Grau en Enginyeria Informàtica

Treball de Fi de Grau

Desenvolupament d'un prototipus d'un gestor de dades mestres al núvol

Autor:
Javier GASCH RUIZ

Supervisor:
Vicente AGOST ALCÓN
Tutor acadèmic:
José Luis LLOPIS BORRÁS

Data de lectura: 26 de juny de 2020
Curs acadèmic 2019/2020

Resum

En aquest document es descriu el procés de desenvolupament d'un prototipus de gestor de dades mestres. Aquest prototipus serveix per a gestionar els recursos que vol emmagatzemar una empresa i realitzar operacions de lectura, creació, modificació i eliminació sobre ells. L'estructura del sistema compta amb el *frontend*, el *backend*, i la base de dades. El *backend* fa d'intermediari per a la comunicació de les parts i la informació que s'envia entre les parts està en format *JSON*. Per a cada recurs que vol gestionar el client es té un arxiu *JSON Schema* i partir d'aquest es crearà una taula en la base de dades. Per a generar les vistes del *frontend* també es farà ús d'aquest esquema. A la memòria es fa una xicoteta introducció per definir els objectius i els conceptes dels que es parlarà durant tot el document, com l'anomenat anteriorment *JSON Schema*, després s'expliquen les fases de planificació, anàlisi, disseny i implementació.

Paraules clau

- Gestor de dades mestres
- Esquema JSON
- Recurs
- CRUD (Crear, llegir, actualitzar, eliminar)

Keywords

- Master Data Management
- JSON Schema
- Resource
- CRUD (Create, read, update, delete)

Índex

Índex de figures

Índex de taules

Capítol 1

Introducció

1.1 Context i motivació del projecte

SiGO información y gestión és una empresa tecnològica que desenvolupa aplicacions per millorar la gestió de les ciutats i empreses. Compten amb una aplicació participativa de gestió y control per a gestionar incidències i per a gestionar xarxes de serveis i infraestructures. El seu pròxim projecte a desenvolupar, solucionarà un problema que tenen molts dels clients amb els que treballen.

En els entorns empresarials moderns hi ha diversos sistemes informàtics que fan ús de dades mestres o recursos de l'empresa, com per exemple els empleats, la maquinària i els materials. Açò suposa un problema, les empreses deuen procurar tindre un únic punt d'informació per mantindre la integritat i seguretat.

A més, deuen fer ús de ferramentes per mantindre aquesta informació ($CRUD$) i altres que permetin l'accés controlat a la mateixa des de sistemes externs (API).

Per a facilitar estes qüestions es molt important disposar de la millor informació possible de les dades mestres o recursos. Per a aquestos en general es tenen les metadades, en concret *JSON Schema*, el qual ens permet posseir unes metadades que siguen interpretables tant per humans com per màquines.

Per tal de solucionar aquest problema d'integritat i seguretat, en *SiGO* volen crear un gestor de dades mestres en el núvol, *MDM (Master Data Management)* per a que els clients puguen gestionar millor les seues dades.

1.2 Objectius del projecte

El principal objectiu d'aquest projecte és crear un prototipus d'un gestor de dades mestres.

L'objectiu principal es pot desglossar en els següents subobjectius:

- Conèixer els sistemes de gestió de dades mestres.
- Formació amb les tecnologies emprades per l'empresa.
- Implementar un *CRUD* (Crear, Llegir, Actualitzar i borrar) per a un recurs mestre mitjançant un esquema *JSON*.
- Implementar la base de dades a la que es connectarà el servidor.
- Fer la *API* a la que es connectarà la part del client.
- Desenvolupar una interfície gràfica que siga capaç d'interpretar un *JSON Schema*.

Els resultats esperats són desenvolupar un prototipus que a la llarga es duga a terme i que els clients siguen els que modifiquen els recursos de les aplicacions segons les seues necessitats i que *SiGO* es puga desentendre d'aquesta part.

Per una banda, l'abast funcional inclou desenvolupar un prototipus d'un gestor de dades mestres. Aquest prototipus permetrà la implementació ràpida d'un *CRUD* (Crear, Llegir, Actualitzar i esborrar) per a un recurs mestre mitjançant la lectura d'un *JSON schema*. Una vegada definit l'esquema *JSON* es crearà de manera automàtica els llistats y formularis per a mantindre el recurs: llistat, filtrat, creació, edició y esborrat.

Al ser un projecte gran, en un principi l'abast funcional no inclou implementar un sistema d'autenticació d'usuaris, el control de permisos i la creació de tests. Però depenent del temps restant es podria arribar a implementar alguna d'aquestes funcionalitats.

Per a l'abast organitzatiu, compta amb l'ajuda del programador de l'empresa i del supervisor durant tota a durada del projecte.

Per a cobrir l'abast informàtic, les tecnologies que s'empraran són *PhpStorm* amb la biblioteca *Laravel* per a realitzar el *API Rest*. Per al servidor y la base de dades utilitzarem *xampp* que fa ús d'*Apache* per la part del servidor i *MySQL* per a la part de la base de dades. Finalment s'utilitzarà VueJS amb Vuetify per a la part del front-end.

1.3 Descripció del projecte

1.3.1 Informació addicional

Primer de tot, per a impedir possibles confusions, en a aquest document em referiré als recursos mestres que gestionen en una empresa (per exemple els empleats o els materials) com a *models*

i *recursos*. Per tant el *model materials* i el *recurs materials*, és el mateix. A més cada recurs tindrà la respectiva taula a la base de dades.

Per tal de millorar el manteniment i l'eficiència, la creació de les taules de la base de dades s'haurà de dur a terme de manera automàtica, ja que si una empresa ha de gestionar molts recursos, es faria pesat el haver de crear cada taula i modificar-la manualment.

Aquestes creacions i modificacions de les taules es faran mitjançant la lectura de *JSON Schema*. Cada taula tindrà un arxiu *.json* i el *backend* serà l'encarregat de llegir aquests arxius per la creació i modificació de les respectives taules.

Com el J*SON Schema* és una part important, per no dir principal, del projecte, vaig a explicar en detall de que es tracta, quina és la seua funcionalitat i com és farà ús d'aquests arxius al projecte.

Abans de començar a definir què és un *JSON Schema*, caldria definir primer què és *JSON*. Les sigles *JSON* signifiquen *JavaScript Object Notation*, és a dir, la notació d'un objecte *JavaScript*.

Aquest format de guardar les dades va sorgir en primer lloc per a la *World Wide Web*. Com la majoria dels navegadors fan ús de *JavaScript*, aquesta notació era molt fàcil d'integrar. Va ser de tanta utilitat per a intercanviar les dades i a la vegada tan fàcil d'utilitzar, que el seu ús no es va quedar aquí i ara s'utilitza en altres contextos que no impliquen la navegació web.

A continuació l'exemple més simple d'un *JSON*:

{"nom": "Javi"}

Amb aquesta informació, es poden representar moltes estructures de dades. Per mostrar un exemple, podem observar la figura 1.1 tret de la pàgina oficial de *JSON Schema* [1].

```
{
  "first_name": "George",
  "last_name": "Washington",
  "birthday": "1732-02-22",
  "address": {
    "street_address": "3200 Mount Vernon Memorial Highway",
    "city": "Mount Vernon",
    "state": "Virginia",
    "country": "United States"
  }
}
```

Figura 1.1: Objecte representat amb la notació *JSON*.

Un mateix objecte es pot estructurar de varies maneres i ninguna ha de ser més correcta que l'altra. Per exemple, a la figura anterior podria haver una clau anomenada *"name"* que arreplega la informació dels dos noms definits, com ho fa "*address*".

Per tant, quan una aplicació requereix d'un *JSON*, no val passar-li l'objecte en qualsevol estructura, cal passar-s'ho amb l'estructura definida en l'aplicació. És ací on entra en joc el *JSON Schema*.

La figura 1.2 seria el *JSON Schema* de l'anterior exemple:

```
{
  "type": "object",
  "properties": {
    "first_name": { "type": "string" },
    "last_name": { "type": "string" },
    "birthday": { "type": "string", "format": "date" },
    "address": {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" },
        "country": { "type" : "string" }
      }
    }
  }
}
```

Figura 1.2: *JSON Schema* de l'anterior objecte.

Com es pot observar, es defineix el tipus per a cada clau de l'objecte. A més d'aquesta informació per cada clau, es poden definir més propietats com es veurà més endavant amb un exemple avançat.

Els tipus de camps que permet la definició de *JSON Schema* són els següents:

- String
- Integer
- Number
- Boolean
- Array
- Object
- null

Com a valor afegit, als *strings* se li pot afegir la propietat "format", aquesta propietat permet la definició de tres nous tipus de camps, com ara són:

- "format"= "date"

- “format”= “time”
- “format”= “date-time”

Tots ells definits com un *string*.

Per al prototipus de gestor de dades mestres, l'estructura que tindran els JSON Schema, serà la mostrada a la figura 1.3.

```
{
  "$id": "https://example.com/geographical-location.schema.json",
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Longitude and Latitude Values",
  "description": "A geographical coordinate.",
  "required": [
    "latitude",
    "longitude"
  ],
  "type": "object",
  "properties": {
    "latitude": {
      "type": "number",
      "minimum": -90,
      "maximum": 90
    },
    "longitude": {
      "type": "number",
      "minimum": -180,
      "maximum": 180
    }
  }
}
```

Figura 1.3: Estructura per al prototipus del *JSON Schema*

Les noves claus que són relevants per a la implentació són “title” i “required”. La primera, *title*, definirà el nom del recurs mestre, és a dir, el nom de la taula que es crearà a la base de dades referent a aquest objecte. La segona, *required*, és un *array* on es ficaran tots els camps obligatoris, és a dir, totes les columnes que no poden emmagatzemar un valor *null*. Les columnes del model seràn definides dins de la clau *properties*.

Finalment, per explicar un poc com serà el funcionament referent al JSON Schema al prototipus, faré com una petita introducció aquí i s'entrarà en detall a la secció d'implementació.

Primer de tot, per cada recurs que es vulgui gestionar, es tindrà un *JSON Schema*. És a dir, si una empresa vol gestionar la informació dels seus empleats i dels seus materials, hi haurà dos arxius al *backend* que faran referència als *JSON Schema* del recursos nombrats.

Bé doncs, aquests arxius tindran dos funcions principals. La primera d'elles, serà la creació de les taules a la base de dades. Els arxius seran llegits per una funció, i es crearan les respectives taules a la base de dades. I la segona d'elles, serà la creació automàtica de formularis al *frontend*. Hi ha un paquet per a *VueJS*, que al obtindre un *JSON Schema*, et genera automàticament un formulari amb tots els camps definits i les respectives regles de validació. Estes dos funcions s'explicaran en més detall en la secció de funcionament del *backend* i *frontend*.

1.3.2 Tecnologies

Per a realitzar el projecte es farà ús d'una serie de tecnologies:

PhpStorm

Es tracta d'un *IDE* per a poder treballar amb *Laravel*. Encara això, serveix també per altres llenguatges que no siguen *php*. De fet la part del *frontend*, també serà desenvolupada en aquest *IDE*.

Laravel

Laravel és un *framework* per a desenvolupar aplicacions i serveis web amb *php*. El seu abast inclou *backend* i *frontend*, però en aquest projecte sols es desenvoluparà la part del *backend*. La gran característica d'aquesta tecnologia és que ja té molta funcionalitat implementada per defecte i gràcies als respectius *facades* es pot accedir a tots aquests mètodes que agiliten molt el treball. Per exemple dos dels *facades* que més es gastarà al projecte seran el *Schema* i el *DB*, aquests dos serveixen per simplificar la comunicació amb la base de dades.

Xampp

Un paquet de *software* lliure que inclou un sistema de gestió de base de dades *MySQL* i el servidor de web *Apache*.

Postman

Aquesta ferramenta serveix per fer testejar les peticions a una *API* sense haver de desenvolupar la part del client.

VueJS

Aquest és un *framework* de *JavaScript* per a desenvolupar interfícies gràfiques d'una sola pàgina. La gran característica d'aquesta tecnologia és que la part *html* i *javascript* es troben al mateix fitxer, per lo que la comunicació entre la part gràfica i la funcionalitat interna és molt fàcil. Aquest marc es basa amb els components, és a dir a les webs desenvolupades amb *VueJS* hi ha una sola pàgina real a la que van apareixen i desapareixen els components en funció de les accions de l'usuari.

A continuació alguns dels paquets utilitzats en *VueJS* que val la pena nombrar en aquesta secció.

Vuetify

Un paquet *Material Design* per a *VueJS*. Aquest component afegeix funcionalitat i estil adicional als components de *VueJS*.

Koumoul/vjsf

Un altre paquet per a VueJS. Aquest paquet serveix per la generació automàtica d'un formulari a partir d'un *JSON Schema*.

Axios

El paquet utilitzat per a la comunicació amb la *API* mitjançant les peticions.

1.4 Estructura de la memòria

La memòria està estructurada de la següent forma.

En primer lloc, al capítol 2 es mostra la planificació del projecte. A la secció de metodologia s'explica quina metodologia s'ha seguit durant el projecte, a la secció de planificació es pot veure com s'ha planificat el projecte i la pila del producte. Després està la secció d'estimació de recursos i costos del projecte i la de seguiment del projecte en la qual es mostre com han anat cadascun dels *sprints*.

En segon lloc, al capítol 3 es mostra l'anàlisi i disseny del sistema. A la secció d'anàlisi del sistema es defineixen els requisits de l'aplicació. Després està la secció sobre l'arquitectura del sistema i la de disseny de la interfície.

Seguidament, al capítol 4 trobem l'explicació de la implementació i proves. En quant a la

secció de detalls d'implementació es divideix en un primer lloc en *backend* i *frontend* i aquestes dos a la vega es divideixen amb estructura, funcionament i problemes. Després està la secció on s'expliquen les proves i la de les posibles millores de l'aplicació.

Finalment, el capítol 5 on es troba la conclusió de la memòria. A més, després de la conclusió s'inclou la bibliografia.

Capítol 2

Planificació del projecte

2.1 Metodologia

La metodologia de treball que es va a seguir durant la realització del projecte és la seguida per l'empresa, que serà de tipus àgil, similar a *Scrum*.

En la planificació es realitza un pila de producte inicial amb les tasques del projecte. Al tractar-se d'una metodologia àgil la pila pot sofrir modificacions al llarg del projecte.

Els *sprints* tindran una durada de dos setmanes i les reunions no seran obligatòries en cada *sprint*, hi haurà *sprints* que tinguen varies reunions, tant a principi, a meitat i a final, i altres *sprints* que no tinguen cap reunió. Tot depenent de com es vaja desenvolupant el projecte.

