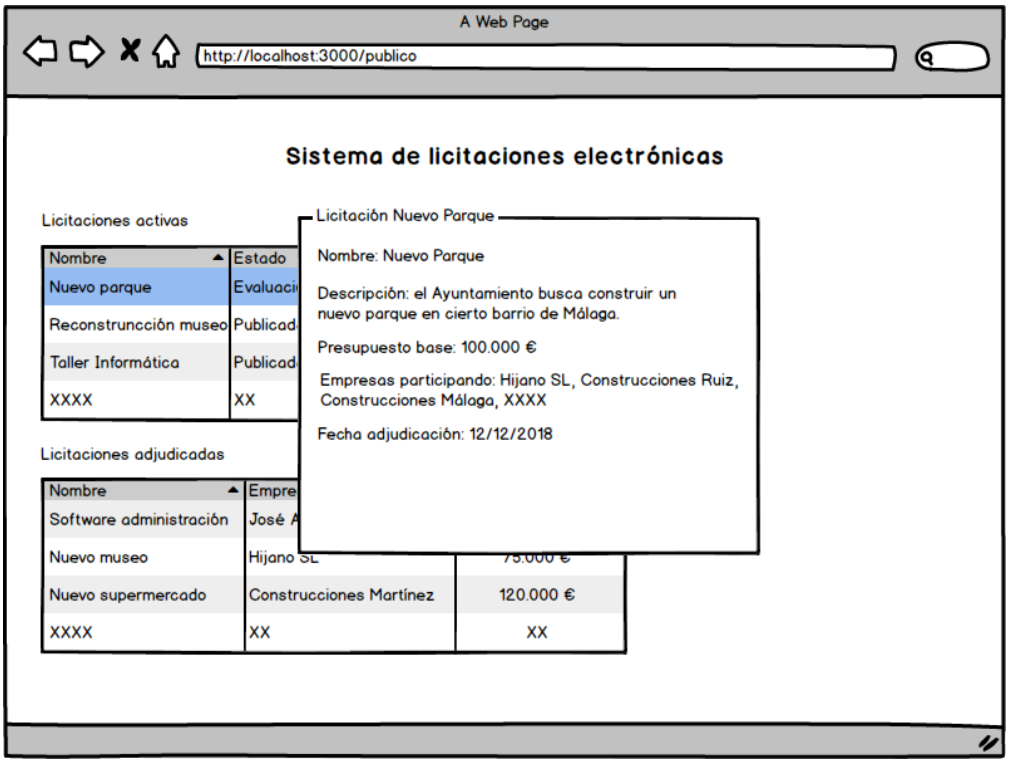
Boceto 19 – Vista principal de un usuario



Boceto 3.19: vista principal de un usuario

Vista principal de un usuario para el que no es necesario registrarse.

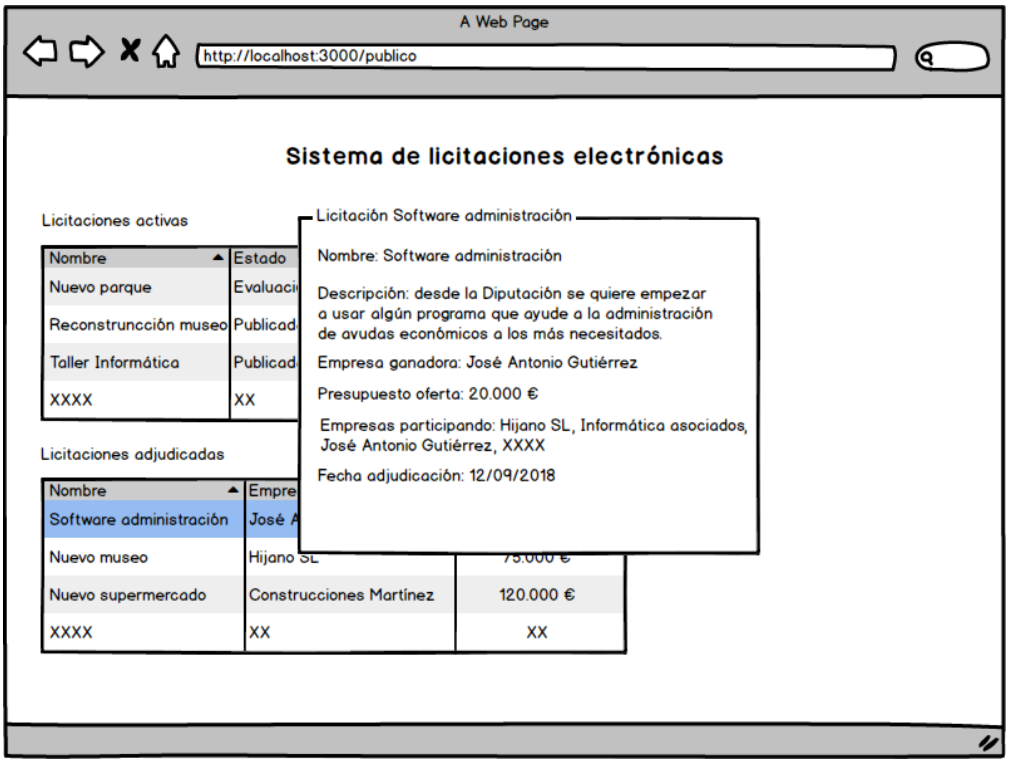
Boceto 20 – Vista usuario con licitación activa seleccionada



Boceto 3.20: vista usuario con licitación activa seleccionada

Vista de usuario con una licitación activa seleccionada.

Boceto 21 – Vista usuario con licitación adjudicada seleccionada



Boceto 3.21: vista usuario con licitación adjudicada seleccionada

Vista de usuario con una licitación adjudicada seleccionada.

Los bocetos de interfaz de usuario anteriormente mencionados son, como se verá más adelante, un reflejo muy cercano al producto final. Aun así, hay datos, como el presupuesto base que aparece en los bocetos que finalmente no aparecen en el producto final. Y otros, como los criterios de adjudicación, que sí que aparecen en el software final, pero no en los bocetos. Esto se debe a que los bocetos de interfaz de usuario han sido desarrollados en una fase muy temprana y en la que no se habían desarrollado en profundidad todas las características del proyecto.

Una vez se habían realizado estos bocetos, ya se podía pasar a la fase de diseño, la cual se centrará principalmente en explicar cómo se almacenan los datos. Cuáles se almacenan en la base de datos local, cuáles se almacenan en la red blockchain y el porqué de esa elección.

4. DISEÑO

En este apartado se tratará de especificar la arquitectura del sistema y el almacenamiento de los datos.

4.1 ARQUITECTURA DEL SISTEMA

En primer lugar, se mostrará la arquitectura del sistema.

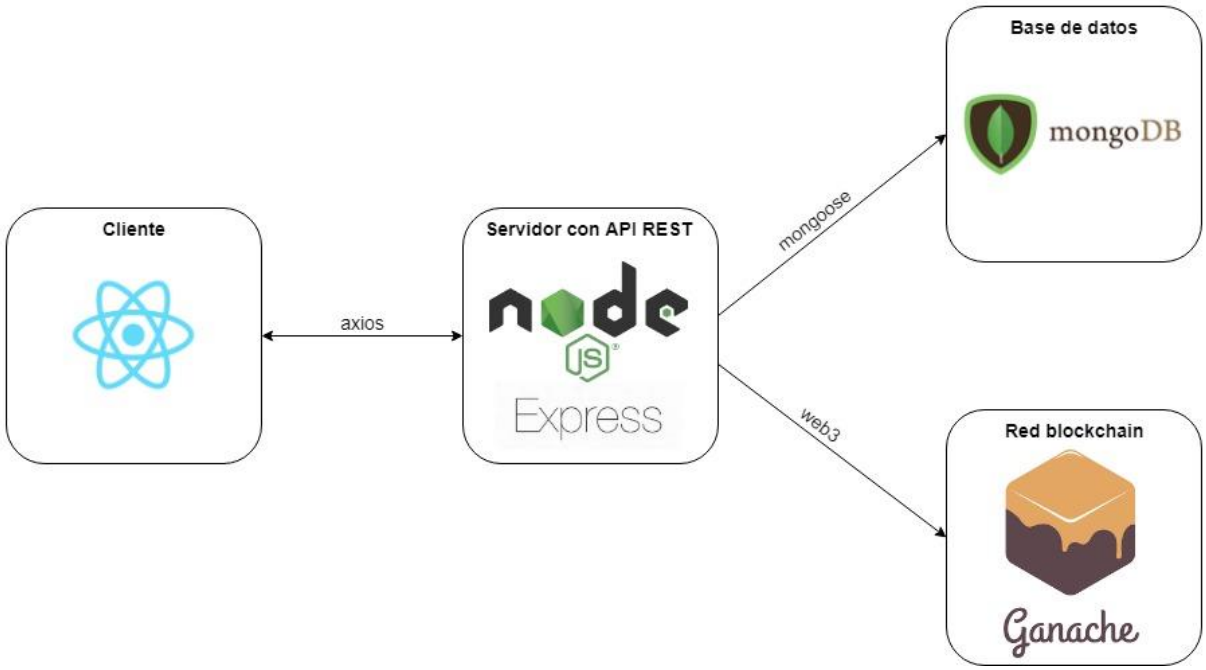


Figura 4.1: arquitectura del sistema

El proyecto tiene las diferentes capas de aplicación bien diferenciadas. El cliente se comunica con el servidor y su API REST mediante la librería axios, que realiza llamadas HTTP al servidor y devuelve promesas (Promises), las cuales nos permiten manejar la asincronía más fácilmente. Se hablará de ellas en más profundidad en el apartado de implementación.

La API REST recibe y responde las peticiones del cliente. El servidor utiliza Express para llevar a cabo estas tareas. Además, cuando la API REST así lo requiere, almacena, cambia u obtiene datos de la base de datos. Esta es una base de datos MongoDB que acepta peticiones por un puerto específico. En este caso, se utiliza la herramienta Mongoose, que se utiliza desde Node.js para comunicarnos con la base de datos.

Por otro lado, el servidor también realiza peticiones a la red blockchain. En este caso, la red privada Ganache, que se ejecuta de forma local en el mismo ordenador. Pueden ser peticiones tanto de lectura como de escritura. Cada vez que se realiza una petición de escritura, se añade un bloque a la red blockchain. Esto también se explicará en más detalle en el apartado de implementación.

4.2 ALMACENAMIENTO DE LOS DATOS

En este apartado es necesario explicar varias cosas. Almacenar información en la red blockchain requiere de gas, que en definitiva es dinero. En este caso, al ser una red privada de pruebas, es gratuito y se podría almacenar todo lo imaginable. Pero, y aunque no es uno de los objetivos de este proyecto, no se cree eficiente de cara al futuro, almacenar todos los datos en la red blockchain, incluso los que no tienen por qué guardarse en ella. El tema del gas y cómo funciona se explicará también en la parte de implementación.

Teniendo en cuenta lo anterior, los datos que no son estrictamente necesarios para el correcto funcionamiento de las licitaciones y que su modificación no acabaría alterando el resultado, se han almacenado en la base de datos. No quiere decir esto, que estén inseguros en la base de datos de MongoDB. Puesto que no sería fácil acceder a la aplicación o modificar datos sin tener los permisos. Pero, sí que se puede afirmar que los datos almacenados en la red blockchain están más seguros aún, ya que son totalmente públicos y si alguien los intentara modificar o incluso lo consiguiera, podría verse sin problemas y no sería difícil volver al estado anterior.

De acuerdo a lo comentado, el diseño de la base de datos es el siguiente:

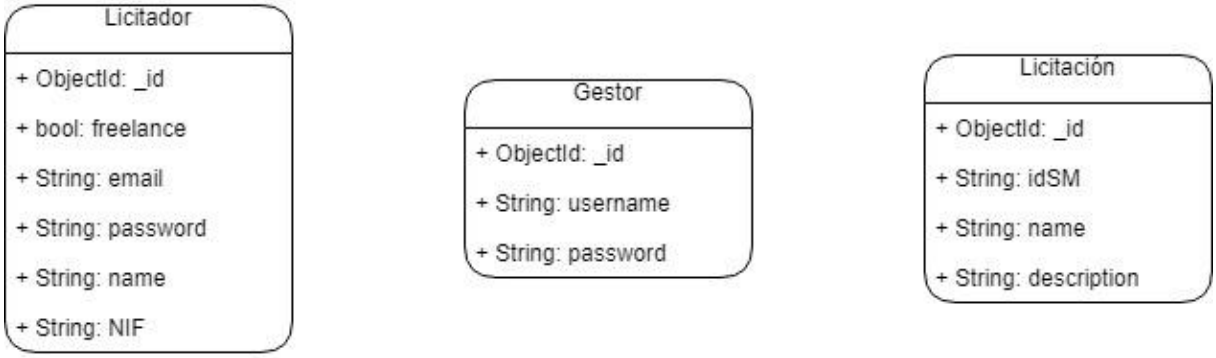


Figura 4.2: diseño de la base de datos

No es un error ni se ha pasado por alto el hecho de que no haya ninguna relación entre las entidades. La explicación es que en este proyecto sólo habrá un gestor, de forma que todas las licitaciones serán creadas por el mismo, por lo que sería un gasto innecesario crear una relación entre las licitaciones y el gestor. Si en un futuro se añadiesen más gestores y quisiese saberse qué gestor es el creador de cada licitación, podría hacerse de forma sencilla relacionando todas las licitaciones creadas hasta el momento de la inserción del nuevo gestor con el gestor inicial que había por defecto en la aplicación.

Tampoco existe una relación entre licitadores y licitaciones porque iría contra uno de los objetivos de este proyecto, que es el no saber qué licitadores participan en

cada licitación hasta que no termine la fecha de presentaciones. Esta relación podrá verse en la figura 4.3 y se explicará el porqué.

