# Guía de Desarrollo: Dashboard de Gestión de Proyectos con Astro

Clase Magistral para Juniors

Desarrollador Senior

February 2, 2026

## Contents

# 1 Introducción

Esta guía te llevará a través del desarrollo de un Dashboard de gestión de proyectos usando Astro. No es un tutorial paso a paso, sino una guía conceptual que te ayudará a entender cómo y por qué hacemos las cosas de cierta manera.

Astro es un framework web moderno que se enfoca en el rendimiento. La idea central es simple: renderiza HTML estático por defecto, y solo agrega JavaScript donde sea necesario. Esto resulta en sitios web más rápidos y mejores experiencias de usuario.

# 2 Conceptos Fundamentales

## 2.1 ¿Qué es la Arquitectura de Islas?

Imagina una página web como un océano. La mayoría del contenido es agua (HTML estático). Pero hay islas (componentes interactivos) que flotan en ese océano. Cada isla es independiente y se carga solo cuando es necesario.

En términos técnicos, la Arquitectura de Islas significa:

- La mayoría de tu página se renderiza como HTML puro
- Los componentes interactivos se marcan explícitamente
- Solo esos componentes reciben JavaScript
- Cada componente interactivo es independiente

## 2.2 Componentes Estáticos vs. Interactivos

En Astro, todos los componentes son estáticos por defecto. Esto significa que se renderan como HTML puro sin JavaScript.

Para hacer un componente interactivo, necesitas agregar una directiva `client:*`. Esta directiva le dice a Astro que debe hidratar (convertir a componente interactivo) ese componente específico.

# 3 Estructura del Proyecto

## 3.1 Organización de Archivos

```
client/
  src/
    components/
      DashboardLayout.tsx      # Layout principal
      DashboardStats.tsx       # Estad sticas
      ProjectCard.tsx          # Tarjeta de proyecto
      ProjectFilter.tsx        # Filtro interactivo
```

```
    pages/
      Home.tsx                # P gina principal
      ProjectDetail.tsx       # Detalles del proyecto
      Tasks.tsx               # P gina de tareas
    lib/
      mockData.ts             # Datos de ejemplo
    index.css                 # Estilos globales
```

### 3.2 Propósito de Cada Carpeta

La carpeta `components/` contiene componentes reutilizables. Algunos son estáticos, otros son interactivos. La carpeta `pages/` contiene las páginas principales de la aplicación. La carpeta `lib/` contiene utilidades y datos.

## 4 Componentes Principales

### 4.1 DashboardLayout: Un Client Island

El layout es un componente interactivo porque necesita manejar el estado del sidebar (abierto/cerrado) en dispositivos móviles. Aquí está la estructura:

```
export default function DashboardLayout({
  children,
  activeNav = 'projects'
}) {
  const [sidebarOpen, setSidebarOpen] = React.useState(false);

  return (
    <div className="flex h-screen">
      <aside>
        {/* Sidebar content */}
      </aside>
      <main>
        {children}
      </main>
    </div>
  );
}
```

Este componente usa React hooks (`useState`) para manejar el estado. Por eso es un Client Island.

### 4.2 ProjectCard: Componente Estático

La tarjeta de proyecto solo muestra información. No necesita interactividad más allá de un enlace. Por eso es completamente estática:

```
export default function ProjectCard({ project }) {
  return (
    <div className="bg-card border border-border rounded-lg p-6">
      <h3>{project.name}</h3>
      <p>{project.description}</p>
      <div className="w-full bg-secondary h-2">
        <div style={{width: `${project.progress}%`}} />
      </div>
      <a href={`/projects/${project.id}`}>Ver detalles</a>
    </div>
  );
}
```

No hay estado, no hay event listeners. Solo props y renderizado.

## 4.3 ProjectFilter: Client Island con client:idle

El filtro es interactivo pero no crítico. Se marca con `client:idle` para mejorar el rendimiento de carga inicial:

```
export default function ProjectFilter({ onFilterChange }) {
  const [selectedStatus, setSelectedStatus] = React.useState('all');
  const [searchTerm, setSearchTerm] = React.useState('');

  const handleStatusChange = (status) => {
    setSelectedStatus(status);
    onFilterChange(status, searchTerm);
  };

  return (
    <div>
      <input
        type="text"
        placeholder="Buscar..."
        onChange={(e) => handleSearchChange(e.target.value)}
      />
      {/* Filter buttons */}
    </div>
  );
}
```

# 5 Datos y Estado

## 5.1 Mock Data

Para esta aplicación, usamos datos de ejemplo en `mockData.ts`. En una aplicación real, estos datos vendrían de una API:

```
export const mockProjects = [
  {
    id: '1',
    name: 'Redise o del Portal',
    status: 'in-progress',
    progress: 65,
    team: ['Ana Garc a', 'Carlos L pez'],
    tasks: [...]
  },
  // m s proyectos...
];
```

