

# Projecte d'Intercanvi de Llibres

Wellington Stephen Baéz Ramírez      May Castells Raga  
Cristina Malena Díaz      Krishna Ocaña Zevallos  
Ainhoa Pérez García      Anna Marín Nuño

Abril 2025

## Índex

<b>0. Enllaç al Repositori</b>	<b>2</b>
<b>1. Autocompletat amb AJAX i l'API de Google Books</b>	<b>2</b>
Com funciona l'autocompletat en el nostre projecte? . . . . .	2
Implementació . . . . .	2
1. Autocompletat per a títols de llibres . . . . .	2
<b>2. Creació d'instàncies</b>	<b>6</b>
Implementació de WishList i HaveList amb Class-Based Views . . . . .	6
Del botó a la vista: Inici del procés . . . . .	6
Processament a la vista (GET): Preparació del formulari . . . . .	7
Renderització del formulari: Presentació a l'usuari . . . . .	8
Processament a la vista (POST): Persistència de dades . . . . .	9
Testing de la Creació d'Instàncies . . . . .	10
Tests per WishList ( <code>add_to_wishlist.feature</code> ) . . . . .	10
Tests per HaveList ( <code>add_to_havelist.feature</code> ) . . . . .	10
<b>3. Actualització de Ressenyes</b>	<b>11</b>
Flux d'Actualització . . . . .	11
1. Model de Ressenyes ( <code>models.py</code> ) . . . . .	12
2. Vista d'Edició ( <code>views.py</code> ) . . . . .	12
3. Template del Formulari ( <code>review_form.html</code> ) . . . . .	13
4. Configuració d'URLs ( <code>urls.py</code> ) . . . . .	14
Testing de l'Actualització de Ressenyes . . . . .	14
Tests d'Actualització ( <code>update_reviews.feature</code> ) . . . . .	14
Implementació dels Steps de Test . . . . .	15
Testing de l'Eliminació de Ressenyes . . . . .	16
Tests d'Eliminació ( <code>delete_reviews.feature</code> ) . . . . .	17
Implementació dels Steps de Test . . . . .	18
<b>5. Model relacional</b>	<b>18</b>
<b>6. Implementacions futures restants</b>	<b>19</b>

## 0. Enllaç al Repositori

<https://github.com/Krisoc123/ProjecteWeb.git>

El codi corresponent a aquesta entrega serà el que es troba a la branca **main** del repositori.

## 1. Autocompletat amb AJAX i l'API de Google Books

En el nostre projecte d'intercanvi de llibres, hem implementat una funcionalitat d'autocompletat que utilitza AJAX (Asynchronous JavaScript and XML) per fer consultes en temps real a l'API de Google Books, proporcionant suggeriments mentre l'usuari escriu en els camps del formulari.

### Com funciona l'autocompletat en el nostre projecte?

Quan un usuari comença a escriure el títol d'un llibre o el nom d'un autor en el nostre formulari de cerca, el sistema realitza una petició en segon pla a l'API de Google Books. L'API processa aquesta petició i retorna una llista de possibles coincidències, que es mostren immediatament sota el camp de text sense necessitat de recarregar la pàgina.

### Implementació

La nostra implementació utilitza jQuery UI Autocomplete i fa crides AJAX a l'API de Google Books. Hem creat dues funcionalitats d'autocompletat separades:

#### 1. Autocompletat per a títols de llibres

Quan l'usuari escriu al camp de títol, el sistema fa una crida a l'API de Google Books cercant llibres que continguin el text introduït en el seu títol. Només s'inicien les cerques quan l'usuari ha escrit almenys 4 caràcters, per evitar resultats massa generals.

```
$(function() {  
    console.log("Document ready, inicialitzant autocompletat");  
  
    if ($("#id_title").length) {  
        console.log("Element #id_title trobat, aplicant autocompletat");  
  
        $("#id_title").autocomplete({  
            source: function(request, response) {  
                console.log("Enviant petició a Google Books:", request.term);  
  
                $.ajax({  
                    url: "https://www.googleapis.com/books/v1/volumes",  
                    dataType: "json",  
                    data: {  
                        q: "intitle:" + request.term,  
                        maxResults: 10  
                    },  
                    success: function(data) {
```

```
        console.log("Resposta rebuda:", data);

        if (!data.items) {
            console.log("No s'han trobat resultats");
            return response([]);
        }

        response($.map(data.items, function(item) {
            const volume = item.volumeInfo;
            return {
                label: volume.title,
                value: volume.title
            });
        }));
    },
    error: function(xhr, status, error) {
        console.error("Error API:", error);
        response([]);
    }
});

},
minLength: 4
});
} else {
    console.log("ERROR: No s'ha trobat l'element #id_title!");
}
});
```

### Explicació detallada del codi:

- `$(function() { ... })`: Aquesta funció s'executa quan el document HTML està completament carregat.
- Garanteix que tots els elements del DOM estan disponibles abans de manipular-los.
- `$("#id_title").autocomplete({ ... })`: Inicialitza el component jQuery UI Autocomplete.
- La propietat `source` és una funció que proporciona les dades per a l'autocompletat.
- `minLength: 4`: Estableix que l'autocompletat només s'activi quan s'hagin escrit almenys 4 caràcters.
- `$.ajax({ ... })`: Realitza una petició asíncrona a l'API de Google Books.
- `url`: Especifica l'endpoint de l'API.
- `dataType: "json"`: Indica que espera rebre les dades en format JSON.
- `data`: Conté els paràmetres de la petició:

- `q`: "intitle:" + `request.term`: Opera amb "intitle:" per buscar el text només als títols dels llibres.
- `maxResults`: 10: Limita el nombre de resultats.

