# Documentación RHEA

## Instalación del proyecto

Todos estos pasos se han seguido en un sobremesa con Ubuntu 20.04 LTS.

```
# Hay que instalar npm y node
sudo apt install npm

sudo apt install curl
# Instalar un gestor de versiones de node (nvm), necesario para Vue
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash
source ~/.bashrc

nvm list-remote
nvm install v12.22.12
nvm use v12.22.12
# Se pueden listar las versiones instaladas con nvm list
```

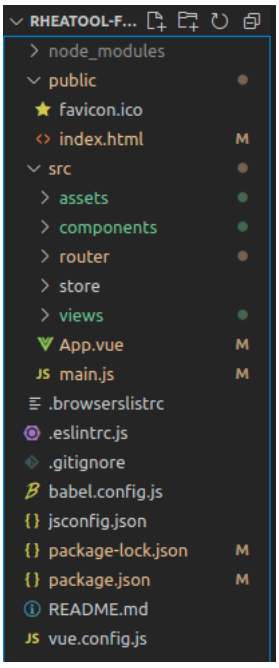
Ahora toca descargar el repositorio e ir a la carpeta ‘rheatool-frontend’. Alli ejecutar los siguientes comandos:

```
npm install
npm rebuild node-sass

# Y por ultimo, para lanzar la aplicación
npm run serve
```

## Sobre Vue 3

El frontend de RheaTool está construido en Vue 3, un framework para la creación de interfaces basada en componentes. Esto hace que los proyectos tengan una estructura concreta y un determinado tipo de ficheros con los que trabaja, como los .vue, que son archivos que combinan 3 tipos de formatos en uno solo: Una parte de Script en Javascript, una parte de etiquetado en HTML5 decorado con clases de Vue3 y por ultimo una parte de estilo en CSS o derivados, como en nuestro caso que es SCSS.

La estructura de un proyecto (puede variar) es la siguiente:



La **carpeta public** incluye el index.html, donde se integran algunas dependencias como los iconos de Google Fonts.

La carpeta src es la que tiene más contenido relevante, y ahí hay varias carpetas y archivos a detallar:

- Assets: Carpeta donde se guardan elementos de diseño como imágenes y fuentes de texto.
- Components: Aquí van los componentes reutilizables que creemos para nuestra aplicación, como puede ser una barra lateral, la plantilla de un modelo o un lenguaje, una sección, un menú...
- Router: Aquí se guarda todo lo relacionado con el direccionamiento de la aplicación.
- Store: Para almacenar las mutaciones y estados en una aplicación con un modelo orientado a las mismas.
- Views: Para almacenar los archivos que corresponden a las vistas, que estarían formadas por varios componentes unidos.
- App.vue: Fichero que describe elementos comunes en la aplicación, como la paleta de color.
- main.js: Fichero que contiene las lineas de ejecución e importación de elementos de la aplicación.

El funcionamiento de Vue 3 es relativamente sencillo en cuanto al modelo de frameworks se refiere. Su documentación tiene bastante información y hay muchos tutoriales de muchos niveles por ahí en video.

https://vuejs.org/guide/introduction.html

De todas formas, aquí van unas nociones básicas:

