

Proyecto RUDA: Registro unificado de actividades

Franco Ganga

9 de diciembre de 2019

Índice

1. Introducción	2
1.1. Resumen	2
1.2. Objetivos	3
1.2.1. Objetivos específicos	4
1.3. Tareas	4
1.3.1. Modelado conceptual de datos	4
1.3.2. Creación de la base de datos	4
1.3.3. Definición de ABMs	6
1.3.4. Desarrollo de API	8
1.3.5. Instalación y configuración de librerías	8
1.4. Cronograma	10
2. Desarrollo	10
2.1. Introducción Symfony	10
2.2. Modelado	13
2.2.1. Introducción a Doctrine	13
2.2.2. Actividad	14

2.2.3.	Miembro de Proyecto	16
2.2.4.	Proyecto	17
2.2.5.	Rol de Proyecto	18
2.3.	Definición de ABMs - Clases Admin	18
2.3.1.	ABMs Compuestas	20
2.3.2.	ABMs Simples y admins <i>hijos</i>	24
2.4.	Librerías	27
2.4.1.	Instalación y Configuración de Sonata-User	27
2.4.2.	API-Platform	36
2.4.3.	Guzzle y EightPointsGuzzle	38
2.5.	Vista de Persona	39
2.5.1.	Introducción Twig	39
2.5.2.	Estilo	39
2.5.3.	Template	41
2.5.4.	Obtención de rutas	42
2.6.	Integración	44
2.6.1.	Eventos	44
2.6.2.	Implementación	46
2.6.3.	Rendimiento	46
2.6.4.	Inserción de Personas	47
3.	Conclusión	47
4.	Reflexión	48

1. Introducción

1.1. Resumen

El presente trabajo expondrá el desarrollo de un proyecto a realizarse en la Universidad Nacional Arturo Jauretche (UNAJ), que consistirá en la elaboración

de un sistema informático para el registro de datos. El mismo tendrá como finalidad la consolidación de conocimientos adoptados en la carrera y requerirá de la capacidad de análisis e investigación para adaptarse a los estándares y herramientas utilizadas por la Dirección de Informática de la institución.

1.2. Objetivos

El objetivo general de este proyecto es el desarrollo de un sistema que permita el registro de datos sobre las actividades realizadas por los integrantes de la UNAJ. El mismo almacenará datos de proyectos de investigación, publicaciones, voluntariados, etc.

Actualmente la universidad no cuenta con una plataforma establecida para el registro de esta información, sólo se dispone de los datos de las personas que integran la institución. Estos datos están distribuidos en dos sistemas:

- **SIU-Mapuche:** Sistema de RRHH de la Universidad, cuenta con toda la información relevante al personal (docente y no-docente) y a los cargos.
- **SIU-Guaraní:** Sistema Académico de la Universidad, aquí es donde se encuentran los datos de alumnos, institutos, materias, comisiones, carreras, etc.

El uso de estos sistemas, como fuente de información y su integración con la plataforma RUDA, permitirá saber en qué actividades está o estuvo involucrada determinada persona en la UNAJ. Se desarrollará una interfaz web para el acceso a los datos, para lo que será necesario relacionar la información en los sistemas fuente con la de RUDA. Además, se implementará un servicio REST, que permitirá acceder fácilmente a los datos.

Este proyecto está pensado para ser utilizado por las siguientes áreas:

- Centro de Política Educativa
- Centro de Política y Territorio
- Secretaría Económico Financiera

1.2.1. Objetivos específicos

Para llevar adelante el mencionado proyecto, se pretenden alcanzar los siguientes objetivos específicos:

- **Modelar la estructura de los datos a almacenar en el sistema.**
- **Desarrollar una interfaz de usuario para la interacción con los datos:** aquí se presentará la información de cada persona y sus actividades.
- **Implementar la integración de los sistemas:** se deberá trabajar en un método para obtener los datos externos e incorporarlos al sistema RUDA.
- **Desarrollar una API REST para la obtención de los datos:** se desarrollará un servicio web que facilite la manipulación de datos a través de internet. Esto brindará a la universidad la capacidad de reutilizar esta fuente de datos en futuros proyectos.

1.3. Tareas

A continuación, se detallarán las tareas por ejecutar en vistas al cumplimiento de los objetivos previamente definidos:

1.3.1. Modelado conceptual de datos

Se definirá el modelado de datos a implementar basándose en los datos que se requieren almacenar y cómo están relacionados.

1.3.2. Creación de la base de datos

Entidades a desarrollar que almacenarán datos sobre la duración de cada actividad:

- Cargos de Autoridad

- Consejeros Superiores
- Miembros y Roles en las Comisiones del Consejo Superior
- Asambleístas
- Consejeros Consultivos
- Responsables de Áreas
- Directores/Subdirectores de Institutos
- Directores/Subdirectores de Carreras
- Proyectos de Investigación
- Proyectos de Extensión
- Programas
- Actividades de Divulgación
- Publicaciones
- Cursos de Extensión
- Voluntariados
- Vinculadores
- Programas
- Becas
- Pasantías
- Movilidad RTF
- Movilidad Conurbano Sur
- Prácticas Profesionales Supervisadas

1.3.3. Definición de ABMs

Una ABM permite al usuario interactuar con los datos, por esto es que se deberá definir cada una en particular:

Relacionadas a la política de la institución:

- Rector
- Vicerrector
- Asambleísta
- Consejero Superior
- Responsable de Área
- Coordinador de Materia
- Miembro de Consejo Consultivo
- Director de Instituto
- Subdirector de Instituto
- Miembro de Comisión de Consejo Superior
- Director de Carrera
- Subdirector de Carrera
- Instituto
- Materia
- Carrera
- Consejo Consultivo

- Área
- Resolución Administrativa
- Comisión de Consejo Superior
- Rol de Comisión de Consejo Superior

No relacionadas a la política de la institución:

- Práctica Profesional Supervisada
- Participación en Práctica Profesional Supervisada
- Actividad de Divulgación
- Curso de Extensión
- Participación en Curso de Extensión
- Pasantía
- Participación en Pasantía
- Programa
- Miembro de Programa
- Movilidad
- Participación en Movilidad
- Actividades de Vinculación
- Becas
- Participación en Beca

- Voluntariado
- Participación en Voluntariado
- Proyecto
- Miembro de Proyecto
- Rol de Proyecto

1.3.4. Desarrollo de API

Para este desarrollo se generarán rutas mediante las cuales se podrá acceder a la misma información detallada en la tarea anterior.

- **Implementación de nodos para obtener cada dato:** cada dato debe ser accesible mediante una petición http.
- **Implementación de autenticación del servicio**

1.3.5. Instalación y configuración de librerías

Se deberá configurar cada librería de acuerdo a las necesidades del sistema. A continuación se listan las librerías a utilizar para el desarrollo:

- Sonata-admin
- Sonata-user
- FOSUser
- Doctrine
- Data-Fixtures
- Faker

- Api-Platform
- Guzzle y Eightpoints

1.4. Cronograma

Tareas	Cuatrimestre
Entidad y Admin: Consejero Superior	Primer Cuatrimestre
Entidad y Admin: Proyecto	Primer Cuatrimestre
Entidad y Admin: Miembro de Proyecto	Primer Cuatrimestre
Instalación: Sonata User	Primer Cuatrimestre
Entidad y Admin: Rol de proyecto	Segundo Cuatrimestre
Pantalla de Persona	Segundo Cuatrimestre
Instalación API-Platform	Segundo Cuatrimestre
Entidades y admins restantes	Segundo Cuatrimestre
Integración con mapuche	Segundo Cuatrimestre

2. Desarrollo

2.1. Introducción Symfony

Symfony es un *framework* PHP de código abierto para desarrollar aplicaciones web. Originalmente fue concebido por la agencia interactiva SensioLabs para el desarrollo de sitios web para sus propios clientes. Symfony se publicó en 2005 bajo la licencia MIT Open Source y hoy se encuentra entre los principales *frameworks* disponibles para el desarrollo de PHP. («About Symfony Project», sf)

Un proyecto Symfony está conformado por varios componentes que proveen funciones básicas para una aplicación web. Entender el funcionamiento de cada uno de ellos es necesario para una buena implementación. Para el desarrollo del presente proyecto se utilizaron los siguientes componentes de Symfony:

- **Asset:** administra la generación de URL y el versionado de hojas de estilo css, archivos JavaScript e imágenes.
- **Console:** permite crear comandos para usar en consola. Bastante útil para tareas recurrentes.
- **Dotenv:** administra las variables de entorno de la aplicación.
- **Expression Language:** permite utilizar expresiones dentro de archivos de configuración para obtener lógica más compleja.
- **Flex:** es un componente que facilita la integración de paquetes de terceros a través de lo que se denomina recetas Symfony. Estas recetas consisten en un conjunto de instrucciones automatizadas.
- **Form:** permite crear, procesar y reutilizar formularios.
- **Framework:** define la configuración principal del framework.
- **Monologbundle:** integra la librería monolog con symfony para el registro de mensajes.
- **ORM Pack:** es la librería encargada del mapeo objeto-relacional.
- **Process:** librería utilizada para la ejecución de subprocessos. Resuelve problemas relacionados con la diferencia entre sistemas operativos y, además, provee una ejecución segura.
- **Security:** componente de seguridad de Symfony. Se encarga de definir el control de acceso, sistemas de autenticación y, además, de establecer los proveedores de usuarios.
- **Serializer:** paquete de Symfony que se encarga de transformar un objeto en un formato adecuado para la transmisión de datos. Ej: JSON.