## 5.2 Gestión de Estado

En la página Home, usamos React hooks para gestionar el estado del filtro:

```
export default function Home() {
  const [filteredProjects, setFilteredProjects] =
    React.useState(mockProjects);

  const handleFilterChange = (status, searchTerm) => {
    let filtered = mockProjects;

    if (status !== 'all') {
      filtered = filtered.filter(p => p.status === status);
    }

    if (searchTerm) {
      filtered = filtered.filter(p =>
        p.name.toLowerCase().includes(searchTerm.toLowerCase())
      );
    }

    setFilteredProjects(filtered);
  };

  return (
    <DashboardLayout>
      <ProjectFilter onFilterChange={handleFilterChange} />
      <div className="grid">
        {filteredProjects.map(project => (
```

```
          <ProjectCard key={project.id} project={project} />
        ))}
      </div>
    </DashboardLayout>
  );
}
```

# 6 Estilos con Tailwind CSS

Usamos Tailwind CSS para los estilos. Los tokens de diseño se definen en `index.css`:

```
:root {
  --primary: oklch(0.55 0.24 264.5);
  --background: oklch(1 0 0);
  --foreground: oklch(0.2 0.01 0);
  /* m s variables... */
}
```

Luego usamos estos tokens en nuestros componentes:

```
<div className="bg-background text-foreground">
  <button className="bg-primary text-primary-foreground">
    Acci n
  </button>
</div>
```

# 7 Mejores Prácticas

## 7.1 Cuándo Usar Componentes Estáticos

Usa componentes estáticos cuando:

- Solo muestran datos (sin estado)
- No tienen event listeners
- Son puramente presentacionales

Ejemplos: tarjetas, estadísticas, listas de solo lectura.

## 7.2 Cuándo Usar Client Islands

Usa Client Islands cuando:

- Necesitan manejar estado (React hooks)
- Tienen event listeners (click, input, etc.)

- Requieren interactividad del usuario

Ejemplos: formularios, filtros, modales, carruseles.

### 7.3 Elegir la Directiva Correcta

- **client:load**: Componentes críticos (header, navegación)
- **client:idle**: Componentes secundarios (filtros, widgets)
- **client:visible**: Componentes debajo del fold (galerías, comentarios)

## 8 Flujo de Desarrollo

### 8.1 Paso 1: Crear un Componente Estático

Comienza siempre con un componente estático. Renderiza datos, nada más:

```
export default function MyComponent({ data }) {
  return <div>{data.name}</div>;
}
```

### 8.2 Paso 2: Agregar Interactividad si es Necesaria

Si necesitas interactividad, conviértelo en un Client Island:

```
export default function MyComponent({ data }) {
  const [expanded, setExpanded] = React.useState(false);

  return (
    <div onClick={() => setExpanded(!expanded)}>
      {data.name}
      {expanded && <p>{data.description}</p>}
    </div>
  );
}
```

### 8.3 Paso 3: Usar en la Página

En tu página, usa el componente. Si es interactivo, agrega la directiva:

```
// Est tico
<MyComponent data={data} />

// Interactivo
<MyComponent client:idle data={data} />
```

# 9 Debugging y Troubleshooting

## 9.1 El Componente no es Interactivo

Si tu componente no responde a clicks o cambios, probablemente olvidaste la directiva `client:*`. Verifica que esté presente en la página.

## 9.2 Demasiado JavaScript

Si tu página carga mucho JavaScript, revisa qué componentes están marcados con `client:load`. Intenta cambiar a `client:idle` o `client:visible`.

## 9.3 Errores de Hidratación

Los errores de hidratación ocurren cuando el HTML renderizado en el servidor no coincide con el que React espera. Asegúrate de que tus componentes renderan el mismo HTML en ambos lados.

# 10 Próximos Pasos

## 10.1 Mejorar la Aplicación

1. Conectar a una API real en lugar de usar mock data
2. Agregar autenticación de usuario
3. Implementar persistencia de datos
4. Agregar más páginas y funcionalidades

## 10.2 Aprender Más

- Lee la documentación oficial de Astro
- Explora ejemplos en GitHub
- Únete a la comunidad de Astro
- Experimenta con otros frameworks (Vue, Svelte)

# 11 Conclusión

Astro es un framework poderoso que te permite construir sitios web rápidos y escalables. La clave es entender la Arquitectura de Islas: renderiza HTML estático por defecto, y solo agrega JavaScript donde sea necesario.

Con esta guía, deberías ser capaz de:

- Entender cómo funciona Astro
- Crear componentes estáticos e interactivos
- Elegir la directiva correcta para cada situación
- Construir aplicaciones web modernas y performantes

Recuerda: el mejor código es el que no existe. Si no necesitas JavaScript, no lo incluyas. Astro te ayuda a lograr exactamente eso.