Desenvolupament Web en Entorn Client

CIFP
Pau Casesnoves

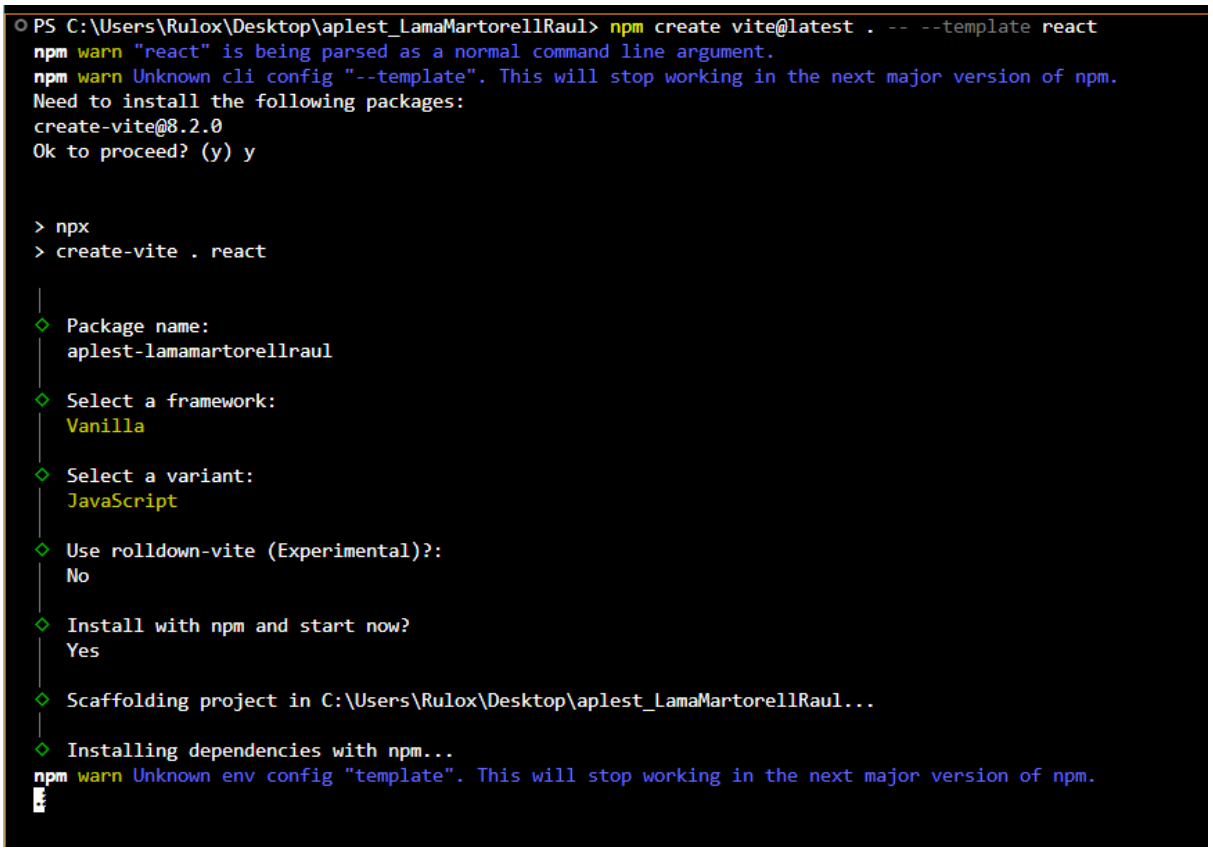
Nom del projecte: React + JavaScript (ES6)

Cicle formatiu de grau superior - Desenvolupament d’interficies web.

CURS: 2n (2025/2026).

ALUMNE: Raül Lama Martorell

Creación del proyecto con React



EJERCICIO 1 – Explicación de arquitectura y tecnologías

Mi aplicación React se ejecuta **en el cliente**, es decir, en el navegador del usuario. El servidor únicamente entrega los archivos estáticos del proyecto (HTML, CSS y JavaScript compilado), pero toda la ejecución del código React, el manejo de eventos, el renderizado de componentes y las actualizaciones del DOM ocurren directamente en el navegador gracias a JavaScript y al motor de React. No hay lógica de React ejecutándose en el servidor.

En la aplicación intervienen varios lenguajes y tecnologías:

- **HTML**: proporciona la estructura base del documento. En React solo existe un archivo HTML principal (index.html), que contiene un ** **. Este elemento es el punto de entrada donde React “inyecta” la aplicación.

