



TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN TECNOLOXÍAS DA INFORMACIÓN



Desenvolvemento dunha Plataforma Web para a xestión Integral de Configuración Técnicas no Entorno Universitario

Estudante: Pablo Rodríguez Pérez
Dirección: Pablo Alejandro Calviño Padín
Agustín Ricardo Rodríguez Fariña

A Coruña, setembro de 2024.

Resumo

Unha tarefa pouco apreciada e minimizada do traballo dun técnico consiste no control administrativo dos activos que xestiona. Este control é imprescindible para unha xestión colectiva dos recursos das empresas, de xeito que poidan expandirse máis aló da capacidade dunha persoa. Porén, non é sinxelo transformar un sistema manual e improvisado, cun longo historial e no que dependen moitos procesos vitais para as empresas, nun sistema automatizado e axeitado para o entorno empresarial multitudinario. O obxectivo deste proxecto é a creación dunha plataforma para a xestión de activos informáticos que lle permita ao persoal técnico axilizar o seu traballo de xestionar e controlar os activos existentes dentro da organización.

Abstract

Administrative control of assets is an often overlooked and underappreciated part of an administrator's job. This is a must for any company that wants to expand its management outside the capabilities of a single person. However, it's hard to transform an ad-hoc process, with a long history and on which many vital processes depend on, to an automatic system fit for big plural enterprise environments. The objective of this project is the creation of a platform for the management of computer assets that helps technical personnel in managing and controlling existing assets inside an organization.

Palabras chave:

Xestión

Sistemas

Java

Web

REST

Thymeleaf

Spring

Keywords:

Management

Systems

Java

Web

REST

Thymeleaf

Spring

Esta memoria, incluíndo todas as súas imaxes, atópase baixo a licencia Creative Commons Atribución-Compartir do mesmo xeito (CC BY-SA 4.0)¹

¹<https://creativecommons.org/licenses/by-sa/4.0/>

Índice Xeral

1	Introdución	1
1.1	Contexto e motivación	1
1.2	Obxectivos do traballo	1
1.3	Estrutura da memoria	1
1.4	Proxectos previos	2
2	Metodoloxía e planificación	3
2.1	Funcionamento de SCRUM	3
2.2	Planificación	4
2.3	Orzamento	5
3	Tecnoloxías empregadas	7
3.1	Servidor	9
3.2	Cliente	11
3.3	Autenticación e seguridade	11
3.4	Ferramentas	12
3.4.1	IntelliJ	12
3.4.2	Git	12
3.4.3	Overleaf/TeXstudio e L ^A T _E X	12
4	Deseño	13
4.1	Análise de requisitos	13
4.1.1	CU-01: Xestionar modelos	13
4.1.2	CU-02: Ver modelos	13
4.1.3	CU-03: Xestionar compoñentes	13
4.1.4	CU-04: Xestionar usuarios	13
4.2	Esquema do sistema	14
4.3	Obxectos	14

4.3.1	Modelos	14
4.3.2	Compoñente	15
4.4	Usuarios	15
4.4.1	Administradores	15
4.4.2	Técnicos	15
4.5	Busca	15
5	Interface gráfica	19
5.1	Funcionamento de <i>Spring MVC</i>	20
5.1.1	Modelo	20
5.1.2	Vista	20
5.1.3	Controlador	21
5.2	Caso de uso: Creación dun novo compoñente	22
5.3	Galería de imaxes	23
6	Arquitectura	31
6.1	Obxectos	31
6.1.1	Template	31
6.1.2	Component	32
6.1.3	TemplateField	32
6.1.4	Field	32
6.1.5	User	34
6.2	API REST do servidor	34
6.2.1	/templates	35
6.2.2	/components	36
6.2.3	/users	37
6.2.4	/search	38
6.3	Organización do código	38
6.3.1	Servidor	38
6.3.2	Cliente	39
6.4	Busca	39
7	Seguridade: Autenticación e autorización	43
7.1	Servidor	43
7.2	Cliente	45
8	Conclusións	51
8.1	Futuras melloras	51

A Material adicional	54
Relación de Acrónimos	55
Glosario	56
Bibliografía	57

Índice de Figuras

2.1	Diagrama de Gantt das fases do proxecto	5
4.1	Diagrama de casos de uso	14
4.2	Diagrama Esquema-Relación dos obxectos do sistema	16
4.3	Árbore de sintaxe de exemplo	18
5.1	Creando o compoñente «PC-02»	23
5.2	Compoñente «PC-02» creado	24
5.3	Creando o modelo «Rato»	24
5.4	Creando o compoñente «Rato»	25
5.5	Compoñente «Rato» creado	25
5.6	Menú de inicio, antes de crear o modelo «Rato»	26
5.7	Menú de inicio, co submenú aberto	26
5.8	Formulario de inicio de sesión	27
5.9	Páxina de xestión de compoñentes	27
5.10	Páxina de xestión de modelos	28
5.11	Páxina de xestión de usuarios	28
5.12	Modificar un usuario	29
5.13	Engadir un novo usuario	29
5.14	Buscando un modelo	30
5.15	Fragmento da páxina de axuda á busca	30
6.1	Diagrama de clases do servidor	35
6.2	Árbore de sintaxe de exemplo	41
7.1	Fluxo de inicio de sesión no servidor	44
7.2	Fluxo de acceso á API	44
7.3	Proceso completo de acceso á API	45

7.4	Acceso á aplicación por parte dun usuario anónimo	48
7.5	Uso típico no cliente	48
7.6	Pechar sesión	49
7.7	Uso típico no cliente	50

Introdución

1.1 Contexto e motivación

Unha parte importante do labor dun técnico de sistemas é o control e seguimento das distintas e misceláneas plataformas que poden atoparse no seno dunha organización típica.

Porén, a falta dun sistema de seguimento axeitado, o persoal técnico vese na obriga de usar outros métodos para gardar e controlar esta información, normalmente en sistemas *ad hoc* e non estruturados, o que dificulta a súa adaptación a circunstancias cambiantes, e cunha pobre documentación que limita e dificulta o paso deste sistema entre persoal. Todo isto representa un importante custo e risco a calquera organización,^[1] inda máis a unha orientada a ofrecer un servizo público. Esta plataforma nace co obxectivo de axudar nesta labor, e axilizar a xestión da infraestrutura dunha empresa.

1.2 Obxectivos do traballo

O obxectivo deste traballo é desenvolver unha plataforma que permita realizar o traballo previamente descrito dun xeito sinxelo, adaptable ás circunstancias de cada aplicación posible, evitando todo o posible restrinxir e obstaculizar o labor do persoal técnico. Por mor disto, a plataforma deberá ser sinxela de usar e fácil de acceder. A plataforma céntrase na xestión de activos, pero tamén lle dedica parte á xestión dos usuarios que a van usar. Todo isto desenvólvese máis amplamente no capítulo 4.

1.3 Estrutura da memoria

Esta memoria estrutúrase dun xeito relativamente temático, pasando de capítulos máis teóricos e organizativos, dedicados á organización do traballo (capítulo 2) e o deseño acadado nel (capítulos 4 e 5), seguindo con capítulos máis prácticos, nos cales explícanse os detalles

técnicos do proxecto (capítulo 6) e o funcionamento do sistema de seguridade (capítulo 7).

Remátase cunha conclusión do traballo e unha serie de melloras posibles para desenvolvementos posteriores (capítulo 8).

1.4 Proxectos previos

Existen moitas **ITAM** (**Informational Technologies Asset Management**) dispoñibles, a meirande parte aplicacións web, tanto de código aberto coma pechado. De código aberto consultáronse as plataformas **Snipe-IT**¹ e **PartKeepr**².

Snipe-IT presenta unha plataforma cunha serie de categorías, como poden ser «Consumibles», «Activos» ou «Licenzas». Cada unha destas categorías ten unha serie de campos, como pode ser custo nos activos ou cantidade restante nos consumibles. A pantalla principal da aplicación presenta unha serie de gráficas que resumen o estado dos dispositivos, cantos hai de cada categoría, etc. PartKeepr presenta unha interface máis típica dunha aplicación de escritorio. Ten unha estrutura de categorías en árbore modificables, dentro das cales pódense crear dous tipos de obxectos. As partes son compoñentes normais, con moitos campos (existencias, condición, prezo, distribuidoras, etc.), mentres que unha «metaparte» só inclúe o nome, a categoría, as unidades de medida das existencias e unha serie de filtros que agrupa as partes que cumpren esas condicións.

Os sistemas de ITAM existentes tamén integran diferentes funcionalidades non directamente relacionadas coa xestión de activos[2], como pode ser a xestión de incidencias e control remoto. Outras, como pode ser GoCodes³, inclúen códigos QR que permiten consultar a información referente a un compoñente. Estas ferramentas non están especialmente orientadas ao ámbito tecnolóxico, senón que tamén permiten xestionar compoñentes misceláneas, como poden ser recursos das oficinas ou ferramentas.⁴

¹<https://demo.snipeitapp.com/>

²<https://demo.partkeepr.org/>

³<https://gocodes.com/>

⁴ Como exemplo dunha plataforma que fai todo isto, véxase [Assetpanda](#)

Metodoloxía e planificación

Para realizar o traballo decidiuse seguir unha metodoloxía áxil e iterativa, ao ser un número reducido de desenvolvedores e clientes non se precisa unha estrutura xerárquica, e todas as partes poden manterse actualizadas do desenvolvemento do proxecto con facilidade. Esta metodoloxía consiste nun proceso iterativo, onde todos os interesados no desenvolvemento poden manterse ao día sobre os problemas e modificacións que van xurdindo.

2.1 Funcionamento de SCRUM

Como metodoloxía específica escolleuse SCRUM, pola súa familiaridade e popularidade. A metodoloxía SCRUM[3] baséase nunha serie de fases («*sprints*»), definidas no tempo, continuas e completas. Cada fase ten as seguintes partes:

Planificación Cada fase comezan coa súa planificación. É aquí onde se definiran os obxectivos da fase, e decidírase como realizar o traballo preciso

Reunión diaria Diariamente os desenvolvedores reuníranse para revisar a planificación das fases, co obxectivo de atopar e arranxar calquera desviación que poida desbaratar a fase. Deberán revisar o traballo realizado dende a última reunión e planificar o traballo a realizar para a seguinte reunión

Revisión O equipo deberá presentar unha revisión do traballo realizado na fase. Isto permitirá realizar cambios para futuras fases

Retrospectiva Tras a revisión, pero de xeito separado, o equipo deberá realizar unha revisión centrada no proceso da fase, de xeito que se identifiquen que bloqueos existiron, como se resolveron e o seu impacto

Para xestionar os obxectivos do proxecto, SCRUM prescribe a creación de tres listaxes:

1. Do produto, centrado na mellora do produto no seu conxunto
2. Da fase, un subconxunto da listaxe do produto no cal vaise traballar nesta fase. Representa un plan, que indica o obxectivo da fase e como acadalo
3. Do incremento, pasos concretos e funcionais que permiten acadar o obxectivo do produto. Pode haber varios incrementos por fase

Os resultados denomínanse «artefactos», e, dependendo da listaxe, poden denominarse obxectivo do produto («*Project Goal*»), da fase («*Sprint Goal*»); ou unha descrición de obxectivo cumprido («*Definition of Done*») no caso dos incrementos. Cada obxectivo representa o punto final de cadanseu ámbito.

Dentro do equipo de SCRUM defínense varios roles:

- Desenvolvedores, que son as persoas que acadan os obxectivos dos incrementos, seguindo as directrices das fases, produtos e incrementos
- Dono do produto, encargado de controlar os obxectivos do produto e asegurarse de que o equipo céntrase na creación de valor para o produto
- Xefe de SCRUM, que xestiona o proceso de SCRUM dende o punto de vista procesual e persoal, axudando ao resto de compoñentes do equipo segundo precisen.

Á hora de aplicar a metodoloxía anterior realizáronse certas modificacións para adaptalas ás particularidades dun desenvolvedor, o alumno, facendo os co-directores do proxecto de donos do produto.

As reunións realizáronse de dous xeitos. Por unha banda, e por mor das diferenzas de horario, a meirande parte das reunións foron asíncronas, mediante a plataforma de Teams. O resto foron reunións telemáticas, tamén mediante a plataforma de Teams, menos frecuentes pero de maior duración. Nestas últimas reunións realizouse unha demostración completa do funcionamento da aplicación.

2.2 Planificación

A organización das distintas tarefas (indicadas na figura 2.1) corresponde a dúas clasificacións: temática e modular.

A clasificación temática presenta tres elementos:

1. Modelos
2. Compoñentes
3. Usuarios

Cada un destes elementos precisa dunha tarefa de deseño, unha tarefa de implantación e integración da API REST, e da creación da interface web correspondente. A clasificación modular corresponde ás dúas partes do sistema: cliente e servidor. Dentro da tarefa de implantación realízase todo o traballo de programación, xestión das entidades na base de datos e conexión coa API REST, incluíndo serialización, deserialización e excepcións.

A parte do servidor inclúe o deseño e implantación da API REST, e a parte de cliente inclúe o deseño e creación da interface web.

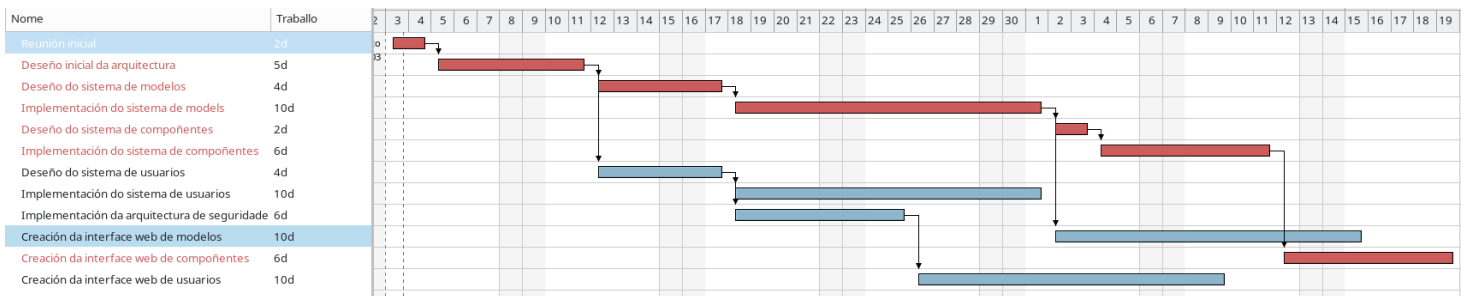


Figura 2.1: Diagrama de Gantt das fases do proxecto

Dentro do esquema de SPRINT definíronse 3 fases: modelo, compoñentes e usuarios. Dentro de cada fase incluíronse como incrementos as tarefas indicadas anteriormente.