Per norma general es realitzaran reunions cada 15 dies per saber l'estat del projecte i com avança i planificar les següents tasques a realitzar al proper *sprint*.

Al ser una empresa xicoteta, no hi ha un *Scrum Manager* com a a tal. Sols el *Product Owner* que en este cas es el supervisor.

Finalment, al ser jo l'únic membre de l'equip de desenvolupament, no es faran *Daily Scrums*.

2.2 Planificació

En primer lloc recopilem totes les tasques que s'hauran de dur a terme. Algunes tasques seran molt grans per lo que es poden descompondre en altres tasques més menudes. D'aquesta manera podem crear la pila de producte inicial.

Després es valoren els punts d'història i es creen les èpiques. Finalment, a continuació es planifica el primer *sprint*. Cada planificació dels següents *sprints* es durà a terme al final de l'actual.

2.2.1 Pila del producte

Durant les dos primeres setmanes, vaig haver de fer la proposta tècnica. En aquest document vaig definir una pila de producte amb les diferents tasques que havia de dur a terme.

En les piles de producte es sol gastar històries d'usuari, però amb la metodologia elegida farem ús de tasques.

Aquesta pila de producte inicial la trobem a la figura 2.1.

Conforme van avançar els *sprints* aquesta pila va sofrir algunes modificacions, el resultat final amb el *sprint* en el que s'ha dut a terme cada tasca, el podem trobar en la Taula 2.2.

2.3 Estimació de recursos y costos del projecte

2.3.1 Recursos

Al projecte disposem de tres tipus de recursos: humans, software i hardware.

Humans

Per una part, per als recursos humans, en aquest projecte he sigut l'únic que ha treballat en ell, pel que s'ha fet ús d'un sol programador.

Software

En quant al *software*, s'ha fet ús de varies tecnologies anomenades en la secció 1.3.2 Tecnologies.

Hardware

Pel que fa al *hardware*, inicialment vaig fer ús d'un ordinador a l'oficina de l'empresa. Però donades les circumstancies vaig començar a teletreballar, les especificacions del meu ordinador són les següents:

- Processador: Intel(R) Core(TM) i5-3470 CPU 3.20Ghz 3.60Ghz
- RAM: 8,00 GB
- Disc dur: 1TB A més, el monitor marca *BenQ*, teclat i ratolí de *logitech* i auriculars *Hyperx*.

2.3.2 Costos

Humans

El sou mitjà d'un programador júnior segons *indeed* [2] és de 18.779€ anuals, per tant:

18779 / 12 = 1564,92 aproximadament 1565€ mensuals

Si suposem que es treballa 8 hores al dia i 22 dies al mes.

1565 / 22 / 8 = 8,89 aproximadament 9€ cada hora treballada.

Per tant amb una duració de projecte de 300 hores els cost humà aproximat del projecte és de: **2.700€**.

Software

El *software* utilitzat és tot gratuït a banda del *PhpStorm*, que encara que esta vegada s'ha fet ús de la llicència gratuïta gràcies a pertànyer a l'estudiantat de l'UJI, té un cost de **184€** [4] per a les empreses.

Hardware

En quant al *hardware*, el meu ordinador té 8 anys, per tant comprar les parts i accessoris a dia de hui seria molt mes barat, el preu adia d'avui estaria al voltant de **320€**.

Per tant sumant els diferents costos, el projecte tindria un cost al voltant de **3.204€** .

2.4 Seguiment del projecte

Durant el desenvolupament del projecte, s'han fet *sprints* aproximadament cada dos setmanes. En cada *sprint* he anat fent les tasques de la pila del producte que corresponien. En alguns dels *sprints* s'ha modificat la pila del producte, ja siga per una modificació de les tasques a realitzar o per una alteració en la duració estimada d'algunes activitats. Estes modificacions són acordades junt amb el supervisor de l'empresa al finalitzar cada *sprint*. Al final de cada *sprint* s'elegeix les tasques a realitzar al següent.

2.4.1 Sprint 1

Per al primer *sprint*, com era d'esperar, vaig començar amb les tasques referents a la formació i em vaig marcar fer el curs de Laravel per a tot el *sprint*, ja que era bastant més llarg de l'esperat.

No obstant, com vaig haver de dedicar la major part del temps a la proposta tècnica, no em

va donar temps a acabar cap tasca de la pila del producte. Tan sols vaig poder iniciar el curs de *Laravel* a *Udemy*.

2.4.2 Sprint 2

En quant al segon sprint, tenia que acabar amb la formació, és a dir, finalitzar el cursos de *Laravel* i *VueJS* i fer una pràctica en cadascun d'ells.

En aquest *sprint* jo ja havia acabat la proposta tècnica, per tant em podia centrar totalment amb les tasques de la pila. A més, ja portava dos setmanes a l'empresa i em vaig aclimatar ràpidament.

Vaig poder acabar ambdós cursos i em disposava a fer una pràctica d'ells per assolir els conceptes. Però el supervisor em va proposar que fera una pràctica conjunta, és a dir, que fera en *Laravel* una *API* que es comuniqués amb la part gràfica de *VueJS*, ja que és el que hauria de fer al projecte i podria resoldre els dubtes que poguera tindre.

Vaig acabar aquesta pràctica i vaig donar per finalitzat el *sprint*.

Com a observació, dir que, la formació em va ajudar molt per a aprendre a utilitzar les tecnologies, de les quals no tenia cap coneixement. I a futur, em va estalviar molt de temps d'haver de buscar tutorials de com fer les coses.

2.4.3 Sprint 3

Per aquest *sprint*, vaig agafar les tasques d'investigació dels GDM (Gestor de Dades Mestres), creació dels casos d'ús i definició dels models.

Una vegada vaig investigar i recollir informació sobre els GDM, vaig tindre una reunió amb el supervisor i l'informàtic que va canviar bastant la pila del producte. Inicialment la reunió tan sols era per a definir els casos d'ús del projecte i de com volien que fora el prototipus. Però durant la reunió vam estar parlant sobre alguns casos d'ús i vam decidir fer les següents modificacions.

Inicialment l'única part que es generaria automàticament seria el *frontend*. És a dir, aquesta part sols hauria d'agafar la informació de la *API* i generar les corresponents pàgines, independentment de les dades de la base de dades.

Però el supervisor va comentar que si per exemple una empresa gestione 50 recursos, la tasca de crear una taula en la base de dades, un model i un controlador per a cadascun dels recursos, era molt pesada.

Per tant, es va decidir que no es faria ús de models, que el controlador fora genèric, és a dir, tots els recurs farien ús del mateix controlador, i que les taules es crearien automàticament a partir d'una funció que llegira un *JSON Schema*.

Així que la pila del producte va sofrir les següents modificacions:

Primer de tot, ja no havia de definir els casos d'ús, en canvi, vam acordar que millor fera una definició dels requisits amb la informació de la reunió.

En segon lloc, l'aplicació ja no faria ús de models i la implementació de la base de dades, passava a ser la implementació d'una funció que creara taules a partir de la lectura d'un *JSON Schema*.

Finalment al no haver models, s'hauria d'implementar un controlador genèric del que farien ús tots els recursos de la base de dades. (Cada *JSON Schema* és una taula i cada taula un recurs).

A més, vam estimar la durada de les tasques que quedaven per fer:

- Creació de la funció per a la base de dades: 1 setmana
- Desenvolupament de la *API*: 2 setmanes
- Desenvolupament del *frontend*: 2 setmanes

Una vegada pactats tots els canvis, vaig acabar el *sprint*, en el qual em va donar temps a acabar la investigació sobre els GDM, i la definició dels requisits.

2.4.4 Sprint 4

En aquest *sprint* vaig elegir la creació de la funció per a la base de dades i el desenvolupament de la *API*. Encara que aquesta última no em donaria temps a acabar-la segons l'estimat en el *sprint* anterior.

Vaig començar amb la funció i el vaig poder acabar en el temps estimat. Així que tenia 1 setmana per avançar la segona tasca assignada.

Em vaig posar a la feina i el que estava inicialment estimat en dos setmanes, ho vaig acabar en menys d'una setmana.

Així que en aquest *sprint* vaig acabar les dos tasques assignades i amb això tota la part del *backend* del projecte, a falta de corregir errors que aniran sorgint i afegir funcionalitat necessària per al *frontend*.

Finalment vaig tindre un reunió amb el supervisor i l'informàtic per definir els requisits de la part del *frontend* i vaig fer una tasca de la pila que inicialment no estava programada per aquest *sprint*, la creació dels *mockups*.

2.4.5 Sprint 5

Per aquest *sprint*, vaig assignar totes les tasques restants, que eren la implementació de la lectura de *JSON schema*, la generació de la vista per mostrar les dades, i gestionar les dades del *JSON*.

Primer de tot vaig crear el *layout* principal de l'aplicació, després vaig crear les dos pàgines que havia de tindre i finalment vaig implementar aquestes pàgines. La primera pàgina es tractava d'una taula on es mostraven les instàncies del recurs seleccionat, a més, aquesta taula tenia tres accions, crear una nova instància, modificar-la i eliminar-la. Les acciones de creació i modificació portaven a la segona pàgina, un formulari creat automàticament a partir de la lectura del *JSON Schema* del recurs seleccionat.

Durant aquest *sprint* vaig tindre diversos problemes, entre ells, les peticions a la base de dades me donaven un error de *CORS (Cross Origin Resource Sharing)*, que vaig poder solucionar creant un *middleware* al *backend*. I l'altre gran problema va ser l'actualització del paquet que gastava per generar el formulari, que em va fer canviar la implementació d'aquest.

Tot i així, vaig poder acabar totes les tasques, i donar per finalitzada la pila de producte.

Finalment vaig tindre una reunió, en la que vaig fer una demostració del prototipus al supervisor i al informàtic de l'empresa. Van quedar contents amb el resultat i per al següent i últim *sprint* em van demanar, afegir funcionalitat que cregués útil, solucionar alguns errors, reestructurar millor el codi i estilitzar el *frontend*.

2.4.6 Sprint 6

En aquest últim *sprint*, he fet una nova tasca de millora del codi i una nova de crear un filtre avançat.

La millora del codi inclou la reestructuració tant del *frontend* com del *backend*, l'eliminació de codi repetit, eliminació de codi que s'ha quedat inutilitzat i la millora en eficiència.

En quant al filtre, hem van dir que afegira funcionalitat útil, pel que he afegit un filtre per a obtindre les instàncies a la base de dades, en aquest filtre es pot comparar una columna amb un valor i es pot ordenar ascendent o descendent per columna.

A més, fora ja de la pila de producte, vaig crear un manual d'usuari dins del prototipus. En aquest manual es mostra informació de totes les pàgines disponibles amb una descripció de que és cada component i les diferents accions que tenen els botons.

Amb aquest *sprint*, donem per acabat el projecte.

Tasques	Èpiques	Punts d'història
Realitzar un curs en Laravel a la plataforma udemy	Formació	3
Realitzar un curs en VueJS a la plataforma udemy	Formació	3
Realitzar una pràctica que assentar el conceptes apresos al curs	Formació	5
Investigació dels GDM	Investigació	2
Creació dels casos d'ús	Back-end	1
Definició dels models	Back-end	3
Implementació de la base de dades	Back-end	5
Implementació de la API Rest	Back-end	13
Creació de mockups	Front-end	2
Implementar la lectura de JSON schema	Front-end	8
Generació de la vista per mostrar les dades	Front-end	5
Gestionar les dades del json	Front-end	5

Taula 2.1: Pila de producte inicial

Tasques	Èpiques	Punts d'història	Sprint
Realitzar un curs en Laravel a la plataforma udemy	Formació	3	1, 2
Realitzar un curs en VueJS a la plataforma udemy	Formació	3	2
Realitzar una pràctica que assentar el conceptes apresos al curs	Formació	5	2
Investigació dels GDM	Investigació	2	3
Definició de requisits del backend	Back-end	1	3
Implementació de algoritme per a la creació de la BBDD	Back-end	8	4
Implementació de la API Rest	Back-end	8	4
Definició de requisits del frontend	Front-end	1	4
Creació de mockups	Front-end	2	4
Implementar la lectura de JSON schema	Front-end	8	5
Generació de la vista per mostrar les dades	Front-end	5	5
Gestionar les dades del json	Front-end	5	5
Implementació d'un filtre avançat	Front-end	3	6
Millora del codi	Full Stack	5	6

Taula 2.2: Pila de producte final

Capítol 3

Anàlisi i disseny del sistema

Aquest capítol del document, és el que menys informació conté per les característiques d'aquest projecte i la manera en la que s'ha treballat a l'empresa. Ja que per a començar, no existeix un disseny de la base de dades, simplement emmagatzemen els recursos en una taula cadascun i no tenen cap tipus de relació entre elles. Segons els clients hi haurà unes taules o altres. I per la part de l'anàlisi del sistema, no s'ha realitzat el diagrama de casos, en canvi es va realitzar una reunió en la que vaig recollir els requisits que ha de complir el prototipus.

Per tot açò, aquesta part del document estarà més poc detallada i en canvi, la part d'implementació s'explicarà molt a fons i en detall, ja que considere que ha sigut la part principal de l'estància en pràctiques.

3.1 Anàlisi del sistema

Per als requisits del sistema, es van dur a terme dos reunions, una primera reunió per definir els requisits del *backend* i una segona quan vaig acabar d'implementar el *backend*, per definir els requisits del *frontend*.

3.1.1 Requisits del *backend*

La funcionalitat principal que ha de complir el *backend* és la de crear i modificar la base de dades, segons les peticions del client i la definició dels *JSON Schema* dels seus recursos.

Per a la part del client es crearà la *API* i per a la part dels *JSON Schema*, hi haurà que tindre un comandament que al executar-ho, cride a una funció que llegisca aquests arxius i modifique la base de dades en conseqüència. Aquesta funcionalitat més el que s'ha acordat durant la reunió es pot desglossar en els següents requisits:

- Un comandament que actualitze la base de dades quan s'executa.

- El comandament ha de cridar a una funció que serà l'encarregada de modificar la base de dades.
- Una funció que llegisca els *JSON Schema* emmagatzemats i faja les respectives modificacions a la base de dades.
- La funció ha de poder crear les taules que no han sigut creades fins el moment, i modificar les que ja estaven creades.
- La modificació ha d'incloure eliminació de columna, afegir una columna i modificar el tipus de la columna.
- Que a la base de dades es pugui emmagatzemar camps de tipus: *integer*, *float*, *string*, *text*, *boolean*, *date*, *date-time*.
- Un controlador genèric del que faran ús totes les taules.
- El controlador ha de definir els mètodes pertinents per a dur a terme les peticions *CRUD* del client.
- Que el mètode d'enviar les instàncies d'una taula sols envie el nombre d'instàncies indicat.
- Les respectives rutes per fer tractar amb les peticions des del client.