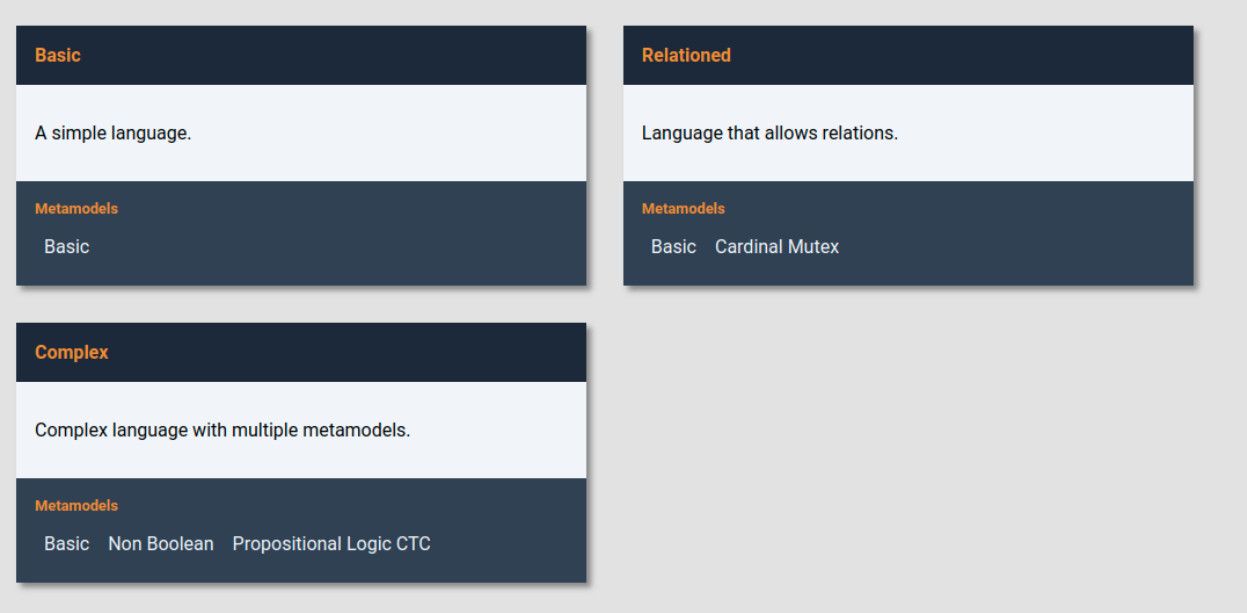
### Estructura de un componente

Como ya comentamos antes, un componente es un elemento reutilizable en la aplicación. Un ejemplo sería lo comunmente conocido como element-card, que son un conjunto de elementos obtenidos de la base de datos que se muestran con el mismo formato (peliculas, posts, recetas...). Nosotros disponemos de momento de ModelCards y LanguageCards para representar los lenguajes y modelos, por ejemplo:

```
<template>
    <section>
        <h4>{{ name }}</h4>
        <p>{{ description }}</p>
        <div class="language">
            <h6>{{ languageName }}</h6>
            <span>
                {{languageId}}
            </span>
        </div>
    </section>
</template>

<script>
export default {
    name: 'model-card',
    props: {
        id: String,
        name: String,
        description: String,
        languageId: String,
        languageName: String,
        features: Array,
        crossTreeConstraints: Array
    }
}
</script>

<style lang="scss" scoped>
section {
    display: flex;
    flex-direction: column;
    justify-content: space-between;
    width: 100%;

    background-color: var(--light);
    margin: 1rem;
    cursor: pointer;

    h4 {
        width: 100%;
        padding: 1rem;
```

```
        background-color: var(--dark);
        color: var(--primary);
    }
}
</style>
```

Como se puede ver, esta dividido en 3 partes: template para el HTML, script para el javascript y style para los stilos scss (scoped significa que estos estilos solo se aplican en este componente, permitiendo muchisima más limpieza en los estilos eliminando busquedas extrañas). La parte de estilo en este código está incompleta, pero en general este código sirve para mostrar algo así.




Pero para mostrar todos los elementos de forma iterativa se utilizaría una directiva de Vue, funciones especiales que permiten diversas funcionalidades, y en este caso nos valdría para recorrer un conjunto de elementos de un conjunto. Esto es algo que nos suena como programadores, para esto se puede usar un bucle for, y Vue implementa para estas situaciones la directiva v-for.

```
<div class="content">
    <article>
        <language-card @click='selectLanguage(language)'
            v-for="(language, index) in languages"
            :key="index"
            :name="language.name"
            :description="language.description"
            :metamodels="language.metamodels"/>
    </article>
```

Con esta directiva se crearía dentro de la etiqueta article un elemento language-card por cada elemento en el array languages. Luego se viene otra directiva llamada v-bind pero resumida en ‘:’ delante de key, name, description y metamodels, que lo que hacen es enlazarla a un modelo de datos. Recomiendo buscar en la documentación más información sobre v-bind y el resto de directivas de Vue 3.