2.3 Orzamento

A previsión do orzamento que pode precisar a realización dun proxecto depende de moitos factores únicos para cada empresa. Aquí preséntase unha posible aproximación, pensando no

ambiente dunha empresa pequena. Hai dúas clases de custos: periódicos e únicos. Os custos periódicos son aqueles que se repiten cada certo tempo. Dentro deste ámbito incluíríanse os salarios dos traballadores ou as licenzas, se as houber. Os únicos corresponden á adquisición do material que precisa cada traballador, e ocorren ao inicio do proxecto.

Inda que o máis probable é que o persoal xa dispoña do material que precisa doutros proxectos, inclúese no cálculo. Para os ordenadores engadiuse un orzamento de 1000 € por desenvolvedor, e o IDE representa un custo de 724,79€ por desenvolvedor (para un ano). Isto representa 1724,79€ por desenvolvedor, e 10348, 74€.

O tamaño do equipo calcúlase en seis persoas como un tamaño óptimo[4]. Considerárase un salario medio anual de 36000€[5][6][7]. A duración dun proxecto de *software* non é sinxelo de calcular, pero para este, considerando o tempo da realización deste traballo de fin de grao, escolléronse 3 meses para esta versión inicial. Isto daría un custo en salarios de $36000 \text{ €/persoa} \cdot \text{ano} \cdot 6 \text{ persoas} \cdot \frac{3 \text{ meses}}{12 \frac{\text{meses}}{\text{ano}}} = 54000\text{€}$ para os tres meses de desenvolvemento.

Isto representa un custo total de 64348, 74€

Tecnoloxías empregadas

As tecnoloxías empregadas foron:

- Para ambas partes:
 - Java
 - Spring Boot
 - Maven
 - OpenAPI
 - Docker
- Só na parte de servidor:
 - Hibernate/JPA
 - Tomcat
 - Jakarta RESTFull Web Services (JAX-RS)
 - Swagger Codegen, para xerar a definición [OpenAPI](#) a partir da interface JAX-RS
 - JUnit5
 - ANTLR4
- Só na parte de cliente:
 - Spring MVC
 - Spring Security
 - Swagger Codegen, para xerar un cliente REST a partir dunha definición [OpenAPI](#)
 - Thymeleaf
 - Bootstrap

- Ferramentas:
 - IntelliJ
 - Git
 - Overleaf/TeXstudio e L^AT_EX

A principal métrica á hora de buscar ferramentas foi a simplicidade na integración, buscando sempre a máxima automatización e evitando a repetición de código.

A linguaxe de programación Java foi escollida pola súa familiaridade, froito do seu uso continuado durante a carreira, especialmente durante as prácticas en empresa, onde tiven a oportunidade de empregar algunhas das tecnoloxías usadas neste proxecto: Spring e OpenAPI[8]. Spring é un conxunto de bibliotecas que expanden Java de moitos xeitos distintos, aforrando traballo e automatizando moitas tarefas, por exemplo mediante a inxección de dependencias, que crea instancias de clases automaticamente con só definilas como parámetros dunha función ou construtor. Neste proxecto úsanse os módulos Spring Boot, que xestiona o ciclo de vida dunha aplicación; Spring MVC, que permite a creación de aplicacións usando a arquitectura Modelo-Vista-Controlador; e Spring Security, que xestiona a autenticación e autorización da aplicación mediante unha API adaptable.

Ambos os módulos están empaquetados en contedores Docker. Para reducir o tamaño dos contedores, estruturáronse en dúas etapas. A primeira etapa, baseada nun contedor base con maven e o JDK 17, encárgase de compilar o código e xerar o JAR resultante usando a orde `mvn package spring-boot:repackage`. A segunda etapa, baseada nun contedor base só co JRE 17, encárgase de coller o JAR da primeira etapa, copialo ao cartafol correcto e definir o punto de entrada do programa.

Un `compose.yml` na raíz do proxecto encárgase de xuntar e conectar os contedores de cada módulo. Xunto cun contedor de PostgreSQL, defínense 3 servizos divididos en dúas redes: unha para o «*backend*», coa base de datos de PostgreSQL e o servizo *pleste-server*, e outra para o «*frontend*» cos servizos de *pleste-server* e *pleste-client*. Tamén se define un contedor para a base de datos de PostgreSQL. A creación dos dous contedores precisa facerse por separado, por mor de que *pleste-client* precisa de conectarse a *pleste-server* para descargar a definición de OpenAPI, pero o módulo de compilación usado por Docker non permite o acceso á rede, nin a definición de dependencias en tempo de compilación. Por isto, o primeiro que se fai é compilar e activar o contedor de *pleste-server*:

```
1 docker compose up -d pleste-server
```


Porén, inda que o servidor estea activo, precisa accederse dende o compilador en tempo de execución. Como o novo módulo de compilación de Docker non permite acceder a redes de Docker[9], hai que forzar o uso do módulo de compilación antigo:

```
1 DOCKER_BUILDKIT=0 docker compose build pleste-client
```

Outra solución sería modificar o `pom.xml` de *pleste-client* para que faga conexión co servidor mediante a rede externa de Docker. Decidiuse deste outro xeito para depender o mínimo do entorno anfitrión.

3.1 Servidor

Para a parte do servidor úsase a plataforma de *Jakarta RESTful Web Services* para implementar unha interface REST a partir dunha interface Java (referida neste caso ao conxunto de métodos públicos dunha clase) normal, aproveitándoa para xerar automaticamente o código para escoitar peticións HTTP, e chamar ás funcións axeitadas segundo o método e ruta da petición. Para automatizar a xeración de clientes, engadiuse tamén o xerador de OpenAPI, que permite xerar un documento JSON.

Para a API escolleuse HTTP con JSON pola súa simplicidade de funcionamento e integración con Jakarta, popularidade e transparencia. Isto permite, entre outras, comprobar o correcto funcionamento cun inspector de paquetes. HTTP é un protocolo cliente/servidor de nivel de aplicación que permite o intercambio de documentos mediante o envío de mensaxes[10]. Estas mensaxes conteñen cabeceiras en texto que definen os metadatos do corpo da mensaxe, se os houber. Para o cliente existen varios tipos de mensaxes, pero neste proxecto úsanse as mensaxes GET, POST, PUT e DELETE; cada mensaxe ten unha semántica distinta e unha serie de posibles respostas do servidor[11]. Inda que orixinalmente orientado ao envío de documentos de texto, é posible usar eses mesmos verbos para crear interfaces remotas, usando JSON para o envío de datos estruturados dentro dos corpos das mensaxes.[12] A documentación da semántica de cada mensaxe e a súa resposta forma a especificación OpenAPI.[13]

Para o almacenamento e xestión da información, úsase como base de datos relacional PostgreSQL. Para acceder á base de datos incluíuse o «*framework*» *Hibernate*, unha implementación da interface *JPA* que permite xerar peticións SQL a partir de etiquetas en obxectos Java, xestionando tamén as sesións, creación e actualización das táboas, mediante unha interface *CRUD*. Estas interfaces fan peticións sen precisar escribir código SQL, usando os nomes dos métodos nas interfaces para identificar que teñen que facer. Tampouco precisan unha clase que as implemente, abonda con usar Spring para inxectar as dependencias precisas. Para que sexa inda máis sinxelo, úsase unha clase `SQLDaoFactoryUtil` que, tamén mediante Spring, inicialízase a si mesma e da acceso a todas as interfaces.

PostgreSQL é unha base de datos relacional, de código aberto e orientada aos ámbitos empresariais e para pequenos usuarios.[14] Esta é unha base de datos axeitada para a carga de traballo pensada, e ademais permite unha escalabilidade que evitará que sexa un posible colo de botella no futuro. Pensouse en usar unha base de datos non relacional, especialmente á hora de modelar o polimorfismo nos campo dos compoñentes, pero rexeitouse por mor da súa complexidade e dificultade de integración co sistema xa montado en Hibernate.

Para xuntar *Hibernate*, Jakarta e correr o servidor web que esta última precisa, incluíuse o «framework» *Spring Boot* con Tomcat, o servidor predeterminado en Spring. Ademais, *Spring Boot* serve para realizar a inxección de dependencias.

Porén, o xeito de integrar Hibernate e os puntos REST nas mesmas clases de Java causa certas incompatibilidades e engadiu complexidade aos métodos responsables das peticións.

Os dous principais problemas relaciónanse por como Hibernate xestiona as claves foráneas. O primeiro é que Hibernate intenta reducir as peticións ás bases de datos e só inxectar os atributos cando se usen; ao serializar a JSON estas clases teñen os atributos como nulos e aparecen métodos «pantasma», que son os que inxectan os atributos pero que Jakarta non sabe serializar. O segundo é que, por omisión, serialízanse os atributos non primitivos como clases completas, o que é ineficiente se só se precisan algúns poucos atributos, ou mesmo só o ID.

Tamén engadíronse probas unitarias que comproban o funcionamento da parte servidor da aplicación, usando JUnit e integrando Spring para a inxección de dependencias. E realizáronse probas de integración de xeito manual, probando todas as funcionalidades da aplicación mediante un navegador web e CURL.

Para a parte de busca valoráronse 3 tecnoloxías: desenvolver un módulo de análise léxico e sintáctico propio, usar ANTLR[15] ou JavaCC[16]. Desenvolver un módulo propio descartouse pola súa complexidade e problemas que pode dar ante casos de uso pouco probables, pero que poderían amosar fallos en partes do código difíciles de probar pola gran cantidade de posibles entradas. Entre JavaCC e ANTLR, valorouse que ambas teñen integración con Maven. Porén, decidiuse usar ANTLR pola súa sintaxe para definir gramáticas, que é máis sinxela e seméllase máis á forma Backus-Naur, mentres que a sintaxe de JavaCC aseméllase máis á de Java.

Con ANTLR, tras xerar as clases axeitadas a partir da definición, existen dous xeitos de recorrer a árbore de sintaxe que crea[17]: un *Listener*, e un *Visitor*. Ambos recorren a árbore en profundidade, pero de xeito distinto. Un *Listener* presenta unha interface con dous métodos por cada regra sintáctica, un que executa ao entrar na regra, e outro ao saír dela. Pola outra banda, un *Visitor* só ten un método por regra, que executa ao entrar na regra e que sae da regra ao devolver o método. Isto fai que o programador poida decidir se entrar ou non nos nodos fillo, e de que xeito.

3.2 Cliente

No cliente usáronse os «*framework*» *Spring Boot* e *Spring MVC*. *Spring Boot* forma a base da aplicación, que despois *Spring MVC* aproveita para construír unha interface de usuario usando o patrón «*Model, View, Controller*». No caso desta aplicación, a «Vista» corresponde coas páxinas web, que están ligadas a unhas clases «Controladoras» que son as que reciben as peticións HTTP e as procesan, chamando ao modelo segundo precisen.

Para facer a parte de «Modelo» usouse a mesma ferramenta de xeración de OpenAPI que a usada no servidor. Esta ferramenta permite que, unha vez configurado o enderezo do que obter o ficheiro `openapi.json`, recrear a interface Java do servidor coma se for unha dependencia máis de Maven, xestionando o xerador toda a parte de abrir a conexión HTTP, acceder ás rutas axeitadas e serializar e deserializar os obxectos de JSON a Java. Grazas a isto, o uso da API faise de xeito practicamente nativo, tendo algunhas dificultades á hora de traballar coas excepcións, que o xerador colapsa nunha clase `ApiException` da que despois hai que extraer os códigos de fallo correspondentes.

Para facer a interface con HTML usouse o motor de modelado Thymeleaf, que intégrase ben con Spring Boot. Para o deseño das páxinas incluíuse o «*framework*» Bootstrap, principalmente a súa parte de CSS, para mellorar o deseño e adaptabilidade das páxinas.

3.3 Autenticación e seguridade

O sistema de seguridade baséase en dúas tecnoloxías distintas, unha no servidor e outra no cliente:

No caso do servidor, a seguridade proporciónaa Jakarta. Poderíase usar Spring Security tamén aquí, pero ao usar as etiquetas de Jakarta, Spring dá problemas á hora de interceptar as peticións.

No lado do cliente, a seguridade é traballo de Spring Security. Este módulo encárgase de todo, dende interceptar as peticións, redirixir á páxina de inicio de sesión e, se se aproveitan as clases incluídas no módulo, almacenar credenciais, autenticalas e autorízasas sen ter que escribir código propio.

Isto explícase máis detallado no capítulo 7.

3.4 Ferramentas

Neste traballo usáronse varias ferramentas non mencionadas anteriormente:

3.4.1 IntelliJ

Escolleuse o IDE de IntelliJ pola súa completitude e integración, non só coa linguaxe Java, se non tamén coas bibliotecas de Spring, Hibernate, JUnit, Jakarta e Thymeleaf.[\[18\]](#)

3.4.2 Git

Como sistema de control de versións escolleuse Git, e especialmente a plataforma de GitHub para gardar o proxecto. Git é practicamente o único sistema de control de versións, polo que a súa elección foi sinxela. Escolleuse a plataforma de GitHub por estar xa integrada co correo corporativo doutras materias previas.

3.4.3 Overleaf/TeXstudio e L^AT_EX

O deseño da memoria baseouse no modelo en L^AT_EX da Facultade. Usouse tanto a plataforma de Overleaf como, cando esta non funcionaba, o programa TeXstudio.