- **Swiftmailer:** permite el envío de emails a través de un servidor propio o de terceros.
- **Translation:** permite definir textos en la aplicación para traducción al idioma local del usuario. Este proceso es comúnmente llamado internacionalización.
- **Twig:** Motor de templates preferido por Symfony, permite renderizar contenido HTML de manera fácil, organizada y segura.
- **Validator:** componente encargado de la validación de datos. Weblink: incrementa la performance de la aplicación al utilizar HTTP2 y funciones de precarga en navegadores modernos.
- **Yaml:** se encarga de convertir archivos YAML en arrays PHP. Gran parte de la configuración de Symfony se encuentra definida en este formato.

2.2. Modelado

En primer lugar, se definieron relaciones básicas de acuerdo a los requerimientos del sistema para poder crear las clases admin y definir las entidades como recursos *API-Platform*.

2.2.1. Introducción a Doctrine

Con Doctrine, cada objeto PHP que se requiera persistir a la base de datos, necesita estar definido en un archivo PHP que contiene información referente a tipos de dato y relaciones entre entidades. A este archivo se lo denomina entidad.

```
<?php
/** @Entity */
class Message
{
    /** @Column(type="integer") */
    private $id;
    /** @Column(length=140) */
    private $text;
    /** @Column(type="datetime", name="posted_at") */
    private $postedAt;
}
```

Figura 1: Ejemplo básico de una entidad.

Fuente: Recuperado de www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/basic-mapping.html

Una relación en **Doctrine** está formado por dos entidades. Una de ellas actúa como el lado propietario de la relación y la otra como el inverso. El lado propietario de la relación es aquel en el que **Doctrine** verifica si hubo cambios. Existen

dos tipos de mapeo de relaciones, bi-direccionales y uni-direccionales. Una relación bi-direccional permite que ambos lados de la relación puedan accederse entre sí. En cambio, una relación uni-direccional sólo puede accederse a través del lado propietario. Al trabajar con relaciones, se debe tener en cuenta la manera en que Doctrine comprueba por cambios en los datos, ya que cambios persistidos en el lado inverso de la relación serán ignorados por el ORM al momento de actualizar información a la base de datos.

2.2.2. Actividad

Se modeló el siguiente esquema de acuerdo a la información recibida por parte del Área de Informática de la universidad. Además, se pensó en utilizar herencia para la definición de las actividades, de esta manera cada cargo o actividad en particular heredaría toda característica común de una entidad padre.

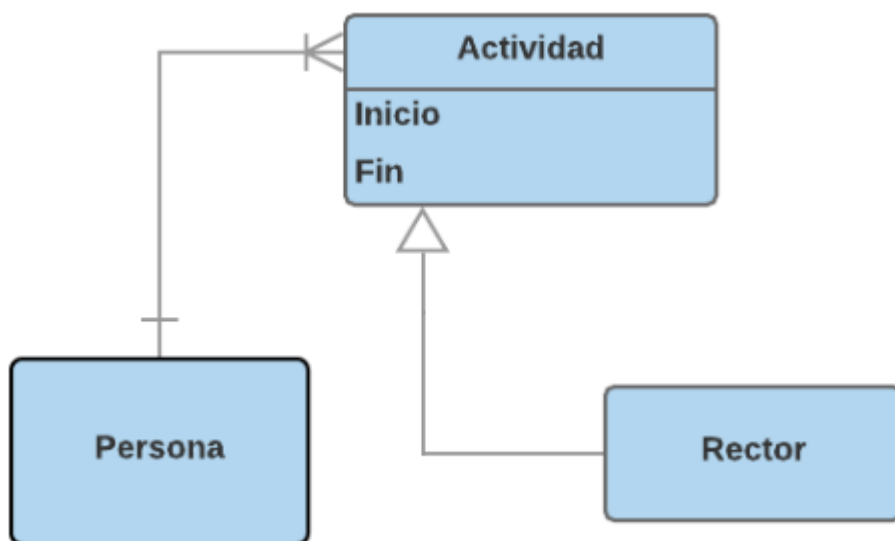


Figura 2: Ejemplo de la definición de un cargo

Fuente: Elaboración propia

El primer paso realizado para la definición de cada cargo o actividad fue la creación de la entidad *Actividad*, la cual contendrá datos de la persona relacio-

nada y el periodo de desarrollo de la actividad o cargo. Se agregó la función de Soft delete o borrado lógico, la que permite que, al borrar una actividad desde la aplicación web, la misma es ocultada del usuario y no borrada de la base de datos. Esta función posibilita almacenar datos históricos del sistema. Además, se agregó la función de Time stamps o etiquetas de tiempo, la cual hace posible el seguimiento de las fechas de creación y actualización de cada actividad.

Se definió el mapeo de esta entidad en Doctrine mediante herencia de clase, una estrategia en la que cada clase en la jerarquía es mapeada a varias tablas: la propia y las de todas las clases padre. La tabla de una clase hija es vinculada a la del padre a través de una clave foránea.

Doctrine implementa esta estrategia a través del uso de una columna denominada discriminator en la tabla más alta en la jerarquía. Esta es la mejor manera de lograr consultas polimórficas con herencia de clase. («Inheritance Mapping», sf)

Esta columna identifica el tipo de entidad, por ejemplo: una fila con un valor de “director instituto” significa que es una actividad del tipo DirectorInstituto. Si no se provee el mapeo correspondiente, doctrine lo generará automáticamente utilizando el nombre de cada clase entidad en minúscula. («Inheritance Mapping», sf)

id	persona_id ▼ 1	inicio	fin	discr
4c90031a-f78a-11e9-ba9e-74d4350a020b	14	2019-10-26	2019-10-15	coordinador_materia
41660746-f78a-11e9-ba9e-74d4350a020b	13	2019-10-26	2021-08-07	coordinador_materia
5ab2b335-f77d-11e9-ba9e-74d4350a020b	11	2019-10-26	2019-10-31	miembro_proyecto
5ab2bdca-f77d-11e9-ba9e-74d4350a020b	11	2019-10-26	2020-06-18	miembro_proyecto
5ab2c38d-f77d-11e9-ba9e-74d4350a020b	11	2019-10-26	2020-07-02	coordinador_materia
5ab2ceb3-f77d-11e9-ba9e-74d4350a020b	11	2019-10-26	2021-01-02	director_carrera

Figura 3: Columna **discr** utilizada como discriminator

Fuente: Elaboración propia, impresión de pantalla del código fuente.

Se agregó la correspondiente metadata de la herencia en *Actividad* y se optó por proporcionar el mapeo de la columna discriminator:

```
@ORM\Entity
@ORM\InheritanceType("JOINED")
@ORM\DiscriminatorColumn(name="discr", type="string")
@ORM\DiscriminatorMap({"director_instituto" = "DirectorInstituto",
"asambleista" = "Asambleista", "consejero_superior" = "ConsejeroSuperior",
"miembro_proyecto" = "MiembroProyecto", "director_carrera" = "DirectorCarrera",
"coordinador_materia" = "CoordinadorMateria"})
```

Figura 4: Mapeo de la columna **discriminator**

Fuente: Elaboración propia, impresión de pantalla del código fuente.

2.2.3. Miembro de Proyecto

Esta entidad es la que almacenará datos acerca de la acción de formar parte de un proyecto. Cada dato de la columna **MiembroProyecto** representará la participación de una persona a un proyecto. Se mapeó la relación entre *miembros* y *proyectos* con una cardinalidad de 1-n y, dado que en este tipo de relaciones el lado n toma la clave foránea, el lado propietario termina siendo *miembros de pro-*

yecto. Con *roles* sucede lo mismo, ya que cada instancia de miembro debe poseer un solo rol.