3.1.2 Requisits del *frontend*

En quant als requisits del *frontend*, la funcionalitat principal que ha de complir és la de poder obtindre les dades emmagatzemades a la base de dades i realitzar sobre elles les accions CRUD.

Per a la part de llegir, es mostraran les dades en una taula i per a crear i actualitzar una nova instància es realitzaran formularis. Aquesta funcionalitat més el que s'ha nombrat durant la reunió es pot desglossar en els següents requisits:

- Una barra lateral amb un llistat de tots el models actuals emmagatzemats a la base de dades.
- La comunicació amb el *backend* mitjançant peticions.
- Una taula on es mostren les instàncies de cada recurs.
- La paginació d'aquesta taula.
- Que cada vegada que es canvie la pàgina es faja la corresponent petició per obtindre les noves dades.
- Que les instàncies es puguen crear, modificar i eliminar.
- La creació automàtica de formularis a partir d'un JSON Schema per a la creació i modificació d'instàncies.
- Un filtre avançat que faja la filtració a la part del *backend* i no al *frontend*.

3.2 Disseny de l'arquitectura del sistema

L'arquitectura del sistema tracta d'una arquitectura de tres capes: client, servidor i base dades. Existeix una comunicació entre el servidor i la base dades i una comunicació entre el client i el servidor. És a dir, el servidor fa d'intermediari entre el *frontend* i la base dades. Aquesta estructura es pot observar a la figura 3.1.

Si el *frontend* vol comunicar-se amb la base de dades per modificar una instància d'una columna, primer li passarà al servidor la informació necessària i serà aquest el que duga a terme l'acció.

A més el servidor també pot modificar la base de dades, sense cap petició del client, quan s'executa el comandament que llegeix els arxius *JSON Schema*.

El *frontend* farà les peticions *GET, PUT, POST* i *DELETE* mitjançant el paquet *axios*, anomenat a la secció 1.3.2. A la figura 4.40 podem veure un exemple d'aquestes peticions. La informació que rep el *frontend* és un *JSON*, prèviament explicat a la secció d'informació addicional.

El *backend* rep les peticions amb les rutes definides i envia la informació al controlador assignat a la ruta, el qual tindrà un mètode definit que serà el que es comunique amb la base dades mitjançant els *facades Schema* i *DB*. A la figura 4.26 es pot veure un exemple de la definició de rutes i a la figura 4.21 un mètode del controlador el qual es comunica amb la base de dades per eliminar una instància.

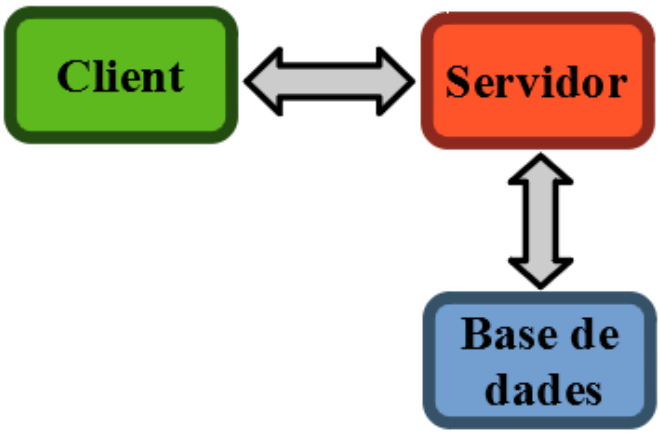

Figura 3.1: Arquitectura del sistema

3.3 Disseny de la interfície

En aquesta secció es mostraran els *mockups* dissenyats per a la interfície d'usuari, explicant un poc el pensament inicial i després el resultat final i com s'ha arribat fins aquest.

3.3.1 Esbossos de les interfícies

Vaig dissenyar l'esbós de tres pàgines, la d'inici, la de la taula *CRUD* i la dels formularis. Totes elles tenen en comú el *layout*, principal de l'aplicació. Aquest compta amb, una barra lateral amb un llistat dels models, una barra de navegació, inicialment buida i un peu de pàgina. El color elegit per a l'aplicació inicialment va ser d'un blau.

En la següent figura 3.2 podem observar la pàgina d'inici, la qual en un principi, anava a tindre un *select* on estarien totes les taules de la base de dades i un botó per accedir a la taula *CRUD* del model seleccionat.

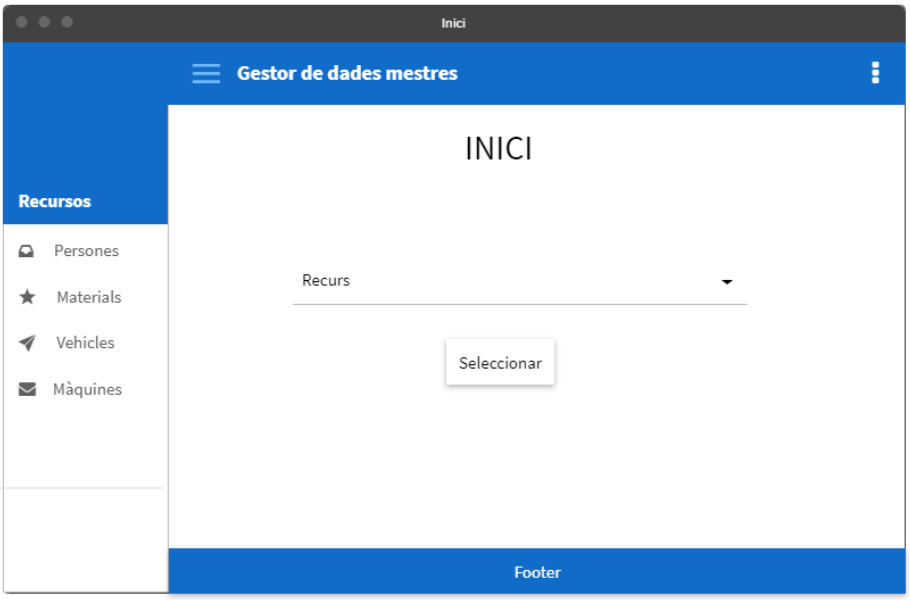

Figura 3.2: *Mockup* de la pàgina d'inici

Després, per a la pàgina de la taula *CRUD*, vaig dissenyar una taula on es mostren les columnes del recurs i totes les instàncies. Al costat de cada fila hi hauria un botó per actualitzar o eliminar la respectiva fila. I baix de la taula el botó per crear una nova instància. A la figura 3.3 podem trobar l'esbós.

Finalment vaig dissenyar la pàgina del formulari, en este cas de creació d'una instància. Aquesta pàgina conté un formulari amb tots els camps accessibles per al client del recurs seleccionat.

Aquest mateix formulari valdria per a la modificació d'una instància, l'únic que canviaria seria el nom del botó i que els camps estarien inicialitzats amb el seu valor actual.

A la figura 3.4 podem trobar l'esbós del formulari.

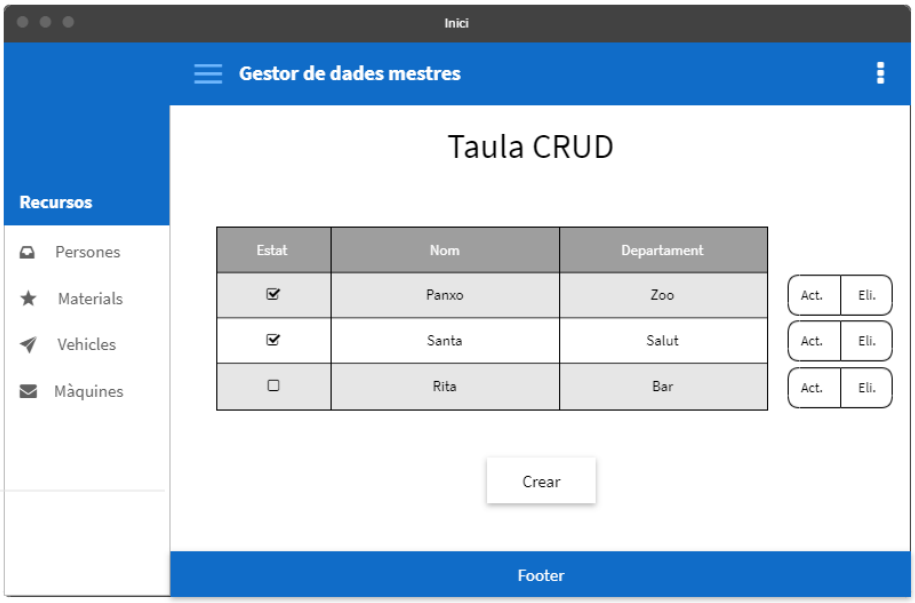

Figura 3.3: *Mockup* de la pàgina de la taula *CRUD*

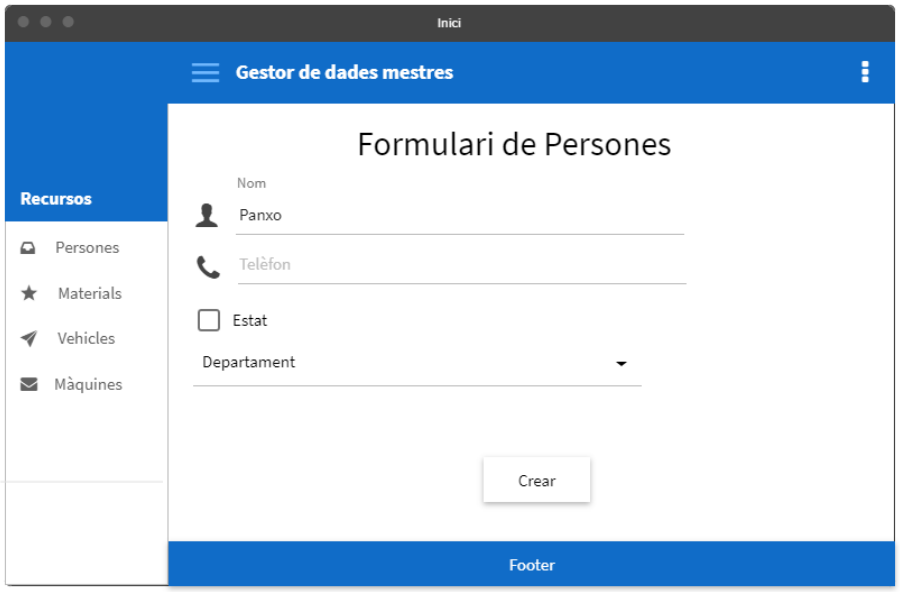

Figura 3.4: *Mockup* de la pàgina del formulari de creació *CRUD*

3.3.2 Interfícies finals

La versió final de les interfícies no és molt diferent als esbossos mostrats, però existeixen alguns canvis notables els quals vaig a comentar a continuació.

Per a poder veure la versió final de les interfícies es pot anar a la secció de funcionament del *frontend*. Allí es mostre el resultat final.

Els canvis que han sofrit respecte als *mockups* han sigut els següents.

En primer lloc, el canvi de color de blau a morat. Aquest canvi el vaig realitzar ja que al començar amb la implementació em pareixia que l'aplicació no tenia personalitat, semblava una aplicació genèrica. Al canviar el color, amb aquest simple canvi, ja va guanyar personalitat. El color va ser elegit per ser el color del logotipus de l'empresa *SiGO*.

Un altre canvi notable va ser el de la pàgina d'inici. El *select* per a seleccionar el recurs no era una solució molt òptima, per tant vaig implementar una taula amb tots els models de la base de dades. A banda, si és un client amb molts recursos que gestionar, es poden ordenar alfabèticament i es pot cercar per nom.

Pel que fa a la interfície de la taula *CRUD* i la del formulari, en la taula el canvi ha sigut afegir uns filtres, un sistema de paginació i un camp per seleccionar quants elements mostrar per pàgina. I en el formulari al ser generat automàticament pel paquet, els canvis sofrits han estat fora del meu abast.

Els altres canvis, o bé són mínims o són insignificants i no val la pena nombrar-los. Si de cas, el afegir una nova pàgina que és com un manual d'usuari de l'aplicació. I afegir el corresponent botó per accedir en la barra de navegació.

Capítol 4

Implementació y proves

4.1 Detalls d'implementació

En quant a la implementació, el prototipus està format per dos projectes. Un primer projecte referent a la part del *backend*, en el que faig ús de *Laravel*, que com hem anomenat abans es tracta d'un *framework* de *php*. I el segon projecte referent a la part de *frontend*, en el qual es fa ús de *VueJS* amb el paquet de *vuetify*, com també he explicat abans.

La part del *frontend* es comunica amb el *backend* mitjançant la *API* implementada, i el *backend* es comunica amb la base de dades.

A continuació es detallarà la implementació d'aquest dos projectes.

4.1.1 Backend

Com hem anomenat en repetides ocasions, per al *backend*, s'ha elegit *Laravel*, aquesta decisió, va ser una recomanació de l'empresa. Ells feien ús d'aquest *framework* i sabien de tots els seues avantatges. Tot i això, em van deixar decidir quina ferramenta utilitzar, haguera pogut utilitzar una que em resultara més familiar, ja que desconeixia totalment *Laravel*, però ja que em van dir que m'ajudarien en qualsevol dubte que poguera tenir, vaig decidir aprendre una nova tecnologia.

Estructura

Aquest *framework* et dona una estructura inicial bastant completa, a la qual he hagut d'afegir poques carpetes, i també no he fet ús de moltes característiques que et donen ja pràcticament implementades.

La estructura final del projecte és la mostrada a la figura 4.1.

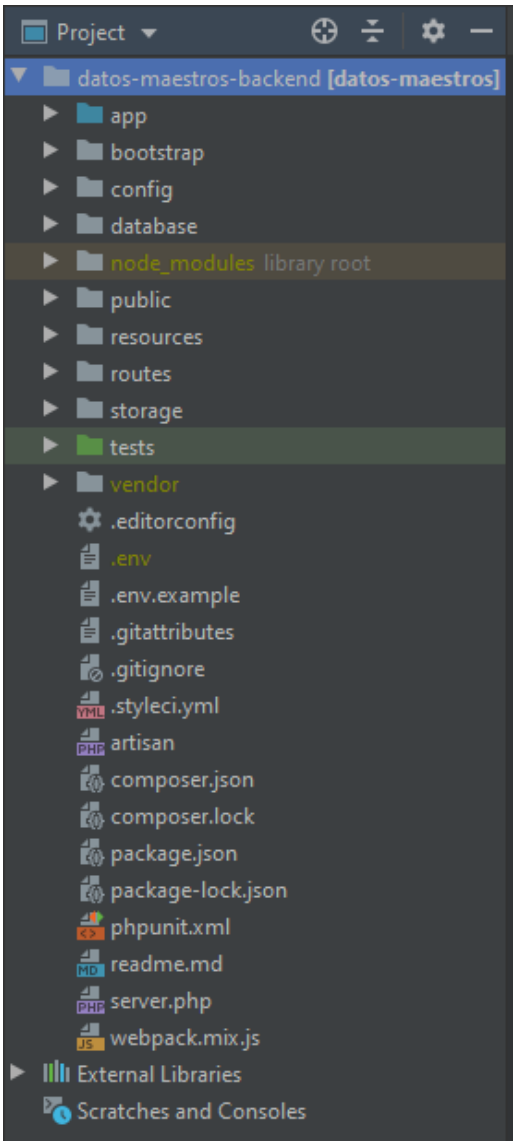

Figura 4.1: Estructura del *backend*.