Capítulo 4

Deseño

4.1 Análise de requisitos

A primeira fase foi recoller requisitos e a delineación do dominio do problema a resolver. Para isto realizouse unha reunión inicial cos co-directores do traballo para entender as casuísticas das que nace a necesidade deste traballo. Despois de varias reunións cos co-directores, chegouse aos seguintes casos de uso (véxase diagrama 4.1):

4.1.1 CU-01: Xestionar modelos

Da xestión de modelos encárganse os actores Administradores, o que inclúe creación, borrado e modificación.

4.1.2 CU-02: Ver modelos

Ver os modelos inclúese separado de xestionalos, por que é a única acción que pode facer un técnico sobre os modelos; ademais, precísao para poder crear os compoñentes.

4.1.3 CU-03: Xestionar compoñentes

A xestión de compoñentes pode realizarse indistintamente polos actores Técnicos e Administradores, creación, borrado, visualización e modificación.

4.1.4 CU-04: Xestionar usuarios

A xestión de usuarios é exclusiva dos administradores.

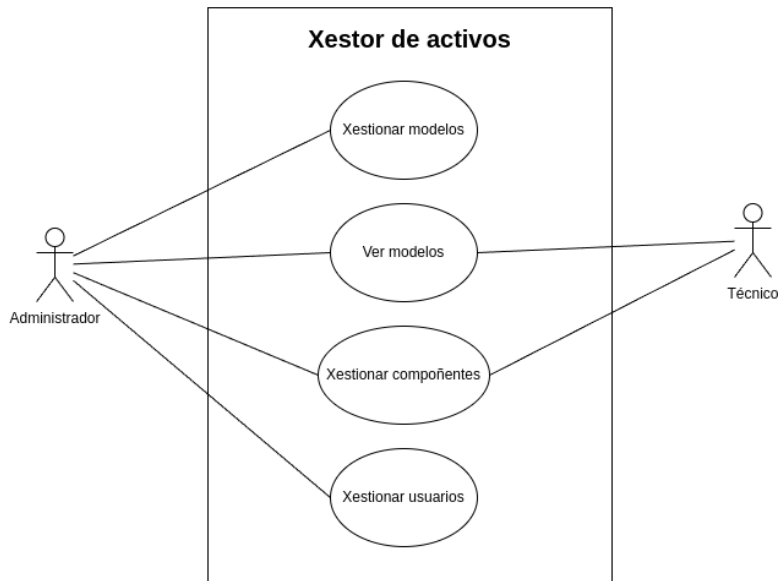


Figura 4.1: Diagrama de casos de uso

4.2 Esquema do sistema

A plataforma está formada por dous tipos de obxectos: modelos ([Template](#)) e compoñentes ([Component](#)). Tamén hai dous tipos de usuarios: administradores e técnicos.

4.3 Obxectos

A primeira versión da aplicación presentaba unha serie de obxectos concretos, tales coma servidores ou programas. Porén, tras falar cos codirectores e ver un chisco o uso real por parte de técnicos, escolleuse facer un sistema máis xenérico, que permita adaptarse ás necesidades de cada ambiente. O obxectivo é que cada técnico poida usar a aplicación do xeito que máis se axuste ao seu traballo. A figura 4.2 amosa a relación entre os distintos obxectos do sistema.

4.3.1 Modelos

Os modelos representan unha clase de compoñentes. Están formados por campos, e a súa creación correspóndelle aos administradores. Cada campo do modelo deberá indicar se se trata de texto libre, dunha ligazón a unha entidade ou dunha data. As ligazóns deberán apuntar a un compoñente.

Dentro da aplicación inclúense algúns modelos predefinidos:

- Ordenador
- Programa
- Licenza

Para evitar inconsistencias, non se poden modificar os campos dos modelos se xa teñen compoñentes asociados.

4.3.2 Compoñente

Os compoñentes corresponden a instancias individuais de modelos. Ao crear un compoñente deberanse encher tamén os campos correspondentes. As entidades, segundo se indique no seu modelo correspondente, poden ligarse entre si: ao visualizar unha entidade tamén se poden consultar as entidades que ligan a, e dende, esta.

4.4 Usuarios

4.4.1 Administradores

Os administradores son os usuarios encargados da creación dos modelos para que usen os técnicos. Tamén poderán encargarse da xestión e rexistro de usuarios.

4.4.2 Técnicos

Os técnicos son os usuarios que xestionan os compoñentes, e encárganse de manter actualizada a base de datos durante o transcurso do seu labor.

4.5 Busca

A busca realízase mediante unha serie de comparacións entre propiedades e valores, unidas mediante operacións lóxicas.

As propiedades serán da forma «*obxecto.atributo*». Os obxectos serán os indicados na sección anterior. Os atributos poderán ser calquera dos indicados na figura 4.2. Poderán compararse os contidos e atributos dos campos de compoñentes e modelos usando a mesma sintaxe de atributos separados por puntos. A propiedade que representa o valor dos campos é especial, e, dependendo da operación, só coincidirá con certos tipos de campos.

Para detalles máis específicos, véxase a sección 6.4.

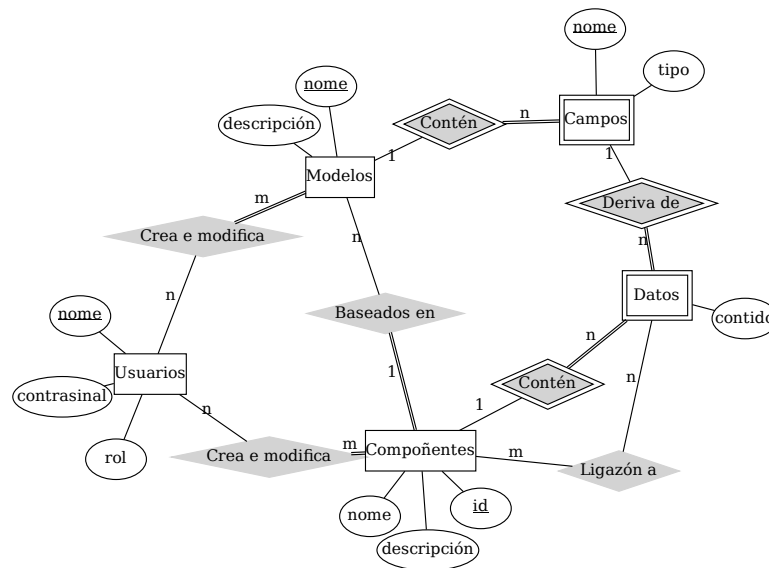


Figura 4.2: Diagrama Esquema-Relación dos obxectos do sistema

Existen tres grupos de operacións a realizar: sobre cadeas, sobre números e sobre conxuntos.

As operacións sobre cadeas realizan unha comparación usando expresións regulares. Para a comparación úsanse os operadores `~` e `!~` para a versión negada. A operación sobre cadeas pódese usar sobre todas as propiedades; se se aplica sobre o valor dos campos só coincidirá en campos de tipo texto ou data (neste caso sobre a representación textual da data). Esta operación pode usarse sobre todas as propiedades, e no caso do valor dos campos sobre os de tipo cadea e data (neste caso sobre a representación textual das mesmas).

As operacións sobre conxuntos son equivalentes a realizar unha serie de operacións sobre cadeas, unidas por disxuncións lóxicas. Úsanse coma operadores `«@»` ou `«!@»` (versión negada), e seguido unha listaxe de cadeas entre corchetes separadas por comas. Deste xeito, `template.name @ ["A", "B", "C"]` é equivalente a facer `template.name ~"A" | template.name ~"B" | template.name ~"C"`. Esta operación pode usarse sobre as mesmas propiedades ca as sobre cadeas, e tamén nos valores dos campos de tipo numérico e ligazóns, que deberán ser números completos e non expresións regulares.

As operacións sobre números aplícanse exclusivamente ao valor de campos, e só aos de tipo número, data e ligazón. As condicións posibles serán de maior (ou igual) que, menor (ou igual) que, igual ou non igual. Nas datas comparárase se o valor é posterior ou anterior á data indicada. Para as ligazóns só se poderán usar as condicións de igualdade ou non igualdade.

As operacións lóxicas son a conxunción (co operador «&») e a disxunción (co operador «|»). Ambas operacións teñen a mesma precedencia e asóciáanse de esquerda a dereita, de xeito que $a \mid b \ \& \ c$ é o mesmo que $(a \mid b) \ \& \ c$. Pódense usar parénteses para indicar a precedencia.

Isto da coma resultado a seguinte gramática independente do contexto en forma Backus–Naur:

```

<expresión> ::= <proba>
| '(' <expresión> ')'
| <expresión> '&' <expresión>
| <expresión> '|' <expresión>

<proba> ::= <propiedade> ('~' | '!~') <cadea>
| <propiedade> ('>=' | '<=' | '>' | '<' | '=' | '!=') ( <número> | <data> )
| <propiedade> ('@' | '!@') '[' <cadea> (',' <cadea>)* ']'

```

Os elementos terminais defínense usando expresións regulares. <número> defínese coma un número racional $([+-]?[0-9]+([0-9]+)?)$, <cadea> defínese coma calquera número de caracteres entre comiñas sen escapar $([^\backslash]" \cdot * [^\backslash]")$ e <data> defínese usando o formato ISO 8601¹ para data e hora sen incluír fuso horario $([0-9][0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]T[0-9][0-9]:[0-9][0-9]:[0-9][0-9])$.

Para a interpretación defínense dúas gramáticas: unha para a análise léxica e a outra para a análise sintáctica.

Na análise léxica defínese un alfabeto que inclúe todos os caracteres Unicode, limitado únicamente polos caracteres que acepta Java e PostgreSQL para cadeas. Sobre este alfabeto defínense certas linguaxes regulares: as xa indicadas previamente para definir números racionais, datas e cadeas; xunto coas que aceptan os operadores, comas, parénteses e corchetes.². A definición da gramática que acepta propiedades sería

$$l \in \{a, b, c, d, e, f, g, i, j, k, l, m, n, , o, p, q, r, s, t, u, v, w, x, y, z\} \quad (4.1)$$

$$p = l^*(.l^*)^* \quad (4.2)$$

¹ UNE-EN 28601:1995 na normativa española[19]

² Non se definen estas últimas pola súa simplicidade

Na análise sintáctica defínese un alfabeto baseado no resultado da análise léxica:

$$\begin{aligned} \Sigma = \{ & \text{propiedade, comparación, non comparación,} \\ & \text{maior que, menor que, maior ou igual que,} \\ & \text{menor ou igual que, igual que, non igual que,} \\ & \text{en conxunto, non no conxunto, paréntese aberto,} \\ & \text{paréntese pechado, corchete aberto, coma} \\ & \text{corchete pechado, cadea, número, data, conxunción, disxunción} \} \end{aligned} \quad (4.3)$$

Entre o paso léxico e sintáctico elimináronse os espazos en branco, saltos de liña ou calquera outros caracteres que non afectan á sintaxe, excepto dentro de cadeas.

Como nesta gramática existen producións con varias expresións non terminais, precísase pasar dunha gramática regular a unha independente do contexto, e deste xeito poder usar operacións de conxunción e disxunción dunha forma moito máis lexible. Se non, habería que usar algunha notación prefixo ou limitar moito que tipo de comparacións poderíanse facer.[20]

A modo de exemplo, vaise converter a cadea «(user.name @ ["root", "12"] & user.role !@ ["NORMAL"]) | template.name ~ "y" & template.name @ ["A"] & component.field.value > 2». Primeiro divídese mediante análise léxica, ignorando os espazos en branco:

$$\begin{aligned} & (\quad user.name \quad @ \quad [\quad "root" \quad , \quad "12" \quad] \quad \& \quad (4.4) \\ & \quad \text{paréntese aberto} \quad \text{propiedade} \quad \text{en conxunto} \quad \text{corchete aberto} \quad \text{cadea} \quad \text{coma} \quad \text{cadea} \quad \text{corchete pechado} \quad \text{conxunción} \\ user.role \quad !@ \quad [\quad "NORMAL" \quad] \quad) \quad | \quad template.name \quad (4.5) \\ & \quad \text{propiedade} \quad \text{non no conxunto} \quad \text{corchete aberto} \quad \text{cadea} \quad \text{corchete pechado} \quad \text{paréntese pechado} \quad \text{disxunción} \quad \text{propiedade} \\ & \quad \quad \quad \sim \quad "y" \quad \dots (4.6) \\ & \quad \quad \quad \text{comparación} \quad \text{cadea} \end{aligned}$$

A partir disto xerase a árbore de sintaxe (figura 4.3):

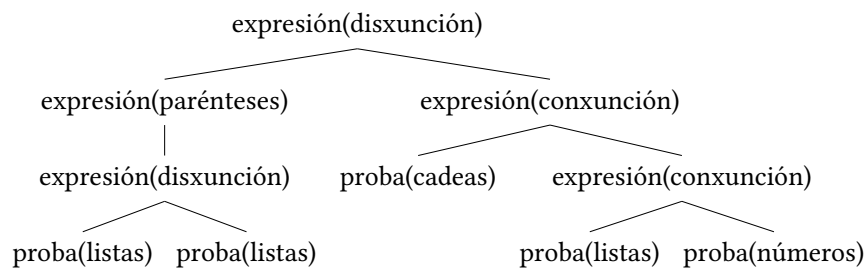


Figura 4.3: Árbore de sintaxe de exemplo

Interface gráfica

A interface gráfica está estruturada coma unha páxina web. Cada ruta representa unha funcionalidade diferente, primando a simplicidade e lixeireza das páxinas web fronte a unha interactividade excesiva, típica dunha SPA.

