GRADO EN INGENIERÍA INFORMÁTICA

TRABAJO DE FINAL DE GRADO

Kalenda: Gestión de recursos humanos en la nube

Autor:
Lucas NIÑO RUIZ

Supervisor:
Alejandro SÁNCHEZ-CAMACHO LÓPEZ
Tutor académico:
Jorge BADENAS CARPIO

Fecha de lectura: 16 de septiembre de 2021
Curso académico 2020/2021

Resumen

A lo largo de este documento se detallan el análisis, planificación, diseño e implementación del proyecto de creación de una aplicación web dirigida a la gestión de los recursos humanos (RRHH) de una empresa. Esta memoria está dentro del marco de desarrollo del Trabajo de Fin de Grado del Itinerario de Sistemas de Información, perteneciente al Grado en Ingeniería Informática de la Universitat Jaume I.

El proyecto engloba la renovación de la aplicación Kalenda, aplicación en la nube para la gestión de RRHH, siendo el alumno en prácticas el encargado de desarrollo del *front-end*. La aplicación, ya existente, fue desarrollada en 2016 con las tecnologías disponibles entonces, razón por la cual había quedado obsoleta, tanto en arquitectura y eficiencia como en funcionalidad y aspecto visual. De esta forma, se han usado las versiones actuales del framework Angular y la biblioteca NG-ZORRO para el *front-end* o SpringBoot como API REST para el *back-end* con el fin actualizar la aplicación acorde a la época actual, revalorizarla y solventar carencias.

La aplicación presenta utilidades para cualquier trabajador de cualquier empresa, si bien es cierto que está más enfocada al usuario cuyo perfil corresponde al responsable de RRHH, permitiendo la creación y gestión de empleados, jornadas laborales y contratos, entre otros.

Palabras clave

Front-end, aplicación web, Angular, TypeScript, componente, evento, módulo, Recursos Humanos (RRHH).

Keywords

Front-end, web application, Angular, TypeScript, component, event, module, Human Resources (HHRR).

Índice general

Índice de figuras

Índice de cuadros

Listings

Capítulo 1

Introducción

1.1. Contexto y motivación del proyecto

La compañía INTEGRA Consultores S.L. está especializada en consultoría y servicios de Tecnologías de la Información y Comunicaciones, ayudando a mejorar la productividad de sus clientes mediante los servicios que ofrece [27]. Entre estos se incluyen: auditoría tecnológica, *outsourcing*, análisis funcional, diseño tecnológico, desarrollo a medida, Business Intelligence, asistencia técnica, soporte técnico *on-site*, migración de infraestructuras y asesoramiento tecnológico. Además, ofrecen una serie de productos como solución a problemas concretos.

El proyecto del que trata este documento se desarrolla sobre Kalenda, una aplicación para la gestión eficiente y unificada de recursos humanos (RRHH) a través de la tramitación de solicitudes, gestión de empleados, confección del calendario laboral, control de horarios, convocatoria de acciones formativas y tablón de anuncios virtual.

La principal motivación de la aplicación viene dada por la dificultad que supone la gestión manual y local de los RRHH, así como por los gastos destinados a este fin [51]. Los beneficios que supone la utilización de esta herramienta incluyen la descarga de trabajo a los equipos de RRHH, el mantenimiento de todos los datos centralizados y el aumento de visibilidad de la información a los empleados (pueden pedir directamente las vacaciones, ver cuántas les quedan, descargar certificados o nóminas...), entre otros.

Figura 1.1: Logos de Integra Consultores S.L. y Kalenda [27].

El problema que presenta la versión de Kalenda desarrollada en 2016 consistía en las tecnologías utilizadas, pues han quedado obsoletas, así como en el alcance funcional, que incluye demasiados aspectos, distorsionando el objetivo final de la aplicación. Es por estas razones que el proyecto consistirá en renovar el aspecto visual de la aplicación así como en actualizar su infraestructura y las tecnologías en las que se basa. También se revisará el alcance funcional, con el objetivo de simplificar la aplicación y mejorar las tareas de automatización.

1.2. Objetivos del proyecto

El objetivo principal de este proyecto es renovar la aplicación existente Kalenda para incrementar la usabilidad y eficiencia de la misma, permitiendo a las organizaciones destinar menos capital a tareas de RRHH.

Esto se puede desglosar en los siguientes subobjetivos:

- Actualizar las versiones de las tecnologías utilizadas en la implementación de la aplicación para sacar mayor partido a las nuevas funcionalidades que estas ofrecen.
- Recodificar y usar patrones de diseño que mejoren el rendimiento de la aplicación.
- Mejorar la comunicación entre el back-end y el front-end.
- Automatizar acciones que actualmente debe hacer el usuario manualmente.
- Desarrollar interfaces de usuario coherentes en toda la aplicación y con un diseño acorde a los estándares actuales.
- Llevar a cabo recortes en el alcance funcional, eliminando funcionalidades redundantes o innecesarias.

1.3. Alcance

Podemos dividir el alcance en funcional, organizativo e informático.

El alcance funcional incluye los siguientes apartados:

1. Publicación de anuncios: se debe poder registrar anuncios cuyos contenidos aparezcan en la página principal de todos los empleados de la empresa.
2. Buzón del empleado: se deben poder revisar las reclamaciones o sugerencias de los empleados y gestionarlas (desestimarlas, desplazarlas a cargos superiores...).
3. Gestión de informes: se deben poder generar y consultar informes referentes a calendarios, vacaciones o fichajes, que recojan y resuman la información pertinente de la organización.

4. Gestión de empleados: se debe poder gestionar los datos de la plantilla, así como los distintos tipos de registros (incidencias, vacaciones, horas extra, excepciones de calendario).

5. Gestión del calendario: se debe poder consultar la información relativa a los calendarios, como son las vacaciones, reuniones, fechas de entrega, etc. Los usuarios de RRHH podrán consultar la información de todos los usuarios.

6. Control de presencia: se debe implementar un sistema de fichajes mediante el cual el usuario de recursos humanos pueda consultar la información.

7. Gestión de alertas: se deben poder registrar alertas de distinto tipo, que se muestren en la pantalla de inicio en las fechas y horas indicadas.

8. Mensajes y correos propios: se mostrará la bandeja de entrada del correo electrónico de la empresa, y se podrán clasificar y redactar correos.

9. Solicitudes: se deben listar y poder ver en detalle las solicitudes del empleado, tanto las que están en curso como las finalizadas.

10. Datos de contacto: se deben mostrar los datos de contacto, así como poder modificarlos.

11. Administración de usuarios: se deben poder crear y gestionar las cuentas de usuarios de una empresa.

12. Administración de roles: se debe poder asignar roles (que den unos permisos específicos) a los usuarios.

El alumno en prácticas ha sido el encargado de llevar a cabo los puntos 4, 5, 6, 7, 10, 11 y 12.

El alcance organizativo de la aplicación se puede dividir en los diferentes perfiles de usuario de la plataforma:

- Recursos Humanos: es el usuario cuyas funcionalidades cobran más peso, debido a que además de poder manejar su propia información, tiene acceso y permisos de edición sobre la información del resto de empleados de la empresa. Engloba los aspectos del 1 al 7 enumerados en el alcance funcional.

- Empleado: representa cualquier tipo de empleado de la organización, exceptuando a los de recursos humanos, con los que comparten varias funcionalidades. Engloban los aspectos del 5 al 10 enumerados en el alcance funcional.

- Administrador: los usuarios (normalmente uno por organización) de este tipo tienen permisos para ejecutar tareas de administración de usuarios y roles, así como de configuración de la aplicación.

En cuanto al alcance informático, la aplicación se conecta a una base de datos donde se almacenan todos los usuarios según sus roles, así como el resto de datos. Para ello, es necesario un servidor en el que desplegar la aplicación del servicio web. En lo referente al usuario, bastará con un navegador web vía ordenador portátil o de sobremesa.

También se debe revisar la lógica de negocio que permite llevar a cabo las acciones descritas en el alcance funcional. Esto incluye la posible reestructuración de la base de datos y los procesos *back-end*, así como el rediseño de las interfaces con las que interactúan los usuarios.

1.4. Descripción del proyecto

El proyecto objeto de esta memoria comprende la creación de una aplicación web para gestión de RRHH en la nube. Lo que se pretende es el rediseño de la aplicación para actualizarla a la época actual, desarrollándola desde cero. Para ello, se quiere incidir en los siguientes aspectos:

- Interfaces: se busca crear una interfaz más responsiva, sobria y elegante, con colores más oscuros y usando mejor el blanco como elemento separador. También guardar la simetría y coherencia a lo largo de todas las interfaces (listados, formularios, diálogos...), cosa que no ocurría en la aplicación ya existente.
- Navegación: el planteamiento ha sido el de una barra de navegación lateral que contiene todos los menús de la aplicación y una cabecera con opciones extra (idioma, perfil, notificaciones). Estos menús llevan a listados, vistas detalle o informes. Si una vista detalle es muy compleja o tiene distintos subapartados, se divide en pestañas, de lo contrario, se usa siempre la misma vista de formulario/detalle. Estos formularios también se usarán cuando se quiera crear un nuevo objeto, salvo cuando este proceso englobe varios pasos, para lo que se utilizará un diálogo.
- Componentes: los componentes desarrollados en este proyecto se han creado para ser reutilizables en otros proyectos si la empresa lo considera necesario, ahorrando carga de trabajo a futuro. Es por esto que muchos componentes propios cuentan con distintas opciones activables según la situación.
- Comunicación con el back-end: durante el desarrollo del *front-end*, la parte del *back-end* no estaba desarrollada al completo, lo que implica que algunas llamadas no funcionasen correctamente. Por tanto, ha sido parte del alumno en prácticas indicar cuáles son las llamadas al servidor necesarias, que serán desarrolladas a lo largo del verano. Asimismo, se ha informado de opciones para mejorar la comunicación, por ejemplo, solicitando la elaboración de un método *back-end* que contenga la lógica de varios métodos distintos, requiriendo de esta forma hacer una única llamada al servidor.
- Biblioteca: para este proyecto se han utilizado componentes de la biblioteca NG-ZORRO, que por norma general están muy pulidos y presentan muchas opciones de base, aunque en alguna ocasión se ha tenido que implementar un *wrapper* o modificar ligeramente su aspecto visual en el fichero *.css* de la aplicación.
- Alcance: anteriormente la aplicación tenía funcionalidades de más, que no estaban planteadas correctamente (había varias rutas para conseguir un mismo fin) y acababan no usándose. Durante el proyecto se ha tenido en cuenta este aspecto, acotando lo necesario.

El alumno recibió una versión de desarrollo con algunos componentes ya desarrollados por la empresa para otros proyectos. Así pues, es tarea del alumno identificar la utilidad de los mismos en

la nueva aplicación para su posible utilización y, en caso de ser necesario, modificarlos para que se adapten a las necesidades de la aplicación a desarrollar.

1.4.1. Tecnologías

Podemos dividir las tecnologías según para qué las utilicemos. Por una parte, tenemos las herramientas de gestión de proyectos software, y por otra las herramientas de desarrollo.

Para la gestión del proyecto se ha usado la suite Atlassian [12], específicamente Jira [30] para el seguimiento de errores y la gestión operativa, Confluence [20] para tareas de documentación y BitBucket [14] como servicio de alojamiento para control de versiones (haciendo uso de Git).

En cuanto a las herramientas de desarrollo, omitiré las usadas únicamente para la configuración del entorno, centrándome en las usadas a diario por el alumno. Como editor de código se ha usado Visual Studio Code [54], que cuenta con una gran cantidad de *plug-ins* útiles para el desarrollo de la aplicación. El entorno de ejecución escogido es Node.js [42], específicamente usando Angular [3] en su versión 11.2.4, que es un framework basado en componentes (para el que se deben usar Typescript [52], HTML y CSS), así como la biblioteca NG-ZORRO [40]. Para visualizar la documentación de la API se usó Swagger UI [53], que facilita el desarrollo de modelos y servicios desde el *front-end*. Al principio del proyecto se usó Draw.io para el diseño de *mock-ups* (bocetos). También se ha empleado Sourcetree [49] conectado a BitBucket (mencionado anteriormente) para el control de versiones ya que lo agiliza de forma sustancial. Por último, se ha utilizado el navegador Google Chrome junto a sus herramientas para desarrolladores.

Figura 1.2: Tecnologías y herramientas de desarrollo.

Estas son las tecnologías que usa la empresa en proyectos similares al que nos ocupa, razón por la cual han sido las escogidas.

1.5. Estructura de la memoria

La memoria se compone de cinco capítulos.

El primer y presente capítulo es introductorio y en él se presentan el contexto y motivación del proyecto, los objetivos, el alcance, la descripción y las tecnologías utilizadas para su desarrollo.

En el segundo capítulo se define la planificación del proyecto, desglosada en metodología, planificación, estimación de recursos y seguimiento del proyecto.

El tercer capítulo trata el análisis del sistema, el diseño de su arquitectura y el aspecto visual con el que interactuará el usuario final.

A lo largo del cuarto capítulo se detalla a fondo el progreso de la implementación del sistema, incluyendo los inconvenientes encontrados y solucionados y las tareas de validación y verificación.

Por último, en el quinto capítulo se exponen las conclusiones a título personal del alumno, diferenciando entre distintos ámbitos.

Capítulo 2

Planificación del proyecto

2.1. Metodología

Para llevar a cabo el proyecto se ha seguido una metodología predictiva basada en PMBOK [26]. Este tipo de metodología en cascada se caracteriza por definir las fases al comienzo del proyecto y seguir los pasos de forma secuencial. Es la elección de la empresa en la mayoría de ocasiones, debido a su sencillez y eficacia, especialmente en proyectos pequeños y con un número de integrantes reducido, razón por la que se consideró que era la adecuada para este caso.

Las cinco fases principales que se van a diferenciar en este proyecto son el análisis, diseño, implementación, pruebas y mantenimiento. En algunos casos puede ser necesario añadir más o dividir alguna de ellas, aunque estas son las más habituales. Al tratarse de un proceso iterativo, se puede volver a una fase anterior dada por finalizada en cualquier punto del desarrollo en caso de que se requiera realizar cambios, como se muestra en la Figura 2.1. En cuanto a la planificación de este proyecto, profundizaremos en la Sección 2.2.

Aunque se realizó una distribución temporal inicial para las tareas, el proyecto sufrió varias modificaciones a lo largo del desarrollo. Esto se debe principalmente a la aparición de aspectos que no se habían tenido en cuenta, cuestión que se detalla en la Sección 2.5. No obstante, todo lo recogido en esta memoria engloba el trabajo realizado durante las 300 horas de prácticas.

2.2. Planificación

Como ya se ha mencionado, en este proyecto se ha seguido una metodología en cascada con unas fases definidas al inicio del mismo. A continuación se explica en qué consistió cada una de estas fases:

1. Inicio: entender y asimilar para qué va a servir el proyecto y el propósito de la aplicación, así como la dinámica de trabajo en la empresa. También se deben definir un alcance y unos requi-

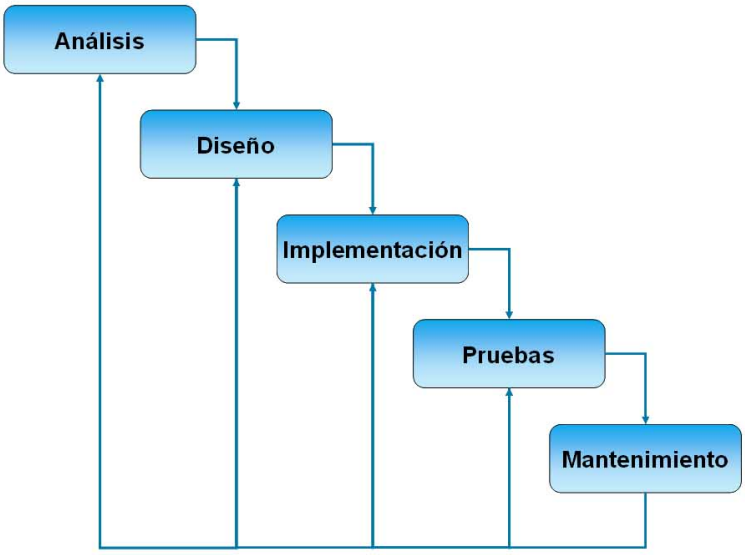


Figura 2.1: Metodología en cascada tradicional [50].

sitos iniciales, sujetos a modificaciones a lo largo del proyecto. Con esta información, desarrollar la Propuesta Técnica.

2. Planificación: esta fase engloba todos los aspectos dedicados a la gestión de recursos (temporales, humanos y materiales), así como los costes que estos suponen. También incluye la elaboración de medidas de prevención y protección ante posibles riesgos, entre los que se encuentran los más comunes en proyectos software.

3. Análisis: en esta etapa se definen los casos de uso y se lleva a cabo la especificación de requisitos. Esto conllevó el estudio de la aplicación desactualizada para identificar los principales aspectos a mejorar, tanto en materia de funcionalidad como de interfaces.

4. Diseño: esta fase sirve para definir aspectos técnicos, entre los que se encuentran el desarrollo del diagrama de clases UML, el diseño de interfaces de usuario y la base de datos y la definición del formato. Gracias a esta fase y al análisis se tiene una guía específica de lo que habrá que implementar.

5. Implementación: es la fase a la que se dedicó más tiempo, pues en ella hay que codificar todo lo definido en los pasos anteriores. Al no haber tenido experiencia previa con las tecnologías de desarrollo, se proporcionó un período de tiempo para la formación en las mismas. En este caso, al recaer todo el desarrollo *front-end* en el alumno, se le proporcionó un esqueleto de la aplicación, que es el que suelen utilizar en la empresa. También existía una versión básica del *back-end* operativa, que posteriormente tendría que actualizarse.