- **CSS**: se utiliza para dar estilo a los componentes y a la interfaz. Puede ser CSS global o módulos CSS, pero su función siempre es controlar la presentación.

- **JavaScript (ES6)**: es el lenguaje donde realmente se ejecuta la lógica de la aplicación. Contiene los componentes, los estados, los handlers y los helpers.

- **JSX**: es una extensión de JavaScript que permite escribir estructura similar a HTML dentro del código JS. JSX no se ejecuta directamente: el bundler (Vite o CRA) lo transforma a JavaScript puro. Su papel es describir cómo debería verse la interfaz según los datos.

El flujo desde **index.html** hasta los componentes funciona así:
Cuando la aplicación arranca, el archivo main.jsx/index.js importa React y el componente <App />.

Usando ReactDOM.createRoot(document.getElementById('root')), React localiza el del HTML y monta dentro de él el componente <App />. A partir de ahí, <App /> renderiza otros componentes hijos (<StudyDashboard />, <TaskList />, etc.), y React se encarga de crear y actualizar el DOM real a partir de la estructura descrita en JSX.

Para desarrollar y probar la aplicación utilizo varias herramientas:

- **El navegador** para visualizar el resultado y ver cómo responde la interfaz.

- **La consola del navegador**, donde hago console.log() de datos, resultados de helpers o estados para verificar que la lógica funciona correctamente.

- **El inspector de elementos** para comprobar el DOM generado y ver si los estilos se aplican correctamente.

- **React DevTools**, que permite inspeccionar el árbol de componentes, ver props y estados, y entender cómo fluye la información dentro de la aplicación.

- **El servidor de desarrollo de Vite/CRA**, que me permite ver cambios en tiempo real sin recargar manualmente la página.

Todas estas herramientas me ayudan a comprender qué está pasando dentro de la aplicación, depurar errores rápidamente y asegurar que tanto la lógica como la interfaz funcionan como deberían.

EJERCICIO 2 – Helpers de datos (studyUtils)

1. Objetivo principal del ejercicio:

El objetivo de este ejercicio es **separar la lógica de tratamiento de datos de la presentación**. En lugar de filtrar o calcular datos directamente en los componentes de React, creamos funciones reutilizables (helpers) que encapsulan esta lógica. Esto mejora la **organización**, la **legibilidad** y la **mantenibilidad** de la aplicación.

2. Responsabilidad de cada helper:

   - **getPendingTasks(tasks)**:
     Se encarga únicamente de filtrar las tareas pendientes y ordenarlas por fecha de entrega. No hace nada más: ni modifica el DOM ni calcula porcentajes.

   - **getTasksByModule(tasks, moduleId)**:
     Devuelve solo las tareas asociadas a un módulo concreto. Permite que los componentes de presentación reciban solo la información relevante sin preocuparse de los filtros.

   - **getCompletionPercentage(tasks)**:
     Calcula un valor numérico que resume el estado de todas las tareas. Es un dato listo para mostrar en un panel de resumen o badge, sin mezclar lógica de interfaz.

3. Ventajas de usar helpers:

   - **Reutilización:** cualquier componente puede llamar a estos helpers con cualquier lista de tareas.

   - **Claridad:** cada helper hace una sola cosa, siguiendo el principio de responsabilidad única.

   - **Testabilidad:** es fácil escribir pruebas unitarias para cada helper de manera independiente.

   - **Separación de capas:** la lógica de negocio (filtrado, cálculo de porcentaje) está separada de la lógica de presentación (JSX y renderizado).

4. Consideraciones de calidad en el código:

   - Se usan nombres claros y descriptivos según el enunciado de la práctica (**getPendingTasks, getTasksByModule, getCompletionPercentage**).

- No hay duplicación de código. Por ejemplo, el filtrado por estado o por módulo se hace de manera directa y reutilizable.

- No se usan “valores mágicos” fuera de lo necesario (por ejemplo, **"pendiente"** o **"entregada"** se usan porque son los estados del modelo y no deberían repetirse en otro contexto).

- Los comentarios son cortos, explicativos y ayudan a entender la función sin repetir lo que ya dice el código.