Se puede apreciar también en la figura 4.2 que en la entidad *Licitación* se almacenan los campos *idSM*, *name* y *description.* El campo *idSM* es obligatorio tenerlo aquí, ya que es la dirección del smart contract en la blockchain. Es la forma de localizarlo para enviar o recibir información de él. Con los campos *name* y *description* sucede algo interesante. Se podrían haber almacenado en la blockchain, pero se pensó que no eran campos tan relevantes como los que sí se almacenan en ella. Esto se debe a que aunque alguien pudiera cambiarlos, el resultado del proceso de licitación no cambiaría. Es decir, el ganador y los participantes serían los mismos, y si el nombre o la descripción han cambiado, tanto el gestor como cualquier participante se darían cuenta de ello.

El resto de datos que no aparecen en la figura 4.2, se almacenan en la red blockchain. Se puede ver en la figura 4.3 la declaración de estos datos en forma de variables.

```
//Model of a bidder
struct Bidder {
    uint id;
    bytes32 hash;
    uint warranty;
    uint price;
    string idBD;
    bool checkedToEvaluate;
}

//Map key-value of bidders
mapping(uint => Bidder) public bidders;

//Size of map. Zero is default value for uint.
uint public biddersCount;

//Dates
uint public endDatePresentation;
uint public evaluationDate;

//Criteria (weights and Score)
uint public warrantyWeight;
uint public priceWeight;
uint public warrantyScore;
uint public priceScore;

//Boolean to indicate that presentation date is over
bool public presentationOver;

//If it is different from 0, we have a winner
uint public winner;
```

Figura 4.3: variables del smart contract

La figura 4.3 es parte del smart contract utilizado en la aplicación. En el apartado de implementación se explicarán todas las funcionalidades, este se centrará en la utilización de las variables o datos.

La estructura *Bidder* representa a un licitador. Como podemos ver en el *map bidders*, tendremos una lista de ellos. Estos licitadores serán los que han presentado oferta a la licitación. Se almacena un *id* para localizarlos dentro del *map*, un *hash* que representa el hash de la oferta enviada, un campo *warranty* que es la garantía que ofrece en la oferta, un campo *price* que es el precio que ofrece en la oferta también, un campo *idBD* que es el id en la base de datos de MongoDB del licitador y un *checkedToEvaluate* que nos ayudará a saber si en el período de evaluación el licitador ha enviado la misma oferta.

El hecho de almacenar el campo *idBD* conlleva una contradicción. Por una parte, se ha asegurado en varias ocasiones en esta memoria que no deberían saberse los licitadores que participan en una licitación antes del período de evaluación, pero aquí se almacenan en el período de presentaciones. La explicación de esto es que si no se almacenase sería imposible saber qué licitadores presentaron en el momento del período de presentaciones. Como alternativas, se planteó en su momento almacenar un hash del id, pero esto no acabaría sirviendo de mucho. Si alguien consiguiese el id de la blockchain, necesitaría luego poder acceder a la base de datos en MongoDB para obtener los datos del licitador. Algo que no sería fácil de conseguir. Si hubiésemos almacenado el hash, igualmente, sería necesario acceder a la base de datos de MongoDB para obtener el id al que poder realizarle el hash para compararlo con el almacenado en la blockchain. Por eso no se optó por esta vía. Como conclusión a este problema, se piensa que aunque el id esté almacenado en la blockchain de forma pública, luego sería necesario acceder a la base de datos de MongoDB, por lo que no sería fácil acceder a los datos. Igualmente, podría ser buena idea plantear una solución alternativa a este problema de cara al futuro.

Otros datos que son necesarios almacenar en el smart contract son las fechas, *endDatePresentation* y *evaluationDate* que representan la fecha en la que acaba el período de presentaciones y la fecha en la que acaba el período de evaluación respectivamente. Es totalmente necesario almacenar estas fechas aquí, ya que deberían ser inmodificables, puesto que los períodos de fechas estipulados deben cumplirse.

En el apartado *Criteria* del smart contract podemos ver los diferentes criterios que se utilizarán en la fórmula para decidir un ganador. Estan las ponderaciones y las

puntuaciones. Sobre ellas se profundizará más en detalle en el apartado de la implementación

También se puede ver la variable *presentationOver*. Esta es usada para saber si ha terminado el período de presentaciones. Bien es cierto que se podría saber comparando la fecha del momento de la petición con las almacenadas, pero es mucho más cómodo tener una variable *bool* con la que saberlo al instante. En caso de que en un futuro se quisiese ahorrar más gas en el proceso de licitación, podría eliminarse esta variable y cambiarse por la comparación de las fechas directamente.

Por último aparece la variable *winner*, que nos indica si existe ya un ganador de la licitación o no. Si es 0, el valor por defecto en Solidity para los enteros, no existe ganador aún. Si es diferente, sí que existe.

Se podría debatir si almacenar algunos valores o eliminar otros. Seguramente, en un futuro, si el proyecto avanza hacia nuevas funcionalidades, haya que almacenar más datos. Queda en manos de los futuros desarrolladores y personas interesadas en el proyecto el fijarse en diferentes parámetros para ver qué variables guardar en la red blockchain.

5. IMPLEMENTACIÓN

En este apartado se explicará en detalle cómo se ha desarrollado la aplicación y por qué se han tomado ciertas decisiones, uso de ciertas tecnologías, etc. Se explicará tanto el desarrollo del cliente, el servidor con la API REST, el smart contract, la base de datos, como la comunicación entre todos ellos.

5.1 SERVIDOR NODE.JS CON API REST

El desarrollo del servidor es quizá el más importante, ya que sirve de nexo de unión entre todas las demás piezas del puzzle que forman nuestra aplicación final.

Una de las primeras cosas a comentar es que durante el desarrollo del servidor se ha utilizado la herramienta nodemon[15]. Normalmente, cada vez que se hace un cambio en el código en Node.js (Node de ahora en adelante), es necesario parar la aplicación y volverla a arrancar. En eso aporta su ayuda nodemon, que se encarga de volver a arrancar la aplicación con los cambios realizados cada vez que guardamos un archivo. Además, lo hace de forma más rápida que parando y volviendo a ejecutar el servidor.

Existe otra librería interesante que cabe mencionar en el desarrollo del servidor. Se trata de bluebird[16]. Es una librería que proporciona un uso muy cómodo y sencillo de las promesas para manejar la asincronía de Node. Si se quiere hacer uso de ella, sólo será necesario "devolver" una promesa en una función y usar los métodos *resolve* y *reject* en caso de que la operación haya ido de la forma esperada o haya habido algún fallo respectivamente. Luego, en la otra parte del código donde se llame a esta función se utilizará *then* o *catch* para obtener los datos pasados por *resolve* y *reject* respectivamente. Un ejemplo de la primera parte explicada puede verse en la figura 5.1.

```
function getChecked(idSM, idBD) {
    return new Promise(function(resolve, reject) {
        let contractInstance = contract.at(idSM);

        if(contractInstance.getChecked.call(idBD)) {
            resolve(true);
        } else {
            reject(false);
        }
    });
}
```

Figura 5.1: ejemplo de uso de bluebird

Un ejemplo de la otra parte de la explicación, la del uso de *then* y *catch* puede verse en la figura 5.22, donde se llama a la función *getChecked* de la figura 5.1.

5.1.1 PASSPORT

Passport[17] (o Passport.js) es un middleware utilizado en aplicaciones que hacen uso de Express. Proporciona un sistema de autenticación. Se puede utilizar para acceder con diversas redes sociales como Facebook o Twitter. En este caso se ha utlizado passport-local, que únicamente utiliza un sistema de usuario y clave para realizar la autenticación.

También se ha utilizado para registrar nuevos usuarios. En este caso nuevos licitadores. Passport recibe el email y clave elegido por el licitador y crea un nuevo licitador. Cabe comentar aquí que como medida de seguridad extra, no se almacenan las contraseñas tal y como vienen, si no que se realiza un hash de estas y se guarda. Este no era uno de los objetivos del proyecto, pero al ser una medida que no lleva mucho tiempo realizar, se optó por ella.

Para realizar la autenticación, Passport recibe el usuario (o email en el caso del licitador) y contraseña y comprueba que existen en la base de datos. Si existe, devuelve algunos datos del usuario, los que el desarrollador elija, y además, un token. Este token es utilizado en el cliente para saber si el usuario o gestor está registrado y para que estos no accedan a páginas a las que no tienen permiso. Este token también puede ser utilizado para que el servidor no acepte peticiones de alguna persona no registrada. Se pensó en realizar esta medida también, pero requería más tiempo que la anterior.

5.1.2 API REST CON EXPRESS

Como se ha mencionado con anterioridad, para desarrollar la API REST se ha utilizado Express. Este framework proporciona muchas utilidades y características para desarrollar una API.

La API REST se ha creado para dar soporte a todos los requisitos mencionados en el DGR. De forma que las distintas peticiones HTTP sirven para administrar los procesos de licitaciones, crear un licitador, etc.

```
router.post('/', function(req, res, next) {
    passport.authenticate('local-signup-bidder', function(err, info) {
        if(err) {
            console.log("ERROR");
            console.log(err);
            return res.status(400).json(err);
        }

        res.status(200).json({info});
    })(req, res, next);
});
```

Figura 5.2: petición POST

En la figura 5.2 podemos ver un ejemplo de una petición POST. En este caso, es la petición para añadir un nuevo licitador. Como se puede apreciar, hace uso de Passport. Si se produce un error a la hora de crear el licitador, se muestra por la consola y se envía al cliente con el código 400, que indica error. En caso de que no haya error se envía un mensaje que informa de que se ha añadido correctamente y el código 200, que indica que ha sido correcta la petición.

Las rutas de las peticiones se declaran en diferentes módulos y se llaman desde un archivo inicial que será el que ejecute nuestra aplicación y arranque el servidor. En nuestro caso, ese archivo es llamado app.js y se darán más detalles sobre él en el siguiente apartado.

```
//Routes
app.use('/login/manager', require('./api/controllers/login/loginManager'));
app.use('/login/bidder', require('./api/controllers/login/loginBidder'));
app.use('/signup/bidder', require('./api/controllers/signup/signupBidder'));
app.use('/files', require('./api/controllers/files'));
app.use('/tender', require('./api/controllers/tender/tender'));
```

Figura 5.3: añadiendo los controladores de las rutas en app.js

En la tercera línea de la figura 5.3 vemos cómo añadimos el módulo *signupBidder*, que es el que contiene la petición POST que aparece en la figura 5.2. De esta manera, si quisiese añadirse un licitador, habría que enviar la petición POST a "http://localhost:3001/signup/bidder".

Cabe mencionar que para probar esta API REST se ha utilizado la aplicación llamada Advanced REST client[18]. Es un programa que nos permite realizar peticiones HTTP al servidor incluyendo las opciones que se vean necesarias. La misma aplicación muestra la respuesta del servidor.

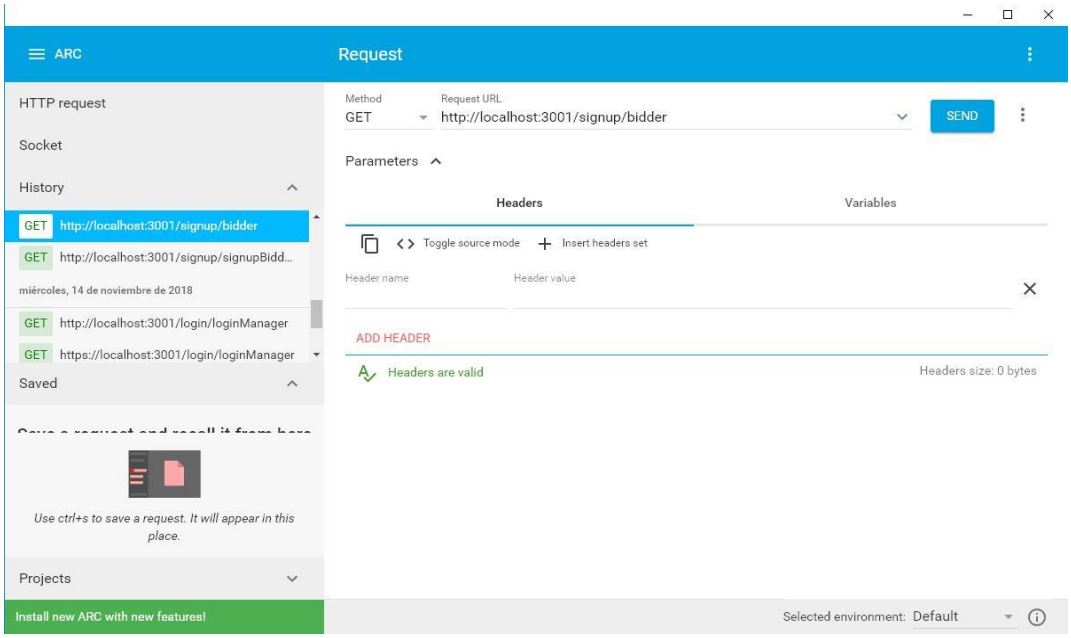


Figura 5.4: aplicación Advanced REST client

En la figura 5.4 se puede ver la aplicación Advanced REST client utilizada para probar la API REST del proyecto.

5.1.3 ARCHIVO APP.JS

El archivo app.js es el encargado de ejecutar nuestra aplicación. Se encarga también de la configuración inicial y de conectarse con la base de datos.

```
const express = require('express'),
      mongoose = require('mongoose'),
      port = process.env.PORT || 3001,
      config = require("./config/default");

let app = express(),
    passport = require('passport'),
    expressSession = require('express-session'),
    bodyParser = require('body-parser');

const optionsMongoDB = {
    useNewUrlParser: true
    //For production, autoIndex: false
}

mongoose.connect('mongodb://localhost:27017/electronicTenders', optionsMongoDB);
```

Figura 5.5: declaración de variables y conexión con MongoDB en app.js

En la figura 5.5 se puede apreciar la declaración de variables y constantes para la configuración del servidor. Se puede apreciar cómo se declaran passport, mongoose, y varias más. Además, en la última línea de la figura, vemos cómo nos conectamos a MongoDB a través de mongoose. Las declaraciones realizadas con *const* son constantes, valores que no vamos a cambiar. Mientras que las declaraciones realizadas con *let* son de variables que vamos a modificar.

```
//Accept HTTP petitions from different domain (CORS)
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
  next();
});
```

Figura 5.6: habilitación de CORS

En la figura 5.6 se puede apreciar la habilitación de CORS (Cross-Origin Resource Sharing). Esto es permitir que nuestro cliente, que se ejecuta en el puerto 3000, pueda realizar peticiones a un servicio que se encuentra en un puerto diferente, como es el caso, ya que nuestro servidor se encuentra en el puerto 3001. Esta es una medida de seguridad que se debe activar para que muchos navegadores web permitan el intercambio de información entre recursos de distinto origen.

```
app.listen(port, () => console.log(`Listening on port ${port}`));
```

Figura 5.7: ejecución del servidor

En la figura 5.7 podemos ver cómo se arranca el servidor. La constante *port*, declarada con anterioridad como se puede ver en la figura 5.5, indica el puerto en el que se va a ejecutar. Si aparece el mensaje que podemos apreciar en la figura 5.7 en consola, significa que se ha arrancado correctamente.