6. Pruebas: engloba las actividades destinadas a la realización de pruebas sobre los resultados y la posterior valoración en base a dichas pruebas. El supervisor hace una revisión para comprobar que se cumple lo establecido al principio del proyecto y tanto él como el alumno en prácticas mantienen una reunión en la que listar posibles mejoras o expansiones de la aplicación.

Todas estas fases se dividieron en distintas tareas más específicas, que permiten una mejor comprensión de todo lo que abarca el proyecto, además de proporcionar una ruta de trabajo más clara. Estas tareas se han plasmado en un WBS (*Work Breakdown Structure*, Esquema de Desglose de Trabajo) [19], elaborado con la herramienta de diagramas Lucidchart [33] que se puede consultar en la Figura 2.2.

El proyecto (la parte del alumno) debía completarse con un total de 300 horas, que con un horario de 5 horas diarias equivalía a 12 semanas laborales. Como se verá en la Sección 2.5, esto acabó alargándose hasta los meses de agosto y septiembre, contenido que no se incluye en esta memoria. Teniendo esto en cuenta, se asignó una cantidad de días a cada una de las tareas listadas en el WBS, que se puede ver en la Figura 2.3, realizada con el programa Microsoft Project [45]. La semana de Pascua no computa como horas trabajadas, ya que el alumno la aprovechó para estudiar exámenes.

Con el propósito de una mejor visualización de la planificación y el reparto de horas, se ha creado un diagrama de Gantt [25], que corresponde con la Figura 2.4. Es evidente que el grueso del proyecto supone la codificación del mismo.

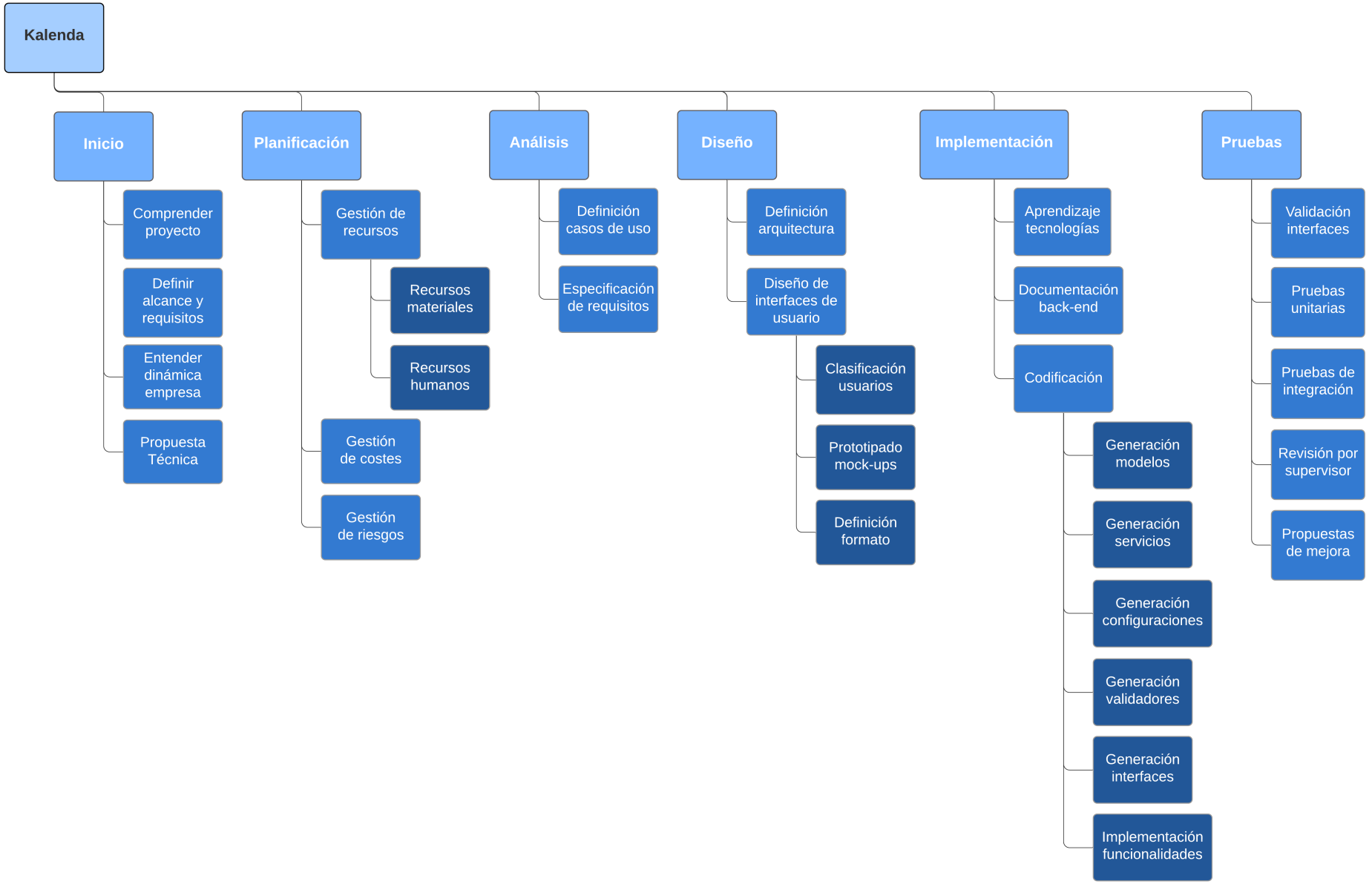


Figura 2.2: Esquema de Desglose de Trabajo.

	Modo de tarea	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		**Kalenda**	**63 días**	**lun 01/03/21**	**mié 02/06/21**	
2		**Reunión de seguimiento**	**62 días**	**lun 01/03/21**	**mar 01/06/21**	
16		**Inicio**	**2 días**	**lun 01/03/21**	**mar 02/03/21**	
17		Comprensión del proyecto	0 días	lun 01/03/21	lun 01/03/21	
18		Definición de alcance y requisitos	1 día	lun 01/03/21	lun 01/03/21	17
19		Entender dinámica empresa	0 días	lun 01/03/21	lun 01/03/21	18
20		Propuesta técnica	1 día	mar 02/03/21	mar 02/03/21	18
21		**Planificación**	**2 días**	**mar 02/03/21**	**jue 04/03/21**	
22		**Gestión de recursos**	**1 día**	**mar 02/03/21**	**mié 03/03/21**	
23		Gestión de recursos materiales	1 día	mié 03/03/21	mié 03/03/21	20
24		Gestión de recursos humanos	0 días	mar 02/03/21	mar 02/03/21	20
25		Gestión de costes	1 día	jue 04/03/21	jue 04/03/21	22
26		Gestión de riesgos	1 día	mié 03/03/21	mié 03/03/21	20
27		**Análisis**	**2 días**	**vie 05/03/21**	**lun 08/03/21**	
28		Definición casos de uso	1 día	vie 05/03/21	vie 05/03/21	21
29		Especificación de requisitos	1 día	lun 08/03/21	lun 08/03/21	28
30		**Diseño**	**6 días**	**mar 09/03/21**	**mar 16/03/21**	
31		Definición de arquitectura	1 día	mar 09/03/21	mar 09/03/21	29
32		**Diseño de interfaces de usuario**	**5 días**	**mar 09/03/21**	**mar 16/03/21**	
33		Clasificación de usuarios	0 días	mar 09/03/21	mar 09/03/21	31
34		Prototipado mock-ups	4 días	mié 10/03/21	lun 15/03/21	33
35		Definición de formato	1 día	mar 16/03/21	mar 16/03/21	34
36		**Implementación**	**57 días**	**mié 03/03/21**	**jue 27/05/21**	
37		Aprendizaje tecnologías	5 días	mié 03/03/21	mar 09/03/21	20
38		Documentación back-end	1 día	mié 17/03/21	mié 17/03/21	37;35
39		**Codificación**	**46 días**	**jue 18/03/21**	**jue 27/05/21**	
40		Generación de modelos	20 días	jue 18/03/21	mié 21/04/21	38
41		Generación de servicios	20 días	jue 18/03/21	mié 21/04/21	38
42		Generación de configuraciones	20 días	jue 18/03/21	mié 21/04/21	38
43		Generación de validadores	20 días	jue 18/03/21	mié 21/04/21	38
44		Generación de interfaces	14 días	jue 22/04/21	mar 11/05/21	41;43;40;42;35
45		Implementación de funcionalidades	12 días	mié 12/05/21	jue 27/05/21	44
46		**Pruebas**	**16 días**	**mié 12/05/21**	**mié 02/06/21**	
47		Validación de las interfaces	2 días	mié 12/05/21	jue 13/05/21	44
48		Pruebas unitarias	2 días	vie 28/05/21	lun 31/05/21	45
49		Pruebas de integración	1 día	mar 01/06/21	mar 01/06/21	48
50		Revisión por supervisor	1 día	mié 02/06/21	mié 02/06/21	49
51		Propuestas de mejora	0 días	mié 02/06/21	mié 02/06/21	50

Figura 2.3: Asignación temporal a tareas.

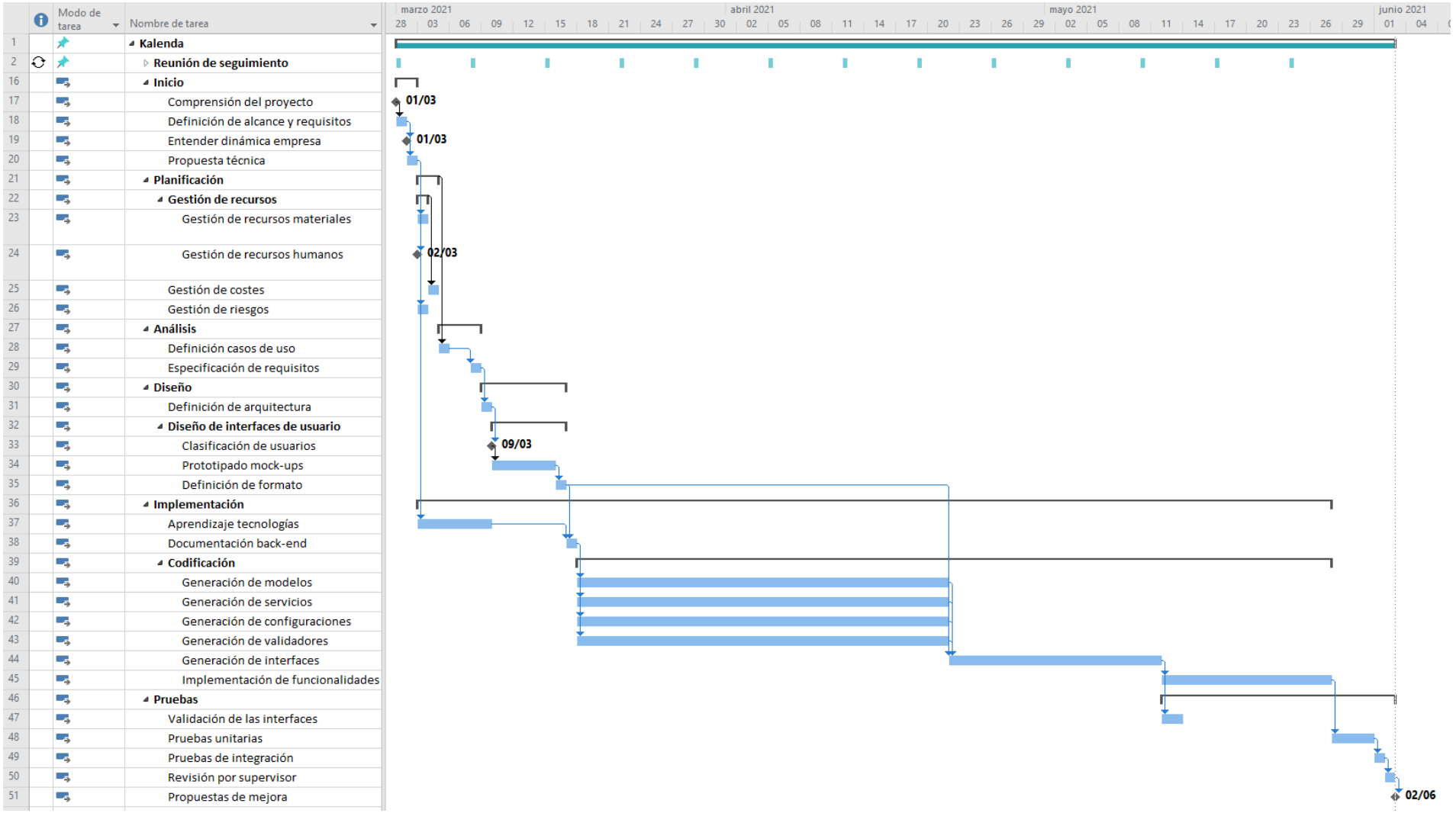


Figura 2.4: Diagrama de Gantt.

2.3. Estimación de recursos y costes del proyecto

Para conseguir el cómputo total del coste del desarrollo de Kalenda, debemos distinguir entre tres tipos de recursos: software, hardware y humanos (donde incluimos los recursos indirectos). Una vez conocemos todos los recursos que han sido necesarios, elaboramos una serie de tablas que muestran los costes aproximados de cada uno de ellos relativos al uso individual del autor de la memoria. Los recursos se desglosan de la siguiente forma:

- **Recursos software:** el editor de código utilizado para implementar la parte del *front-end* fue Visual Studio Code. El framework utilizado ha sido Angular en su versión 11.2.4 junto a la biblioteca NG-ZORRO. Para la visualización de la documentación de los métodos disponibles desde el *back-end* se ha utilizado Swagger UI, que permite un desarrollo más cómodo de los servicios necesarios en el *front-end*. Como herramienta de control de versiones se ha utilizado la aplicación perteneciente a la suite Atlassian, BitBucket, que es un servicio de alojamiento basado en web para proyectos que usan Mercurial o Git (nuestro caso). También se ha usado Sourcetree, programa que ha facilitado la gestión de subidas y descargas de código, ya que simplifica la interacción con los repositorios mediante el uso de interfaces gráficas claras. También hay que tener en cuenta las herramientas de gestión de proyectos, en este caso Jira y Confluence, también pertenecientes a la suite Atlassian. Por último, para la elaboración de *mock-ups* se ha usado el software Draw.io, compatible con Confluence y Jira. También incluimos en este apartado el dominio destinado a la aplicación. Cabe destacar que este dominio no se ha utilizado durante el desarrollo de la aplicación, pero recordemos que la aplicación ya fue desarrollada en el año 2016, por lo que es este el dominio que se utilizará una vez la renovación de la aplicación sea finalizada (se espera que esto ocurra a principios de octubre). Es por esto que también lo hemos contabilizado en el cómputo del coste total del proyecto.

  Los costes de todos estos recursos pueden consultarse en el Cuadro 2.1 [13].

Recurso	Coste	Meses	Coste total
Visual Studio Code	Licencia gratuita	-	0 €
Angular	Licencia gratuita	-	0 €
Swagger UI	Licencia gratuita	-	0 €
Draw.io	Licencia gratuita	-	0 €
Sourcetree	Licencia gratuita	-	0 €
Dominio	12,00 €/mes	3	36,00 €
Atlassian BitBucket	2,50 €/mes	3	7,50 €
Atlassian Jira	5,00 €/mes	3	15,00 €
Atlassian Confluence	3,50 €/mes	3	10,50 €
Total			69,00 €

Cuadro 2.1: Costes software.

- **Recursos hardware:** los recursos hardware utilizados por el alumno en prácticas incluyen un ordenador de sobremesa y dos monitores de la marca Lenovo, así como un teclado y ratón de la marca Logitech, proporcionados por la empresa Integra Consultores S.L. Por último, debemos tener en cuenta el servidor que la aplicación necesita para funcionar. Normalmente se suele practicar *outsourcing* en cuanto a la contratación de servidores, pero en el caso de esta empresa disponen de una serie de aplicaciones propias, por lo que sale más rentable la adquisición

de un servidor propio que dé soporte a todas ellas. El coste del servidor reflejará un porcentaje teniendo en cuenta el número de aplicaciones que ofrece la empresa.

Toda la información referente a los recursos hardware se encuentra en el Cuadro 2.2, en el que se muestran los costes amortizados, consultados con diferentes miembros de la empresas.

Recurso	Precio	Coste amortizado
Ordenador sobremesa	500,00 €	15 €
Monitor 1	120,00 €	3,60 €
Monitor 2	90,00 €	2,70 €
Teclado	20,00 €	0,50 €
Ratón	10,00 €	0,25 €
Servidor	1000 €	20,00 €
Total	740,00 €	41,25 €

Cuadro 2.2: Costes hardware.

- **Recursos humanos:** esta parte del proyecto, que se centra en el desarrollo del *front-end*, ha sido realizada por dos integrantes: el estudiante en prácticas y su supervisor. Además de aconsejar y estar disponible para las dudas del alumno, el supervisor mantuvo una reunión de carácter semanal para comprobar cómo iba avanzando el proyecto, indicar aspectos a mejorar y proponer nuevas tareas. En el Cuadro 2.3 únicamente se tienen en cuenta estas horas como remuneradas para el supervisor. También cabe recalcar que, a pesar de que el alumno no percibiera salario, se ha utilizado el precio por hora que la empresa suele asignar a los programadores junior, que es la labor desempeñada por el alumno.

  En este apartado incluimos también los costes indirectos, representados en el Cuadro 2.4 que derivan de la contratación de un empleado, como pueden ser el alquiler de la oficina o el consumo eléctrico. Al haber 8 trabajadores en la empresa trabajando de forma presencial y tener una duración de 3 meses, el coste que supone el estudiante en prácticas se ha dividido por 8 y multiplicado por 3.