D'aquesta manera, la consulta a l'endpoint de Google Books es realitza amb el següent format:

`https://www.googleapis.com/books/v1/volumes?q=intitle:{{text}}&maxResults=10`

- `success`: `function(data) { ... }`: S'executa quan la petició té èxit.
- Comprova si hi ha resultats amb `if (!data.items)`.
- Transforma les dades rebudes al format que necessita l'autocomplete mitjançant `$.map()`.
- Cada ítem es transforma en un objecte amb `label` (text visible) i `value` (valor seleccionat).
- `error`: `function(xhr, status, error) { ... }`: S'executa si hi ha un error en la petició.
- Registra l'error a la consola i retorna un array buit.

De manera similar, quan l'usuari escriu al camp d'autor, el sistema cerca autors que coincideixin amb el text introduït. En aquest cas, hem implementat una lògica addicional per extreure tots els autors únics dels resultats retornats per l'API.

```
$(function() {  
    $("#id_author").autocomplete({  
        source: function(request, response) {  
            $.ajax({  
                url: "https://www.googleapis.com/books/v1/volumes",  
                dataType: "json",  
                data: {  
                    q: "inauthor:" + request.term,  
                    maxResults: 10  
                },  
                success: function(data) {  
                    if (!data.items) {  
                        return response([]);  
                    }  
  
                    const authors = new Set();  
  
                    $.each(data.items, function(i, item) {  
                        const volume = item.volumeInfo;  
                        if (volume.authors && Array.isArray(volume.authors)) {  
                            volume.authors.forEach(function(author) {  
                                authors.add(author);  
                            });  
                        }  
                    });  
                    response([...authors]);  
                }  
            });  
        }  
    });  
});
```

```

        const uniqueAuthors = Array.from(authors).map(function(author) {
            return {
                label: author,
                value: author
            };
        });

        response(uniqueAuthors);
    },
    error: function(xhr, status, error) {
        console.error("Error API:", error);
        response([]);
    }
});
},
minLength: 3,
delay: 300
});
});

```

- **Inicialització de l'autocompletat:**

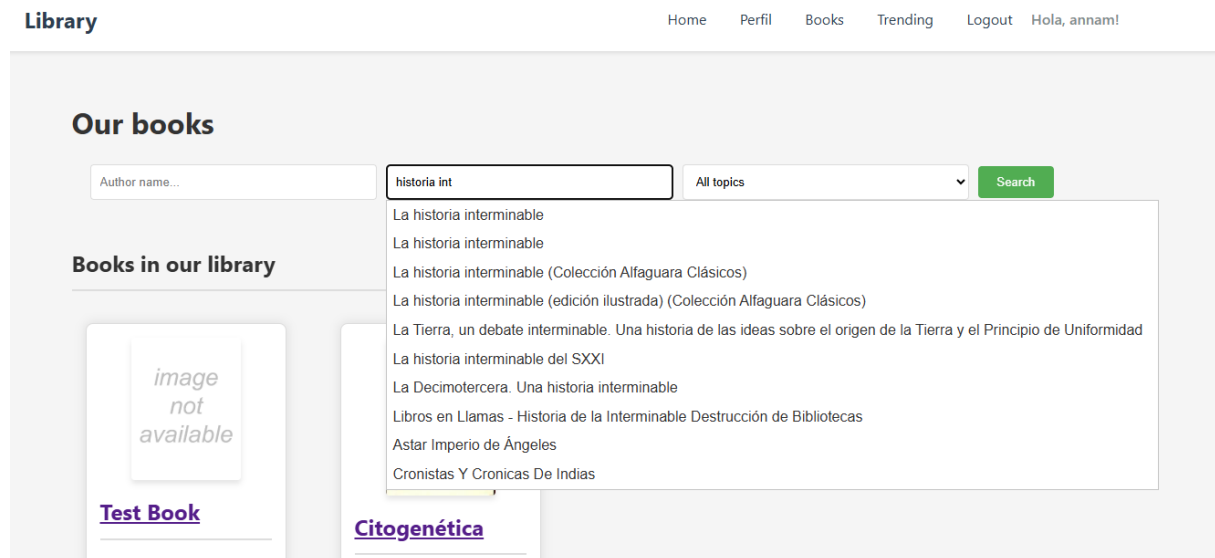
- `$("#id_author").autocomplete({ ... }):` Configura el component d'autocompletat al camp d'autor.
- A diferència del cas anterior, aquest s'activa amb només 3 caràcters (`minLength: 3`).
- Utilitza el mateix endpoint de Google Books, però amb el paràmetre `inauthor:` per cercar específicament autors.
- De manera similar a la cerca de títols, es limita a 10 resultats.
- A diferència de la cerca de títols, un llibre pot tenir múltiples autors.
- `const authors = new Set():` Crea un conjunt (Set) per emmagatzemar autors únics.
- `volume.authors.forEach(function(author) { authors.add(author) }):` Afegeix cada autor al conjunt, automàticament eliminant duplicats.
- `Array.from(authors).map(...):` Converteix el conjunt d'autors en un array i després aplica una transformació.
- Crea un objecte amb `label` i `value` per cada autor, que és el format que espera l'autocompletat.
- `response(uniqueAuthors):` Retorna la llista d'autors únics al component d'autocompletat.
- De manera similar a la cerca de títols, gestiona els errors retornant un array buit.