5.1.4 CREACIÓN DE LOS MODELOS

Como se pudo ver en la figura 4.2, en este proyecto existen 3 modelos en la base de datos. Para crear los modelos, añadir documentos, modificarlos y cualquier acción que se pretenda hacer con ellos se ha utilizado mongoose.

```
let mongoose = require('mongoose'),
    Schema = mongoose.Schema,
    bcrypt = require('bcrypt');

let bidderSchema = new Schema({
    email: {type: String, required: true, unique: true},
    password: {type: String, required: true},
    freelance: {type: Boolean, default: false},
    name: {type: String, required: true},
    NIF: {type: String, required: true}
});
```

Figura 5.8: schema de un licitador

En la figura 5.8 se puede apreciar un *schema* de un licitador. Es la forma de modelar los documentos que se van a insertar en la base de datos. Una característica muy interesante es que todos los documentos que haya en una colección, esto es como una tabla en una base de datos relacional, no tienen por qué tener el mismo *schema*.

En los *schemas* también se pueden añadir métodos para realizar diversas operaciones con los datos.

```
bidderSchema.methods.validPassword = function(password) {
    return bcrypt.compare(password, this.password)
            .then(res => {
                return res;
            })
            .catch(err => {
                console.log("Error comparando contraseña.");
                console.log(err);
                return false;
            });
}

let Bidder = mongoose.model('Bidder', bidderSchema);
```

Figura 5.9: método de un schema

En la figura 5.9 se puede apreciar el método *validPassword*, que sirve para comparar una contraseña pasada como parámetro con la almacenada por parte de un licitador. En este caso es interesante la creación de este método, puesto que para comparar una contraseña con la almacenada, es necesario realizar el hash de la primera, ya que como se ha comentado anteriormente, lo que se almacena en la base de datos es el hash de una contraseña.

En la última línea de la figura 5.9 también se puede ver la creación del modelo en sí. En estos modelos podemos realizar todas las funciones que forman un CRUD. Se pueden utilizar los método predefinidos o desarrollarlos personalizados para introducir características que se vean necesarias.

En la figura 5.10 se puede ver una función personalizada para añadir un licitador. Antes de añadirlo a la base de datos, se realiza el hash de la contraseña que se nos pasa para guardarlo así. Una vez realizado el hash, se añade de forma normal con el método que nos proporciona mongoose.

```
function addBidder(bidder) {
    //Get password hash and save
    return bcrypt.hash(bidder.password, 10)
        .then(hash => {
            let newBidder = new Bidder({
                email: bidder.email,
                password: hash,
                freelance: bidder.freelance,
                name: bidder.name,
                NIF: bidder.NIF
            });
            return newBidder.save();
        })
        .catch(err => {
            console.log(err);
            return undefined;
        });
}
```

Figura 5.10: función personalizada para añadir licitador

5.2 SMART CONTRACT

Ya pudo verse en la figura 4.3 la declaración de variables que se utilizaban en el smart contract. En este apartado tratarán de explicarse sus funciones y el resto de características.

Antes de entrar en materia, se explicará cómo se puede leer y escribir en la red blockchain, utilizando las variables vistas en la figura 4.3 y las funciones declaradas en el smart contract. En el momento en el que se declara una variable, esta se almacena

en la red. Existen distintos tipos de visibilidad, que permiten que las variables y funciones puedan ser llamadas desde otros smart contracts o no. En este caso nos centraremos en la que se ha utilizado en el smart contract del proyecto, *public*. Cabe aclarar que aunque las variables fueran de otro tipo, como *private,* y no puedan ser llamadas por otro smart contract, siguen estando en la red blockchain, de forma que no se pueden ocultar.

Con la visibilidad *public* se indica que las variables pueden ser llamadas tanto interna como externamente, por parte de otro smart contract. Además, existen distintos tipos de variables dependiendo de dónde se declaren. Algo similar a lo que ocurre con las variables globales y locales de otros lenguajes. Las variables que se pueden apreciar en la figura 4.3 son todas variables de estado y pueden ser llamadas en todo el smart contract. Es decir, como si fueran variables globales. En otras partes del código que se verán más adelante, se podrá ver que se declaran variables dentro de funciones. Estas variables sólo tienen vida cuando se llama a la función y se eliminan al terminar su ejecución. Es decir, como las variables locales de otros lenguajes.

Las variables declaradas como *public*, llevan asociado un *getter* que se crea al compilar el smart contract. Gracias a este *getter*, podrá ser leído por la librería web3 desde el servidor de forma sencilla. De esta forma puede leerse la información del smart contract almacenada en la red blockchain. En el apartado de web3 podrán verse ejemplos de cómo leer cualquiera de estas variables.

Para escribir en la red blockchain, únicamente es necesario hacer un cambio en cualquiera de las variables de estado declaradas. En el momento en el que cambian uno o más valores de las variables de estado, se almacena un nuevo bloque guardando la información actual del smart contract en ese momento. Cuando, desde el servidor, web3 llame a cualquier función que modifique o asigne un nuevo valor de una variable de estado, provocará la creación de un nuevo bloque en la red blockchain una vez termine la ejecución.

Una vez explicado cómo se puede leer y escribir en la red blockchain, se tratará de mencionar las características y funciones más relevantes del smart contract desarrollado para este proyecto.

Se lanzará un smart contract por cada licitación que se cree. Este smart contract representará un bloque en la red blockchain en el momento de su creación, y sucederá lo mismo con cada cambio que realicemos en alguna de las variables de estado, como ya se ha explicado. Con un solo cambio en cualquiera de las variables, se enviará una

transacción a la red blockchain que añadirá un nuevo bloque con la información actual del smart contract en ese momento. Esto puede apreciarse en la figura 5.11.

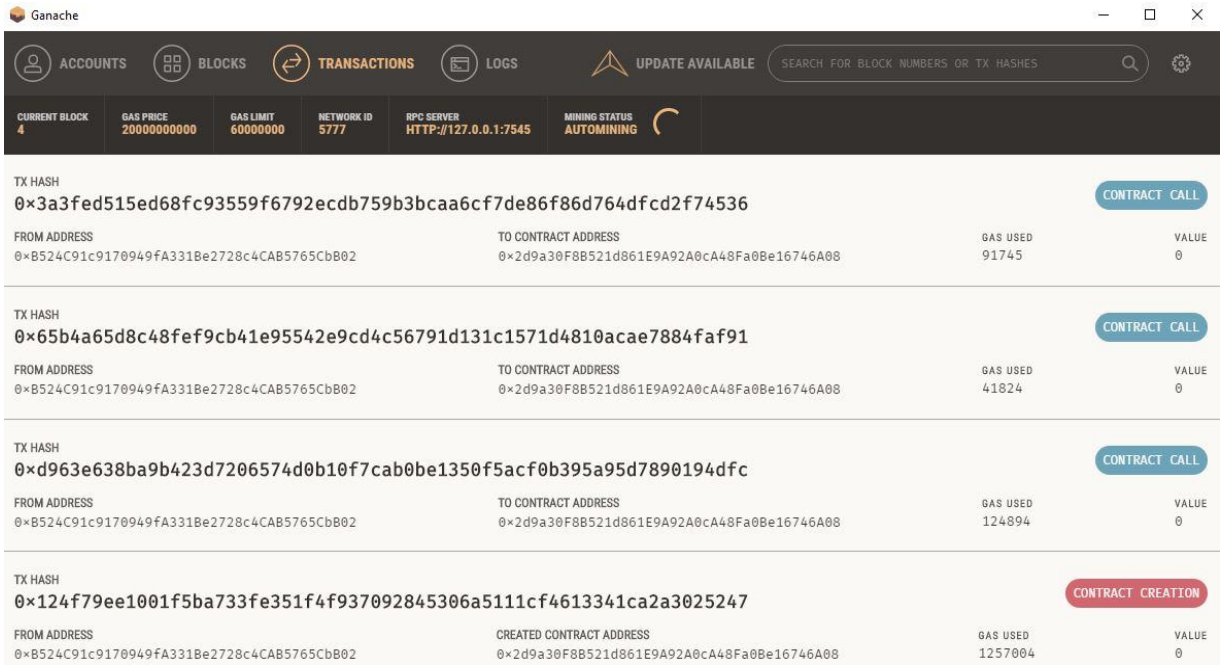


Figura 5.11: Ganache

Para realizar cualquier acción que implique cambios en la red blockchain, es necesario el uso de gas. Estos cambios se realizan en la Máquina Virtual de Ethereum (EVM, por sus siglas en inglés). Todas las licitaciones y cambios que generemos en ellas se ejecutarán en la EVM. El gas es la forma de pagar a los nodos (mineros) que se encargan de ejecutar los contratos inteligentes u otras acciones que no entran dentro del ámbito de este proyecto, como realizar transacciones de un monedero a otro. El gas, como tal, no tiene valor. Se utiliza para calcular el coste de las operaciones realizadas. Dependiendo del número de instrucciones y la complejidad de las operaciones, se usará más gas o menos. El valor lo tiene el Ether, que es la moneda, o criptomoneda, que se utiliza en Ethereum. Se calcula la equivalencia en ether del gas utilizado y eso es lo que se le paga al nodo.

En la EVM sólo tiene coste el escribir en la red, como se ha mencionado. Es decir, operaciones de lectura se pueden hacer todas las que se vean oportunas y no tendrán ningún coste. Igualmente, en este proyecto se ha utilizado una red privada de forma local, donde no existe coste para ningún tipo de operación. Si en un futuro se usa la aplicación con una red pública, debe tenerse en cuenta que la entidad que la utilice o algún organismo debe hacerse cargo de los costes.

El contrato inteligente se desarrolla con el lenguaje de programación Solidity. Este es un lenguaje creado específicamente para desarrollar smart contracts y tiene una sintaxis muy parecida a la de JavaScript. El smart contract del proyecto tiene un

constructor, como tendría cualquier clase en JavaScript. Esto puede verse en la figura 5.12.

```
//Constructor
function Tender(uint endDatePre, uint evDate, uint warrW, uint pricW, uint warrS, uint pricS) public {
    endDatePresentation = endDatePre;
    evaluationDate = evDate;
    warrantyWeight = warrW;
    priceWeight = pricW;
    warrantyScore = warrS;
    priceScore = pricS;
}
```

Figura 5.12: constructor del smart contract

En la figura 5.12 pueden verse los parámetros que se pasan al smart contract para ser almacenados como variables en la blockchain. Entre estos parámetros se encuentran las fechas, tanto de fin de presentación como de evaluación, y los criterios para seleccionar al ganador.

Una función interesante es la de añadir un participante o licitador a la licitación. Esta se encarga de añadir un licitador a la lista en caso de que la fecha actual esté dentro del rango correcto. Realiza el hash de los datos de la oferta pasada y lo almacena en la lista junto al id de la base de datos del licitador. Esto puede verse en la figura 5.13.

```
function addBidder (string description, uint warranty, uint price, string idBD, uint currentDate) public {
    //Check dates
    if(!presentationOver && currentDate < evaluationDate) {
        biddersCount++;

        //hash
        bytes32 hash = keccak256(
            abi.encodePacked(description, warranty, price)
        );

        bidders[biddersCount] = Bidder(biddersCount, hash, 0, 0, idBD, false);
    }
}
```

Figura 5.13: función para añadir licitador

Sobre esta figura, la 5.13, hay varias cosas que explicar. Los datos de una oferta constan de descripción, precio y garantía. Como se puede ver, se hace el hash de todos ellos, pero no se almacena ninguno en la blockchain, ya que no se asignan a ninguna variable de estado. El algoritmo utilizado para realizar el hash es *keccak256*, que es el nombre que se le ha puesto en Solidity a *SHA-3*. Existen otros algoritmos para obtener el hash de algunos datos, pero *keccak256* es el que menos gas utiliza.

Los datos pasados a *Bidder*, que es el *struct* que los contiene, incluye dos ceros que representan a la garantía y al precio. La explicación para esto es que el proceso de licitación se encuentra en este momento en el período de presentaciones, por lo que no se quiere que haya información sobre las ofertas almacenada en la red blockchain aún. Estos datos, la garantía y el precio, se almacenarán posteriormente, cuando el período de presentaciones haya acabado y comience el de evaluación.

Además, en la figura 5.13 puede verse que la fecha actual, *currentDate*, es pasada como parámetro. Esto quiere decir que en el caso del proyecto, es pasada desde el servidor Node. Para hacer el sistema más fiable aún, hubiese sido deseable que la fecha pudiese obtenerse directamente de la red blockchain. Pero esto último no es posible debido a que la red blockchain no tiene ninguna función ni característica para tener la hora actual, debido a que al ser una red formada por nodos que deben ponerse de acuerdo, tendrían que tener todos la misma hora, cosa que es imposible. Existen alternativas como APIs externas y demás que proporcionan la fecha actual, pero el tiempo que habría que dedicarle se saldría dentro del estipulado para este proyecto. Una opción de mejora para el futuro podría ser el utilizar alguna de estas APIs y que así sea más fiable el uso del tiempo en el proyecto.

