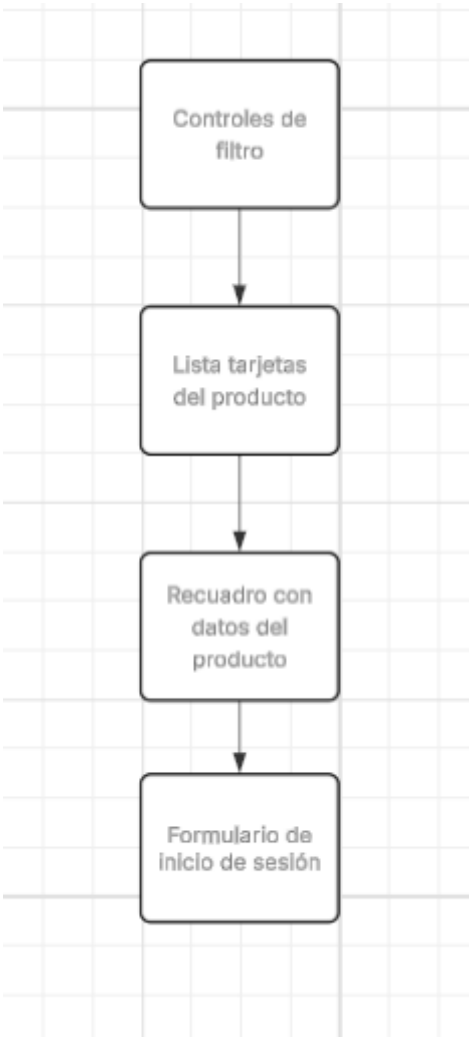
Visión General

Diagrama de arquitectura



Se filtran los datos que introduce el usuario (caja de texto y lista de categorías), luego, se muestran los datos filtrados (tarjetas de producto), dentro de la tarjeta al hacer clic se abre un

recuadro que muestra la información del producto, finalmente para comprar el producto es necesario el formulario de inicio de sesión.

Estructura de archivos

La jerarquía de archivos será de la siguiente manera:

Carpeta raíz/

├— index.html

├— styles.css

├— script.js

├— readme.md

|

├— data/

| └— datos.json

|

└— imagenes/

└— (archivos de imagen aquí)

División de responsabilidades

App.js

├– Coordina módulos

├– Inicializa app

├– Maneja eventos

├– Control de sesión

└ Gestiona estado global

Tareas.js

├– Filtrado

├– Búsqueda

├– Validación login

└ Lógica de negocio (sin DOM)

Dom.js

├– Render tarjetas

├– Render login

├– Mensajes UI

└ Gestión visual de eventos

Storage.js

├– Leer datos JSON

├– Guardar usuario

├– Recuperar sesión

└ Acceso a almacenamiento local

Gestión del estado

Objeto de estado

```
const state = {
  usuario: null,
  productos: [],
  filtrados: [],
  filtros: {
    texto: "",
    categoria: "",
```

```
    autor: "",
    precioMax: null
  },
  ui: {
    cargando: true,
    errorLogin: null,
    vistaActual: "catalogo"
  }
};
```

Funciones para mutar el estado (dentro de App.js)

```
function setUsuario(datosUsuario) {
  state.usuario = datosUsuario;
  Storage.guardarUsuario(datosUsuario);
}

function setProductos(lista) {
  state.productos = lista;
  state.filtrados = lista;
}

function setFiltros(nuevos) {
  state.filtros = { ...state.filtros, ...nuevos };
  actualizarFiltrados();
}

function setVista(vista) {
  state.ui.vistaActual = vista;
  Dom.renderVista(vista, state);
}
```

```
function setErrorLogin(msg) {
   state.ui.errorLogin = msg;
   Dom.mostrarErrorLogin(msg);
}
```

Función de actualización de filtrados (en App.js usando Tareas.js)

```
function actualizarFiltrados() {
   state.filtrados = Tareas.aplicarFiltros(
      state.productos,
      state.filtros
   );
   Dom.renderCatalogo(state.filtrados);
}
```

Ejemplo de flujo de inicialización

```
async function init() {
   const datos = await Storage.cargarDatos();
   setProductos(datos);
   state.ui.cargando = false;
   Dom.renderCatalogo(state.filtrados);
}
```

Flujo de datos

Inicialización

1. **App.init()** se ejecuta al cargarse la página.
2. App llama a:
    - Storage.cargarDatos() → Obtiene catálogo de productos.
    - Storage.cargarUsuario() → Obtiene sesión previa si existe.
3. App actualiza el **estado global**:
    - setProductos()
    - setUsuario()
    - setFiltros() (en vacío)
4. App ordena a **Dom** el primer render:
    - Dom.renderCatalogo(productos)
    - Dom.renderUIUsuario(usuario)
    - Dom.renderVista("catalogo")
    -

Interacciones dentro de la página

A) Filtrado sin recarga

1. El usuario introduce texto o selecciona un filtro en el catálogo.
2. **Dom** captura el evento y llama a App.setFiltros(nuevosFiltros).
3. App actualiza el estado global.
4. App ejecuta Tareas.aplicarFiltros(productos, filtros).
5. App recibe la lista filtrada y ordena actualizar la UI:
    - Dom.renderCatalogo(listaFiltrada)

Inicio de sesión

1. Usuario envía el formulario de login (misma SPA).
2. **Dom** pasa los datos a App.procesarLogin(credenciales).
3. App valida credenciales:
    - Si es correcto:
        - setUsuario(usuario)

        - Storage.guardarUsuario(usuario)
        - Dom.renderUIUsuario(usuario)
        - Dom.renderVista("catalogo") o vista correspondiente
    - Si es incorrecto:
        - setErrorLogin()
        - Dom.mostrarErrorLogin()

Cambio de "secciones"

Ejemplo: usuario pulsa "Mi cuenta", "Catálogo", o "Favoritos".

1. Dom captura el clic y llama: App.setVista("cuenta").
2. App actualiza el estado interno.
3. App ordena a Dom:
    - Dom.renderVista("cuenta")

Actualización del DOM

Solo ocurren cambios parciales en la pantalla:

- Renderizado de las tarjetas de libros.
- Actualización del usuario visible.
- Cambio de la sección visible.
- Mensajes de estado (errores, validaciones, etc.)

Persistencia dentro de la SPA

Storage.js se usa para:

- Guardar usuario logueado.
- Cargar datos del catálogo.
- Guardar configuraciones básicas si existieran.