Aparte da páxina de inicio (ver figura 5.7), existen 12 páxinas principais: 4 por cada tipo de compoñente. Estas son:¹

- Para os modelos:
 - /newtemplate (figura 5.3)
 - /edittemplate?id=(id do modelo) (mesmo deseño ca /newtemplate, pero cos campos xa con texto)
 - /managetemplates (figura 5.10)
 - /displaytemplate?id=(id do modelo)
- Para os compoñentes:
 - /newcomponent?template=(id do modelo base) (figura 5.1)
 - /editcomponent?id=(id do compoñente) (mesmo deseño ca /newcomponent, pero cos campos xa con texto)
 - /managecomponents (figura 5.9)
 - /displaycomponent?id=(id do compoñente) (figura 5.5)
- Para os usuarios:
 - /newuser (figura 5.13)

¹ Non se intentou seguir ningunha arquitectura ao deseñar as rutas das páxinas, ao considerarse un detalle interno de implementación

- /edituser?id=(id do usuario) (mesmo deseño ca /newuser, pero cos campos xa con texto)
- /manageusers (figura 5.11)
- /displayuser?id=(id do usuario) (figura 5.12)

Pódese acceder a calquera desas rutas mediante os botóns que se atopan en todas as páxinas. Ademais, a cabeceira da páxina contén un menú despregable que permite saltar rapidamente ás seccións importantes.

Por suposto, antes de poder acceder á aplicación en si é preciso iniciar sesión. Pódese ver na figura 5.8 como é a pantalla de inicio de sesión.

5.1 Funcionamento de Spring MVC

No contexto de Spring MVC, os tres compoñentes implantase de xeito totalmente distinto:

5.1.1 Modelo

A parte de modelo é completamente interna a Spring. Este módulo encárgase de crear os obxectos asociados ao modelo, pasándoos aos compoñentes que precisen automaticamente.

5.1.2 Vista

Nesta parte é onde se aplica o sistema de modelado Thymeleaf. Para usalo, engádese o espazo de nomes de Thymeleaf á páxina que se vai usar. As páxinas HTML créanse no cartafol `resources/templates`. Thymeleaf permite incluír a información do modelo dentro da páxina, accedendo á información dentro duns atributos especiais, ademais de permitir usar construcións típicas de linguaxes de programación, como poden ser as condicións e os bucles. Amósase coma exemplo a páxina de inicio da aplicación:

```
1 <!DOCTYPE HTML>
2 <html xmlns:th="http://www.thymeleaf.org" lang="gl">
```

Thymeleaf precisa de incluír este espazo de nomes para poder usar os seus atributos propios

```

1 <head>
2 <title>pleste: PLataforma de xEstión de configuraciónS
    ↳ Técnicas</title>
3 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"
    ↳ />
4 <th:block th:insert=~{includes}" />
5 </head>
6 <body>
7 <header th:insert=~{header}"></header>

```

Thymeleaf permite incluír outras páxinas, e deste xeito reutilizar elementos comúns usando o atributo `insert`.

```

1 <main class="container mt-5">
2 <a class="btn btn-info" role="button" href="/newtemplate">Crear un
    ↳ novo modelo</a>
3 <h2>Modelos xa creados</h2>
4 <ul>
5 <li th:each="template: ${templates}"><a
    ↳ th:href="/template?id=${template.id}" th:text="|Modelo
    ↳ ${template.name} (nº ${template.id})|"></a></li>
6 </ul>
7 </main>
8 </body>
9 </html>

```

Nesta sección inclúese un bucle. O bucle defínese usando o atributo `each`, que inclúe o nome do elemento iterado e, usando a sintaxe para acceder ao modelo (`${atributo}`), a lista. Con isto, Thymeleaf repetirá ese elemento HTML tantas veces coma elementos conteña a lista, asignando o valor á variable `template`. Para que a etiqueta `<a>` poida usar os valores da variable, precísanse engadir os campos propios de Thymeleaf, co prefixo `th:`, para que poida substituír o contido entre corchetes polo valor da propiedade dentro do obxecto.

5.1.3 Controlador

O controlador é onde está toda a lóxica da interface, e é o encargado de engadir os datos precisos ao Modelo. Dentro de Spring, para crear un controlador só se precisa crear unha clase, etiquetala con `@Controller`, e engadir os métodos que procesarán as peticións. Cada método deberá etiquetarse co tipo de petición e ruta que procesa, usando a etiqueta `@GetMapping("/")` (para procesar GET) ou `@PostMapping("/")` (para procesar POST). No cliente, todas as clases no cartafol `view` son controladores. Son estas clases as que realizan as chamadas á API REST, procesan os resultados e lanzan as excepcións.

Despois, a definición de cada método indicará de que xeito vai procesar a petición. Devolver unha cadea de texto indica que a cadea conterá o nome do ficheiro HTML a usar na vista, pero tamén se pode devolver unha clase `Model` ou `ModelView`, que permiten devolver directamente o modelo, pasando o ficheiro no construtor da clase, ou incluso calquera outra resposta, devolvendo un `ResponseEntity` creado co código, corpo e cabeceiras que se precisen. Dependendo dos parámetros da función, Spring pasaralle os obxectos que se precisen, dende o modelo (pasando unha clase `Model` ou `ModelView`) ou a sesión (clase `HttpSession`) ate os parámetros GET e POST da petición. Neste último caso, precisa que os parámetros sexan de tipo `String`, e que estean etiquetados con `@RequestParam`.

De xeito especial, pódese poñer un parámetro de tipo `Map<String, String>`, que conterá todos os parámetros da petición.

Para amosar o funcionamento da interface, vaise amosar exemplo de uso típico:

5.2 Caso de uso: Creación dun novo compoñente

Para este caso de uso de exemplo, vaise crear un novo compoñente rato, vinculado a un ordenador.

O primeiro paso é crear o ordenador. Para isto aprovéitase o modelo xa creado «Ordenador». A partir deste modelo créase o compoñente «PC-02», tal coma se pode ver nas figuras 5.1 e 5.2.

Despois vaise engadir un rato e vinculalo ao ordenador. Para isto créase o modelo «Rato» (figura 5.3), a partir do cal créase un compoñente rato (figura 5.4), e engádese nun campo unha ligazón ao compoñente «PC-02», como se pode ver no campo PC da figura 5.5.

5.3 Galería de imaxes

root

Menú ▾

Pegar sesión

Modelo a usar:

1 - Ordenador

↕

Cambiar

Nome do compoñente:

PC-02

Descrición do compoñente:

PC da esquina

Campos

Os campos obrigatorios atópanse en negra

Para os campos de ligazóns, engadir o número do compoñente

Nome	Tipo	Valor
Modelo	Texto libre	HP 2020
Data de compra	Data	02 / 05 / 2024, 00 : 00
Sistema operativo	Ligazón	

Crear compoñente

Figura 5.1: Creando o compoñente «PC-02»

root

Menú ▾

Pegar sesión

Compoñente PC-02 (nº 1)

PC da esquina

Baseado en: [Ordenador \(nº 1\)](#)

Modificar

Eliminar

Campos

Os campos obrigatorios atópanse en negra

Nome	Tipo	Valor
Modelo	Texto libre	HP 2020
Data de compra	Data	2024-05-02T02:02
Sistema operativo	Ligazón	

Figura 5.2: Compoñente «PC-02» creado

root

Menú ▾

Pegar sesión

Nome:

Rato

Descrición:

Rato unido a un ordenador

Crear novo campo

☐

PC

Ligazón ▾

Borrar campo

☒

Data da compra

Data ▾

Borrar campo

Crear modelo

Figura 5.3: Creando o modelo «Rato»

root

Menú

Pegar sesión

Modelo a usar:

4 - Rato

Cambiar

Nome do compoñente:

Rato PC-2

Descrición do compoñente:

Rato unido ao PC-2

Campos

Os campos obrigatorios atópanse en negra

Para os campos de ligazóns, engadir o número do compoñente

Nome	Tipo	Valor
PC	Ligazón	1
Data de compra	Data	01 / 01 / 2024 , 00:00

Crear compoñente

Figura 5.4: Creando o compoñente «Rato»

root

Menú

Pegar sesión

Compoñente Rato PC-2 (nº 2)

Rato unido ao PC-2

Baseado en: [Rato \(nº 4\)](#)

Modificar

Eliminar

Campos

Os campos obrigatorios atópanse en negra

Nome	Tipo	Valor
Data de compra	Data	2024-01-01T00:00
PC	Ligazón	PC-02

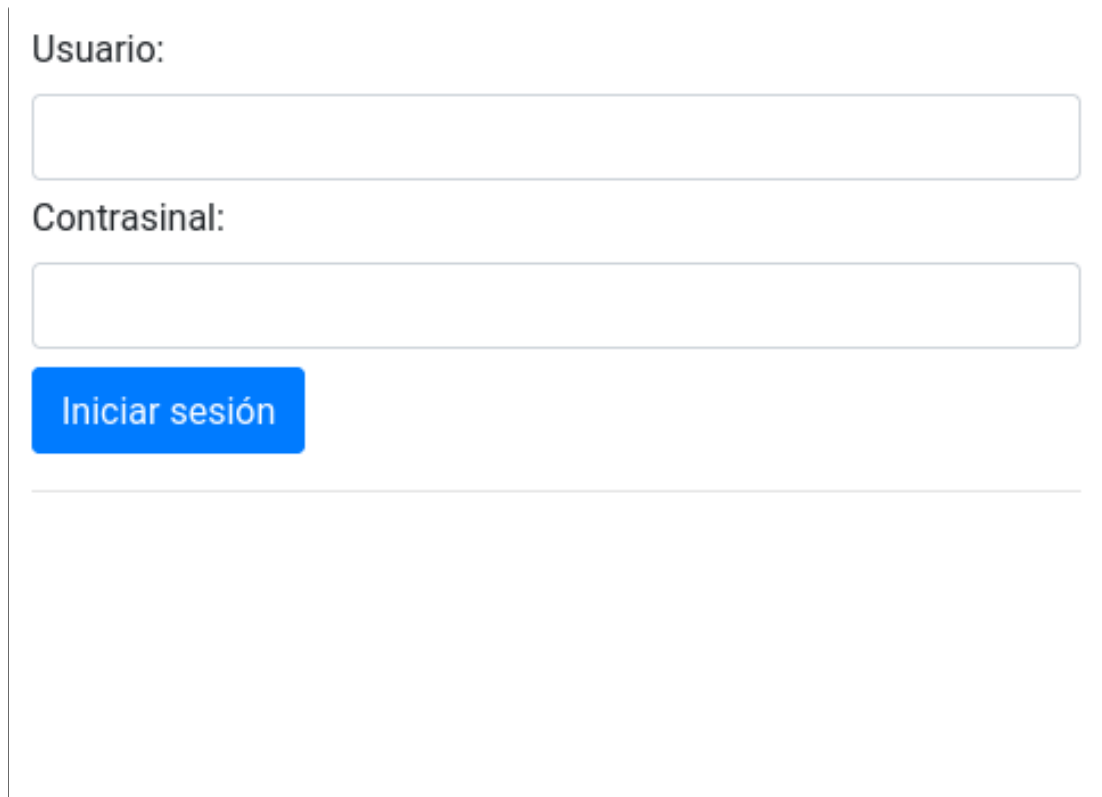
Figura 5.5: Compoñente «Rato» creado



Figura 5.6: Menú de inicio, antes de crear o modelo «Rato»



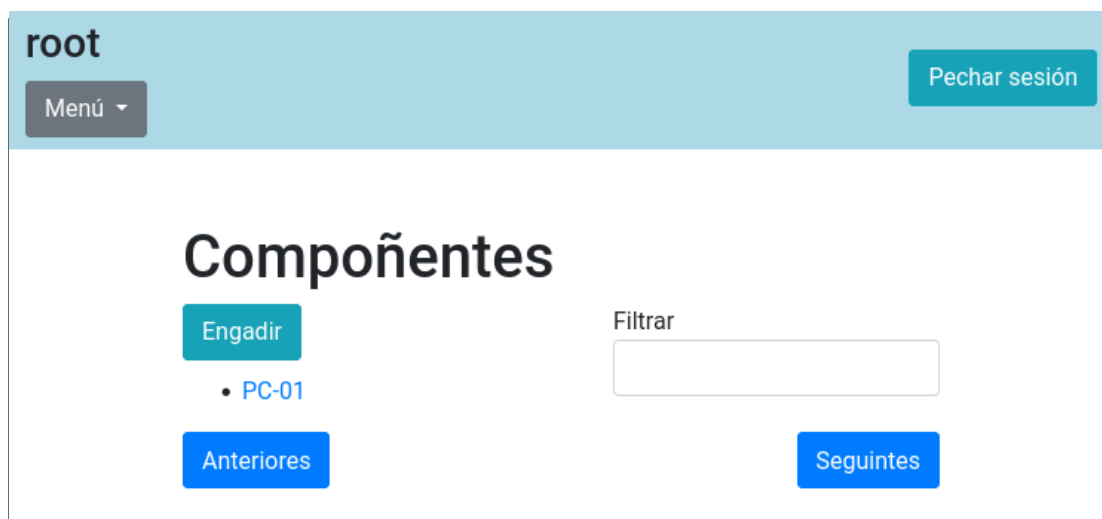
Figura 5.7: Menú de inicio, co submenú aberto



Formulario de inicio de sesión con los siguientes elementos:

- Etiqueta: **Usuario:**
- Caja de entrada de texto para el usuario.
- Etiqueta: **Contraseña:**
- Caja de entrada de texto para la contraseña.
- Botón azul: **Iniciar sesión**
- Línea horizontal de separación.

Figura 5.8: Formulario de inicio de sesión



Interfaz de gestión de componentes con los siguientes elementos:

- Barra superior azul con el texto **root** a la izquierda y el botón **Pegar sesión** a la derecha.
- Botón gris: **Menú ▾**
- Título principal: **Componentes**
- Botón verde: **Engadir**
- Lista de componentes:
 - PC-01
- Botón azul: **Anteriores**
- Botón azul: **Seguintes**
- Etiqueta: **Filtrar**
- Caja de entrada de texto para filtrar.