Para solventar esto, como se ha comentado, se envía la fecha desde el servidor y se compara con las almacenadas en el smart contract para proceder a realizar una operación o no. Para todo el tema de las fechas y sus distintos usos en el servidor, se ha utilizado Moment[19] (o Moment.js), una librería que proporciona muchas utilidades y facilita mucho el uso de las fechas.

Para finalizar con el tema de las fechas, también se puede apreciar en la figura 5.13 y en la figura 4.3, que no utilizan ningún tipo *Date* ni nada que se le parezca, sino que utilizan *uint*, es decir enteros positivos o sin signo (unsigned integers). Esto se debe a que Solidity no proporciona ningún tipo de dato para las fechas. Por lo que la solución general suele ser utilizar *uint* y almacenar las fechas en su formato UNIX. Esto puede parecer que es un problema y complica el desarrollo, pero la verdad es que este formato es muy cómodo para realizar las comparaciones y manejar las fechas, ya que se pueden comparar como si de números se tratase. Para comparar fechas con formatos como *Date* en JavaScript u otros similares, se necesitan funciones especiales, por lo que acaba siendo algo más complicado. En el momento en que se quiere mostrar estas fechas y enviarlas al cliente sólo hay que pedírselas al smart contract y transformarlas con Moment, algo que se hace de una forma bastante sencilla.

Otra función interesante del smart contract del proyecto es la encargada de comprobar las ofertas de nuevo en el período de evaluación. Una vez ha terminado el período de presentaciones, comienza el período de evaluación. En este período el licitador debe enviar por segunda vez su oferta y ahí es donde entra en juego la función comentada. Esta función determina qué participantes acaban siendo evaluados y cuáles no, dependiendo de si han enviado la misma oferta que enviaron en el período de presentaciones o si han modificado alguna parte de ella.

```
function checkBidder(string description, uint warranty, uint price, string idBD, uint currentDate) public {
    //Check if bidder is here
    if(isBidderHere(idBD)) {
        //Check dates
        if(presentationOver && currentDate < evaluationDate) {
            bytes32 hashAux = keccak256(
                abi.encodePacked(description, warranty, price)
            );

            //Search bidder
            uint index = 1;
            while(index <= biddersCount) {
                if(keccak256(bidders[index].idBD) == keccak256(idBD)) {
                    //Found!
                    if(hashAux == bidders[index].hash) {
                        //Same hashes!
                        bidders[index].checkedToEvaluate = true;
                        bidders[index].warranty = warranty;
                        bidders[index].price = price;
                    }
                }
                index++;
            }
        }
    }
}
```

Figura 5.14: función encargada de comprobar la oferta en el período de evaluación

Como se puede ver en la figura 5.14, la función empieza llamando a otra función del smart contract para comprobar que el licitador se encuentra entre los ya almacenados en la lista de licitadores. Una vez comprueba que se encuentra, se centra en las fechas, en comprobar que esté dentro del período de evaluación. Seguidamente, pasa a localizar al licitador para comprobar el hash de la oferta enviada con el que está ya almacenado.

Si se presta atención, se puede observar que dentro del bucle *while* no se comparan los *strings* que se refieren a los ids tal cual, si no que realiza el hash de cada uno y se comparan los hashes. Esto se debe a que Solidity no da la posibilidad de comparar dos *strings*, ni siquiera a través de una función especial. Por lo que la opción más utilizada es realizar el hash de un *string*, que devuelve un tipo de dato *bytes32,* y compararlo con el hash del otro *string*. Para los tipos de dato *bytes32* sí que permite la comparación.

Una vez que en la figura 5.14 se ha localizado al licitador, se realiza el hash de la oferta enviada. Si ese hash concuerda con el almacenado en la blockchain, significa que el documento es el mismo que se envió en el período de presentaciones. En caso contrario, significa que se ha modificado alguna parte de él. En caso de que sea el mismo, se almacena la información del precio y de la garantía y se cambia el valor de un campo a verdadero para indicar que se ha enviado la misma oferta. En caso de que no sea el mismo archivo, no se modifica nada y el servidor se encarga de enviar un mensaje de error.

Tras realizar el proceso de comprobar las ofertas en el período de evaluación, se procedería, llegada la fecha de evaluación, a elegir un ganador. Para ello, se utiliza una función que asigna una puntuación en base a la garantía y el precio ofrecido por el licitador. Esta función puede apreciarse en la figura 5.15.

```
function calculateScore(uint id) public returns (uint){
    uint bidderPrice = bidders[id].price;
    uint bidderWarranty = bidders[id].warranty;

    //We multiply by 10000 so that the values don`t depend on the decimals
    //Directly proportional
    uint warrantyPoints = (((bidderWarranty * 10000) * warrantyWeight) * warrantyScore) / 100;

    //Inversely proportional
    uint pricePoints = (((priceWeight * 10000) * priceScore) * 100) / bidderPrice;

    return (warrantyPoints + pricePoints);
}
```

Figura 5.15: función para calcular puntuación de oferta

La figura 5.15 muestra la fórmula utilizada para obtener la puntuación de una oferta. Esta fórmula es parecida a las que utilizan en Aragón[20], salvo que allí establecen límites inferiores y superiores para los valores, precio y garantía en este caso. En este proyecto no se especificó nada de esos límites, por lo que no se tuvo en cuenta. La fórmula tiene dos partes, una para la garantía y otra para el precio. En el caso de la garantía, la puntuación es directamente proporcional. Es decir, cuanto mayor sea la garantía, mayor puntuación se obtendrá. En el caso del precio, es al revés, inversamente proporcional. A menor precio, mayor puntuación.

Se puede apreciar en el caso de ambas puntuaciones, garantía y precio, que se multiplica por 10.000. Esto se debe a que en Solidity no existen aún los decimales ni tipos de datos como *float* o *double*. Existen librerías que tratan de dar solución a esto, pero como en casos anteriores, o eran de pago o requerían bastante tiempo para entenderlas. Para solventar este problema, se ha multiplicado por 10.000 y se ha dejado la única división que se realiza en cada caso como operación final. Así los decimales no afectan tanto al resultado final. En una prueba realizada, dos ofertas se diferenciaban por tan solo un euro en el precio y se eligió a la correcta. Se ha utilizado 10.000 como número por ser un número relativamente grande, pero en caso de que 10.000 se quede corto y pueda ocasionar que en algunos casos se vea afectado el resultado por los decimales, se puede utilizar un número mayor. El tipo de dato *uint* tiene un valor máximo de 2^256 -1, que es muchísimo mayor que los valores con los que se han operado en las pruebas, por lo que no se cree que haya problema en aumentar el valor de 10.000.

Por último, cabe comentar sobre esta función, de que si en un futuro se cambia, que será lo más probable, no existiría ningún problema con el resto del smart contract. Esta función es llamada por otra, que es la encargada de recorrer la lista de ofertas,

calcular la puntuación, llamando a la función de la figura 5.15, y obtener el mayor valor para declarar un ganador. Por lo que, en caso de que quiera cambiarse, sólo habría que cambiar la que aparece en la figura 5.15, haciendo los cálculos que se vean convenientes y no sería necesario cambiar nada más en el smart contract.

5.3 WEB3

En este apartado tratará de explicarse la conexión entre el apartado 5.2, el smart contract, y el 5.1, el servidor en Node. Aunque bien es verdad, que la utilización de web3 se encuentra dentro del servidor Node como tal, se considera necesario dedicarle un apartado diferente por su importancia.

Como se comentó con anterioridad, web3 es una librería, o conjunto de librerías, que se utiliza para comunicarse con la red blockchain y sus smart contracts desplegados. Tiene una relevancia enorme, ya que todas las funciones vistas en el apartado 5.2, y las que no aparecen ahí, son ejecutadas en algún momento mediante la utilización de web3.

Existe un fichero en el servidor, llamado *controlTender.js*, que es el encargado, como su nombre indica, de controlar los procesos de licitación. Es este fichero el que usa web3 para realizar cualquier operación en la red blockchain e informar al servidor de ello.

```
// Connect to local Ethereum Blockchain
let web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:7545"));
```

Figura 5.16: conexión a red blockchain

En la figura 5.16 puede verse el primer paso para empezar a utilizar web3: seleccionar una red blockchain a la que conectarse. En este caso la red blockchain apunta a *localhost*, ya que como se ha explicado, se utiliza Ganache, una red blockchain privada para realizar pruebas. Para utilizar una red pública en lugar de Ganache, sólo habría que cambiar la url que aparece en la figura 5.16. También sería necesario para poder realizar cualquier operación en una red blockchain pública tener una cuenta con una dirección válida en esta red pública.

Con la red blockchain elegida, se necesita compilar el smart contract. Este paso es necesario para cualquier operación que queramos realizar con el smart contract. Este proceso puede verse en la figura 5.17.

```
// Compile the source code
const tenderPath = path.resolve(__dirname, '../contracts', 'Tender.sol');

const source = fs.readFileSync(tenderPath, 'UTF-8');

const output = solc.compile(source, 1);

const bytecode = output.contracts[':Tender'].bytecode;

const abi = JSON.parse(output.contracts[':Tender'].interface);

//Contract object
const contract = web3.eth.contract(abi);
```

Figura 5.17: compilación y creación del smart contract

En la figura 5.17 puede verse el proceso seguido para obtener el smart contract como objeto. Primeramente, es necesario leer el archivo fuente donde se encuentra. Una vez leído, se compila con *solc*[21], un compilador de Solidity. Tras esto, sólo queda obtener el *bytecode* obtenido por el compilador y el Application Binary Interface (ABI). Con todos los pasos anteriores ya podemos crear el objeto del smart contract a partir del ABI. Con este objeto es posible o bien desplegar un nuevo smart contract con los parámetros que se vean convenientes, o localizar uno ya desplegado en la red blockchain.

Este objeto está declarado de forma global para que pueda ser utilizado por cualquier función y sólo sea necesario compilarlo una vez al ejecutar el servidor. Se realiza de esta forma ya que el proceso que aparece en la figura 5.17 es un proceso lento, por lo que no interesa realizarlo cada vez que se quiera desplegar o interactuar con un smart contract. De esta forma, se tiene almacenado en memoria y sólo hay que llamar al objeto para poder operar con él.

En la figura 5.18 se puede ver la función encargada de desplegar un smart contract en la red blockchain. En este proyecto, el smart contract representa siempre una licitación. A esta función se le pasan como parámetros la fecha de fin de presentación, la fecha de evaluación y los criterios para decidir a un ganador. Como se puede ver en la figura 5.12, estos son los mismos parámetros que se le pasan al constructor del smart contract. En caso de que se añadiese cualquier nuevo parámetro al constructor del smart contract, también sería necesario añadirlo en esta función. En caso de que no concuerden, se lanza una excepción.

```
//Returns contract's address
function newTender(endDatePresentation, evaluationDate, warrantyW, priceW, warrantyS, priceS) {
    return new Promise(function(resolve, reject) {
        const gasEstimate = web3.eth.estimateGas({data: contract.new.getData(
            endDatePresentation, evaluationDate, warrantyW, priceW, warrantyS, priceS, {data: bytecode}
        )});

        const contractInstance = contract.new(
            endDatePresentation, evaluationDate,
            warrantyW, priceW,
            warrantyS, priceS,
        {
            data: '0x' + bytecode,
            from: web3.eth.coinbase,
            gas: gasEstimate
        }, (err, res) => {
            if (err) {
                console.log("ERROR DESPLEGANDO SMART CONTRACT");
                console.log(err);
                reject(err);
            }

            if(res.address) {
                resolve(res.address);
                }
            });
        });
}
```

Figura 5.18: función que genera una nueva licitación

También se puede apreciar en la figura 5.18 la constante llamada *gasEstimate*, que indica una estimación del gas que costará realizar la operación. Cuando se va a lanzar un smart contract o a hacer una modificación en él, es necesario pasar como parámetro el gas para realizar la operación. Para realizar una estimación de ese gas se utiliza la función *estimateGas* de web3. Esta función ejecuta la operación en la máquina virtual pero sin llegar a minar el bloque y devuelve el gas utilizado para ello.

Uno de los parámetros a la hora de lanzar el smart contract a la red blockchain es *from*, que debe ser una dirección de una cuenta de la red blockchain. Será la encargada de "pagar" la operación. En este caso, es *web3.eth.coinbase*, que representa la cuenta por defecto que elige web3 y es la cuenta 0 de las que proporciona Ganache para utilizar con las operaciones. Si en un futuro este proyecto se lleva a un ámbito de utilización en una red blockchain pública, será necesario que apunte a una cuenta real que forme parte de la red. Seguramente, una cuenta gestionada por la entidad pública encargada de las licitaciones.

Una función muy interesante por su importancia es *getTender*. Esta función recibe como parámetro la dirección de una licitación o smart contract y devuelve los datos almacenados en ella. También se encarga de diferentes comprobaciones, como ver si ya existe un ganador, si ha acabado el período de presentaciones, etc., y pide al smart contract realizar distintas operaciones dependiendo del resultado de las comprobaciones. En la figura 5.19 puede apreciarse lo primero que es necesario para realizar esto, que es localizar al smart contract.

```
let contractInstance = contract.at(id);
```

Figura 5.19: variable que contiene la instancia del smart contract

Puede verse en la figura 5.19 que para localizar al smart contract y tener una instancia de este es necesario utilizar la variable global que representaba el objeto del smart contract que se comentó sobre la figura 5.17. Cualquier operación que se quiera hacer sobre el smart contract ya desplegado tiene que ser con la variable *contractInstance*. Esta no podemos tenerla declarada de forma global, ya que habrá diferentes licitaciones desplegadas y puede pedirse información o hacer una modificación sobre varias diferentes, no siempre sobre la misma. De manera que será necesario realizar la operación de la figura 5.19 cada vez que se quiera pedir o cambiar información del smart contract.