Aquestes funcionalitats d'autocompletat s'integren en el nostre formulari de cerca de llibres, aquests són els camps HTML corresponents:

```

<input id="id_title" type="text" name="title" placeholder="Book title..." value="
<input id="id_author" type="text" name="author" placeholder="Author name..." valu

```



## 2. Creació d'instàncies

S'han implementat diverses funcionalitats on es creen instàncies a la base de dades, a continuació s'explica detalladament la creació d'instàncies en les relacions **Have** i **Want** (WishList) del model relacional. Aquestes funcionalitats es poden trobar a `books.html` i `book-entry.html`, on es troben els botons pertinents. S'han utilitzat Class-Based Views i ModelForms per a la creació, actualització i eliminació d'instàncies a la base de dades.

### Implementació de WishList i HaveList amb Class-Based Views

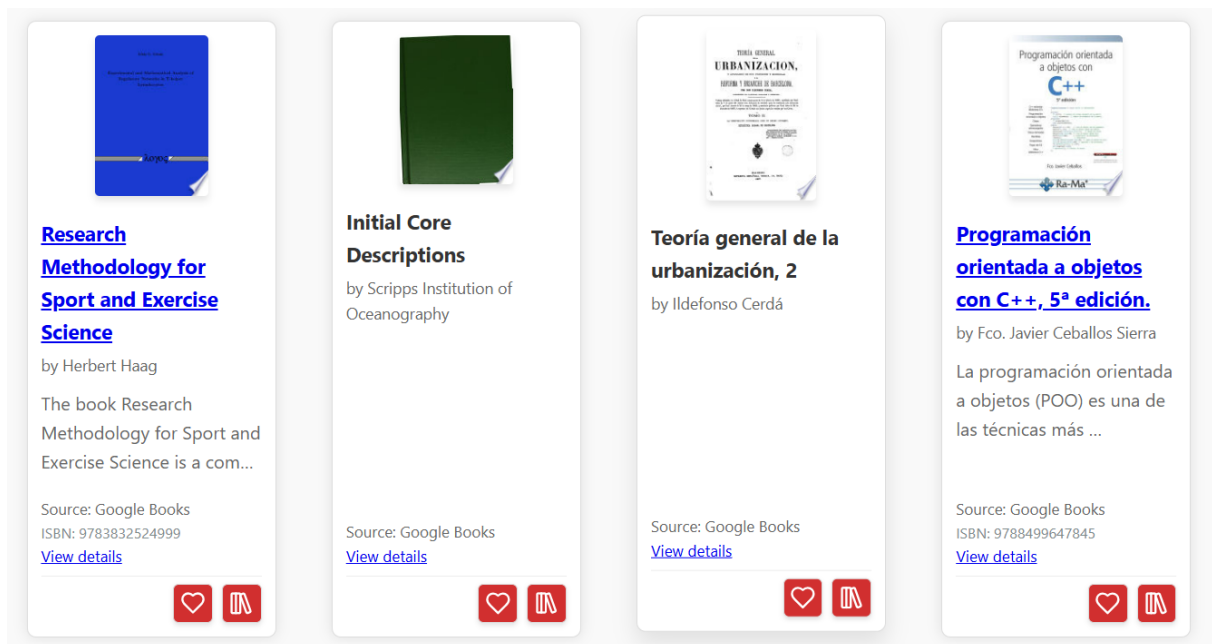
La implementació de les funcionalitats WishList i HaveList al nostre sistema segueix un flux complet des de la interfície d'usuari fins a la persistència a la base de dades. Hem utilitzat les vistes basades en classes de Django tal com es recomanava a l'enunciat, aconseguint un codi més organitzat i reutilitzable.

El flux complet de l'aplicació comença a les targetes de llibres, on l'usuari inicia el procés fent clic en un botó. En aquest punt, el nostre sistema segueix una seqüència de passos que explicarem detalladament.