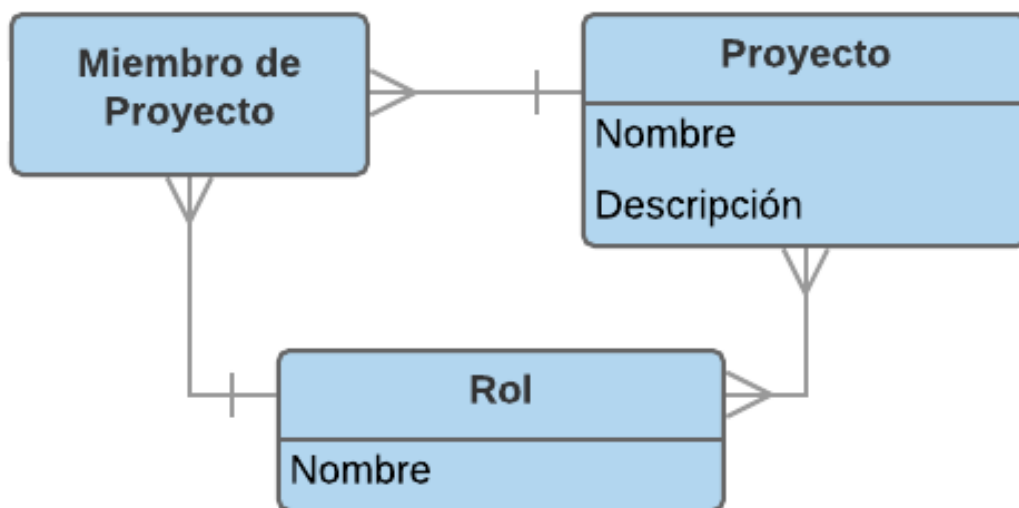


Figura 5: Diagrama que representa la relación entre miembro, rol y proyecto

Fuente: Elaboración propia utilizando una herramienta online de elaboración de diagramas.

2.2.4. Proyecto

En el sistema se necesita representar dos tipos de proyectos, de extensión y de investigación. Por ende, se decidió agregar **herencia** para la definición de cada proyecto. Esto evita tener que crear una nueva entidad para representar los miembros del otro proyecto. De la misma forma que con Actividad se utilizó herencia de clase.

En cuanto a sus relaciones, posee la relación con miembros desde el punto de vista de Proyecto. Con una cardinalidad de n-1 y siendo **Proyecto** el lado inverso de la relación.

Por otro lado, también se definió su relación con **Roles de Proyecto** con una cardinalidad de n-n.

```

/**
 * @ORM\Entity(repositoryClass="App\Repository\MiembroProyectoRepository")
 */
class MiembroProyecto extends Actividad
{
    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\Proyecto", inversedBy="miembros")
     */
    private $proyecto;

    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\RolProyecto", inversedBy="miembros")
     */
    private $rol;
}

```

Figura 6: Metadata de mapeo de relaciones de miembro de proyecto.

Fuente: Elaboración propia.

2.2.5. Rol de Proyecto

Esta entidad representa un rol de proyecto, a cada instancia de miembro se le puede asignar un rol a ejercer en el Proyecto.

Se estableció la relación de roles del tipo n-n, es decir, con una cardinalidad de muchos a muchos; ya que un proyecto puede tener muchos roles y además un rol puede tener muchos proyectos. Como la entidad de roles se actualizará a partir de los proyectos, se definió a Proyecto como dueño o propietario de la relación. En el caso de una relación de este tipo el lado propietario es aquel que contiene el parámetro **inversedBy** en su definición. («Inheritance Mapping», sf)

2.3. Definición de ABMs - Clases Admin

Para definir un ABM, es decir, una interfaz capaz de manejar el alta, baja y modificación de registros es necesario una entidad. Por este motivo, para obtener una idea general del funcionamiento de Sonata-Admin, se decidió definir algunas entidades y generar una clase admin. En sonata admin, una clase admin es aquella que, dada una entidad, permite añadir un servicio en la plataforma web sonata que se hace cargo de cada una de estas funciones.

Para crear un admin es necesario extender de la clase **AbstractAdmin** que provee sonata y, mediante métodos heredados de la misma, configurar la manera en que se muestra la información. Algunos de estos métodos son:

- *configureFormFields*: este método define los campos a mostrarse durante la acción de crear y editar.
- *configureListFields*: define los campos a mostrar durante la acción de listar datos.
- *configureShowFields*: establece la información a mostrar durante la acción de ver una entrada.

Además de lo expresado, una clase admin en **Sonata** permite:

- Validar información
- Agregar acciones de acuerdo a eventos de cada entidad
- Establecer jerarquías entre clases admin
- Crear un menú de forma fácil en las vistas del admin
- Establecer filtros de búsqueda.

Éstas son las funcionalidades que se utilizaron en este proyecto, sin embargo, sonata cuenta aún con más opciones de configuración.

Una clase admin contiene una referencia a la entidad base del mismo a través de un objeto denominado **subject**, el mismo es utilizado para realizar cada operación de alta, baja y modificación de la entidad.

Para configurar las diferentes secciones del admin, es necesario especificar la información que se desea incluir, esto es posible utilizando el nombre del campo o un método que otorgue acceso a la propiedad en cuestión. Si se desea acceder a asociaciones, (datos que representan la relación de una entidad) se puede hacerlo

mediante una notación de puntos, por ejemplo: si se tiene una instancia de **Actividad**, la misma tendrá una asociación con una entidad persona. Por lo tanto, se puede acceder al nombre de una persona referenciándola como “persona.nombre”.

```
protected function configureShowFields(ShowMapper $showMapper): void
{
    $showMapper
        ->add('persona.nombre')
        ->add('persona.apellido')
        ;
}
```

Figura 7: Ejemplo de definición de campos a mostrar durante la creación/edición.

Fuente: Elaboración propia.

2.3.1. ABMs Compuestas

En el sistema se separaron las ABM en dos tipos, en primer lugar se tienen aquellas que representan una relacion de 1-n con otros elementos del sistema, es decir, actividades que pueden relizarse por muchas personas; en segundo lugar están las ABMs de actividades que son realizadas por sólo una persona.

En el primer caso, la implementación se hizo utilizando admins *hijos*, una característica de **Sonata** que permite definir una jerarquía de clases admin que represente un elemento *padre* compuesto por uno o muchos elementos *hijos*.

Cuando se asigna un admin como *hijo* se obtienen rutas anidadas, de forma que se puede acceder a acciones del *hijo* desde la vista del padre. Ej: Una ruta para acceder a un miembro en particular sería de la forma:

/miembroproyecto/id/(show | edit).

Cuando se accede a los miembros de cada proyecto se obtienen rutas de la forma:

**/proyecto/id/miembroproyecto/id/(show | edit),
/proyecto/id/miembroproyecto/list**

A continuación se listan las ABM implementadas de esta forma:

- Proyecto de Investigación
- Proyecto de Extensión
- Rol de Proyectos
- Comisión de Consejo Superior
- Práctica Profesional Supervisada
- Actividad de Divulgación
- Proyecto de Extensión
- Curso de Extensión
- Voluntariado
- Programa

Implementación de una ABM Compuesta: Proyecto

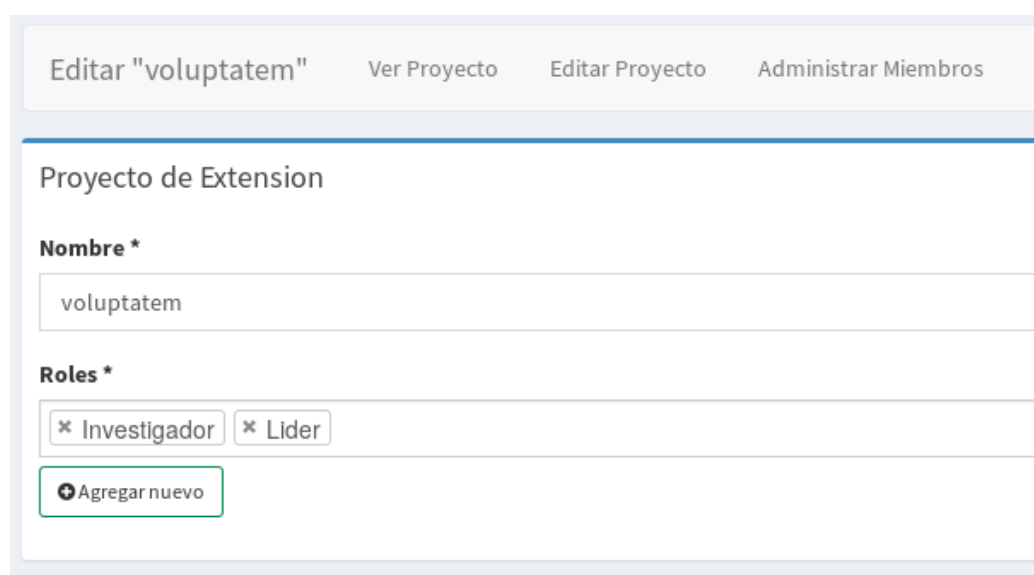
En el admin de Proyecto se agregaron los siguientes campos en cada acción:

- Lista: nombre del proyecto.
- Ver: nombre del proyecto
- Inserción: nombre y roles de proyecto.