Recurso	Horas	Salario por hora	Salario mensual	Coste total
Programador junior	300	8,75 €	1400 €	2625,00 €
Programador full-stack (supervisor)	60	15,62 €	2500 €	937,20 €
Total				3562,20 €

Cuadro 2.3: Costes humanos.

Recurso	Coste
Consumo eléctrico	42,00 €
Consumo de agua	10,00 €
Servicio de limpieza	15,00 €
Internet	16,50 €
Alquiler	300,00 €
Mobiliario	140,00
Total	523,50 €

Cuadro 2.4: Costes indirectos.

Por tanto, sumando todos los costes que hemos ido enumerando, obtenemos un total de 4195,45 € para desarrollar la parte del proyecto que se describe en esta memoria.

Tipo de recurso	Coste total
Software	69,00 €
Hardware	41,25 €
Humanos	3562,20€
Indirectos	523,00 €
Total	4195,45 €

Cuadro 2.5: Costes totales.

2.4. Gestión de riesgos

La gestión de riesgos sirve para definir medidas ante imprevistos y problemáticas que puedan surgir a medida que avanza un proyecto, minimizando el impacto que estos tendrán en el resultado final [23]. Los principales riesgos que se tuvieron en cuenta fueron los siguientes [15]:

- Desconocimiento de las herramientas de desarrollo: es el principal riesgo, ya que es muy común que los alumnos recién egresados de la universidad no se hayan especializado y por tanto no sepan manejar las tecnologías habituales de una empresa. Para solucionar este problema, al inicio del proyecto se destinó unos días al estudio y práctica de Angular [29], entre otros.

- Distribución temporal errónea: otro problema bastante frecuente en los proyectos de desarrollo de software es la incorrecta asignación de tiempos a las tareas a realizar. Para disminuir el efecto que esto pueda tener, se asignan unos intervalos con un margen de error, aunque de haber una desviación importante se podrá acotar en alcance o redistribuir el tiempo dedicado a cada tarea.

- Descoordinación equipo *back-end* y *front-end*: desde un principio se puso en conocimiento del alumno en prácticas que el *back-end* carecía de la implementación de la mayor parte de los métodos. Esto podría restringir los avances alcanzables por parte del alumno, ya que estos métodos son esenciales para comprobar que el *front-end* funciona de la forma que se espera. La solución a este problema reside en las reuniones semanales, que actualizan al supervisor del punto en que se encuentra el alumno y, de estar cercano a necesitar más métodos, desarrollarlos y comunicárselo al alumno, que podrá visualizar los cambios en Swagger UI.

- Baja por Covid-19: este riesgo se tuvo en cuenta debido a que dos empleados contrajeron la enfermedad en 2020, lo que causó retrasos en entregas de proyectos. En caso de que algún empleado la contrajera de nuevo, se pasaría al teletrabajo los días indicados en el momento por el gobierno. Es por esto que al inicio del proyecto se configuró el entorno de desarrollo en el ordenador personal del alumno en prácticas. También se reservaría una hora para la reunión semanal con el supervisor.

2.5. Seguimiento del proyecto

En esta sección comprobaremos cómo ha avanzado el proyecto, los riesgos que se han manifestado y cómo se han solucionado estas desviaciones.

Al comienzo del proyecto tuvieron lugar varias reuniones con el fin de esclarecer los objetivos y requisitos del sistema, así como la forma de trabajar de la empresa. Se explicó el esqueleto que usa habitualmente la entidad, la división por paquetes, cómo esto afecta a la funcionalidad final de la aplicación y también se introdujo la tecnología Angular. Se estableció una serie de hitos iniciales para que, una vez alcanzados, pudieran ser comprobados por el revisor. También se dejó una ventana de tiempo para que el alumno practicara con Angular. Todas estas tareas se llevaron a cabo en el tiempo estimado.

Una vez quedó claro el camino a seguir, el alumno diseñó *mock-ups* de las interfaces que habría que implementar posteriormente. Esta tarea ocupó dos días más de lo esperado, pues en una de las reuniones que se realizaron durante la fase de diseño surgió una idea para simplificar las interfaces, lo que ocasionó el rediseño de algunos *mock-ups* ya finalizados.

Cuando se pasó a la implementación de la aplicación, desde un inicio se advirtió que el alcance inicial, que incluía el módulo de empleados, el módulo del calendario y el módulo de control de presencia (como explicaba la Propuesta Técnica) era demasiado ambicioso. La razón principal es que al ser una aplicación de corte profesional, con funcionalidades avanzadas y aspectos particulares de la empresa, el alumno avanzó a un ritmo inferior al estimado, pues aún se estaban asimilando conceptos de las tecnologías. El principal obstáculo fue Typescript, un lenguaje bastante distinto a los que se estudian en la rama de Sistemas de la Información y con una elevada curva de aprendizaje. La solución que se encontró a este problema fue la de acotar el alcance, incluyendo únicamente el primer módulo y mediante un contrato Fundación UJI-Empresas, continuar desarrollando los otros dos módulos (considerablemente menos extensos) durante los meses de agosto y septiembre.

Otro problema consistió en que una vez el alumno adquirió conocimientos suficientes, el ritmo de desarrollo se vio acelerado, razón por la que, a pesar de haberlo tenido en cuenta en la gestión de riesgos 2.4 y mantener reuniones semanales, llegó un punto en el que el alumno requería de métodos del *back-end* para poder continuar desarrollando el *front-end*. Esto se debe a que los desarrolladores responsables de desarrollo *back-end* de la aplicación (supervisor y programador junior) se vieron sobrecargados de trabajo por el encargo de otra aplicación. Para no paralizar el proyecto, se optó por dejar una serie de *placeholders*, avanzar todo lo posible en las interfaces que lo permitían y elaborar una lista de tareas pendientes a resolver una vez se hubieran implementado los métodos *back-end* necesarios. Estas tareas se revisarían durante los meses de agosto y septiembre, previamente al desarrollo del módulo de calendario (que guarda una estrecha relación con el de empleados) y el de control de presencia. También se destinó tiempo a la depuración de las interfaces ya implementadas y el refinamiento del control de errores.

Hacia el final de la estancia en prácticas, se decidió realizar una pausa de dos semanas para la redacción de la memoria del TFG por parte del alumno y el desarrollo del *back-end* por parte del supervisor y el programador junior una vez finalizaran el desarrollo de la otra aplicación.

Capítulo 3

Análisis y diseño del sistema

3.1. Análisis del sistema

En esta sección vamos a presentar el Diagrama de Casos de Uso y los requisitos funcionales de los mismos. Nos centraremos únicamente en los que el alumno tuvo que desarrollar a lo largo de la estancia en prácticas, enumerados en la Sección 1.3.

Un Diagrama de Casos de Uso permite identificar fácilmente qué debe hacer el sistema que se está desarrollando, mostrando cada uno de los casos de uso [17]. También se muestran los actores y los casos de uso a los que tendrán acceso. En el caso de Kalenda se puede distinguir entre tres actores principales: administrador, recursos humanos y empleado. El administrador se define como la persona encargada de crear nuevos usuarios para los clientes que adquieran la licencia de Kalenda y también puede modificar cualquier tipo de información disponible, mientras que los perfiles de recursos humanos y empleados son más similares, con la principal diferenciación de que el administrador actúa como un "administrador" propio de la empresa. Como se puede ver en diagrama de casos de uso representado por el Cuadro 3.1, hay algunas acciones pueden ser realizadas por distintos actores, aunque de manera y a escala diferente según el tipo de usuario (p.e: evidentemente, un empleado cualquiera no puede modificar la información de otro, ya que para eso se requieren permisos de administrador o recursos humanos).

Requisitos funcionales	**Actores**
RF01 - Iniciar sesión	Administrador, recursos humanos, empleado
RF02 - Gestión de usuarios	Administrador
RF03 - Asignar roles	Administrador
RF04 - Realizar fichaje (control de presencia)	recursos humanos, empleado
RF05 - Gestión de calendario	Administrador, recursos humanos, empleado
RF06 - Gestión de alertas	Administrador, recursos humanos, empleado
RF07 - Gestión de empleados	Administrador, recursos humanos
RF08 - Gestión de contratos	Administrador, recursos humanos, empleado
RF09 - Gestión de contactos	Administrador, recursos humanos, empleado
RF10 - Gestión de festivos	Administrador, recursos humanos

Cuadro 3.1: Requisitos Funcionales.

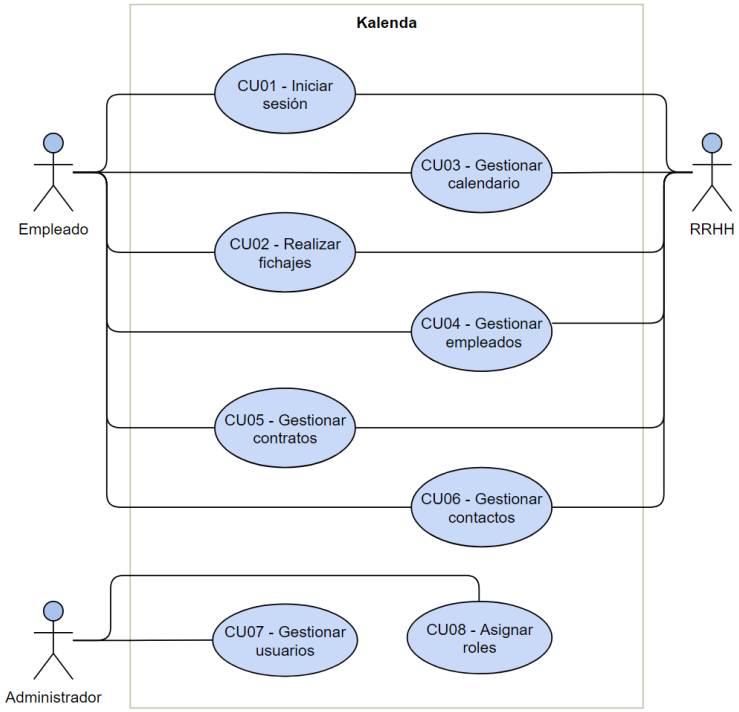


Figura 3.1: Diagrama de Casos de Uso simplificado.

A continuación se muestran los requisitos funcionales de cada caso de uso, que sirven para detallar las funciones del sistema y el comportamiento que se espera del software [17]. Cabe destacar que en los casos de uso que engloban la gestión, como "Gestión de contactos", sólo se incluye la creación de contactos (omitiendo la modificación y borrado) en pos de mayor legibilidad del documento.

Requisito funcional RF01	
Identificador	RF01
Nombre	Iniciar sesión
Autor	Lucas Niño Ruiz
Descripción	El sistema debe permitir al usuario acceder a la aplicación.
Actor principal	Recursos humanos, empleado, administrador
Precondición	El usuario está registrado en la base de datos del sistema.
Disparador	El usuario introduce sus credenciales en un formulario.
Secuencia	1. El usuario accede a la página de *login*. 2. El usuario introduce sus credenciales. 3. El usuario envía los datos al sistema. 4. El sistema comprueba si los datos son correctos y el usuario está registrado. 5. El sistema redirige a la página principal de la aplicación según el usuario.
Excepciones	1. El usuario no está registrado en la aplicación. 2. Las credenciales son incorrectas.
Prioridad	Alta

Cuadro 3.2: Requisito funcional RF01.

Requisito funcional RF02	
Identificador	RF02
Nombre	Gestión de usuarios
Autor	Lucas Niño Ruiz
Descripción	El sistema debe permitir al administrador crear un nuevo usuario.
Actor principal	Administrador
Precondición	El administrador tiene sesión iniciada.
Disparador	Un administrador quiere registrar un nuevo usuario en la aplicación.
Secuencia	1. El administrador introduce los datos del nuevo usuario. 2. El administrador envía los datos al sistema. 3. El sistema comprueba si los datos son correctos y el usuario no está registrado en la base de datos. 4. El sistema redirige al listado de usuarios.
Excepciones	1. Algún dato introducido es erróneo. 2. Las credenciales coinciden con las de un usuario ya existente.
Prioridad	Alta

Cuadro 3.3: Requisito funcional RF02.

Requisito funcional RF03	
Identificador	RF03
Nombre	Asignar roles
Autor	Lucas Niño Ruiz
Descripción	El sistema debe permitir al administrador asignar un rol a un usuario.
Actor principal	Administrador
Precondición	El administrador tiene sesión iniciada.
Disparador	Un administrador quiere asignar un rol a un usuario.
Secuencia	1. El administrador selecciona un usuario. 2. El administrador selecciona un rol. 3. El sistema actualiza el rol del usuario al seleccionado por el administrador. 4. El sistema redirige al listado de usuarios.
Excepciones	1. El rol del usuario ya era el seleccionado.
Prioridad	Media

Cuadro 3.4: Requisito funcional RF03.

Requisito funcional RF04	
Identificador	RF04
Nombre	Realizar fichaje
Autor	Lucas Niño Ruiz
Descripción	El sistema debe permitir a los empleados de una empresa dejar constancia de las horas laborales.
Actor principal	Recursos humanos, empleado
Precondición	El usuario tiene sesión iniciada.
Disparador	Un usuario quiere indicar que comienza o termina su jornada.
Secuencia	1. El usuario selecciona la opción de fichar. 2. El sistema registra la hora.
Excepciones	-
Prioridad	Baja

Cuadro 3.5: Requisito funcional RF04.

Requisito funcional RF05	
Identificador	RF05
Nombre	Gestión de calendario
Autor	Lucas Niño Ruiz
Descripción	El sistema debe permitir al usuario añadir eventos a su calendario.
Actor principal	Administrador, recursos humanos, empleado
Precondición	El usuario tiene sesión iniciada.
Disparador	Un usuario quiere añadir un evento a su calendario.
Secuencia	1. El usuario selecciona uno de los días del calendario. 2. El sistema muestra los eventos de ese día. 3. El usuario introduce los datos del evento. 4. El sistema registra el evento. 5. El sistema redirige al usuario al calendario.
Excepciones	1. La zona horaria ya está ocupada. 2. El día seleccionado es festivo.
Prioridad	Alta

Cuadro 3.6: Requisito funcional RF05.

Requisito funcional RF06	
Identificador	RF06
Nombre	Gestión de alertas
Autor	Lucas Niño Ruiz
Descripción	El sistema debe permitir al usuario crear alertas.
Actor principal	Administrador, recursos humanos, empleado
Precondición	El usuario tiene sesión iniciada.
Disparador	Un usuario quiere crear una alerta.
Secuencia	1. El usuario selecciona el tipo de alerta. 2. El usuario introduce los datos de la alerta. 3. El sistema registra la alerta. 4. El sistema muestra la alerta en el día y hora indicados. 5. El usuario cierra la alerta.
Excepciones	-
Prioridad	Media

Cuadro 3.7: Requisito funcional RF06.

Requisito funcional RF07	
Identificador	RF07
Nombre	Gestión de empleados
Autor	Lucas Niño Ruiz
Descripción	El sistema debe permitir al usuario crear empleados.
Actor principal	Administrador, recursos humanos
Precondición	El usuario tiene sesión iniciada.
Disparador	Se quiere crear un nuevo empleado para la empresa.
Secuencia	1. El usuario selecciona la opción crear empleado. 2. El sistema muestra un diálogo multipasos. 3. El usuario introduce los datos de cada paso, a saber: información de empleado, de contacto, de contrato, de jornada laboral, de puesto de trabajo. 4. El sistema muestra un paso resumen. 5. El usuario comprueba que la información es correcta. 6. El sistema registra todos los datos y crea el empleado.
Excepciones	1. El empleado ya está registrado en la base de datos. 2. Alguno de los datos introducidos es incorrecto.
Prioridad	Alta

Cuadro 3.8: Requisito funcional RF07.

Requisito funcional RF08	
Identificador	RF08
Nombre	Gestión de contratos
Autor	Lucas Niño Ruiz
Descripción	El sistema debe permitir al usuario crear contratos.
Actor principal	Administrador, recursos humanos, empleado
Precondición	El usuario tiene sesión iniciada.
Disparador	Un usuario quiere crear un contrato.
Secuencia	1. El usuario selecciona la opción para crear un contrato. 2. El sistema redirige a un formulario. 3. El usuario introduce los datos. 4. El sistema comprueba si los datos son correctos y añade el contrato a la base de datos.
Excepciones	1. El contrato se solapa con otro actual. 2. Algún dato introducido es incorrecto.
Prioridad	Alta

Cuadro 3.9: Requisito funcional RF08.

Requisito funcional RF09	
Identificador	RF09
Nombre	Gestión de contactos
Autor	Lucas Niño Ruiz
Descripción	El sistema debe permitir al usuario crear contactos.
Actor principal	Administrador, recursos humanos, empleado
Precondición	**E**l usuario tiene sesión iniciada.
Disparador	Un usuario quiere crear un nuevo contacto.
Secuencia	1. El usuario selecciona la opción para crear un contacto. 2. El usuario introduce los datos. 3. El sistema registra el nuevo contacto. 4. Si el contacto se marca como favorito y ya había otro contacto del mismo tipo marcado como favorito, este último se desmarca.
Excepciones	1. Algún dato introducido es incorrecto. 2. El trabajador ya tiene ese contacto asignado.
Prioridad	Alta

Cuadro 3.10: Requisito funcional RF09.

Requisito funcional RF10	
Identificador	RF10
Nombre	Gestión de festivos
Autor	Lucas Niño Ruiz
Descripción	El sistema debe permitir a los usuarios crear festivos.
Actor principal	Administrador, recursos humanos
Precondición	El usuario tiene sesión iniciada.
Disparador	Un usuario quiere crear un festivo.
Secuencia	1. El usuario selecciona la opción para crear un festivo. 2. El usuario introduce los datos. 3. El sistema registra el festivo.
Excepciones	1. Algún dato introducido es incorrecto. 2. El festivo creado ya estaba registrado.
Prioridad	Media

Cuadro 3.11: Requisito funcional RF10.