Luego, los componentes pueden tener funcionalidades más complejas, tener métodos propios descritos por nosotros o incluso métodos que se realicen al iniciarse o destruirse como los del siguiente componente dedicado a la creación de popups.

```
<template>
    <div class="popup">
        <div class="popup-inner">
            <div class="popup-header">
                <slot/>
            </div>
            <div class="popup-content">
                <FormKit type="form" :actions="false" @submit="submitHandler(element)" v-model="element">
```

```
                    <FormKitSchema :schema="schemaSelected" />
                </FormKit>
                <div class="button-container">
                    <button @click="submitForm(element)">Create</button>
                    <button @click="togglePopup()">Cancel</button>
                </div>
            </div>


        </div>
    </div>
</template>

<script>
import { FormKitSchema } from '@formkit/vue';

export default {
    name: 'popup-form',
    components: {FormKitSchema},
    props: {
        schemaSelected: Array
    },
    data() {
        return {
            element: {}
        }
    },
    methods: {
        togglePopup() {
            this.$emit("togglePopup");
        },

        submitForm(element) {
            this.$emit("submitForm", element)
        }
    }
}
</script>

<style lang="scss" scoped>
...
</style>
```

### Estructura de una vista

Las vistas son elementos que no deberían reutilizarse y a las que el router de la aplicación tiene acceso. Su código es similar al de un componente, dividido también en 3 partes. Este es el de la vista de lenguajes.

```
<template>
    <main class="language-page">
        <rhea-upsidebar
            title="Languages List"
            element="language"
            @togglePopup="togglePopupForm" />
        <PopupForm
            :schemaSelected="schemaSelected"
            v-if="formPopup"
            @togglePopup="togglePopupForm"
            @submitForm="createLanguage">
            <h3 class="popup-title">Language Creation Form</h3>
        </PopupForm>
        <div class="content">
            <article>
                <language-card @click='selectLanguage(language)'
                    v-for="(language, index) in languages"
                    :key="index"
                    :name="language.name"
                    :description="language.description"
                    :metamodels="language.metamodels"/>
            </article>
            <aside>
                <FormKit type="form" ref="languageForm" v-if="languageSelected" :actions="false" @submit="submitHandler(languageSelecte
d)" v-model="languageSelected">
                    <FormKitSchema :schema="schemaSelected" />
                </FormKit>
                <div class="button-container" v-if="languageSelected">
                    <button @click="submitForm">Save changes</button>
```

```
                    <button @click="deleteSelected">Delete Language</button>
                </div>
                <h3 class="no-selection" v-else>No language selected</h3>
            </aside>
        </div>
    </main>
</template>

<script>
import RheaUpsidebar from '@/components/RheaUpsidebar.vue';
import LanguageCard from '@/components/LanguageCard.vue';

import PopupForm from '@/components/PopupForm.vue';
import { FormKitSchema } from '@formkit/vue';

export default {
    name: 'language-view',
    components: {RheaUpsidebar, LanguageCard, FormKitSchema, PopupForm},

    data () {
        return {
            languages: [...]
                }
            ],
            showContextMenu: false,
            formPopup: false,
            languageSelected: null,
            schemaSelected: [...]
        }
    },

    methods: {
        selectLanguage(language) {
            this.languageSelected = language;
        },

        submitForm() {
            const node = this.$refs.languageForm.node;
            node.submit();
        },

        submitHandler(language) {
            alert('Changes on language ' + language.name + ' saved!');
        },

        togglePopupForm() {
            this.formPopup = !this.formPopup;
        },

        createLanguage(element) {
            this.languages.push(element);
            this.formPopup = false;
        },

        deleteSelected() {
            let language = this.languageSelected;
            this.languageSelected = null;
            this.languages = this.languages.filter(item => item.id !== language.id);
        }
    },


}
</script>

<style lang="scss" scoped>
.content {
    display: flex;
}
article {
    padding: 1rem;
    display: flex;
    align-content: flex-start;
    width: 75%;
    flex-flow: row;
    flex-wrap: wrap;
}
...
</style>
```

Como se puede ver, integra diversos componentes, algunos de forma iterativa con directivas, añade métodos para tratar con ellos y dispone de estilos.

