VAADIN FLOW CON SPRING BOOT & DBMS H2.

Desarrollaremos una aplicación básica, que realice las operaciones **CRUD** Integrada con **Vaadin Flow, Spring Data JPA**, **y H2 DBMS,** con la nueva versión del framework de vaadin 14, por lo cual nos permite construir aplicaciones web modernas.

A continuación, las herramientas a utilizar.

Requisitos:

Descripción	Software	Link de Descarga
Lenguaje de programación	Java SE 8	http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
(IDE)	Intellij	https://www.jetbrains.com/idea/download/
Framework	Vaadin	Versión 14.x
	Spring Boot	Version 2.x
DBMS	H2	

El resultado final del Taller es una página web como se muestra en la **Figura 1** donde se pueden crear nuevas vistas personalizadas facilmente, agregar tablas (denominada Grid en Vaadin Flow) que tiene capacidades de funcionalidad para agregar, eliminar y actualizar datos de los **Estudiantes**.

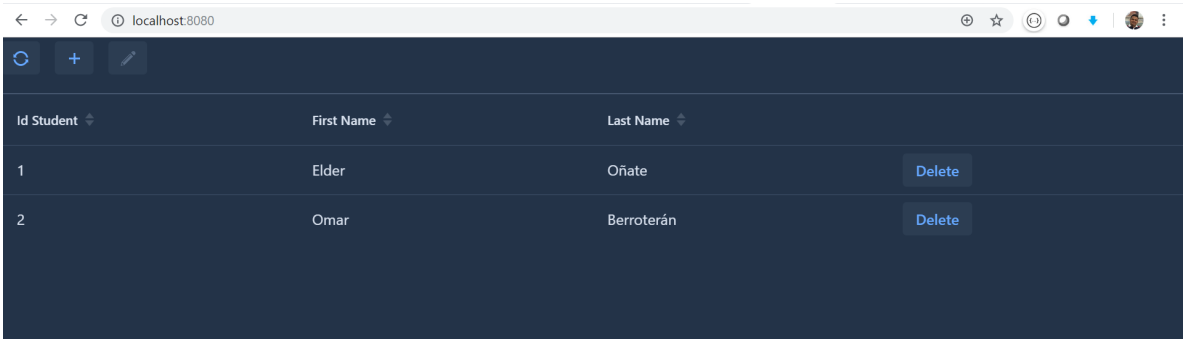

Figura 1

Características

- Estilos personalizados
- Instalación en escritorio y móvil
- Plantillas para CRUD.
- Plantillas 100% basadas en Java.

La forma más fácil de crear un nuevo proyecto en Vaadin es utilizar el iniciador **Project Base.**

¿En qué cosiste el Project Base? Es un proyecto listo para usarse y a su vez incluye la configuración, dependencias y un código de ejemplo. Para generar un nuevo proyecto

Vaadin nos direccionamos al siguiente link: Vaadin Start Spring y Seleccionamos la opción del **Project Base** como se muestra en la **Figura 2**