3.2. Diseño de la arquitectura del sistema

Esta sección describe el diseño de la arquitectura, que en esta aplicación es de tipo microservicios [37]. En este aspecto, la versión obsoleta de Kalenda contaba con 6 servicios (solicitudes, calendario, informes, empresa, fichajes y autenticación), que se han unificado en 2 (autenticación y funcionalidad) con el propósito de mejorar la eficiencia. Como en este proyecto el alumno ha trabajado en la parte cliente de la aplicación con Angular, se detallará el patrón MVC (*Model View Controller*, Modelo Vista Controlador) [24]. Dicho patrón es muy frecuente en este tipo de aplicaciones, ya que permite la separación entre los datos y la lógica de negocio, la representación mediante interfaces y la gestión de eventos. Los distintos elementos constituyentes de este patrón de diseño, que se pueden ver en la Figura 3.2, son los siguientes:

- Modelo: son los datos. Se encarga de representar la información que maneja el sistema, gestionando los accesos y modificaciones y manteniendo la vista actualizada.
- Vista: es lo que ve el usuario final. Presenta los datos solicitados al modelo (que estarán actualizados) en interfaces gráficas de usuario con un formato fácilmente operable por usuarios humanos.
- Controlador: gestiona el flujo de información de la aplicación y las operaciones a ejecutar sobre los datos. Comunica la vista y el modelo, pues actúa sobre el modelo y una vez recibe los datos del mismo, los envía a la vista.

La aplicación abarca dos módulos: el del servidor y el del cliente. El primero se relaciona más con el modelo del patrón MVC, conectando con la base de datos, mientras que el segundo representa los datos y ofrece interacción al usuario. Nos centraremos en el del cliente, pues recordemos que el alumno en prácticas era el encargado de desarrollar parte del *front-end*, que usa la tecnología Angular. Precisamente por el uso de este framework conviene detallar su impacto en el patrón MVC.

Como se puede observar en la Figura 3.3, en Angular las vistas se identifican con plantillas codificadas en HTML y CSS, mientras que los controladores están programados en Typescript y se corresponden a los servicios. Estos servicios envían peticiones al servidor, que es el que conecta con

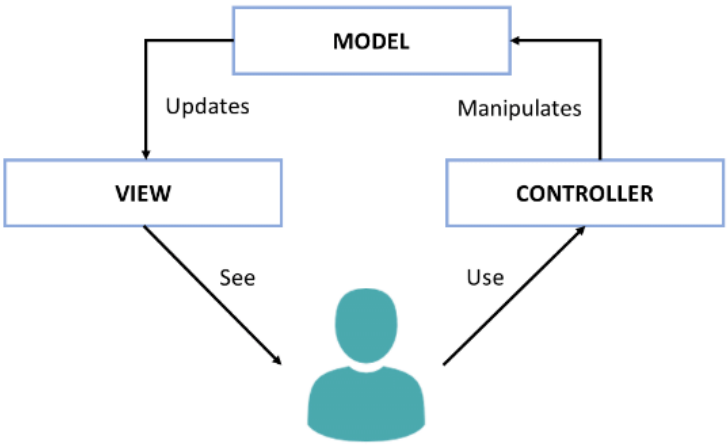


Figura 3.2: Esquema MVC [46].

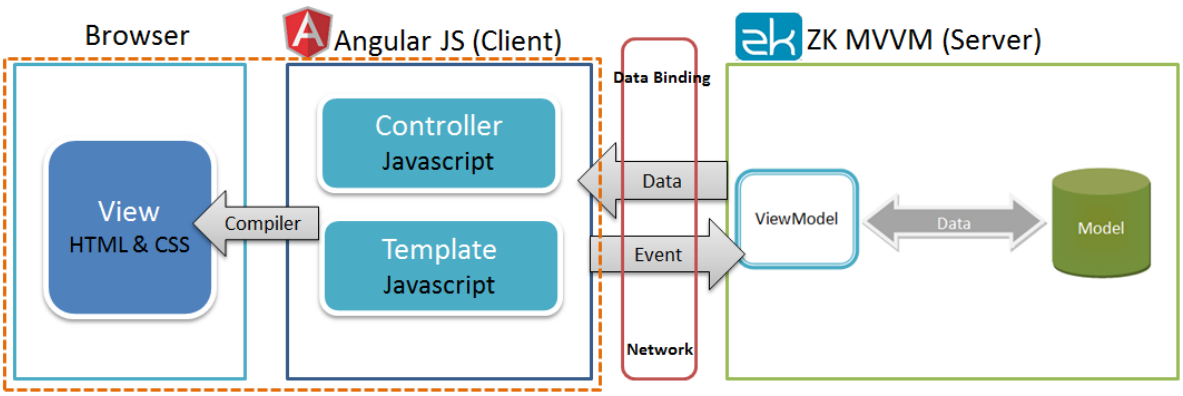


Figura 3.3: Concepto Angular + MVC [21].

la base de datos (es decir, el modelo) y devuelve al controlador, situado en el cliente, la información correspondiente.

Angular se estructura por componentes, que normalmente están compuestos de un fichero CSS, uno HTML y otro Typescript [7]. Pueden ser genéricos, es decir, públicos para que todo el mundo pueda usarlos, de bibliotecas específicas, o propios, que suele ser el caso cuando se busca un comportamiento e interacción con el entorno muy específicos. Esta es una cualidad que lo dota de una gran modularidad, ya que de estar bien ideados, podremos usar unos dentro de otros estableciendo jerarquías padre-hijo. Todos estos aspectos los detallaremos en la Sección 4.1, exponiendo ejemplos que se dan en el proyecto.

La base de datos fue poblada previamente al comienzo de la estancia en prácticas para disponer de datos con los que trabajar, ya que forma parte del *back-end*, que no se incluye en esta memoria porque no lo desarrolló el alumno, como también ocurre con el diseño de la base de datos. Muchos de estos datos están incompletos o erróneos, lo que también sirve para validar la funcionalidad de la aplicación una vez implementada. Tampoco se incluyen los requisitos de datos en esta sección puesto que las tablas han sido creadas previamente y los datos (atributos y otros) están sujetos a las necesidades en la implementación del proyecto.

Por último están los maestros, que son tablas relacionadas con la información principal del sistema. Un ejemplo de maestros son los títulos educativos, necesarios para crear un contrato, o las nomenclaturas, gracias a las cuales se pueden obtener los datos a cargar en atributos de tipo *select*. Se profundiza en los maestros en la Sección 4.1.

3.3. Diseño de la interfaz

En esta sección se van a explicar los criterios seguidos para el diseño de interfaces a implementar. Para ilustrar el proceso, se usarán imágenes de varios *mock-ups* (bocetos) desarrollados por el alumno.

La toma de decisiones en cuanto al diseño de las vistas constó de varias reuniones entre el supervisor y el alumno en prácticas al inicio del proyecto, en las que se focalizaron esfuerzos sobre todo en la navegación por la interfaz, así como en dotarla de un aspecto simétrico [18, 31, 36].

A continuación se enumeran una serie de decisiones tomadas en las citadas reuniones:

- El principal problema que encontrábamos en la versión de Kalenda desarrollada en 2016 era la desorganización, pues disponía de muchas opciones que eran difícil de encontrar, lo que restaba eficiencia al usuario. Para ello, como ya se menciona en la Sección 1.3, se han reducido el número de opciones en el menú lateral, que sí se ha mantenido para dividir la aplicación por funcionalidades.
- Se ha mantenido el menú superior, relegado a opciones de configuración de la aplicación, como los botones de cerrar sesión o el de cambiar idioma. También muestra el logo de la empresa en la esquina superior izquierda.
- Anteriormente, en algunos listados se podía realizar acciones sobre los datos mostrados. Por ejemplo, añadir una baja a un contrato mediante un botón que abría un diálogo. Esto se ha desestimado, ya que los listados únicamente deberían servir para mostrar datos. Ahora, en cambio, se podría abrir el contrato deseado y añadir la baja al mismo.
- Los listados que mostraban objetos sencillos y con pocos atributos ahora permiten las operaciones de creación, modificación y borrado en el propio listado, sin necesidad de cargar una página aparte para el objeto en concreto.
- Si la creación de alguna entidad requiere de varios pasos, se utiliza un diálogo multipasos, con un resumen de los datos introducidos como último paso.
- Si la página de detalle de alguna entidad consta de varios subapartados, se debe hacer uso de pestañas para la organización de los mismos.
- Todas las interfaces deben poder mostrar la información en una única pantalla, minimizando al máximo el uso del *scroll* en el eje vertical.
- Los campos de los formularios deben tener siempre el mismo tamaño, a excepción del uso de algunos *inputs* que justifiquen el cambio de dimensiones.
- En la aplicación anterior se usaban un azul, rojo y verde fuertes para resaltar acciones. En la versión actual se han seleccionado unos iconos más descriptivos y en color grisáceo para mantener una apariencia más corporativa. Para propiciar esto, también se ha usado un tono azul más oscuro para los títulos o el *banner* superior.
- Se ha mantenido el color gris claro de fondo, mientras que los formularios, listados y menús utilizan un fondo blanco. Esto permite identificar más fácilmente los elementos mostrados en pantalla.
- Los botones, salvo en diálogos, no contienen texto alguno.

Todos estos aspectos se mantienen en la aplicación de forma transversal para reforzar la cohesión.

En la Figura 3.4 vemos el listado de empleados, en el que se muestran los atributos más importantes. De querer abrir la vista detalle, basta con seleccionar cualquiera de los atributos, ya que el botón que aparece con el icono de un ojo se desestimó. La pantalla que se mostraría entonces es la que aparece en la Figura 3.5, en la que podemos ver toda la información relativa al empleado seleccionado. También se puede percibir que los datos no son editables, pues para modificarlos hay que pulsar el botón en la esquina superior derecha.

Figura 3.4: *Mock-up* listado de empleados.

Figura 3.5: *Mock-up* detalle información empleado.

En el caso del empleado, crear uno nuevo conlleva asignarle un contrato y un puesto de trabajo, entre otros. Es por esto que, como mencionamos en una de las decisiones anteriores, debe usarse un asistente multipasos. Las Figuras 3.6 y 3.7 muestran los pasos descritos.

Por último, se muestra en la Figura 3.8 un tipo de listado mucho más sencillo que el de empleado, ya que los contactos tienen mucha menos información. Por esta razón se se puede realizar la operación de creación en el propio listado. También cabe destacar la parte superior del prototipo, pues fue una idea de navegación descartada debido a la innecesaria complejidad que añadía a la navegación.

Figura 3.6: *Mock-up* diálogo creación de empleado, paso contrato.

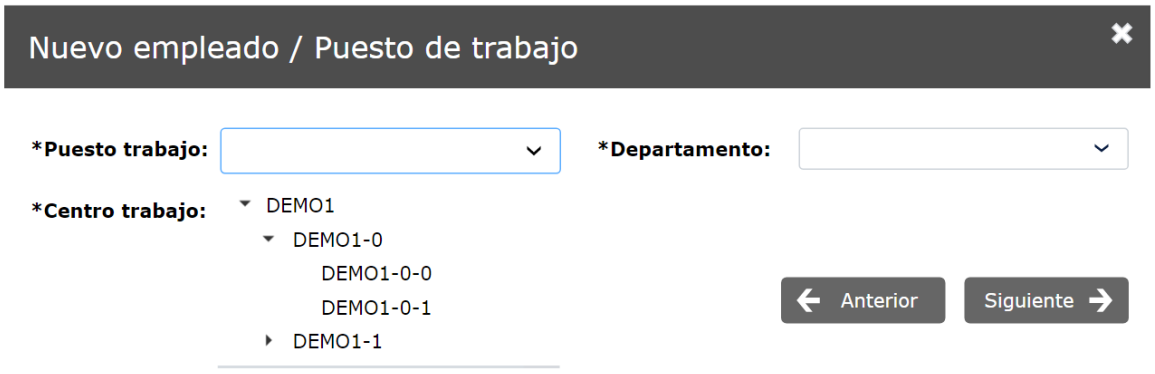


Figura 3.7: *Mock-up* diálogo creación de empleado, paso puesto de trabajo.

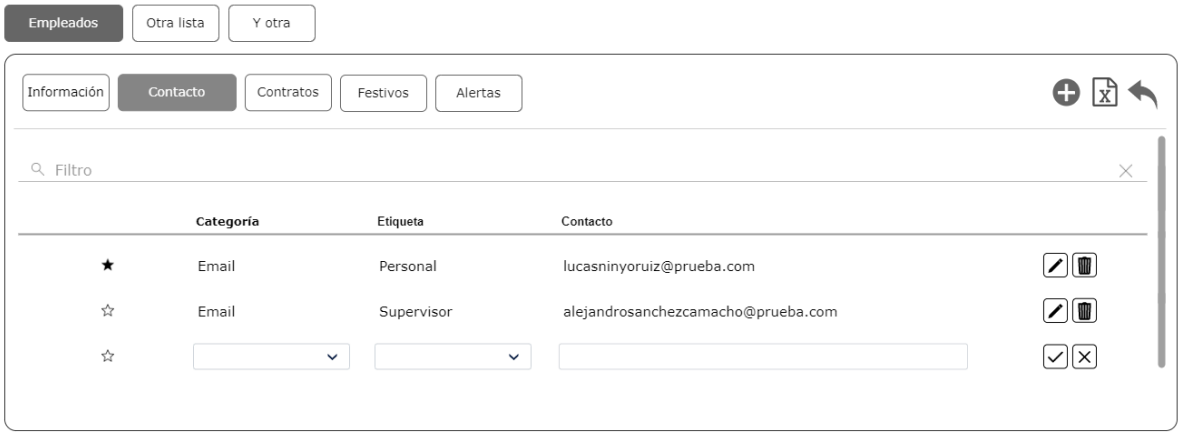


Figura 3.8: *Mock-up* creación contacto.

Capítulo 4

Implementación y pruebas

4.1. Detalles de implementación

En esta sección se explica el proceso de implementación en detalle, profundizando en los patrones de programación empleados y las estrategias seguidas. Primero se expone la estructura que sigue el proyecto, y después se comentan el desarrollo, los resultados obtenidos, los problemas que han aparecido y las soluciones que se han encontrado.

4.1.1. Estructura del proyecto

La estructura del proyecto es la que marca Integra, ya que tras muchos proyectos han encontrado que esta es la forma más clara y eficiente de dividir los paquetes. Al inicio de la implementación se dieron algunas pautas al alumno respecto a la organización de nuevos ficheros y directorios, siendo la máxima el separar siempre por funcionalidad.

Recordemos que para desarrollar el *front-end* se utilizó el framework Angular, para lo que la empresa tiene un esqueleto inicial asociado [6]. Este esqueleto cuenta con algunos módulos y componentes desarrollados por la empresa y presentes en la mayoría de proyectos, como un menú lateral o grids para listar elementos.

La estructura general se puede observar en la Figura 4.1. Muchos de estos archivos son de configuración del entorno Node.js, mientras que en los directorios “common” y “shared” se encuentran los módulos y componentes mencionados en el párrafo anterior, que serán usados a conveniencia para no tener que volver a implementar elementos que ya se han usado en otros proyectos. Los directorios en los que el alumno ha desarrollado la mayor parte de su labor son el de “infra”, “empleado”, “maestros” y “session”.

Figura 4.1: Estructura del proyecto y archivos de configuración.

4.1.2. Desarrollo y resultados obtenidos

Listado de empleados

La primera vista desarrollada se trata del listado de empleados, que se explica con detalle a continuación. También se crearon listados de contactos, contratos, festivos y empresas, que salvo algunas diferencias que no se mencionarán aquí para evitar la redundancia y no alargar la memoria, siguen un patrón similar.

En primer lugar tenemos el Listing 4.1, que muestra las propiedades y la inicialización de las mismas del componente, así como el constructor, que recibe como parámetros los servicios necesarios para realizar las operaciones necesarias [2]. Estas propiedades se asignan a los componentes "app-loader" y "lib-listado" (Listing 4.2 [38]) en lo que en Angular se conoce como *property binding* [9]. Estos componentes, disponibles en el directorio "shared" que ya mencionamos, son los que forman nuestro listado de empleados.

```
export class BuscadorEmpleadosComponent implements OnInit {
  empleados: Array<EmpleadoModel>;
  editable = true;
  configuracionGrid = Object.assign({}, EmpleadosConfig);
  listasValores: any;
  loading = false;
  filtros = new FiltroEmpleadoModel().initialize(
    this.configuracionGrid.columnas);
  cargaInicial = true;
  nuevoModel = new EmpleadoModel().initialize();

  constructor(
    public http: _HttpClient,
    private router: Router,
    public service: EmpleadosService,
    private storeService: EmpleadosStoreService) {
  }
```

Listing 4.1: Propiedades y constructor del listado de empleados.

```
<app-loader [loading]="loading"></app-loader>
<nz-card class="margin-no padding-md">
  <lib-listado [elementos]="empleados"
               [configuracionGrid]="configuracionGrid"
               [listasValores]="listasValores"
               [modelNuevo]="nuevoModel"
               [filtros]="filtros"
               [editable]="editable"
               (editarEvent)="editar($event)"
               (nuevoEvent)="nuevo()"
               (eliminarEvent)="eliminar($event)"
               (refrescarEvent)="recargar($event)">
  </lib-listado>
</nz-card>
```

Listing 4.2: Asignación de propiedades y eventos al listado de empleados.