EJERCICIO 3 – Componentes React + handlers que usan helpers

1 - Estructura de archivos .js i css de los componentes que he utilizado

```
src/
├── components/
│   ├── dashboard/
│   │   └── StudyDashboard/
│   │       └── StudyDashboard.jsx
│   ├── layout/
│   │   ├── Header/
│   │   │   ├── Header.jsx
│   │   │   └── Header.css
│   │   ├── Footer/
│   │   │   ├── Footer.jsx
│   │   │   └── Footer.css
│   │   └── MainContent/
│   │       ├── MainContent.jsx
│   │       └── MainContent.css
│   ├── modules/
│   │   ├── ModuleCard/
│   │   │   ├── ModuleCard.jsx
│   │   │   └── ModuleCard.css
│   │   └── ModuleList/
│   │       ├── ModuleList.jsx
│   │       └── ModuleList.css
│   └── tasks/
│       ├── TaskCard/
│       │   ├── TaskCard.jsx
│       │   └── TaskCard.css
│       └── TaskList/
│           ├── TaskList.jsx
│           └── TaskList.css
└── data/
    └── data.js
```

2 - Esquema del árbol de componentes

```jsx
import Header from './components/layout/Header/Header';
import Footer from './components/layout/Footer/Footer';
import MainContent from './components/layout/MainContent/MainContent';
import StudyDashboard from './components/dashboard/StudyDashboard/StudyDashboard';
import './App.css';

function App() {
  return (
    <div className="app">
      <Header />
      <MainContent>
        <StudyDashboard />
      </MainContent>
      <Footer />
    </div>
  );
}

export default App;
```

Si ponemos los componentes así dentro del componente MainContent deberemos de poner algo como children o alguna palabra como se ve en la siguiente imagen y pasalo dentro de la etiqueta main del componente MainContent.

```jsx
import './MainContent.css';

function MainContent({ children }) {
  return (
    <main className="app-main">
      {children}
    </main>
  );
}

export default MainContent;
```

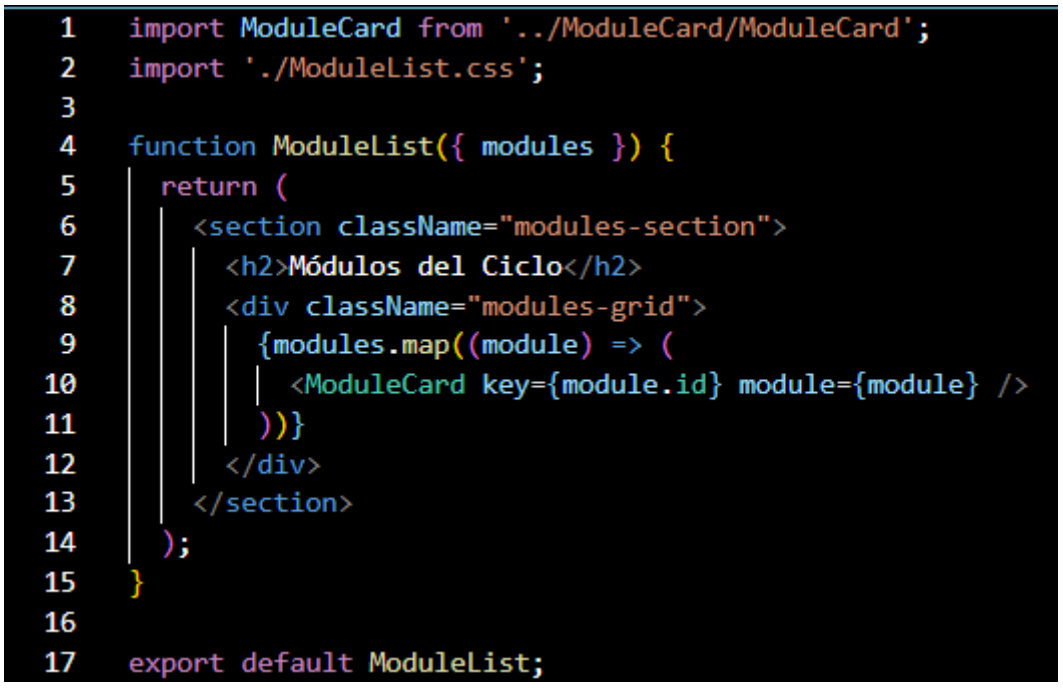
Para que no de errores, sino también se pueden poner los componentes directamente donde pone children dentro de la etiqueta main en lugar de como se ve hecho en la anterior imagen, en la práctica se han realizado las dos formas.

En la siguiente imagen podremos seguir viendo más la estructura del árbol de componentes, donde veremos el archivo StudyDashboard y como se pasan los datos obtenidos de los handlers a los diferentes componentes como ModuleList, TaskList.