Figura 5.9: Páxina de xestión de componentes

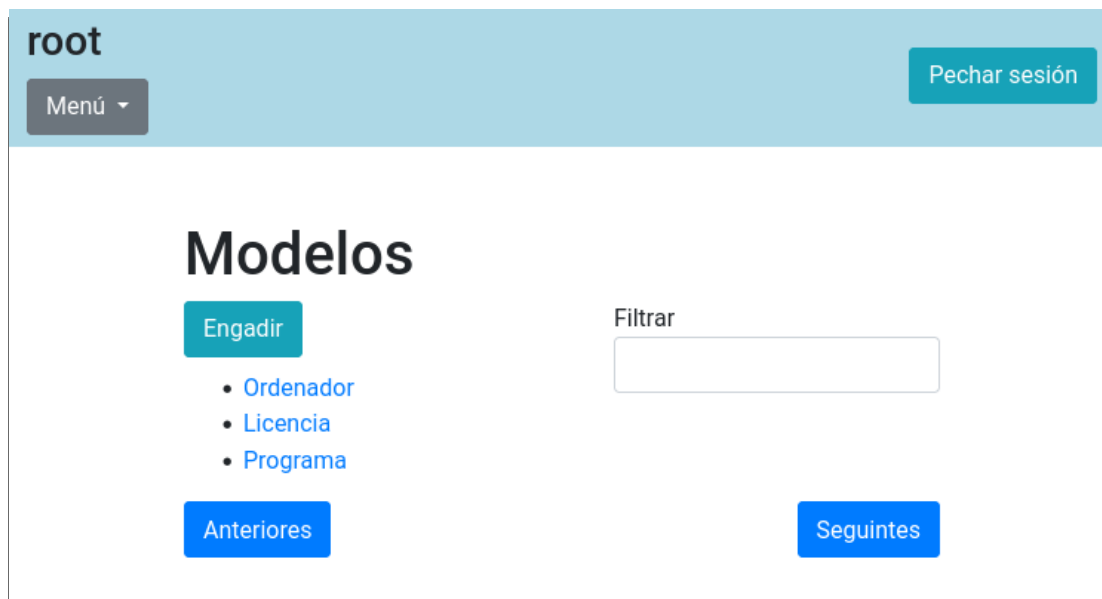


Figura 5.10: Páxina de xestión de modelos

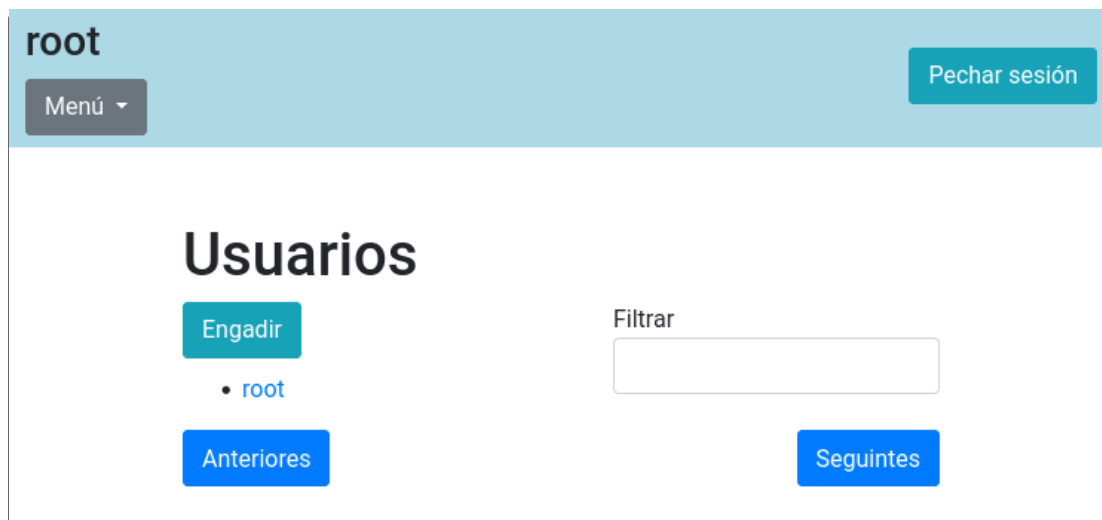


Figura 5.11: Páxina de xestión de usuarios

The image shows a web interface for modifying a user. At the top, there is a header bar with the text 'root' on the left and a 'Pegar sesión' button on the right. Below the header, there is a 'Menú' button. The main form area contains the following fields:

- Nome:** A text input field containing the value 'root'.
- Correo:** A text input field containing the value 'root@localhost'.
- Contrasinal:** An empty text input field.
- Rol:** A dropdown menu with 'Administrador' selected.

At the bottom of the form, there is a 'Gardar' button.

Figura 5.12: Modificar un usuario

The image shows a web interface for creating a new user. At the top, there is a header bar with the text 'root' on the left and a 'Pegar sesión' button on the right. Below the header, there is a 'Menú' button. The main form area contains the following fields:

- Nome:** A text input field containing the value 'novo'.
- Correo:** A text input field containing the value 'u@b'.
- Contrasinal:** A text input field filled with dots, indicating a password.
- Rol:** A dropdown menu with 'Técnico' selected.

At the bottom of the form, there is a 'Crear usuario' button.

Figura 5.13: Engadir un novo usuario

The screenshot shows a web interface with a light blue header. On the left, the word 'root' is displayed above a 'Menú' button with a downward arrow. On the right, there is a 'Pegar sesión' button. The main content area has a title 'Introducir termos de busca'. Below it is a search bar containing the text 'template.name ~ \".*\"'. To the right of the search bar is a 'Modo' dropdown menu and a blue 'Buscar' button. Below the search bar, the word 'Axuda' is written in blue. To the right of 'Axuda' is a checkbox labeled 'Borrar caché'. Below this, there is a list of results: 'Ordenador', 'Licencia', and 'Programa', each preceded by a blue dot. At the bottom of the search results, there are two blue buttons: 'Anteriores' on the left and 'Seguientes' on the right.

Figura 5.14: Buscando un modelo

The screenshot shows a web interface with a light blue header. On the left, the word 'root' is displayed above a 'Menú' button with a downward arrow. On the right, there is a 'Pegar sesión' button. The main content area contains a paragraph explaining search syntax: 'Unha petición de busca componse dunha comprobación, ou varias unidas polos operadores | e &, opcionalmente agrupados con parénteses. Por exemplo, a petición `template.name ~ "y" & template.description @ ["A", "B"]` busca calquera modelo cuxo nome sexa unha «y» e a descrición sexa «A» ou «B».' Below this paragraph is a section titled 'Operandos:' in bold. Under this section is a list of operators: '• ~ e !~: Aplican unha expresión regular á propiedade indicada', '• <, >, <=, >=, = e !=: Aplican unha comparación numérica', and '• @ e !@: Aplican ~ e !~, respectivamente, á listaxe de cadeas entre corchetes e separadas por comas'.

Figura 5.15: Fragmento da páxina de axuda á busca

Capítulo 6

Arquitectura

A plataforma está formada por dous compoñentes: unha parte servidor, implementada en Java e que realiza as labores de conexión coa base de datos, e que presenta unha [API REST](#) ao exterior; e unha parte cliente, un servidor Web en Java que conecta á parte servidor mediante peticións á [API REST](#). Decidiuse engadir unha API REST, e non só integrar todo no mesmo sistema, para permitir futuras extensións e aplicacións alternativas, que permita un acceso máis cómodo e adaptado ao persoal técnico. O esquema de deseño fundamental foi «*API first*», pensando en presentar un mesmo método de interacción mediante peticións HTTP con JSON, no canto de integrar o cliente dentro da aplicación e que chamar directamente aos métodos dende Java.

6.1 Obxectos

Existen cinco obxectos principais. A figura [6.1](#) representa as relacións entre as clases:

6.1.1 Template

Representa un modelo a partir do cal sacar [Component](#).

Campos:

id Valor numérico que representa o ID do modelo

name Cadea que contén o nome do modelo

description Cadea que contén unha pequena descrición do modelo

fields Lista de clases [TemplateField](#), que son os campos do modelo

6.1.2 Component

Representa un compoñente creado a partir dun [Template](#).

Campos:

id Valor numérico que representa o ID do compoñente

name Cadea que contén o nome do compoñente

description Cadea que conten unha pequena descrición do compoñente

template Clave foránea/punteiro que apunta ao [Template](#) do que se creou este Component

fields Lista de clases [Field](#), que son os campos do compoñente

6.1.3 TemplateField

Representa os campos do modelo, que son a base para crear os campos do [Component](#).

Campos:

id Valor numérico que representa o ID do campo

name Cadea que contén o nome do campo. Non se poden repetir no mesmo [Template](#)

mandatory Valor binario que indica se o campo é obrigatorio

type Indica o tipo do campo con un enumerado.

Os tipos de campos son:

TEXT Permite gardar texto plano e sen formato

LINK Representa unha ligazón a outro compoñente

DATETIME Representa unha data

6.1.4 Field

Representa un campo dun [Component](#).

Campos:

id Valor numérico que representa o ID do campo

type Tipo do campo, copiado do [TemplateField](#) e usado para diferenciar durante a serialización

content Atributo «virtual», que serve para indicarlle a OpenAPI que clases poden conterse dentro do campo.

templateField Ligazón/clave foránea no modelo relacional ao campo do [Template](#) no que se basea este campo o [Component](#)

O obxecto `Field` é unha clase abstracta, e parametrizada cunha clase xenérica que serve para devolver distintos tipos de obxectos. Para poder usar herdanza na base de datos hai tres tipos de estratexias:[21]

- **MappedSuperclass**, que crea unha táboa por cada subclase (pero non para a superclase) replicando todas as columnas herdadas e propias. O problema que ten isto é que non se pode referenciar á superclase dende outras táboas da base de datos, o que neste caso impediría que `Component` tivera unha foránea a `Field`, e tería que ter foráneas a todos os campos individuais.
- Reunir nunha soa táboa todas as subclases. Isto permite engadir foráneas, pero o número de subclases e o rendemento están limitados polo número de columnas que acepte a base de datos.
- Unha táboa por subclase, de xeito que para acceder a unha entidade precísase facer unha operación de JOIN entre a táboa da subclase e da superclase. Non ocupa tanto espazo coma a seguinte opción, pero é máis lenta ca a anterior opción por mor de precisar acceder a varias táboas.
- Unha táboa completa por subclase, tendo cada táboa da subclase os seus atributos máis os da superclase. Non se precisan facer JOIN, pero ocupa máis espazo ao ter que duplicar todos os atributos.

Escolleuse a 2ª opción por ser a máis sinxela, eficiente e ser a opción predeterminada en Jakarta. Ademais, non se considera que o número de tipos de campos vaia medrar dabondo coma para ser un problema. E, en calquera caso, poderíase migrar a calquera das outras opcións só cambiando unha anotación.

As subclases existentes son:

TextField Representa un campo de texto. Substitúe a clase xenérica por unha clase `String`

DatetimeField Representa un campo de texto. Substitúe a clase xenérica por unha clase `JSONDatetime`

LinkField Representa unha ligazón a outro compoñente. Substitúe a clase xenérica por unha clave foránea a unha clase `Component`

NumberField Representa un valor numérico decimal. Substitúe a clase xenérica por `BigDecimal`

Inda que cada subclase ten unha propiedade `content` distinta para evitar colisións na base de datos, o contido é o mesmo.

6.1.5 User

Representa un usuario no sistema.

Campos:

id Valor numérico que representa o ID do usuario

username Cadea que contén o alcume do usuario

email Cadea que contén o enderezo electrónico do usuario

password Cadea que contén o contrasinal do usuario, cifrado con `BCrypt`

role Enumerado que contén o tipo de rol do usuario

Os posibles valores para o campo `role` son `ADMINISTRATOR` e `NORMAL_USER`. Para máis información, consulte a sección 4.4.

Para os usuarios tamén existe a clase `Token`, que, aparte do usuario ao que referencia e un ID, garda unha `Testemuña` nunha cadea.

6.2 API REST do servidor

Todas as rutas que inclúen algún id ou nome, devolven un «*HTTP 404 Not Found*» se non existe algún dos identificadores ou nomes. A API acepta tanto JSON coma XML, pero o cliente só usa JSON.

A meirande parte das peticións precisan dunha testemuña válida pasa na cabeceira «*X-API-KEY*». Para máis información sobre permisos consultar a sección 3.3

Todas as rutas descritas son relativas á ruta `$SERVIDOR/api/v0/$SUBSECCIÓN`.

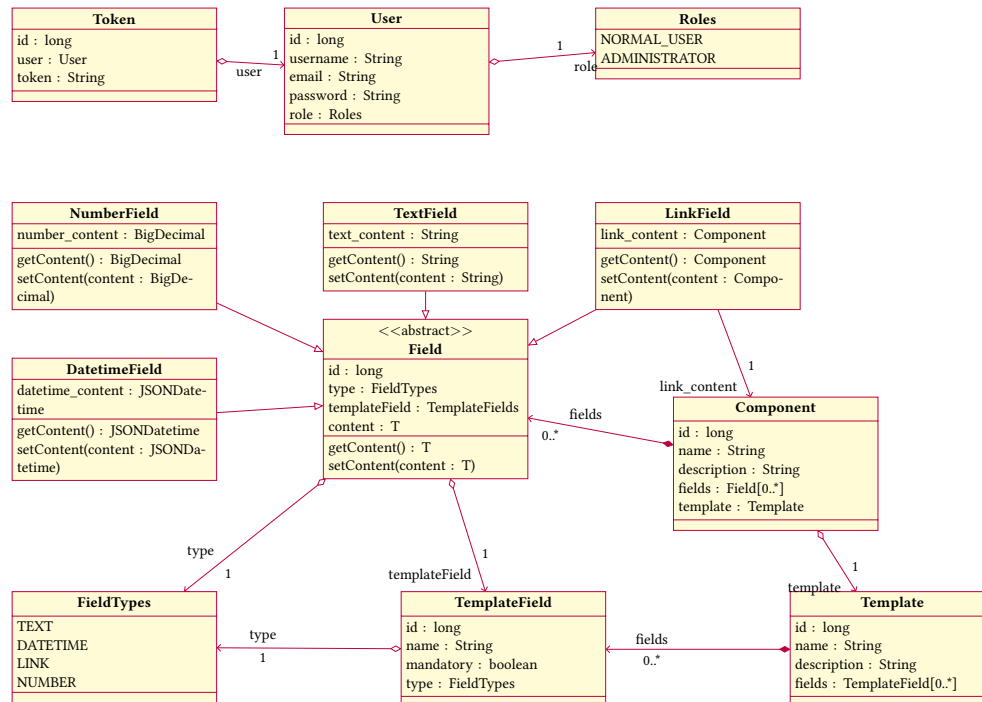


Figura 6.1: Diagrama de clases do servidor

6.2.1 /templates

Nesta sección xestiónanse os modelos os seus campos.