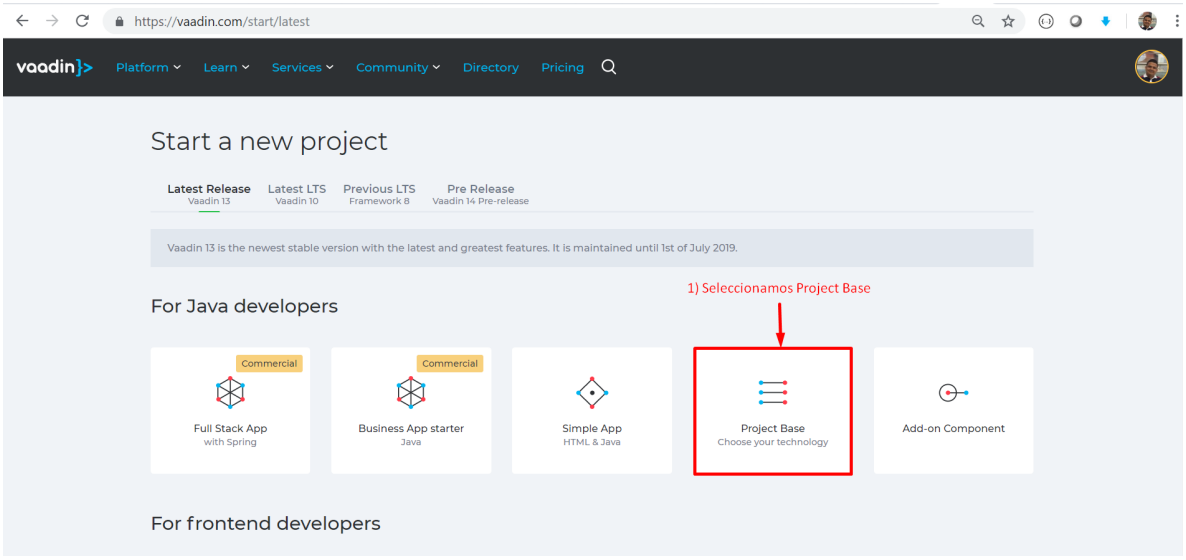

Figura 2

En el siguiente paso en el **Get Started**, seleccionamos la opción **Spring Boot,** introduzca:

ID de grupo:	com.mail.nombre
Nombre de la aplicación:	demoUMAG

y presionamos el botón Descarga como se muestra en la **Figura 3** y extraiga el archivo **demoUMAG.zip**. En el caso que no este activo, es posible que deba iniciar sesión en la plataforma de Vaadin.