La función *getTender* devuelve los datos almacenados en el smart contract. Dependiendo de la etapa del proceso en el que se encuentre, devuelve más datos o menos. Si el proceso ya ha acabado y existe un ganador, lo devuelve también entre esos datos. Los datos que siempre devuelve son los que se pasaron como parámetros al constructor. Es decir, las fechas y los criterios de evaluación. Todo lo que se aprecia en la figura 5.20 son llamadas a los *getters* ya mencionados anteriormente. Por ejemplo, para obtener la fecha de evaluación sería *contractInstance.evaluationDate()*.

```
let tender = {
    endDatePresentation: moment(contractInstance.endDatePresentation().toNumber(), "x").
    evaluationDate: moment(contractInstance.evaluationDate().toNumber(), "x").format("DD
    warrantyW: contractInstance.warrantyWeight().toNumber(),
    priceW: contractInstance.priceWeight().toNumber(),
    warrantyS: contractInstance.warrantyScore().toNumber(),
    priceS: contractInstance.priceScore().toNumber(),
    presentationOver: contractInstance.presentationOver()
}
```

Figura 5.20: información básica que devuelve el smart contract

En la figura 5.21 puede verse la comprobación de ganador en la licitación.

```
//Check if we have a winner now
if(contractInstance.winner().toNumber() !== 0) {
    //We have a winner
    let winner = contractInstance.getWinner.call();
    tender.presentationOver = true;

    //Search winner's name
    Bidder.findBidder({_id: winner})
        .then(res => {
            tender.winner = res[0].name;
            resolve(tender);
        });
} else {
```

Figura 5.21: comprobación de ganador de la licitación

Como pudo verse en la figura 4.3, la variable *winner* es *uint* (unsigned intenger). Esta variable se devuelve a JavaScript como un *BigNumber*, de forma que para utilizarla de forma más fácil, como un entero, se utiliza la función *toNumber*. Cuando *winner* tenía el valor 0, significaba que aún no existía un ganador. Por lo que si el valor es diferente de 0, ya hay ganador elegido para la licitación. En caso de que sea así, se busca el ganador a través de la función *getWinner*, que básicamente devuelve el id del licitador en la base de datos. Una vez se tiene este id, se realiza una llamada a la base de datos para obtener el nombre y devolverlo al cliente o quien lo pida.

Un ejemplo de cómo modificar algún parámetro en el smart contract puede ser el momento en el que el licitador envía la oferta por segunda vez para que se compruebe en el período de evaluación. Este caso puede verse en la figura 5.22.

```
function checkBidder(id, description, warranty, price, idBD) {
    return new Promise(function(resolve, reject) {
        let contractInstance = contract.at(id);

        let estimateGas = contractInstance.checkBidder.estimateGas(description, warranty, price

        contractInstance.checkBidder(description, warranty, price, idBD, moment().format("x"),
            if(err) {
                console.log("ERROR CHECK BIDDER");
                console.log(err);
                resolve(err);
            } else {
                getChecked(id, idBD)
                    .then(res => {
                        resolve(res);
                    })
                    .catch(err => {
                        reject(err);
                    });
            }
        })
    })
}
```

Figura 5.22: función que comprueba la oferta en el período de evaluación

La figura 5.22 presenta un caso curioso, puesto que es una función que puede modificar o no la red blockchain. Esta función llama a la función con su mismo nombre del smart contract. Esta función del smart contract pudo verse en la figura 5.14. Se explicó que si los hashes de la oferta coincidían, se cambiarían algunos datos en el smart contract. En caso de que los hashes no coincidieran, no se cambiaba nada y el servidor informaba de ello. Debido a esto se utiliza la función *getChecked* que puede verse en la figura 5.22. Esta función *getChecked* llama al smart contract y comprueba el campo *checkedToEvaluate* al que se le asignaba el valor *true* en la figura 5.14. Si ese campo es *true*, devuelve un resultado positivo y si es *false* devuelve un resultado negativo.

5.4 CLIENTE

En este apartado se mostrarán imágenes en las que se podrá ver el aspecto del cliente en el proyecto final. También se explicarán algunos detalles de la implementación del frontend.

Figura 5.23: página principal de la aplicación

Vista de la página principal de la aplicación. En la figura 5.23 se puede apreciar que existen 3 tipos de acceso para los distintos tipos de usuario de la aplicación. Gestor, licitadores y ciudadanos.

Figura 5.24: acceso del gestor

Vista de la página para el acceso del gestor.

Figura 5.25: acceso de un licitador

Vista de la página para el acceso de un licitador. Se puede apreciar que es similar que la figura 5.24, salvo que en este caso pide un correo electrónico en lugar de un nombre de usuario.

Figura 5.26: vista del registro sin tipo de usuario seleccionado

Vista de la página de registro de licitador cuando no se ha seleccionado ningún tipo de usuario.

Figura 5.27: vista del registro con Empresa seleccionado

Vista de la página de registro de licitador con el tipo de usuario Empresa seleccionado. La vista con el tipo de usuario Autónomo seleccionado es exactamente la misma, pero en lugar de *Nombre de la empresa* aparece *Nombre y apellidos*. En caso de que falte algún campo por rellenar o las contraseñas no coincidan, aparece un mensaje de error.

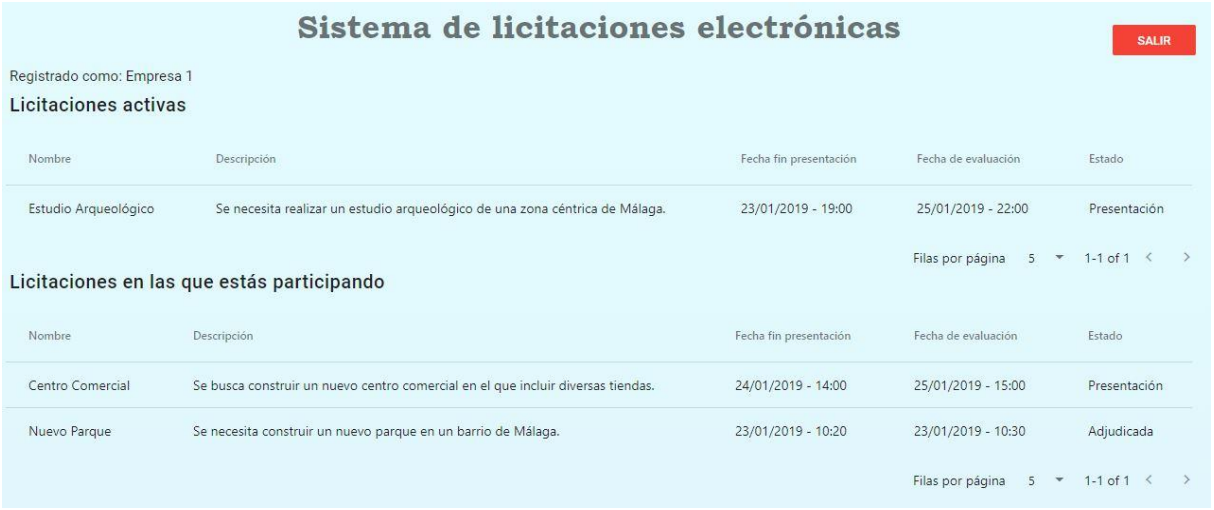


Figura 5.28: vista principal de licitador

Vista principal de un licitador una vez ha accedido a la aplicación.

Figura 5.29: vista principal del gestor

Vista principal del gestor una vez ha accedido a la aplicación.

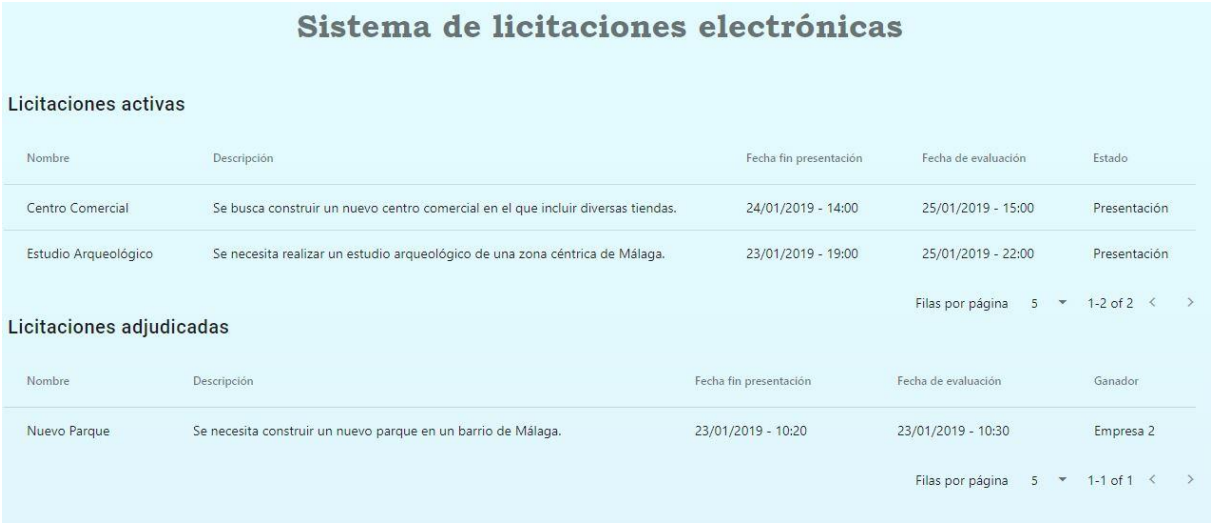


Figura 5.30: vista principal de un ciudadano

Vista principal de un ciudadano que no necesita realizar ningún proceso de acceso. Se puede apreciar que las figuras 5.28, 5.29 y 5.30 son muy similares en cuanto al contenido y sólo cambian las funcionalidades que pueden realizar los usuarios.

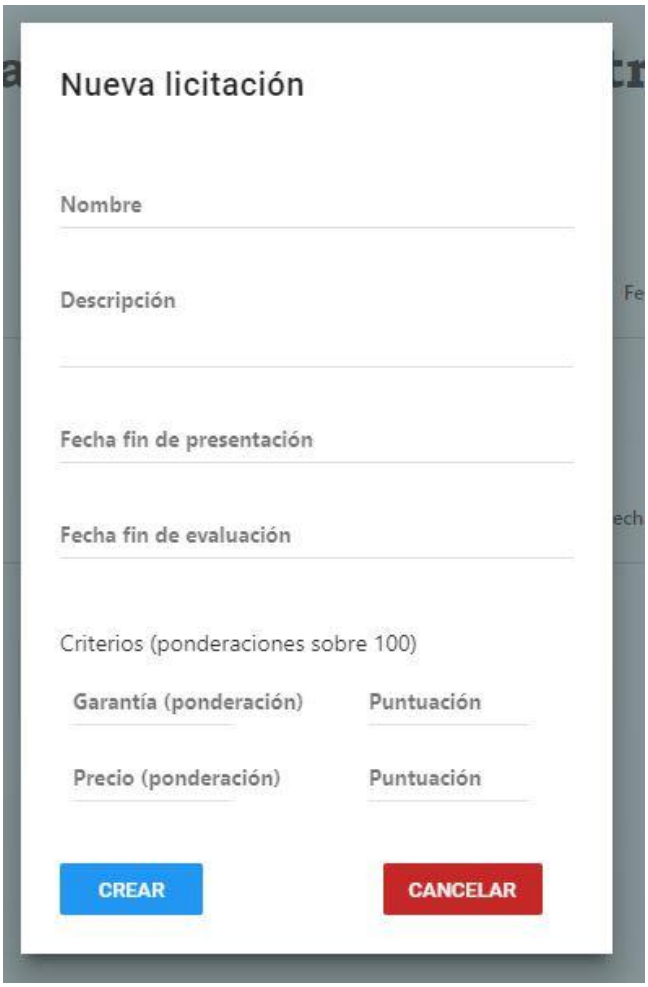


Figura 5.31: vista del popup para crear licitación

Vista del popup para crear una nueva licitación. Esta acción sólo la puede realizar el gestor. En caso de que haya algún error en los datos salta un mensaje.

Figura 5.32: vista de una licitación en período de presentaciones

Vista de una licitación en período de presentaciones. Se accede a esta vista seleccionando alguna de las licitaciones que se encuentran en la tabla de *Licitaciones acivas.* Esta vista sólo puede verla un licitador y únicamente en el período de presentaciones de una licitación en la que aún no ha participado. Al seleccionar *Descargar fichero oferta* se descarga un archivo XML genérico para introducir los datos de su oferta y poder enviarlo al seleccionar *Presentar oferta*.