Después tanto ModuleList como TaskList dentro tienen otro componente dentro mas especifico por cada módulo o tarea, en el caso de ModuleList el componente de dentro se llamará ModuleCard que detalla mas como mostrar los módulos y lo mismo para TaskList en este caso solo mostraremos un ejemplo.

```jsx
{/* Todos los módulos */}
<ModuleList modules={modules} />

{/* Tareas pendientes */}
<TaskList
  tasks={pendingTasks}
  modules={modules}
  title="Tareas Pendientes (Ordenadas por Fecha)"
/>

{/* Tareas del módulo DWC */}
<TaskList
  tasks={dwcTasks}
  modules={modules}
  title="💻 Tareas de DWC"
/>

{/* Todas las tareas */}
<TaskList tasks={tasks} modules={modules} title="Todas las Tareas" />
```

Archivo de muestra del StudyDashboard.jsx para visualizar la estructura del árbol.



```jsx
import ModuleCard from '../ModuleCard/ModuleCard';
import './ModuleList.css';

function ModuleList({ modules }) {
  return (
    <section className="modules-section">
      <h2>Módulos del Ciclo</h2>
      <div className="modules-grid">
        {modules.map((module) => (
          <ModuleCard key={module.id} module={module} />
        ))}
      </div>
    </section>
  );
}

export default ModuleList;
```

Los datos recibidos que son los módulos en ModuleList son los pasados en la primera imagen de esta página como props, con ellos hacemos un map para que que cada módulo se muestre como se especifique en ModuleCard.

Así se vería el ModuleCard para acabar viendo desde la documentación una idea bien clara de cómo está hecha la estructura.

```jsx
import './ModuleCard.css';

function ModuleCard({ module }) {
  return (
    <div className="module-card">
      <h3>{module.nombre}</h3>
      <p>Curso: {module.curso}</p>
      <p>Horas semanales: {module.horasSemanales}h</p>
    </div>
  );
}

export default ModuleCard;
```

Mostrando el nombre del módulo, el curso y las horas semanales de ese módulo.

2 - Definición de los handlers dentro del componente contenedor <StudyDashboard/>

En StudyDashboard.jsx, importamos los tres helpers desde el archivo studyUtils.js utilizando la sintaxis de destructuring de ES6. Estas funciones son reutilizables y diseñadas para procesar los datos de las tareas sin modificar el array original.

```jsx
// En StudyDashboard.jsx
import {
  getPendingTasks,           // ← Función 1
  getTasksByModule,          // ← Función 2
  getCompletionPercentage    // ← Función 3
} from '../../../utils/studyUtils';
```

Una vez importadas, las llamamos pasándoles el array tasks como parámetro, y en el caso de getTasksByModule, también especificamos el moduleId (3 para DWC). Cada función devuelve datos procesados que luego pasamos como props a los componentes de presentación, manteniendo así una separación clara entre la lógica de negocio (helpers) y la interfaz de usuario (componentes React) como se pide en la práctica.

```jsx
const pendingTask = getPendingTasks();
const dwcTask = getTasksByModule('DWC');
const completionSummary = getCompletionPercentage();
```