Ara vaig a procedir a explicar la utilitat i funcionalitat de cada carpeta, entrant en detall en aquelles en les que he implementat part del codi.

En primer lloc, trobem la carpeta *app*, es tracta de la carpeta principal del projecte, en ella encontrem la major part del codi del projecte. Primer explicaré les altres carpetes, finalment entraré en detall en aquesta.

En segon lloc, el directori *bootstrap*, en aquest no entrarem en detall, ja que es tracta d'un directori de configuració i optimització del *framework*, el qual no es sol modificar.

Seguidament, la carpeta *config*, que com el seu nom indica serveix per configurar l'aplicació. A diferència de l'anterior directori, aquest si es sol modificar. Podem trobar en ell arxius diversos per configurar, des de la memòria cau, fins al sistema d'emmagatzematge i la base de dades.

Després ens trobem amb el directori *database*, com es pot deduir, aquest tracta sobre la base de dades. Aquesta 4.2 és la seua estructura.

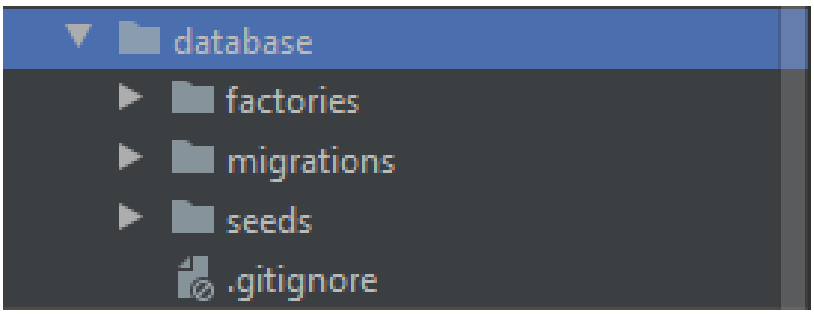

Figura 4.2: Estructura del directori*database*.

En la carpeta de *factories*, es du a terme la creació massiva d'instàncies. Açò és molt útil per a l'etapa de desenvolupament, ja que ens estalvia molt de temps de crear manualment les instàncies per provar el bon funcionament de la base de dades. Per contra, en l'etapa de producció ens deixa ja d'importar.

Després es troba la carpeta de *migrations*, aquesta és la principal d'aquest directori, en ella es creen i modifiquen les taules a la base de dades. Cada arxiu d'aquesta carpeta sol fer referència a la creació d'una taula. *Laravel* inclou el següent comandament *php artisan migrate*, aquest el que fa és executar els arxius que es troben dins de *migrations*, i en conseqüència, crea les respectives taules. Ara bé, si ja estava creada no l'executa l'arxiu, per tant si es vol modificar una taula creada s'hauria de fer un *php artisan migrate:refresh*. No obstant, aquest últim comandament et borra totes les instàncies de la base de dades, per tant si no volem que passe això, es por crear un arxiu en *migrations*, que no cree una taula, si no que modifique una existent. D'aquesta manera, no eliminem les instàncies actuals i modifiquem una taula.

Finalment, el directori *seeds* serveix per a inserir instàncies per defecte a la base de dades, per a no haver d'estar creant-les cada vegada que es borra una taula. Açò al igual que *factories* ens serveix de molt durant el desenvolupament. De fet aquesta carpeta pot fer ús de les instàncies de *factories* per a omplir les taules.

Sorprenentment, en aquest projecte, sols s'ha fet ús de la carpeta *seeds*, ja que les taules seran diferents per a cada client. Per tant la creació de les taula a la base de dades, es durà a terme en una funció d'una classe, el qual explicaré quan entrem en detall de la carpeta *app*.

Una vegada, explicat aquesta carpeta, tornem a l'estructura del directori principal. Toca el torn de *public*, el qual no anem a entrar en detall perquè es tracta d'una carpeta on no em interactuat per a res. Ací es troben el recursos estàtics de la nostra aplicació, com ara arxius *css*, *js*, imatges i fonts.

La següent carpeta és *resources*, aquesta es sol utilitzar quan la part del frontend es troba al projecte, per a dissenyar les vistes, així que com el *frontend* l'he fet en un altre projecte, no he modificat per a res aquesta carpeta. Encara que si per lo que fora, hagués d'enviar algun correu, ací es pot dissenyar el contingut del correu. Dins d'aquesta carpeta també es posen els arxius d'idioma per traduir l'aplicació, però de nou, això és més quan es treballa amb el *frontend*.

Seguidament trobem el directori de *routes*. En aquest directori es defineixen les diferents rutes de l'aplicació. Aquests 4.3 són els arxius que *Laravel* ens dona per defecte:

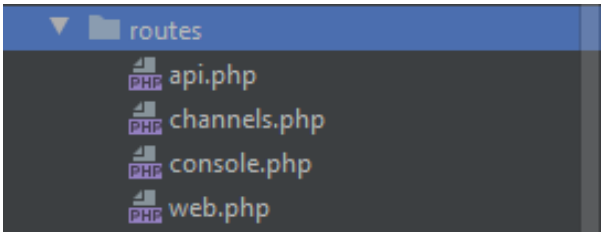

Figura 4.3: Estructura del directori *routes*.

Al tractar-se aquest projecte d'una *API*, totes aquelles rutes de les que farà ús el *frontend* per comunicar-se amb aquest projecte seran definides dins de l'arxiu *api.php*. Per exemple, si el nostre projecte estiguera tot ací junt, es definirien les rutes en *web.php*

El següent directori en la llista és *storage*. Com el seu nom indica, és on es guarden els diferents arxius que fa ús l'aplicació i els que rep o crea l'aplicació. Aquesta 4.4 és l'estructura:

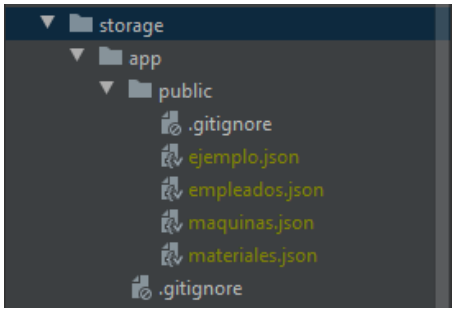

Figura 4.4: Estructura del directori *storage*.

En aquest projecte, en */storage/app/public/* és on emmagatzema els *JSON Schema* referents als recursos del client, aquest arxius són llegits tant per l'algoritme per crear les taules de la

base de dades, tant pel *frontend* per generar els formularis respectius al recurs.

Finalment trobem els directoris *tests* i *vendor*. Per una part en la carpets tests, trobaríem els tests de l'aplicació, però donada la llagaria del projecte, no entraven dins de l'abast, per lo que no està buida. Per altra part en *vendor*, trobem les dependències de *Composer*, que es tracta d'un instal·lador de paquets com *npm*. Les dependències de *npm* les trobem a la carpeta *node-modules* la qual no he anomenat per creure innecessari.

Una vegada explicats els anteriors directoris, podem entrar de detall amb el directori principal, *app*. Que té la estructura mostrada a la figura 4.5.

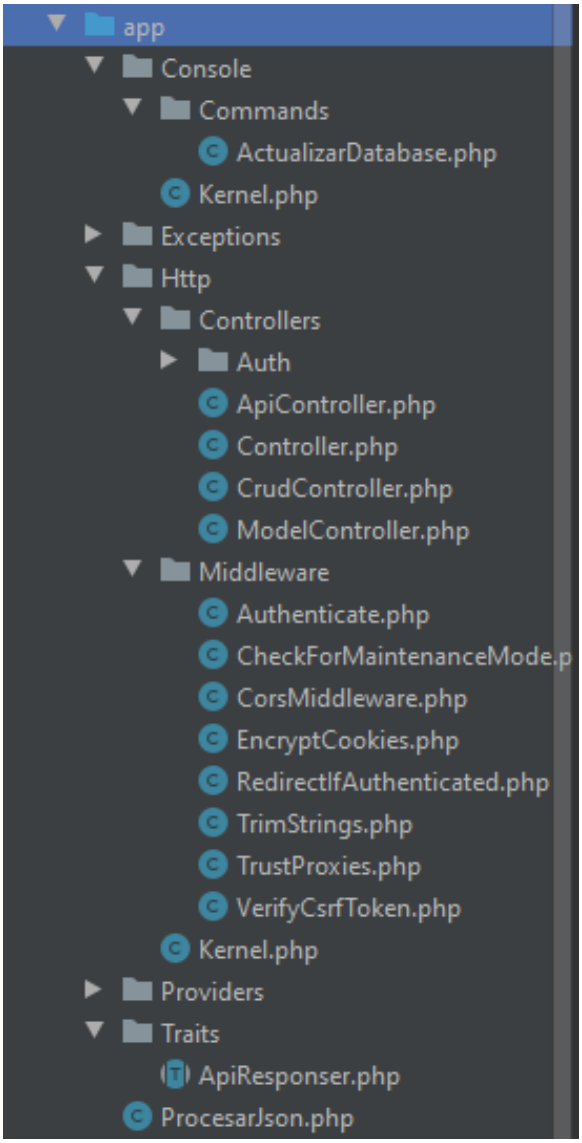

Figura 4.5: Estructura del directori *app*.

Abans de començar a desglossar aquest directori, vull fer un incís de que a diferència dels altres directoris que les carpetes ja venen creades totes per defecte en aquest sols es creen unes poques per defecte i les altres sols es creen si s'utilitzen, per lo tant aquest directori podria

incloure molta més funcionalitat de la que inclou actualment.

En primer lloc, la carpeta *console*, aquesta carpeta apareix quan es crea una nou comandament. Per aquesta aplicació com he explicat als requisits, feia falta un comandament personalitzat que al introduir-ho en la consola actualitzarà la base de dades. El funcionament l'explicaré més endavant a l'apartat de funcionament. A més, trobem l'arxiu *Kernel.php*, on s'han de definir tots els nous comandaments per a que es puguen fer ús a la consola.

En segon lloc, el directori *exceptions*, el qual ve creat per defecte. En ell, es poden crear excepcions personalitzades de les que farà ús l'aplicació, però jo per aquest projecte no he vist necessari crear cap.

En tercer lloc, trobem la carpeta *Http*, una de les més importants, ja que emmagatzema la major part de la lògica de tractament de les peticions. En ella es troben dos carpetes. La de *Controllers*, on estan els controladors dels que fa ús l'aplicació, tots ells creats per mi llevat els de *Auth* que venen per defecte i serveixen per implementar el sistema d'autenticació, el qual no hem implementat ja que no entrava als requisits del projecte. I la de *Middlewares*, aquests arxius són essencials per a la seguretat de la nostra aplicació, venen tots per defecte a banda de CorsMiddleware, el qual vaig haver de crear per solucionar un error quan feia les peticions des del *frontend*. Açò últim es comentarà més a fons a la secció de problemes.

Seguidament, el directori *Providers*, el qual ve per defecte i no he modificat res. En aquest directori es troben tots els serveis de proveïdors que venen amb *Laravel* i es pot afegir aquells que crees tu mateix.

Després, la carpeta *Traits*, no ve per defecte amb *Laravel*, la vaig crear per definir funcions les quals es repetien molt als controladors, per tant vaig crear aquesta classe de la qual estenen els meus controladors.

Finalment, després de passar per tots els subdirectoris, trobem un fitxer anomenat *ProcesarJson.php*, aquest arxiu es tracta d'una classe on es troba la funció que crea les taules de la base de dades. Ací en un principi anaven tots els models referents a cada taula, però com es va quedar fer-ho tot automàtic, vam decidir prescindir de models.

Funcionament

En aquest apartat explicaré amb detall com funciona la part del *backend* de l'aplicació.

Per a començar explicaré la funció de creació de les taules a la base de dades a partir dels arxius *JSON Schema*. Està part l'explicaré amb detall, perquè em pareix una de les parts més interessants del projecte.

Primer de tot, la funció agafa tots els arxius emmagatzemats al directori establert per guardar tots els *JSON Schema*, els fica dins d'un bucle *for*, on es comprova el títol del recurs, si no hi ha cap taula a la base de dades amb eixe títol, es crida al mètode de crear taula, si en canvi ja estava creada, es crida al mètode d'actualitzar taula. Finalment quan passen tots els arxius, dins d'una variable estarà guardat un *boolean* amb un valor depenent de si la base de

dades ha sofrit alguna modificació.

Aquest 4.6 és el mètode:

```php
class ProcesarJson
{
    //torna un bolean, true si se ha modificat algo, false si no se ha modicat res
    public static function procesarFicheros(){
        error_log( message: 'Procesando ficheros...');

        $files = File::files( directory: 'storage/app/public'); //obté un array amb els fitxers
        $tables = DB::select( query: 'SHOW TABLES');
        $tables = array_map( callback: 'current', $tables); // obté un array de les taules de la BBDD
        $modificada= false;

        foreach ($files as $file){
            $pathname = $file->getPathname();
            $jsonString = file_get_contents(base_path($pathname));
            $data = json_decode($jsonString,  assoc: true); //obté les dades del equema
            $keys = array_keys($data['properties']); //array amb les columnes de la taula
            $length = count($keys);
            if(in_array($data['title'], $tables)){ //check si ja s'ha creat

                $mod = (new ProcesarJson)->actualizartabla($data, $keys, $length);
                if ($modificada==false){
                    $modificada=$mod;
                }
            }
            else{
                $modificada = true;
                (new ProcesarJson)->crearTabla($data, $keys, $length);
            }
        }
        return $modificada;
    }
```

Figura 4.6: Mètode per processar fitxers.

En el cas de que la taula a la que fa referència l'arxiu, no estava creada, accedeix al mètode de crear taula. Normalment les taules es creen en la carpeta de *migrations*, com bé hem explicat abans. Es crearia com es mostra en la figura 4.7.