Oferta.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Tender>
3      <Description>Inserte aquí la descripción de su oferta.</Description>
4      <Warranty>Inserte aquí el tiempo de garantía, indicado en meses.</Warranty>
5      <Price>Inserte aquí el precio de su oferta.</Price>
6  </Tender>
7
```

Figura 5.33: fichero Oferta.xml

La figura 5.33 es el fichero XML que se descarga al seleccionar *Descargar fichero oferta* en la figura 5.32. El licitador que quiera participar deberá rellenarlo con los datos que crea convenientes y enviarlo.

Figura 5.34: vista del popup para enviar oferta

En la figura 5.34 se puede apreciar la vista del popup que aparece al seleccionar la opción *Presentar oferta* en la figura 5.32.

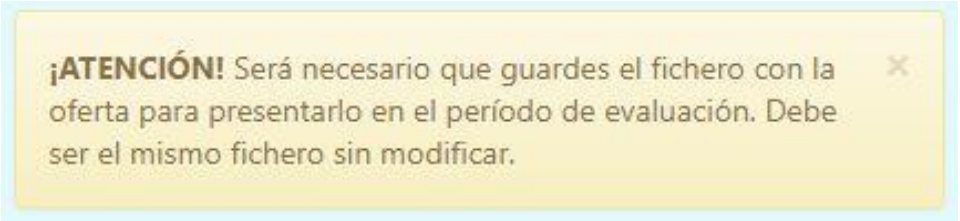


Figura 5.35: notificación que aparece al enviar oferta en período de presentaciones

Notificación que aparece cuando se envía la oferta por parte del licitador en el período de presentaciones. Se le indica al licitador que es importante que almacene el fichero de la oferta que ha enviado, ya que se le pedirá de nuevo en el período de evaluación.

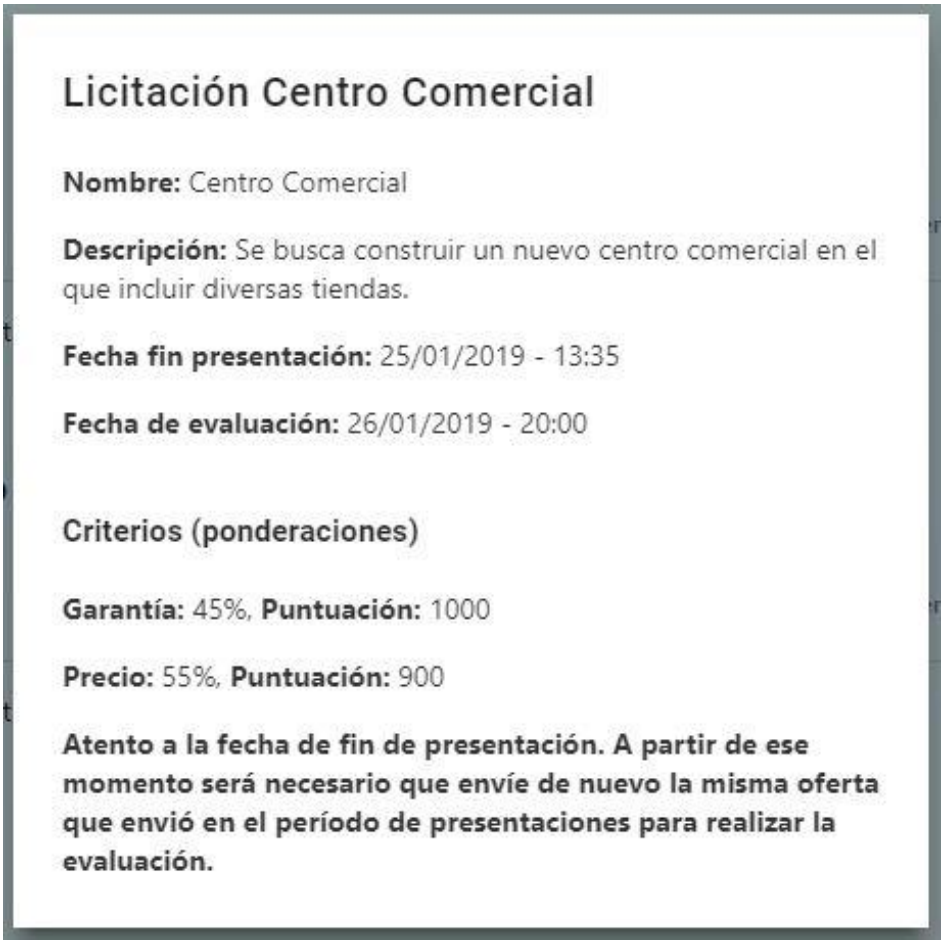


Figura 5.36: vista de una licitación en la que se participa en período de presentaciones

Vista de una licitación en la que está participando el licitador en período de presentaciones. Una vez ha presentado la oferta no puede realizar ninguna otra acción hasta que finalice el período de presentaciones y empiece el de evaluación. Se puede apreciar que también se avisa aquí al licitador de que será necesario que envíe la misma oferta en el período de evaluación que la que envío en el período de presentaciones.

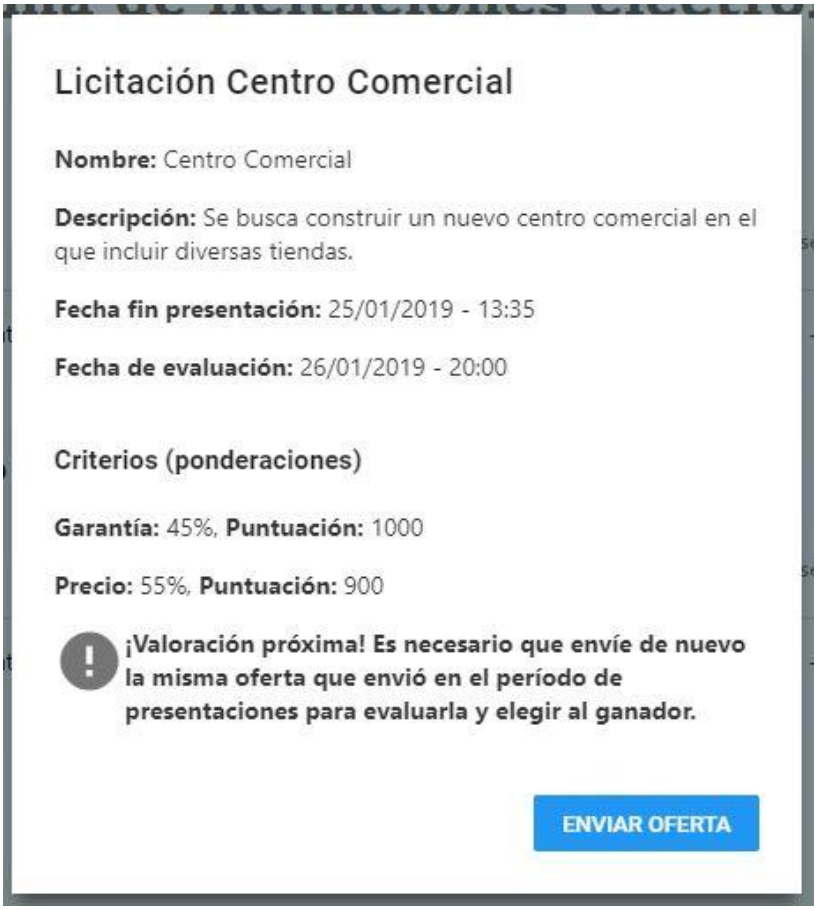


Figura 5.37: vista de una licitación en la que se participa en período de evaluación

Vista de una licitación en la que está participando el licitador en período de evaluación. Si no envía la misma oferta que envió en el período de presentaciones de nuevo en este período su oferta no será evaluada y por tanto no podrá resultar ganador. Al seleccionar *Enviar oferta* se muestra el mismo popup que en la figura 5.34.

Figura 5.38: vista del popup para enviar oferta cuando no se ha enviado la oferta original

Vista del popup para enviar oferta cuando se ha enviado una oferta que no coincide con la que se envió originalmente. Si no envía la oferta original no podrá seguir participando en la licitación.

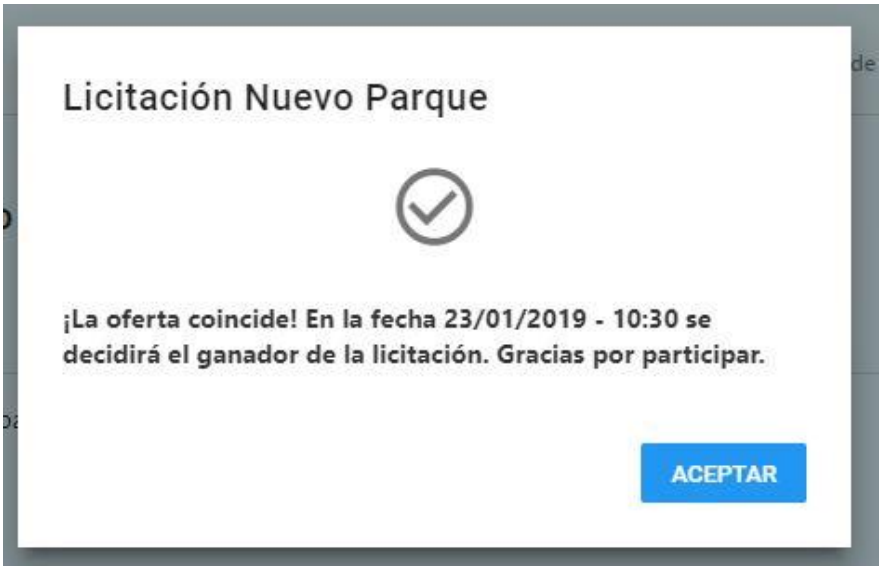


Figura 5.39: vista del popup de enviar oferta cuando se envía la misma oferta que la original

Vista del popup de enviar oferta cuando se ha enviado la misma oferta que la que se envió originalmente. En este momento el licitador ya no tiene que realizar más acciones en esta licitación y sólo debe esperar a la fecha de evaluación para saber si ha resultado ganador o no.

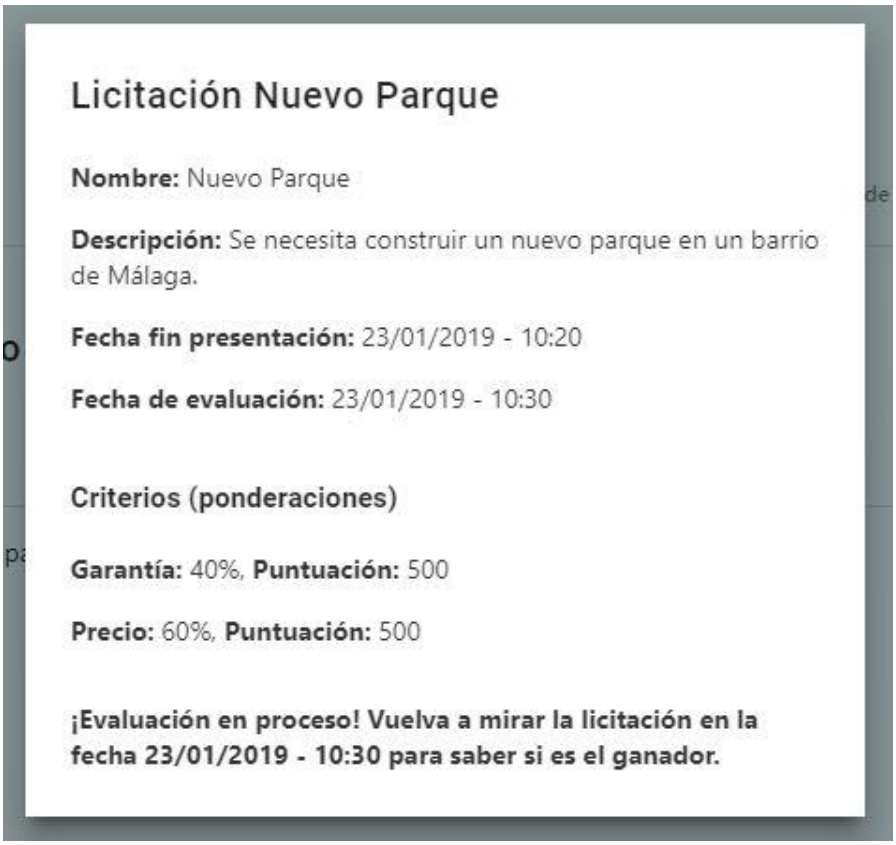


Figura 5.40: vista de una licitación cuando se ha enviado por segunda vez la misma oferta

Vista de una licitación en la que está participando un licitador cuando ya ha enviado la oferta de nuevo en el período de evaluación. En caso de que no hubiese enviado la misma oferta seguiría apareciendo lo mismo que en la figura 5.37.

Figura 5.41: vista de una licitación en la que ha ganado el licitador registrado

Vista de una licitación por parte de un licitador en la que ya se ha decidido el ganador y es el licitador registrado en la aplicación en ese momento.

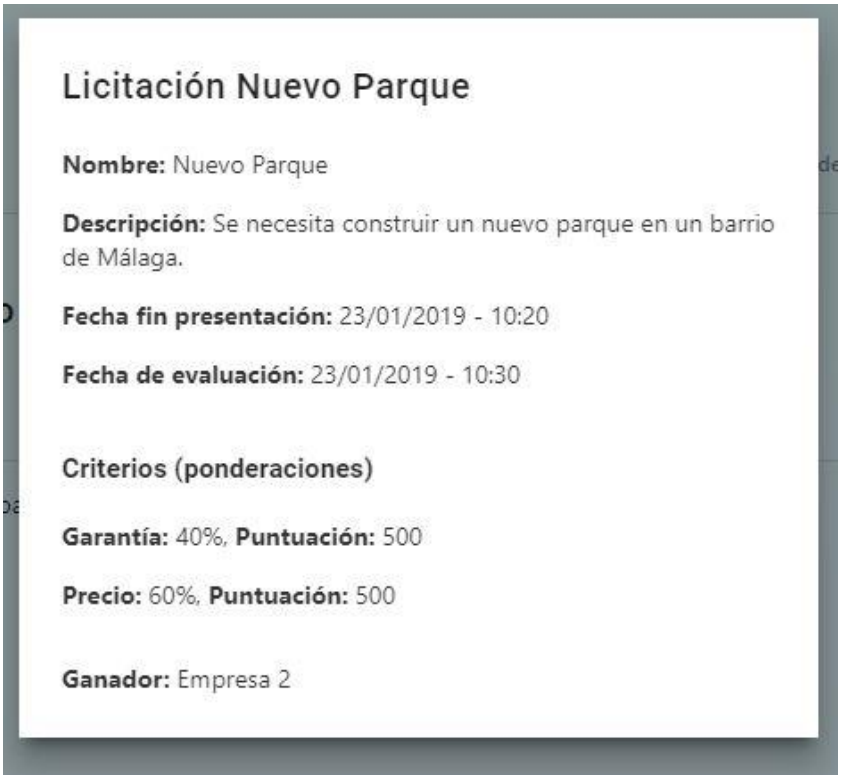


Figura 5.42: vista de una licitación en la que no ha ganado el licitador registrado

Vista de una licitación por parte de un licitador en la que ya se ha decidido el ganador y no es el licitador registrado en la aplicación en ese momento.

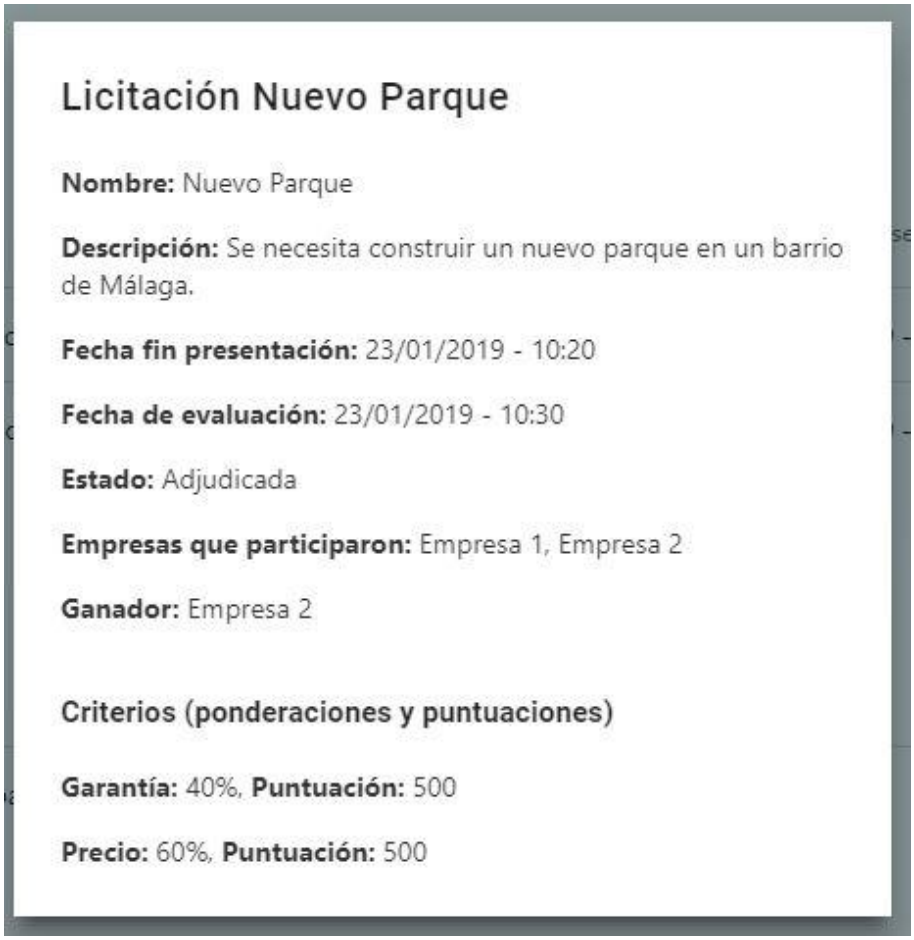


Figura 5.43: vista de una licitación adjudicada por parte de un ciudadano

Vista de una licitación adjudicada por parte de un ciudadano. Puede ver el ganador, las empresas que han participado y el resto de información. El gestor también puede ver las licitaciones y tiene las mismas vistas que las de un ciudadano.

En muchas partes de las vistas de la aplicación pueden verse mensajes de error o de éxito dependiendo de las acciones realizadas. Estos mensajes aparecen tanto en formularios como después de conseguir realizar operaciones satisfactoriamente, como crear una licitación. Un ejemplo ya se ha visto en la figura 5.35.

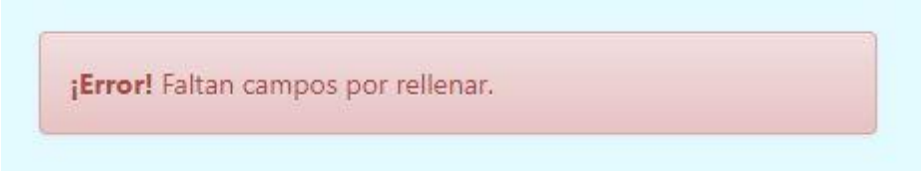


Figura 5.44: mensaje de error

Vista de un mensaje de error típico. Todos los mensajes de error tienen un aspecto similar al mostrado en la figura 5.44, pero cambiando el texto mostrado.

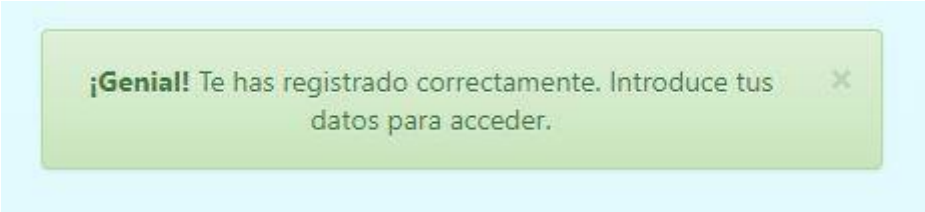


Figura 5.45: mensaje de éxito

Vista de un mensaje de éxito típico. Como en el caso de los de error, todos los mensajes de éxito tienen un aspecto similar, pero cambiando únicamente el texto mostrado.

Se puede ver en las distintas figuras mostradas en este apartado que las vistas en la parte del cliente han sido muy parecidas a lo que se mostraba en los bocetos de interfaz de usuario, teniendo prácticamente todas las características similares.

Tras mostrar el aspecto del cliente, se procederá a explicar algunos puntos interesantes sobre el desarrollo de este.

Como ya se ha comentado con anterioridad, para la comunicación mediante peticiones HTTP entre el cliente y el servidor se ha utilizado la librería axios. Se puede ver un ejemplo de su uso en la figura 5.46.