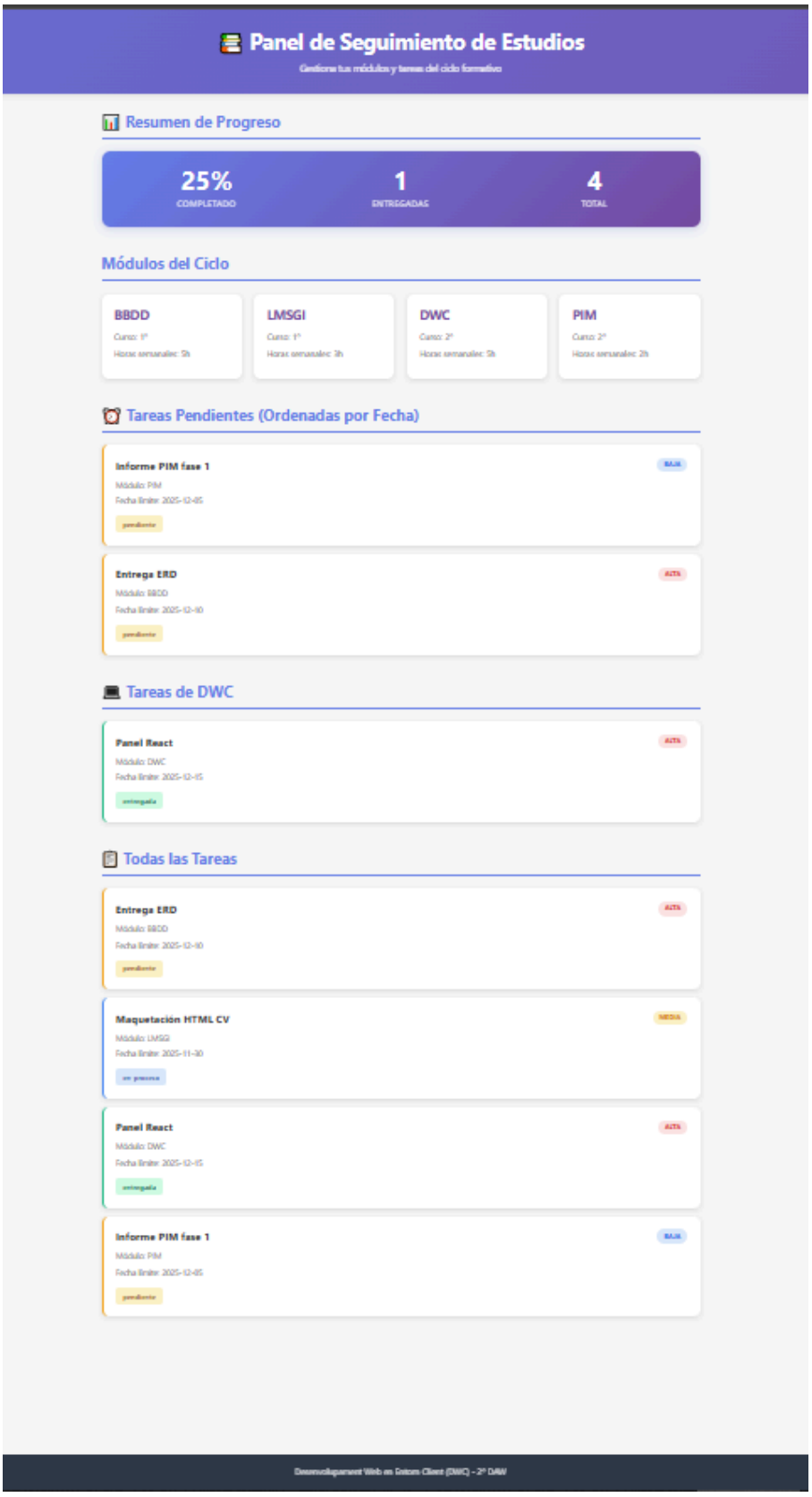
```jsx
import { modules, tasks } from '../../../data/data';
import {
  getPendingTasks,
  getTasksByModule,
  getCompletionPercentage,
} from '../../../utils/studyUtils';
import ModuleList from '../../modules/ModuleList/ModuleList';
import TaskList from '../../tasks/TaskList/TaskList';
import './StudyDashboard.css';

function StudyDashboard() {
  // Handler 1:  lista de tareas pendientes ordenadas por fecha
  const buildPendingTasksHandler = () => {
    const pendingTasks = getPendingTasks(tasks);
    console.log('Tareas pendientes:', pendingTasks);
    return pendingTasks;
  };

  // Handler 2:  lista de tareas de un módulo específico (DWC)
  // Buscar primero por nombre
  const buildModuleTasksHandler = (nomModul) => {
  const module = modules.find(m => m.nombre === nomModul);
  const moduleTasks = module ? getTasksByModule(tasks, module.id) : [];
  console.log(`Tareas del módulo ${module?.nombre || nomModul}:`, moduleTasks);
  return moduleTasks;
};

  // Handler 3: Construir resumen de porcentaje de tareas completadas
  const buildCompletionSummaryHandler = () => {
    const percentage = getCompletionPercentage(tasks);
    console.log('Porcentaje de tareas completadas:', percentage);
    return {
      percentage,
      completed: tasks.filter((t) => t.estado === 'entregada').length,
      total: tasks.length,
    };
  };

  // Ejecutar handlers
  const pendingTasks = buildPendingTasksHandler();
  const dwcTasks = buildModuleTasksHandler('DWC'); // DWC tiene id=3 ( Inicialmente solo se le pasaba id, ahora se busca por nombre i se busca su id con el uso de find())
  const completionSummary = buildCompletionSummaryHandler();
```

```jsx
  return (
    <>
      {/* Resumen de progreso */}
      <section className="completion-summary">
        <h2>Resumen de Progreso</h2>
        <div className="summary-card">
          <div className="summary-stat">
            <span className="stat-value">{completionSummary.percentage}%</span>
            <span className="stat-label">Completado</span>
          </div>
          <div className="summary-stat">
            <span className="stat-value">{completionSummary.completed}</span>
            <span className="stat-label">Entregadas</span>
          </div>
          <div className="summary-stat">
            <span className="stat-value">{completionSummary.total}</span>
            <span className="stat-label">Total</span>
          </div>
        </div>
      </section>

      {/* Todos los módulos */}
      <ModuleList modules={modules} />

      {/* Tareas pendientes */}
      <TaskList
        tasks={pendingTasks}
        modules={modules}
        title="Tareas Pendientes (Ordenadas por Fecha)"
      />

      {/* Tareas del módulo DWC */}
      <TaskList
        tasks={dwcTasks}
        modules={modules}
        title="💻 Tareas de DWC"
      />

      {/* Todas las tareas */}
      <TaskList tasks={tasks} modules={modules} title="Todas las Tareas" />
    </>
  );
}
export default StudyDashboard;
```