#### Del botó a la vista: Inici del procés

Tot comença a les targetes de llibres (`book_card.html` i `external_book_card.html`), on implementem botons que permeten als usuaris afegir un llibre a les seves llistes personals. Aquests botons són enllaços HTML que inclouen els paràmetres necessaris:

```
<!-- book_card.html -->
<a href="{% url 'add_to_wishlist' %}?isbn={{ book.ISBN }}&title={{ book.title|urlencode }}">
  <svg>...</svg>
</a>
```



Quan l'usuari clica aquest botó, l'aplicació redirigeix a la URL especificada, afegint com a paràmetres GET les dades del llibre. Al fitxer `urls.py` definim les rutes corresponents que connecten aquestes URLs amb les vistes adequades:

```
# urls.py
path('add-to-wishlist/', views.CreateWantView.as_view(), name='add_to_wishlist'),
path('add-to-havelist/', views.CreateHaveView.as_view(), name='add_to_havelist'),
```

## Processament a la vista (GET): Preparació del formulari

Quan la petició arriba a la vista, s'executa el mètode GET. Hem implementat dues classes: `CreateWantView` i `CreateHaveView`, que hereten de `LoginRequiredMixin` (per assegurar que l'usuari està autenticat) i `CreateView` (per gestionar la creació de nous objectes).

Si l'usuari no està autenticat, el sistema redirigeix automàticament a la pàgina d'inici de sessió (`LoginRequiredMixin` s'encarrega d'això).

```
# views.py
class CreateWantView(LoginRequiredMixin, CreateView):
    model = Want
    form_class = WantForm
    template_name = 'want_form.html'
    success_url = reverse_lazy('books')
```

El mètode `get_initial()` s'encarrega de recollir els paràmetres de la URL i inicialitzar el formulari:

```
def get_initial(self):
    initial = super().get_initial()
    initial['isbn'] = self.request.GET.get('isbn', '')
    initial['title'] = self.request.GET.get('title', '')
    initial['author'] = self.request.GET.get('author', '')
    initial['topic'] = self.request.GET.get('topic', '')
```

```
return initial
```

## Renderització del formulari: Presentació a l'usuari

La vista renderitza el formulari al template corresponent, mostrant la informació del llibre i els camps necessaris per completar l'acció. El template `want_form.html` mostra un formulari amb dades precàrregades:

```
<!-- want_form.html -->
<div class="book-info2">
  <h3>{{ request.GET.title }}</h3>
  <p>by {{ request.GET.author }}</p>
  <p class="isbn">ISBN: {{ request.GET.isbn }}</p>
</div>

<div class="form-group">
  <label for="{{ form.priority.id_for_label }}">Priority:</label>
  {{ form.priority }}
  <small class="form-text">{{ form.priority.help_text }}</small>
</div>

{{ form.isbn }} <!-- Camp ocult -->
{{ form.title }} <!-- Camp ocult -->
{{ form.author }} <!-- Camp ocult -->
```

Home Perfil Books Trending Logout

**Add to Your Wishlist**

**Reguero de ratón**  
by Laura Driscoll  
ISBN: 9781684440962

Priority:

Priority (1-5)

Set the priority of this book (1=lowest, 5=highest)

**Add to Wishlist**

Els camps específics (prioritat per WishList, estat per HaveList) es mostren visiblement, mentre que els camps amb la informació del llibre es mantenen ocults:



```
# forms.py
class WantForm(forms.ModelForm):
    priority = forms.IntegerField(
        min_value=1, max_value=5,
        widget=forms.NumberInput(attrs={'class': 'form-control'}),
        help_text='Set the priority of this book (1=lowest, 5=highest)'
    )
```

### Processament a la vista (POST): Persistència de dades

Quan l'usuari envia el formulari, s'executa el mètode POST que activa `form_valid()`. Aquest mètode realitza les següents accions:

1. Verifica si el llibre existeix a la base de dades, creant-lo si és necessari:

```
try:
    book = Book.objects.get(ISBN=isbn)
except Book.DoesNotExist:
    book = Book(
        ISBN=isbn,
        title=title,
        author=author,
        topic=topic or "General",
        publish_date=timezone.now().date(),
        base_price=10
    )
    book.save()
```

2. Comprova si ja existeix una relació usuari-llibre, actualitzant-la o creant-ne una nova:

```
# Per WishList:
existing_want = Want.objects.filter(user=custom_user, book=book).first()

if existing_want:
    existing_want.priority = form.cleaned_data['priority']
    existing_want.save()
else:
    want = form.save(commit=False)
    want.user = custom_user
    want.book = book
    want.save()
```

3. Redirigeix l'usuari a la pàgina de llibres (`success_url = reverse_lazy('books')`).

Aquest flux complet garanteix poder afegir llibres a les llistes personals amb només uns pocs clics, mentre que el sistema s'encarrega de la validació, la consistència de les dades i la gestió d'errors.