```
login() {
    axios.post('http://localhost:3001/login/bidder', {email: this.state.email, password: this.state.password})
        .then(res => {
            Auth.authenticateUser(res.data.token, res.data.idBD, res.data.name);
            this.props.history.push("/bidder/tenders");
        })
        .catch(err => {
            this.setState({
                messageAlert: err.response.data,
                messageStrongAlert: "¡Error!",
                typeMessageAlert: "danger",
                showAlert: true
            });
        });
}
```

Figura 5.46: ejemplo de uso de axios

En el caso de la figura 5.46, se está utilizando axios para realizar una petición POST. Se trata del proceso de acceso por parte de un licitador. Se envía el email y la contraseña y en caso de que los datos sean correctos, se accede a la aplicación. En caso de error, se muestra el mensaje enviado por el servidor.

Se comentó anteriormente que se usaba un sistema de tokens para saber si un usuario había conseguido acceder a la aplicación y que no se le permitía acceder a páginas a las que no tenía permiso. Esto se hace utilizando módulo llamado *Auth*, implementado por el propio desarrollador que hace uso del almacenamiento local del navegador. Se puede ver en la figura 5.46 cómo se llama a un método de *Auth* y se le

pasan varios parámetros una vez el servidor ha validado la identidad del licitador. *Auth* almacena estos datos en el almacenamiento local para luego ser llamados por el cliente y ver si el usuario tiene permiso para acceder a cierta página.

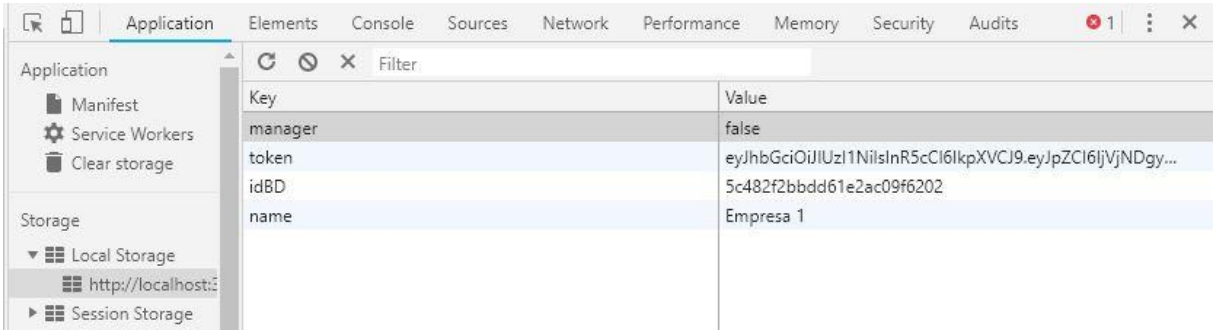


Figura 5.47: uso del almacenamiento local

Se pueden ver en la figura 5.47 los datos almacenados por parte del cliente en el almacenamiento local. Al estar el campo *manager* con el valor negativo, no podrá acceder a las páginas del gestor.

Cabe mencionar que se han usado varias librerías para utilizar componentes de React. La más importante y la más utilizada es react-md[22], una librería que proporciona componentes con un estilo y colores muy estéticos.

6. CONCLUSIONES Y FUTURO

En este apartado se detallarán las conclusiones obtenidas de la realización del proyecto y se comentarán posibles funcionalidades o ideas relacionadas con el entorno del proyecto para implementar en un futuro.

Se ha completado el desarrollo de una aplicación web que se encarga de manejar procesos de licitaciones en un entorno informático. Se considera que se han cumplido con los objetivos previstos en el Documento General de Requisitos y se han seguido las especificaciones detalladas en los bocetos de interfaz de usuario presentados en las primeras fases del proyecto.

Se trata de una tecnología, blockchain, para la cual el desarrollador no tenía ningún tipo de formación previa al inicio de este proyecto. Tras horas de investigación, realización de cursos y mucha documentación[23] [24] [25] [26], se ha realizado un proyecto que puede servir de base para futuros desarrollos en esta misma línea.

Es un tipo de proyecto que se está llevando a cabo en otros lugares, como Aragón, que demuestra que las instituciones empiezan a apostar por este tipo de tecnologías para suplir las deficiencias de los actuales sistemas en uso. Centrar los esfuerzos en una tecnología en la cual es casi imposible hacer un uso fraudulento de sus funcionalidades puede ser interesante para ahorrar grandes cantidades de dinero a las arcas públicas.

Lo ideal sería que este proyecto fuese el inicio de una sucesión de trabajos de fin de estudios o similares como continuación de este que permitiese ir añadiéndole cada vez más funciones y que se acabase convirtiendo en un sistema real en uso por parte de una o varias entidades públicas. En este momento es un sistema con unas funcionalidades básicas. Se han implementado unas conexiones con la red blockchain y un uso de esta que pueden servir de ejemplo para futuros desarrolladores interesados en continuar con esta línea de trabajo.

Existen multitud de mejoras que se le pueden hacer para continuar con el proyecto. Se expondrán aquí las más cercanas al momento actual en el que se encuentra y algunas pensadas para un futuro, se espera, no muy lejano.

Actualmente, y dado la situación de que es un proyecto inicial, toda la información relativa a las licitaciones se pide directamente a la red blockchain. La tecnología blockchain tiene muchos beneficios, pero la rapidez no es uno de ellos. Podría ser interesante implementar algún sistema que almacenase la información en

memoria y la pidiese de nuevo a la red blockchain cuando fuera necesario realizar alguna modificación o cada cierto tiempo estipulado. Esto permitiría seguir teniendo información real y validada por la red y obtenerla de forma rápida por parte del cliente al estar almacenada en la memoria del servidor.

Como ya se ha comentado en esta memoria, actualmente el sistema hace uso de Moment para enviar la hora a un smart contract y que este realice ciertas operaciones o no en función de la hora recibida. Esto le quita algo de fiabilidad al sistema, ya que no permite al smart contract de la red blockchain ser totalmente independiente, una vez creado, del servidor y le obliga a fiarse de su información en cuanto al tema del horario. Una función interesante a desarrollar sería el uso, por parte del smart contract, de una API o servicio web externo que diese la hora. Alguno ajeno al sistema y que no pueda tener conflictos de interés. Parece ser que se usa Oraclize, pero queda en manos del futuro desarrollador o algún interesado en el proyecto qué servicio usar.

También se ha comentado en esta memoria que actualmente no se usan números decimales en el smart contract. Es una función que Ethereum tiene, pero no implementada aún por completo. Cuando se permita su uso de forma normal, o se quiera utilizar algún servicio externo, se podría cambiar la fórmula para elegir ganador del smart contract. Si bien la actual cumple su función, una función que utilizase decimales sería más correcto.

Otro problema que se ha planteado durante el desarrollo del proyecto ha sido el hecho de que era necesario almacenar los ids de la base de datos de los licitadores que participan en una licitación. Esto, en el período de presentaciones, supone un problema. La cuestión es que es necesario para luego localizarlos en la base de datos. Podría ser interesante buscarle una solución al problema comentado.

Ya se comentó que el ahorro de gas, que en definitiva es dinero por parte de la entidad que acabe utilizando este sistema, no era uno de los objetivos de este proyecto. Aun así, en un futuro, la entidad que lo utilice estará interesada en ahorrar lo máximo posible para llevar a cabo los procesos de licitaciones. Por lo que también podría estudiarse fórmulas para gastar el mínimo gas necesario.

Por último, se planteó antes de comenzar este proyecto, en una reunión, una idea que podía ser muy interesante. Se podría desarrollar una red blockchain a nivel estatal. Esta red, ya utilice tecnología Ethereum o alguna similar para utilizar servicios como los de los smart contracts, podría servir para suplir las necesidades de aplicaciones como la que se ha desarrollado en este proyecto. Y más aún, podrían

desarrollarse aplicaciones similares para otros ámbitos que no sean las licitaciones que tendrían cabida en esa red blockchain que se menciona. La red de nodos podría desplegarse provincialmente, por ejemplo, y dedicar un presupuesto por parte del gobierno estatal o de cada gobierno provincial a desarrollarla y/o mantenerla.

Esta última idea, obviamente, no es algo que esté en manos del desarrollador ni de los interesados en el proyecto. Pero estaría bien tratar de comentarla, de hacerla visible y que llegase a manos de los políticos para tratar de hacerla realidad. Igual que en los presupuestos de los diferentes gobiernos, tanto estatal, como autonómicos, provinciales, etc., se destina unas cantidades para ciertas necesidades, podría incluirse una cantidad para crear esta red blockchain a nivel estatal. Tendría que ser estudiado por expertos en la materia, pero todo parece indicar que con el dinero que se ahorraría al evitar casos de corrupción, llevados a cabo por la poca transparencia y todo el mal uso que se hace de ciertos procesos administrativos, se pagarían los costes de esta red.

Bibliografía

[1] Página web de Ethereum https://ethereum.org/
Consultado el 25/01/2019

[2] Página web de Impulso TFE UMA con la descripción del proyecto
https://www.impulsotfeuma.es/jobs/Uso-de-la-Tecnolog%EDa-Blockchain-en-contrataci%F3n-p%FAblica/
Consultado el 25/01/2019

[3] Página web de Bitbucket https://bitbucket.org/
Consultado el 25/01/2019

[4] Documentación de Node.js https://nodejs.org/es/docs/
Consultado el 25/01/2019

[5] Página web de Express https://expressjs.com/
Consultado el 25/01/2019

[6] Documentación de React https://reactjs.org/docs/getting-started.html
Consultado el 25/01/2019

[7] Documentación de axios https://github.com/axios/axios
Consultado el 25/01/2019

[8] Página web de MongoDB https://www.mongodb.com/es
Consultado el 25/01/2019

[9] Página web de Mongoose https://mongoosejs.com/
Consultado el 25/01/2019

[10] Documentación de Ganache
https://truffleframework.com/docs/ganache/overview
Consultado el 25/01/2019

[11] Documentación de Web3
https://github.com/ethereum/wiki/wiki/JavaScript-API
Consultado el 25/01/2019

[12] Página web de Sublime Text https://www.sublimetext.com/
Consultado el 25/01/2019

[13] Documentación de Solidity https://solidity-es.readthedocs.io/es/latest/
Consultado el 25/01/2019

[14] Página web de Balsamiq Mockups https://balsamiq.com/wireframes/
Consultado el 26/01/2019

[15] Página web de nodemon https://nodemon.io/
Consultado el 26/01/2019

[16] Página web de instalación de bluebird
https://www.npmjs.com/package/bluebird
Consultado el 26/01/2019

[17] Documentación de Passport
http://www.passportjs.org/docs/downloads/html/
Consultado el 26/01/2019

[18] Página de descarga de Advanced REST Client
https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo?hl=es
Consultado el 26/01/2019

[19] Página web de Moment https://momentjs.com/
Consultado el 26/01/2019

[20] PDF con información del proyecto de Aragón
https://contrataciondelestado.es/wps/wcm/connect/73a57190-a4bc-4119-b0d3-563afef522d9/DOC20180917180500PPT+HAP+SCC+24+2018.pdf?MOD=AJPERES
Consultado el 29/01/2019

[21] Página web de instalación de solc https://www.npmjs.com/package/solc
Consultado el 26/01/2019

[22] Página web de react-md https://react-md.mlaursen.com/
Consultado el 26/01/2019

[23] Página web de Stack Overflow https://stackoverflow.com/
Consultado el 25/01/2019

[24] Página web de Ethereum StackExchange
https://ethereum.stackexchange.com/
Consultado el 25/01/2019

[25] Tutorial de JavaScript https://www.w3schools.com/js/
Consultado el 25/01/2019

[26] Página web de Udemy https://www.udemy.com/
Consultado el 26/01/2019

Anexo I: Documento General de Requisitos

En este apartado se incluye el Documento General de Requisitos.

Documento general de requisitos

Proyecto: Sistema de licitaciones electrónicas

Autor: Pablo Hijano Calderón