Para asignar un admin como *hijo* se debe configurar el servicio padre a través de un método denominado **addChild**. Como parámetros requiere el servicio del admin a actuar de *hijo* y el campo mediante el cual está relacionado al *padre*.

Al agregar un admin como *hijo* se generan las rutas anidadas (sección: 2.3.1), pero no se tiene una forma de acceder a ellas desde la aplicación web. Una buena forma de solucionar este problema es, según la documentación de **Sonata**, agregar un menú en la barra superior de la página que contenga botones mediante los cuales acceder a estas rutas.(«Create child admins», sf)

Para crear un proyecto se deben proporcionar los datos de nombre y roles. Los roles de proyecto están representados en la entidad en forma de Colección, por lo tanto, se utilizó un tipo de formulario que permite la elección de opciones múltiples y, además, permite agregar nuevas instancias de rol (figura 8).



The screenshot shows a web application interface for editing a project. At the top, there is a navigation bar with four buttons: "Editar 'voluptatem'", "Ver Proyecto", "Editar Proyecto", and "Administrar Miembros". Below this, the main content area is titled "Proyecto de Extension". It contains two main sections: "Nombre *" and "Roles *". The "Nombre *" section has a text input field with the value "voluptatem". The "Roles *" section has a multi-select dropdown menu with two selected options: "x Investigador" and "x Lider". Below the dropdown menu, there is a button labeled "Agregar nuevo" with a plus icon.

Figura 8: Admin: Acción de editar Proyectos.

Fuente: Elaboración propia, captura de pantalla de la aplicación web.

Miembro de Proyecto

Para la definición del admin de miembros de proyecto se lo estableció como un *hijo* del admin Proyecto, esto es debido a que tiene sentido desde el punto de vista de la relación que forman. Un proyecto estará integrado por muchos miembros, por lo tanto, al asignar a miembros de proyecto como admin *hijo* de Proyecto permite administrar los miembros desde la interfaz de proyectos. Al fin y al cabo, la clase admin de miembros no tiene sentido por sí sola.

```
admin.proyecto_extension:
  class: App\Admin\ProyectoExtensionAdmin
  arguments: [~, App\Entity\ProyectoExtension, ~]
  tags:
    - { name: sonata.admin, manager_type: orm, group:
public: true
calls:
  - [addChild, ['@admin.miembro_proyecto', 'proyecto']]
```

Figura 9: Servicio admin de Proyecto de Extensión

Fuente: Elaboración propia, captura de pantalla de código fuente.

Rol de proyecto

El admin de roles es muy básico pues solo contiene el campo de nombre. Por lo tanto, en cada una de sus acciones se agregó este único campo.

De igual manera que con **Proyecto**, se agregó al admin de miembros como *hijo*. De esta forma se puede listar los miembros que cumplen cada rol en particular. Esto se logra mediante el método **addChild** (figura 9) y creando un menú del mismo tipo que el definido en el admin de proyectos.

2.3.2. ABMs Simples y admins *hijos*

En el caso que una actividad solo fuera completada por una persona, o se tratase de un admin *hijo*, la definición del admin solo contendría los datos de la persona y la actividad. Las ABMs implementadas de esta manera se listan a continuación:

- Asambleísta
- Consejero Superior
- Director de Instituto
- Coordinador de Materia
- Director de Carrera
- Miembro de Proyecto
- Miembro de Comisión de Consejo Superior
- Miembro de Práctica Profesional Supervisada
- Rector
- Miembro de Curso de Extensión
- Miembro de Actividad de Divulgación
- Miembro de Pasantía
- Miembro de Voluntariado
- Responsable de Área
- Miembro de Programa
- Vinculador

- Vice Rector
- Secretario
- Beca Befat
- Movilidad Conurbano Sur
- Publicación
- Movilidad RTF
- Sub-Secretario

Implementación ABM Simple: Publicación

Se implementó la entidad extendiendo de la clase **Actividad** y se definió el admin de manera que:

- La acción de crear una publicación requiere especificar la persona, la fecha de inicio y la fecha de fin de la actividad.
- Las acciones de listar y visualizar el elemento muestran los datos de la persona y la actividad.

Mediante la función *configureFormFields* se establecieron los campos a llenar durante la creación de una Publicación.

```

1 public function configureFormFields(
2     FormMapper $formMapper
3 ): void {
4     $formMapper
5         ->add('persona', ModelListType::class)
6         ->add('inicio', DatePickerType::class)
7         ->add('fin', DatePickerType::class)
8     ;

```

9 }

Ejemplo de código 1: Definición de campos durante la creación de Publicaciones.

Fuente: Elaboración propia

Se utilizó un tipo de formulario de **Sonata** llamado **ModelListType**, que permite seleccionar las personas a través de una lista y se agregó un elemento **Datepicker** para especificar las fechas. Además, se separó la funcionalidad común en un Trait de PHP de manera que si se necesita algo en específico, se puede sobrescribir los métodos en la clase admin base.

2.4. Librerías

Si bien en muchos de los casos las librerías utilizadas en el desarrollo fueron configuradas por el componente **Flex** de **Symfony**, algunos paquetes requirieron de una configuración más compleja y son los aquí listados.

2.4.1. Instalación y Configuración de Sonata-User

Esta librería integra el componente **FOSUser** de **Symfony** con **Sonata Admin** y agrega algunas características adicionales.

Para su instalación es necesario tener FOSUser instalado y configurado además de SonataAdmin y SonataEasyExtends. Para este proceso se siguieron los pasos establecidos en la documentación de **Sonata-User** («Sonata-user Installation», sf)

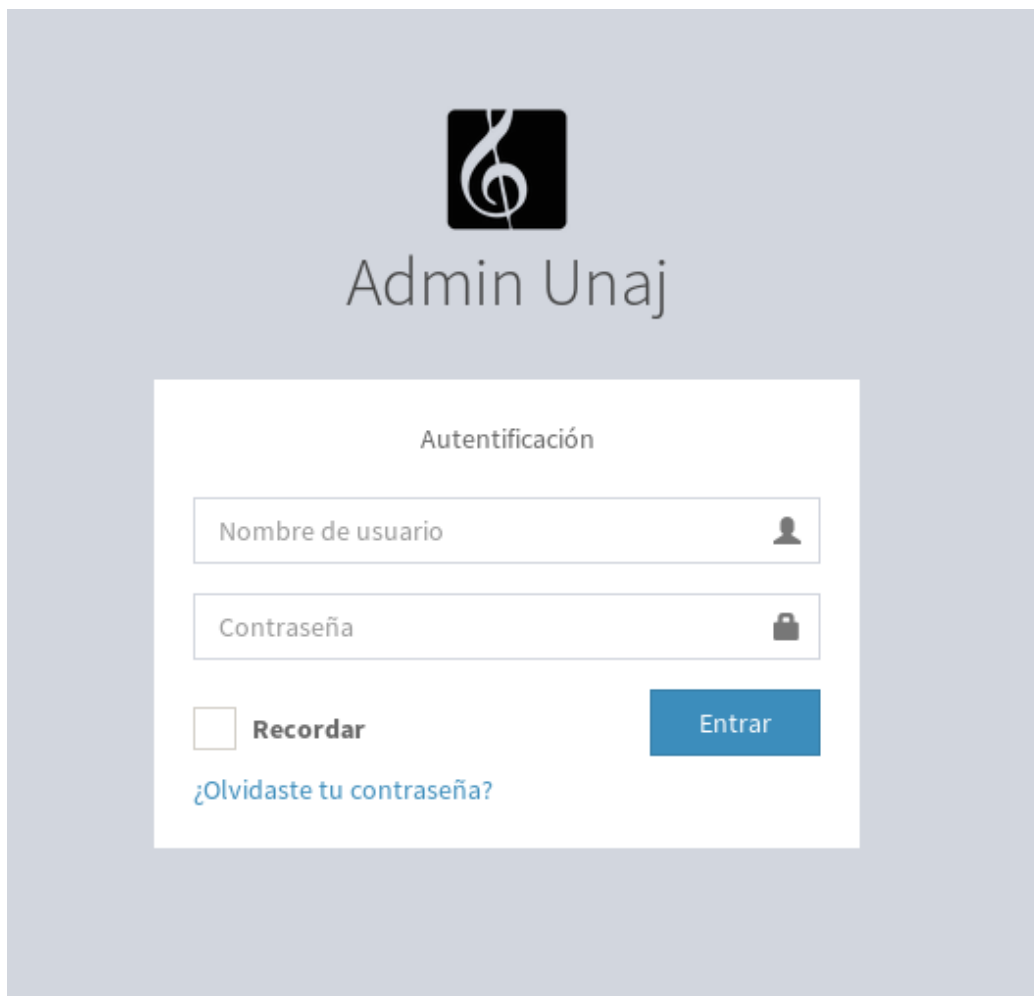


Figura 10: Pantalla de login de Sonata-User

Fuente: Elaboración propia, captura de pantalla de la aplicación web

Configuración básica: Sonata User involucra varias librerías y elementos del sistema que deben ser configurados adecuadamente, estos son:

- **Doctrine:** se debe definir el mapeo de entidades necesarias para el funcionamiento de la librería.
- **FOSUser:** se requiere especificar entidades y servicios para la administra-

ción de usuarios y grupos.