Además de la configuración inicial, se necesitan los métodos que aparecen en el Listing 4.3 para el correcto funcionamiento del componente. Lo que se muestra en el código es el gancho de ciclo de

vida (*lifecycle hook*) [8] más importante de Angular, "ngOnInit", que una vez completado indica que Angular ya ha creado el componente. Un componente no se creará hasta que el gancho de ciclo de vida finalice su ejecución. En este método se incluye la carga de las listas de valores constantes y la de todos los empleados almacenados en el servidor mediante el uso de los servicios pasados como parámetros al constructor. El constructor se diferencia del gancho en que únicamente se ejecuta una vez el componente se haya instanciado y sirve para la correcta inicialización de los campos de la clase y las subclases [48].

```
1   ngOnInit() {
2     this.obtenerDatosRelacionados();
3   }
4
5   obtenerDatosRelacionados() {
6     this.listasValores = { activos: LISTA_ACTIVOS };
7   }
8
9   cargarDatos() {
10    this.loading = true;
11    this.storeService.empleadosFiltrados$(this.filtros).subscribe((response:
       Array<any>) => {
12      if (response) {
13        this.empleados = response.map((not: EmpleadoModel) => new
      EmpleadoModel().deserialize(not));
14        this.loading = false;
15      }
16    });
17  }
```

Listing 4.3: Estructura del gancho de ciclo de vida para el listado de empleados.

También se puede observar en la línea 4 del Listing 4.1 que se asigna una configuración al componente. Esta configuración la podemos ver en detalle en el Listing 4.4. En ella se determina la funcionalidad de la misma: las columnas que se muestran, el título, el tipo de interacción (si abre un diálogo o redirige a una nueva página), los botones disponibles... Al ser un fichero muy extenso, únicamente se muestra la configuración de una columna.

```
1  export const EmpleadosConfig: IGridConfiguration = {
2      nombre: 'gridEmpleados',
3      eliminar: false,
4      editar: true,
5      botonera: {
6          mostrarNuevo: true,
7          nuevoConDialogo: true,
8
9          tituloDialogo: 'Nuevo empleado',
10         dialogo: DialogEmpleado,
11         with: 75,
12
13         titulo: 'app.listado',
14         tituloSeparado: 'empleados.buscador',
15         mostrarFiltro: true,
16         mostrarColumnas: true,
17         mostrarExportacion: true,
18     },
19     columnas: [
20         {
21             width: '10%',
22             name: 'empleados.numero',
```

```
            field: 'idPropio',
            sort: true,
            selected: true,
            editable: false,
            required: true,
            type: 'text',
            typeFilter: 'text',
            order: 1
        },
    ...
    ]
};
```

Listing 4.4: Estructura del gancho de ciclo de vida.

También están en el propio componente los métodos necesarios para realizar las operaciones de creación, modificación y eliminación de empleados. Como se puede ver en el Listing 4.5, el método "nuevo" únicamente redirige al listado de empleados, pues se relega la creación a un diálogo multipasos que se detalla más adelante en la sección. Todos estos métodos se asocian a los eventos del componente "lib-listado", como se observa en el Listing 4.2. La forma en la que Angular ejecuta estos métodos es suscribiéndose a un objeto de tipo "observable" para devolver el último objeto que este emita.

```
  nuevo() {
    this.router.navigate(['/empleados']);
  }

  editar(empleado: EmpleadoModel) {
    this.router.navigate(['/empleados/detalle', empleado.id]);
  }

  eliminar(id: number) {
    this.service.delete(id).subscribe(() => {
      this.storeService.setEmpleadosFiltrados(this.filtros);
    }, (error) => {
      this.storeService.setEmpleadosFiltrados(this.filtros);
    });
  }

  recargar(filtros: any) {
    this.filtros = filtros;
    if (this.cargaInicial) {
      this.cargarDatos();
      this.cargaInicial = false;
    } else {
      this.storeService.setEmpleadosFiltrados(this.filtros);
    }
  }
```

Listing 4.5: Métodos CRUD para empleados.

Tanto en la línea 11 del Listing 4.3 como en los métodos del Listing 4.5, que se encuentran en el componente del buscador de empleados, se emplean métodos del servicio que mantiene comunicación con el servidor. La definición de los métodos de este servicio se ha implementado siguiendo la documentación disponible en Swagger UI (Figura 4.2), desde donde también se puede consultar los datos en detalle, dando lugar al fichero que se muestra en el Listing 4.6.

Figura 4.2: Documentación de la API mostrada por Swagger UI.

```
export class EmpleadosService {
  urlBase = environment.apiUrl + '/' + environment.apiVersion + '/empleados'
    ;

  constructor(public dataService: DataService) {
  }

  listar(): Observable<any> {
    const parametros = new HttpParams().set('soloActivos', 'false');
    return this.dataService.get(this.urlBase, parametros);
  }

  filtrados(filtros: any): Observable<any> {
    const parametros = new HttpParams().set('soloActivos', 'false');
    return this.dataService.get(this.urlBase, parametros);
  }

  get(id: any): Observable<any> {
    return this.dataService.get(this.urlBase + '/' + id, null);
  }

  post(datos: EmpleadoModel): Observable<any> {
    return this.dataService.post(this.urlBase, datos);
  }

  put(id: number, datos: EmpleadoModel): Observable<EmpleadoModel> {
    return this.dataService.put(this.urlBase, datos, id);
  }

  delete(id: number): Observable<any> {
    return this.dataService.delete(this.urlBase, id);
  }

}
```

Listing 4.6: Servicios del empleado.

El servidor devuelve datos del empleado [47], cuyo modelo es el que podemos ver en el Listing 4.7. Muchos de los atributos no se muestran en el propio listado, pero son necesarios tanto para la modificación de la información de un empleado como para el correcto funcionamiento de los componentes. El método "prepare", que aparece al final, sirve para modificar los datos a nuestro beneficio, ya que, por ejemplo, en este caso, el servidor devuelve un nombre y un apellido, pero lo interesante en el listado es mostrar ambos como un único atributo al que llamamos "descripción".

```
export class EmpleadoModel extends BaseModel {
 descripcion: string;
 nombre: string;
 apellidos: string;
 idPropio: string;
 dni: string;
 nuss: string;
 fechaNacimiento: Date;
 alias: string;
 usuarioId: number;
 email: string;
 telefono: string;
 nombreUsuarioConDominio: string;
 tipoEmpleadoId: number;
 tipoEmpleadoDescripcion: string;
 nivelFormativoId: number;
 nivelFormativoDescripcion: string;
 fechaAntiguedad: Date;
 direccion: string;
 organigramaCentroTrabajoId: number;
 organigramaCentroTrabajoDescripcion: string;
 organigramaDepartamentalId: number;
 organigramaDepartamentalDescripcion: string;
 organigramaPuestoTrabajoId: number;
 organigramaPuestoTrabajoDescripcion: string;
 empresaNombre: string;
 activo: boolean;
 activoMostrar: string;
 responsableDirectoId: number;
 responsableDirectoDescripcion: string;

 initialize() {
   this.activo = true;
   return this;
 }

 deserialize(input: any) {
   Object.assign(this, input);
   return this.prepare();
 }

 prepare(): this {
   if (this.activo) { this.activoMostrar = 'app.si'; }
   else { this.activoMostrar = 'app.no'; }
   this.descripcion = this.nombre + ' ' + this.apellidos;
   return this;
 }

}
```

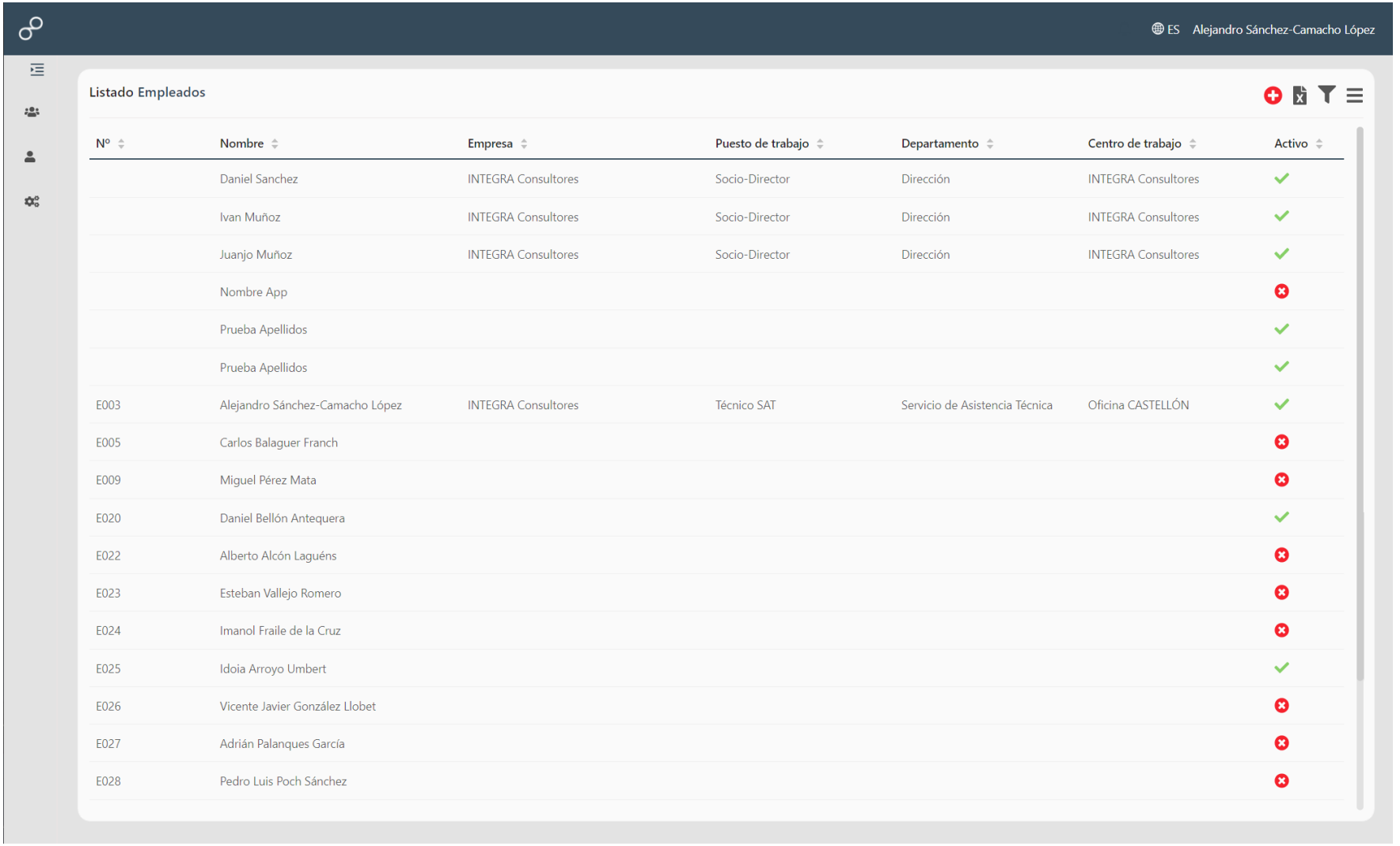
Listing 4.7: Modelo del empleado.

De momento, los filtros no funcionan, porque no se ha terminado su implementación en el *back-end*, aunque ya se han incluido en el *front-end* debido a que no impiden el funcionamiento de la aplicación. La configuración de los filtros para el listado de empleados se puede ver en el Listing 4.8. Estos filtros se pueden seleccionar gracias a un componente desarrollado por la empresa que se añade a la parte superior del listado.

```
export class FiltroEmpleadoModel extends BaseModel {
  descripcion: string;
  nomenclaturaTipo: number;
  activoId: number;

  initialize(columnas: Array<GridColumnItem>): any {
    this.generarFiltros(columnas);
    return this;
  }

  deserialize(input: any) {
    Object.assign(this, input);
    return this.prepare();
  }

  parse() {
    return this;
  }

  generarFiltros(columnas: Array<GridColumnItem>) {
    this.obtenerDatosColumna(columnas, 'descripcion', 'descripcion');
    this.obtenerDatosColumna(columnas, 'nomenclaturaTipoDescripcion', '
    nomenclaturaTipo');
    this.obtenerDatosColumna(columnas, 'activo', 'activoId');
  }

  obtenerDatosColumna(columnas: Array<GridColumnItem>, nombreCol: string,
    nombreCampo: string) {
    this.limpiarValores(columnas, nombreCampo);
    const columna = columnas.filter((col) => col.field === nombreCol &&
    col.value);
    if (columna && columna.length > 0) {
      this[nombreCampo] = columna[0].value;
    }
  }

  limpiarValores(columnas: Array<GridColumnItem>, nombreCampo: string) {
    columnas.map(() => {
      this[nombreCampo] = null;
    });
  }

}
```

Listing 4.8: Configuración de filtros para los empleados.

El aspecto final de la interfaz que muestra el listado de empleados es el de la Figura 4.3. Si lo comparamos con el *mock-up* de la Figura 3.4, se puede apreciar que son muy similares.



Nº	Nombre	Empresa	Puesto de trabajo	Departamento	Centro de trabajo	Activo
	Daniel Sanchez	INTEGRA Consultores	Socio-Director	Dirección	INTEGRA Consultores	✓
	Ivan Muñoz	INTEGRA Consultores	Socio-Director	Dirección	INTEGRA Consultores	✓
	Juanjo Muñoz	INTEGRA Consultores	Socio-Director	Dirección	INTEGRA Consultores	✓
	Nombre App					✗
	Prueba Apellidos					✓
	Prueba Apellidos					✓
E003	Alejandro Sánchez-Camacho López	INTEGRA Consultores	Técnico SAT	Servicio de Asistencia Técnica	Oficina CASTELLÓN	✓
E005	Carlos Balaguer Franch					✗
E009	Miguel Pérez Mata					✗
E020	Daniel Bellón Antequera					✓
E022	Alberto Alcón Laguéns					✗
E023	Esteban Vallejo Romero					✗
E024	Imanol Fraile de la Cruz					✗
E025	Idoia Arroyo Umbert					✓
E026	Vicente Javier González Llobet					✗
E027	Adrián Palanques García					✗
E028	Pedro Luis Poch Sánchez					✗

Figura 4.3: Listado de empleados.

Vista detalle de empleado

Tras haber entendido el funcionamiento del listado de empleados, se estudiará ahora el proceso de implementación de la vista de detalle del empleado. Para poder llegar hasta ella, se ha configurado el listado de manera que baste pulsar cualquier atributo de un empleado en concreto. En Angular son necesarios los módulos de enrutamiento para poder navegar por la aplicación, ya que se definen para asignar un módulo o componente a cada una de las *urls* de la aplicación [4]. Los que se muestran en los Listings 4.9 y 4.10 son los que se usan para poder navegar hasta el detalle del empleado.

```
const routes: Routes = [
  {
    path: '',
    component: LayoutDefaultComponent,
    children: [
      { path: '', redirectTo: 'dashboard', pathMatch: 'full' },
      { path: 'dashboard', loadChildren: () => import('./dashboard/
    dashboard.module').then(m => m.DashboardModule) },
      { path: 'usuarios', loadChildren: () => import('./usuarios/
    usuarios.module').then(m => m.UsuariosModule) },
      { path: 'empleados', loadChildren: () => import('./empleado/
    empleado.module').then(m => m.EmpleadoModule) },
      { path: 'maestros', loadChildren: () => import('./maestros/
    maestros.module').then(m => m.MaestrosModule) },
      { path: 'exception', loadChildren: () => import('./session/views/
    exception/exception.module').then(m => m.ExceptionModule) },
    ]
  }
];
```

Listing 4.9: Módulo de rutas principales de la aplicación.

```
const routes: Routes = [
  { path: '', component: BuscadorEmpleadosComponent, data: { title: '
    Empleados' } },
  { path: 'detalle/:id', component: EmpleadoComponent, data: { title: '
    Empleado' } },
  { path: 'notificaciones', component: BuscadorNotificacionesComponent,
    data: { title: 'Notificaciones' } },
];
```

Listing 4.10: Módulo de rutas del empleado.

Como se expone a continuación, para desarrollar la vista detalle del empleado, que contiene varias interfaces, se requiere de muchos componentes externos. Para poder usarlos, tanto en esta vista como en el buscador, es necesario importarlos y declararlos como constante en el módulo correspondiente (en este, caso, el módulo de empleados) [10]. La forma de hacerlo se muestra en el Listing 4.11.

```
const COMPONENTS = [
  BuscadorNotificacionesComponent, BuscadorContratosComponent,
  ContratoComponent, BuscadorContactosComponent,
  DialogNotificacionComponent, BuscadorEmpleadosComponent,
  EmpleadoComponent, BuscadorFestivosComponent,
  ResumenLateralComponent, PasoEmpleadoComponent,
  PasoContratoComponent, PasoJornadaComponent,
  PasoPuestoTrabajoComponent, PasoResumenComponent, DialogEmpleado,
];
```

Listing 4.11: Declaración de componentes en el módulo empleado.

Una vez se dispone de todos los componentes, se puede comenzar a desarrollar la vista compuesta. Cuando se selecciona un empleado, se abre su vista detalle, que está formada por un resumen lateral permanente y una sección con distintas pestañas, que incluyen: información, contratos, contactos, festivos y alertas. Excepto la primera pestaña, el resto son listados, por lo que funcionan de forma similar al listado de empleados y no se detallarán. En las Figuras 4.4, 4.5, 4.6 y 4.7 se pueden ver algunos de los ejemplos descritos.