Archivo completo StudiyDashboard.jsx

3 - Organización de la interfaz

1. Visualització final de la interfaz



2. Secciones.

Mi mini aplicación incluye cinco secciones, las 3 requeridas y 2 más, todas diferenciadas como se ve en la anterior imagen.

Sección 1: Resumen de Progreso

- Handler: `buildCompletionSummaryHandler()`
- Helper* `getCompletionPercentage()`
- Visualización: Tarjeta destacada con fondo morado mostrando 25% completado (1 de 4 tareas)

- Contenido:
  - 3 estadísticas destacadas: Porcentaje, Completadas, Total
  - Ubicada al inicio para dar visión general del progreso
  - Elementos DOM: `<section class="completion-summary">`, `<div class="summary-card">`, `<div class="summary-stat">`

Sección 2: Lista de Módulos del Ciclo

- Datos: Array `modules` completo (sin filtrar)
- Componente: `<ModuleList modules={modules}>`

- Contenido: Grid con 4 tarjetas de módulos (BBDD, LMSGI, DWC, PIM)
  - Muestra: Nombre, curso y horas semanales de cada módulo
  - Elementos DOM: `<section class="modules-section">`, `<div class="modules-grid">`, `<div class="module-card">`

Sección 3: Lista de Tareas Pendientes

- Handler: `buildPendingTasksHandler()`
- Helper: `getPendingTasks()`
- Componente: `<TaskList tasks={pendingTasks}>

- Contenido: Solo tareas con estado "pendiente", ordenadas por fecha límite
  - Muestra: 2 tareas: "Informe PIM fase 1" (2025-12-05) y "Entrega ERD" (2025-12-10)
  - Verificación: No contiene tareas "entregada" ni "en-proceso"

Sección 4: Lista de Tareas del Módulo DWC

- Handler: `buildModuleTasksHandler('DWC')` → busca módulo por nombre
- Helper: `getTasksByModule(tasks, 3)` → filtra por moduloId = 3
- Componente: `<TaskList tasks={dwcTasks}>`

- Contenido: Solo tareas del módulo DWC (id = 3)
  - Muestra: 1 tarea: "Panel React" (entregada)
  - Verificación: El título indica claramente "Tareas de DWC"