/

GET Devolve unha listaxe de todos os modelos dispoñibles. Permite devolver só un rango engadindo os parámetros `skip` e `count`

POST Crea un novo modelo incluído no corpo da petición. Devolve un «*HTTP 409 CONFLICT*» se xa existe outro modelo co mesmo nome

/find?name=(nome)

GET Devolve unha listaxe de todos os modelos dispoñibles cuxo nome coincide con «nome»

/(id)

GET Devolve o modelo con id `id`

POST Actualiza o modelo co contido do corpo da petición

DELETE Elimina o modelo. Devolve un *HTTP 409 Bad Request* se inda quedan *Components* vinculados a este modelo

`/(id)/components`

GET Devolve unha listaxe de todos os compoñentes derivados deste modelo

`/(id)/fields`

GET Devolve unha listaxe de todos os campos deste modelo

POST Engade un novo campo ao compoñente. Falla se o modelo está en uso

`/(id)/fields/(id)`

GET Devolve o campo indicado

POST Modifica o campo indicado, co contido do corpo da petición

DELETE Elimina o campo indicado.

Tanto a modificación coma o borrado fallan se o modelo está en uso.

6.2.2 `/components`

Nesta sección xestiónanse os compoñentes e os seus campos.

`/`

GET Devolve unha listaxe de todos os compoñentes dispoñibles. Permite devolver só un rango engadindo os parámetros `skip` e `count`

POST Crea un novo compoñente incluído no corpo da petición

`/find?name=(nome)`

GET Devolve unha listaxe de todos os compoñentes dispoñibles cuxo nome coincide con «nome»

`/(id)`

GET Devolve o compoñente con id `id`

POST Actualiza o compoñente co contido do corpo da petición

DELETE Elimina o compoñente

`/(id)/fields`

GET Devolve unha listaxe de todos os campos deste modelo

POST Engade un novo campo ao compoñente. Falla se o modelo está en uso

`/(id)/fields/(nome)`

GET Devolve o campo chamado «nome»

PUT Modifica o valor campo indicado, co contido do corpo da petición. A columna modificada na base de datos depende do tipo do dato

DELETE Elimina o campo indicado

Se se intenta cambiar un campo obrigatorio a unha cadea baleira ou nula, ou eliminalo, devólvese unha mensaxe «*HTTP 400 Bad Request*».

6.2.3 /users

Nesta sección xestiónanse os usuarios. Os usuarios só os poden modificar e crear usuarios administradores.

`/`

GET Devolve unha listaxe de todos os usuarios: só se inclúe o ID por seguridade. Permite devolver só un rango engadindo os parámetros `skip` e `count`.

POST Crea un novo usuario.

`/login?user=(nome)`

POST Inicia sesión para o usuario «nome». O método precisa que se pase o contrasinal sen cifrar no corpo da mensaxe, e devolve unha [Testemuña](#)

`/find?name=(nome)`

GET Devolve o ID do usuario co nome indicado

`/(id)`

GET Devolve o usuario con id `id`

POST Actualiza o usuario co contido do corpo da petición

DELETE Elimina o usuario.

`/(id)/role`

GET Devolve o rol do usuario

PUT Cambia o rol do usuario ao contido do corpo da petición

`/(id)/logout`

POST Pecha a sesión do usuario, eliminando a [Testemuña](#) indicado no corpo da petición.
Devolve un código «*HTTP 404 Not Found*» se o usuario non ten rexistrada esa testemuña

Se se intenta cambiar un campo obrigatorio a unha cadea baleira ou nula, ou eliminalo, devólvese unha mensaxe «*HTTP 400 Bad Request*».

6.2.4 `/search`

Baixo esta ruta existen 3 subrutas (`components`, `templates` e `users`). Todas elas funcionan do mesmo xeito: Envíaselle unha petición GET, e devolve unha listaxe dos obxectos axeitados. Coa petición GET pódense incluír os seguintes atributos:

query Cadea que contén a petición a buscar. Véxase a sección [6.4](#)

skip Número de elementos que saltar ao inicio da listaxe

count Número de elementos a devolver

Cada subruta ten, ademáis, unha subruta `/clear` que permite borrar a caché facéndolle un POST, opcionalmente engadindo a petición a borrar.

6.3 Organización do código

6.3.1 Servidor

A definición das API atópase dividida en tres clases: `UsersResource` para `/users`, `TemplateResource` para `/templates` e `ComponentResource` para `/components`. Dentro desas clases defínese un método por ruta. A clase `OpenApiResource` indica baixo que ruta pode accederse á definición OpenAPI.

No paquete raíz atópanse tamén unha serie de clases auxiliares ou de configuración:

- O punto de entrada da aplicación en `Application`
- Unha serie de valores predeterminados na base de datos, en `DataLoader`
- A configuración de seguridade, en `AuthenticationManager`. Véxase o capítulo 7
- A definición dunha cadea de texto serializable por JSON. Usando a clase `String` directamente Jakarta entende que o método devolve JSON xa serializado, e causa problemas ao enviar e recibir cadeas.

O paquete `exceptions` recolle todas as excepcións, separadas en subpaquetes. Tamén inclúe unha excepción xenérica, `RESTException`, que é superclase de todas as excepcións. O serializador `RESTExceptionMapper` encárgase de serializar os `RESTException`, transformando a un `RESTExceptionSerializable` só cos atributos que se queren enviar. doutro xeito, se se enviase directamente o `RESTException` este incluíría tamén atributos da clase `Exception`, que non interesan; e a clase `RESTExceptionSerializable` non se pode lanzar coma excepción ao non ser subclase de `Exception`.

O paquete `dao` recolle as definicións dos datos, e tamén as interfaces que definen as posibles peticións ás bases de datos. Hibernate encárgase de xerar peticións usando os nomes dos métodos definidos nas interfaces.[13]

Por mor dun fallo de OpenAPI[22] non se usan directamente as clases de tempo de Java, polo que no seu lugar úsase unha versión propia `JSONDateTime`, que inclúe métodos `getter`, `setter` e `parse` (ademais do construtor); e encapsula un `OffsetDateTime`, que é a clase que usa OpenAPI para datos de tipo temporal.

6.3.2 Cliente

O cliente organízase de xeito semellante ao servidor. No paquete raíz atópanse o punto de entrada (`Main`), a configuración de `SecurityConfig` (véxase o capítulo 7). Tamén ten paquetes para as excepcións, para a seguridade e un chisco de configuración en `service` e para as vistas. A clase `CSSandJSResources` encárgase de procesar as peticións ás rutas `/css/` e `/js/` e devolver os recursos axeitados.

6.4 Busca

O sistema de busca está composto de dúas fases. Na primeira, convértese a cadea da petición nunha árbore de nodos usando ANTLR. Despois, e de xeito recursivo, compróbase obxecto a obxecto se se cumpren as condicións.

A definición usada por ANTLR para xerar o código axeitado atópase no ficheiro `Find.g4`, na ruta `pleste-server/src/main/antlr4/gal/udc/fic/prperez/pleste/service`. As diferenzas máis importantes entre a gramática Backus–Naur indicada no capítulo 4 e o implementado no ficheiro son engadir a regra «top», que contén só unha `<expresión>` e o carácter EOF por mor dun fallo en ANTLR que non se vai arranxar[23]; e unha definición un chisco distinta de cadeas, para que o analizador léxico non consuma o carácter previo ás comiñas de inicio.

Tras executar a definición sobre unha cadea de entrada (véxase a sección 3.1, no parágrafo referente a ANTLR) xérase unha árbore propia, formada por unha serie de nodos, cada ún dos cales implementa un método `matches(Object)`, que comproba se o obxecto indicado cumpre coa condición indicada (no caso da regra `<proba>`) ou devolve o resultado de chamar, recursivamente, este método nos seus nodos fillos (no caso das regras `<expresión>`).

A estrutura destes nodos é a seguinte:

- Node: Clase abstracta contén un atributo `child` que contén o único fillo deste nodo
 - RootNode: Versión concreta da clase Node, usada só para o nodo raíz da árbore.
 - TestIn: Comproba sobre cadeas
 - TestNumber: Comproba sobre comparacións numéricas
 - TestString: Comproba sobre unha cadea
 - BinaryNode: Esta clase abstracta introduce un segundo nodo fillo, para poder implementar unha árbore binaria.
 - * AndNode: Implementa a operación de conxunción sobre os dous nodos fillos
 - * OrNode: Implementa a operación de disxunción sobre os dous nodos fillos

Deste xeito, a árbore da figura 4.3 quedaría tal como se amosa na figura 6.2:

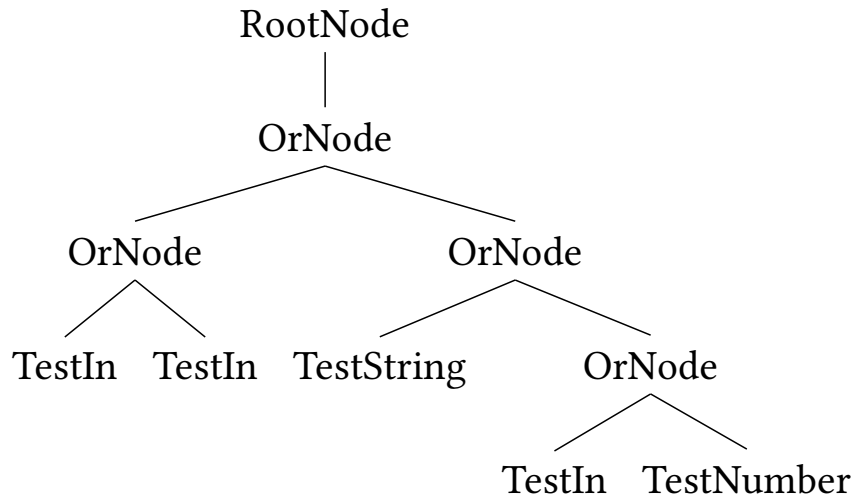


Figura 6.2: Árbore de sintaxe de exemplo

Decidiuse usar un método recursivo, e non iterativo, por que resultaba máis sinxelo implementalo e non se considera que as posibles peticións que se fixeran poidan sobrepasar o límite da pila de chamadas. De todos xeitos, o uso dos operadores lóxicos de Java (&& e |) con cortocircuíto reducen tamén o número de operacións a realizar.

Para implementar a paxinación úsase unha caché LRU, que usando coma índice do mapa a petición devolve unha listaxe de obxectos comprobados para esa petición. Existe unha caché para cada tipo de obxecto.

As propiedades dispoñibles son:

- template.name
- template.description
- template.field.name
- template.field.type
- component.name
- component.description
- component.field.name
- component.field.type
- component.field.value

- user.name
- user.email
- user.role

Comparar un valor dun campo só devolverá os compoñentes con campos do tipo axeitado (TEXT, DATETIME para as cadeas; NUMBER, LINK e DATETIME para os números). Para comparar con propiedades que conteñan enumeracións, deberá usarse o nome orixinal da enumeración, non o que poida amosar a interface web.

Seguridade: Autenticación e autorización

Dentro da aplicación existen dous ámbitos de seguridade: o presente na parte servidor, e o presente na parte do cliente.

7.1 Servidor

Dentro do servidor, xúntanse autenticación e autorización. No servidor úsase Jakarta para xestionar a autorización, dentro da clase propia `AuthenticationManager`, etiquetada axeitadamente para que se poida incluír na cadea de filtros sen precisar de configuración. Esta clase implanta a interface `ContainerRequestFilter`, de xeito que Jakarta inclúa coma filtro cando recibe unha petición.^[24]

É aquí onde se decide se se permite o acceso, e se se continúa cos filtros ate chegar aos controladores, que son os que procesan as peticións REST. O algoritmo aquí usa, para decidir, o rol do usuario, a ruta e o método a acceder. Máis abaixo explicárase con detalle o proceso.

Sen unha identificación válida, só se permite o acceso á ruta `/users/login` por POST, por precisarse para iniciar sesión.

Os usuarios con rol `ADMINISTRATOR` teñen acceso ilimitado ao sistema, mentres que os usuarios co rol `NORMAL_USER` teñen vedada a modificación de todo excepto compoñentes, polo tanto só poden acceder a:

- POST**
 - `/users/{id}/logout`: Esta ruta precísase para pechar a sesión.
 - `/components/*`
- GET**
 - `/users/*`
 - `/templates/*`

Como a xestión de usuarios é dominio exclusivo dos administradores, tampouco se lle permite a un técnico modificar o seu propio usuario. Enténdese que a creación do usuario responde a políticas empresariais, e probablemente en versións futuras dependa de sistemas de usuarios.