Para poder mostrar todos estos componentes en una misma pantalla, se han introducido en cada una de las pestañas del componente "tabset" proporcionado por la biblioteca NG-ZORRO. Esto se ha realizado de la forma que muestran los Listings 4.12, 4.13 y 4.14.

```
<div nz-col [nzSpan]="6">
      <resumen-lateral [empleado]="empleado"></resumen-lateral>
</div>
```

Listing 4.12: Comunicación del modelo empleado al resumen lateral.

```
<nz-tab nzTitle="Informacion">
    <div style="padding-bottom: 35px;">
        <lib-botonera-detalle [editable]="editable" [form]="form" [modelo]="
    empleado" [esContraible]="false" (guardarChange)="guardar()" (
    editarChange)="editable = !editable" (volverChange)="volver()">
        </lib-botonera-detalle>
    </div>
```

Listing 4.13: Asociación de propiedades y eventos en la pestaña de información del empleado.

```
<nz-tab nzTitle="Contratos">
    <buscador-contratos [(idEmpleado)]="empleado.id" (
    mostrarBuscadorContratosEvent)="cambioMostrarBuscadorContratos($event)"
    (contratoIdEvent)="cambioIdContrato($event)" *ngIf="
    mostrarBuscadorContratos">
    </buscador-contratos>
    <contrato [(idEmpleado)]="empleado.id" [(idContrato)]="idContrato" (
    mostrarBuscadorContratosEvent)="cambioMostrarBuscadorContratos($event)"
    (contratoIdEvent)="cambioIdContrato($event)" *ngIf="!
    mostrarBuscadorContratos">
    </contrato>
</nz-tab>
```

Listing 4.14: Introducción del listado de contratos en una pestaña.

En el Listing 4.14 se podría haber mostrado la forma de introducir el listado de contactos o festivos, pero se muestra la de introducir el listado de contratos ya que es peculiar y se distingue del resto en que al crear un nuevo contrato, se debe mostrar una nueva vista **sin cargar otra página**. Esto no siempre estuvo pensado así [35] [44], ya que inicialmente para crear o editar un elemento de un listado relativo a un empleado la aplicación redirigía a otra página. Se tomó la solución de cargarlo todo en una misma página para evitar la recarga de todos los elementos de un empleado cada vez que se quiera añadir o modificar un contrato, así como para mantener en todo momento el resumen lateral presente. Para que la aplicación funcione de esta manera lo que se ha hecho es crear una propiedad booleana que sirve para comprobar si se debe mostrar el listado o la vista de creación o detalle de un contrato. Esta tarea la lleva a cabo la directiva "*ngIf", que dependiendo del valor de la mencionada propiedad, muestra una vista u otra en la pestaña de contratos.

Figura 4.4: Información de un empleado.

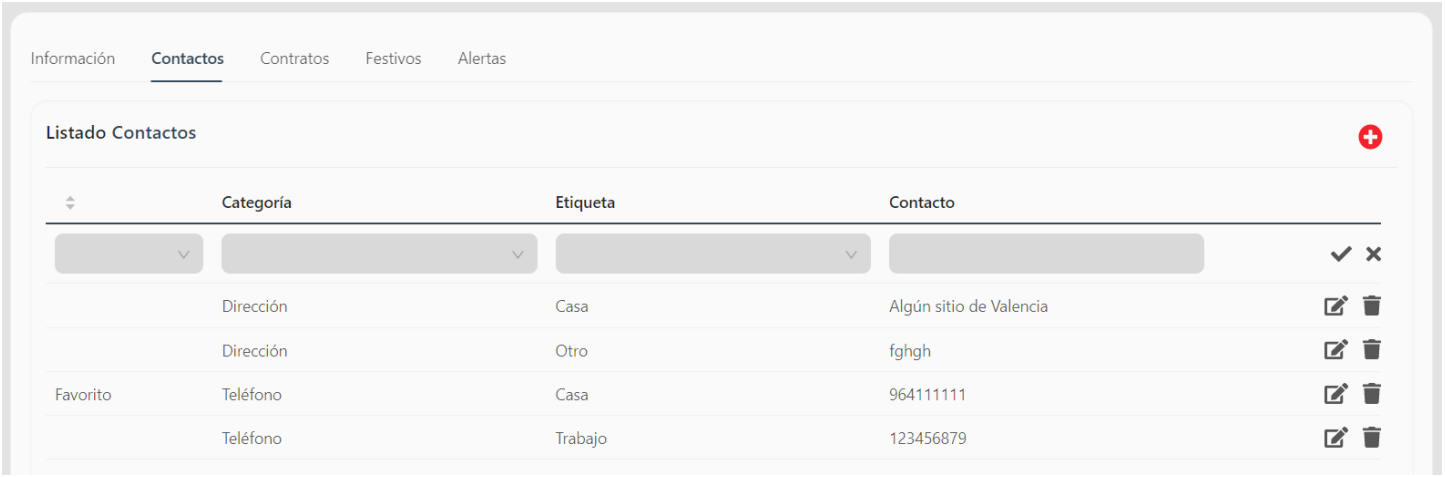


Figura 4.5: Listado de contactos de un empleado.

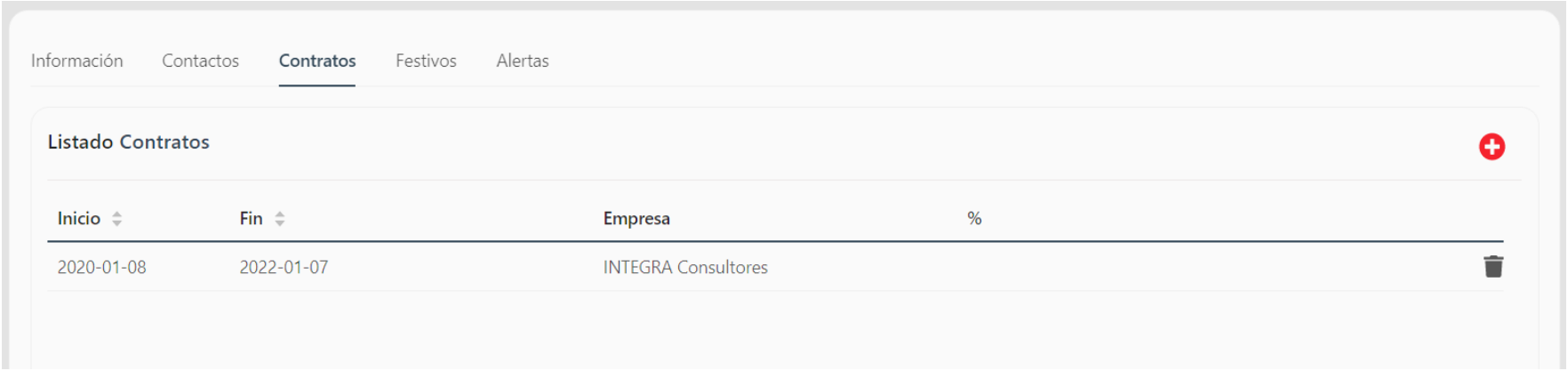


Figura 4.6: Listado de contratos de un empleado.

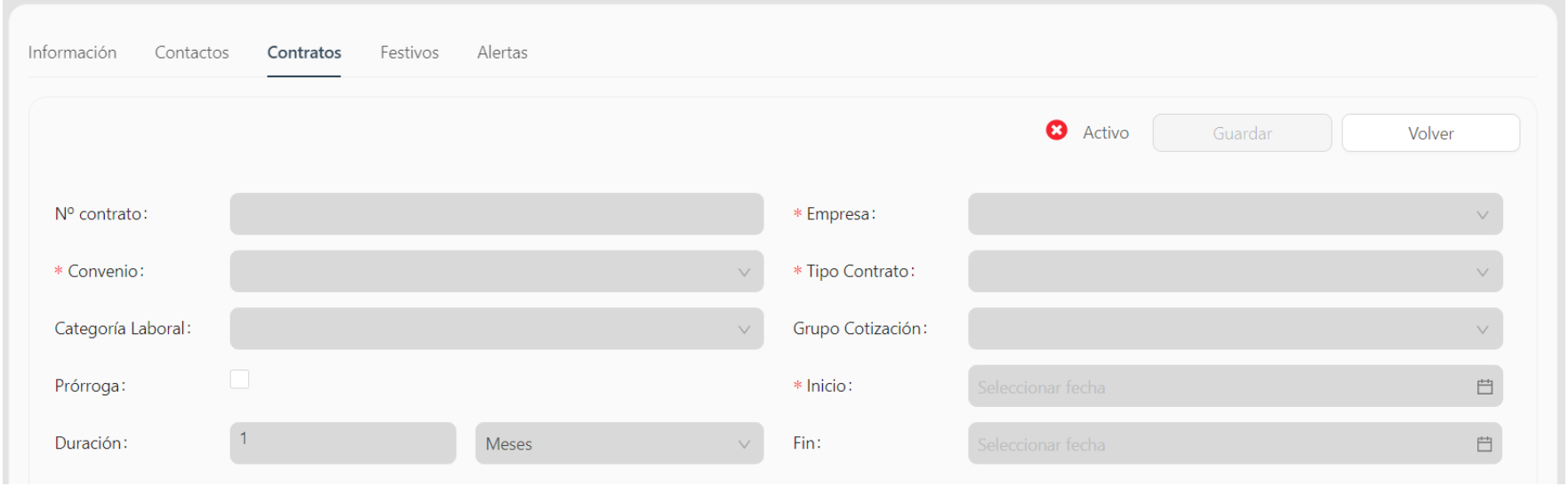


Figura 4.7: Creación de un nuevo contrato para un empleado.

Para las vistas destinadas a la creación de elementos se usan ficheros llamados formularios. Un ejemplo de uso se puede observar en la línea 3 del Listing 4.13, en la que se asocia la propiedad con el atributo "form" ya definido [22]. Como muestra el Listing 4.15, los formularios asignan validaciones a cada uno de los *inputs* de la vista, de forma que el botón de guardado no se activará hasta que no se cumplan las condiciones definidas en el fichero formulario. También se pueden definir mensajes para indicar qué datos se están introduciendo erróneamente.

```
export const EmpleadoForm = {
  idPropio: [null],
  tipoEmpleado: [null],
  dni: [null, [Validators.required]],
  nuss: [null],
  nombre: [null, [Validators.required]],
  apellidos: [null, [Validators.required]],
  fechaAntiguedad: [null],
  nivelFormativo: [null],
  fechaNacimiento: [null],
  direccion: [null],
};
```

Listing 4.15: Configuración del formulario para la modificación de información de un empleado.

Diálogo multipasos

Como último aspecto importante a implementar en cuanto a empleados, se encuentra el diálogo multipasos [39, 34]. El desarrollo de este diálogo sirvió para entender muy bien la forma que tienen los componentes de Angular de comunicarse entre ellos. Se creó un componente diálogo principal, que es al que se llama una vez se pulsa el botón de añadir un nuevo empleado. Este diálogo principal, a su vez, crea hijos, uno por cada paso. Las tareas del diálogo padre son:

- Inicializar y transmitir los modelos de datos a los hijos. Esto se consigue mediante *property binding* [9], que se ve en el Listing 4.16.
- Controlar qué paso se muestra en cada momento, para lo cual se crea una propiedad llamada "paso" y se usa la directiva "ngSwitch", mostrada también en el Listing 4.16. El valor del paso cambia mediante la gestión de eventos o *event binding* [5], pues los diálogos hijos pueden emitir un evento y actualizar el valor de la propiedad que contiene el padre [11]. Los hijos emiten estos eventos una vez se pulsa el botón mostrado en el Listing 4.17, que ejecuta el método del Listing 4.18. Únicamente se puede avanzar de paso si todos los datos están introducidos correctamente.

```
<div [ngSwitch]="paso">
    <div *ngSwitchCase="'pasoEmpleado'">
        <paso-empleado [empleado]="empleado" (pasoEvent)="pasoActual($event)
    "></paso-empleado>
    </div>
    ...
    <div *ngSwitchCase="'pasoResumen'">
        <paso-resumen [empleado]="empleado" [contrato]="contrato" [jornada]=
    "jornada" [puestoTrabajo]="puestoTrabajo"
            (pasoEvent)="pasoActual($event)"></paso-resumen>
    </div>
</div>
```

Figura 4.8: Paso contrato del diálogo para creación de un empleado.

Listing 4.16: Diálogo padre y comunicación con sus hijos.

```
<button nz-button nzType="primary" nzSize="default" type="button" (click)="
    cambiarPasoJornada()" [disabled]="!form.valid" nzBlock>
        {{ 'app.continuar' | translate }}
    <fa-icon [icon]="iconos.faArrowRight"></fa-icon>
</button>
```

Listing 4.17: Botón para acceder al paso de selección de jornada.

```
cambiarPasoJornada() {
        const validaciones = new Validaciones();
        this.errorFormularioTexto = validaciones.obtenerErrores(this.form);
        this.pasoEvent.emit('pasoJornada');
}
```

Listing 4.18: Evento que emite un valor al diálogo padre para pasar al paso de selección de jornada.

El resto de los diálogos hijos están construidos de forma similar a la interfaz de la Figura 4.5. El resultado final del diálogo y sus pasos se muestra en las Figuras 4.8 y 4.9. Hay que destacar que algunos datos del paso resumen son incorrectos por una configuración errónea del diálogo que se solucionó más tarde.

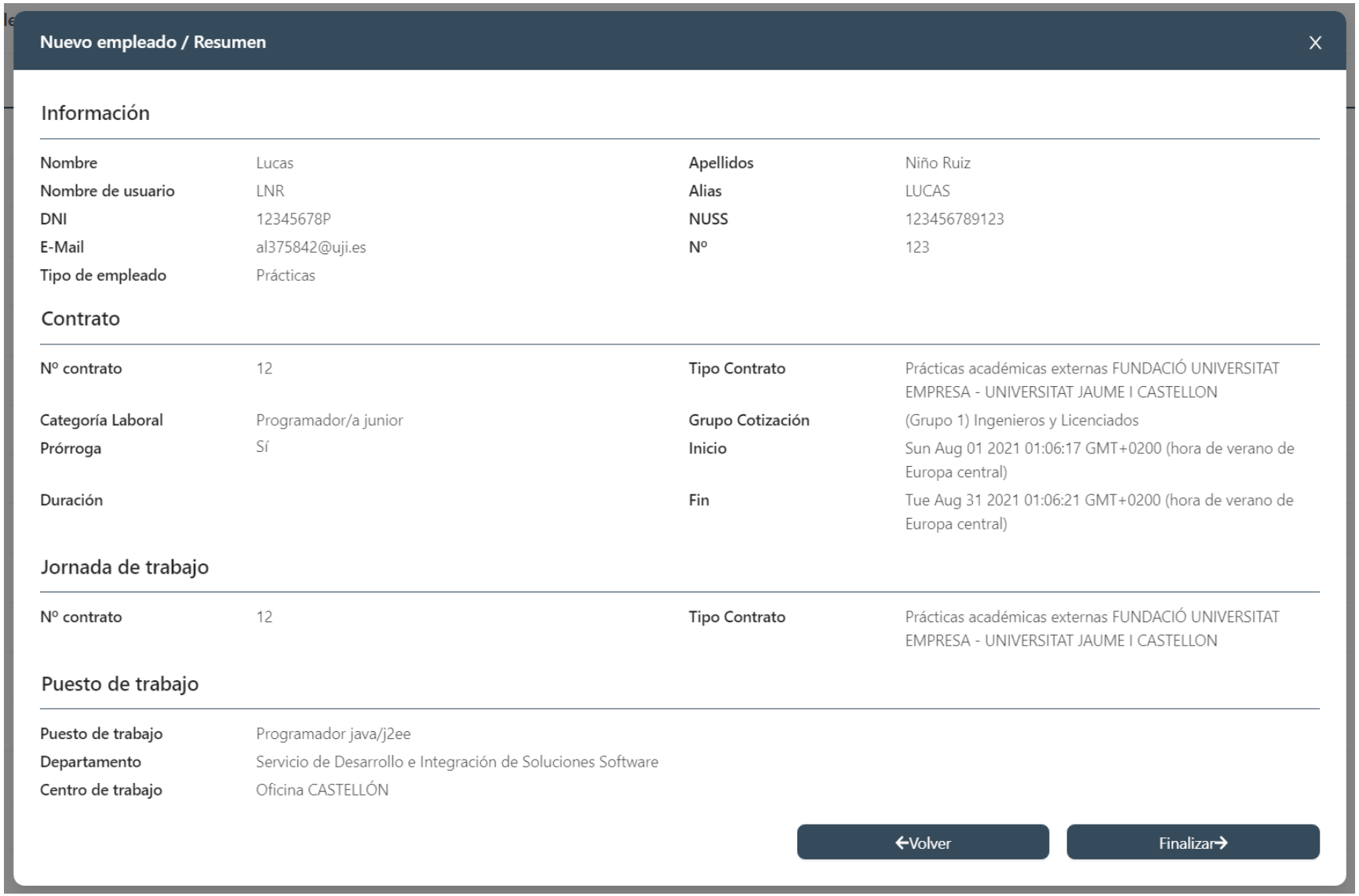


Figura 4.9: Paso resumen del diálogo para creación de un empleado.

Inputs select y tree-select

Como se puede observar en las Figuras 4.4, 4.7 y 4.8 algunos campos son de tipo *select*. Los valores a elegir mostrados por estos *inputs* son de un tipo de maestro llamado "nomenclaturas". Las nomenclaturas provienen del *back-end* y cada tipo contiene unos valores distintos. Desde el *front-end* se dispone de una lista constante en la que se asignan identificadores a cada tipo de nomenclatura (siguiendo la documentación del *back-end*) de la forma que muestra el Listing 4.19. Esta información se carga en el componente correspondiente mediante la llamada al método mostrado en el Listing 4.20. Haciendo esto se consigue obtener en cada uno de esos atributos una lista con varias nomenclaturas, cada una de ellas con su id propio y la descripción.

```
export const TIPOS_NOMENCLATURA = {
  'CONTACTO':1,
  'TIPO_PLANTILLA':2,
  'TIPO_CONTRATO':3,
  'CATEGORIA_LABORAL':4,
  ...
}
```

Listing 4.19: Lista de asignación de nomenclaturas e identificadores.

```
obtenerDatosRelacionados() {
    this.listaTiposDuracion = LISTA_TIPOS_DURACION;
    this.tiposContratos = this.nomenclaturasStoreService.obtenerPorTipo(
    TIPOS_NOMENCLATURA['TIPO_CONTRATO']);
    this.tiposGrupoCotizacion =
    this.nomenclaturasStoreService.obtenerPorTipo(TIPOS_NOMENCLATURA['
    GRUPO_COTIZACION']);
    this.tiposCategoriaLaboral =
    this.nomenclaturasStoreService.obtenerPorTipo(TIPOS_NOMENCLATURA['
    CATEGORIA_LABORAL'])
}
```

Listing 4.20: Carga de nomenclaturas en el componente paso contrato.