Sección 5: Lista de Todas las Tareas

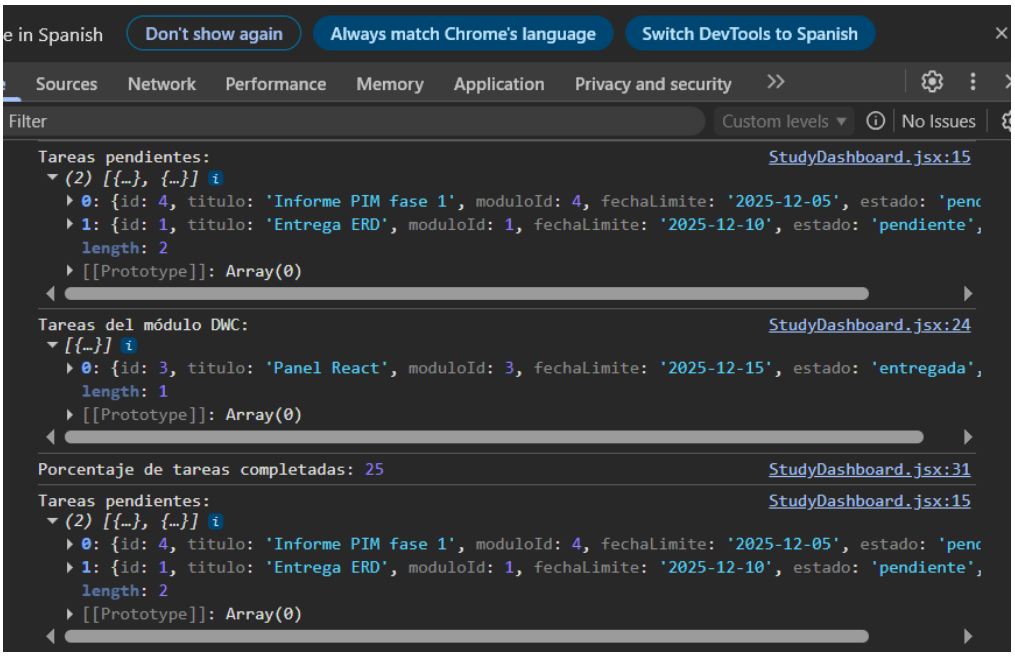
- Datos: Array `tasks` completo (sin filtrar)
- Componente: `<TaskList tasks={tasks} title="Todas las Tareas">`

- Contenido: Vista completa de las 4 tareas en cualquier estado
  - Muestra: Todas las tareas del array