La integració amb fonts externes com Google Books és transparent per a l'usuari, ja

que el sistema crea automàticament els llibres que no existeixen prèviament a la nostra base de dades, realitzant validacions addicionals de l'ISBN i altres camps per mantenir la integritat de les dades.

```
# models.py
class Have(models.Model):
    STATUS_CHOICES = [
        ('new', 'New'),
        ('used', 'Used'),
        ('damaged', 'Damaged')
    ]
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    book = models.ForeignKey(Book, on_delete=models.CASCADE)
    status = models.CharField(max_length=20, choices=STATUS_CHOICES, default='used')
```

Després de finalitzar el procés de creació, l'usuari és redirigit a la pàgina de llibres.

## Testing de la Creació d'Instàncies

Per verificar la correcta implementació de les funcionalitats de creació d'instàncies, s'han desenvolupat tests end-to-end utilitzant el framework Behave. Aquests tests simulen les accions reals d'un usuari interactuant amb la interfície i validen que el sistema respon correctament.

### Tests per WishList (add\_to\_wishlist.feature)

Scenario: Add a book to wishlist when logged in

```
Given I login as user "user1" with password "password"
When I visit the book details page for ISBN "1234567890"
And I add the book to my wishlist
    | priority |
    | 2       |
Then I can see the book with ISBN "1234567890" in my wishlist
And Server responds with page containing "Test Book"
```

Scenario: Redirect to login when trying to add book to wishlist without being logged

```
Given I'm not logged in
When I visit the book details page for ISBN "1234567890"
And I click on "Add to my Wishlist" button
Then I'm redirected to the login form
```

Els tests verifiquen: - Un usuari autenticat pot afegir un llibre a la seva WishList. - Comprovació que la instància es crea correctament a la base de dades. - Verificació que usuaris no autenticats són redirigits a la pàgina de login. - Validació que la prioritat assignada al llibre es registra correctament.

### Tests per HaveList (add\_to\_havelist.feature)

Scenario: Add a book to havelist when logged in

```
Given I login as user "user1" with password "password"
```

```
When I visit the book details page for ISBN "1234567890"
And I add the book to my havelist
  | status |
  | used   |
Then I can see the book with ISBN "1234567890" in my havelist
And Server responds with page containing "Test Book"
```

Scenario: Redirect to login when trying to add book to havelist without being logged in

```
Given I'm not logged in
When I visit the book details page for ISBN "1234567890"
And I click on "Add to my Havelist" button
Then I'm redirected to the login form
```

Els tests comprovats: - Un usuari autenticat pot afegir un llibre a la seva HaveList. - La propietat “status” es guarda correctament a la base de dades. - Es verifica que només usuaris autenticats poden afegir llibres. - Es confirma que la UI mostra correctament els llibres afegits.

La implementació d'aquests tests garanteix que les operacions de creació d'instàncies funcionen correctament, tant a nivell de base de dades com navegació, a la vegada que es respecten les restriccions d'accés i seguretat del sistema.

### 3. Actualització de Ressenyes

S'ha implementat la funcionalitat perquè els usuaris puguin modificar les seves pròpies ressenyes, seguint un flux integrat amb les **Class-Based Views** de Django. Aquesta funcionalitat es troba a la pàgina de detalls del llibre (`book-entry.html`), amb enllaços d'edició per a cada ressenya.

#### Flux d'Actualització

##### 1. Accés des de la UI:

Els usuaris veuen un enllaç “Edit” a les seves ressenyes.

```
<!-- A book-entry.html -->
{% if review.user.auth_user == request.user %}
  <a href="{% url 'review-update' review.pk %}" class="edit-link">Edit</a>
{% endif %}
```

Aquest codi assegura que només l'usuari propietari de la ressenya pot veure l'enllaç d'edició.

**Reviews**

[Write a Review](#)

wellin May 16, 2025 [Edit](#) [Delete](#)

I love this book! c:

## 2. Autorització:

El sistema comprova automàticament que l'usuari és el propietari de la ressenya abans de mostrar el formulari.

## 3. Processament:

- Es mostra un formulari amb el text actual.
- Les modificacions es validen i persisteixen a la base de dades.

## 1. Model de Ressenyes (models.py)

```
class Review(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE) # Relació amb CustomUser
    book = models.ForeignKey(Book, on_delete=models.CASCADE) # Llibre associat
    text = models.TextField() # Contingut de la ressenya
    date = models.DateTimeField(auto_now_add=True) # Data de creació automàtica
```

### • Relacions:

- user: Connecta amb el model personalitzat d'usuari (CustomUser).
- book: Vincula la ressenya amb un llibre específic.

## 2. Vista d'Edició (views.py)

```
class ReviewUpdateView(LoginRequiredMixin, UserPassesTestMixin, UpdateView):
    model = Review
    fields = ['text'] # Camp editable
    template_name = 'review_form.html'

    # Verifica que l'usuari sigui el propietari
    def test_func(self):
        review = self.get_object()
        return self.request.user == review.user.auth_user # Comparació amb l'AuthUser

    # Redirecció després de l'èxit
    def get_success_url(self):
        return reverse_lazy('book-entry', kwargs={'ISBN': self.object.book.ISBN})
```

Aquesta classe exten `UpdateView` i utilitza `LoginRequiredMixin` per assegurar que l'usuari estigui autenticat. La validació de propietat es realitza mitjançant `UserPassesTestMixin`, que comprova si l'usuari autenticat és el mateix que el propietari de la ressenya.