Però en aquest projecte, al voler automatitzar la creació de la base de dades, per a no haver de crear d'un en un cada recurs de cada client, no fem la creació d'aquesta manera. D'una manera pareguda, comptem amb una funció a la que li passarem les dades de l'esquema que li facen falta i anirà comprovant atribut per atribut de quin tipus és, si té una longitud màxima, si pot ser *null*, etc. Una part d'aquesta creació es mostra en la figura 4.8.

```php
Schema::create( table: 'users', function (Blueprint $table) {
    $table->bigIncrements( column: 'id');
    $table->string( column: 'name');
    $table->string( column: 'email')->unique();
    $table->timestamp( column: 'email_verified_at')->nullable();
    $table->string( column: 'password');
    $table->rememberToken();
    $table->timestamps();
});
```

Figura 4.7: Creació d'una tabla en una migració

```php
public function crearTabla($data, $keys, $length){
    error_log( message: 'Creando tabla ' . $data['title'] . '...');
    $required = $data['required'];
    Schema::create($data['title'], function (Blueprint $table) use ($required, $data, $keys, $length) {
        $table->bigIncrements( column: 'id');
        for($i = 1; $i < $length; $i++){
            $format= null;
            $maxLength = null;
            $name = $keys[$i]; //nombre de la columna
            $type = $data['properties'][$keys[$i]]['type']; //el tipo del que es la columna
            if (array_key_exists( key: 'format',$data['properties'][$keys[$i]] )){
                $format = $data['properties'][$keys[$i]]['format'];
            }
            elseif(array_key_exists( key: 'maxLength',$data['properties'][$keys[$i]] )){
                $maxLength = $data['properties'][$keys[$i]]['maxLength'];
            }

            if($type == 'string'){
                if ($format!=null){
                    if($format=='date'){
                        (in_array($name, $required) ? $table->date($name) : $table->date($name)->nullable() );
```

Figura 4.8: Creació d'una tabla a partir d'un fitxer

El *JSON Schema* compta amb una *array* on es guarda informació dels atributs, per tant es recorre aquest vector i en cada element del *array* es mira quines propietats té, per fer la inserció corresponent. La part del *if-elseif* serà tractada en detall a la secció de problemes. Després de les comprovacions necessàries s'arriba a l'última línia de la figura, que en este cas seria una inserció d'un atribut del tipus data. La funció segueix d'una manera similar per a la resta de tipus d'atributs com podem veure en la següent figura 4.9.

```
        if($type == 'string'){
            if ($format!=null){
                if($format=='date'){
                    (in_array($name, $required) ? $table->date($name) : $table->date($name)->nullable() );
                }
                elseif ($format=='date-time'){
                    (in_array($name, $required) ? $table->dateTime($name) : $table->dateTime($name)->nullable() );
                }
                elseif($format=='text'){
                    (in_array($name, $required) ? $table->text($name) : $table->text($name)->nullable() );
                }
            }
            elseif ($maxLength != null){
                (in_array($name, $required) ? $table->string($name, $maxLength) : $table->string($name, $maxLength)->nullable() );
            }

            else{
                (in_array($name, $required) ? $table->string($name) : $table->string($name)->nullable() );
            }
        }
        elseif ($type == 'integer'){
            (in_array($name, $required) ? $table->integer($name) : $table->integer($name)->nullable() );
        }
        elseif ($type == 'number'){
            (in_array($name, $required) ? $table->float($name) : $table->float($name)->nullable() );
        }
        elseif (($type)=='boolean'){
            (in_array($name, $required) ? $table->boolean($name) : $table->boolean($name)->nullable() );
        }
    }
    $table->timestamps();
});
```

Figura 4.9: Comprovació i inserció dels atributs

En aquesta última figura donem per acabat el mètode de crear la taula a partir del fitxer de dades, ara bé, si el client volguera fer alguna modificació a la taula, seria convenient que també es poguera fer automàticament. Per a complir aquest desig, quan processem els fitxers, si la taula ja està creada, se crida al mètode d'actualitzar taula.

En aquest mètode és tractaran els canvis referents a l'eliminació d'una columna, afegir una nova columna i modificar el tipus d'una columna, aquesta última depenent a quin tipus canvie serà necessari buidar la taula.

Tot açò es durà terme mitjançant la comparació entre les columnes definides al esquema i les columnes actuals de la taula en la base de dades.

Primer de tot comprova si hi ha alguna columna que està en la taula de la base de dades, però ja no està al *JSON Schema* referent a la taula. Si és el cas, es borra la columna. Podem observar-ho a la figura 4.10.

```php
private function actualizarTabla($data, $newKeys, $newLength){
    error_log( message: 'Comprobando diferencias en la tabla ' . $data['title'] . '...');
    $modificada=false;
    $oldKeys = Schema::getColumnListing($data['title']);
    $oldLength = count($oldKeys);
    $required = $data['required'];

    for($j=1; $j <$oldLength-2; $j++){
        if(!in_array($oldKeys[$j], $newKeys)){ //eliminar columna
            error_log( message: 'Eliminando la columna ' . $oldKeys[$j] . '...');
            Schema::table($data['title'], function (Blueprint $table) use ($j, $oldKeys) {
                $table->dropColumn($oldKeys[$j]);
            });
            $oldKeys--;
            $modificada=true;
        }
    }
```

Figura 4.10: Comprovació i eliminació de columnes

Després, d'una manera similar a la creació de la taula, es recorreran totes les columnes definides a l'esquema. Primer es comprovarà si la columna està present a la taula de la base de dades, si no és el cas, s'afegirà darrere de l'anterior columna comprovada. Podem veure com en la figura 4.12.

Finalment, es farà d'una forma pareguda per la modificació del tipus de la columna actual. Per a poder dur a terme la modificació, primer es compara el tipus de la columna en l'esquema amb el tipus de la columna en la taula de la base de dades. De nou en aquesta part hi ha hagut problemes que tractarem a la corresponent secció. La modificació es pot veure com es fa en la figura 4.13, encara que abans s'han fet varies comprovacions necessàries pel que explicaré als problemes.

Per a poder posar en marxa aquesta funció s'ha creat un comandament personalitzat que al introduir-ho per la consola executa aquesta funció. La classe del comandament compta amb un mètode el qual s'invoca quan es fica el comandament a la consola. El mètode el trobem a la figura 4.11.

```php
public function handle()
{
    $procesar = ProcesarJson::procesarFicheros();

    $this->info( string: $procesar ? 'Base de datos actualizada correctamente.' : 'La base de datos ya estaba actualizada');
}
```

Figura 4.11: Mètode del comandament

La funció com havia dit abans, retorna un valor *boolean*, per tant, si és *true* es mostrarà per la consola que la base de dades ha estat actualitzada correctament i en canvi, si és *false* es mostrarà que la base de dades ja estava actualitzada.

```
if(!in_array($newKeys[$j], $oldKeys)){ //añadir columna nueva
    error_log( message: 'Añadiendo la columna ' . $newKeys[$j] . '...');
    Schema::table($data['title'], function (Blueprint $table) use ($required, $format, $j, $oldKeys, $name, $newType, $maxLength) {
        if($newType == 'string'){
            if ($format != null){
                if($format=='date'){//date
                    (in_array($name, $required) ? $table->date($name)->after($oldKeys[$j-1]) : $table->date($name)->nullable()->after($oldKeys[$j-1]) );
                }
                elseif ($format=='date-time'){
                    (in_array($name, $required) ? $table->dateTime($name)->after($oldKeys[$j-1]) : $table->dateTime($name)->nullable()->after($oldKeys[$j-1]) );
                }
                elseif ($format=='text'){
                    (in_array($name, $required) ? $table->text($name)->after($oldKeys[$j-1]) : $table->text($name)->nullable()->after($oldKeys[$j-1]) );
                }
            }
            elseif ($maxLength!=null){
                (in_array($name, $required) ? $table->string($name)->after($oldKeys[$j-1]) : $table->string($name)->nullable()->after($oldKeys[$j-1]) );
            }
            else{
                (in_array($name, $required) ? $table->string($name)->after($oldKeys[$j-1]) : $table->string($name)->nullable()->after($oldKeys[$j-1]) );
            }
        }
        elseif ($newType == 'integer'){
            (in_array($name, $required) ? $table->integer($name)->after($oldKeys[$j-1]) : $table->integer($name)->nullable()->after($oldKeys[$j-1]) );
        }
        elseif ($newType == 'number'){
            (in_array($name, $required) ? $table->float($name)->after($oldKeys[$j-1]) : $table->float($name)->nullable()->after($oldKeys[$j-1]) );
        }
        elseif (($newType)=='boolean'){
            (in_array($name, $required) ? $table->boolean($name)->after($oldKeys[$j-1]) : $table->boolean($name)->nullable()->after($oldKeys[$j-1]) );
        }
    });
```

Figura 4.12: Afegir una nova columna

```
Schema::table($data['title'], function (Blueprint $table) use ($maxLength, $format, $name, $newType) {
    if($newType == 'string'){
        if($format=='date'){//date
            $table->date($name)->charset( charset: null)->change();
        }
        elseif ($format=='date-time'){//datetime
            $table->dateTime($name)->charset( charset: null)->change();
        }
        elseif ($format=='text'){//text
            $table->text($name)->charset( charset: null)->change();
        }

        else{
            if ($maxLength!=null){
                $table->string($name, $maxLength)->charset( charset: null)->change();
            }
            else{
                $table->string($name)->charset( charset: null)->change();
            }
        }
    }
    elseif ($newType == 'integer'){
        $table->integer($name)->charset( charset: null)->change();
    }
    elseif ($newType == 'number'){
        $table->float($name)->charset( charset: null)->change();
    }
    elseif (($newType)=='boolean'){
        $table->boolean($name)->charset( charset: null)->change();
    }
});
```

Figura 4.13: Modificar el tipus d'una columna

Una vegada explicat el funcionament d'aquesta part, podem passar al funcionament de l'*API*. Formada principalment per l'arxiu on es troben totes les rutes, i la carpeta amb els controladors que fan ús aquestes rutes.

Com les rutes fan ús dels controladors, explicaré primer la part dels controladors.

Aquesta aplicació compta amb dos controladors principals. El *CrudController* , el qual és l'encarregat de tractar les peticions *CRUD*, i el *ModelController* que és l'encarregat de tractar aquelles peticions de dades que són necessàries per al *frontend*, fora del les *CRUD*, com per exemple enviar el *JSON Schema* d'un recurs.

Abans d'explicar els controladors, anomenar que ambdós controladors fan ús de mètodes que és repeteixen all llarg dels seus mètodes, pel que com ja havia dit a la secció d'estructura, he creat una classe amb aquests mètodes.

El primer, s'encarrega de comprovar que el model, recurs o taula passada al controlador existeix actualment a la base de dades. Es pot veure a la figura 4.14.

```php
protected  function comprobarModelo($modelo){
    $tables = DB::select( query: 'SHOW TABLES');
    $tables = array_map( callback: 'current', $tables);
    $message=[];
    if(!in_array($modelo, $tables)){
        return $message = 'La tabla ' . $modelo . ' no se encuentra en la base de datos';
    }
    else {
        return $message;
    }
}
```

Figura 4.14: Mètode per a comprovar el model

El segon, s'encarrega de que existeix una instància en la taula amb l'identificador passat com a paràmetre. Podem observar-ho a la figura 4.15

```php
protected  function comprobarId($modelo, $id){
    $indices = DB::table($modelo)->select( columns: 'id')->get();
    $indices = json_decode($indices,  assoc: true);
    $indices = array_map( callback: 'current', $indices);
    $message = [];
    if(!in_array($id, $indices)){
        return $message = 'La instancia de la tabla ' . $modelo . ' con el ID '. $id .' no se encuentra en la base de datos';
    }
    else{
        return $message;
    }
}
```

Figura 4.15: Mètode per comprovar l'identificador

I per últim el tercer s'encarrega de llegir l'esquema del recurs passat i obtindre les regles de validació dels tipus de totes les columnes. Es mostra a la figura 4.16.

```php
protected function reglasTipos($keys, $data){
    $length = count($keys);
    for ($i=1;$i<$length;$i++){
        $campo = $keys[$i];
        $type =  DB::getSchemaBuilder()->getColumnType( $data['title'], $campo);

        if ($type == 'string' || $type=='text'){
            $typeRules[$campo] = 'string';
        }
        elseif ($type == 'date' || $type == 'datetime'){
            $typeRules[$campo] = 'string';
        }
        elseif ($type == 'integer'){
            $typeRules[$campo] = 'integer';
        }
        elseif ($type == 'number'){
            $typeRules[$campo] = 'numeric';
        }
        elseif ($type == 'boolean'){
            $typeRules[$campo] = 'string';
        }
    }
    return $typeRules;
}
```

Figura 4.16: Mètode per obtindre les regles de validació

Ara passaré a explicar cadascun dels mètodes dels dos controladors. Primer serà el torn del *CRUD*.

Com ja he explicat al document en repetides ocasions les sigles *CRUD* fan referència a les accions de crear, llegir, actualitzar i eliminar, òbviament en anglès. Per tant hi haurà al controlador un mètode per cada acció, dos en el cas de llegir.

En primer lloc, per a llegir trobem dos mètodes, el primer d'ells anomenat *filtrar*. Aquest mètode primer de tot comprova que que el model existeix i després fa un *get* d'una taula de la base de dades segons els filtres passats com a paràmetres. A més, per si de cas la taula tingués moltes instàncies, agafe tan sols el nombre d'elements indicat, amb una paginació. El mètode en qüestió el trobem a la figura 4.17.