Habilitar el acceso a esta pagina es algo que se realiza en el fichero de router

```
import { createRouter, createWebHistory } from 'vue-router'
import LanguageView from '../views/LanguageView.vue'

const routes = [
  {
    path: '/',
    name: 'language',
    component: LanguageView
  }
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})

export default router
```

## Progreso del proyecto

En el estado actual del proyecto solo se cuenta con el front-end, que aun está en desarrollo. Las funcionalidades que realiza son:

- Mostrar una lista de lenguajes creados, dando la posibilidad de crear nuevos de forma local (no se guardan en un backend ni base de datos de momento).
- Mostrar una lista de modelos creados, dando la posibilidad de crear nuevos de forma local (no se guardan en un backend ni base de datos de momento).

La idea es disponer de una gestión de usuarios más adelante gestionada en el backend, que solo devuelva los modelos y lenguajes creados por el usuario a través de request con una librería como puede ser **Axios**.

Actualmente se están utilizando en Vue dos herramientas que pueden ser muy importantes en el desarrollo de los requisitos del proyecto: **FormKit** y **Context Menu**.

### Formkit

Se trata de una herramienta externa para Vue 3 (https://formkit.com/essentials/what-is-formkit) que se centra en la creación y validación de formularios, pero lo verdaderamente importante es que permite la creación de formularios a través de esquemas JSON, pudiendo desacoplar los formularios y los tipos de datos del frontend creando dichas plantillas en el backend, según las necesidades. Esto puede ser vital para gestionar luego los atributos de las Features dependiendo de los metamodelos seleccionados, ya que si, por ejemplo, el Frontend requiere modificar una Feature con elementos numéricos añadidos, le Backend le pasará en ese momento una plantilla JSON que disponga de dichos campos numéricos, y si luego se quiere cambiar una que no tenga campos numéricos el backend devolverá una plantilla de formulario sin dicho campo.

A continuación se presenta una plantilla JSON de ejemplo, ubicada dentro de la aplicación en la vista de lenguajes.

```
schemaSelected: [
    {
        $formkit: 'text',
        name: 'name',
        label: 'Name',
        validation: 'required'
    },
    {
        $formkit: 'text',
        name: 'description',
        label: 'Description',
        validation: 'required'
    },
    {
        $formkit: 'checkbox',
        name: 'metamodels',
        label: 'Language Constructors',
        options: [
```

```
            {
                value: 'Basic',
                label: 'Features and Feature Models',
                help: 'Basic Language Constructors that define a Feature and the attributes it owns.'
            },
            {
                value: 'Cardinal Mutex',
                label: 'Cardinality and Mutex groups, Multiplicity',
                help: 'Includes new advanced groups of Feature Models that are the Mutex and the Cardinal, last of them with Multiplicity.'
            },
            {
                value: 'Non Boolean',
                label: 'Non Boolean Attributes',
                help: 'Add Data Types to the Features.'
            },
            {
                value: 'Propositional Logic CTC',
                label: 'Propositional Logic Cross-Tree Constraints',
                help: 'Add several types of relations as Implies, Excludes, Not, Xor...'
            },
        ],
        help: 'Select the Language Constructors'
    }
]
```

### Context Menu

Permite cambiar las acciones que realiza el clic del ratón en elementos de la aplicación, mostrando por ejemplo un menú con diversas acciones. El prototipo que estaba construyendo funciona de la siguiente forma:

```
<context-menu :display="showContextMenu" ref="menu">
        <ul>
            <li @click="modifyLanguage"> Modify </li>
            <li @click="deleteLanguage"> Delete </li>
        </ul>
    </context-menu>
    <language-card @click='openContextMenu($event, language)'
        v-for="(language, index) in languages"
        :key="index"
        :name="language.name"
        :description="language.description"
        :metamodels="language.metamodels"/>
```

Se crea un elemento de context-menu con menu como referencia. Luego, en los elementos sobre los que quieres aplicar la gestión de un context menu, creas un evento (cada componente referenciando a un lenguaje con evento @click en este caso, el clic izquierdo) que abra dicho menú. La función OpenContextMenu incluiría lo siguiente:

```
openContextMenu(e, language) {
    this.$refs.menu.open(e, language);
    console.log('Clicked element: ' + language.name);
}
```