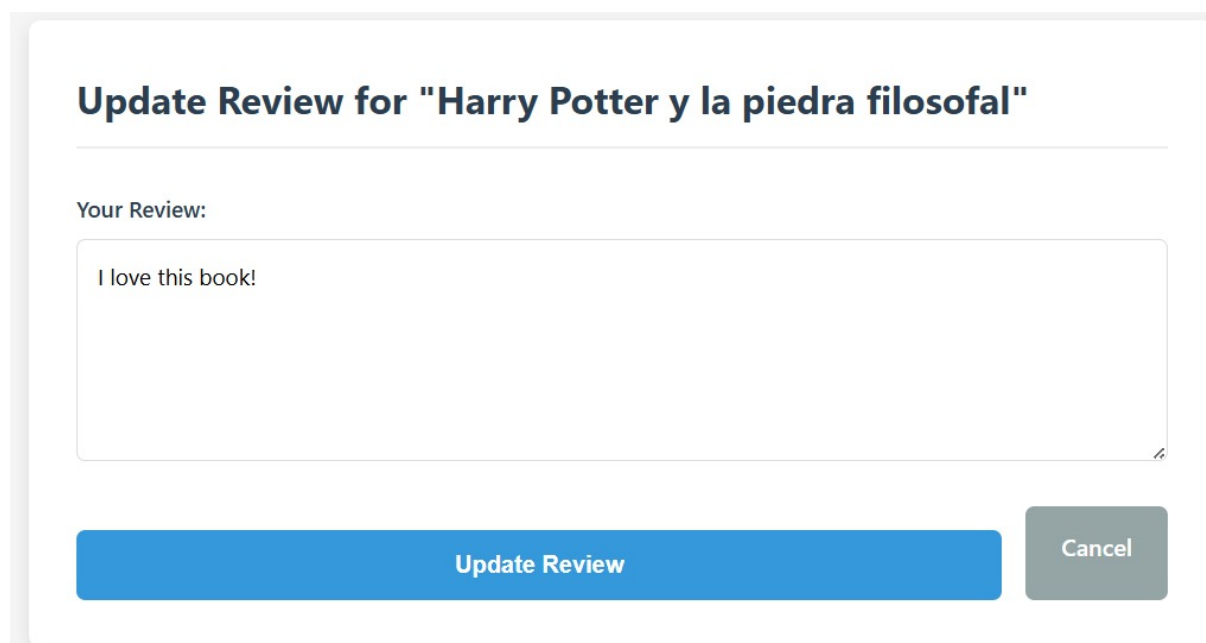
Cal destacar que la validació de l'usuari es realitza comparant l'usuari autenticat amb el propietari de la ressenya a través del model `AuthUser`. Si no es fes aquesta validació, qualsevol usuari podria intentar modificar qualsevol ressenya si aconseguís un enllaç directe.

### 3. Template del Formulari (`review_form.html`)

```
<form method="post">
  {% csrf_token %}

  <!-- Camp de text -->
  <div class="form-group">
    <label for="id_text">Your Review:</label>
    {{ form.text }} <!-- Textarea amb el contingut actual -->
    {{ form.text.errors }} <!-- Errors de validació -->
  </div>

  <!-- Botons d'acció -->
  <button type="submit">Update Review</button>
  <a href="{% url 'book-entry' book.ISBN %}">Cancel</a>
</form>
```



**Update Review for "Harry Potter y la piedra filosofal"**

Your Review:

I love this book!

**Update Review** **Cancel**

*Formulari d'edició simplificat amb validació inclosa*

#### 4. Configuració d'URLs (urls.py)

```
path('review/<int:pk>/update/', ReviewUpdateView.as_view(), name='review-update')
```

- **Paràmetres:**
  - pk: Identificador únic de la ressenya a editar.

#### Integració amb la UI Existents

- **Enllaç d'Edició:** Visible només per al creador de la ressenya.
- **Redirecció Inteligent:** Després de l'actualització, l'usuari retorna a la pàgina del llibre.
- **Validació en Temps Real:** Errors de formulari es mostren dinàmicament (p.e., camps buits).

Aquesta implementació assegura que les ressenyes reflecteixin sempre les opinions actualitzades dels usuaris, mantenint alhora l'integritat i seguretat de les dades.

### Testing de l'Actualització de Ressenyes

Per validar la correcta implementació de la funcionalitat d'actualització de ressenyes, s'han desenvolupat tests exhaustius utilitzant Behave i Selenium. Aquests tests automatitzats simulen les interaccions de l'usuari i comproven tant el funcionament correcte com els controls d'accés.