O fluxo de inicio de sesión é o amosado no diagrama da figura 7.1¹:

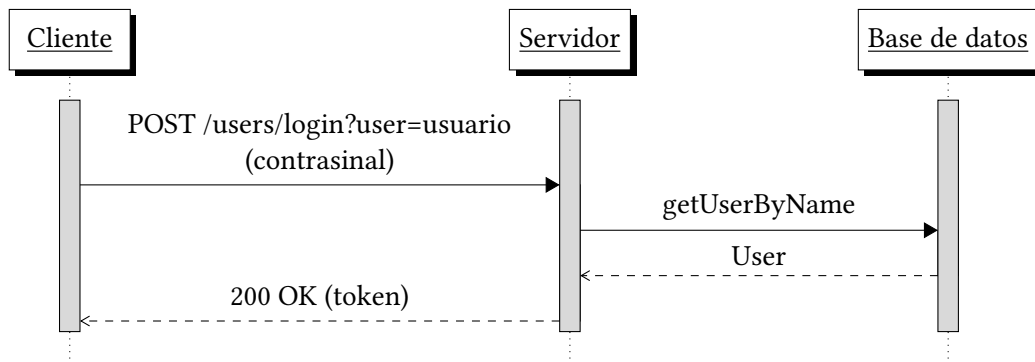


Figura 7.1: Fluxo de inicio de sesión no servidor

Tras recibir o usuario, se o contrasinal (enviado en texto plano e cifrado no servidor) coincide co gardado na base de datos, o servidor xera unha **Testemuña** aleatoria² que devolve ao cliente. Se o contrasinal non coincide, ou non se atopou o usuario indicado, devólvese un código 404 e a mensaxe «*User not found*».

Despois diso todas as peticións enviarán a testemuña nunha cabeceira, chamada X-API-KEY. Esta cabeceira será comprobada, tal como se indicou arriba, polo filtro `AuthenticationManager`, que comprobará a súa validez e os permisos asignados ao usuario que a creou. Isto pode verse no diagrama da figura 7.2

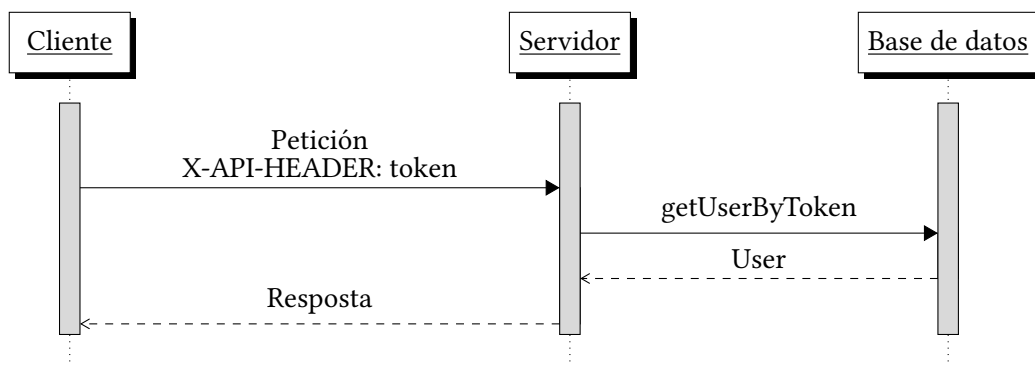


Figura 7.2: Fluxo de acceso á API

¹ O que hai entre parénteses é o corpo da petición

² Xerada a partir da clase `Random` de Java

Se o filtro non atopa a testemuña na base de datos, devolverá unha mensaxe 404 Not Found, mentres que se atopa a testemuña, pero o usuario asociado non ten o rol axeitado, devolverá unha mensaxe 403 Forbidden.³

Por último, só queda explicar a ruta `/users/(id)/logout`. Esta ruta, que precisa dunha testemuña válida para acceder, o único que fai é eliminar a testemuña que se lle pase polo corpo da mensaxe. Se non existe a testemuña baixo este usuario, devolve unha mensaxe 404 Not Found. Polo tanto, o fluxo completo quedaría tal coma se indica no diagrama da figura 7.3

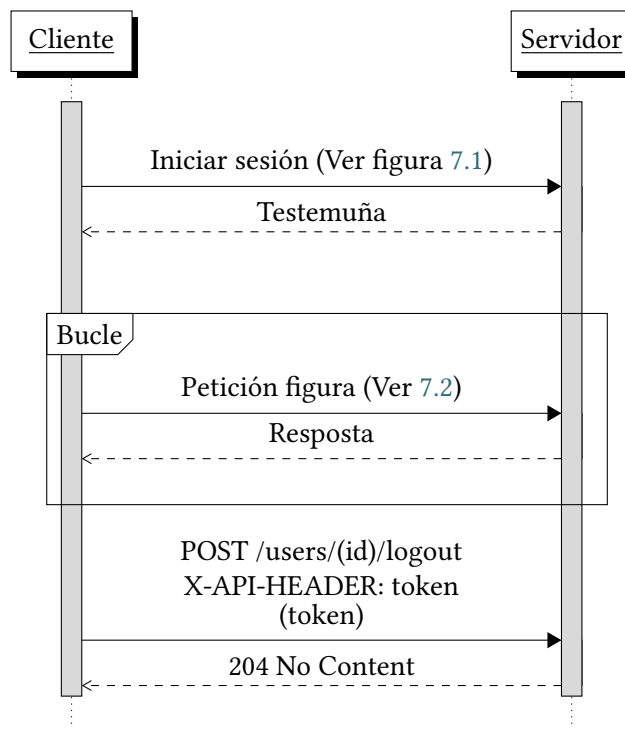


Figura 7.3: Proceso completo de acceso á API

7.2 Cliente

Na parte do cliente ambas, tanto a autenticación como a autorización, están xestiónaas Spring Security. Isto funciona mediante a implementación local de varias clases, que Spring encárgase de integrar no sistema.

³ Tal coma se indica na sección 6.5.3 do RFC 7321[11]

De xeito semellante ao servidor, o sistema baséase nunha serie de filtros[25] que deciden se permiten ou non o acceso. Para un usuario externo, o proceso sería o seguinte:

1. O usuario intenta acceder á ruta /
2. Spring invoca unha instancia da interface `AuthorizationManager`, pasándolle coma obxecto un `AnonymousAuthenticationToken`
3. Se o `AuthorizationManager` denega acceso, Spring redirixe ao usuario a unha ruta indicada para que inicie sesión, neste caso `/login`
4. Tras introducir as credenciais, Spring invoca unha instancia da clase `AuthenticationProvider`, pasándolle unha clase `UsernamePasswordAuthenticationToken`. Esta clase decide se as credenciais son correctas, e nese caso devolve unha autenticación.
5. O usuario rediríxese á páxina de inicio

Tras isto, gárdase a testemuña dentro do almacenamento da sesión, e recupérase para facer peticións. Cada vez que o usuario acceda a unha nova ruta, Spring chamará a `AuthorizationManager` antes de permitilo.

No caso desta aplicación, todo isto configúrase na clase `pleste.client.SecurityConfig`. Nesta clase defínese un método

```
1 @Bean
2 public SecurityFilterChain filterChain(HttpSecurity http,
3     AuthorizationManager<RequestAuthorizationContext>
4     ↪ authorizationManager)
5     throws Exception {
```

que, usando unha DSL formada mediante métodos da clase `HttpSecurity` e usando o patrón construtor, permiten segregar os métodos de autorización segundo a ruta á que se acceda:

```
1 return http.csrf(AbstractHttpConfigurer::disable)
2     .authorizeHttpRequests(
3         authorizationManagerRequestMatcherRegistry ->
4             authorizationManagerRequestMatcherRegistry
5                 .requestMatchers("/login*").anonymous())
```

Todos os métodos de autorizar peticións seguen o mesmo formato: a ruta a comprobar dentro do lambda, seguido do tipo de acceso permitido. Neste caso, o acceso anónimo permite que só os usuarios que inda non iniciaron sesión poidan acceder ao formulario de inicio de sesión.

```

1      .authorizeHttpRequests(
2          authorizationManagerRequestMatcherRegistry ->
3              authorizationManagerRequestMatcherRegistry
4                  .requestMatchers("/favicon*", "/logout*", "/css/*",
5                      "/js/*", "/error*").permitAll())

```

Aquí contrólase certas rutas relacionadas con recursos (*favicon*, CSS e JS) e cousas técnicas (/logout e /error). En todos estes casos permítese acceso indiscriminado, sen importar se se iniciou sesión ou non.

```

1      .authorizeHttpRequests(
2          authorizationManagerRequestMatcherRegistry ->
3              authorizationManagerRequestMatcherRegistry
4                  ↪ .requestMatchers("/**").access(authorizationManager))

```

Esta é a regra por omisión. Para o resto de rutas, precísase iniciar sesión, usando a clase `authorizationManager` (inxeitada por Spring mediante parámetro).

```

1      .formLogin(httpSecurityFormLoginConfigurer ->
2          httpSecurityFormLoginConfigurer
3              .loginPage("/login")
4              .successForwardUrl("/")
5              .failureForwardUrl("/login?error=true"))

```

Nesta parte indícaselle a Spring Security como funciona o formulario de inicio de sesión personalizado. Indícase a ruta a acceder en caso de non iniciar sesión (/login), a onde redirixir en caso de que o inicio de sesión funcione (/) e en caso de que non (/login?error=true).

```

1      .logout(
2          httpSecurityLogoutConfigurer -> httpSecurityLogoutConfigurer
3              .addLogoutHandler(new
4                  ↪ LocalLogoutHandler(defaultApi))).build();

```

Para rematar, tamén configúrase o peche de sesión. Neste caso engádese a clase que Spring chamará en caso de intentar acceder a /logout.

Unha vez configurado todo, queda implementar as clases precisas para que a autenticación e autorización fágase mediante peticións REST ao servidor. Amósase de exemplo como sería o fluxo típico dun usuario accedendo á aplicación.

Para iniciar sesión o fluxo sería o amosado no diagrama da figura 7.4:

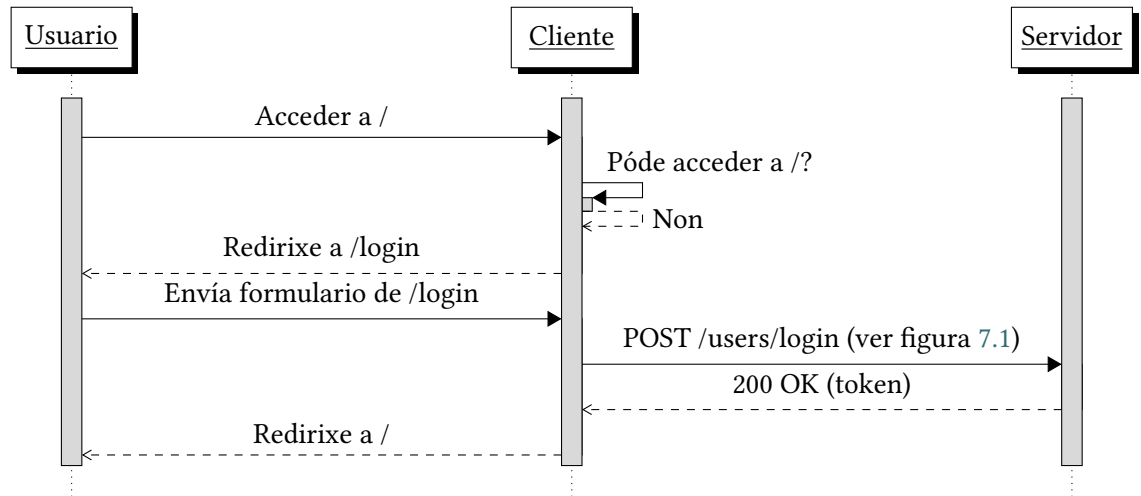


Figura 7.4: Acceso á aplicación por parte dun usuario anónimo

Aquí encárgase de iniciar sesión a clase `LocalAuthenticationProvider`, que implementa a interface `AuthenticationProvider`. Esta clase é a encargada de, tras recibir o usuario e contrasinal dende o formulario de inicio de sesión, comunicarse co servidor e validar os datos do usuario. Tras facer isto, garda no almacenamento da sesión o nome de usuario, id e testemuña para uso futuro. Para gardar o resultado da autenticación creouse unha clase `LocalAuthentication`, que implementa a interface `Authentication` esta clase só é un almacén de datos compatible cos subsistemas do módulo de *Spring Security*.

Unha vez iniciada sesión, calquera ruta na que intente acceder o cliente seguirá o proceso amosado no diagrama da figura 7.5:

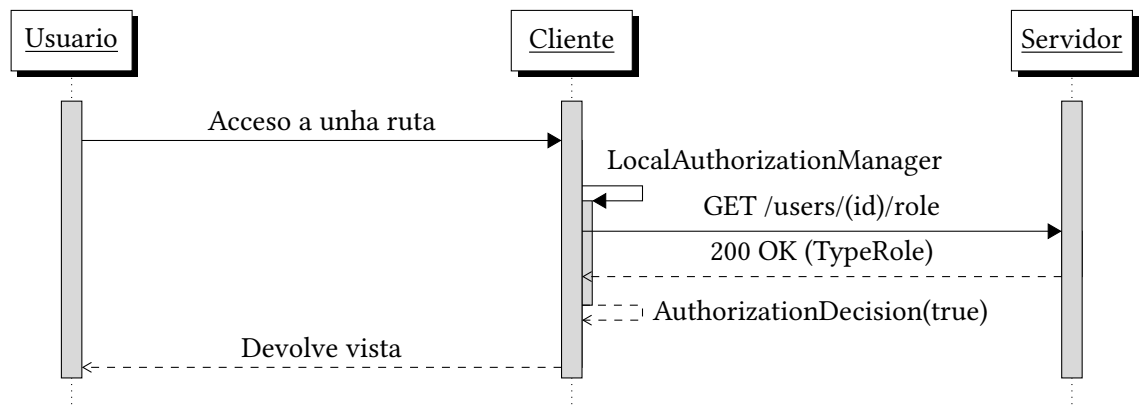


Figura 7.5: Uso típico no cliente

Neste proceso, Spring Security chama á clase `LocalAuthorizationManager`, que implementa a interface `AuthorizationManager`. Esta clase encárgase de buscar o rol do usuario actual, usando para iso o usuario que Spring lle pasa mediante un parámetro `LocalAuthentication`, e comprobando que o dito rol pode acceder á ruta indicada. Unha vez confirmado isto, devolve unha instancia da clase `AuthorizationDecision(true)` afirmando que o usuario pode acceder ao recurso. Feito isto, Spring continúa procesando a petición e devolve a vista axeitada (para máis información, consulte o capítulo 5).