El segon mètode referent a l'acció de llegir tracta de *show*, aquest mètode torna les dades de la instància amb l'identificador passat com a paràmetre. Es pot observar el mètode a la figura 4.18.

```php
public function filtrar($modelo, $numElem, $columna, $operador, $valor, $orden, $colorden){
    $esta = $this->comprobarModelo($modelo);

    if($esta!=[]){
        return $this->errorResponse($esta, code: 404);
    }
    else{
        if ($operador=='nulo'){
            $instances = DB::table($modelo)->orderBy($colorden, $orden)->paginate($numElem);
            return response()->json( ['data' => $instances]);
        }
        else{
            if ($operador=='igual'){
                $operador='=';
            }elseif ($operador=='menor'){
                $operador='<';
            }elseif ($operador=='mayor'){
                $operador='>';
            }elseif ($operador=='menorig'){
                $operador='<=';
            }elseif ($operador=='mayorig'){
                $operador='>=';
            }
            $instances = DB::table($modelo)->where($columna, $operador, $valor)->orderBy($colorden, $orden)->paginate($numElem);
            return response()->json( ['data' => $instances]);
        }
    }
}
```

Figura 4.17: Mètode filtrar

```php
public function show($modelo, $id)
{
    $estaModelo = $this->comprobarModelo($modelo);

    if($estaModelo!=[]){
        return $this->errorResponse($estaModelo, code: 404);
    }
    else{
        $estaId= $this->comprobarId($modelo, $id);

        if($estaId!=[]){
            return $this->errorResponse($estaId, code: 404);
        }
        else{
            $instance = DB::table($modelo)->where( column: 'id', operator: '=', $id)->get();
            return response()->json($instance);
        }
    }
}
```

Figura 4.18: Mètode *show*

En segon lloc, per a l'acció de crear tenim el mètode *store*. Aquest mètode serà l'encarregat de que quant des del *frontend*, s'envie la respectiva petició, agafe les dades, les valide, cree la instància amb les dades, i finalment l'emmagatzeme a la taula en la base de dades. En la següent figura 4.19 es veu la part principal del mètode.

```
for ($i=1;$i<$length;$i++){
    if ($request[$keys[$i]] == 'true'){
        $insertData[$keys[$i]] = 1;
    }
    else if ($request[$keys[$i]] == 'false') {
        $insertData[$keys[$i]] = 0;
    }
    else if(strpos($request[$keys[$i]], needle: "T") !== false && strpos($request[$keys[$i]], needle: "+") !== false){
        $fecha = Carbon::createFromFormat( format: 'Y-m-d\TH:i:sP', $request[$keys[$i]]);
        $insertData[$keys[$i]]= $fecha->format( format: "Y-m-d H:i:s");



    }
    else{
        $insertData[$keys[$i]] = $request[$keys[$i]];
    }
}
$insertData['created_at'] = Carbon::now();
$insertData['updated_at'] = Carbon::now();
DB::table($modelo)->insert($insertData);
return response()->json($insertData);
```

Figura 4.19: Mètode *store*

D'una manera molt similar, per a l'acció d'actualitzar, està el mètode *update*. Aquest mètode rep els camps actualitzats i els modifique a la taula de la base de dades, sempre abans amb les respectives validacions. A continuació la figura 4.20 amb una part del mètode.

```
for ($i=1;$i<$length;$i++) {
    if ($request[$keys[$i]] == 'true') {
        $updateData[$keys[$i]] = 1;
    } else if ($request[$keys[$i]] == 'false') {
        $updateData[$keys[$i]] = 0;
    } else if (strpos($request[$keys[$i]], needle: "T") !== false && strpos($request[$keys[$i]], needle: "+") !== false) {
        $fecha = Carbon::createFromFormat( format: 'Y-m-d\TH:i:sP', $request[$keys[$i]]);
        $updateData[$keys[$i]] = $fecha->format( format: "Y-m-d H:i:s");
    } else {
        $updateData[$keys[$i]] =  $request[$keys[$i]];
    }
    $updateData['updated_at'] = Carbon::now();
}

DB::table($modelo)->where( column: 'id', operator: '=', $id)->update($updateData);
```

Figura 4.20: Mètode *update*

Finalment, l'acció d'eliminar, que com no podia ser d'altra manera, compta amb el mètode *destoy*. Aquest mètode busca la instància a la taula amb l'identificador passat com a paràmetre i l'elimina. En la següent figura 4.21 es pot observar el mètode.

```php
public function destroy($modelo, $id)
{
    $estaModelo = $this->comprobarModelo($modelo);

    if($estaModelo!=[]){
        return $this->errorResponse($estaModelo,  code: 404);
    }
    else{
        $estaId= $this->comprobarId($modelo, $id);

        if($estaId!=[]){
            return $this->errorResponse($estaId,  code: 404);
        }
        else{
            $instance = DB::table($modelo)->where( column: 'id',  operator: '=', $id)->get();
            DB::table($modelo)->where( column: 'id',  operator: '=', $id)->delete();
            return response()->json($instance);
        }
    }
}
```

Figura 4.21: Mètode *destroy*

Una vegada explicat aquest controlador, passem a l'altre, el *ModelController*. Podríem dir que aquest controlador és un controlador auxiliar, ja que la seua principal funció és passar-li les dades necessàries al *frontend* per poder generar automàticament les seues vistes.

El primer mètode d'aquest controlador és l'anomenat *index*, aquest mètode retorna un llistat amb el nom de totes les taules actuals a la base de dades. A continuació la figura 4.22 amb el mètode.

```php
public function index(){
    $tables = DB::select( query: 'SHOW TABLES');
    $tables = array_map( callback: 'current', $tables);
    return response()->json([
        'modelos' => $tables
    ]);
}
```

Figura 4.22: Mètode *index*

El segon mètode és *show*, aquest *show* a diferència de l'anterior, retorna l'esquema d'un recurs. És a dir, tot el contingut del fitxer *JSON Schema* del model seleccionat. Aquest fa falta al *frontend* per tal de generar automàticament els formularis de creació i modificació d'una instància. Es pot veure a la figura 4.23.

```php
public function show($modelo){
    $esta = $this->comprobarModelo($modelo);

    if($esta!=[]){
        return $this->errorResponse($esta, code: 404);
    }
    else{
        $pathname = "storage/app/public/". $modelo .".json";
        $jsonString = file_get_contents(base_path($pathname));
        $data = json_decode($jsonString, assoc: true);
        return response()->json($data);
    }

}
```

Figura 4.23: Mètode *show* de ModelController

Seguim als mètodes i els dos següents tenen una funció pràcticament igual, el primer d'ells *form*, retorna les columnes necessàries per al formulari. Un poc més dalt havia dit que els formularis es generen automàticament a partir d'un esquema, aleshores, aquesta informació no caldria enviar-la. Bé doncs, a la secció de problemes explicaré perquè ha fet falta passar-li-la al *frontend*. El segon d'ells *columns*, li passa les columnes necessàries per a la taula *CRUD*. La raó per la qué no són les mateixes columnes, és l'explicada a la secció d'informació addicional, a la tabla *CRUD* es mostren totes les columnes de la taula i als formularis tan sols aquelles columnes que l'usuari pot modificar, això exclou de l'equació les columnes de *id*, *created-at*, *updated-at*, les quals són inicialitzades i actualitzades automàticament pel sistema. A continuació la figura 4.24 amb un dels dos mètodes, ja que són pràcticament iguals.

```php
public function columns($modelo){
    $esta = $this->comprobarModelo($modelo);

    if($esta!=[]){
        return $this->errorResponse($esta, code: 404);
    }
    else{
        $pathname = "storage/app/public/". $modelo .".json";
        $jsonString = file_get_contents(base_path($pathname));
        $data = json_decode($jsonString, assoc: true);
        $columnas= Schema::getColumnListing($data['title']);
        return response()->json( ['columnas' => $columnas]);

    }
}
```

Figura 4.24: Mètode *columnas*

Finalment els dos últims mètodes. Aquests ja no són tant auxiliars, si no que afegeixen

funcionalitat al *frontend*, el primer d'ells *truncate*, el que fa és eliminar totes les instàncies d'una taula. El segon, *drop*, com és d'esperar, el que fa és eliminar una taula. A continuació la figura 4.25 amb els mètodes en qüestió.

```
public function truncate($modelo){
    $esta = $this->comprobarModelo($modelo);

    if($esta!=[]){
        return $this->errorResponse($esta,  code: 404);
    }
    else{
        DB::table($modelo)->truncate();
    }
}
public function drop($modelo){
    $esta = $this->comprobarModelo($modelo);

    if($esta!=[]){
        return $this->errorResponse($esta,  code: 404);
    }
    else{
        Schema::drop($modelo);
    }
}
```

Figura 4.25: Mètode *drop*

En açò últim, podem donar per conclosa, l'apartat de controladors. Tan sols quedaria per explicar l'arxiu on es troben totes les rutes de la *API* definides. En aquest arxiu simplement es troben totes les rutes de les que farà ús el *frontend*, hi ha rutes que tenen paràmetres, i totes les rutes executen un mètode d'un controlador, és a dir, un dels anomenats anteriorment. Com la definició de les rutes és casi igual per a totes, mostraré en la figura 4.26 sols una de cada tipus (*GET, POST, PUT, DELETE*).

```
Route::get( uri: '/modelo/{modelo}/show/{id}', action: 'CrudController@show')->name( name: 'modelo.show');
Route::post( uri: '/modelo/{modelo}/store', action: 'CrudController@store')->name( name: 'modelo.store');
Route::post( uri: '/modelo/{modelo}/update/{id}', action: 'CrudController@update')->name( name: 'modelo.update');
Route::delete( uri: '/modelo/{modelo}/destroy/{id}', action: 'CrudController@destroy')->name( name: 'modelo.destroy');
```

Figura 4.26: Rutes de la *API*

Com es pot veure, hi ha dos *POST* i cap *PUT*, la raó d'aquesta situació serà explicada a la secció de problemes.

Ara ja, l'apartat de funcionalitat del *backend* està explicat amb detall.

Problemes

Durant el desenvolupament del *backend* he tingut alguns problemes. No han sigut tant problemes greus, com problemes que podríem qualificar com problemes de principiant al estar fent ús d'unes tecnologies totalment noves per a mi.

El primer problema que vaig tindre, va ser en la funció de crear la base de dades a partir dels fitxers *JSON*. I és que com s'ha explicat a la secció d'informació detallada, els *JSON Schema* per les columnes, tenen definits un tipus, aquest és referent al tipus de la columna, ja siga *string*, *boolean*, *integer*, etc. No pareix que hi hage un problema ací, però es dona el cas de que per a les dates, el tipus que utilitza *JSON Schema* és també *string*. Per tant per a cada *string* he hagut de mirar també el camp *format*, que aquest camp sols està quan és una data.

Millor dir, sols estava, ja que ja no és així. El motiu és el següent, als requisits de l'aplicació el supervisor em va demanar que es pogueren ficar tipus *text* a la base de dades, però els *JSON Schema* no tenen aquesta funcionalitat, tan sols tenen *string*, per tant vaig haver d'afegir el camp format per als *string* que fossin *text*. Açò està fora de les especificacions del *JSON Schema* però ha valgut per a poder implementar els tipus *text*.

Una vegada solucionat aquest problema, no es quedava aquí, ja que per la modificació del tipus de columna, el que faig es comparar el tipus de l'esquema, amb el tipus de la columna a la base de dades. Per tant, quan comparava un *datetime*, estava comparant un *string* amb un *datetime*. Aleshores, he de fer comprovacions del tipus, si es *string*, veure si té *format*, si el format és *date-time*, fer la comparació amb *datetime* en lloc de *string*.

Però el problema que més temps m'ha consumit i a la vegada el problema amb la solució més incomprensible, ha sigut el problema de *CORS*, anomenat anteriorment a la secció d'estructura quan explicava els *middlewares*.

Aquest error ve donat quan es fa una petició des de *VueJS* a la *API*. Els navegadors protegeixen les aplicacions web per a que sols puguen interactuar amb aplicacions amb el mateix origen, aquesta política de seguretat és anomenada *same-origin*. Però hi ha voltes, que és necessària la comunicació amb servidors externs, com és ara el nostre cas. Per poder fer aquestes comunicacions existeix l'estàndard de CORS (*Cross-Origin Resource Sharing*).

Vaig buscar solucions i la que més es repetia era la de crear un *middleware* al *backend* que afegira capçaleres a les peticions. I una vegada creat el *middleware*, assignar al arxiu on es troben les rutes el *middleware* a les rutes que volguera que feren ús d'aquest *middleware*, que en el meu cas són totes.

Vaig fer això, pareixia que anava a funcionar, de fet vaig fer una *GET* per a provar i ja funcionava. El problema va sorgir quan feia peticions que no eren *GET*. Per a aquests casos seguia donant el mateix problema i em vaig passar dies intentant solucionar-ho sense cap mena de sort.

Finalment podríem dir que em vaig retre, i vaig optar per una solució pèssima que era instal·lar un *plugin* per a *Chrome* que anul·lava aquesta seguretat.

Amb el *plugin* ja anaven totes les peticions, però no m'agradava gens com a solució i si a la llarga el prototipus ja no fora un prototipus i els clients es tingueren que instal·lar un *plugin* per fer anar l'aplicació, doncs no era molt correcte. Per tant, quan tenia un poc de temps seguia mirant si encontrava alguna solució entre els comentaris d'altres programadors en fòrums. Fins que un dia, vaig encontrar un que va dir que ficant el *middleware* per a que fora general a l'aplicació i no per a les rutes seleccionades com havia fet jo, li solucionava l'error.

Vaig anar a provar-ho sense cap esperança, ja que no pareixia una solució amb sentit i per a sorpresa i alegria meua, va funcionar.

Aquesta solució em val ja que totes les rutes fan ús d'aquest *middleware*, però en el cas de que volguera que alguna ruta no fera ús d'aquest, ja no seria òptima.

A dia de hui seguisc sense trobar-li cap sentit a la solució.

A continuació a la figura 4.27 el *middleware* en qüestió.

```php
public function handle($request, Closure $next)
{
    header( string: "Access-Control-Allow-Origin: *");

    $headers = [
        'Access-Control-Allow-Methods'=> 'POST, GET, OPTIONS, PUT, DELETE',
        'Access-Control-Allow-Headers'=> 'Content-Type, X-Auth-Token, Origin'
    ];
    if($request->getMethod() == "OPTIONS") {

        return response()->make( content: 'OK', status: 200, $headers);
    }

    $response = $next($request);
    foreach($headers as $key => $value)
        $response->header($key, $value);
    return $response;
}
```

Figura 4.27: *CorsMiddleware*

4.1.2 Frontend

Pel que fa al *frontend*, la decisió de fer ús de *VueJS*, va ser de l'empresa. És la ferramenta en la que ells actualment desenvolupen les seues aplicacions, per tant al agafar un alumne en pràctiques volien que fera ús d'aquesta tecnologia per si de cas, el volgueren contractar, així aquest projecte valdria com aprenentatge del *software*. A més una de les raons per la que vaig elegir fer pràctiques a aquesta empresa, és perquè era de les poques que farien ús d'una tecnologia desconeguda per a mi, i volia aprofitar l'estància per aprendre-la.

Estructura

L'estructura del *frontend* és la mostrada a la figura 4.28.