#### Tests d'Actualització (update\_reviews.feature)

Feature: Update book reviews

In order to correct or improve my reviews

As a user

I want to update reviews I've written

Background: There is a registered user and a review

Given Exists a user "user1" with password "password"

And Exists a user "user2" with password "password"

And Exists a book with ISBN "1234567890"

And Exists a review for book with ISBN "1234567890" by "user1"

text	
------	--

This is a great book!	
-----------------------	--

Scenario: Update my own review

Given I login as user "user1" with password "password"

When I visit the book details page for ISBN "1234567890"

And I edit the review with text "This is a great book!"

text	
------	--

This book is amazing!	
-----------------------	--

Then There are 1 reviews for book with ISBN "1234567890"

And There is a review with text "This book is amazing!" for book with ISBN "1234567890"

And There is no review with text "This is a great book!" for book with ISBN "1234567890"

Scenario: Cannot edit another user's review

Given I login as user "user2" with password "password"

When I visit the book details page for ISBN "1234567890"

Then There is no "Edit" link available

Els tests verifiquen diversos aspectes clau de l'actualització de ressenyes:

- Es comprova que un usuari pot modificar el text de la seva pròpia ressenya.
- Es verifica que el text actualitzat es guarda correctament a la base de dades.
- Es confirma que l'antiga versió de la ressenya no roman al sistema.
- Es valida que els usuaris no poden editar ressenyes d'altres usuaris, verificant que no apareix l'enllaç d'edició.

## Implementació dels Steps de Test

Els tests utilitzen diversos steps personalitzats per simular el comportament de l'usuari:

```
@when('I edit the review with text "{review_text}"')
def step_impl(context, review_text):
    # S'identifica l'element de la ressenya a la pàgina
    review_element = context.browser.find_by_text(review_text).first
    if review_element:
        review_element.scroll_to()

    # S'obté la ressenya de la base de dades i s'actualitza
    auth_user = AuthUser.objects.get(username='user1')
    custom_user = CustomUser.objects.get(auth_user=auth_user)
    book = Book.objects.filter().first()

    review = Review.objects.filter(book=book, user=custom_user, text=review_text).first()

    if review:
        new_text = "This book is amazing!"
        # S'extreu el nou text de la taula de test
        for row in context.table:
            if 'text' in row.headings:
                new_text = row['text']

        review.text = new_text
        review.save()
```

Els tests cubrixen totes les possibilitats d'actualització i totes les restriccions d'accés (o almenys totes les que hem pogut pensar). # 4. Eliminació d'instàncies: El cas de les Reviews

De manera similar a la creació d'instàncies, també hem implementat la forma d'eliminar elements de la base de dades, com és el cas de les reviews. El procés d'eliminació segueix un patró similar però amb algunes particularitats enfocades a garantir que només l'usuari apropiat pot eliminar el contingut i un formulari de confirmació abans de procedir a

l'eliminació.

## Testing de l'Eliminació de Ressenyes

Per assegurar la robustesa de la funcionalitat d'eliminació de ressenyes, s'han desenvolupat tests end-to-end que validen tant la correcta eliminació de les dades com els controls d'accés que protegeixen el sistema.

El flux d'eliminació comença quan un usuari visualitza una ressenya a la pàgina de detalls d'un llibre (`book-entry.html`). Un aspecte fonamental és que **només l'usuari creador d'una ressenya pot eliminar-la**, implementant així un control d'accés restringit. Aquesta restricció s'aplica des de la mateixa interfície d'usuari, on els botons d'eliminació només es mostren al propietari de la ressenya:

```
<!-- book-entry.html -->
{% if user.is_authenticated and user.id == review.user.auth_user.id %}
    <div class="review-actions">
        <a href="{% url 'review-update' review.pk %}" class="review-edit">Edit</a>
        <a href="{% url 'review-delete' review.pk %}" class="review-delete">Delete</a>
    </div>
{% endif %}
```

La condició `user.id == review.user.auth_user.id` garanteix que aquests controls només apareixen per a l'autor original de la ressenya, ocultant-los completament per a la resta d'usuaris. Això és només un primer pas, ja que si algú dedueix la URL d'eliminació, encara podria intentar eliminar la ressenya.

Quan l'usuari fa clic en el botó "Delete", el sistema inicia una seqüència controlada per la classe `ReviewDeleteView`. Aquesta vista, configurada al fitxer `urls.py`, intercepta la petició:

```
# urls.py
path('review/<int:pk>/delete/', ReviewDeleteView.as_view(), name='review-delete'),
```

La vista d'eliminació hereta de tres classes fonamentals per garantir seguretat i funcionalitat adequades: - `LoginRequiredMixin`: Assegura que només usuaris autenticats poden accedir a aquesta vista - `UserPassesTestMixin`: **Implementa la restricció que garanteix que només el creador de la ressenya pot eliminar-la** - `DeleteView`: Proporciona la funcionalitat bàsica per eliminar objectes del model

```
# views.py
class ReviewDeleteView(LoginRequiredMixin, UserPassesTestMixin, DeleteView):
    model = Review
    template_name = 'review_confirm_delete.html'

    def test_func(self):
        review = self.get_object()
        return self.request.user == review.user.auth_user
```

El mètode `test_func()` constitueix el punt central del nostre sistema de seguretat. Aquesta funció és invocada automàticament per Django abans de permetre l'accés a la vista i proporciona una segona capa de protecció (més enllà de la interfície d'usuari)



que verifica que l'usuari actual és efectivament el propietari de la ressenya. Si aquesta comprovació falla —per exemple, si algú intentés manipular les URL directament per eliminar una ressenya aliena— Django bloquejarà completament l'accés, retornant un error 403 Forbidden i impedit qualsevol intent d'eliminació no autoritzada.