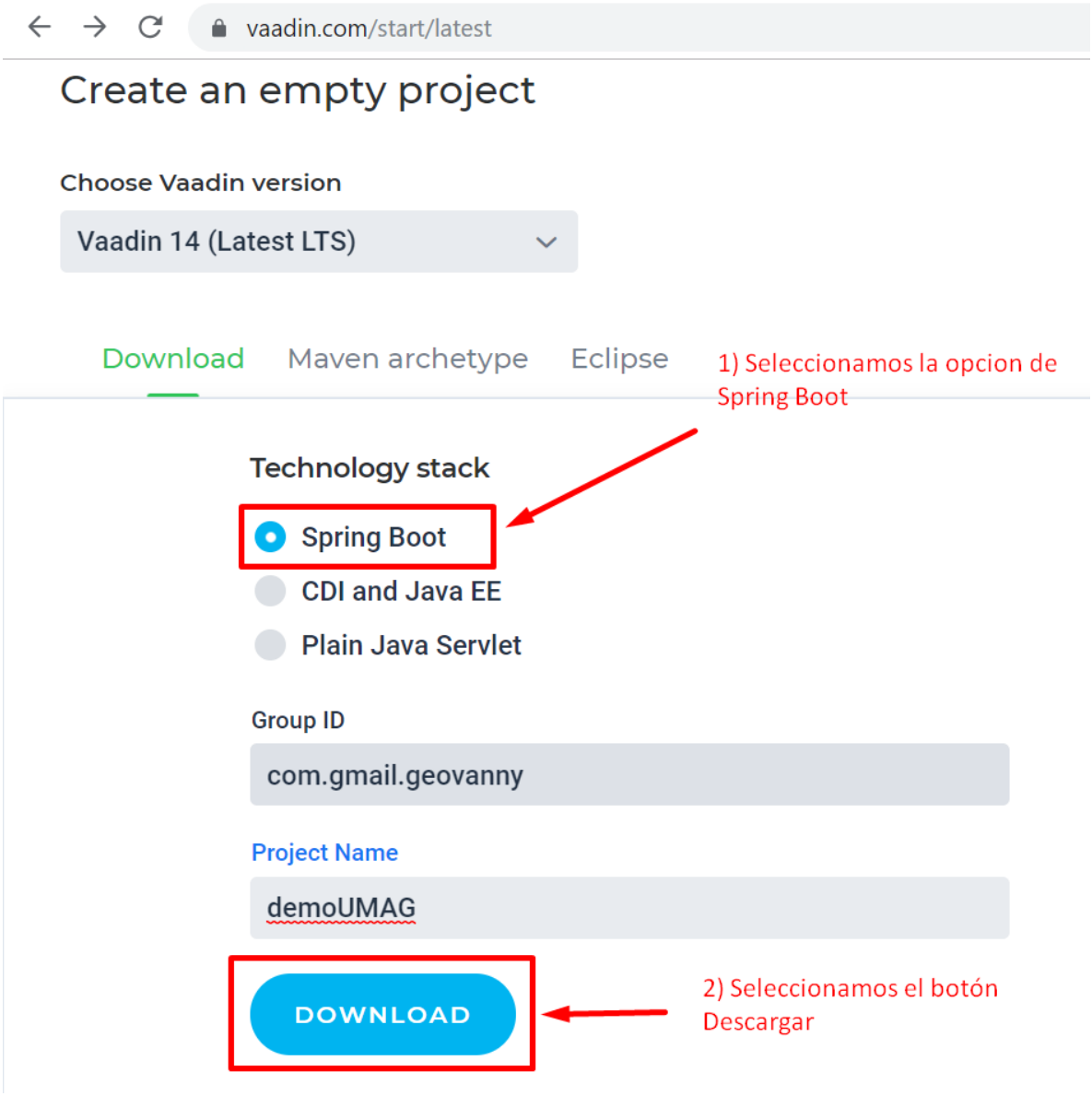

Figura 3

Abrir Proyecto:

Importar el proyecto generado en el IDE favorito, en esta ocasión se utilizará **Intellij IDEA,** como se muestra en la **Figura 4**

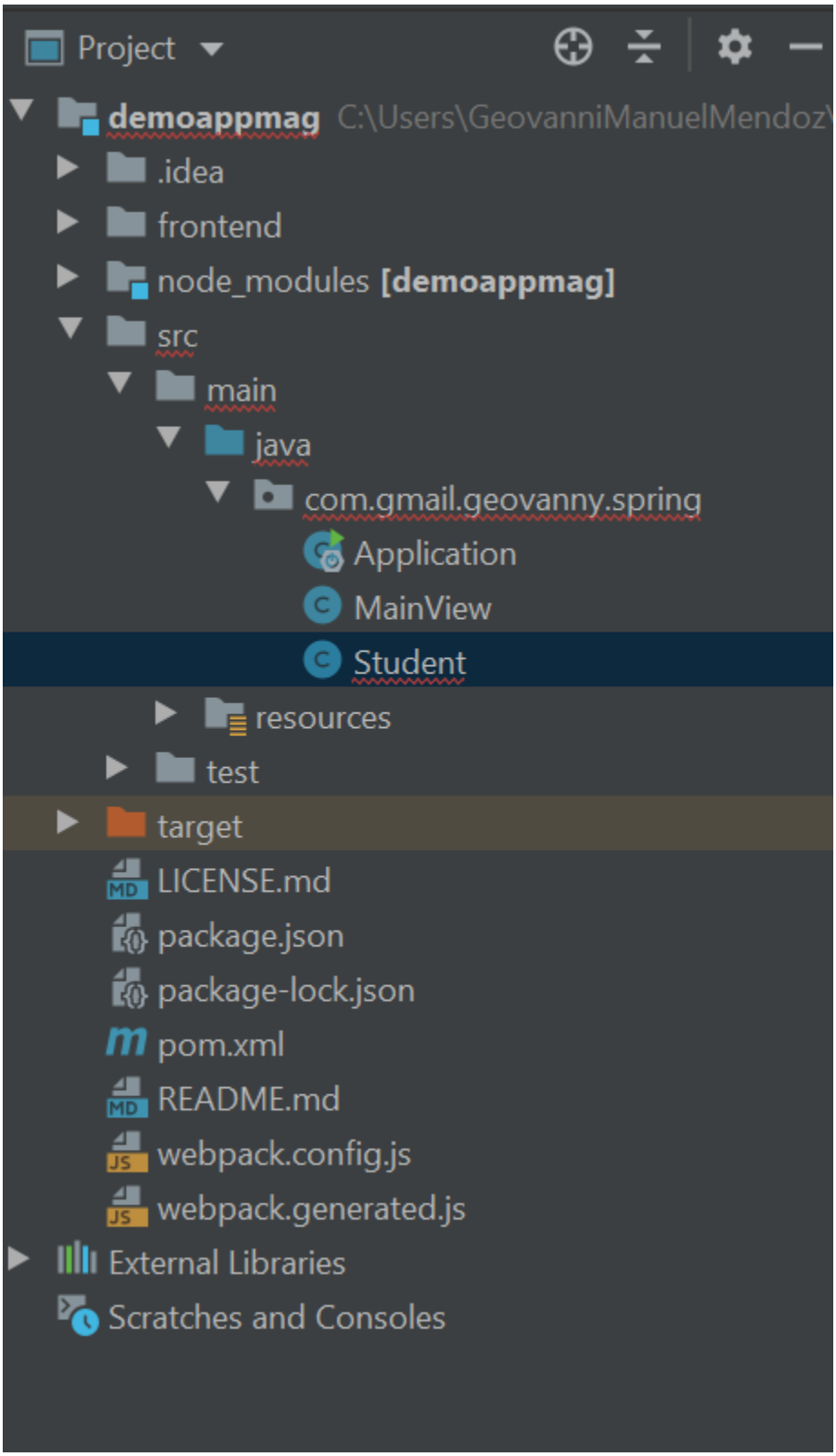
Project
demoappmag C:\Users\GeovanniManuelMendoz\
.idea
frontend
node_modules [demoappmag]
src
main
java
com.gmail.geovanny.spring
Application
MainView
Student
resources
test
target
LICENSE.md
package.json
package-lock.json
pom.xml
README.md
webpack.config.js
webpack.generated.js
External Libraries
Scratches and Consoles