Para rematar, amósase o fluxo de peche de sesión no diagrama da figura 7.6:

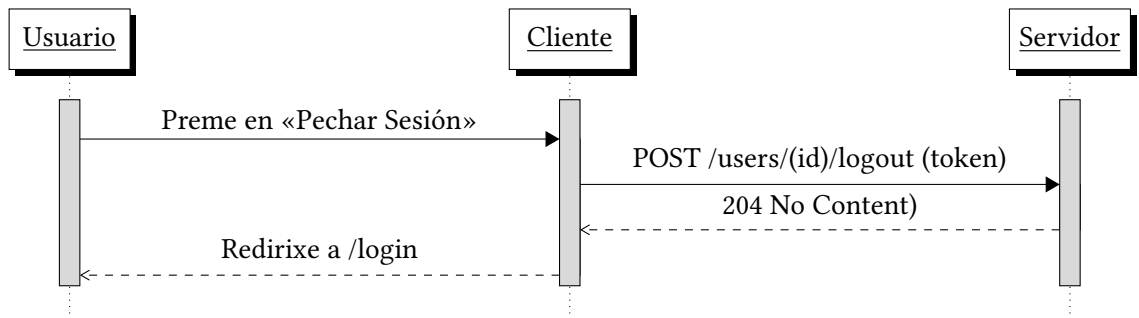


Figura 7.6: Pegar sesión

Neste caso tamén intervén unha clase propia, `LocalLogoutHandler`, que implementa a interface `LogoutHandler`. Esta clase o único que fai é facer unha petición ao servidor para que elimine a testemuña (véxase o final da figura 7.3), e limpa o almacenamento da sesión. Sexa como for, Spring Security da por rematada a sesión e volve tratar o usuario coma anónimo.

Todo isto xunto da, coma fluxo final, o amosado no diagrama da figura 7.7:

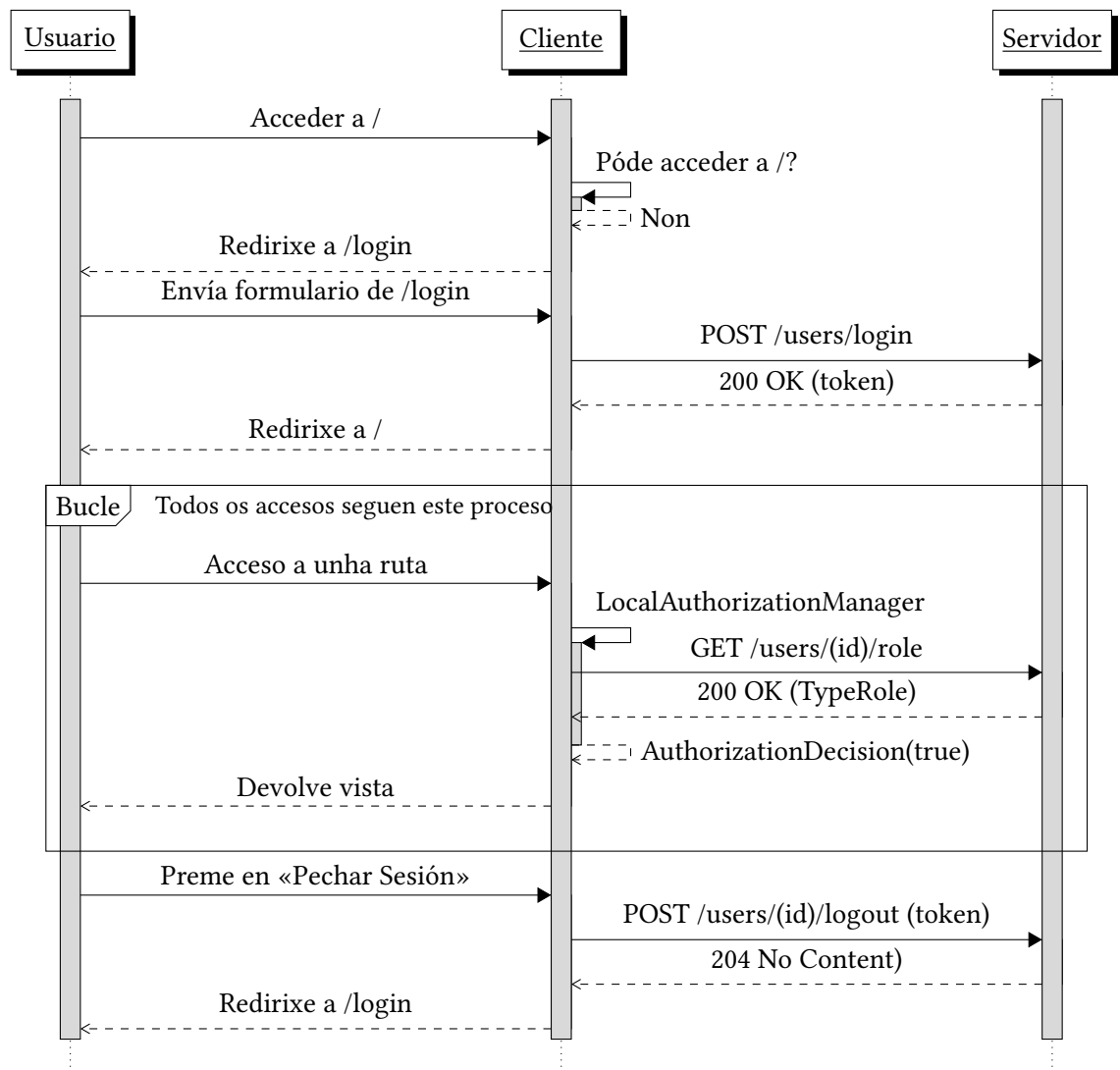


Figura 7.7: Uso típico no cliente

Conclusións

Como conclusión do proxecto, o traballo acadado preséntase cun balance positivo, non só polo programa creado e xa descrito nesta memoria, senón que tamén por todo o proceso de aprendizaxe realizado ante tantas ferramentas e tecnoloxías practicamente novas. Inda que a base bebe moito do aprendido durante a carreira, especialmente da materia de *Integración de Aplicacións*, a experiencia de montar o proxecto dende cero e engadindo novas tecnoloxías resultou didáctica, tanto nos acertos coma nos fallos no deseño da aplicación.

Por desgraza, e a pesar da axuda prestada polos directores do proxecto, esta plataforma inda precisa probarse nun ambiente real, tras o cal seguramente se precisen facer máis melloras para adaptalo a unha carga de traballo real. Entre outras, seguramente se precisará dunha integración cun sistema de identificación e autenticación externo. Porén, por mor da falta de tempo decidiuse reducir o alcance do proxecto ao actual.

Por sorte, o deseño empregado na construción da plataforma permite unha adaptación sinxela a calquera engadido que se lle queira realizar en datas futuras.

8.1 Futuras melloras

Sexa como for, existen varias melloras posibles á aplicación, que por falta de tempo e dificultade, ou outros motivos, non se puideron engadir.

A posibilidade máis útil é a integración con outras ferramentas xa existentes. A API actual presenta unha interface completa dabondo para que outras aplicacións poidan xestionar o contido da base de datos, pero esta integración só é unidireccional. Actualmente non existe ningún xeito de executar accións automáticas, nin realizar interaccións activas con outras aplicacións. Probablemente isto precisará engadir un novo tipo de entidade á aplicación, que usando algunha linguaxe dinámica (Lisp ou Lua, por exemplo) permita definir condicións de activación e accións a executar.

Unha característica que quedou fora da aplicación é un rexistro. Facer un sistema de rexistro útil precisaría de rexistrar todas as actividades que realiza a aplicación, vinculándoas a un usuario e obxecto precisos.

Xunto cunha futura integración cun sistema de autenticación externo, poderíase engadir un sistema de permisos máis detallado e configurable.

Unha característica menor, pero que precisárase para unha futura expansión da aplicación, é a localización. Actualmente a aplicación inserta directamente cadeas de texto dentro das páxinas da aplicación e das mensaxes dos fallos. Centralizar todo isto axudaría á tradución e arranxar erratas.

Apéndices

Material adicional

Pódese acceder ao código desta aplicación dende a ruta https://github.com/Parodper/tfg_plataforma_xestion_tecnica/

Relación de Acrónimos

API Application Programming Interface[26]. 30

CRUD Create, Read, Update, Delete. 9

ITAM Informational Technologies Asset Management. 2

JPA Jakarta Persistence [API](#). 9

MVC Model, View, Controller. 10

REST REpresentational State Transfer. 30

SPA [SPA](#), Single Page Application. 19

Glosario

BCrypt Algoritmo de [Suma de verificación](#) para contrasinais, que inclúe unha sal. 32

Component Compoñente xerado a partir dun [Template](#) (véxase tamén a sección 6.1.2). 14, 30–33

Field Campo dun [Component](#), creado a partir dun [TemplateField](#) (véxase tamén a sección 6.1.4). 31

ITAM Ferramenta de xestión de activos, que axuda ao persoal técnico a controlar os distintos activos dunha empresa. 2

OpenAPI Normativa que define unha sintaxe para describir interfaces REST[8]. 7

Template Modelo a partir do cal sácanse [Component](#) (véxase tamén a sección 6.1.1). 14, 31, 32

TemplateField Campo dun [Template](#), a partir do cal sácanse [Field](#) (véxase tamén a sección 6.1.3). 30, 31

testemuña Unha testemuña (en inglés «*token*»), é un valor que serve como proba ou identificación do seu portador fronte un servizo. 32, 35, 40

Bibliografía

- [1] B. Bowers, “It’s easy, and expensive, to forget about old equipment,” *The New York Times*. [En línea]. Disponible en: <https://www.nytimes.com/2008/03/13/business/smallbusiness/13hunt.html>
- [2] N. Cusson and K. Main, “8 best it asset management software (2024),” *Forbes*. [En línea]. Disponible en: <https://www.forbes.com/advisor/business/software/best-it-asset-management-software/>
- [3] K. Schwaber and J. Sutherland, *The Scrum Guide*, 2020. [En línea]. Disponible en: <https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf>
- [4] The Standish Group, “The Chaos Manifesto,” 2013. [En línea]. Disponible en: https://www.standishgroup.com/sample_research_files/CM2013.pdf
- [5] Consejo General de Colegios Profesionales de Ingeniería Informática, “Estudio nacional sobre la situación laboral de los profesionales del sector de tecnologías de la información,” 2015. [En línea]. Disponible en: https://www.cci.es/images/ccii/documentos/Informe_Situacion_Laboral_TI_CCII.pdf
- [6] C. Otto, “El mito de la falta de informáticos en España: la universidad deja fuera a 20.000 alumnos,” *El Confidencial*, 2021. [En línea]. Disponible en: https://www.elconfidencial.com/tecnologia/2021-05-31/mito-falta-talento-informatica-empleo-tecnologico_3097099/
- [7] Instituto Nacional de Estadística, “Encuesta anual de estructura salarial,” 2021. [En línea]. Disponible en: https://www.ine.es/prensa/ees_2021.pdf
- [8] Swagger, “OpenAPI.” [En línea]. Disponible en: <https://www.openapis.org/>
- [9] M. Morelli. network mode “custom_network” not supported by buildkit. [En línea]. Disponible en: <https://github.com/docker/buildx/issues/175>

- [10] R. T. Fielding, M. Nottingham, and J. Reschke, "HTTP/1.1," RFC 9112, Jun. 2022. [En línea]. Disponible en: <https://www.rfc-editor.org/info/rfc9112>
- [11] R. T. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," RFC 7231, Jun. 2014. [En línea]. Disponible en: <https://www.rfc-editor.org/info/rfc7231>
- [12] R. T. Fielding, "Architectural styles and the design of network-based software architectures," 2000.
- [13] Spring. JPA Query Methods :: Spring Data JPA. [En línea]. Disponible en: <https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html>
- [14] The PostgreSQL Global Development Group, "PostgreSQL." [En línea]. Disponible en: <https://www.postgresql.org/about/>
- [15] T. Parr, "ANTLR." [En línea]. Disponible en: <https://www.antlr.org/>
- [16] S. Viswanadha and S. Sankar, "JavaCC." [En línea]. Disponible en: <https://javacc.github.io/javacc/>
- [17] G. Tomassetti. Listeners and Visitors - Strumenta. [En línea]. Disponible en: <https://tomassetti.me/listeners-and-visitors/>
- [18] JetBrains. Features - IntelliJ IDEA. [En línea]. Disponible en: <https://www.jetbrains.com/idea/features/>
- [19] AENOR, "UNE-EN 28601:1995," ISO, [AENORMás](#), Tech. Rep., 12 1995, elementos de datos y formatos de intercambio. Intercambio de información. Representación de la fecha y de la hora. (ISO 8601, 1ª Edición 1988 y Erratum 1:1991).
- [20] D. Kelley, *Teoría de autómatas y lenguajes formales*, 1st ed. Madrid: Prentice hall, 1995.
- [21] baeldung and J. C. V. Sánchez. Hibernate inheritance mapping. [En línea]. Disponible en: <https://www.baeldung.com/hibernate-inheritance>
- [22] M. Bethibande. oneOf/anyOf validateJsonElement cannot find symbol. [En línea]. Disponible en: <https://github.com/OpenAPITools/openapi-generator/issues/18547>
- [23] S. Harwell. No viable alternative can be incorrectly thrown for start rules without explicit EOF. [En línea]. Disponible en: <https://github.com/antlr/antlr4/issues/118>
- [24] L. Saeed. Intercepting REST requests with Jakarta REST request filters. [En línea]. Disponible en: <https://blog.payara.fish/intercepting-rest-requests-with-jakarta-rest-request-filters>

- [25] Spring. Architecture :: Spring Security. [En línea]. Disponible en: <https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-securityfilterchain>
- [26] C. Malamud, *Analyzing Novell Networks*, ser. VNR computer library. Van Nostrand Reinhold, 1990.