- **Security:** es necesario configurar la autenticación de usuarios y el control de acceso.
- **Routing:** se debe agregar información necesaria para la generación de rutas.
- **Sonata-User:** configuración que define el tipo de datos y las clases que definen la estructura de los datos de usuarios y grupos.

Como primer paso para la instalación y configuración de Sonata-User, se agregó la librería FOSUser y Sonata-User a través del gestor de paquetes Composer, luego se comenzó con la configuración:

Sonata-User

Se especificó a Sonata-User que los datos son administrados por un ORM.

```
1 #src/config/packages/sonata_user.yaml
2
3 sonata_user:
4     manager_type: orm
```

Ejemplo de código 2: archivo de configuración de sonata-user

Fuente:

<https://sonata-project.org/bundles/user/master/doc/reference/installation.html>.

FOSUser

En cuanto a FOSUser se definió el driver de base de datos como ORM y el sistema de autenticación que utilizará. También se especificaron clases base proporcionadas por Sonata-User, que serán utilizadas para la generación de las entidades finales de usuario y grupo. Por último, se definen los servicios que administran estas entidades e información sobre el servicio de correo.

```

1 #src/config/packages/fos_user.yaml
2
3 fos_user:
4     db_driver: orm
5     firewall_name: main
6     user_class: Sonata\UserBundle\Entity\BaseUser
7     group:
8         group_class: Sonata\UserBundle\Entity\BaseGroup
9
10         group_manager: sonata.user.orm.group_manager
11     service:
12         user_manager: sonata.user.orm.user_manager
13
14     from_email:
15         address: "test@domain.com"
16         sender_name: "test@domain.com"

```

Ejemplo de código 3: archivo de configuración de FOSUser

Fuente:

<https://sonata-project.org/bundles/user/master/doc/reference/installation.html>.

Doctrine

La única configuración necesaria para el ORM es la definición explícita de cada una de las entidades mapeadas. Doctrine cuenta con una característica llamada auto mapping, que permite cargar la configuración de entidades almacenadas bajo el directorio Entity/ de cada uno de los bundles. Para la configuración de doctrine, cada entidad utilizada por Sonata-User y FOSUser debe estar definida en su archivo de configuración; pero dado que se tiene habilitado la función de auto mapping y cada entidad está bajo un directorio de nombre “Entity” no es necesario agregar nada a la configuración existente.

Routing

Se agregaron las configuraciones de las rutas necesarias para ambas librerías.

```
1 #src/config/routes/fos_user.yaml
2
3 fos_user:
4     resource: "@FOSUserBundle/Resources/config/routing/all.
        xml "
```

Ejemplo de código 4: archivo de configuración de rutas de FOSUser

Fuente:

<https://sonata-project.org/bundles/user/master/doc/reference/installation.html>.

```
1 #src/config/routes/sonata_user.yaml
2
3 sonata_user_admin_security:
4     resource: '@SonataUserBundle/Resources/config/routing/
        admin_security.xml '
5     prefix: /admin
6
7 sonata_user_admin_resetting:
8     resource: '@SonataUserBundle/Resources/config/routing/
        admin_resetting.xml '
9     prefix: /admin/resetting
```

Ejemplo de código 5: archivo de configuración de rutas de sonata-user

Fuente:

<https://sonata-project.org/bundles/user/master/doc/reference/installation.html>.

Security

En cuanto a la configuración de seguridad, se definieron dos sistemas de autenticación (denominados firewall). Uno se encargará de administrar la seguridad en los usuarios admin y el otro en usuarios básicos.

```
1 #src/config/packages/security.yaml
```

```

2
3 # -> custom firewall for the admin area of the URL
4     admin:
5         pattern:          /admin(.*?)
6         context:          user
7         form_login:
8             provider:      fos_userbundle
9             login_path:    /admin/login
10            use_forward:   false
11            check_path:    /admin/login_check
12            failure_path:  null
13            default_target_path: /admin/dashboard
14        logout:
15            path:          /admin/logout
16            target:        /admin/login
17        anonymous:        true

```

Ejemplo de código 6: Firewall para el área admin del sistema.

Fuente:

<https://sonata-project.org/bundles/user/master/doc/reference/installation.html>.


```

1 #src/config/packages/security.yaml
2     main:
3         pattern:          .*
4         context:          user
5         form_login:
6             provider:      fos_userbundle
7             login_path:    /login
8             use_forward:   false
9             check_path:    /login_check
10            failure_path:   null
11        logout:            true
12        anonymous:         true

```

Ejemplo de código 7: Firewall para el área de registro y login de usuarios básicos.

Fuente:

<https://sonata-project.org/bundles/user/master/doc/reference/installation.html>.

Además se especificó la jerarquía de roles y proveedor de usuarios:

```

1     role_hierarchy:
2         ROLE_ADMIN:       [ROLE_USER, ROLE_SONATA_ADMIN]
3         ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
4     ]
5     SONATA:
6         - ROLE_SONATA_PAGE_ADMIN_PAGE_EDIT
7
8     providers:
9         fos_userbundle:
10             id: fos_user.user_provider.username
11
12     encoders:
13         FOS\UserBundle\Model\UserInterface: bcrypt

```

Ejemplo de código 8: Jerarquía de roles, tipo de encriptación y proveedor de usuarios

Fuente:

<https://sonata-project.org/bundles/user/master/doc/reference/installation.html>

Por último, se define el control de acceso de manera que se pueda ingresar anónimamente a cada página de registro, inicio de sesión, reinicio de contraseña, etc. También se especifica qué roles tienen permitido ingresar a la parte de administración del sistema.

```
access_control:
  # Admin login page needs to be accessed without credential
  - { path: ^/admin/login$, role: IS_AUTHENTICATED_ANONYMOUSLY }
  - { path: ^/admin/logout$, role: IS_AUTHENTICATED_ANONYMOUSLY }
  - { path: ^/admin/login_check$, role: IS_AUTHENTICATED_ANONYMOUSLY }
  - { path: ^/admin/resetting, role: IS_AUTHENTICATED_ANONYMOUSLY }

  # Secured part of the site
  # This config requires being logged for the whole site and having the
  # Change these rules to adapt them to your needs
  - { path: ^/admin/, role: [ROLE_ADMIN, ROLE_SONATA_ADMIN] }
  - { path: ^/.*, role: IS_AUTHENTICATED_ANONYMOUSLY }
```

Figura 11: Control de acceso

Fuente: Recuperado de

<https://sonata-project.org/bundles/user/master/doc/reference/installation.html>

Generación de entidades finales

A partir de esta configuración se generaron las entidades de usuario y grupo mediante el comando:

```
1 bin/console sonata:easy-extends:generate SonataUserBundle --
   dest=src --namespace_prefix=App
```

Esto da como resultado un directorio en **Application\Sonata\UserBundle** el cual contiene las entidades a utilizar por las librerías. Por último, se configuró Sonata-User y FOSUser para que utilicen las nuevas entidades:

```
1 #src/config/packages/sonata_user.yaml
2
3 sonata_user:
4   manager_type: orm
5   class:
```

```
6      user: App\Application\Sonata\UserBundle\Entity\User
7      group: App\Application\Sonata\UserBundle\Entity\Group
```

Ejemplo de código 9: Archivo de configuración de Sonata-User.

Fuente:

<https://sonata-project.org/bundles/user/master/doc/reference/installation.html>

```

1 #src/config/packages/fos_user.yaml
2
3 fos_user:
4     db_driver: orm # other valid values are 'mongodb' and '
    couchdb '
5     firewall_name: main
6     user_class: App\Application\Sonata\UserBundle\Entity\User
7     group:
8         group_class: App\Application\Sonata\UserBundle\Entity
        \Group
9         group_manager: sonata.user.orm.group_manager
10    service:
11        user_manager: sonata.user.orm.user_manager
12
13    from_email:
14        address: "test@domain.com"
15        sender_name: "test@domain.com"

```

Ejemplo de código 10: Archivo de configuración de FOSUser.