Figura 4

El próximo paso es configurar el archivo **pom.xml**, para agregar las dependencias a utilizar en esta aplicación, como se puede observar en la **Figura 5**:

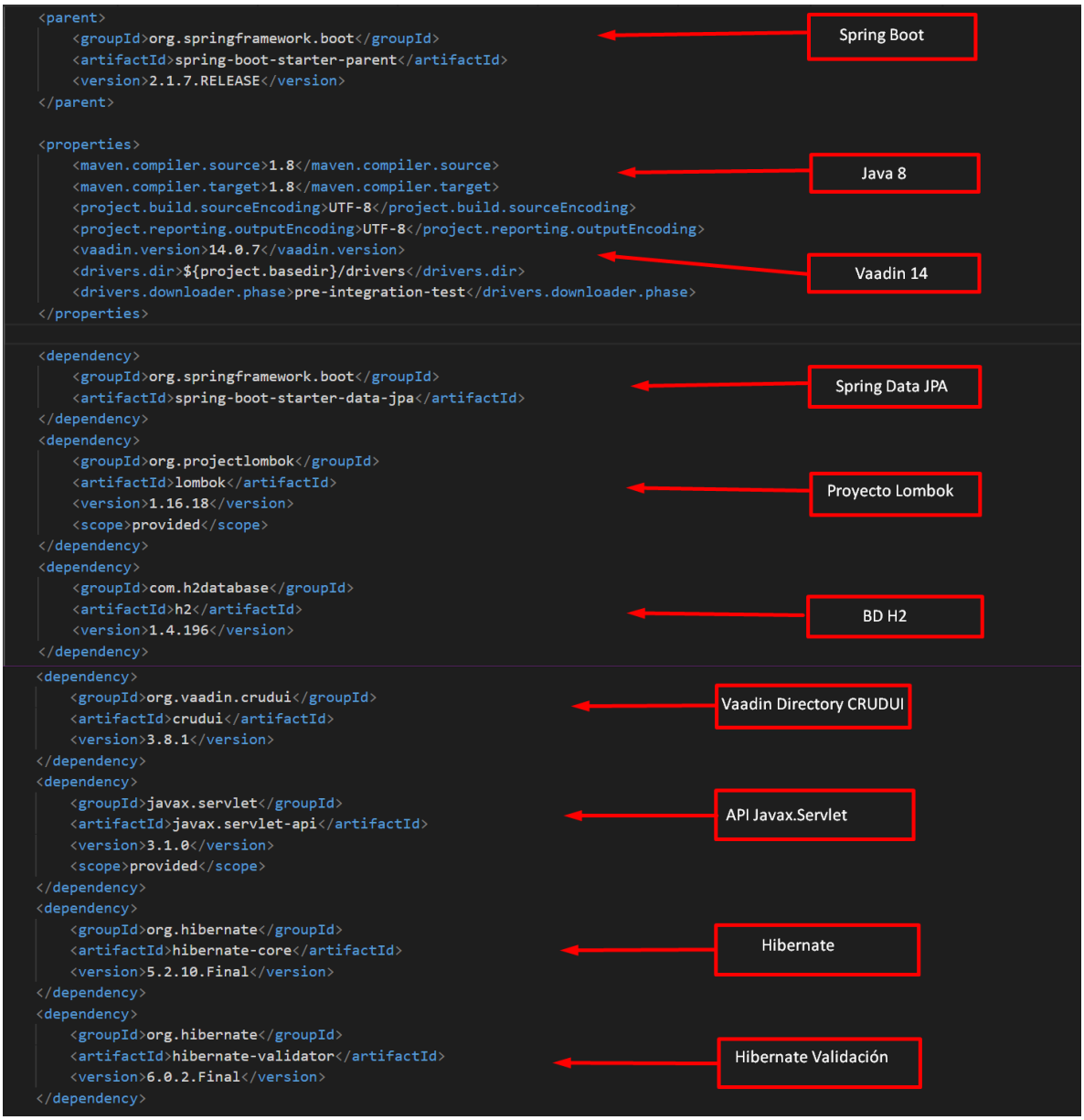

Figura 5

Crear objeto de entidad (que será mapeado en la base de datos).

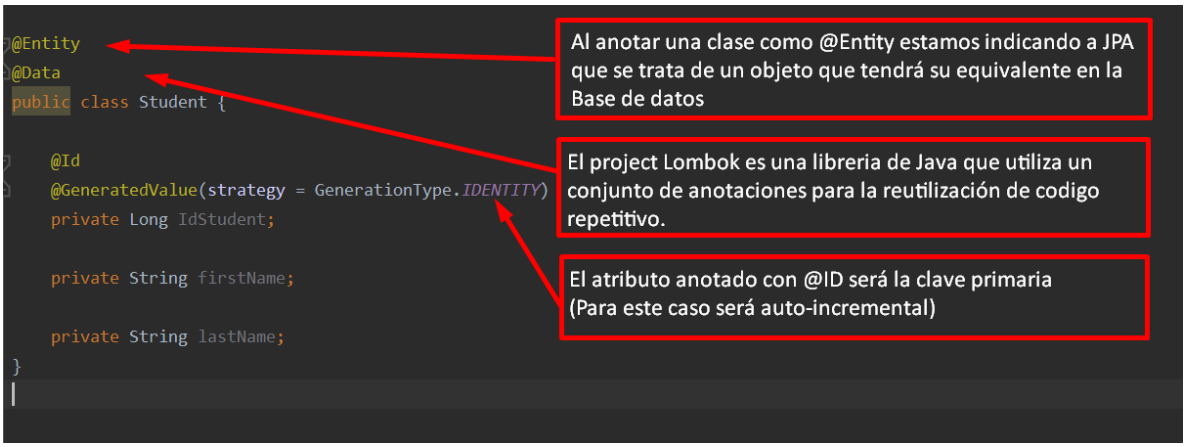

Figura 6

Configurar el archivo **application.properties,** adicionando el puerto y la conexión a la base de datos **H2** con su respectivo usuario **SA** (Sin contraseña para este ejemplo).