Lo que ve el usuario final por pantalla (p.e: "Programador/a junior" en la Figura 4.8) es la "descripción", mientras que lo que se actualiza en realidad es el "id". Para conseguir esta funcionalidad se ha codificado el *input* de manera que una vez se detecte un cambio en el valor se dispare un evento que ejecute un método para actualizar el valor de la descripción mostrado por pantalla (Listings 4.21 y 4.22).

```
<lib-input-select [(valor)]="contrato.categoriaLaboralId" [control]="
    form.controls.categoriaLaboral" (cambioValor)="
    seleccionarTipoCategoriaLaboral()" [elementos]="tiposCategoriaLaboral">
</lib-input-select>
```

Listing 4.21: Selección de una categoría laboral.

```
seleccionarTipoCategoriaLaboral() {
    let tipoCategoriaLaboralSeleccionada = this.tiposCategoriaLaboral.find(
    elem => elem.id == this.contrato.categoriaLaboralId);
    if (typeof tipoCategoriaLaboralSeleccionada !== "undefined") {
        this.contrato.categoriaLaboralDescripcion =
    tipoCategoriaLaboralSeleccionada.descripcion;
    }
}
```

Listing 4.22: Actualización de la descripción de la categoría laboral.

Además de los *inputs* de tipo *select*, en la aplicación están presentes los *inputs* de tipo *tree-select* [41]. En este proyecto se han usado los que proporciona la biblioteca NG-ZORRO. Los datos que representa este tipo de *input* es, como su nombre indica, un árbol compuesto de nodos, que en nuestra aplicación se corresponde con los maestros denominados "organigramas", presentes en el modelo del empleado (líneas 20-25 Figura 4.7).

El funcionamiento de este *input* comienza en la inicialización del diálogo padre, que tras ejecutar el código del Listing 4.23, envía el árbol obtenido al diálogo correspondiente mediante *property binding*, en este caso al paso puesto de trabajo. Es necesario que sea el diálogo padre el que obtiene los árboles ya que si relegamos esta funcionalidad al paso puesto de trabajo, los árboles se deberán cargar cada vez que se muestre dicho paso, lo que supone realizar llamadas a la API innecesarias y genera problemas visuales en el diálogo.

```
1 obtenerDatosRelacionados() {
2     this.organigramaDepartamentalService.getArbol().subscribe((response: any
      ) => {
3         this.arbolDepartamental = response.map((arbol) => new
      OrganigramaDepartamentalModel().deserialize(arbol));
4     })
5     ...
6 }
```

Listing 4.23: Obtención del árbol de departamentos por parte del diálogo padre.

Para que este *input* funcione correctamente con los datos de la aplicación se codifica un *wrapper* que añade la funcionalidad que se busca. En el modelo del "organigrama departamental" se deserializan los datos obtenidos y se asignan las propiedades del componente de NG-ZORRO a sus equivalentes en nuestra aplicación. Para poder hacer esto es también necesario utilizar la recursividad, ya que de lo contrario los datos de los nodos hijos no se cargarían correctamente. Este proceso está contenido en el Listing 4.24.

```
1  deserialize(input: any) {
2      Object.assign(this, input);
3      return this.prepare();
4  }
5
6  prepare(): this {
7      this.title = this.descripcion;
8      this.key = this.id;
9      this.deserializeHijos();
10     return this;
11 }
12
13 deserializeHijos() {
14     if (this.hijos.length > 0) {
15         this.hijos = this.hijos.map((hijo) => new
       OrganigramaDepartamentalModel().deserialize(hijo));
16         this.children = this.hijos;
17     } else {
18         this.isLeaf = true;
19     }
20 }
```

Listing 4.24: Inicialización del modelo del organigrama de departamentos.

La forma en la que el paso puesto de trabajo actualiza el “identificador” y la “descripción” del nodo es similar a lo mostrado en los Listings 4.21 y 4.22, con el añadido de que el componente propio “input-tree-select” emite un evento al diálogo paso puesto de trabajo al detectar algún cambio en el valor. Dicho método es el que muestra el Listing 4.25. El resultado final se puede observar en la Figura 4.10.

```
cambiar(id: any): void {
    this.cambioValor.emit(this.treedom.selectedNodes[0].origin);
}
```

Listing 4.25: Método que se ejecuta al seleccionar un valor en un input de tipo tree-select.

Figura 4.10: Ejemplo de tree-select en la selección del departamento en el paso puesto de trabajo.

Inicio de sesión y ficheros de idiomas

La última funcionalidad que se implementó fue la de inicio de sesión. En esta aplicación web se utiliza el protocolo de autenticación Oauth 2.0 [28], que funciona mediante *tokens*. De manera resumida, esto significa que una vez un usuario accede a la aplicación, le es asignado un *token*, sin el cual no podrá hacer uso de la aplicación. El fragmento de código del Listing 4.26 es el que lleva a cabo esta tarea una vez se introducen los datos en el formulario de la Figura 4.11.

```
submit() {
    this.error = '';
    const userName = this.form.controls.userName;
    const password = this.form.controls.password;
    if (this.form.invalid) {
      return;
    }
    this.sessionService.login({
      secreto: 'prueba',
      username: userName.value,
```

```
      password: password.value,
    }).subscribe((response: UserModel) => {
      if (!response.correcto) {
        this.error = response.mensaje;
        return;
      }
      this.tokenService.set(response);
      this.startupSrv.load().then(() => {
        let url = this.tokenService.referrer.url || '/';
        if (url.includes('/session')) {
          url = '/';
        }
        this.router.navigateByUrl(url);
      });
    });
}
```

Listing 4.26: Asignación de sesión y token y carga del servicio de puesta en marcha.

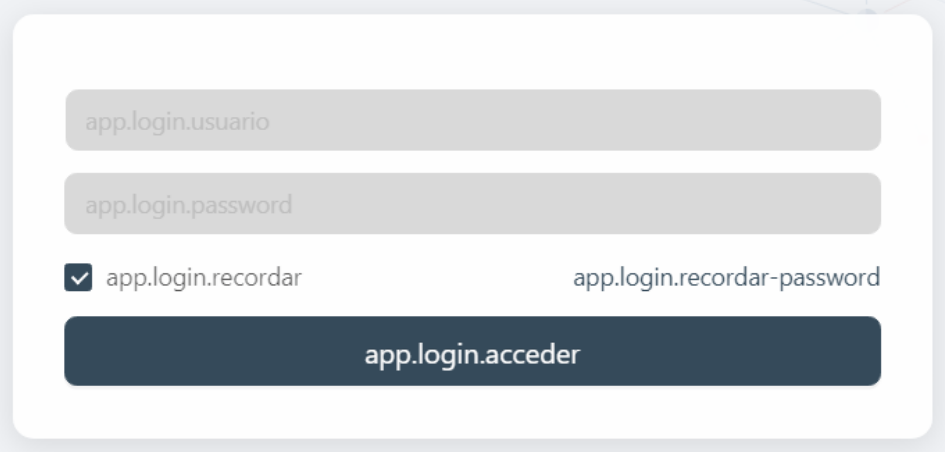


Figura 4.11: Formulario de inicio de sesión.

Como se puede observar, el texto del formulario para iniciar sesión no concuerda con lo que debería aparecer por pantalla. Los datos que muestra la aplicación están en español ya que esa es la información que se recibe de la API, pero todo el texto restante, como títulos o etiquetas, están configurados para que se muestren en un idioma u otro según el fichero de idioma que se haya cargado. En el caso de la Figura 4.11 no se ha cargado ninguno de estos ficheros y por eso muestra esos valores. La carga de dicho fichero no ha sido implementada por el alumno, pero sí se ha ido actualizando a lo largo del proyecto. El Listing 4.27 ilustra este proceso en el que se usa la tubería de traducción de Angular.

```
// Archivo JSON
{...  "app.login.usuario": "Usuario",
    "app.login.recordar": "Recordar", ...}

// Plantilla HTML
<input ... [placeholder]="'app.login.usuario' | translate"/>
```

Listing 4.27: Traducción de textos según fichero de idioma.

4.2. Verificación y validación

En esta sección se detallan las pruebas de verificación y validación utilizadas para comprobar el correcto funcionamiento de la aplicación y cuáles de los requisitos definidos inicialmente se han cumplido. Es una labor esencial para evitar entregar un producto defectivo o incompleto al cliente.

Comenzando por la verificación y siguiendo la dinámica habitual de Integra, en este proyecto no se han implementado tests para realizar comprobaciones. En su lugar, el supervisor ha ido examinando el funcionamiento del código según iban avanzando las semanas. De encontrar algún error, se comunica al alumno para que lo resuelva, examinando con él mismo algunas de las posibles soluciones.

Las pruebas realizadas han consistido en la introducción manual de valores en la aplicación, corroborando que el programa funcionaba como se espera. El alumno se ayudó de la herramienta Google Chrome Devtools. Se usó principalmente para:

- Comprobar y cambiar estilos en tiempo real, como se ve en la Figura 4.12. Esto ayuda a identificar errores y cambios a incluir en las plantillas.
- Consultar y modificar código fuente para lograr identificar fallos en las funcionalidades implementadas. Como se muestra en la Figura 4.13, también se pueden añadir *debuggers*, que son herramientas muy útiles para comprobar la información que contienen las variables en momentos específicos.
- Visualizar la información que devuelven los servicios del back-end para determinar si es correcta o se debe llevar a cabo algún cambio tanto en el *back-end* como en el *front-end*. Esto se ve en el apartado "Network", mostrado en la Figura 4.14.
- Consultar la consola (Figura 4.15) para posibles errores o advertencias.

También se debería mencionar que durante el desarrollo se utilizó la herramienta Swagger UI, que es un software de código abierto pensado para documentar el *back-end* y ayudar en la tarea de verificación en servicios web RESTful. Normalmente permite la interacción con la API sin necesidad de haber implementado la parte cliente, pero esta opción no se configuró para este proyecto, por lo que únicamente se usó para consultar el funcionamiento, atributos y parámetros, asegurando la correcta implementación del *front-end*.

Tras la verificación, en la que comprobamos que todo funciona correctamente, pasamos a la validación, en la que revisamos si los resultados obtenidos cumplen los objetivos y requisitos establecidos en la planificación inicial.

En cuanto a los objetivos iniciales, hay que recordar que al poco de empezar el proyecto se descartaron el módulo de calendario y el de control de presencia, pues se decidió que serían desarrollados durante los meses de agosto y septiembre. Al no entrar en las 300 horas de trabajo de prácticas, no se puede dar por cumplida esa parte.

Centrándonos en el módulo de empleados, que es sin lugar a dudas el más extenso e importante, sí se han cumplido prácticamente todos los requisitos propuestos. Los únicos que no se han com-

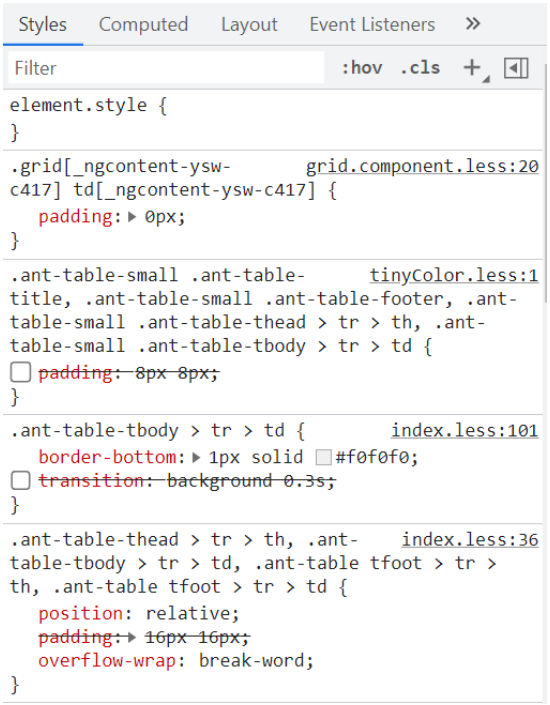


Figura 4.12: Google Chrome Devtools, sección de estilos.

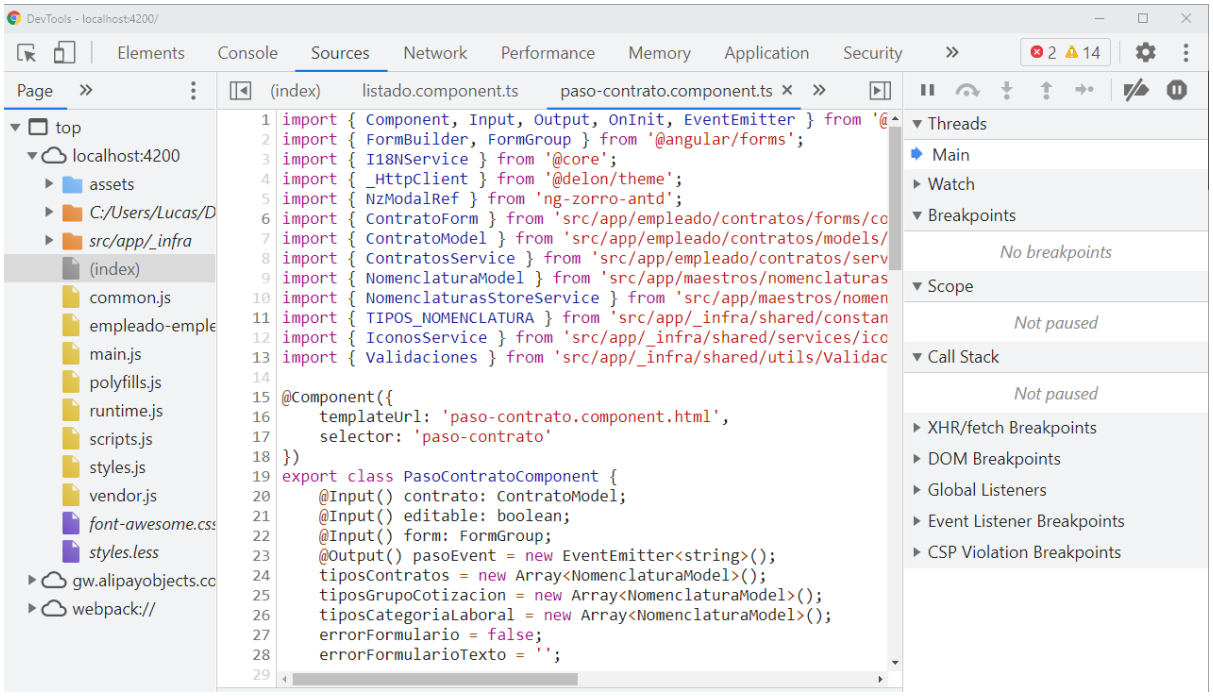


Figura 4.13: Google Chrome Devtools, sección de código fuente.

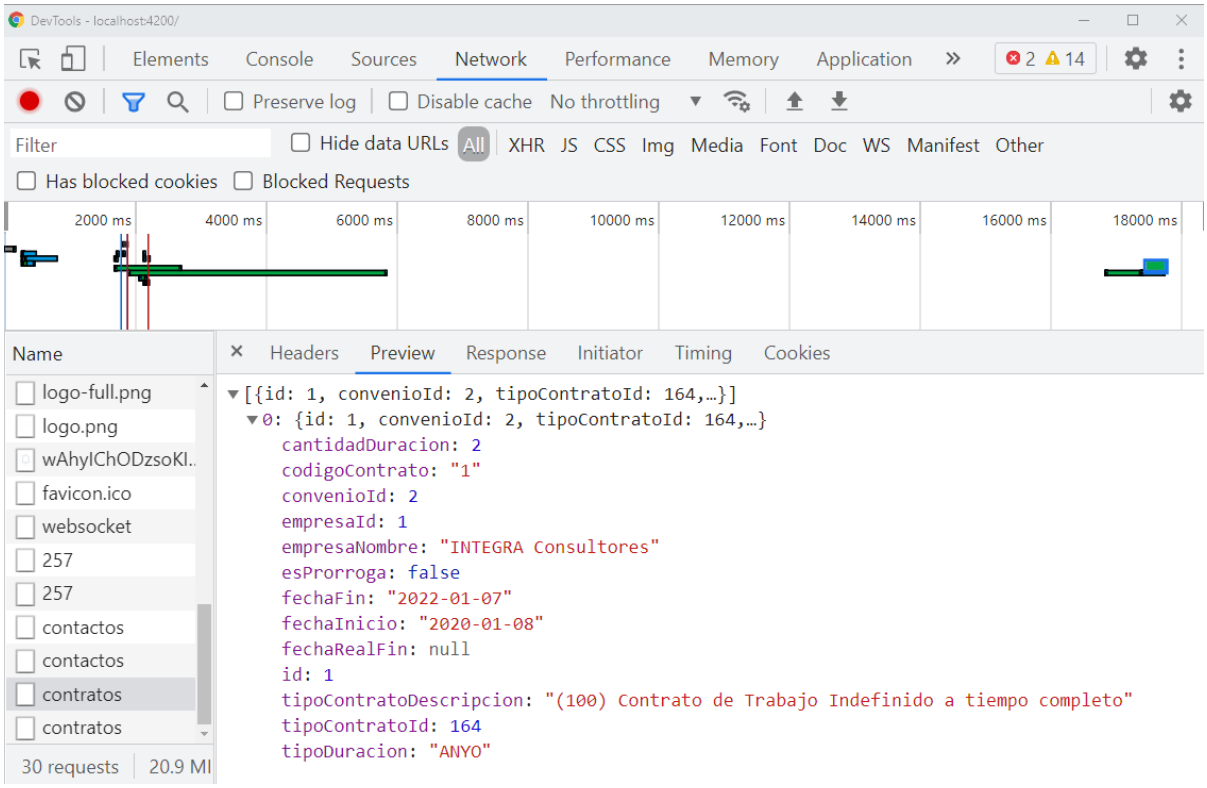


Figura 4.14: Google Chrome Devtools, sección de red.