Índice

1 INTRODUCCIÓN

1.1 OBJETIVOS

El objetivo principal de este TFG es desarrollar, siguiendo las peticiones del cliente (Diputación de Málaga) un sistema de licitaciones electrónicas que permita realizar procesos de licitación con un alto grado de transparencia y con garantías que impidan la modificación de las ofertas enviadas.

1.2 ALCANCE

El alcance de este proyecto abarca tanto a la Diputación de Málaga (y entidades similares) como a los usuarios que podrán hacer uso de él. Entre estos usuarios se encuentran empresas, autónomos y ciudadanos de a pie.

1.3 DEFINICIONES, ACRÓNIMOS Y ABREVIATURAS.

Licitación: subasta por parte de una entidad pública en la que se lanza un proyecto para que cualquier empresa o autónomo que quiera pueda presentar una oferta. Tras un proceso de selección se elige a un ganador en base a varios criterios.

Licitador: puede referirse tanto a la entidad que publica la licitación como a los participantes que presentan las ofertas a dicha licitación.

Blockchain: tecnología basada en cadenas de bloques. Consiste en una red (pública o privada) en la que todos los nodos que forman dicha red mantienen los mismos bloques de datos y validan los nuevos.

Smart Contract: programa informático ejecutado en una red blockchain. Una vez lanzado, el código es inmodificable.

1.4 REFERENCIAS

La nueva ley 9/2017 en la que aparece el artículo 159, donde se explica el Procedimiento abierto simplificado.

- https://www.boe.es/buscar/pdf/2017/BOE-A-2017-12902-consolidado.pdf

Sobre licitaciones con el sistema actual. Vídeos realizados por la Diputación de Málaga.

- https://www.youtube.com/watch?v=YxA2QYwzHDI (Parte 1 - Introducción)
- https://www.youtube.com/watch?v=Hkc4TnrBdqQ (Parte 2 – Introducción a la Plataforma)
- https://www.youtube.com/watch?v=dVI93vgkXNc (Parte 3 – Servicio a usuarios registrados: Mis Comunicaciones)
- https://www.youtube.com/watch?v=PTjDh_zp80I (Parte 4 – Presentación de Ofertas)

Sobre Ethereum.

- https://www.ethereum.org/

Sobre truffle (framework de Ethereum).

- https://trufflеframework.com/docs/truffle/overview

Sobre Ganache (Entorno de pruebas que simula una red blockchain).

- https://trufflеframework.com/docs/ganache/overview

1.5 RESUMEN

El sistema que se busca diseñar pretende hacer de las licitaciones un proceso mucho más transparente, para permitir la auditoria del proceso por agentes externos, incluyendo a los ciudadanos.

Además se busca que sea imposible la modificación a posteriori de una oferta ya presentada. Con ello se impedirían irregularidades, como que aumente el presupuesto de una oferta una vez que se ha sabido ganadora.

Existen otros objetivos secundarios, como que el sistema calcule automáticamente una valoración, con la cual se decidirá la oferta ganadora.

2 DIRECTIVAS DEL PROYECTO

2.1 ANTECEDENTES

La metodología que se usa actualmente, para el funcionamiento de las licitaciones en la gran mayoría de entidades, consiste en el envío de sobres con información de las ofertas presentadas. A posteriori, se elige un ganador tras revisar todas las ofertas.

La problemática principal viene de que una vez se han sabido ganadoras, muchas empresas inflan su presupuesto para generar mayor beneficio propio. Esto, obviamente, perjudica a las arcas públicas.

Además de esto, existe un problema de transparencia muy grande, ya que cualquier ciudadano que quiera no puede ver qué empresas se han presentado a una licitación.

2.2 PROPUESTA DE SOLUCIÓN

La solución que se propone para solventar dichos problemas es un sistema que haga uso de la tecnología blockchain. Utilizando una red de cadena de bloques podremos garantizar que una vez se ha enviado una oferta, esta no se puede modificar. Para ello, la red comprobará que el hash enviado durante la fecha de presentación de ofertas coincide con el hash enviado durante la evaluación de las ofertas.

La red blockchain, al ser pública, nos permite ver la información almacenada en ella. De manera que podremos ver qué empresas están participando en una licitación, no sólo la que resulte ganadora.

3 DESCRIPCIÓN DE PARTICIPANTES Y USUARIOS

3.1 RESUMEN DE LOS PARTICIPANTES

Nombre	**Rol**
Diputación de Málaga	Se encargará de utilizar el sistema para crear nuevas licitaciones en las que las empresas puedan participar.

3.2 RESUMEN Y ENTORNO DE LOS USUARIOS

Nombre	**Descripción**
Empresa o autónomo.	Presenta su oferta a la licitación y ve el resultado una vez llega la fecha de publicación de los resultados.
Ciudadano	Puede ver las empresas que se han presentado a una licitación y ver quién ha ganado y el resto de detalles cuando se publiquen los resultados.

3.3 PERFILES DE LOS PARTICIPANTES

3.3.1 Diputación de Málaga

Representante	Juan Antonio Ríos Peláez
Tipo	Responsable de Departamento Informático
Responsabilidades	Especificación de requisitos y validador de los resultados.
Criterio de éxito	La implementación de un sistema que permita gestionar licitaciones de forma transparente y segura con las funcionalidades requeridas.
Entregables	Proyecto software que cumpla con los requisitos especificados.
Comentarios	-

3.4 PERFILES DE USUARIO

3.4.1 Empresa o autónomo.

Representante	Entidad que hará uso del sistema.
Descripción	Podrá utilizar el sistema para participar en licitaciones
Tipo	Personal especializado en sistemas IT.
Responsabilidades	Responsabilidades cubiertas por Juan Antonio, que actúa como representante.
Criterio de éxito	Agilizar los trámites en el proceso de las licitaciones.

3.4.2 Ciudadano

Representante	Persona que hará uso del sistema.
Descripción	Podrá utilizar el sistema para ver qué empresas participan en ciertas licitaciones.
Tipo	Persona que no tiene por qué estar habituada al uso de sistemas IT.
Responsabilidades	Responsabilidades cubiertas por Juan Antonio, que actúa como representante.
Criterio de éxito	Permitir la transparencia de las licitaciones.

3.5 ALTERNATIVAS Y COMPETENCIA

3.5.1 Aragón

En Aragón se está implementando un sistema con características muy similares al que se busca desarrollar en este proyecto. Algunas características han sido obtenidas a partir de dicho proyecto.

Por ejemplo, el procedimiento para comprobar que la oferta a una licitación no se ha modificado será el mismo. El usuario enviará el hash de su oferta, dentro del plazo de presentaciones, y este se almacenará en la red blockchain. Una vez acabe el plazo y llegue el periodo de evaluación, el usuario volverá a mandar el hash y este será comparado con el ya almacenado. Si no coincide no podrá participar en la licitación.

En su caso, harán un sistema mucho más completo que permita ser utilizado en un entorno real. Este proyecto abarca solo una parte del funcionamiento total que presentará Aragón, dado que es un TFG. Se necesitarían algunas continuaciones posteriores para acabar teniendo un sistema similar.

Se puede encontrar más información en distintas webs. Por ejemplo:
http://www.expansion.com/aragon/2018/09/17/5b9fd66dca4741b1408b4590.html

4 VISION GENERAL DE PRODUCTO

4.1 ENTORNO DE DESPLIEGUE

A expensas de que lo decida el cliente, lo más seguro es que el sistema se encontrará funcionando en algún servidor dedicado de la Diputación de Málaga. Estando operativo para recibir cualquier petición hecha por alguno de los interesados.

4.1.1 Entorno para la implementación del sistema actual

Será decisión del cliente.

4.1.2 Aplicaciones colaboradoras

El sistema trabajará de forma aislada a priori. En un futuro, con mayores avances, seguramente trabaje con plataformas otras plataformas de licitaciones.

4.2 RESUMEN Y CARACTERÍSTICAS

Beneficios para el cliente	Características de soporte
Acceder a licitaciones	Visualización y participación en licitaciones.
Información de participaciones	Tanto ciudadanos como empresas y autónomos podrán ver las participaciones en una licitación.
Seguridad	Se garantizará la no modificación de los datos una vez enviados.
Valoración automática	Se calculará el ganador de la licitación de forma automática.

4.3 SUPOSICIONES Y DEPENDENCIAS

4.3.1 Factores externos que tienen un efecto en el producto, pero no son restricciones obligatorias

No se encuentra ninguno en principio.

4.3.2 Suposiciones que asume el equipo en torno al proyecto

Se supone que las personas que vayan a hacer uso del sistema tienen un mínimo conocimiento sobre los procedimientos electrónicos.

4.4 PRECIO Y COSTE

El único coste puede ser el de la utilización de la red blockchain de Ethereum. Una idea futura sería que esta red fuese privada de ámbito estatal. Se comentará en detalle en futuros documentos.

4.5 LICENCIAS DE INSTALACIÓN

No requiere licencia.

4.6 PAQUETES COMERCIALES

- El sistema de base de datos será MongoDB.
- Se utilizará el framework Express, basado en NodeJS, para la parte del servidor.
- Para la parte del cliente, se utilizará React.
- Para las peticiones HTTP entre cliente y servidor, se utilizará la API axios.
- El proyecto utilizará el framework Truffle para las comunicaciones con la red blockchain.
- El entorno de pruebas de la red blockchain será Ganache.

5 REQUISITOS FUNCIONALES

Nota: tendremos 2 tipos de usuarios. Los que tendrán acceso a la participación en licitaciones (empresas y autónomos) y los que únicamente podrán ver las participaciones en licitaciones. Estos últimos no necesitarán estar registrados para acceder a la información. Para simplificar llamaremos a los primeros *licitadores* y a los segundos simplemente *usuarios*.

1. *Registrar un nuevo licitador:* el sistema permitirá registrar nuevos licitadores a partir de unos datos pedidos.
2. *Acceso por parte de un licitador:* el sistema permitirá acceder, mediante un sistema de acceso, a un licitador para que pueda hacer uso del sistema.
3. *Configurar una nueva licitación:* el sistema permitirá la creación de licitaciones por parte de un gestor registrado.
4. *Presentar propuesta a una licitación*: el sistema permitirá la recepción de archivos por parte de un licitador, de los cuales únicamente almacenará su hash y algunos datos para identificarlo y así poder incluirlo en la red blockchain. Estos datos no tendrán información de la oferta en sí.
5. *Enviar propuesta para valoración:* el sistema permitirá enviar la oferta por parte de un licitador. Si el hash de esta concuerda con el almacenado en la red blockchain, se procederá a su evaluación.
6. *Ver ofertas presentadas a una licitación:* el sistema permitirá a cualquier usuario ver las ofertas presentadas a una licitación una vez finalizado el periodo de presentación de ofertas.
7. *Seleccionar oferta ganadora:* el sistema, mediante la blockchain, seleccionará la oferta ganadora en base a la fórmula utilizada para la valoración.

6 PRECEDENCIA Y PRIORIDAD

Todavía estamos en una etapa muy temprana del desarrollo.

7 REQUISITOS NO FUNCIONALES

1. *Función hash para comparación de documentos:* se utilizará una función hash para comparar los documentos enviados por parte del cliente durante los procesos de presentación y evaluación.
2. *Uso de la red Ethereum*: se utilizará la red de blockchain Ethereum y los Smart Contracts para algunos requisitos del proyecto.

8 MODELO DE DOMINIO