Fuente:

<https://sonata-project.org/bundles/user/master/doc/reference/installation.html>

2.4.2. API-Platform

Para la instalación de esta librería sólo es necesario agregarla a través de **Composer** y Flex se hace cargo de la configuración. Luego de realizado esto, se configuró las entidades a exponer, que en este caso serían todas las entidades que representan datos en el sistema.

API-Platform permite definir las entidades a utilizar durante el proceso de serialización mediante anotaciones o archivos de configuración. Por defecto, este componente serializa todos los campos y todas aquellas funciones que retornen algún valor. Probablemente se tengan entidades con funciones o datos que no se quieran exponer a los usuarios. Además, se pueden encontrar referencias circulares entre algunas entidades, que fue lo que sucedió entre miembros y roles de

proyecto (un miembro tiene un rol, y un rol tiene muchos miembros). Por este motivo, se decidió configurar la información que es expuesta a través de estas entidades.

El proceso de serialización

La serialización es el proceso de convertir estructuras de datos u objetos en un formato que puede ser almacenado (por ejemplo, en un archivo o búfer de memoria) o transmitido (por ejemplo, a través una conexión de red) y reconstruido luego (posiblemente en un entorno diferente). («Serialization», 2019)

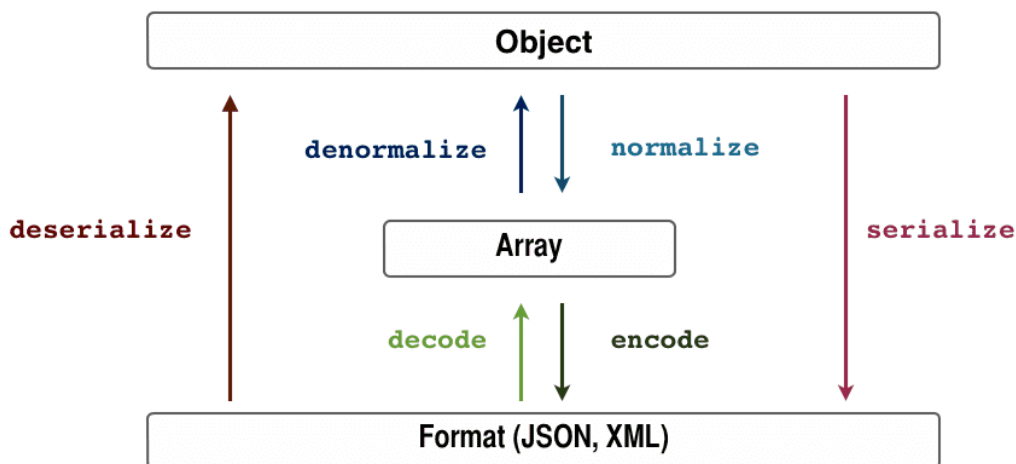


Figura 12: El proceso de serialización

Fuente: Recuperado de <https://api-platform.com/docs/core/serialization/>

En **Symfony** esto es logrado a través del componente **Serializer**. El proceso en sí, sigue el esquema presentado en la figura 12. Para la conversión primero se convierte el objeto en un *array* (proceso denominado normalización) y luego se convierte al formato específico que se necesite (a este proceso se lo llama codificación).

API-Platform permite modificar la información que se expone a los usuarios mediante grupos de *serialización* y *deserialización*. A través del uso de anotacio-

nes en cada propiedad de la entidad se puede definir si agregar la propiedad al proceso de normalización o no dependiendo de la acción que se esté realizando. Si se desean funciones más complejas, es posible definir *normalizadores* y *codificadores* personalizados o decorar *normalizadores* existentes. («The Serialization Process», sf)

Modificación del proceso de serialización

Mediante anotaciones se agregó cada propiedad relevante a cada acción en particular y se omitieron aquellas propiedades o funciones que no necesitan estar presentes o que no se desean exponer a los usuarios.

Al definir un grupo de serialización se logra que, al agregar una propiedad al contexto de normalización, sólo se serialice las propiedades pertenecientes al grupo en cuestión. Esto significa, que si se agrega una relación al contexto de normalización, se serializarán los campos de la entidad relacionada que pertenezcan a dicho grupo.

2.4.3. Guzzle y EightPointsGuzzle

Guzzle es una librería que facilita la interacción con servicios web, al proveer métodos simples para realizar solicitudes **HTTP**. Su funcionamiento consiste en la creación de un objeto llamado **cliente**, el cual provee acceso al servicio web a utilizar.

Eightpoints integra Guzzle con Symfony y permite definir un cliente como servicio de manera que sea posible utilizarlo en cualquier parte de la aplicación.

Se utilizaron estas dos librerías para configurar los servicios web a integrar con RUDA. Su configuración requiere de la ruta base de la API para funcionar y es posible definir diversas opciones como headers y autenticación.

Se definió un cliente Guzzle para el servicio web de mapuche definiendo su ruta base, método de autenticación y headers. Como autenticación se utilizó HTTP digest y se definieron headers que definen el formato a utilizar como **json**.

2.5. Vista de Persona

Se creó una vista mediante la cual se pueda acceder a la información de cada persona y a la lista de actividades en las que se encuentra involucrada. Su implementación se basó en la iteración de la colección de actividades de la persona y la generación de html mediante **Twig**. Además, se agregó estilo mediante **CSS** y una simple animación en **Javascript**.

2.5.1. Introducción Twig

Twig es el motor de *templates* por defecto de **Symfony**, éste permite reutilizar elementos y agregar lógica a la vista en distintas partes de una aplicación web.

Un motor de *templates* posibilita, entre otras cosas, reutilizar la estructura básica de una página y cambiar o sobrescribir aquello que varía.

Twig funciona mediante el uso de **bloques**, entendidos éstos como etiquetas que delimitan secciones del *template*. Estas etiquetas tienen la función de indicarle al motor de *templates* que un *template hijo* puede sobrescribir estas partes del *template*. De esta manera se consigue separar las partes estáticas del sitio, de aquellas que son dinámicas.

2.5.2. Estilo

Para la implementación del estilo de la vista, se utilizó **Sass**, un preprocesador del lenguaje de hojas de estilo CSS. El mismo permite, entre otras cosas, utilizar funciones, variables y herencia de selectores. Se empleó en el desarrollo por el hecho de que al utilizar selectores anidados se simplifica la lectura y comprensión de algunas partes de la implementación. Además, la opción de poder definir variables y funciones facilita mucho el desarrollo.

Se diseñó la distribución de elementos utilizando *grid*, una característica de **CSS** que permite acomodar elementos en una cuadrícula. En un principio, se decidió por asignar los elementos de acuerdo a la figura 14. Esto significa que se

separó el contenido en tres contenedores: el título , la lista de actividades y los datos personales.

Se determinaron las reglas para distribuir los elementos de esta forma mediante CSS Grid:

```
1 display: grid;
2 grid-template-columns: 1fr 1fr;
3 grid-template-rows: 50px auto;
```

Ejemplo de código 11: Definición de filas y columnas de la cuadrícula.

Fuente: Elaboración propia.

De esta manera, se establecieron dos columnas ocupando todo el espacio disponible y dos filas: una de 50px para el título y otra con un alto automático de manera que se ajuste a la cantidad de actividades. Una vez definida la cuadrícula, se estableció el espacio que utilizará cada contenedor.

```
1 .actividades{
2     grid-column-start: 1;
3     grid-column-end: 2;
4     padding: 10px 20px;
5 }
```

Ejemplo de código 12: Orientación del contenedor de actividad en la cuadrícula.

Fuente: Elaboración propia.

```
1 .persona-header{
2     grid-row-start: 1;
3     grid-column-start: 1;
4     grid-column-end: 2;
5     h3{
6         margin-left: 10px;
7     }
8 }
```

Ejemplo de código 13: Orientación del contenedor del título de la acción.

Fuente: Elaboración propia.


```

1 .datos-personales{
2     grid-column-start: 2;
3     grid-row-start: 1;
4     grid-row-end: 3;
5     padding: 20px;
6 }

```

Ejemplo de código 14: Orientación del contenedor de datos personales.

Fuente: Elaboración propia.

2.5.3. Template

En primer lugar, se comenzó heredando de un template general de **Sonata-Admin** denominado **standard_layout** y se sobrescribió la sección o el bloque **sonata_admin_content**, que es la parte en la cual se muestra el contenido de cada acción.