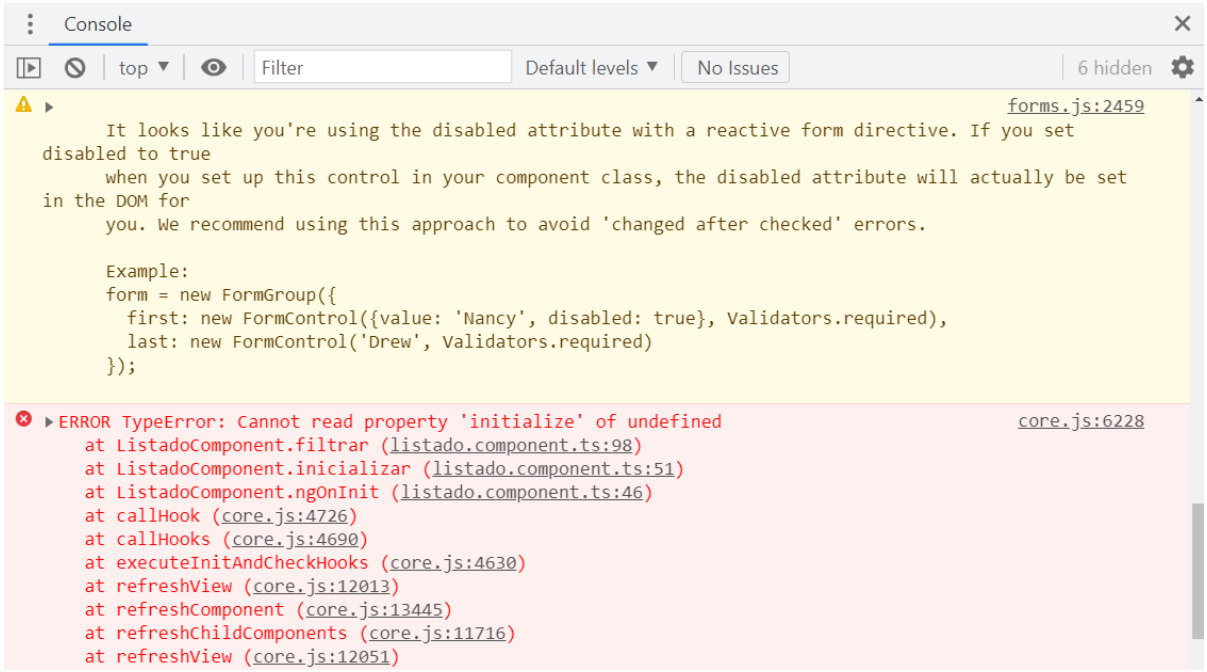


Figura 4.15: Google Chrome Devtools, sección de consola.

pletado (Cuadros 3.3 y 3.4) ha sido a causa de la falta de implementación del *back-end* por causas ya comentadas en la Sección 2.5.

Por último, en lo que a interfaces se refiere, sí se han cumplido todos los objetivos propuestos (lista completa en la Sección 3.3), pues la aplicación es coherente en todas las vistas desarrolladas y ha visto mejorada la navegación.

Capítulo 5

Conclusiones

En este último capítulo el alumno expresa sus pensamientos acerca de la estancia en prácticas. Para ello, se divide en ámbitos: formativo, indicando los conocimientos adquiridos; profesional, explicando tanto el ambiente de la empresa como el trato con el resto de empleados, y personal, donde se detallan los comportamientos del alumno y las consecuencias que tendrá esta experiencia.

5.1. Ámbito formativo

En cuanto a este ámbito, considero que la estancia en prácticas ha sido una experiencia sumamente enriquecedora a nivel de conocimientos técnicos, ya que he aprendido varias tecnologías y patrones que hasta ahora no había estudiado en el grado, especialmente Angular y los patrones que se usan en una empresa profesional. Si bien es cierto que en un principio suponía una gran dificultad programar en este framework, con el paso de las semanas me fui volviendo más ágil y eficiente.

Además de adquirir estos nuevos conocimientos, también he podido aplicar los aprendidos durante la carrera tanto en el desarrollo de la memoria como en la implementación, por ejemplo incluyendo recursividad en los organigramas.

Al haber resultado en un desarrollo accidentado (no se han cumplido todos los requisitos iniciales), principalmente por la falta de desarrollo de la parte *back-end*, también he comprobado de primera mano algunos de los errores que se cometen en proyectos de desarrollo software, por lo que sabré evitarlos y minimizar su impacto en futuras ocasiones.

Por último, en cuanto a la redacción de la memoria, estoy muy contento de haber aprendido a utilizar LaTeXpara elaborar un documento técnico completo [1, 16, 32, 43]. Sin duda es una habilidad que requeriré en mi futuro profesional.

5.2. Ámbito profesional

Considero que a nivel profesional ha sido un gran primer contacto con el mundo laboral real. El ambiente de la empresa es idóneo para un estudiante en prácticas, y tanto el supervisor como los compañeros no dudan en ayudar y ofrecer explicaciones de conceptos nuevos al estudiante. Desde aquí me gustaría agradecer la simpatía de todos ellos (Dani, Diego, Idoia, Adrián, Toni, Rubén), en especial del supervisor, Alex, que a pesar de tener mucha carga de trabajo y no poder avanzar el *back-end* me iba indicando qué aspectos podía mejorar o continuar desarrollando en el *front-end*. También destacar que al inicio del proyecto me ayudó a configurar el entorno de desarrollo en mi equipo personal para poder progresar de forma telemática.

Personalmente entiendo que se priorice un proyecto de un cliente con una fecha de entrega, y la solución ofrecida, que consiste en continuar el proyecto los meses de agosto y septiembre, me parece correcta para seguir formándome en la empresa.

5.3. Ámbito personal

En primer lugar, me gustaría destacar la comodidad que ha supuesto la elección de Integra como empresa en la que realizar mi estancia de prácticas. Además de seguir un desarrollo híbrido en el que dos o tres días por semana se trabajaba presencialmente y el resto de forma telemática, también han ofrecido conciliación en períodos de exámenes, dejando días libres al alumno para poder estudiar.

En el aspecto comunicativo mantuve buena relación con los miembros de la empresa y mi supervisor, ya que las reuniones, sobre todo las iniciales, ayudaron a establecer una vía comunicativa. Como ya he comentado, el hecho de que todos los trabajadores se mostraran amables ayudó mucho a que me sintiera integrado. También quiero agradecer la ayuda de mi tutor Jorge Badenas, ya que a pesar de haber realizado alguna entrega con retraso, siempre me ha indicado consejos y correcciones sensatas.

En cuanto a mi actitud frente al proyecto, estoy gratamente sorprendido. Para ofrecer un contexto, escogí Integra debido a que ofertaban un proyecto más relacionado con el *back-end*, que finalmente fue desechado y sustituido por el rediseño de Kalenda, cuya parte a realizar por el alumno se componía únicamente de *front-end*. Al tener menos experiencia en este aspecto y tener que desarrollar parte de una aplicación web de corte profesional y destinada a la comercialización, el inicio fue lento y no vi resultados durante los primeros días, pero me alegro de no haberme desanimado, pues al final he conseguido aprender mucho. Un punto clave para ello ha sido el de buscar mucha información por cuenta propia, viendo ejemplos, incluso con tecnologías distintas, y relacionándolo con lo que se necesitaba hacer en el proyecto.

Al inicio de la estancia me interesaba más el desarrollo *back-end*, opinión que mantengo, pero el conocimiento es conocimiento, y ahora sé mucho más sobre desarrollo *front-end*, lo que me será útil en futuras ocasiones. También me ha ayudado en mi elección de máster, ya que he descubierto que quiero especializar mi carrera profesional en el Big Data y Data Science, para lo cual también resultan útiles estas prácticas al ayudarme a entender de dónde proviene la información, cómo mostrarla y los procesos que hay detrás de las interfaces.

Por último, como ya he dicho varias veces, el desarrollo del proyecto continúa y tengo interés por ver hasta qué punto puedo desarrollar la aplicación con lo que ya he aprendido.

Bibliografía

[1] Andrew Roberts. Tables - Getting to grips with LaTeX. https://www.andy-roberts.net/writing/latex/tables. [Consulta: 25 de Julio de 2021].

[2] Angular. Angular coding style guide. https://angular.io/guide/styleguide. [Consulta: 25 de Julio de 2021].

[3] Angular. Angular.io. https://angular.io/. [Consulta: 25 de Julio de 2021].

[4] Angular. Common routing tasks. https://angular.io/guide/router. [Consulta: 25 de Julio de 2021].

[5] Angular. Event binding. https://angular.io/guide/event-binding. [Consulta: 25 de Julio de 2021].

[6] Angular. Getting started with angular. https://angular.io/start. [Consulta: 25 de Julio de 2021].

[7] Angular. Introduction to Angular concepts. https://angular.io/guide/architecture. [Consulta: 25 de Julio de 2021].

[8] Angular. Lifecycle hooks. https://angular.io/guide/lifecycle-hooks. [Consulta: 25 de Julio de 2021].

[9] Angular. Property binding. https://angular.io/guide/property-binding. [Consulta: 25 de Julio de 2021].

[10] Angular. Providing dependencies in modules. https://angular.io/guide/providers. [Consulta: 25 de Julio de 2021].

[11] Angular. Sharing data between child and parent directives and components. https://angular.io/guide/inputs-outputs. [Consulta: 25 de Julio de 2021].

[12] Atlassian. Atlassian Suite. https://www.atlassian.com/. [Consulta: 25 de Julio de 2021].

[13] Atlassian. How much does Atlassian Access cost? https://www.atlassian.com/software/access/pricing. [Consulta: 25 de Julio de 2021].

[14] Bitbucket. Bitbucket | The Git solution for professional teams. https://bitbucket.org/product. [Consulta: 25 de Julio de 2021].

[15] Carlos Blanco. Lista de comprobación de riesgos en proyectos software. https://ocw.unican.es/pluginfile.php/274/course/section/196/Lista%20de%20Riesgos%20en%20proyectos%20SW.pdf. [Consulta: 25 de Julio de 2021].

[16] Carsten Heinz. The Listings Package. http://users.ece.utexas.edu/~garg/dist/listings.pdf. [Consulta: 25 de Julio de 2021].

[17] Karla Cevallos. UML: Casos de Uso. https://ingsotfwarekarlacevallos.wordpress.com/2015/06/04/uml-casos-de-uso/. [Consulta: 25 de Julio de 2021].

[18] Maham S. Chappal. The 6 key principles of UI design. https://maze.co/collections/ux-ui-design/ui-design-principles/. [Consulta: 25 de Julio de 2021].

[19] Esther Cohen. What is WBS (Work Breakdown Structure) in Project Management? https://www.workamajig.com/blog/guide-to-work-breakdown-structures-wbs. [Consulta: 25 de Julio de 2021].

[20] Confluence. Confluence | Your Remote-Friendly Team Workspace. https://www.atlassian.com/software/confluence. [Consulta: 25 de Julio de 2021].

[21] Potix Corporation. Introduction of the ZK Angular Project. https://www.zkoss.org/zk-angular-demo/. [Consulta: 25 de Julio de 2021].

[22] Dayana Jabif. Angular Forms and Validations. https://angular-templates.io/tutorials/about/angular-forms-and-validations. [Consulta: 25 de Julio de 2021].

[23] Gerens Escuela de Postgrado. Gestión de riesgos: ¿Qué es? ¿Por qué emplearla? ¿Cómo emplearla? https://gerens.pe/blog/gestion-riesgo-que-por-que-como/. [Consulta: 25 de Julio de 2021].

[24] Erin Doherty. MVC Architecture in 5 minutes: a tutorial for beginners. https://www.educative.io/blog/mvc-tutorial. [Consulta: 25 de Julio de 2021].

[25] Georgina Guthrie. Everything you need to know about Gantt charts. https://backlog.com/blog/everything-need-know-gantt-charts/. [Consulta: 25 de Julio de 2021].

[26] Diego Alejandro Ríos Herrera. Dirección de proyectos de software desde la metodología PMBOK. http://repositorio.utp.edu.co/dspace/bitstream/handle/11059/6543/00538R586.pdf?sequence=1. [Consulta: 25 de Julio de 2021].

[27] Integra Consultores S.L. Integra Consultores - Expertos en soluciones tecnológicas a medida. https://integraconsultores.es/. [Consulta: 25 de Julio de 2021].

[28] Didin J. Angular 10: Oauth2 Login and Refresh Token. https://www.djamware.com/post/5f8f99673b3daf2033c40cac/angular-10-tutorial-oauth2-login-and-refresh-token. [Consulta: 25 de Julio de 2021].

[29] Jim Cooper. Angular Practice Exercises. https://jcoop.io/angular-practice-exercises/. [Consulta: 25 de Julio de 2021].

[30] Jira. Jira | Issue & Project Tracking Software. https://www.atlassian.com/software/jira. [Consulta: 25 de Julio de 2021].

[31] Judit Casacuberta. Perfect your ux design process – a guide to prototype design. https://www.toptal.com/designers/prototyping/guide-to-prototype-design. [Consulta: 25 de Julio de 2021].

[32] Latex. Documentation. https://www.overleaf.com/learn. [Consulta: 25 de Julio de 2021].

[33] Lucidchart. Lucidchart: Intelligent Diagramming. https://www.lucidchart.com/pages/. [Consulta: 25 de Julio de 2021].

[34] Maina Wycliffe. How to build a reusable Modal Overlay/Dialog Using Angular CDK. https://codinglatte.com/posts/angular/reusable-modal-overlay-using-angular-cdk-overlay/. [Consulta: 25 de Julio de 2021].

[35] Marcin Haręza. How to refactor Angular component and stay alive. https://softwareplant.com/how-to-refactor-angular-component/. [Consulta: 25 de Julio de 2021].

[36] Masooma Memon. 16 Important UX Design Principles for Newcomers. https://www.springboard.com/blog/design/ux-design-principles/. [Consulta: 25 de Julio de 2021].

[37] Miguel Rodríguez. Arquitectura de microservicios. En qué consiste. https://vanadis.es/arquitectura-de-microservicios-como-funciona-y-como-puede-ayudar-a-tu-negocio/. [Consulta: 25 de Julio de 2021].

[38] NG-ZORRO. Card. https://ng.ant.design/components/card/en. [Consulta: 25 de Julio de 2021].

[39] NG-ZORRO. Modal. https://ng.ant.design/components/modal/en. [Consulta: 25 de Julio de 2021].

[40] NG-ZORRO. Ng-zorro - Angular UI component library. https://ng.ant.design/docs/introduce/en. [Consulta: 25 de Julio de 2021].

[41] NG-ZORRO. Treeselect. https://ng.ant.design/components/tree-select/en. [Consulta: 25 de Julio de 2021].

[42] Node.js. Node for JavaScript. https://nodejs.org/en/. [Consulta: 25 de Julio de 2021].

[43] Olga Lapko. The makecell package. https://osl.ugr.es/CTAN/macros/latex/contrib/makecell/makecell.pdf. [Consulta: 25 de Julio de 2021].

[44] Piotr Nalepa. How to convert Angular component into reusable web Component? https://blog.piotrnalepa.pl/2020/02/02/how-to-convert-angular-component-into-reusable-web-component/. [Consulta: 25 de Julio de 2021].

[45] Microsoft Project. Microsoft Project | Project Management Software. https://www.microsoft.com/en-us/microsoft-365/project/project-management-software. [Consulta: 25 de Julio de 2021].

[46] M.J.M. Razi. MVC. https://www.researchgate.net/figure/Interaction-within-MVC-pattern-The-Model-component-correlates-with-all-the-data-related_fig2_328716094. [Consulta: 25 de Julio de 2021].

[47] RxJS. Reactive Extensions library for JavaScript. https://rxjs.dev/. [Consulta: 25 de Julio de 2021].

[48] Kevin Schuchard. What is the difference between Constructor and ngOnInit? https://blog.briebug.com/blog/what-is-the-difference-between-constructor-and-ngoninit. [Consulta: 25 de Julio de 2021].

[49] Sourcetree. Sourcetree | Free Git GUI. https://www.sourcetreeapp.com/. [Consulta: 25 de Julio de 2021].

[50] Director TIC. Software ágil frente al software en cascada. https://directortic.es/estrategia-it/software-agil-frente-al-software-en-cascada-2015092214539.htm. [Consulta: 25 de Julio de 2021].

[51] TIC Portal. Precio del software de recursos humanos. https://www.ticportal.es/temas/software-gestion-recursos-humanos/precio-software-recursos-humanos. [Consulta: 25 de Julio de 2021].

[52] TypeScript. TypeScript: Typed JavaScript at Any Scale. https://www.typescriptlang.org/. [Consulta: 25 de Julio de 2021].

[53] Swagger UI. Swagger UI | REST API Documentation Tool. https://swagger.io/tools/swagger-ui/. [Consulta: 25 de Julio de 2021].

[54] Visual Studio Code. VSCode | Code editing redefined. https://code.visualstudio.com/. [Consulta: 25 de Julio de 2021].