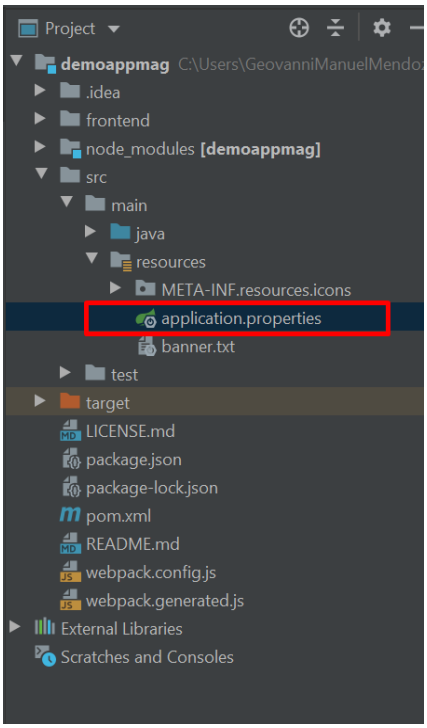

Figura 7

Figura 8

- **Creación Interfaz Student repositorio**

Con **Spring Data JPA** proporciona una interfaz CrudRepository para las operaciones CRUD, dándole funcionalidades a la clase entidad.

Ahora creamos la interfaz del repositorio extendiendo de **CrudRepository** como se puede observar en la **Figura 9.**

```
public interface IStudentRepository extends CrudRepository<Student, Long> {

}
```

Figura 9

Al extender de **CrudRepository** automáticamente dispondremos de los siguientes métodos:

- **save**(Student)
- **delete**(Student)
- **deleteBy**(Long)
- **find**(Student)
- **find**(Long)
- **findAll**()

- **Implementar CRUD usando Grid y Formulario**

Para realizar un buen diseño, utilizamos el componente **GRID** para listar los datos de los **Estudiantes** desde la clase **"Student"**, Estos nombres son iguales como se declaró en la clase "**Stundet"**. Por último creamos tres (3) métodos Listar**()** que se visualizara todo los datos de los **"Students"** como se puede observar la **Figura 10**.

```
@PWA(name = "Project Base for Vaadin Flow with Spring", shortName = "Project Base")
public class MainView extends Composite<VerticalLayout> {

    private Button refresh = new Button( text: "", VaadinIcon.REFRESH.create());
    private Button add = new Button( text: "", VaadinIcon.PLUS.create());
    private Button edit = new Button( text: "", VaadinIcon.PENCIL.create());

    private final IStudentRepository repository;
    private Grid<Student> grid = new Grid<>(Student.class);

    public MainView(IStudentRepository repository) {
        this.repository=repository;
        initLayout();
        initBehavior();
        refresh();
    }
```

1) Creación de los Botones

2) Creamos una instacia al GRID e inyectamos la interfaz al repositorio

3) Inicializamos la variable Repositorio y creamos los metodos para la operación

Figura 10

Creamos el método **initLayout()** donde se configura toda la parte del diseño de la página, agregando los nombres de la columna para configurar el **Grid** usando el método **setColumns** para mostrar las propiedades "idStudent", "firstName" y "lastName". Por último, la dimensión y la posición como se va a ver reflejada como se puede observar la figura 11.

En el método **initBehavior()** crearemos las operaciones CRUD de crear y actualizar solamente presionar click sobre el botón.