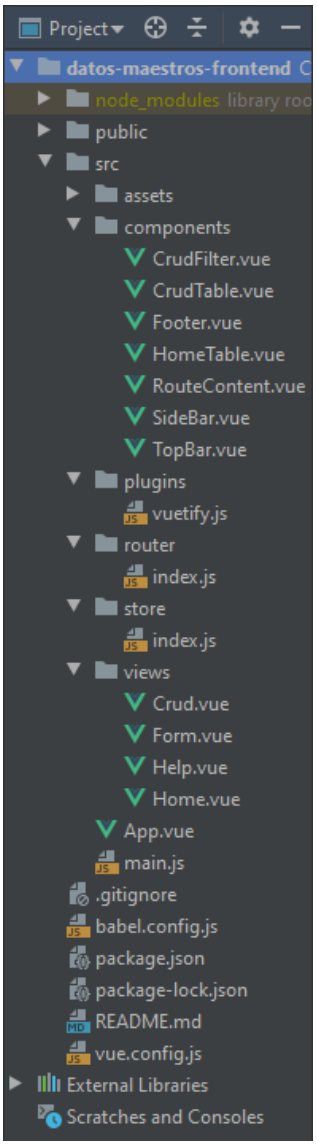

Figura 4.28: Estructura del *frontend*

La part important del projecte és la que es troba a l'interior de la carpeta *src*, per tant, tan sols entraré en detall amb aquest directori.

En primer lloc, trobem la carpeta *assets*, dins d'aquesta carpeta es guarden els arxius que importes als teus components, per exemple si volgués ficar el logo de l'empresa en l'aplicació, l'arxiu que el conté, es guardaria aquí. En este cas, no guarde cap arxiu a aquesta carpeta.

En segon lloc, trobem una de les carpetes principals, per no dir la principal. Es tracta de *components*, ací es guarden tots els components de l'aplicació. Com hem anomenat abans en

la secció tecnologies, *VueJS* es basa en la creació de components, cada part de l'aplicació és un component. En aquesta carpeta estarien tots els components, de no ser perquè vaig crear una nova carpeta anomenada *views*, la qual entrarem en detall més endavant.

Seguidament, el directori *plugins*, creat automàticament al instal·lar el paquet *vuetify*, dins d'aquest directori trobem l'arxiu de configuració de *vuetify*.

Després està la carpeta *router*, aquesta carpeta es creada al instal·lar el paquet *Router*. Aquest paquet ens ajuda amb la navegació entre els components que són pàgines. En l'aplicació tot és un component, però no tots els components són pàgines. Aleshores, per accedir a les pàgines, es farà amb les rutes definides en aquesta carpeta.

La següent carpeta és *store*, aquesta carpeta es creada al instal·lar el paquet *Vuex*. Aquest paquet serveix per crear una mena d'emmagatzematge, al que poden accedir i modificar tots els components. No és un paquet que hagi gastat molt, però des de l'empresa em van recomanar fer ús d'ell,

L'últim directori és l'anomenat anteriorment, *views*. Aquest directori no fa cap falta crear-ho, però és una pràctica habitual entre els programadors de VueJS. En aquest directori es posen tots els components que són pàgines, per tal de separ-los d'aquells components que no ho són i tindre una millor idea de l'arquitectura de l'aplicació ràpidament.

Finalment es troba l'arxiu *App*, podríem dir que aquest és el component arrel. *VueJS*, serveix per crear el que es diu aplicacions, d'una sola pàgina. Bé doncs, *App* és aquesta única pàgina i des d'aquesta pàgina es van cridant a tots els altres components que van apareixent i desapareixent en base a les accions de l'usuari.

Funcionament

Com acabe d'explicar, aquesta aplicació té un component arrel, així que explicaré el funcionament a partir d'aquest component. En aquest cas, a la figura 4.29 sols mostraré la part de *html*, ja que la part de *javascript* en aquest és insignificant.

```
<template>
  <v-app id="app">
    <side-bar :tablas="modelos"></side-bar>
    <top-bar></top-bar>
    <route-content></route-content>
    <pie></pie>
  </v-app>
</template>
```

Figura 4.29: Part *html* del component arrel *App*

Com podem veure a l'imatge, a aquest component es crida als components *SideBar*, *TopBar*, *RouteContent*, *Pie*. Tots aquests components estaran presents tota l'estona en l'aplicació. La part interessant és el *RouteContent*, aquest component el que fa es mostrar la pàgina que se li passa per la ruta. Ara mostraré cada component i com es forma la pàgina.

En primer lloc la *sideBar*, a la qual se li passen els models actuals de la base de dades per llistar-los. Podem observar-ho a la figura 4.30.

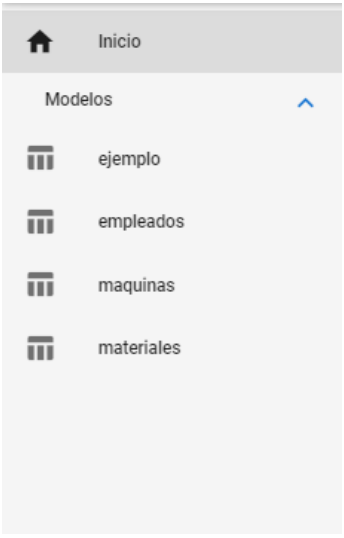

Figura 4.30: Component *Sidebar*

En segon lloc tenim *TopBar*, aquest component pot mostrar i amagar el *SideBar* amb un botó gràcies a la comunicació entre components implementada. Es mostra a la figura 4.31.

Figura 4.31: Component *TopBar*

La comunicació entre els components és una part important del desenvolupament en *VueJS*, per tant faré una petita interrupció en l'explicació de crear la vista i mostraré una forma de fer aquesta comunicació amb aquest exemple.

Al *TopBar* al pressionar el botó, s'emet un esdeveniment a l'aplicació a l'espera de que algú l'escolte. En la figura 4.32 es pot veure l'enviament.

```
<v-app-bar-nav-icon @click.stop="$root.$emit( event: 'changeDrawer', args: 1)"></v-app-bar-nav-icon>
```

Figura 4.32: Enviament de l'esdeveniment

Per tal de fer la recepció, al component *Sidebar*, he creat com un *listener*, que captura l'esdeveniment amb el nom definit i actua en conseqüència. En aquest cas, canvia el valor d'una

variable booleana, que és l'encarregada d'amagar o mostrar el *SideBar*. A la figura 4.33 es mostra la recepció.

```
mounted() {
    this.$root.$on( event: 'changeDrawer', callback: () =>{
        this.drawer=!this.drawer
    });
}
```

Figura 4.33: Recepció de l'esdeveniment

Una vegada acabada la interrupció seguim amb com es mostren els components. El següent és el *RouteContent*, que com hem dit abans, mostra la pàgina corresponent amb la ruta actual. En aquest cas com encara no hem canviat la ruta, és la pàgina principal "/", que en este cas mostra la pàgina *Home*. Aquesta pàgina mostra una taula amb tots els recursos emmagatzemats a la base de dades. Cada fila de la taula fa referència a una taula de la base de dades i compta amb tres accions. La primera canvia la ruta per mostrar la taula *CRUD* d'aquest recurs, la segona elimina les instàncies del model, i la tercera elimina la taula a la base de dades. En aquest cas no entraré amb detalls de codi, ja que es senzill. Podem veure-ho a la figura 4.34.

Figura 4.34: Component *Home*

Finalment l'últim component que crida *app*, es *pie*. Aquest component és el *footer* de l'aplicació, no li anomenat d'aquesta manera perquè era una paraula reservada en *VueJS*. A continuació en la figura 4.35 el component en qüestió.

El resultat final que es mostra a la web ajuntant tots els components és mostrat a la figura 4.36.

Figura 4.35: Component *Pie*

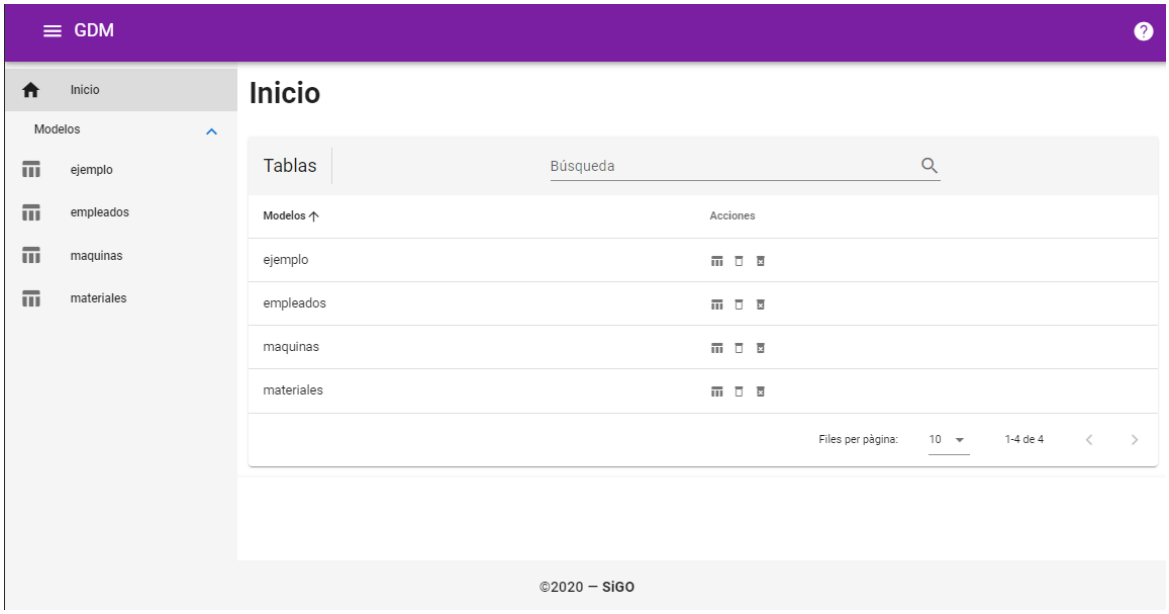

Figura 4.36: Pàgina de l'aplicació amb la ruta "/"

Per a les altres rutes es mantindran tant la barra de navegació i lateral com el *footer* i l'únic que canviarà és el contingut de *RouteContent*. A la figura 4.37 les rutes definides dins de la carpeta *router*.

```
const routes = [
    {
        path: '/',
        name: 'inicio',
        component: Home
    },
    {
        path: '/crud/:modelo',
        name: 'crud',
        component: Crud
    },
    {
        path: '/form/:modelo/:action',
        name: 'form',
        component: Form
    },
    {
        path: '/help',
        name: 'help',
        component: Help
    },
```

Figura 4.37: Rutes per a la navegació entre components

Com es pot observar, també es poden passar paràmetres a les rutes.

Ara passem a la pàgina principal del projecte. Es pot accedir mitjançant la barra lateral o la taula en la pàgina d'inici. Al seleccionar un model, es mostrarà la ruta */crud* del component.

Aquest component és on es duran a terme les accions *CRUD* del gestor de dades mestres. Està format per una taula on es mostren les instàncies del recurs, açò cobreix la R de *read*. A més la taula compta amb un botó per crear una nova instància, que cobreix la C. Per cada fila, que fa referència a una instància de la taula, es tenen dos botons amb les accions de modificar i eliminar, que cobreixen les sigles restants de *CRUD*. Podem observar-ho a la figura 4.38.

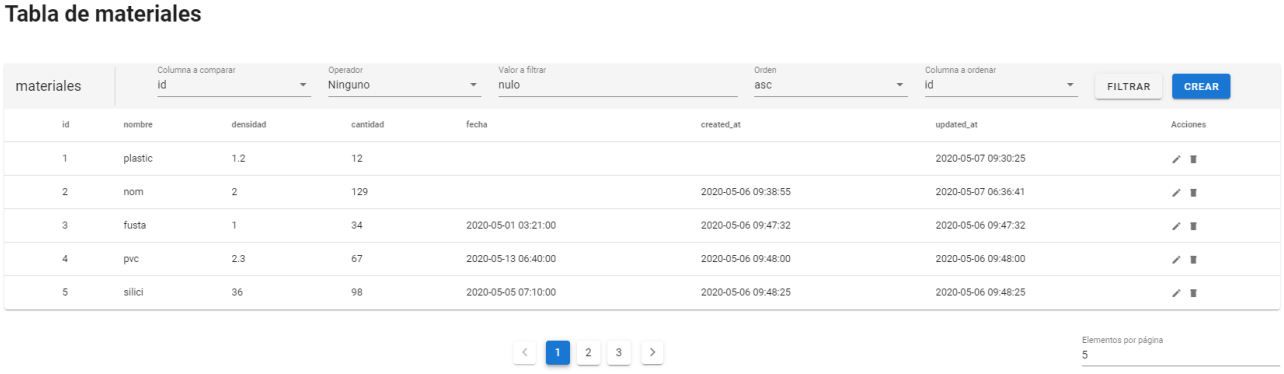

Figura 4.38: Component *CRUD*

A més, com a funcionalitat addicional, es pot seleccionar quants elements mostrar per pàgina, i aplicar uns filtres a la cerca. Als filtres es pot seleccionar la columna a comparar, l'operador,

el valor a comparar, l'ordre a mostrar i la columna sobre la que s'ordena.

Per a millorar l'eficiència del prototipus, la paginació de la taula funciona de la següent manera. Al haver la possibilitat de que un client puga tindre centenars d'instàncies en una taula, serà ineficient que s'agafaren totes les instàncies per a mostrar-les. Per això tant sols s'agafem el nombre d'instàncies que l'usuari a seleccionat i quan es passa de pàgina es fa una nova cerca. D'aquesta manera en lloc de rebre una quantitat massiva d'instàncies que pot causar una espera llarga, sols es rep una quantitat suportable per al sistema.

En quant a la creació i modificació d'una instàncies es fa sobre la mateixa pàgina, per tant a la ruta d'aquest component se li passa per paràmetre un valor per diferenciar la creació de la modificació.

Aquesta pàgina s'accedeix amb la ruta "/form", que com el seu nom indica, conté un formulari amb els camps del recurs accessibles per a l'usuari. Aquest formulari estarà omplit amb les dades actuals en cas de tractar-se de la modificació, i estarà buit a l'espera d'omplir-ho en el cas de la creació, com s'ha explicat a la secció d'informació addicional, aquest formulari es creat automàticament a partir del *JSON Schema* del recurs gràcies al paquet *koumoul/vjsf*.

Cada camp del formulari té les seues regles de validació i el formulari té camps obligatoris a omplir, quan es compleixen aquestes regles, apareixerà el botó respectiu per a crear o modificar la instància.

A continuació en la figura 4.39 un exemple d'aquest component quan s'accedeix mitjançant la creació. Per a la modificació seria el mateix però els camps ja estarien omplits amb les dades actuals. El haver de mostrar les dades actual dins dels camps ha sigut el problema principal que he tingut al *frontend*, l'explicaré a la secció problemes.

Finalment l'última pàgina tracta d'un manual d'ajuda per als usuaris on s'explica cada pàgina, cada component i cada botó. Per accedir a aquesta pàgina s'ha de pressionar el botó de la barra de navegació amb un símbol d'interrogació. El manual compta amb un index per dirigir a l'usuari a la secció en la que està interessat.