Dado que entre persona y actividad hay una relación de 1-n, cada *objeto Persona* cuenta con una colección de actividades. Por lo tanto, se decidió definir un contenedor para cada una, mediante la utilización de un bucle *for* en **Twig**. Como resultado, se crea un contenedor y se insertan los datos por cada actividad presente en la colección. El resultado de esta operación puede observarse en la figura 13.

Cada ítem de actividad está compuesto por un contenedor que actúa de *header*, y otro que almacena los datos.

Se agregaron links en cada ítem de actividad, uno para visualizar la actividad y otro para modificarla, esto se logró obteniendo el nombre de la ruta desde una función en la entidad y generando la ruta mediante la función *path* que provee **Twig**. Además, se agregó un link que permite, mediante **Javascript**, expandir o colapsar el contenedor de los datos de actividad. Estos links se agregaron en el *header* de cada actividad.

Cada actividad cuenta con un método en su entidad que comprueba si se encuentra activa o no. Este valor es utilizado en el template para dar indicación

visual acerca del estado de la actividad.

2.5.4. Obtención de rutas

En **Symfony**, el nombre de una ruta actúa de identificador de la misma. Si se desea generar una ruta, es necesario proveer como parámetro el nombre.

Sonata define sus rutas de la siguiente forma:

admin_app_{nombre_de_clase}_action

Por ende, se optó por generar el nombre de la ruta a partir del nombre de la clase de la entidad. Esto quiere decir que, si se tiene una nombre de entidad igual a **Secretario**, mediante operaciones de manipulación de *cadenas*, transformarla de **Secretario** a **admin_app_secretario_acción**. Para lograr esto, se creó un método en cada entidad, que permite obtener el nombre de la ruta, a través del nombre de la clase.

```
1 public function getRoute()
2 {
3     $name = get_class($this); // $name=App\Entity\Rector
4     $result = substr($name, 11); // $result=Rector
5     $result = strtolower($result); // $result=rector
6     $result = "admin_app_". $result . "_";
7     // $result=admin_app_rector_
8     return [
9         "id" => $this->getId(),
10        "route" => $result
11    ];
12 }
```

Ejemplo de código 15: Obtención del nombre de una ruta.

Fuente: Elaboración propia.

Para generar una ruta desde un template, es necesario el nombre de la ruta y todos los parámetros que la misma requiera. En el caso del método del ejemplo 15,

se necesita el id como parámetro.

```
1 {{path(routeData.route ~ "edit", {id: routeData.id})}}
```

Ejemplo de código 16: Generación de una ruta desde un template.

Fuente: Elaboración propia.



Figura 13: Listado de actividades.

Fuente: Elaboración propia, captura de pantalla de aplicación web.

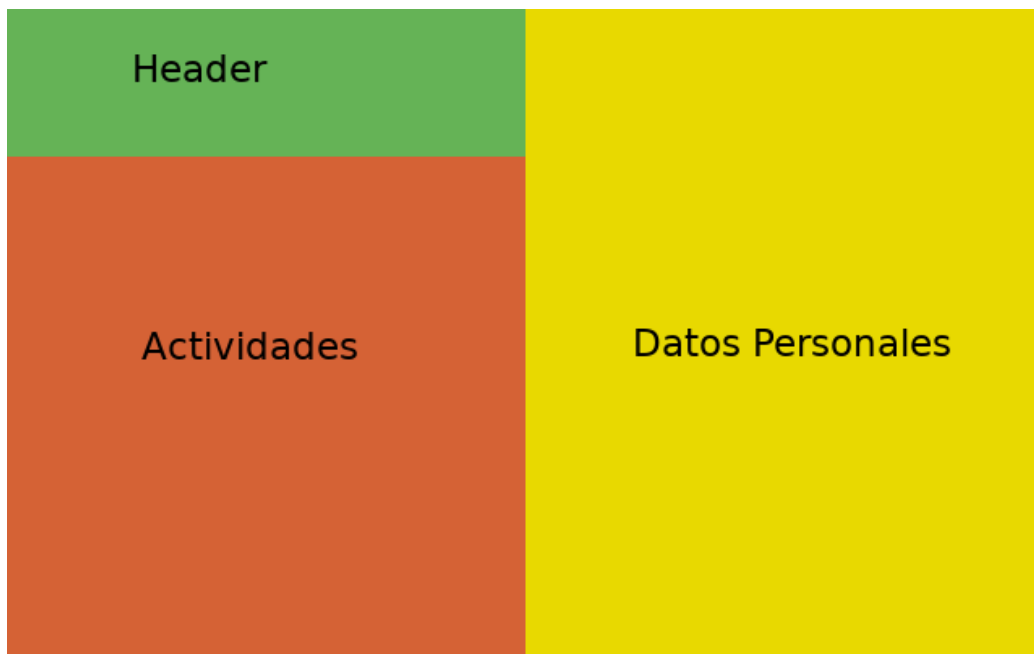


Figura 14: Distribución de columnas y filas.

Fuente: Elaboración propia.

2.6. Integración

2.6.1. Eventos

Durante la ejecución de una aplicación **Symfony**, muchos eventos son disparados. En consecuencia, es posible esperar por la ocurrencia de estos eventos y responder ejecutando una sección de código a través del uso de **Event Listeners** o **Event Subscribers**. El funcionamiento básico de estos componentes es el mismo, sin embargo, difieren en algunos conceptos: un **Event Listener** siempre *escuchará* por el evento establecido, en cambio, un **Event Subscriber** solo *escuchará* por eventos a los que está subscripto. Además, un **Event Subscriber** puede *de-subscribirse* de un evento en cualquier momento.

Doctrine

Doctrine tiene su propio conjunto de eventos a los cuales se puede escuchar durante el ciclo de vida de una entidad:

- `preRemove`: antes de una operación de borrado.
- `postRemove`: después de una operación de borrado.
- `prePersist`: antes de una operación *persist*.
- `postPersist` después de una operación *persist*.
- `preUpdate`: antes de una operación de actualización.
- `postUpdate`: después de una operación de actualización.
- `postLoad`: luego que se carga la entidad.
- `loadClassMetadata`: luego de la carga de la *metadata* de clase.
- `onClassMetadataNotFound`: cuando no se encuentra la correspondiente *metadata* de la clase.
- `preFlush`: antes de una operación *flush*.
- `onFlush`: luego de los cambios en una operación *flush*.
- `postFlush`: después de una operación *flush*.
- `onClear`: luego de una operación *clear*.

Doctrine permite definir distintos tipos de **Event Listeners**. Por un lado, están aquellos que *escuchan* por eventos en todas las entidades y por otro lado, es posible definir *listeners* específicos para cada entidad.

2.6.2. Implementación

Para realizar la integración entre los sistemas se pensó en almacenar en RUDA identificadores reconocibles desde los sistemas externos. De esta manera, cada dato podrá obtener información adicional mediante solicitudes HTTP.

Se definió una entidad para representar las personas y sus propiedades se definieron de la siguiente manera:

- **id:** clave primaria del sistema RUDA.
- **id_mapuche:** referencia a legajo en mapuche.
- **id_guaraní:** referencia a una propiedad identificadora en el sistema guaraní.

Además, se establecieron propiedades no mapeadas por el ORM, de manera de llenarlas con la información obtenida de los sistemas externos.

Cada vez que una entidad de persona es cargada por el ORM será necesario realizar una solicitud HTTP a **Mapuche** para obtener los datos. Por consiguiente, se hizo uso del evento *postLoad* del ciclo de vida de una entidad.

Se implementó un **Entity Listener** para la entidad **Persona**, esto quiere decir que cada vez que se cargue una entidad de este tipo se ejecutará el método definido en el *listener*. El algoritmo básico consiste en realizar una petición al sistema **Mapuche** con su correspondiente identificador, obtener los datos y asignar cada información específica a cada propiedad en la persona.

2.6.3. Rendimiento

Obtener los datos de esta manera genera una carga adicional al tiempo de respuesta de la solicitud general, en especial en el caso de acciones de listado de personas, ya que por cada persona se realizará una solicitud HTTP a **Mapuche**.

Esto se piensa mitigar principalmente mediante el uso de caché.

2.6.4. Inserción de Personas

Una vez realizada la integración entre **RUDA** y **Mapuche** se requirió de una vista que permita agregar personas desde **Mapuche**. Por lo tanto, se procedió a implementar una pantalla de búsqueda por nombre que haga posible agregar personas a través de los resultados. El funcionamiento básico sería el siguiente: se listan los resultados mediante **Twig** y se generan botones para agregar o visualizar (en caso que el resultado ya se encuentre agregado) los datos. El botón para insertar el resultado, funciona mediante *javascript* y *Ajax*: se prepara un objeto *json*, el cual al momento de agregar el dato es enviado a la ruta de inserción de personas de **Api-Platform**. En cuanto al estilo de la vista, se emuló el template de **Sonata** utilizado para listar elementos en una tabla.