En el método **refresh()** como su nombre lo indica es para refresca el grid.

```
    private void initLayout() {
        HorizontalLayout header = new HorizontalLayout(refresh, add, edit);
        grid.setColumns("idStudent","firstName", "lastName");
        grid.setSizeFull();
        getContent().add(header, grid);
        getContent().expand(grid);
        getContent().setSizeFull();
        getContent().setMargin(false);
        getContent().setPadding(false);
    }

    private void initBehavior() {
        refresh.addClickListener(e -> refresh());
    }

    public void refresh() {
        grid.setItems((Collection<Student>) repository.findAll());
    }
```

Figura 11

Creamos un nuevo método **updateHeader** como se muestra en la **Figura 12** para habilitar o deshabilitar el botón de **editar** dependiendo del estado de selección en el **Grid**. Tiene sentido tener el botón de **editar** habilitado solo cuando hay una fila seleccionada. Necesitamos llamar a este método cuando actualizamos la lista y cuando cambia el valor seleccionado en el Grid.

```
private void updateHeader() {
    boolean selected = !grid.asSingleSelect().isEmpty();
    edit.setEnabled(selected);
}
```

Figura 12

- **Implementar las operaciones de Crear y Actualizar.**

La operación de creación del CRUD comienza cuando el usuario hace clic en cualquier botón Agregar o Actualizar. Implementamos el siguiente código de los metodos, como se puede observar en la **Figura 13**:

```
private void deleteClicked(Student student) {
    showRemoveDialog(student);
    refresh();
}

private void showAddDialog() {
    UserFormDialog dialog = new UserFormDialog( caption: "New Student", new Student());
    dialog.open();
}

private void showEditDialog() {
    UserFormDialog dialog = new UserFormDialog( caption: "Update Student", grid.asSingleSelect().getValue());
    dialog.open();
}


private void showRemoveDialog(Student student) {
    RemoveDialog dialog = new RemoveDialog(student);
    dialog.open();
}
```

Figura 13

Cuando se hace clic en cualquiera de los botones, mostramos un UserFormDialog. Para el botón de agregar, pasamos una nueva instancia de un estudiante. Para el botón de actualización, pasamos la instancia de un estudiante seleccionada en el **GRID**. Podemos implementar UserFormDialog como una clase interna dentro de Crud. Como se puede observar en la **Figura 14.**

```
private class UserFormDialog extends Dialog {

    private TextField firstName = new TextField( label: "First name");
    private TextField lastName = new TextField( label: "Last name");
    private Button cancel = new Button( text: "Cancel");
    private Button save = new Button( text: "Save", VaadinIcon.CHECK.create());

    public UserFormDialog(String caption, Student student) {
        initLayout(caption);
        initBehavior(student);
    }
```

Figura 14

Todos los campos de entrada en el formulario son miembros de la clase UserFormWindow. Por otra parte se agregan el diseño dentro del método **initLayout** como se puede observar en la **Figura 15**.

```
private void initLayout(String caption) {
    save.getElement().setAttribute("theme", "primary");
    HorizontalLayout buttons = new HorizontalLayout(cancel, save);
    buttons.setSpacing(true);
    firstName.setRequiredIndicatorVisible(true);
    FormLayout formLayout = new FormLayout(new H2(caption), firstName, lastName );
    VerticalLayout layout = new VerticalLayout(formLayout, buttons);
    layout.setAlignSelf(FlexComponent.Alignment.END, buttons);
    add(layout);
}
```

Figura 15

El método **initBehaviour** debe configurar el enlace de datos entre la instancia del usuario y los campos de entrada. También se debe agregar comportamiento a los botones de cancelar y guardar.

Necesitamos **data-binding**. En el framework de Vaadin, eso usualmente significa usar un **binder**.

Por lo tanto algo que no podemos dejar de pasar, suceden en el código que esta dentro del metodo **initBehaviour** como se puede observar en la **Figura 16** es lo siguiente:

- Todos los campos Java que también son campos de *entrada* en la clase UserFormWindow están vinculados a los campos Java en la clase **Student** (con la invocacion **bindIntanceFields**);
- Todos los valores en los campos Java de la clase Student se configuran en los campos de entrada correspondientes en la clase UserFormWindow (con el **readBean**).

```
private void initBehavior(Student student) {
    BeanValidationBinder<Student> binder = new BeanValidationBinder<>(Student.class);
    binder.bindInstanceFields( objectWithMemberFields: this);
    binder.readBean(student);
```