Esta funcion en su primera línea actúa sobre $refs, es decir, referencias, y nuestro elemento context-menu tiene la referencia ‘menu’, por lo que $refs.menu.open realiza la funcion open del elemento context-menu, la cual se ubica en su componente y lo que hace es almacenar el elemento sobre el que se ha clicado en una variable llamada element.

```
open(evt, element) {
    // updates position of context menu
    this.left = evt.pageX || evt.clientX;
    this.top = (evt.pageY || evt.clientY) - window.pageYOffset;
    // make element focused
    // @ts-ignore
    nextTick(() => this.$el.focus());
    this.show = true;
    this.element = element;
}
```

## Pasos a seguir

Para seguir con el proyecto hay que centrar muchos esfuerzos en crear un backend competente, encargado de **realizar traducciones de los archivos .ecore a JSON y viceversa**, para crear por ejemplo las plantillas a leer por Formkit, además de facilitar la manipulación de esos datos luego con formularios, ya que JSON es muy facil de tratar.

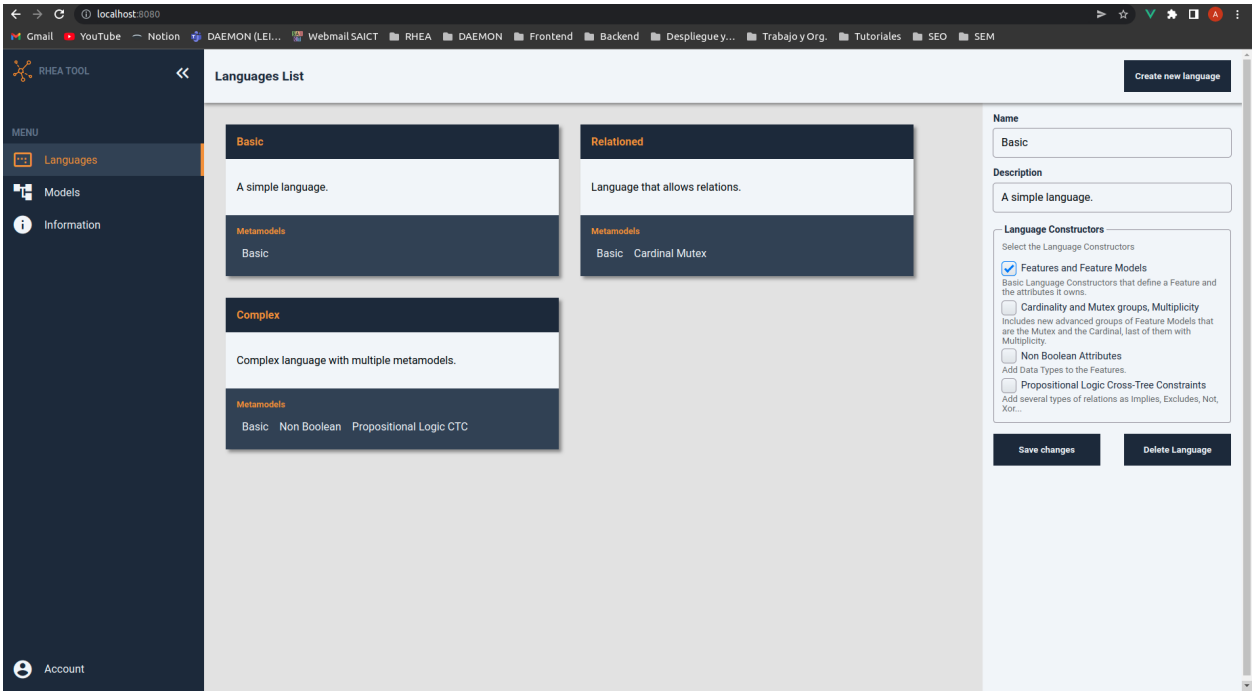
Otro punto a tener en cuenta es **la gestión de usuarios**, algo que se debe integrar para mostrar solo los lenguajes creados por un usuario y no todos los de la base de datos. También, por supuesto, opciones del backend para facilitar la **creación y eliminación de modelos o lenguajes en la base de datos**, además de dar los **entrypoints** para disponerlos al frontend.