3. Conclusión

El hecho de no contar con un sistema que almacene la información de cada actividad extracurricular realizada por las personas de la UNAJ presenta un problema para la institución y sus integrantes. Estos datos no solo son valiosos debido a propósitos administrativos, sino que cada una de estas actividades le otorga un valor agregado a la historia académica del estudiante y deberían ser registradas al igual que sus materias y calificaciones.

En cuanto al desarrollo, se consiguió implementar una interfaz web que ayudará a la administración de estos datos. Será posible solicitar la información de una determinada persona y obtener una lista sus actividades extracurriculares y cargos. Cada actividad contará con una interfaz administrativa, lo que permitirá agregar personas al sistema y asignarle las actividades o cargos que se requieran. Además, se terminó por definir un servicio REST que obtiene parte de su información del sistema **Mapuche** e integra los datos de ambos sistemas.

Con respecto al sistema **Guaraní**, no se alcanzó a implementar la integración por razones de tiempo, ya que se debe ajustar el sistema RUDA para que los dos tipos de datos de personas coexistan entre sí: una persona puede estar traba-

jando en la universidad, es decir, registrada en el sistema **Mapuche** y al mismo tiempo estudiando. Estos dos datos deben vincularse para representar una sola persona y además, indicar estos tipos de situaciones.

Contar con esta información puede dar lugar a análisis de datos, estadística e incluso el desarrollo de otros proyectos. Será posible, por ejemplo, identificar alumnos que sean muy activos en áreas de investigación, lo que podría dar lugar a un diferente tipo de orientación o tutoría para un mejor aprovechamiento de sus habilidades. Además de lo mencionado, quizás en un futuro se pueda listar esta información en un comprobante o anexo al título universitario de manera de resaltar los méritos de la persona.

Mejorar los procesos de administración de la información de la Universidad es muy necesario y útil a futuro, ya que beneficia a todos los que la integran.

4. Reflexión

Aprender un *framework* no resulta ser un proceso trivial, requiere de aprender los distintos componentes que lo conforman y en parte, su funcionamiento interno. Es necesario utilizar y leer código con el que no se está familiarizado y que pasó por muchas revisiones hasta llegar al público.

Trabajar con código de terceros es una buena manera de ejercitar la habilidad de *leer* código. Esta habilidad puede abrir la puerta a contribuciones a proyectos open source o personales.

Durante el transcurso de esta PPS, no sólo aprendí a trabajar con un nuevo *framework*, también me familiaricé con diferentes flujos de trabajo y herramientas. Definitivamente es el proyecto más grande del que formé parte y, por consiguiente, el que más aprendizaje me otorgó.

El proyecto en sí me presentó un desafío por el hecho de tener que sobrellevar el desarrollo del sistema con la escritura y organización del documento. Escribir un documento académico puede volverse una tarea bastante desorganizada (al menos en mi caso) utilizando procesadores de texto comunes, esto es

por que ni bien crece el documento, crece la cantidad de secciones que se deben modificar y mantener. Esto es lo que me llevó a aprender el sistema \LaTeX . El mismo solucionó gran parte de mis problemas de organización, al poder separar el documento en archivos individuales para cada sección. Además, la inserción de gráficos o secciones de código están manejadas internamente y no hay necesidad de preocuparse por los epígrafes, ya que son enumerados automáticamente. Como si esto fuera poco \LaTeX permite definir funciones: esto hace posible programar diferentes tipos de contenido reutilizable en todo el documento.

Por otro lado, este proyecto me llevó a descubrir el área en la cual quiero desenvolverme como futuro Ingeniero, me interesa bastante la optimización y/o automatización de procesos de todo tipo. Quizás también relacionado con la programación o administración de sistemas. Previo a este proyecto no tenía una idea bastante formada respecto a este tema, así que me fue de gran ayuda.

REFLEXIÓN SOBRE LA PRÁCTICA PROFESIONAL SUPERVISADA COMO ESPACIO DE FORMACIÓN:

Aprender un *framework* no resulta ser un proceso trivial, requiere de aprender los distintos componentes que lo conforman y en parte, su funcionamiento interno. Es necesario utilizar y leer código con el que no se está familiarizado y que pasó por muchas revisiones hasta llegar al público.

Trabajar con código de terceros es una buena manera de ejercitar la habilidad de *leer* código. Esta habilidad puede abrir la puerta a contribuciones a proyectos open source o personales.

Durante el transcurso de esta PPS, no sólo aprendí a trabajar con un nuevo *framework*, también me familiaricé con diferentes flujos de trabajo y herramientas. Definitivamente es el proyecto más grande del que formé parte y, por consiguiente, el que más aprendizaje me otorgó.

El proyecto en sí me presentó un desafío por el hecho de tener que sobrellevar el desarrollo del sistema con la escritura y organización del documento. Escribir un documento académico puede volverse una tarea bastante desorganizada (al menos en mi caso) utilizando procesadores de texto comunes, esto es por que ni bien crece el documento, crece la cantidad de secciones que se deben modificar y mantener. Esto es lo que me llevó a aprender el sistema $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$. El mismo solucionó gran parte de mis problemas de organización, al poder separar el documento en archivos individuales para cada sección. Además, la inserción de gráficos o secciones de código están manejadas internamente y no hay necesidad de preocuparse por los epígrafes, ya que son enumerados automáticamente. Como si esto fuera poco $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ permite definir funciones: esto hace posible programar diferentes tipos de contenido reutilizable en todo el documento.

Por otro lado, este proyecto me llevó a descubrir el área en la cual quiero desenvolverme como futuro Ingeniero, me interesa bastante la optimización y/o automatización de procesos de todo tipo. Quizás también relacionado con la programación o administración de sistemas. Previo a este proyecto no tenía una idea bastante formada respecto a este tema, así que me fue de gran ayuda.

REFLEXIÓN SOBRE LA PRÁCTICA PROFESIONAL SUPERVISADA COMO ESPACIO DE FORMACIÓN:

Aprender un *framework* no resulta ser un proceso trivial, requiere de aprender los distintos componentes que lo conforman y en parte, su funcionamiento interno. Es necesario utilizar y leer código con el que no se está familiarizado y que pasó por muchas revisiones hasta llegar al público.

Trabajar con código de terceros es una buena manera de ejercitar la habilidad de *leer* código. Esta habilidad puede abrir la puerta a contribuciones a proyectos open source o personales.

Durante el transcurso de esta PPS, no sólo aprendí a trabajar con un nuevo *framework*, también me familiaricé con diferentes flujos de trabajo y herramientas. Definitivamente es el proyecto más grande del que formé parte y, por consiguiente, el que más aprendizaje me otorgó.

El proyecto en sí me presentó un desafío por el hecho de tener que sobrelevar el desarrollo del sistema con la escritura y organización del documento. Escribir un documento académico puede volverse una tarea bastante desorganizada (al menos en mi caso) utilizando procesadores de texto comunes, esto es por que ni bien crece el documento, crece la cantidad de secciones que se deben modificar y mantener. Esto es lo que me llevó a aprender el sistema \LaTeX . El mismo solucionó gran parte de mis problemas de organización, al poder separar el documento en archivos individuales para cada sección. Además, la inserción de gráficos o secciones de código están manejadas internamente y no hay necesidad de preocuparse por los epígrafes, ya que son enumerados automáticamente. Como si esto fuera poco \LaTeX permite definir funciones: esto hace posible programar diferentes tipos de contenido reutilizable en todo el documento.

Por otro lado, este proyecto me llevó a descubrir el área en la cual quiero

desenvolverme como futuro Ingeniero, me interesa bastante la optimización y/o automatización de procesos de todo tipo. Quizás también relacionado con la programación o administración de sistemas. Previo a este proyecto no tenía una idea bastante formada respecto a este tema, así que me fue de gran ayuda.

Referencias

- About Symfony Project. (sf). Recuperado desde <https://symfony.com/about>
- Create child admins. (sf). Recuperado desde https://symfony.com/doc/master/bundles/SonataAdminBundle/reference/child_admin.html
- Inheritance Mapping. (sf). Recuperado desde <https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/inheritance-mapping.html>
- Serialization. (2019). Recuperado desde <https://en.wikipedia.org/wiki/Serialization>
- Sonata-user Installation. (sf). Recuperado desde <https://sonata-project.org/bundles/user/master/doc/reference/installation.html>
- The Serialization Process. (sf). Recuperado desde <https://api-platform.com/docs/core/serialization/>