L'últim detall del funcionament, però no el menys important, vaig a explicar com es comunica el *frontend* amb el *backend*. Aquesta comunicació es porta a terme mitjançant el paquet *axios*. Aquest paquet, facilita la comunicació amb les diferents peticions *GET*, *POST*, *PUT*, *DELETE*.

A continuació a la figura 4.40 un exemple d'una petició *GET* i un d'una *POST*.

Aquesta petició *GET*, rep una resposta la qual guarda a la variable anomenada *modelos*. Podem veure-ho a la figura 4.41.

La petició *POST* a banda de la *url* de la petició se li passa les dades que es vol guardar a la base de dades.

Amb aquesta última explicació, donem per acabada la secció de funcionament del *frontend*.

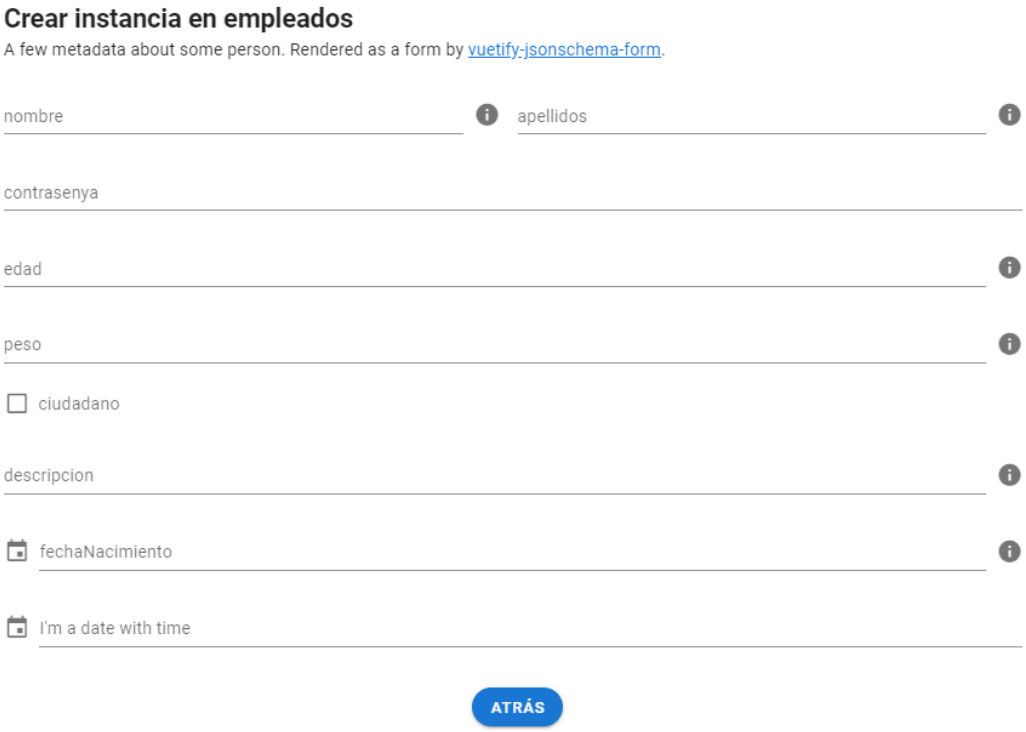

Figura 4.39: Component *Form* per a la creació

```
this.$http.get( this.url)
        .then((result) => {
          this.modelos = result.data.modelos
        });
```

Figura 4.40: Petició *GET*

```
this.$http.post( url: this.$store.state.url + '/api/modelo/' + this.$route.params.modelo + '/store' ,
    formStore,  config: {headers: {
            'Content-Type': 'multipart/form-data'
        }
    })
```

Figura 4.41: Petició *POST*

Problemes

Encara que aquesta tecnologia era totalment nova per a mi, i a al grau no havia treballat molt amb *frontend*, gràcies al curs realitzat al principi de l'estància, no he tingut grans problemes durant el desenvolupament del *frontend*.

L'únic gran problema que cal destacar, és l'anomenat anteriorment: la modificació d'una instància. Aquest problema es pot desglossar en dos. El primer d'ells el haver de posar els valors actuals en cada camp en un formulari que es crea automàticament. I el segon, que no actualitzava cap camp, és adir la petició per actualitzar la instància no funcionava.

Com a solució temporal per al primer problema, vaig proposar al supervisor el mostrar les dades actuals baix del formulari. De fet ho vaig arribar a implementar però al fer-li la demostració de l'aplicació estàvem d'acord en que no era una solució molt correcta.

Aleshores vaig començar a intentar solucionar-ho, vaig provar a buscar si algú havia tingut el mateix problema i sabia com solucionar-ho, però al tractar-se d'un paquet nou i poc conegut no hi ha gaire res als fòrums.

Finalment vaig observar que el component del paquet fa ús d'una variable *v-model* anomenada *dataObject*, en la qual es guarden les dades quan s'envia el formulari. A continuació la figura 4.42 amb la implementació del component.

```
<v-form v-model="formValid">
    <v-jsf v-if="ok" :schema="schema" v-model="dataObject" :options="options"
           @error="showError" @change="showChange" @input="showInput" />
</v-form>
```

Figura 4.42: Component *vjsf*

Aleshores, vaig inicialitzar aquesta variable amb les dades actuals de la instància i màgia, les dades actuals apareixien en cada camp del formulari. En la figura 4.43 podem observar el comportament.

Solucionat aquest problema, sorgia un nou problema, la petició que actualitzava la instància a la base de dades no anava. Jo estava convençut de que li enviava correctament les dades al *backend*, aleshores no podia ser un problema de *frontend*. A continuació en la figura 4.44 la comprovació de que el *frontend* era correcte i guardava bé les dades.

Finalment, quan ja casi em retria, l'informàtic de l'empresa va encontrar la raó per la que fallava [3]. La raó era que a les noves versions de *Laravel*, en els formularis ja no estaven permeses les peticions *PUT*, *PATCH* i *DELETE*, i com a li passava des del *frontend* un objecte generat a partir d'un formulari a una petició *PUT*, no funcionava.

La solució va ser tant senzilla com canviar el mètode de la petició d'un *PUT* a un *POST*. Aquesta es la raó per la que no hi ha una ruta *PUT* a les rutes *CRUD* definides al *backend*, com havia anomenat a la secció de funcionament del *backend*.

Figura 4.43: Component *Form* per a la modificació

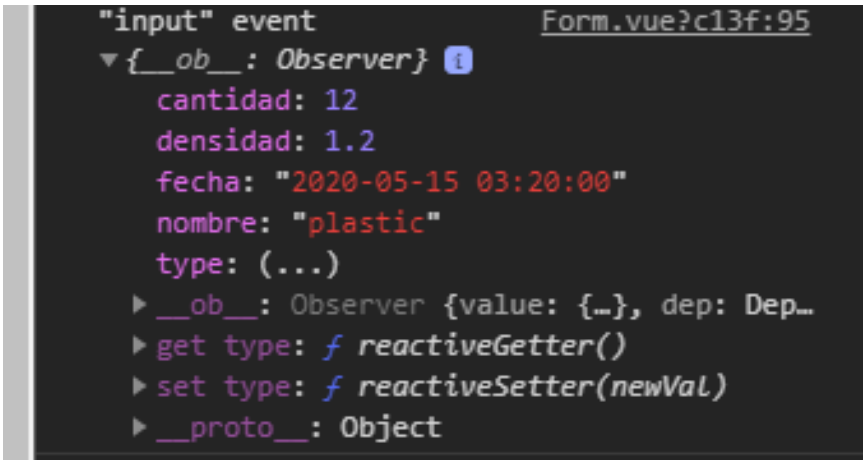

Figura 4.44: Consola amb les dades de *dataObject*

4.2 Proves

Com s'ha definit als requisits, els *tests* per comprovar el correcte funcionament de l'aplicació no estaven inclosos a l'abast del projecte, ja que es tracta d'un prototipus i el temps era molt justet. Encara així, durant la implementació, s'han anat fent proves per comprovar que funcionava correctament. A continuació una explicació de com s'han dut a terme tant al *backend* com al *frontend*.

4.2.1 Backend

Per al *backend* s'han fet dos tipus de proves. Una per a la part de la base de dades i l'altra per a la *API*.

Per a la base de dades, al haver de crear les taules a partir de la lectura d'un arxiu, les proves eren molt senzilles de realitzar. Quan la funció acabava de llegir els arxius i havia fet les respectives creacions i modificacions, es comparava la informació del *JSON Schema* al que s'havia creat a la base de dades, gràcies al gestor de base de dades inclòs en *xampp* anomenat *phpMyAdmin*. Un exemple de les proves a la figura 4.45

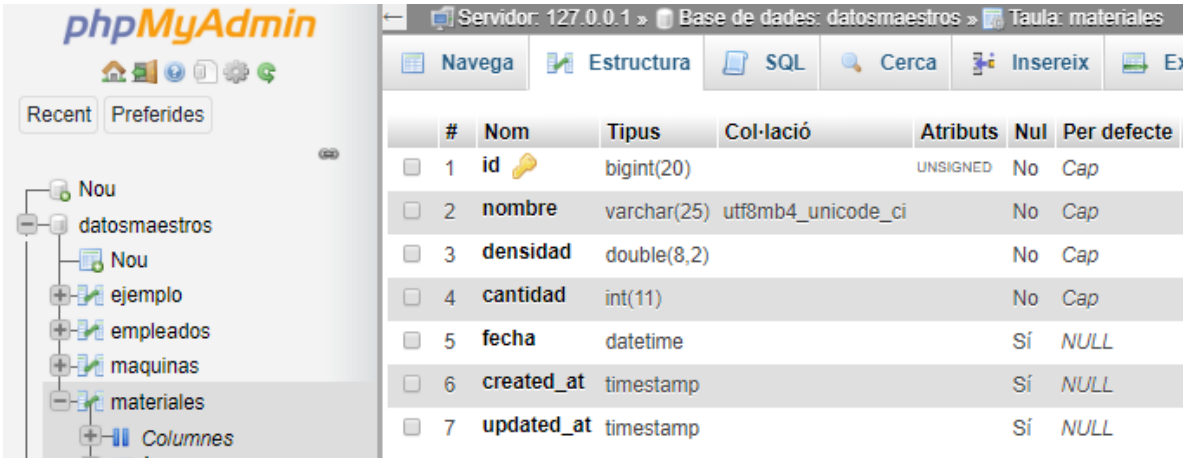

Figura 4.45: Interfície de *phpMyAdmin*

En quant a la part de la *API*, vaig gastar el programa per tots conegut *Postman*. En ell es poden fer les peticions *API* que vulgues per comprovar el correcte funcionament de totes elles, sense haver d'implementar el *frontend*.

4.2.2 Frontend

Per al *frontend* les proves que s'han fet han sigut simplement mostrar l'informació requerida per la consola del navegador, ja siga per veure quines dades rep des del *backend*, quines dades envia, l'estructura de les dades, l'estat actual dels atributs, etc. Un exemple d'aquestes proves és el que trobem a l'imatge 4.44 prèviament mostrada.

4.3 Millores

Al tractar-se d'un prototipis aquesta no és la versió final del producte. I hi ha algunes millores, que ja s'han comentat amb l'empresa, que es podrien dur a terme.

Òbviament, al ser pràcticament un principiant de les tecnologies emprades, es podrien fer millores tant a l'estructura del codi, com a l'estil del *frontend*. Però hi ha altres millores per afegir funcionalitat que podrien ser de gran utilitat.

Entre elles, la possibilitat d'elegir l'idioma de l'aplicació, *SiGO* treballa amb moltes empreses de la zona, i algunes d'elles com per exemple els ajuntaments, demanen a l'empresa que les aplicacions estiguen en valencià.

Un altra millora seria un sistema d'autenticació. Principalment per temes de seguretat, per a que no pugui accedir a les dades qualsevol. A més, amb aquest sistema es podrien definir permisos, per a que depenent del càrrec en l'empresa, pogueres tindre diferents nivells d'accessibilitat.

Finalment la implementació de tests, aquesta millora suposaria un gran avanç en temes de mantenibilitat. Ja que si es dugueren a terme actualitzacions es podrien passar els tests per a veure que el funcionament no ha canviat i ha deixat de funcionar.

Capítol 5

Conclusions

Per concloure aquesta memòria, vaig a explicar perquè vaig elegir aquesta oferta i de que m'ha servit l'estància, tant a nivell laboral com a nivell personal.

Aquest projecte el vaig elegir per varies raons, la primera d'elles i la principal va ser que anava a fer ús de tecnologies totalment noves per a mi.

Un altra de les raons va ser que no m'agradava molt treballar en la part del *backend*, i en la definició de les tasques que es mostrava a l'oferta ficava que tan sols hauria de treballar en la part del *frontend*, ja que la part del *backend* me la facilitaria l'empresa.

Ara pel que fa a l'experiència, per una part a nivell laboral m'ha servit per saber com és el dia a dia de treballar en una empresa informàtica i les relacions entre els companys de l'empresa. He de mencionar que des d'un principi m'he sentit molt còmode i ha sigut molt agradable treballar amb el supervisor i l'informàtic de l'empresa.

Per l'altra part, a nivell personal aquesta estància m'ha servit per a madurar com a persona, al entrar ja al món laboral i començar a comportar-me com una persona adulta. També m'ha servit per a reafirmar que vaig elegir molt bé tant el grau com l'itinerari, ja que aquestes eleccions me les vaig qüestionar durant alguns moments de la carrera.

A més, si tot això no era prou, les tasques definides inicialment van canviar després de parlar amb el supervisor i vam decidir que també implementara jo la part del *backend*, i gràcies a açò, m'he donat compte que si que m'agrada la part del *backend*, al contrari del que pensava inicialment i un dels motius que me va fer elegir aquesta proposta. El que no m'agradava eren les tecnologies utilitzades a la carrera per fer la part del *backend*, no així aquesta part.

Finalment, agrair a les persones involucrades en aquest treball de fi de grau. Al tutor José Luis Llopis, el supervisor Vicente Agost, l'informàtic de l'empresa Eric Esteban, a tot el professorat que he tingut durant la carrera i com no, a la meua família.

Bibliografia

[1] Michael Droettboom. Understanding json schema. https://json-schema.org/understanding-json-schema/index.html. [Consulta: 25 d'Abril de 2020].

[2] Indeed. Sou mitjà d'un programador júnior a l'estat espanyol. https://www.indeed.es/salaries/programador-junior-Salaries. [Consulta: 13 de Maig de 2020].

[3] Laravel. Method field. https://laravel.com/docs/5.8/blade#method-field. [Consulta: 16 de Maig de 2020].

[4] PhpStorm. Preu de compra del phpstorm. https://www.g2.com/products/phpstorm/pricing. [Consulta: 13 de Maig de 2020].