Quan l'usuari accedeix a aquesta vista, se li mostra una pantalla de confirmació (`review_confirm_delete.html`) que detalla quina ressenya està a punt d'eliminar i demana confirmació per procedir:

```
<!-- review_confirm_delete.html -->
<div class="delete-confirmation">
  <h1>Delete Review</h1>
  <p>Are you sure you want to delete your review for "{{ book.title }}"?</p>
  <form method="post">
    {% csrf_token %}
    <div class="form-actions">
      <button type="submit" class="button delete-button">Delete Review</button>
      <a href="{% url 'book-entry' book.ISBN %}" class="button cancel-button">Cancel</a>
    </div>
  </form>
</div>
```

Si l'usuari confirma l'eliminació mitjançant el botó “Delete Review”, s'envia una petició POST que la vista processa eliminant la ressenya de la base de dades. El sistema després redirigeix l'usuari a la pàgina de detalls del llibre, utilitzant el mètode `get_success_url()`:

```
def get_success_url(self):
    return reverse_lazy('book-entry', kwargs={'ISBN': self.get_object().book.ISBN})
```

## Tests d'Eliminació (`delete_reviews.feature`)

Feature: Delete book reviews

In order to remove my reviews that are no longer relevant

As a user

I want to delete reviews I've written

Background: There is a registered user and a review

Given Exists a user "user1" with password "password"

And Exists a user "user2" with password "password"

And Exists a book with ISBN "1234567890"

And Exists a review for book with ISBN "1234567890" by "user1"

text	
------	--

This is a great book!	
-----------------------	--

Scenario: Delete my own review

Given I login as user "user1" with password "password"

When I visit the book details page for ISBN "1234567890"

And I delete the review with text "This is a great book!"

Then There are 0 reviews for book with ISBN "1234567890"

And There is no review with text "This is a great book!" for book with ISBN "1234567890"

Scenario: Cannot delete another user's review

Given I login as user "user2" with password "password"

When I visit the book details page for ISBN "1234567890"

Then There is no "Delete" link available

Els tests verifiquen aspectes clau de l'eliminació de ressenyes:

- Es comprova que un usuari pot eliminar completament la seva pròpia ressenya.
- Es verifica que després de l'eliminació, la ressenya ja no existeix a la base de dades.
- Es valida que els usuaris no poden eliminar ressenyes d'altres usuaris.
- Es confirma que els enllaços d'eliminació només són visibles per als propietaris de les ressenyes.

## Implementació dels Steps de Test

Els tests utilitzen steps especialitzats per simular el procés d'eliminació:

```
@when('I delete the review with text "{review_text}"')
def step_impl(context, review_text):
    # S'identifica l'element de la ressenya
    review_element = context.browser.find_by_text(review_text).first
    review_element.scroll_to()

    # Es fa clic al botó d'eliminació
    delete_button = context.browser.links.find_by_text('Delete').first
    delete_button.scroll_to()
    delete_button.click()

    # Es confirma l'eliminació al formulari de confirmació
    confirm_button = context.browser.find_by_css('input[type="submit"]', button[type="submit"])
    confirm_button.scroll_to()
    confirm_button.click()
```

La verificació de l'eliminació correcta es realitza amb aquest step:

```
@then('There are {count:n} reviews for book with ISBN "{isbn}"')
def step_impl(context, count, isbn):
    book = Book.objects.get(ISBN=isbn)
    actual_count = Review.objects.filter(book=book).count()
    assert count == actual_count
```

Aquest es només un resum de tot el que s'ha testejat i implementat.

## 5. Model relacional

Respecte al model relacional, dissenyat en la primera entrega, hem mantingut totes les relacions. Només s'ha afegit a la classe `Have` un nou camp `status` que permet identificar l'estat del llibre (nou, usat o danyat) i a la classe `User` un nou camp `profile_picture`

que permet identificar la imatge de perfil de l'usuari i un camp **description** que permet identificar la descripció de l'usuari.

## 6. Implementacions futures restants

Tot i que, la implementació actual, compleix, en principi, amb els requisits de l'enunciat, hi ha algunes funcionalitats que caldria dissenyar i implementar per tal de donar sentit al projecte. - Caldria establir algun sistema per fixar els preus dels llibres (punts), en funció de l'estat del llibre, el preu base segons alguna API externa o la data de publicació. - També caldria millorar la gestió dels intercanvis, utilitzant la ubicació dels usuaris per tal de facilitar l'intercanvi físic dels llibres. - Afegir algun pas més en l'establiment del intercanvi, com per exemple un xat, o alguna comunicació entre els usuaris per tal de pactar la data i hora de la trobada.