Figura 16

Finalmente, en el siguiente código se adiciona el comportamiento a los botones cancelar y guardar como se puede observar la **Figura 17**:

```
        cancel.addClickListener(e -> close());
        save.addClickListener(e -> {
            try {
                binder.validate();
                binder.writeBean(student);
                repository.save(student);
                close();
                refresh();
                Notification.show("Student saved");
            } catch (ValidationException ex) {
                Notification.show("Please fix the errors and try again");
            }
        });
    }
```

Figura 17

Al escuchar en el botón de **cancelar** solo tiene que llamar a **Window.close()** (heredado).

Al escuchar en el botón de **guardar** llama a **writeBean** para escribir los valores en los campos de entrada en la instancia de **Student**.

Nota: El **writeBean** lanza una **ValidationException**. Por lo tanto, no hay validaciones en este momento. Agregar las restricciones de Validación de JavaBean que tenemos en la clase de **Student** es tan simple como cambiar la implementación de **Binder**: como se observa en la **Figura 17.1**

```
BeanValidationBinder<Student> binder = new BeanValidationBinder<>(Student.class);
```

Figura 17.1

Así queda toda la implementación del código, se puede observar en la **Figura 18**

```
private void initBehavior(Student student) {
    BeanValidationBinder<Student> binder = new BeanValidationBinder<>(Student.class);
    binder.bindInstanceFields( objectWithMemberFields: this);
    binder.readBean(student);
    cancel.addClickListener(e -> close());
    save.addClickListener(e -> {
        try {
            binder.validate();
            binder.writeBean(student);
            repository.save(student);
            close();
            refresh();
            Notification.show("Student saved");
        } catch (ValidationException ex) {
            Notification.show("Please fix the errors and try again");
        }
    });
}
```

Figura 18

- **Implementar la operación Eliminar.**

Implementemos la operación CRUD el botón **eliminar**, utilizando un enfoque diferente. En lugar de simplemente agregar un solo botón para la operación, agregaremos un botón de eliminar en cada fila del **GRID**. La forma más sencilla de agregar un componente de UI dentro de una **GRID** es mediante el método **addComponentColumn** como se puede observar en la **Figura 19.**

```
private void initLayout() {
    HorizontalLayout header = new HorizontalLayout(refresh, add, edit);
    grid.setColumns("idStudent","firstName", "lastName");

    grid.addComponentColumn(student -> new Button( text: "Delete", e -> deleteClicked(student)));

    grid.setSizeFull();
    getContent().add(header, grid);
    getContent().expand(grid);
    getContent().setSizeFull();
    getContent().setMargin(false);
    getContent().setPadding(false);
}
```

Figura 19

Implementación de la caja de dialogo eliminar como se muestra en la **Figura 20** y **Figura 21**

```
private class RemoveDialog extends Dialog {
    private Button cancel = new Button("Cancel");
    private Button delete = new Button("Delete", VaadinIcon.TRASH.create());

    public RemoveDialog(Customer customer) {
        initLayout(customer);
        initBehavior(customer);
    }
```

Figura 20

```
    private void initLayout(Student student) {
        Span span = new Span("Do you really want to delete the user " + student.getFirstName() + " " + student.getLastName() + "?");
        delete.getElement().setAttribute("theme", "error");
        HorizontalLayout buttons = new HorizontalLayout(cancel, delete);
        VerticalLayout layout = new VerticalLayout(new H2("Confirm"), span, buttons);
        layout.setAlignSelf(FlexComponent.Alignment.END, buttons);
        add(layout);
    }

    private void initBehavior(Student student) {
        cancel.addClickListener(e -> close());
        delete.addClickListener(e -> {
            repository.deleteById(student.getIdStudent());
            refresh();
            close();
        });
    }
```

Figura 21

- **EJECUCION DEL PROYECTO**

Para esta demo existen dos (2) formas de ejecutar el proyecto una directamente por el IDE y la segunda con el siguiente comando: **mvn spring-boot:run**