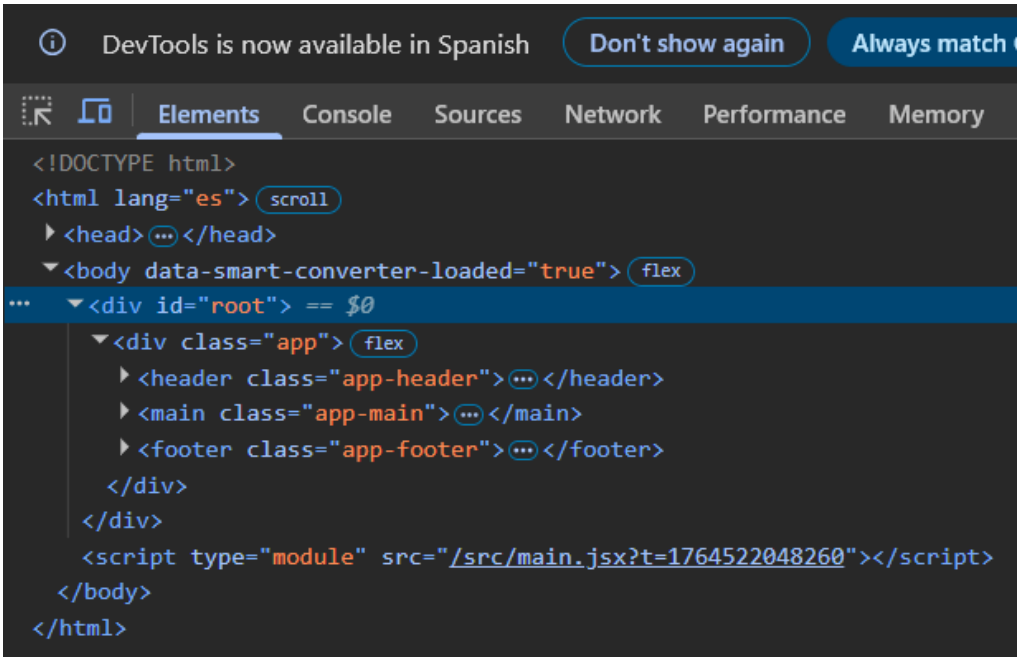
EJERCICIO 4 - Depuración

1. Inspector de elementos

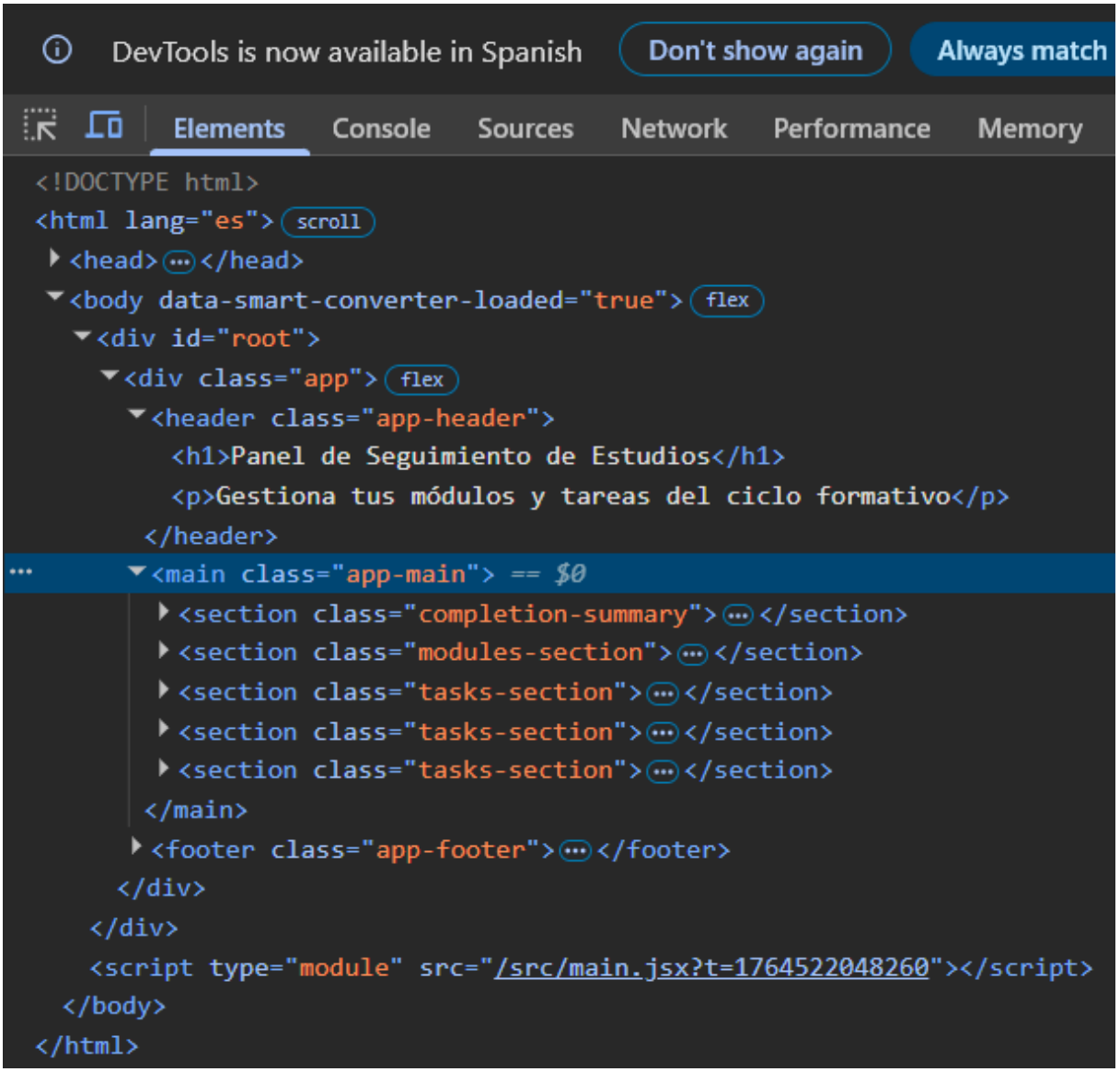
Al abrir la app en el navegador he utilizado la consola del inspector de elementos del navegador pulsando F12 para inspeccionar los datos de los helpers sean los esperados y como se puede ver sale lo esperado, lo que hemos puesto y todo bien sin problemas.



También he usado el inspector de elementos o las React DevTools para ver el árbol de componentes/DOM coincida con lo esperado y hecho en código.



Así podemos ver y observar cómo es la estructura de cualquier página web y esto es lo que acaba renderizando React en este caso en nuestra aplicación.



En la imagen anterior podemos observar las 5 secciones mencionadas anteriormente en la documentación, por lo que vemos que renderiza correctamente las cosas que hemos escrito.

En la siguiente imagen podremos observar como se ve la sección de componentes ordenados por fecha

Conclusion final

Y con estas comprobaciones y explicaciones terminaremos la práctica comentando que es una práctica muy completa y divertida para demostrar el funcionamiento básico de react, organización y estructura de archivos, traspaso y obtención de datos a través de los componentes de haciendo uso de handlers i helpers para mantener una organización clara y separada de datos, lógica y presentación de los componentes ya que hacer uso de ellos me ha ayudado y me ha hecho más fácil entender mejor todo y saber mejor donde ir a buscar cada cosa en un futuro mantenimiento.