En el frontend habrá que **profundizar más en el Context Menu y darle muchas más funcionalidades dependiendo de los metamodelos seleccionados**, al igual que con los formularios, que aunque se los de el backend dependiendo de los metamodelos, hay que asegurar su correcto funcionamiento.

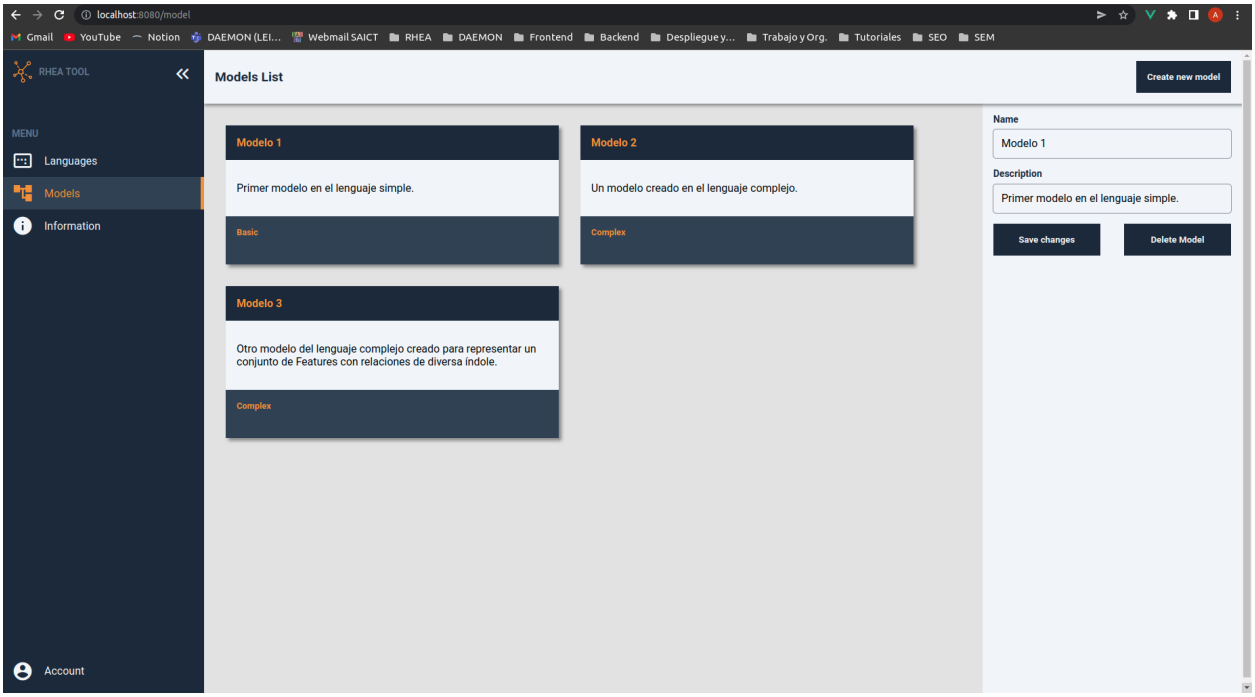
## Capturas de la aplicación

Aquí se mostrarán capturas de las dos páginas creadas actualmente, que son la de lenguajes y modelos. La página de información tiene una versión de la página de lenguajes que trabaja el context-menu a modo de ejemplo.

### Muestra de la página de Lenguajes




### Muestra de la página de Modelos


localhost:8080/model
Gmail
YouTube
Notion
DAEMON (LEI...
Webmail SAICT
RHEA
DAEMON
Frontend
Backend
Despliegue y...
Trabajo y Org.
Tutoriales
SEO
SEM
RHEA TOOL
MENU
Languages
Models
Information
Account
Models List
Create new model
Modelo 1
Primer modelo en el lenguaje simple.
Basic
Modelo 2
Un modelo creado en el lenguaje complejo.
Complex
Modelo 3
Otro modelo del lenguaje complejo creado para representar un conjunto de Features con relaciones de diversa índole.
Complex
Name
Modelo 1
Description
Primer modelo en el lenguaje simple.
Save changes
Delete Model